Log-Time K-Means for 1D Data:
Novel Approaches with Proof and Implementation

(1D 데이터에 대한 로그 시간 K-평균 클러스터링:
새로운 접근법의 증명과 구현)

지도교수: 허 충 길

이 논문을 공학학사 학위 논문으로 제출함.

2024년 12월 19일

서울대학교 공과대학
컴 퓨 터 공 학 부
현 재 익

2025년 2월

Log-Time K-Means for 1D Data:

Novel Approaches with Proof and Implementation

(1D 데이터에 대한 로그 시간 K-평균 클러스터링:

새로운 접근법의 증명과 구현)

지도교수: 허 충 길

이 논문을 공학학사 학위 논문으로 제출함.

2024년 12월 19일

서울대학교 공과대학

컴 퓨 터 공 학 부

현 재 익

2025년 2월

Abstract

Log-Time K-Means for 1D Data:

Novel Approaches with Proof and Implementation

Jake Hyun
Computer Science and Engineering
College of Engineering
Seoul National University

Clustering is a key task in machine learning, with k-means being widely used for its simplicity and effectiveness. While 1D clustering is common, existing methods often fail to exploit the structure of 1D data, leading to inefficiencies. This thesis introduces optimized algorithms for k-means++ initialization and Lloyd's algorithm, leveraging sorted data, prefix sums, and binary search for improved computational performance.

The main contributions are: (1) an optimized k-cluster algorithm achieving $O(l \cdot k^2 \cdot \log n)$ complexity for greedy k-means++ initialization and $O(i \cdot k \cdot \log n)$ for Lloyd's algorithm, where l is the number of greedy k-means++ local trials, and i is the number of Lloyd's algorithm iterations, and (2) a binary search-based two-cluster algorithm, achieving $O(\log n)$ runtime with deterministic convergence to a Lloyd's algorithm local minimum.

Benchmarks demonstrate over 4500x speedup compared to `scikit-learn` for large datasets while maintaining clustering quality measured by within-cluster sum of squares (WCSS). Additionally, the algorithms achieve a 300x speedup in an LLM quantization task, highlighting their utility in emerging applications. This thesis bridges theory and practice for 1D k-means clustering, delivering efficient and sound algorithms implemented in a JIT-optimized open-source Python library.

Keywords: k-means clustering, Lloyd's algorithm, k-means++ initialization, one-dimensional clustering, binary search, prefix sums

Contents

1. Introduction

Clustering is a fundamental task in data analysis and machine learning, with applications in diverse fields such as image segmentation, natural language processing, financial modeling, and bioinformatics [1]. Among clustering methods, k-means [2] is one of the most widely used algorithms due to its conceptual simplicity and computational efficiency. However, finding the optimal solution to the k-means problem is NP-hard in general for d-dimensional data [3], prompting practical implementations to rely on heuristic approaches such as Lloyd's algorithm [4, 5].

One-dimensional (1D) clustering problems arise frequently in a wide range of real-world scenarios, including social network analysis, bioinformatics, and the retail market [6, 7, 8]. For this special case, there have been significant advances in achieving globally optimal solutions efficiently. Wang and Song [9] introduced a $O(k \cdot n^2)$ dynamic programming algorithm for the 1D k-means problem, and Grønlund et al. [10] later improved this to $O(n)$, demonstrating that optimal clustering can be computed in linear time for one dimension.

While globally optimal algorithms for the 1D k-means problem exist, they are not always suitable for scenarios where speed and scalability are paramount. In many real-world applications—particularly those involving large datasets or latency-critical tasks—achieving a near-optimal solution quickly can be more valuable than computing the exact global minimum. Practical libraries, such as `scikit-learn`'s k-means [11], do not exploit the structure of 1D data and instead treat it as a general case, leaving room for further optimization. Under such conditions, improving Lloyd's algorithm for 1D data provides a route to faster performance.

This thesis presents a novel set of algorithms that optimize Lloyd's algorithm for the 1D setting. By carefully exploiting the properties of sorted data, these methods achieve logarithmic runtime, dramatically reducing computational costs while maintaining high-quality clustering outcomes. The contributions include the following:

1. **An optimized k-means++ initialization and Lloyd's algorithm** for approximating

general k-cluster problems in one dimension. By carefully leveraging the properties of sorted data, the proposed approach replaces the linear dependence on the dataset size n with logarithmic factors, resulting in substantial runtime improvements. Specifically, the greedy k-means++ initialization achieves a time complexity of $O(l \cdot k^2 \cdot \log n)$, where l is the number of local trials, followed by Lloyd's algorithm iterations with $O(i \cdot k \cdot \log n)$, where i is the number of iterations. Additional preprocessing, such as sorting and prefix sum calculations, contributes $O(n \log n)$ and $O(n)$, respectively, when required.

   This method improves upon standard k-means implementations, where greedy k-means++ initialization requires $O(l \cdot k \cdot n)$ time, and Lloyd's algorithm iterations require $O(i \cdot k \cdot n)$. By reducing the dependence on n, the dataset size, the proposed optimizations achieve significant speedups, as experimentally demonstrated in Section 4.4.

2. **A binary search-based algorithm for the two-cluster case**, which achieves $O(\log n)$ runtime and deterministically converges to a Lloyd's algorithm solution, skipping iterative refinements entirely. Additional preprocessing costs include $O(n)$ for prefix sums and $O(n \log n)$ for sorting, if not already provided. While the global minimum is not guaranteed, this method is highly desirable for scenarios requiring very fast and deterministic clustering.

Along with the thesis, a complete library implementation in Python 3 is provided, optimized with Numba just-in-time (JIT) compilation [12] to enable efficient integration into various applications.

Benchmarks against the highly optimized and widely used `scikit-learn` k-means implementation [11] highlight the efficiency of these algorithms, as detailed in Section 4.4. The results demonstrate the following:

- **Orders-of-magnitude speedups**, even when including preprocessing steps such as sorting and prefix sum calculations.
- **Equivalent or comparable clustering results** in terms of within-cluster sum of squares (WCSS), the objective function of K-means.

The proposed algorithms also find utility in emerging and highly relevant applications such as **quantization for large language models (LLMs)**, where efficient quantization can be achieved by running 1D k-means clustering on model weights [13]. In particular, cutting-edge quantization methods like Any-Precision LLM [14] rely on repeated executions of the two-cluster approach to hierarchically subdivide clusters of weights. The novel algorithm presented here is exceptionally well-suited for such scenarios, providing over a **300-fold speedup** compared to `scikit-learn`, as demonstrated in Section 4.5. This practical importance is underscored by the direct use of the proposed library implementation, `flash1dkmeans`, within the official Any-Precision LLM implementation[1].

Overall, this thesis contributes both theoretical advancements and practical tools for one-dimensional k-means clustering. By providing a rigorous theoretical foundation and an optimized implementation, the proposed methods demonstrate the feasibility and efficiency of adapting k-means and Lloyd's algorithm to the one-dimensional setting. These contributions establish not only a proof of concept but also a practical solution ready for deployment in diverse computational tasks.

[1]`https://github.com/SNU-ARC/any-precision-llm`

2. Background

This chapter provides the necessary background on the k-means clustering problem, Lloyd's algorithm, and the k-means++ initialization, along with an overview of relevant works. The focus is on the theoretical foundations, time complexities, and practical implementations relevant to this thesis.

2.1 k-Means Clustering

The k-means clustering problem is a widely studied unsupervised learning problem. Given a set of n data points $X = \{x_1, x_2, \ldots, x_n\}$ in a metric space and a positive integer k, the goal of k-means clustering is to partition the data into k disjoint clusters $C_1, C_2, \ldots, C_k$ such that the within-cluster sum of squared distances (WCSS) is minimized. Formally, the objective function is:

$$\text{WCSS} = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2,$$

where μ_i is the centroid of cluster C_i, defined as the mean of all points in C_i.

Finding the globally optimal solution to the k-means problem is NP-hard in general for d-dimensional data, even for $k = 2$ [3]. As a result, heuristic algorithms such as Lloyd's algorithm are commonly used in practice to approximate solutions efficiently.

2.2 Lloyd's Algorithm

Lloyd's algorithm [4, 5] is a popular iterative method for solving the k-means problem. It alternates between assigning data points to their nearest cluster centroid and updating the centroids based on the current cluster assignments. The steps of the algorithm are as follows:

1. **Initialization:** Choose k initial centroids $\mu_1, \mu_2, \ldots, \mu_k$.

2. **Assignment step:** Assign each point $x_j \in X$ to the cluster C_i with the closest centroid:

$$C_i = \{x_j \mid \|x_j - \mu_i\| \leq \|x_j - \mu_m\|, \forall m \neq i\}.$$

3. **Update step:** Update each centroid μ_i as the mean of all points assigned to C_i:

$$\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x.$$

4. Repeat the assignment and update steps until convergence, typically when the centroids no longer change significantly or a maximum number of iterations is reached.

Remark: Both the assignment and update steps of Lloyd's algorithm ensure that the within-cluster sum of squares (WCSS) decreases monotonically after each iteration. As a result, the algorithm is guaranteed to converge to a local minimum of the WCSS objective function.

Time Complexity: The time complexity of Lloyd's algorithm for general d-dimensional data is $O(i \cdot k \cdot n \cdot d)$, where i is the number of iterations. For 1D data, the complexity simplifies to $O(i \cdot k \cdot n)$.

2.3 k-Means++ Initialization

The quality of solutions obtained by Lloyd's algorithm depends heavily on the choice of initial centroids. The k-means++ initialization algorithm [15] improves the centroid selection process by probabilistically choosing points based on their distances to already selected centroids. The steps of the algorithm are as follows:

1. **Initialization:** Choose the first centroid μ_1 uniformly at random from the data points.

2. **Candidate selection:** For each subsequent centroid μ_i, select a point x_j with probability proportional to its squared distance from the nearest already chosen centroid:

$$P(x_j) = \frac{\text{Distance}(x_j, C_{\text{nearest}})^2}{\sum_{x_i \in X} \text{Distance}(x_i, C_{\text{nearest}})^2}.$$

3. Repeat the candidate selection step until k centroids are chosen.

Remark: The k-means++ initialization improves the spread of centroids compared to random initialization, significantly reducing the likelihood of poor clustering outcomes. However, this improvement comes at the cost of additional computation during the initialization phase.

Time Complexity: The time complexity of standard k-means++ initialization for d-dimensional data is $O(k \cdot n \cdot d)$, where k is the number of clusters, n is the number of data points, and d is the dimensionality of the data.

Greedy k-Means++ Initialization: In the greedy version of k-means++ initialization, briefly mentioned in the conclusion of the original paper [15], l candidate centroids are evaluated at each step, and the one minimizing the WCSS is selected. The total time complexity for the greedy version is $O(l \cdot k \cdot n \cdot d)$, where l is the number of local trials. A common choice for l is $O(\log k)$ [16, 17], and `scikit-learn` adopts a similar approach with $l = 2 + \log k$ [11]. For 1D data, where $d = 1$, the complexity simplifies to $O(l \cdot k \cdot n)$.

2.4 Weighted k-Means

Weighted k-means generalizes the standard k-means problem by assigning each data point x_j a weight w_j. The objective function becomes:

$$\text{WCSS} = \sum_{i=1}^{k} \sum_{x_j \in C_i} w_j \|x_j - \mu_i\|^2,$$

where the centroid μ_i is updated as the weighted mean:

$$\mu_i = \frac{\sum_{x_j \in C_i} w_j \cdot x_j}{\sum_{x_j \in C_i} w_j}.$$

Changes to Lloyd's Algorithm: The **update step** computes weighted centroids instead of simple means. The assignment step remains unchanged.

Changes to k-Means++ Initialization: The probability of selecting a point x_j as a centroid becomes proportional to its weighted squared distance from the nearest already chosen centroid:

$$P(x_j) = \frac{w_j \cdot \text{Distance}(x_j, C_{\text{nearest}})^2}{\sum_{x_i \in X} w_i \cdot \text{Distance}(x_i, C_{\text{nearest}})^2}.$$

Relevance: Weighted k-means is particularly useful in applications where data points contribute unequally to the clustering objective, such as quantization for large language models (LLMs) [13, 14].

Implementation Note: The algorithms detailed in this thesis support both unweighted and weighted k-means clustering, ensuring flexibility for practical use cases.

2.5 Relevant Works

For the special case of 1D k-means clustering, significant progress has been made in achieving globally optimal solutions. Wang and Song [9] introduced a dynamic programming algorithm with a time complexity of $O(k \cdot n^2)$. This was later improved by Grønlund et al. [10], who developed an $O(n)$-time algorithm for computing the exact optimal clustering in 1D.

Despite these advancements, widely used implementations such as `scikit-learn`'s k-means algorithm [11] do not include optimizations for the 1D case. Instead, they treat 1D data as a general instance of higher-dimensional clustering. `scikit-learn` uses Cython [18] to achieve efficient performance for general-purpose Lloyd's algorithm and k-means++ initialization, but it still operates with $O(l \cdot k \cdot n)$ initialization time and $O(i \cdot k \cdot n)$ iteration time for 1D data.

This thesis is the first to address the gap of optimizing Lloyd's algorithm and k-means++ initialization specifically for 1D clustering, targeting scenarios where speed and scalability are paramount. While prior works achieve globally optimal solutions, they do not target scenarios where speed and scalability are more critical than exact optimality. By leveraging the properties of sorted data, the proposed algorithms achieve logarithmic runtime while delivering high-quality clustering results, offering a practical alternative for latency-critical and large-scale applications.

3. Novel Approaches and Proof of Validity

This chapter presents our proposed approaches for solving general k-cluster problems in one dimension. We introduce two algorithms: the *k-cluster algorithm* and the *2-cluster algorithm.* Both methods exploit the structure of one-dimensional data and utilize sorting, prefix sums, and binary search to achieve significant computational efficiency compared to traditional clustering methods.

Finding Cluster Boundaries: For one-dimensional data, determining a point's cluster assignment involves identifying the interval it falls into. The boundaries between clusters are the arithmetic midpoints of consecutive centroids. When both the data and centroids are sorted, these boundaries can be efficiently located using binary search, requiring $O(k \cdot \log n)$ time. If centroids need sorting, an additional $O(k \cdot \log k)$ time is needed; however, as $k \leq n$, the total time remains $O(k \cdot \log n)$. This approach forms the basis for subsequent optimizations.

3.1 The k-Cluster Algorithm

The k-cluster algorithm is a one-dimensional adaptation of greedy k-means++ initialization followed by Lloyd's algorithm iterations, and can be defined for both weighted and unweighted data. The algorithm leverages:

1. **Sorted Data:** Sorting the input array X of size n in ascending order allows quick determination of cluster assignments via binary search on cluster boundaries.
2. **Prefix Sums:** Precomputing prefix sums enables constant-time computation of weighted and unweighted sums, means, and inertia values over arbitrary intervals. Specifically, for weighted data:

$$W[j] = \sum_{i=1}^{j} w_i, \quad (WX)[j] = \sum_{i=1}^{j} w_i x_i, \quad (WX^2)[j] = \sum_{i=1}^{j} w_i x_i^2.$$

For unweighted data, this simplifies to:

$$X^{(1)}[j] = \sum_{i=1}^{j} x_i, \quad X^{(2)}[j] = \sum_{i=1}^{j} x_i^2.$$

3. **Binary Search:** Binary search is central to efficiently determining cluster boundaries and performing weighted random sampling during initialization.

3.1.1 WCSS and Prefix Sums

The within-cluster sum of squares (WCSS) for weighted data is defined as:

$$\text{WCSS} = \sum_{i=1}^{k} \sum_{x_j \in C_i} w_j (x_j - \mu_i)^2,$$

where C_i is the i-th cluster, μ_i is its centroid, and w_j is the weight of point x_j. Expanding the squared term:

$$(x_j - \mu_i)^2 = x_j^2 - 2x_j\mu_i + \mu_i^2,$$

yields:

$$\text{WCSS}_i = \sum_{x_j \in C_i} w_j x_j^2 - 2\mu_i \sum_{x_j \in C_i} w_j x_j + \mu_i^2 \sum_{x_j \in C_i} w_j.$$

Using prefix sums:

$$\begin{aligned}
\sum_{x_j \in C_i} w_j x_j^2 &= (WX^2)[b_i] - (WX^2)[b_{i-1}], \\
\sum_{x_j \in C_i} w_j x_j &= (WX)[b_i] - (WX)[b_{i-1}], \\
\sum_{x_j \in C_i} w_j &= W[b_i] - W[b_{i-1}].
\end{aligned}$$

where b_{i-1} and b_i are the cluster boundaries. The centroid is:

$$\mu_i = \frac{(WX)[b_i] - (WX)[b_{i-1}]}{W[b_i] - W[b_{i-1}]}.$$

All these queries take $O(1)$ time per cluster once the prefix sums are computed. Hence, WCSS and centroid calculations are efficient, requiring only $O(k)$ time across all k clusters, if cluster boundaries are known. Determining cluster boundaries costs $O(k \log n)$, so the total cost for WCSS calculation given centroids is $O(k \log n)$.

3.1.2 Greedy k-Means++ Initialization

The k-means++ initialization selects centroids such that new centroids are chosen with probabilities proportional to their squared distances from the closest existing centroid. Our method efficiently implements this using **binary search** combined with **cumulative sum queries**.

Steps for Initialization:

1. **First Centroid Selection:** The first centroid is chosen randomly, weighted by the point weights w_j. To achieve this:
   - Given the cumulative sum of weights $W[j] = \sum_{i=1}^{j} w_i$,
   - Generate a random number $r \in [0, W[n]]$,
   - Perform binary search on W to find the smallest j such that $W[j] \geq r$, and thus the corresponding point x_j. This step costs $O(\log n)$.
2. **Subsequent Centroid Selection:** For each new centroid:
   (a) **Binary Search for Cluster Assignments:** Given the existing centroids, determine the cluster boundaries using a binary search of consecutive centroid midpoints. This step costs $O(k \log n)$.

   (b) **Cumulative Sum for Squared Distances:** To sample a new centroid, we need the cumulative sum of squared distances D_i^2, where D_i is the distance of x_i to its closest centroid. The cumulative sum $S[j]$ is defined as:

   $$S[j] = \sum_{i=1}^{j} D_i^2.$$

   Importantly: S is not explicitly constructed. Instead for each query j on S:

   - The sum of squared distances D_i^2 are obtained using prefix sums over the k clusters, up to the jth point. This is equivalent to calculating the WCSS up to the jth point. For each cluster, this sum can be retrieved in $O(1)$ time, as the cluster boundaries are known.

- Querying $S[j]$ for any j requires $O(k)$ time, as it aggregates contributions from all relevant clusters.

(c) **Binary Search on S:**

- Generate a random number $r \in [0, S[n]]$,
- Perform binary search on S to find the smallest j such that $S[j] \geq r$.

Each binary search involves $O(\log n)$ queries of S, where each query takes $O(k)$. Thus, the total cost for sampling one new centroid is $O(k \cdot \log n)$, and for l candidates, $O(l \cdot k \cdot \log n)$.

(d) **Greedy Candidate Selection:** For each candidate:

- Update cluster boundaries using binary search ($O(k \cdot \log n)$),
- Compute the total WCSS using prefix sums ($O(k)$).

The candidate minimizing the total WCSS is selected as the next centroid. For l candidates, this step costs $O(l \cdot k \cdot \log n)$.

(e) **Combined Initialization Time Complexity:** Combining the steps, the total cost for generating and evaluating l candidates per new centroid is $O(k \cdot \log n \cdot + l \cdot k \cdot \log n + l \cdot k \cdot \log n) = O(l \cdot k \cdot \log n)$.

3.1.3 Complexity Analysis

The overall time complexity of the k-cluster algorithm is as follows:

Greedy k-Means++ Initialization: As detailed in the previous section:

- Selecting the first centroid using weighted sampling costs $O(\log n)$,
- Each subsequent centroid requires $O(l \cdot k \cdot \log n)$, where l is the number of local trials.

The total cost for initialization across k centroids is therefore:

$$O(l \cdot k^2 \cdot \log n).$$

Lloyd's Algorithm Iterations: Each iteration of Lloyd's algorithm consists of:

- Updating cluster boundaries via binary search: $O(k \cdot \log n)$,
- Updating centroids using prefix sums: $O(k)$.

For i iterations, the total cost is:

$$O(i \cdot k \cdot \log n).$$

Overall Time Complexity: The combined cost of greedy k-means++ initialization and Lloyd's algorithm is:

$$O(l \cdot k^2 \cdot \log n) + O(i \cdot k \cdot \log n).$$

This does not account for the initial overhead of sorting the data and calculating prefix sums, which cost $O(n \log n)$ and $O(n)$, respectively.

Comparing against conventional implementations of $O(l \cdot k \cdot n) + O(i \cdot k \cdot n)$, note how the dependence on n (dataset size) has decreased, at the cost of quadratic complexity in k during initialization. However, since $k \ll n$ in most practical cases, this tradeoff is justified. For experimental speedup proofs, see Chapter 4.

3.2 The 2-Cluster Algorithm

For the 2-cluster problem in one-dimensional sorted data, the task reduces to finding a single **cluster boundary** that divides the data into two contiguous clusters. To efficiently locate this boundary, we iteratively refine a **search scope** to identify the correct **division interval**. A division interval is defined as the interval between two consecutive points in the sorted data that contains the cluster boundary.

Note: For all discussions in this section, all notions of direction (i.e., left or right) are with respect to the one-dimensional coordinate axis along which the data points are sorted. Thus, *left* refers to decreasing x-values and *right* refers to increasing x-values.

3.2.1 Definitions and Key Observations

- A **division interval** is defined as the interval between two consecutive points $x_{\text{div\_left}}$ and $x_{\text{div\_right}}$ in sorted data that contains the cluster boundary (note that, of course, $\text{div\_left} + 1 = \text{div\_right}$).

- The **midpoint** for a division interval is defined as:

$$\text{Midpoint} = \frac{\mu_{\text{left}} + \mu_{\text{right}}}{2},$$

  where μ_{left} and μ_{right} are the centroids of the left and right clusters defined by the division interval, respectively. These centroids are computed with prefix sums, using:

$$\mu_{\text{left}} = \frac{\sum_{i=1}^{\text{div\_left}} w_i x_i}{\sum_{i=1}^{\text{div\_left}} w_i}, \quad \mu_{\text{right}} = \frac{\sum_{i=\text{div\_right}}^{n} w_i x_i}{\sum_{i=\text{div\_right}}^{n} w_i}.$$

  The prefix sums W and WX, as defined for the k-cluster algorithm, allow this calculation to be done in $O(1)$ time.

- A division interval is classified as follows:

  - **Right-pointing:** The midpoint lies to the right of $x_{\text{div\_right}}$.
  - **Left-pointing:** The midpoint lies to the left of $x_{\text{div\_left}}$.
  - **Convergent:** The midpoint lies within the division interval itself, indicating a Lloyd's algorithm convergence.

  Note that every division interval can be classified into exactly one of these three categories.

- The **search scope** refers to the range of candidate division intervals, which is iteratively refined during the binary search to locate a convergent interval.

3.2.2 Algorithm Description

The algorithm aims to identify the correct division interval (i.e., a convergent interval) using binary search. The key steps are as follows:

1. **Initialize the Search Scope:** Start with the whole scope—that is, all possible division intervals ranging from the first interval $[x_1, x_2]$ to the last interval $[x_{n-1}, x_n]$.

2. **Iteratively Query the Center Interval:** At each step:

   - Select the **center division interval** within the current search scope.
   - Compute the centroids μ_{left} and μ_{right} and calculate the midpoint:

   $$\text{Midpoint} = \frac{\mu_{\text{left}} + \mu_{\text{right}}}{2}.$$

3. **Refine the Search Scope:** Compare the midpoint to the endpoints $x_{\text{div\_left}}$ and $x_{\text{div\_right}}$ of the queried division interval:

   - If the interval is **right-pointing**, exclude all intervals to the left of the current interval, including itself.
   - If the interval is **left-pointing**, exclude all intervals to the right of the current interval, including itself.
   - If the interval is **convergent**, terminate; the cluster boundary has been found.

4. **Repeat Until Convergence:** Continue the process until a convergent interval is found.

The binary search guarantees that the number of candidate intervals is halved at each iteration, ensuring $O(\log n)$ convergence.

3.2.3 Proof of Validity

To prove the correctness of the algorithm, we rely on the monotonic behavior of the centroids' midpoint and the structure of the division intervals.

1. Monotonic Behavior of the Midpoint: When the division interval $[x_{\text{div\_left}}, x_{\text{div\_right}}]$ is shifted one step to the right, i.e., to $[x_{\text{div\_right}}, x_{\text{div\_right}+1}]$:

- The point $x_{\text{div\_right}}$, which was previously the leftmost point in the right cluster, is excluded from the right cluster and added to the left cluster.

- This change causes the **left centroid** μ_{left} to increase, as a point with a larger value has been included in the left cluster.
- Simultaneously, the **right centroid** μ_{right} also increases, as the smallest point in the right cluster has been removed.
- Since both centroids increase, the new **midpoint**:

  $$\text{Midpoint}^{\text{new}} = \frac{\mu_{\text{left}}^{\text{new}} + \mu_{\text{right}}^{\text{new}}}{2}$$

  is greater than the old midpoint:

  $$\text{Midpoint}^{\text{new}} > \text{Midpoint}^{\text{old}}.$$

Similarly, shifting the division interval one step to the left causes the midpoint to strictly decrease.

2. Behavior of Right-Pointing and Left-Pointing Intervals: The monotonic behavior of the midpoint ensures the following:

- **Right-Pointing Intervals:** Suppose a division interval $[x_{\text{div\_left}}, x_{\text{div\_right}}]$ is right-pointing, so:

  $$x_{\text{div\_right}} < \text{Midpoint}^{\text{old}}.$$

  After shifting the interval one step to the right to the new interval $[x_{\text{div\_right}}, x_{\text{div\_right}+1}]$, the corresponding new midpoint strictly increases:

  $$x_{\text{div\_right}} < \text{Midpoint}^{\text{old}} < \text{Midpoint}^{\text{new}}.$$

  For the new interval $[x_{\text{div\_right}}, x_{\text{div\_right}+1}]$ to become left-pointing, the new midpoint, $\text{Midpoint}^{\text{new}}$, would have to be less than $x_{\text{div\_right}}$. But directly from the inequality above:

  $$x_{\text{div\_right}} < \text{Midpoint}^{\text{new}}.$$

  Thus, a right-pointing interval cannot suddenly become left-pointing after a rightward shift; it remains right-pointing or becomes convergent.

- **Left-Pointing Intervals:** By a symmetric argument, for a left-pointing interval, shifting one step to the left strictly decreases the midpoint, ensuring it cannot suddenly become right-pointing. Instead, it remains left-pointing or becomes convergent.

This ensures that from the perspective of shifting the intervals (one step at a time in the chosen direction), right-pointing and left-pointing intervals cannot directly switch roles in a single move.

Convergent Interval in a Search Scope In any search scope where the first interval is right-pointing and the last interval is left-pointing, there must exist at least one convergent interval between them. This follows from the fact that a right-pointing interval cannot immediately precede a left-pointing interval when moving stepwise to the right.

Convergence of the Binary Search For the entire search scope, the first division interval (between x_1 and x_2) is either right-pointing or convergent, and the last division interval (between x_{n-1} and x_n) is either left-pointing or convergent. Therefore, by the principle established above, there must exist at least one convergent interval within the entire scope. The previously detailed binary search iteratively reduces this scope in a way such that a convergent interval is always included, eventually narrowing the scope to a single convergent interval.

3.2.4 Limitations

While the monotonicity argument ensures that right-pointing and left-pointing intervals cannot directly switch roles when shifting in the direction they point (rightward for right-pointing, leftward for left-pointing), this guarantee does not hold when shifting in the opposite direction. Multiple local minima in Lloyd's algorithm can produce patterns like:

$$RRRCLLLRRRCLLL,$$

where R denotes right-pointing, L denotes left-pointing, and C denotes convergent intervals. In such scenarios:

- Moving a left-pointing interval to the right can produce a right-pointing interval (and vice versa).
- The algorithm still finds at least one convergent interval in $O(\log n)$ time, but the found cluster boundary may correspond to a local minimum rather than the global optimum.

3.2.5 Algorithm Guarantees

By leveraging the monotonic behavior of the midpoint and the fact that a convergent interval must exist between the first right-pointing and last left-pointing intervals, the binary search efficiently identifies a convergent interval representing a cluster boundary. The algorithm achieves this in $O(\log n)$ time without relying on iterative steps of Lloyd's algorithm.

In the presence of multiple local minima, the algorithm is guaranteed to converge to a valid solution, though it may not necessarily find the global optimum. Similar to the k-cluster algorithm, an initial preprocessing step is required, which includes sorting the data in $O(n \log n)$ time and computing prefix sums in $O(n)$ time.

3.3 Summary

In this chapter, we introduced two novel algorithms for solving the k-cluster problem in one-dimensional sorted data: the **k-cluster algorithm** and the **2-cluster algorithm**. Both methods leverage key observations about the structure of one-dimensional data to achieve significant computational efficiency.

- The **k-cluster algorithm** combines greedy k-means++ initialization with efficient Lloyd's iterations. By exploiting sorted data, prefix sums, and binary search:

  - The initialization process achieves a complexity of $O(l \cdot k^2 \cdot \log n)$, where k is the number of clusters and l is the number of local trials.
  - Each iteration of Lloyd's algorithm requires $O(k \log n)$ time, ensuring efficient updates of cluster boundaries and centroids.

- The **2-cluster algorithm** focuses on the specific case of $k = 2$, where the problem reduces to locating a single cluster boundary. Using a binary search over division intervals:
  - The midpoint of centroids behaves monotonically, allowing us to refine the search scope iteratively.
  - The algorithm achieves a total complexity of $O(\log n)$ and guarantees convergence to a local minimum of Lloyd's algorithm.

Both algorithms demonstrate how the structure of one-dimensional data enables faster computations compared to traditional clustering methods. In cases with multiple local minima, the solutions produced may not be globally optimal—a limitation shared with Lloyd's algorithm. Nonetheless, the proposed methods strike an effective balance between accuracy and computational efficiency, making them highly suitable for practical applications.

4. Experiments

This chapter presents experimental results comparing the proposed 1D k-means algorithms against the widely used `scikit-learn` implementation. The evaluation focuses on runtime performance and clustering quality, using both real and synthetic datasets.

4.1 Implementation Details

The proposed algorithms have been implemented as an open-source Python package, `flash1dkmeans`, available on GitHub[1] and PyPI[2]. The package is built on top of `NumPy` [19] and `Numba` [12] for efficient computation. The implementation includes both the k-cluster and 2-cluster algorithms, as well as the preprocessing steps for the sorting and prefix sum computations. The preprocessing step is optional and can be disabled if the input data is already in adequate form. More details on the implementation can be found in the Appendix.

4.2 Experimental Setup

The experiments were conducted on two processors: an Intel i9-13900K and an Apple M3. For the clustering quality and runtime comparisons, only the i9-13900K results are presented, as trends were consistent across both processors. Both real and synthetic 1D datasets were used, with varying sizes and numbers of clusters. The real datasets include the `iris` and `california housing` datasets from `scikit-learn`, while synthetic datasets were generated using the `sklearn.datasets.make_blobs` and `numpy.random.random_sample` functions.

Comparisons were made against `scikit-learn`'s implementation of k-means[3] at default configuration, which utilizes greedy k-means++ initialization paired with Lloyd's algorithm. For fair comparison, only a single thread was used. The number of local trials for the greedy k-means++ initialization, l, defaults to $2 + \log k$ in `scikit-learn`. This was matched in `flash1dkmeans`.

[1]`https://github.com/SyphonArch/flash1dkmeans`
[2]`https://pypi.org/project/flash1dkmeans/`
[3]`https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html`

4.3 Results: Clustering Quality

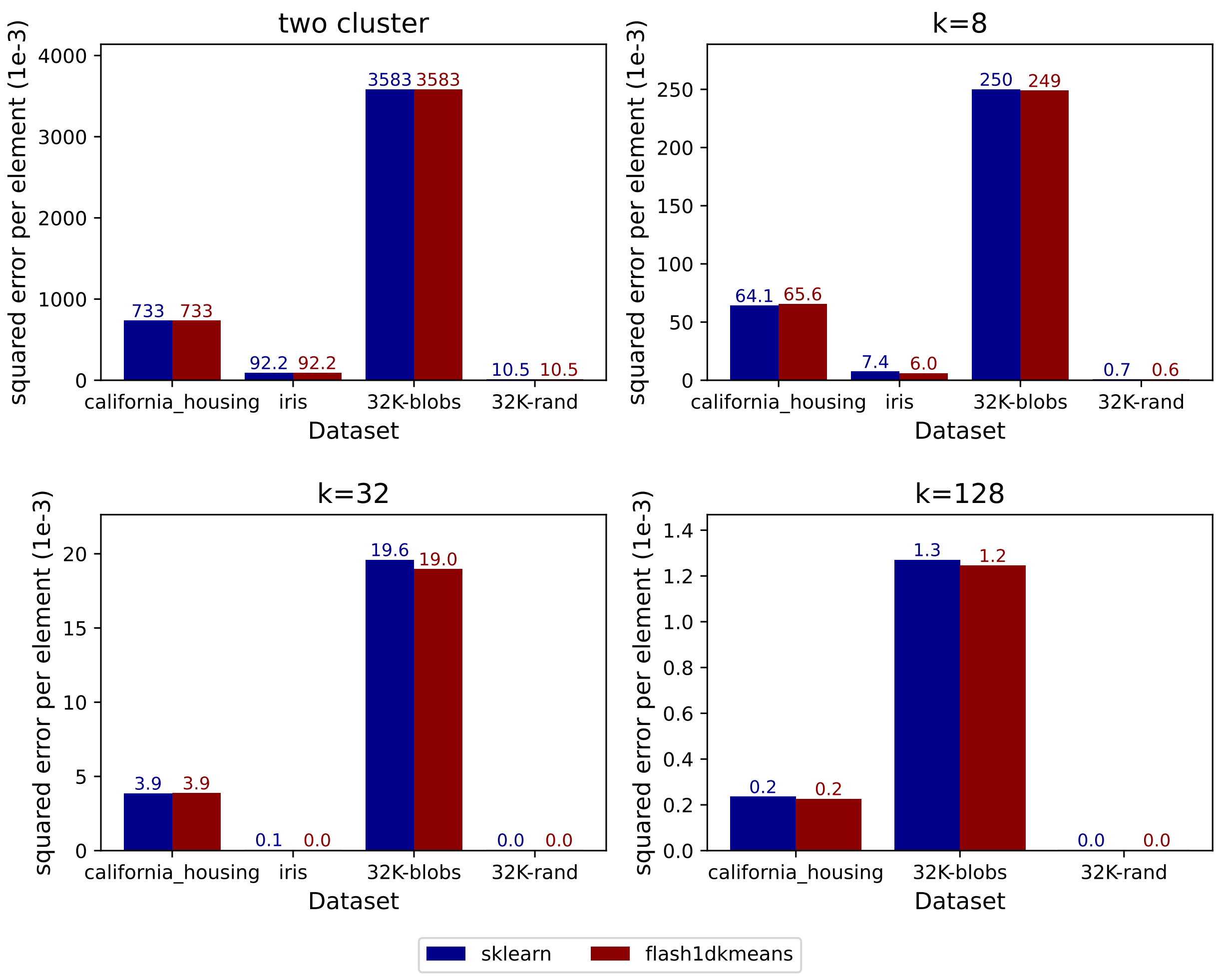

Figure 4.1: Squared error comparison of the proposed two-cluster algorithm and k-cluster algorithms in `flash1dkmeans` against `scikit-learn` on real and synthetic datasets. Lower is better.

The clustering quality comparison between the proposed 2-cluster algorithm and the k-cluster algorithms in `flash1dkmeans` and `scikit-learn` is shown in Figure 4.1. The `iris` dataset was excluded for $k = 128$ as the number of data points was insufficient. Across the configurations, the clustering quality, measured by squared error, is consistent with the baseline k-means algorithm. For both the 2-cluster and k-cluster algorithms, this confirms that the proposed method produces clustering results equivalent or close to the k-means algorithm in `scikit-learn`.

4.4 Results: Runtime Performance

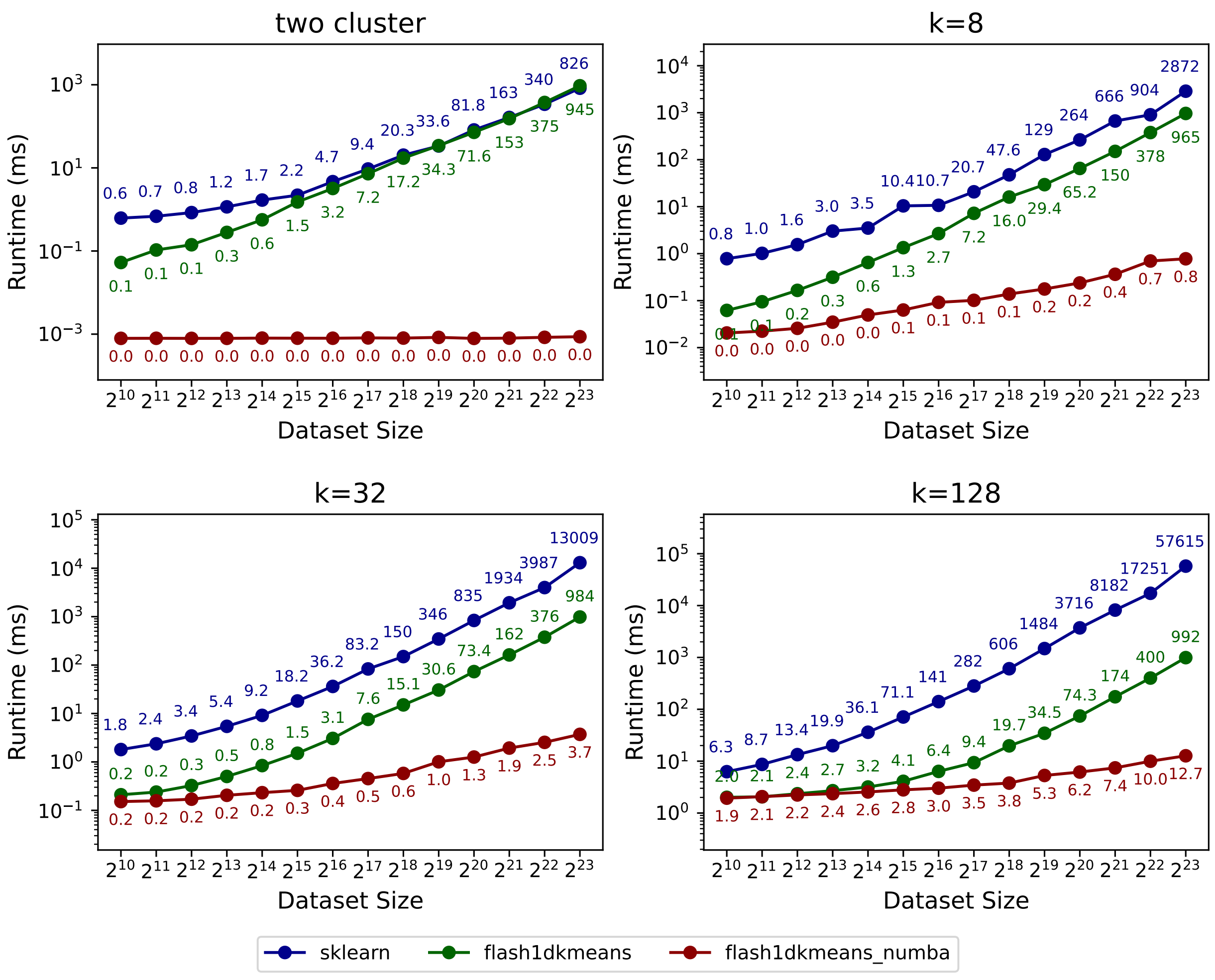

Figure 4.2: Runtime comparison of the proposed two-cluster algorithm and k-cluster algorithms in `flash1dkmeans` against `scikit-learn` on datasets of varying sizes. Lower is better.

Figure 4.2 compares the runtime of the proposed 2-cluster and k-cluster algorithms in `flash1dkmeans` to `scikit-learn`. The runtime of `flash1dkmeans` includes preprocessing time, while `flash1dkmeans_numba` measures only the main algorithm runtime, assuming preprocessed and sorted input data.

The results confirm that the k-cluster algorithm in `flash1dkmeans` achieves substantial runtime improvements compared to `scikit-learn`, showcasing the logarithmic dependence on dataset size. For example, for $k = 128$ and dataset size 2^{23}, the speedup exceeds 4500x. However, for the 2-cluster algorithm on larger datasets, the $O(n \log n)$ preprocessing time becomes a bottleneck, limiting the overall speedup. Nevertheless, the main algorithm runtimes confirm the log-time efficiency of the proposed optimizations, and `flash1dkmeans` with the $O(n \log n)$ preprocessing still outperforms `scikit-learn` by a large margin in most cases.

4.5 Results: LLM Quantization

	Seed (k-cluster algorithm)			**Upscale (2-cluster algorithm)**		
Processor	**sklearn**	**flash1dkmeans**	**speedup**	**sklearn**	**flash1dkmeans**	**speedup**
i9-13900K	433 ms	56 ms	7.7x	10551 ms	34 ms	310x
Apple M3	572 ms	114 ms	5.0x	7585 ms	29 us	262x

Table 4.1: Performance comparison of `flash1dkmeans` against `scikit-learn` for both k-cluster and 2-cluster algorithms on i9-13900K and Apple M3 processors.

Table 4.1 compares the performance of `flash1dkmeans` and `scikit-learn` in the two-step LLM quantization process proposed in Any-Precision LLM [14]. As an example, we quantize the 214th output channel (size 14,336) of the `mlp.down_proj` linear module in the 8th layer of Llama-3-8B [20] from 3-bit to 8-bit, as described in the original work. We measure the total time over 100 runs for reliability.

The first step involves generating a seed model using k-means clustering, followed by incrementally subdividing clusters into two via k-means in the second step. The preprocessing from the seed generation step is reusable during the upscale phase, enabling the $O(\log n)$-time 2-cluster algorithm to be particularly efficient. Results demonstrate that `flash1dkmeans` achieves remarkable speedups, surpassing 300x in the upscale step on an i9-13900K processor.

4.6 Summary

Experimental results demonstrate that the proposed algorithms in `flash1dkmeans` achieve comparable clustering quality to `scikit-learn` while offering significant runtime improvements. The k-cluster algorithm exhibits logarithmic time complexity with speedups exceeding 4500x on large datasets, while the 2-cluster algorithm achieves remarkable efficiency when given preprocessed inputs. These results validate the effectiveness of the proposed optimizations for both general clustering tasks and emerging applications such as LLM quantization.

5. Conclusion

This thesis presents optimized algorithms for k-means++ initialization and Lloyd's algorithm, specifically designed for one-dimensional (1D) clustering. By leveraging the structure of sorted data, the proposed methods replace the linear dependence on dataset size n with logarithmic factors, significantly improving computational efficiency while maintaining clustering quality. The optimized greedy k-means++ initialization achieves a time complexity of $O(l \cdot k^2 \cdot \log n)$, while Lloyd's algorithm iterations achieve $O(i \cdot k \cdot \log n)$, where l is the number of local trials, and i the number of iterations. Additionally, the binary search-based algorithm for the two-cluster case deterministically converges to a Lloyd's algorithm solution in $O(\log n)$, bypassing iterative refinements entirely.

These methods offer significant benefits for latency-critical tasks and large-scale clustering problems. The practical relevance of these contributions is demonstrated in key applications such as quantization for large language models (LLMs), where the optimized algorithms achieve a 300-fold speedup over `scikit-learn`, as discussed in Section 4.5. To ensure ease of use, a Python implementation of the proposed methods, optimized with just-in-time (JIT) compilation, is provided for practical deployment.

By exploiting the structure of 1D data, this thesis demonstrates that substantial computational improvements are achievable for k-means clustering, reducing dataset size dependencies from linear to logarithmic. These contributions address an important gap in the existing literature and deliver efficient, high-performance clustering tools ready for real-world applications.

References

[1] R. Xu and D. Wunsch, “Survey of clustering algorithms,” *IEEE Transactions on Neural Networks*, vol. 16, no. 3, pp. 645–678, 2005.

[2] J. MacQueen, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, L. M. Le Cam and J. Neyman, Eds. Berkeley, CA: University of California Press, 1967, pp. 281–297.

[3] P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay, “Clustering large graphs via the singular value decomposition,” *Mach. Learn.*, vol. 56, no. 1–3, p. 9–33, Jun. 2004. [Online]. Available: https://doi.org/10.1023/B:MACH.0000033113.59016.96

[4] S. Lloyd, “Least squares quantization in pcm,” *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[5] J. Max, “Quantizing for minimum distortion,” *IRE Transactions on Information Theory*, vol. 6, no. 1, pp. 7–12, 1960.

[6] V. Arnaboldi, M. Conti, M. La Gala, A. Passarella, and F. Pezzoni, “Ego network structure in online social networks and its impact on information diffusion,” *Computer Communications*, vol. 76, p. 26–41, Feb. 2016. [Online]. Available: http://dx.doi.org/10.1016/j.comcom.2015.09.028

[7] O. Jeske, M. Jogler, J. Petersen, J. Sikorski, and C. Jogler, “From genome mining to phenotypic microarrays: Planctomycetes as source for novel bioactive molecules,” *Antonie van Leeuwenhoek*, vol. 104, 08 2013.

[8] D. Pennacchioli, M. Coscia, S. Rinzivillo, F. Giannotti, and D. Pedreschi, “The retail market as a complex system,” *EPJ Data Science*, vol. 3, 12 2014.

[9] H. Wang and M. Song, “Ckmeans.1d.dp: Optimal k-means clustering in one dimension by dynamic programming,” *The R Journal*, vol. 3, pp. 29–33, 12 2011.

[10] A. Grønlund, K. G. Larsen, A. Mathiasen, J. S. Nielsen, S. Schneider, and M. Song, "Fast exact k-means, k-medians and bregman divergence clustering in 1d," 2018. [Online]. Available: https://arxiv.org/abs/1701.07204

[11] 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 Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. [Online]. Available: https://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html

[12] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: a llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2833157.2833162

[13] S. Kim, C. R. C. Hooper, A. Gholami, Z. Dong, X. Li, S. Shen, M. W. Mahoney, and K. Keutzer, "SqueezeLLM: Dense-and-sparse quantization," in *Proceedings of the 41st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp, Eds., vol. 235. PMLR, 21–27 Jul 2024, pp. 23 901–23 923. [Online]. Available: https://proceedings.mlr.press/v235/kim24f.html

[14] Y. Park, J. Hyun, S. Cho, B. Sim, and J. W. Lee, "Any-precision LLM: Low-cost deployment of multiple, different-sized LLMs," in *Proceedings of the 41st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp, Eds., vol. 235. PMLR, 21–27 Jul 2024, pp. 39 682–39 701. [Online]. Available: https://proceedings.mlr.press/v235/park24e.html

[15] D. Arthur and S. Vassilvitskii, "k-means++: the advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07. USA: Society for Industrial and Applied Mathematics, 2007, p. 1027–1035.

[16] 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, vol. 40, no. 1, pp. 200–210, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417412008767

[17] A. Bhattacharya, J. Eube, H. Röglin, and M. Schmidt, "Noisy, greedy and not so greedy k-means++," 12 2019.

[18] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[19] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[20] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, A. Yang, A. Fan, A. Goyal, A. Hartshorn, A. Yang, A. Mitra, A. Sravankumar, A. Korenev, A. Hinsvark, A. Rao, A. Zhang, A. Rodriguez, A. Gregerson, A. Spataru, B. Roziere, B. Biron, B. Tang, B. Chern, C. Caucheteux, C. Nayak, C. Bi, C. Marra, C. McConnell, C. Keller, C. Touret, C. Wu, C. Wong, C. C. Ferrer, C. Nikolaidis, D. Allonsius, D. Song, D. Pintz, D. Livshits, D. Wyatt, D. Esiobu, D. Choudhary, D. Mahajan, D. Garcia-Olano, D. Perino, D. Hupkes, E. Lakomkin, E. AlBadawy, E. Lobanova, E. Dinan, E. M. Smith, F. Radenovic, F. Guzmán, F. Zhang, G. Synnaeve, G. Lee, G. L. Anderson, G. Thattai, G. Nail, G. Mialon, G. Pang, G. Cucurell, H. Nguyen, H. Korevaar, H. Xu, H. Touvron, I. Zarov, I. A. Ibarra, I. Kloumann, I. Misra, I. Evtimov, J. Zhang, J. Copet, J. Lee, J. Geffert, J. Vranes, J. Park, J. Mahadeokar, J. Shah, J. van der Linde, J. Billock, J. Hong, J. Lee, J. Fu, J. Chi, J. Huang, J. Liu, J. Wang, J. Yu, J. Bitton, J. Spisak, J. Park, J. Rocca, J. Johnstun, J. Saxe, J. Jia, K. V. Alwala, K. Prasad, K. Upasani, K. Plawiak, K. Li, K. Heafield, K. Stone, K. El-Arini, K. Iyer, K. Malik, K. Chiu, K. Bhalla, K. Lakhotia, L. Rantala-Yeary, L. van der Maaten, L. Chen, L. Tan, L. Jenkins, L. Martin, L. Madaan, L. Malo, L. Blecher, L. Landzaat, L. de Oliveira, M. Muzzi, M. Pasupuleti, M. Singh,

M. Paluri, M. Kardas, M. Tsimpoukelli, M. Oldham, M. Rita, M. Pavlova, M. Kambadur, M. Lewis, M. Si, M. K. Singh, M. Hassan, N. Goyal, N. Torabi, N. Bashlykov, N. Bogoychev, N. Chatterji, N. Zhang, O. Duchenne, O. Çelebi, P. Alrassy, P. Zhang, P. Li, P. Vasic, P. Weng, P. Bhargava, P. Dubal, P. Krishnan, P. S. Koura, P. Xu, Q. He, Q. Dong, R. Srinivasan, R. Ganapathy, R. Calderer, R. S. Cabral, R. Stojnic, R. Raileanu, R. Maheswari, R. Girdhar, R. Patel, R. Sauvestre, R. Polidoro, R. Sumbaly, R. Taylor, R. Silva, R. Hou, R. Wang, S. Hosseini, S. Chennabasappa, S. Singh, S. Bell, S. S. Kim, S. Edunov, S. Nie, S. Narang, S. Raparthy, S. Shen, S. Wan, S. Bhosale, S. Zhang, S. Vandenhende, S. Batra, S. Whitman, S. Sootla, S. Collot, S. Gururangan, S. Borodinsky, T. Herman, T. Fowler, T. Sheasha, T. Georgiou, T. Scialom, T. Speckbacher, T. Mihaylov, T. Xiao, U. Karn, V. Goswami, V. Gupta, V. Ramanathan, V. Kerkez, V. Gonguet, V. Do, V. Vogeti, V. Albiero, V. Petrovic, W. Chu, W. Xiong, W. Fu, W. Meers, X. Martinet, X. Wang, X. Wang, X. E. Tan, X. Xia, X. Xie, X. Jia, X. Wang, Y. Goldschlag, Y. Gaur, Y. Babaei, Y. Wen, Y. Song, Y. Zhang, Y. Li, Y. Mao, Z. D. Coudert, Z. Yan, Z. Chen, Z. Papakipos, A. Singh, A. Srivastava, A. Jain, A. Kelsey, A. Shajnfeld, A. Gangidi, A. Victoria, A. Goldstand, A. Menon, A. Sharma, A. Boesenberg, A. Baevski, A. Feinstein, A. Kallet, A. Sangani, A. Teo, A. Yunus, A. Lupu, A. Alvarado, A. Caples, A. Gu, A. Ho, A. Poulton, A. Ryan, A. Ramchandani, A. Dong, A. Franco, A. Goyal, A. Saraf, A. Chowdhury, A. Gabriel, A. Bharambe, A. Eisenman, A. Yazdan, B. James, B. Maurer, B. Leonhardi, B. Huang, B. Loyd, B. D. Paola, B. Paranjape, B. Liu, B. Wu, B. Ni, B. Hancock, B. Wasti, B. Spence, B. Stojkovic, B. Gamido, B. Montalvo, C. Parker, C. Burton, C. Mejia, C. Liu, C. Wang, C. Kim, C. Zhou, C. Hu, C.-H. Chu, C. Cai, C. Tindal, C. Feichtenhofer, C. Gao, D. Civin, D. Beaty, D. Kreymer, D. Li, D. Adkins, D. Xu, D. Testuggine, D. David, D. Parikh, D. Liskovich, D. Foss, D. Wang, D. Le, D. Holland, E. Dowling, E. Jamil, E. Montgomery, E. Presani, E. Hahn, E. Wood, E.-T. Le, E. Brinkman, E. Arcaute, E. Dunbar, E. Smothers, F. Sun, F. Kreuk, F. Tian, F. Kokkinos, F. Ozgenel, F. Caggioni, F. Kanayet, F. Seide, G. M. Florez, G. Schwarz, G. Badeer, G. Swee, G. Halpern, G. Herman, G. Sizov, Guangyi, Zhang, G. Lakshminarayanan, H. Inan, H. Shojanazeri, H. Zou, H. Wang, H. Zha, H. Habeeb, H. Rudolph, H. Suk, H. Aspegren, H. Goldman, H. Zhan, I. Damlaj, I. Molybog, I. Tufanov, I. Leontiadis, I.-E. Veliche, I. Gat, J. Weissman, J. Geboski, J. Kohli, J. Lam,

J. Asher, J.-B. Gaya, J. Marcus, J. Tang, J. Chan, J. Zhen, J. Reizenstein, J. Teboul, J. Zhong, J. Jin, J. Yang, J. Cummings, J. Carvill, J. Shepard, J. McPhie, J. Torres, J. Ginsburg, J. Wang, K. Wu, K. H. U, K. Saxena, K. Khandelwal, K. Zand, K. Matosich, K. Veeraraghavan, K. Michelena, K. Li, K. Jagadeesh, K. Huang, K. Chawla, K. Huang, L. Chen, L. Garg, L. A, L. Silva, L. Bell, L. Zhang, L. Guo, L. Yu, L. Moshkovich, L. Wehrstedt, M. Khabsa, M. Avalani, M. Bhatt, M. Mankus, M. Hasson, M. Lennie, M. Reso, M. Groshev, M. Naumov, M. Lathi, M. Keneally, M. Liu, M. L. Seltzer, M. Valko, M. Restrepo, M. Patel, M. Vyatskov, M. Samvelyan, M. Clark, M. Macey, M. Wang, M. J. Hermoso, M. Metanat, M. Rastegari, M. Bansal, N. Santhanam, N. Parks, N. White, N. Bawa, N. Singhal, N. Egebo, N. Usunier, N. Mehta, N. P. Laptev, N. Dong, N. Cheng, O. Chernoguz, O. Hart, O. Salpekar, O. Kalinli, P. Kent, P. Parekh, P. Saab, P. Balaji, P. Rittner, P. Bontrager, P. Roux, P. Dollar, P. Zvyagina, P. Ratanchandani, P. Yuvraj, Q. Liang, R. Alao, R. Rodriguez, R. Ayub, R. Murthy, R. Nayani, R. Mitra, R. Parthasarathy, R. Li, R. Hogan, R. Battey, R. Wang, R. Howes, R. Rinott, S. Mehta, S. Siby, S. J. Bondu, S. Datta, S. Chugh, S. Hunt, S. Dhillon, S. Sidorov, S. Pan, S. Mahajan, S. Verma, S. Yamamoto, S. Ramaswamy, S. Lindsay, S. Lindsay, S. Feng, S. Lin, S. C. Zha, S. Patil, S. Shankar, S. Zhang, S. Zhang, S. Wang, S. Agarwal, S. Sajuyigbe, S. Chintala, S. Max, S. Chen, S. Kehoe, S. Satterfield, S. Govindaprasad, S. Gupta, S. Deng, S. Cho, S. Virk, S. Subramanian, S. Choudhury, S. Goldman, T. Remez, T. Glaser, T. Best, T. Koehler, T. Robinson, T. Li, T. Zhang, T. Matthews, T. Chou, T. Shaked, V. Vontimitta, V. Ajayi, V. Montanez, V. Mohan, V. S. Kumar, V. Mangla, V. Ionescu, V. Poenaru, V. T. Mihailescu, V. Ivanov, W. Li, W. Wang, W. Jiang, W. Bouaziz, W. Constable, X. Tang, X. Wu, X. Wang, X. Wu, X. Gao, Y. Kleinman, Y. Chen, Y. Hu, Y. Jia, Y. Qi, Y. Li, Y. Zhang, Y. Zhang, Y. Adi, Y. Nam, Yu, Wang, Y. Zhao, Y. Hao, Y. Qian, Y. Li, Y. He, Z. Rait, Z. DeVito, Z. Rosnbrick, Z. Wen, Z. Yang, Z. Zhao, and Z. Ma, “The llama 3 herd of models,” 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

Appendix

A. k-Cluster Algorithm Implementation

Provided below is the Python 3 implementation of the k-cluster algorithm discussed in this work. The Numba and Numpy packages are required, as well as the definition of macros like ARRAY_INDEX_DTYPE. For the fully integrated library please refer to section C..

```python
@numba.njit(cache=True)
def numba_kmeans_1d_k_cluster(
        sorted_X,
        n_clusters,
        max_iter,
        weights_prefix_sum, weighted_X_prefix_sum,
        weighted_X_squared_prefix_sum,
        start_idx,
        stop_idx,
        random_state=None,
):
    """An optimized kmeans for 1D data with n clusters.
    Exploits the fact that the data is 1D to optimize the calculations.
    Time complexity: O(k ^ 2 * log(k) * log(n) + i * log(n) * k)

    Args:
        sorted_X: np.ndarray
            The input data. Should be sorted in ascending order.
        n_clusters: int
            The number of clusters to generate
        max_iter: int
            The maximum number of iterations to run
        weights_prefix_sum: np.ndarray
            The prefix sum of the weights. Should be None if the data is
                unweighted.
        weighted_X_prefix_sum: np.ndarray
            The prefix sum of the weighted X
        weighted_X_squared_prefix_sum: np.ndarray
            The prefix sum of the weighted X squared
        start_idx: int
            The start index of the range to consider
        stop_idx: int
            The stop index of the range to consider
        random_state: int or None
            The random seed to use.

    Returns:
        centroids: np.ndarray
            The centroids of the clusters
        cluster_borders: np.ndarray
            The borders of the clusters
    """
    # set random_state
    set_np_seed_njit(random_state)

    cluster_borders = np.empty(n_clusters + 1, dtype=ARRAY_INDEX_DTYPE)
    cluster_borders[0] = start_idx
    cluster_borders[-1] = stop_idx

    centroids = _kmeans_plusplus(
        sorted_X, n_clusters,
        weights_prefix_sum, weighted_X_prefix_sum,
        weighted_X_squared_prefix_sum,
        start_idx, stop_idx,
    )
    sorted_centroids = np.sort(centroids)

    for _ in range(max_iter):
        new_cluster_borders = _centroids_to_cluster_borders(sorted_X,
            sorted_centroids, start_idx, stop_idx)

        if np.array_equal(cluster_borders, new_cluster_borders):
            break

        cluster_borders[:] = new_cluster_borders
        for i in range(n_clusters):
            cluster_start = cluster_borders[i]
            cluster_end = cluster_borders[i + 1]

            if cluster_end < cluster_start:
                raise ValueError("Cluster␣end␣is␣less␣than␣cluster␣start")

            if cluster_start == cluster_end:
                continue

            cluster_weighted_X_sum = query_prefix_sum(weighted_X_prefix_sum, cluster_start, cluster_end)
            cluster_weight_sum = query_prefix_sum(weights_prefix_sum, cluster_start, cluster_end)

            if cluster_weight_sum == 0:
                # if the sum of the weights is zero, we set the centroid to the mean of the cluster
                sorted_centroids[i] = sorted_X[cluster_start:cluster_end].mean()
            else:
                sorted_centroids[i] = cluster_weighted_X_sum / cluster_weight_sum

    return sorted_centroids, cluster_borders


@numba.njit(cache=True)
def _rand_choice_prefix_sum(arr, prob_prefix_sum, start_idx, stop_idx):
    """Randomly choose an element from arr according to the probability distribution given by prob_prefix_sum

    Time complexity: O(log n)

    Args:
        arr: np.ndarray
            The array to choose from
        prob_prefix_sum: np.ndarray
            The prefix sum of the probability distribution
        start_idx: int
            The start index of the range to consider
        stop_idx: int
            The stop index of the range to consider

    Returns:
        The chosen element
    """
    total_prob = query_prefix_sum(prob_prefix_sum, start_idx, stop_idx)
    selector = np.random.random_sample() * total_prob

    # Because we are using start_idx as the base, but the prefix sum is calculated from 0,
    # we need to adjust the selector if start_idx is not 0.
    adjusted_selector = selector + prob_prefix_sum[start_idx - 1] if start_idx > 0 else selector

    # Search for the index of the selector in the prefix sum, and add start_idx to get the index in the original array
    idx = np.searchsorted(prob_prefix_sum[start_idx:stop_idx], adjusted_selector) + start_idx

    return arr[idx]


@numba.njit(cache=True)
def _centroids_to_cluster_borders(X, sorted_centroids, start_idx, stop_idx):
    """Converts the centroids to cluster borders.
    The cluster borders are where the clusters are divided.
    The centroids must be sorted.

    Time complexity: O(k * log n)

    Args:
        X: np.ndarray
            The input data. Should be sorted in ascending order.
        sorted_centroids: np.ndarray
            The sorted centroids
        start_idx: int
            The start index of the range to consider
        stop_idx: int
            The stop index of the range to consider
```

```python

    Returns:
        np.ndarray: The cluster borders
    """
    midpoints = (sorted_centroids[:-1] + sorted_centroids[1:]) / 2
    cluster_borders = np.empty(len(sorted_centroids) + 1, dtype=ARRAY_INDEX_DTYPE)
    cluster_borders[0] = start_idx
    cluster_borders[-1] = stop_idx
    cluster_borders[1:-1] = np.searchsorted(X[start_idx:stop_idx], midpoints) + start_idx
    return cluster_borders


@numba.njit(cache=True)
def _calculate_inertia(sorted_centroids, centroid_ranges,
                       weights_prefix_sum, weighted_X_prefix_sum, weighted_X_squared_prefix_sum,
                       stop_idx):
    """Calculates the inertia of the clusters given the centroids.
    The inertia is the sum of the squared distances of each sample to the closest centroid.
    The calculations are done efficiently using prefix sums.

    Time complexity: O(k)

    Args:
        sorted_centroids: np.ndarray
            The centroids of the clusters
        centroid_ranges: np.ndarray
            The borders of the clusters
        weights_prefix_sum: np.ndarray
            The prefix sum of the weights. Should be None if the data is unweighted.
        weighted_X_prefix_sum: np.ndarray
            The prefix sum of the weighted X
        weighted_X_squared_prefix_sum: np.ndarray
            The prefix sum of the weighted X squared
        stop_idx: int
            The stop index of the range to consider
    """
    # inertia = sigma_i(w_i * abs(x_i - c)^2) = sigma_i(w_i * (x_i^2 - 2 * x_i * c + c^2))
    #         = sigma_i(w_i * x_i^2) - 2 * c * sigma_i(w_i * x_i) + c^2 * sigma_i(w_i)
    #         = sigma_i(weighted_X_squared) - 2 * c * sigma_i(weighted_X) + c^2 * sigma_i(weight)
    #  Note that the centroid c is the CLOSEST centroid to x_i, so the above calculation must be done for each cluster

    inertia = 0
    for i in range(len(sorted_centroids)):
        start = centroid_ranges[i]
        end = centroid_ranges[i + 1]

        if start >= stop_idx:
            break
        if end >= stop_idx:
            end = stop_idx

        if start == end:
            continue

        cluster_weighted_X_squared_sum = query_prefix_sum(weighted_X_squared_prefix_sum, start, end)
        cluster_weighted_X_sum = query_prefix_sum(weighted_X_prefix_sum, start, end)
        cluster_weight_sum = query_prefix_sum(weights_prefix_sum, start, end)

        inertia += (cluster_weighted_X_squared_sum - 2 * sorted_centroids[i] * cluster_weighted_X_sum +
                    sorted_centroids[i] ** 2 * cluster_weight_sum)

    return inertia


@numba.njit(cache=True)
def _rand_choice_centroids(X, centroids,
                           weights_prefix_sum, weighted_X_prefix_sum, weighted_X_squared_prefix_sum,
                           sample_size, start_idx, stop_idx):
```

```python
    """Randomly choose sample_size elements from X, weighted by the distance to the closest centroid.
    The weighted logic is implemented efficiently by utilizing the _calculate_inertia function.

    Time complexity: O(l * k * log n)

    Args:
        X: np.ndarray
            The input data. Should be sorted in ascending order.
        centroids: np.ndarray
            The centroids of the clusters
        is_weighted: bool
            Whether the data is weighted. If True, the weighted versions of the arrays should be provided.
        weights_prefix_sum: np.ndarray
            The prefix sum of the weights. Should be None if the data is unweighted.
        weighted_X_prefix_sum: np.ndarray
            The prefix sum of the weighted X
        weighted_X_squared_prefix_sum: np.ndarray
            The prefix sum of the weighted X squared
        sample_size: int
            The number of samples to choose
        start_idx: int
            The start index of the range to consider
        stop_idx: int
            The stop index of the range to consider

    Returns:
        np.ndarray: The chosen samples
    """
    sorted_centroids = np.sort(centroids)  # O(k log k)
    cluster_borders = _centroids_to_cluster_borders(X, sorted_centroids, start_idx, stop_idx)  # O(k log n)
    total_inertia = _calculate_inertia(sorted_centroids, cluster_borders,  # O(k)
                                       weights_prefix_sum, weighted_X_prefix_sum,
                                       weighted_X_squared_prefix_sum, stop_idx)
    selectors = np.random.random_sample(sample_size) * total_inertia
    results = np.empty(sample_size, dtype=centroids.dtype)

    for i in range(sample_size):  # O(l k log n)
        selector = selectors[i]
        floor = start_idx + 1
        ceiling = stop_idx
        while floor < ceiling:
            stop_idx_cand = (floor + ceiling) // 2
            inertia = _calculate_inertia(sorted_centroids, cluster_borders,  # O(k)
                                         weights_prefix_sum, weighted_X_prefix_sum,
                                         weighted_X_squared_prefix_sum, stop_idx_cand)
            if inertia < selector:
                floor = stop_idx_cand + 1
            else:
                ceiling = stop_idx_cand
        results[i] = X[floor - 1]

    return results


@numba.njit(cache=True)
def _kmeans_plusplus(X, n_clusters,
                     weights_prefix_sum, weighted_X_prefix_sum, weighted_X_squared_prefix_sum,
                     start_idx, stop_idx):
    """An optimized version of the kmeans++ initialization algorithm for 1D data.
    The algorithm is optimized for 1D data and utilizes prefix sums for efficient calculations.

    Time complexity: = O(k ^ 2 * log k * log n)

    Args:
        X: np.ndarray
            The input data
        n_clusters: int
            The number of clusters to choose
        weights_prefix_sum: np.ndarray
```

```
            The prefix sum of the weights. Should be None if the data is unweighted.
        weighted_X_prefix_sum: np.ndarray
            The prefix sum of the weighted X
        weighted_X_squared_prefix_sum: np.ndarray
            The prefix sum of the weighted X squared

    Returns:
        np.ndarray: The chosen centroids
    """
    centroids = np.empty(n_clusters, dtype=X.dtype)
    n_local_trials = 2 + int(np.log(n_clusters))

    # First centroid is chosen randomly according to sample_weight
    centroids[0] = _rand_choice_prefix_sum(X, weights_prefix_sum, start_idx, stop_idx)  # O(log n)

    for c_id in range(1, n_clusters):  # O(k^2 l log n)
        # Choose the next centroid randomly according to the weighted distances
        # Sample n_local_trials candidates and choose the best one

        centroid_candidates = _rand_choice_centroids(  # O(l k log n)
            X, centroids[:c_id],
            weights_prefix_sum, weighted_X_prefix_sum,
            weighted_X_squared_prefix_sum, n_local_trials,
            start_idx, stop_idx
        )

        best_inertia = np.inf
        best_centroid = None
        for i in range(len(centroid_candidates)): # O(l k log n)
            # O(k log k + k log n + k) = O(k log n), as k <= n
            centroids[c_id] = centroid_candidates[i]
            sorted_centroids = np.sort(centroids[:c_id + 1]) # O(k log k), I think we could avoid centroid sorting and use some linear algorithm, but the gain would be minimal, especially considering that k <= n, and most times k << n
            centroid_ranges = _centroids_to_cluster_borders(X, sorted_centroids, start_idx, stop_idx)  # O(k log n)
            inertia = _calculate_inertia(sorted_centroids, centroid_ranges,  # O(k)
                                         weights_prefix_sum, weighted_X_prefix_sum,
                                         weighted_X_squared_prefix_sum, stop_idx)
            if inertia < best_inertia:
                best_inertia = inertia
                best_centroid = centroid_candidates[i]
        centroids[c_id] = best_centroid

    return centroids
```

B. 2-Cluster Agorithm Implementation

Provided below is the Python 3 implementation of the 2-cluster algorithm discussed in this work. The `Numba` and `Numpy` packages are required, as well as the definition of macros like `ARRAY_INDEX_DTYPE`. For the fully integrated library please refer to section C..

```
@numba.njit(cache=True)
def numba_kmeans_1d_two_cluster(
        sorted_X,
        weights_prefix_sum,
        weighted_X_prefix_sum,
        start_idx,
        stop_idx,
):
    """An optimized kmeans for 1D data with 2 clusters, weighted version.
    Utilizes a binary search to find the optimal division point.
    Time complexity: O(log(n))

    Args:
        sorted_X: np.ndarray
            The input data. Should be sorted in ascending order.
        weights_prefix_sum: np.ndarray
            The prefix sum of the sample weights. Should be None if the data is
                unweighted.
        weighted_X_prefix_sum: np.ndarray
            The prefix sum of (the weighted) X.
        start_idx: int
            The start index of the range to consider.
        stop_idx: int
            The stop index of the range to consider.

    Returns:
        centroids: np.ndarray
            The centroids of the two clusters, shape (2,)
        cluster_borders: np.ndarray
            The borders of the two clusters, shape (3,)

    WARNING: X should be sorted in ascending order before calling this function.
    """
    size = stop_idx - start_idx
    centroids = np.empty(2, dtype=sorted_X.dtype)
    cluster_borders = np.empty(3, dtype=ARRAY_INDEX_DTYPE)
    cluster_borders[0] = start_idx
    cluster_borders[2] = stop_idx
    # Remember to set cluster_borders[1] as the division point

    if size == 1:
        centroids[0], centroids[1] = sorted_X[start_idx], sorted_X[start_idx]
        cluster_borders[1] = start_idx + 1
        return centroids, cluster_borders

    if size == 2:
        centroids[0], centroids[1] = sorted_X[start_idx], sorted_X[start_idx +
            1]
        cluster_borders[1] = start_idx + 1
        return centroids, cluster_borders

    # Now we know that there are at least 3 elements

    # If the sum of the sample weight in the range is 0, we assume that the data
         is unweighted
    if query_prefix_sum(weights_prefix_sum, start_idx, stop_idx) == 0:
        # We need to recalculate the prefix sum, as previously it would have
            been all zeros
        X_casted = sorted_X.astype(PREFIX_SUM_DTYPE)
        X_prefix_sum = np.cumsum(X_casted)
        return numba_kmeans_1d_two_cluster_unweighted(sorted_X, X_prefix_sum,
            start_idx, stop_idx)
    else:
        # Check if there is only one nonzero sample weight
        total_weight = query_prefix_sum(weights_prefix_sum, start_idx, stop_idx)
        sample_weight_prefix_sum_within_range = weights_prefix_sum[start_idx:
            stop_idx]
        final_increase_idx = np.searchsorted(
            sample_weight_prefix_sum_within_range,
            sample_weight_prefix_sum_within_range[-1]
        )
        final_increase_amount = query_prefix_sum(weights_prefix_sum,
                                                  start_idx + final_increase_idx,
                                                  start_idx + final_increase_idx
                                                      + 1)
        if total_weight == final_increase_amount:
            # If there is only one nonzero sample weight, we need to return the
                corresponding weight as the centroid
            # and set all elements to the left cluster
            nonzero_weight_index = start_idx + final_increase_idx
            centroids[0], centroids[1] = sorted_X[nonzero_weight_index],
                sorted_X[nonzero_weight_index]
            cluster_borders[1] = stop_idx
            return centroids, cluster_borders

    # Now we know that there are at least 3 elements and at least 2 nonzero
        weights

    # KMeans with 2 clusters on 1D data is equivalent to finding a division
        point.
    # The division point can be found by doing a binary search on the prefix sum
        .

```

```python
    # We will do a search for the division point,
    # where we search for the optimum number of elements in the first cluster
    # We don't want empty clusters, so we set the floor and ceiling to start_idx + 1 and stop_idx - 1
    floor = start_idx + 1
    ceiling = stop_idx - 1
    left_centroid = None
    right_centroid = None

    while floor < ceiling:
        division_point = (floor + ceiling) // 2
        # If the left cluster has no weight, we need to move the floor up
        left_weight_sum = query_prefix_sum(weights_prefix_sum, start_idx, division_point)
        if left_weight_sum == 0:
            floor = division_point + 1
            continue
        right_weight_sum = query_prefix_sum(weights_prefix_sum, division_point, stop_idx)
        # If the right cluster has no weight, we need to move the ceiling down
        if right_weight_sum == 0:
            ceiling = division_point - 1
            continue

        left_centroid = query_prefix_sum(weighted_X_prefix_sum, start_idx, division_point) / left_weight_sum
        right_centroid = query_prefix_sum(weighted_X_prefix_sum, division_point, stop_idx) / right_weight_sum

        new_division_point_value = (left_centroid + right_centroid) / 2
        if sorted_X[division_point - 1] <= new_division_point_value:
            if new_division_point_value <= sorted_X[division_point]:
                # The new division point matches the previous one, so we can stop
                break
            else:
                floor = division_point + 1
        else:
            ceiling = division_point - 1

    # recalculate division point based on final floor and ceiling
    division_point = (floor + ceiling) // 2

    # initialize variables in case the loop above does not run through
    if left_centroid is None:
        left_centroid = (query_prefix_sum(weighted_X_prefix_sum, start_idx, division_point) /
                         query_prefix_sum(weights_prefix_sum, start_idx, division_point))
    if right_centroid is None:
        right_centroid = (query_prefix_sum(weighted_X_prefix_sum, division_point, stop_idx) /
                          query_prefix_sum(weights_prefix_sum, division_point, stop_idx))

    # avoid using lists to allow numba.njit
    centroids[0] = left_centroid
    centroids[1] = right_centroid

    cluster_borders[1] = division_point
    return centroids, cluster_borders
```

C. Library Implementation

The algorithms detailed in this thesis have been published open-source as `flash1dkmeans`.

Github respository: `https://github.com/SyphonArch/flash1dkmeans`

PyPI: `https://pypi.org/project/flash1dkmeans/`

국문초록

클러스터링은 머신러닝에서 핵심적인 과제로, k-means는 단순성과 효율성 덕분에 널리 사용되는 알고리즘이다. 1차원(1D) 클러스터링은 많은 실제 응용에서 발생하지만, 기존 k-means 구현체들은 1D 데이터의 구조를 효과적으로 활용하지 못해 비효율이 존재한다. 본 논문에서는 정렬된 데이터, 누적합 배열, 이진탐색의 특성을 활용하여 1D 클러스터링에 최적화된 k-means++ 초기화 및 Lloyd 알고리즘을 제안한다.

본 논문은 다음과 같은 로그 시간 알고리즘을 제시한다: (1) k-cluster 알고리즘은 greedy k-means++ 초기화에서 $O(l \cdot k^2 \cdot \log n)$ 시간복잡도, Lloyd 알고리즘에서 $O(i \cdot k \cdot \log n)$ 시간복잡도를 달성한다. 여기서 n은 데이터셋 크기, k는 클러스터 개수, l은 greedy k-means++ local trials 수, i는 Lloyd 알고리즘 반복 횟수를 나타낸다. (2) 2-cluster 알고리즘은 이진탐색을 활용하여 $O(\log n)$ 시간복잡도로 작동하며, 반복 없이 Lloyd 알고리즘의 국소 최적해에 빠르게 수렴한다.

벤치마크 결과, 제시된 알고리즘은 대규모 데이터셋에서 `scikit-learn` 대비 4500배 이상의 속도 향상을 달성하면서도 within-cluster sum of squares (WCSS) 품질을 유지한다. 또한, 대규모 언어 모델(LLMs) 양자화와 같은 최신 응용에서도 300배 이상의 속도 향상을 보여준다.

본 연구는 1D k-means clustering의 이론과 실제 간 간극을 좁히는 효율적이고 실용적인 알고리즘을 제안한다. 제시된 알고리즘은 JIT 컴파일을 통해 최적화된 오픈소스 Python 라이브러리로 구현되었으며, 실제 응용에 쉽게 통합할 수 있도록 설계되었다.

주요어: k-means 클러스터링, Lloyd 알고리즘, k-means++ 초기화, 일차원 클러스터링, 이진탐색, 누적합