Fast Coreset Generation for K-Means

Andrew Draganov

November 25, 2022

1 Introduction

As the reliance on large datasets grows, the need for effective compression algorithms grows as well. To this end, *coresets* have become popular as a method to reduce the number of points in a dataset while preserving all of its characteristics. Specifically, given a pointset $P \in \mathbb{R}^{n \times D}$ a solution space $\mathcal{C}$, and a loss function $\mathcal{L}(P, \mathcal{C})$, a coreset Q of $S \ll n$ points has the property that $\mathcal{L}(Q, \mathcal{C}_i) \in (1 \pm \epsilon)\mathcal{L}(P, \mathcal{C}_i)$, where $\mathcal{C}_i$ is any point in the solution space. Importantly, for many problems the coreset size S does not depend on the number of input points n.

Since one of the purposes of coresets is to accelerate downstream tasks, one also requires that the coreset is constructed as efficiently as possible. In this paper we study acceleration techniques for k-means coresets. Recent work has shown that these can be obtained in $O(nd + nk)$ time, where d is the dimensionality after a Johnson-Lindenstrauss (JL) transform. Their algorithm consists of three main steps on the JL transformed input.

First, one produces an $O(1)$ approximation of the solution in $\tilde{O}(nd)$ time. Second, this approximation is used to upper-bound the sensitivity values $\sigma_p = \max_{C \in \mathcal{C}} \frac{\text{cost}(p,C)}{\text{cost}(P,C)}$. These can be interpreted as the worst-case impact that a point can have on a solution. The last step, then, is to sample S points proportional to their upper-bounded sensitivities, where $S = \tilde{O}(k^2 d)$ guarantees the coreset property.

The computational bottleneck here is the second step – estimating the sensitivities. It is not known how to compute them exactly, so the available solution is to obtain an upper-bound using all $n \times k$ point-center assignments in the approximate solution. Importantly, the assignments are sufficient for obtaining a bound on the sensitivities in no additional time.

The key insight, then, is that the point-center assignments can be obtained in faster than $O(nk)$ time. Specifically, one can use Hierarchically-Separated Trees (HSTs) to simplify the approximation. HSTs impose a distance metric d_{tree} such that $d(x, y) \leq d_{tree}(x, y) \leq \alpha d(x, y)$, where $\alpha \in O(d^2)$ for the k-means problem. This requires $\tilde{O}(nd)$ time, which is an unavoidable expense during coreset construction as it is the time needed to read the dataset. Importantly, however, we show that one can obtain a perfect k-means clustering on the tree in $O(n)$ time. This means that the cluster-assignment burden has been reduced from $O(nk)$ to $\tilde{O}(nd)$ time at the expense of an $\alpha \in O(\log n)$ distortion.

The remaining issue is that the size of our coreset, S, now contains an extra dependency on $\alpha \in O(\log d^2)$. To this end, we point out that we can iterate on this coreset by producing a new coreset $\bar{Q}$ for it. Importantly, $\bar{Q}$'s construction will *only* depend on the parameters d, k, and ε, as $S = |Q|$ does not depend on n. Thus, the final coreset can be constructed from Q with no time dependency on n and is absorbed into the existing complexity of obtaining Q.

2 Related Work

3 Algorithm Description

Before discussing our algorithm for fast coreset generation, we first explain the existing method and where it can be improved upon.

3.1 $\tilde{O}(nD + nk)$ time coreset generation

The existing state of the art algorithm for generating a coreset consists of three steps:

1. Obtain an $O(1)$ approximation of the optimal loss in $O(nDk)$ time. This can be done, for example, by utilizing K-Means++ with $l = 2k$ centers (ref needed).

2. Use this approximation to calculate an upper bound on the sensitivities by $\sigma_p \leq \hat{\sigma}_p = \frac{\text{cost}(p,C_i)}{\sum_{q \in C_i} \text{cost}(q,C_i)} + \frac{1}{|C_i|}$.

3. Sample $S = O(\frac{k^2 d}{\varepsilon} \log^2 \frac{kd}{\delta}$ points where each point p_i has likelihood $\mathbb{P}[p_i] = \frac{\hat{\sigma}_i}{\sum_j \hat{\sigma}_j}$.

The above process guarantees a Euclidean k-means coreset with probability at least $1 - \delta$.

There are two bottlenecks here. First, we require a solution that provides an $O(1)$ approximation of the cost. Second, we must obtain all $n \times k$ point-center assignments so that we can upper-bound the sensitivities. Notice, however, that these are not equivalent. Let $\mathcal{C} = \{c_1, \ldots, c_k\}$ be an O(1) approximation obtained by some algorithm $\mathcal{A}$. If $\mathcal{A}$ did not explicitly store the cost of every point with respect to its closest center, then we would need to still perform all $n \times k$ point-to-center comparisons to find the cluster assignments. On the other hand, consider that we have upper-bounded the sensitivities $\hat{\sigma}_i \forall i \in P$ using a α-approximate clustering. Then the approximation of the solution is inherent to solving the problem.

An important note is that the number of sampled points S is directly proportional to the approximation quality. That is, if we have an (α, β)-approximation of the optimal cost, then we would need $S' = O((\alpha+\beta)m)$ samples to guarantee a coreset (ref needed).

Let us consider the first easy step to accelerating this algorithm – the JL transform. It was shown in [**makarychev2019performance**] that a JL transform to $d = O(\log(k/\epsilon)/\epsilon^2)$ dimensions will preserve the optimal solution to a factor of $(1 + \epsilon)$. Assume that our dataset has been transformed from $\mathbb{R}^{n \times D}$ to $\mathbb{R}^{n \times d}$ in $O(D \log(D) + \epsilon^{-3} \log^2(n))$ time using a fast JL transform [**thickstunfast**]. Then computing the $O(1)$ approximation in step 1 will require $O(ndk) = O(nk \log(k/\epsilon)/\epsilon^2) = \tilde{O}(nk)$ time.

We now use this to upper bound the sensitivities, which requires us to assign n points to $O(k)$ clusters. Each point-to-center comparison must sum over the reduced dimensionality d, so assigning a single point to a cluster requires $O(\log(k/\epsilon)/\epsilon^2)$ operations. Thus, our sensitivity bounds are obtained in $O(nkd) = \tilde{O}(nk)$ time.

3.2 Accelerating the sensitivity sampling

It is unreasonable to expect that we can remove the $O(nD)$ complexity. For example, let $(n \times D) - 1$ entries of P be in the range $(0, 1)$ and let the remaining entry have value 1000. Then a coreset must capture the single extreme element, implying an $O(nD)$ lower-bound on the runtime.

What is possible, however, is accelerating the $\tilde{O}(nk)$ runtime for bounding the sensitivities. The bottleneck is in obtaining the assignments with respect to our approximate solution, requiring $O(nk)$ point-to-center comparisons. Intuitively, this brute-force method contains unnecessary operations. For example, if centers i and j are close to one another and point p is far from center i, then we can reasonably conclude that it is also far from center j.

Thus, we want a method that can obtain cluster assignments in faster than $\tilde{O}(nk)$ time while preserving a reasonable approximation guarantee so that our coreset remains small. To see this, let us assume that the method for accelerating the sensitivity bounding induces an (α, β)-error on the quality of the approximation. Then, as stated, we would need $m' = O((\alpha+\beta)m)$ samples to satisfy the coreset guarantee. This implies that our acceleration's additive or multiplicative error must be less than $O(log(n))$, or else we risk transferring the computational burden from bounding the sensitivities to sampling $O(nm)$ points.

One natural method for doing this is to use tree embeddings. A pointset in a metric space can be embedded in a weighted *hierarchically well-separated tree* (HST) space such that the distance $d(x, y) \leq d_{tree}(x, y) \leq O(\log(n))d(x, y)$, where $d_{tree}(x, y)$ is the length of the shortest path between leaves x and y in the tree. Embedding our dataset into an HST provides several key benefits. Second, it only takes $\tilde{O}(nd)$ time to produce an HST embedding, which is already the time that we need to read the dataset.

3.3 k-median clustering in $O(n)$ time on a 2-RHST

It appears to be possible to get an optimal k-median clustering in a 2-RHST in $O(n)$ time. The underlying reason for this is that each center c_i has one closest center c_j and, for all leaves $v \in c_i$, v's second closest center is c_j. Thus, we can recursively find the k-median clustering quickly since the merges only depend on positions of the centers, not the points associated to them.

The time complexity comes from the fact that we will recurse down until the base case of k leaves in a cell. There are $O(n/k)$ of these. We now go up the recursive stack by merging the centers of subtrees together. Since every center's closest center is either directly to its left or its right, we only need to perform $2k$ comparisons to find the k closest centers. Since we are do not need these to be sorted, finding the k smallest elements in a list takes $O(k)$ time. Since we start with $O(n/k)$ separate subtrees and each merge reduces the number of subtrees by one, we will perform $O(n/k)$ merges where each merge requires $O(k)$ time. Thus, our runtime is $O(n)$.

We now present the algorithm for clustering a 2-RHST in $O(n)$ time. We say that centers can be placed on any node in the tree, not just on leaves.

Assume that we have created a binary tree embedding where all leaves are at the same height and each edge to a node's child is half the weight of the edge to that node's parent. Assume also that during the creation of the embedding, we stored a list of nodes with fewer than k leaves. We define this list as A such that $|a| \leq k \; \forall \; a \in A$, where the cardinality of a node is the number of leaves it has under it. Call these the *base nodes*.

Furthermore, for each leaf l in base node a, we store the tree-distance to the leaf on its left and the leaf on its right in a. Thus, for every base node, we have its list of leaves as well as how far each leaf is to its left and right neighbors. These can all be obtained during the creation of the HST in no additional time. We now describe the recursive step.

We first consider the base case. If our current root node a is a base node, then we assign one center to each leaf in the base node and store the distance of each center to its left and right neighbors. If a has fewer than k leaves, then this just means there are unassigned centers. Notice that this is inherently an optimal clustering.

If we are not in the base case, we can assume that we have left and right children, each with k centers[1] Our algorithm will proceed by merging the $2k$ centers until we have k of them. Theorem 2 shows that these merges preserve optimality.

We define the merge function $M : v \times v \to v$ such that $M(c_i, c_j)$ is the node that minimizes the sum of distances to all the leaves in c_i and c_j. The merges take place in the order of smallest incurred cost over all the leaves connected to the merge. We now look through the $2k$ centers to see which ones should be merged together[2]. Importantly, the 2-RHST structure guarantees that each merge will occur between neighboring left-right centers.

Theorem 1. Let node N in a 2-RHST have left and right children with optimal k-median clusterings. Then the optimal merge is $M(c_i, c_{i+1})$ for $0 \leq i < 2k$.

Proof. For the sake of contradiction, assume that $M(c, b)$ has minimum cost where b is not c's closest center. WLOG, let $M(c, b)$ be closer to b than to c, implying that $|b| > |c|$. Lastly, let $\hat{c}$ be c's closest center.

There exists a node where the paths $c \to o$ and $c \to \hat{c}$ diverge from each other. By the 2-RHST configuration, the edge above this fork has larger weight than the entire path from the fork to any leaf below it. We know that $\hat{c}$ is below this fork.

Clearly, then, it is cheaper for c's leaves to be assigned to $M(c, \hat{c})$ than to $M(c, b)$. This reduces our cost by at least $|c| \cdot d\left(M(c, b), M(c, \hat{c})\right)$.

The remaining question is whether the added cost of connecting $\hat{c}$'s leaves to $M(c, \hat{c})$ can be paid for. If $|\hat{c}| > |c|$, then $M(c, \hat{c}) = \hat{c}$ and the added cost from $\hat{c}$'s leaves is 0. On the other hand, if $|\hat{c}| \leq |c|$, then we know that $|\hat{c}| < |b|$, since $M(c, b)$ was closer to b than to c. Then the cost of connecting $\hat{c}$'s leaves to $M(c, \hat{c})$ is equal to

[1] Assume that neither child has unassigned centers. The algorithm won't change if a child has an unassigned center, it just requires fewer merges.

[2] assuming that the previous centers were optimal for their respective subtrees

$$|\hat{c}| \cdot d(\hat{c}, M(c, \hat{c})) \leq |c| \cdot d(\hat{c}, M(c, \hat{c})) \leq |c| \cdot d\left(M(c, b), M(c, \hat{c})\right)$$

Thus, it will never be cheaper to connect c to any center other than $\hat{c}$. □

We have now shown that the optimal merge occurs among closest centers. However, it is not yet clear that performing the merges will give the optimal solution. This turns out to be the case.

Theorem 2. Let node N in a 2-RHST have left and right children with optimal k-median clusterings. Then performing k minimum-cost merges produces the optimal k-median clustering on N.

Proof. Let $\mathcal{C}_L$ and $\mathcal{C}_R$ be the optimal k-median centers on N's left and right children, respectively. Let $\mathcal{C}$ be the result of performing k merges and OPT be the optimal k centers on N. If we assume that $\mathcal{C} \neq OPT$, then there exists a leaf v whose cost under $\mathcal{C}$ is greater than its cost under OPT. WLOG, let v belong to center $\mathcal{C}_L^i$. We can assume that that $\mathcal{C}_L^i$ was involved in a merge as, if it wasn't, v would belong to its previous optimal clustering and have cost less than or equal to its cost under OPT.

Let $\mathcal{C}_{opt}$ be the center that v is assigned to in OPT. It must then be the case that $\mathcal{C}_{opt}$ is higher than $\mathcal{C}_L^i$ but lower than $M(\mathcal{C}_L^i, \mathcal{C}_L^j)$, where $\mathcal{C}_L^j$ is the center that $\mathcal{C}_L^i$ merges with. We know that $\mathcal{C}_{opt}$ must cover additional leaves compared to $\mathcal{C}_L^i$. If it didn't, it would not be optimal for it to be above $\mathcal{C}_L^i$. Let $v_j \in \mathcal{C}_{opt}; v_j \notin \mathcal{C}_L^i$ be one such leaf. We know that $v_j \notin \mathcal{C}_L^j$ since, if it were, merging $\mathcal{C}_L^i$ and $\mathcal{C}_L^j$ would be optimal. We also know that v_j's lowest common ancestor with $\mathcal{C}_L^i$ is below $\mathcal{C}_L^j$. Thus, we must conclude that v_j is closest to either $\mathcal{C}_L^i$ or $\mathcal{C}_L^j$ but did not belong to either of them in the optimal clustering, providing us a contradiction.

□

4 Questions

- In [**cohen2022scalable**], it is stated on page 10 that "two points p and q have probability $\frac{d \cdot \text{dist}(p,q)}{2^i}$ to be cut at level i. Thus, the expected distance squared between p and q is $d \cdot \text{dist}(p, q) \cdot \sum_i \sqrt{d} 2^i$ which means that the squared distance can be distorted by an arbitrarily large factor in expectation. Where does the expected distance squared come from?
- The k-median dynamic program works on the assumption that we have binary trees. This is done by splitting along one dimension at a time. Does the same process work for the k-means tree embedding without losing any quality? Seems like it should...