# Fast Coreset Generation for K-Means

Andrew Draganov

January 16, 2023

## 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 . 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, once one has the clustering matrix the sensitivities are available in no linear 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 $\varepsilon$, 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 Preliminaries

### 3.1 Sensitivity

Modern coreset construction methods rely on estimating the sensitivity $\sigma_p = \max_{C \in \mathcal{C}} \frac{\text{cost}(p,C)}{\text{cost}(P,C)}$ of each point in the dataset. Given the sensitivities, one simply samples $S$ points from $P$ proportional to $\sigma$, where $S$ depends on $k, D, \varepsilon, \delta$. Since it is not known how to obtain the sensitivities directly, it suffices to sample

according to the upper-bound $\hat{\sigma}_p = \text{Cost}(p, \mathcal{C}_i) / \left( \sum_{q \in \mathcal{C}_i} \text{Cost}(q, \mathcal{C}_i) \right) + 1/|\mathcal{C}_i|$, where $\mathcal{C}_i$ is the center that point $p$ belongs to. This requires obtaining both an approximate solution $\mathcal{C}$ and all $n \times k$ point-center assignments, imposing a bottleneck on the coreset construction time complexity.

The other consequence of the approximation is that we must oversample the coreset to account for the error that the approximation introduced. Specifically, if we have an $(\alpha, \beta)$-approximation of the optimal solution, our number of samples $S$ must scale linearly with $\alpha$ in order to satisfy the coreset requirements.

### 3.2 Tree Embeddings

Tree embeddings have proven to be very effective for accelerating clustering tasks. They rely on the premise that a space can be recursively subdivided such that, with appropriately chosen edge weights, the distance between two points in the original space is upper-bounded by their distance in the tree. In this paper we use randomly-shifted binary kd-trees. We start by shifting all points by a vector $s \in [0, \text{MaxDist}]^d$, where MaxDist is the largest difference in $P$ along any one dimension. Thus, all points in a single cell of diameter $2\sqrt{d}\text{MaxDist}$ with weight to its children equal to its diameter.

We then subdivide the points one dimension at a time such that after $d$ splits, the diameter of the cell is $\frac{1}{2}$ that of its $d$-th grandparent. This results in a tree with one point in each leaf and all leaves at equal depth. Then the distance between two points in the tree is the sum of cell diameters along the path between the two corresponding leaves. It was shown in **ref needed** that this produces a tree embedding such that $\text{dist}(x, y) \leq \mathbb{E}[\text{dist}_{tree}(x, y)] \leq 24 \log(n) \text{dist}(x, y)$, implying a distortion of at most $O(log(n))$ on the original distances.

### 3.3 Tree Clustering

One benefit of working with trees is that they admit an optimal $k$-means and $k$-median clustering on the tree metric. This is achieved through a dynamic program that maintains the solutions of each cell's subtrees, providing an inductive proof that optimality is preserved under appropriate merges. Although tree clustering requires $O(ndk^2)$ time, it can be parallelized to run in $O(n \log n)$ time by exploiting the fact that the output of a cell depends on the outputs of its children. Thus, one can obtain a $k$-median solution by embedding $P$ into an HST and obtaining the optimal clustering on the tree metric, incurring $O(log(n))$ error on the optimal $k$-median solution for $P$.

This proves trickier for $k$-means as the HST distortion bound is *not* guaranteed on the squared distances, implying that the incurred error does not scale with the distance distortion. To this end, **ref needed** showed that one can instead fit 3 HST's and compare the squared Euclidean distances to the squared minimum distance over the trees. Under this setting, one has that $\text{dist}^2(x, y) \leq \text{min\_tree\_dist}^2(x, y) \leq d^2 \text{dist}^2(x, y)$.

## 4 Algorithm Description

### 4.1 $\tilde{O}(nD + nk)$ time coreset generation

Before discussing our algorithm for fast coreset generation, we first explain where the existing method, Algorithm 1, can be improved upon.

There are two bottlenecks in Algorithm 1. 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. We point out, however, that these are not equivalent tasks. To see this, let $\mathcal{C} = \{c_1, \ldots, c_k\}$ be an O(1) approximation obtained by $k$-means++. Since $k$-means++ does not explicitly store the cost of every point with respect to its closest center, would need to still perform all $n \times k$ point-to-center comparisons to find the cluster assignments. On the other hand, if we have all the assignments then we can quickly get the cost of all the centers (and the sensitivities) in $O(n)$ time. Thus, accelerating algorithm 1 can be accomplished by obtaining the assignments in faster than $O(nk)$ time.

**Algorithm 1** $\tilde{O}(nd + nk)$-time coreset construction

**Input**
$P$ : Pointset to be compressed.
$k$ : Number of centers for downstream clustering.
$\varepsilon$, $\delta$ : Approximation constants.
**Output**
$Q$ : Coreset for $P$ with $|Q| \ll |P|$

**function** Approx Sensitivities($P, \mathcal{C}$)
  $\hat{\sigma} \leftarrow [\,]$
  **for** $0 \leq i < |P|$ **do**
    $\mathcal{C}_i \leftarrow$ Center that $P[i]$ belongs to ▷ Takes $O(kd)$ time if not precomputed
    $\hat{\sigma}[i] \leftarrow \frac{\text{Cost}(P[i], \mathcal{C}_i)}{\sum_{q \in \mathcal{C}_i} \text{Cost}(q, \mathcal{C}_i)} + \frac{1}{|\mathcal{C}_i|}$
  **end for**
  **return** $\hat{\sigma}$
**end function**

**function** Construct Optimal Coreset($P, k, \varepsilon, \delta$)
  $P \leftarrow$ JLTransform($P, O(\log k)$)
  $\mathcal{C} \leftarrow O(1)$-approximation of OPT on $P$
  $\hat{\sigma} \leftarrow$ Approx Sensitivities($P, \mathcal{C}$)
  $Q \leftarrow$ Sample Points($P$, weights $= \hat{\sigma}$, NumSamples $= \frac{dk^2}{\varepsilon} \log^2 \frac{kd}{\delta}$)
  **return** $Q$
**end function**

## 4.2 Identifying clusters quickly

Our goal, then, is to find an approximation algorithm that will *simultaneously* identify the cluster centers and the cluster assignments in faster than $O(nk)$ time. On an intuitive level, this means that we want to exploit the following likely condition: if point $p$ is far from center $i$ and center $i$ is close to center $j$, then we can reasonably conclude that $p$ is also far from center $j$. We point out, however, that a generalization of this is exactly the reasoning underpinning tree-embeddings – distances are bounded by the dimensions of the subspaces points belongs to.

Furthermore, HSTs are easy to work with as they can be constructed quickly in $O(nd \log n \log(d\Delta))$[1], where $\Delta$ is the maximum ratio between distances in $P$, and can be clustered similarly quickly in $O(n \log n)$ time [2].

Given this context, our first algorithm can be found in Algorithm 2. We first project the data down to $O(\log k)$ dimensions using a JL transform. Next, we embed the dataset into an HST and find the cluster centers on the tree. Due to the tree's structure, knowing the centers leads to having the point-cluster assignments in linear time, as we can traverse the tree from left to right and identify each point with one of its neighboring centers. We can now obtain a coreset $Q$ by sampling $P$ according to the sensitivities bounded by these clusters. However, since our centers are based on an $O(\log n)$ distortion of the distances, we must oversample by the errors incurred during the approximations in order to satisfy the coreset property.

## 4.3 Obtaining the optimal coreset

Although the algorithm described above produces a coreset $Q$ in near-linear time, it contains a factor of $\log(n)$ unnecessary points in order to account for the approximation errors. However, note that we now have a dataset $Q$ whose size does not depend at all on $n$. Thus, we are now free to utilize $Q$ to obtain an optimal coreset in any manner we choose, as we can be confident that this will take negligible time compared to the

[1] I think?...
[2] Again, I'm not sure on the exact bound here

complexity of producing $Q$ in the first place. If it turns out that $|Q|$ is still too big, our above fast HST algorithm can be repeated multiple times, where $i$ iterations will produce a coreset with $\log(\dots\log(n)\dots)$ unnecessary points, where the expression has $i$ nested logarithms. Putting all of this together, we present our full algorithm in 3.

**Algorithm 2** $\tilde{O}(nd)$-time coreset construction

**class** HST
    **variables** $\leftarrow P$
**end class**

**function** FIT(root, split_dim $= 0, \Delta = 0,$ split_location $= [0, \dots, 0]$)
    **if** len(root.$P$) $\leq 1$ **then**
        **return**
    **end if**
    $\mathcal{I}_l \leftarrow (node.P[:, \text{split\_dim}] \leq (\text{split\_location}[\text{split\_dim}] + \Delta))$
    $S_l \leftarrow \text{root}.P[\mathcal{I}_l]$
    $S_r \leftarrow \text{root}.P[\neg\, \mathcal{I}_l]$

    // Add 1 to dim or wrap around to 0. Divide $\Delta$ by 2 when wrap around.
    split_dim, $\Delta \leftarrow$ NextSplit(split_dim, $\Delta$)
    FIT(HST($S_l$, split_dim, split_location, $\Delta$)
    split_location[split_dim] $+ = \Delta$
    FIT(HST($S_r$, split_dim, split_location, $\Delta$)
**end function**

**function** CLUSTER HST(root, $k$)
    Do $O(n \log n)$-time optimal clustering on HST
**end function**

**function** CONSTRUCT FAST CORESET($P, k, \varepsilon, \delta$)
    $P \leftarrow$ JL Transform($P, O(\log k)$)
    root $\leftarrow$ HST($P$)
    FIT(root)
    $\tilde{\mathcal{C}} \leftarrow$ Cluster HST(root, $k$)
    $\hat{\sigma} \leftarrow$ Approx Sensitivities($P, \tilde{\mathcal{C}}$)
    $Q \leftarrow$ Sample Points($P$, weights $= \hat{\sigma}$, NumSamples $= \frac{dk^2}{\varepsilon} \log^2 \frac{kd}{\delta}$)
    **return** $Q$
**end function**

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

**Algorithm 3** $\tilde{O}(nd)$-time optimal coreset construction

**function** CONSTRUCT FAST OPTIMAL CORESET($P, k, S, \varepsilon, \delta$)
    $\tilde{Q} \leftarrow P$
    **while** $|\tilde{Q}| > S$ **do**
        $\tilde{Q} \leftarrow$ Construct Fast Coreset($\tilde{Q}, k, \varepsilon, \delta$)
    **end while**
    $Q \leftarrow$ Construct Optimal Coreset($\tilde{Q}, k, \varepsilon, \delta$)
    **return** $Q$
**end function**

**Algorithm 4** $O(n)$-time algorithm for $k$-median in a 2-RHST

**Input**
Root : Top-level node of tree.
Dists : Array of $n-1$ distances from leaf $i$ to leaf $i+1$. Obtained during HST creation.

**function** 2-RHST $k$-MEDIAN(Root, Dists, $k$)
*Base Case*:
// Make a center for each leaf
**if** Root **is** BaseNode **then** ▷ A BaseNode is any node with $< k$ leaves
$\mathcal{C} \leftarrow$ Array( Center$(l, [l], l, l)$ **for** $l$ **in** Root.leaves )
**return** $\mathcal{C}$
**end if**
*Recursive Call*:
// Combine two sets of centers together
$\mathcal{C}_L \leftarrow$ 2-RHST $k$-Median(Root.LeftChild, Dists, $k$)
$\mathcal{C}_R \leftarrow$ 2-RHST $k$-Median(Root.RightChild, Dists, $k$)
MinCenterDists $\leftarrow$ FindMinDists($\mathcal{C}_L$, $\mathcal{C}_R$, $k$)
$\mathcal{C} \leftarrow$ Array( )
**for** $0 \leq i < k$ **do**
MinDist $\leftarrow$ MinCenterDists[$i$]
$\mathcal{C}[i] \leftarrow$ Merge(MinDist.LeftCenter, MinDist.RightCenter)
**end for**
**return** $\mathcal{C}$
**end function**

**class** CENTER
**variables** $\leftarrow$ Loc, PointList, RightMostPoint
**end class**

**function** MERGE($c_1$, $c_2$)
Points $\leftarrow c_1$.PointList $\cup$ $c_2$.PointList
**if** $c_1$.size $\neq$ $c_2$.size **then**
Loc $\leftarrow$ (argmax($|c_1|, |c_2|$)).Loc ▷ Center will be the center of the larger one
**return** Center(Loc, Points, $c_2$.RightMostPoint)
**end if**
**return** Center($c_1$.Loc, Points, $c_2$.RightMostPoint) ▷ WLOG
**end function**

**function** FINDMINDISTS($\mathcal{C}_L$, $\mathcal{C}_R$, $k$, Dists)
// Get $k$ smallest weighted distances between centers
$\mathcal{C} \leftarrow \mathcal{C}_L \cup \mathcal{C}_R$
WeightedDists $\leftarrow$ Array( )
**for** $0 \leq i < 2k-1$ **do**
D $\leftarrow$ Dists[$\mathcal{C}_i$.RightMostPoint] ▷ "Distance from this leaf to the one on its right"
Size $\leftarrow \min(|\mathcal{C}_i|, |\mathcal{C}_{i+1}|)$ ▷ "How many points will pay the cost"
WeightedDists[$i$] $\leftarrow$ D $\times$ Size
**end for**
MinCenterDists $\leftarrow$ minimum $k$ values from WeightedDists (unsorted) ▷ Requires $O(k)$ time
**return** MinCenterDists
**end function**