Coresets for Streaming and Clustering
CS 543 - Algorithms for Big Data Project

Andrew Roberts, Bhushan Suwal, Karan Vombatkere

December 10, 2021

1 Overview

This project is an open-source Python implementation of various Coreset algorithms for clustering and streaming. All Python code, data and relevant notebooks/output is available in the GitHub repository below.

GitHub Repository link: https://github.com/kvombatkere/CoreSets-Algorithms

Note that the project presentation has been appended to this report for reference.

1.1 Coresets

Given a set of input points P, a *Coreset* is a weighted subset:

$$(C, w), C \subset P, w : C \rightarrow [0, \infty)$$

such that cost(C, w) well-approximates cost(P), where *cost* is the cost function for some problem of interest. Coresets are much smaller than the input (typically poly-logarithmic). Coresets have wide range of applications such as streaming processing, dimensionality and data reduction, distributed processing.

1.2 Implementation Notes

All algorithms use the following standard Python libraries: *numpy, pandas, matplotlib.pyplot, time*. The code contains further detailed documentation about parameters and values relevant to the algorithms. *helper_functions.py* and *coreset_util.py* contain some common functions used across the other modules. The data folder contains 2 datasets: UN CO2 Emissions and Finland 2012 Locations data.

2 Coreset Algorithms Implemented

2.1 Median Estimation

Given a sequence of sorted numbers $x_1, \ldots, x_n$ the median value is the 'middle' element, which for odd n is given by $x_{(n+1)/2}$, and for even n is given by $(x_{(n/2)} + x_{(n/2)+1})/2$. The best known algorithm for finding the median takes $O(n)$ runtime.

The following implementation of the median approximation algorithm computes a ϵn approximate median in near-constant runtime. We first partition the sequence of numbers into $O(1/\epsilon)$ subsequences, and then estimate the median by computing a coreset on each subsequence by taking the minimum value of the subsequence. Then we merge all coresets together and estimate the true median by returning the median value of the coresets of each subsequence.

The implemntation was tested on synthetic data with samples drawn randomly from $\Gamma(k, \theta) = \Gamma(5, 50)$. The Γ distribution has no known closed-form expression for the median and we compared the runtime of our estimation algorithm with *numpy.median()* for different values of ϵ. The figure below shows plots of the median estimation and runtime for different values of n (on a logarithmic scale). We observe that the runtime *numpy.median()* has a linear relation with the size of n (as expected) but our median estimation algorithm runs in near constant time for large n tested up to 10^8. We also observed that our algorithm provided a good approximation to the true median, within the ϵn error bound.

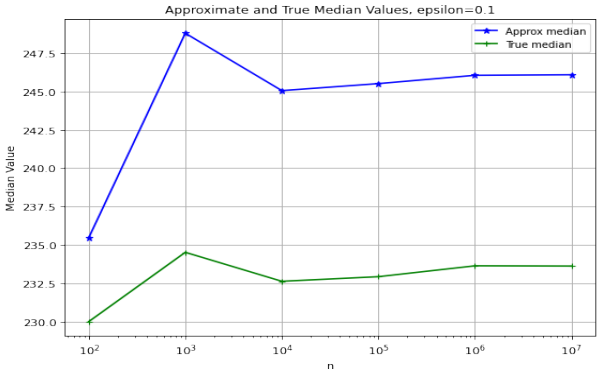

Figure 1: Median Estimation comparison with numpy.median

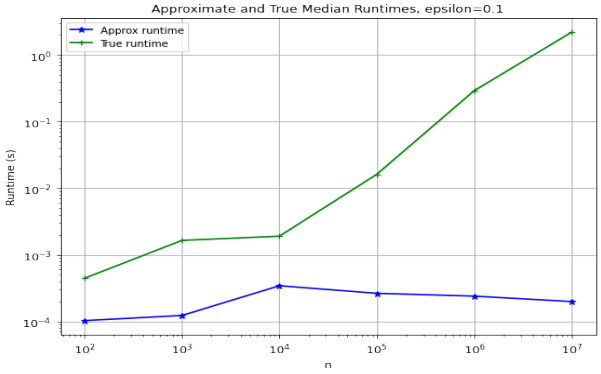

Figure 2: Median Estimation runtime comparison with numpy.median

2.2 Minimum Enclosing Ball (MEB)

Given a set of points P in $\mathbb{R}^d$, the Minimum Enclosing Ball problem consists of finding the smallest ball that encloses the points in P. Formally the task is to find the minimum of a cost function C_P, where:

$$C_P(x) = \max_{y \in P} ||x - y||_2$$

i.e., the radius of the smallest ball centered at x that encloses all point in P.

The implementation of our algorithm involved computing a θ-grid consisting of $O(1/\theta^{(d-1)})$ vectors, and then computing a $(1+\epsilon)$-coreset of size $O(1/\epsilon^{(d-1)/2})$, by selecting an extreme point along each direction of the θ-grid.

The implementation was tested on synthetic 2D data with $P \subset \mathbb{R}^2$ with samples drawn randomly from uniform and Gaussian distributions, and the 2D plot below shows a sample Minimum Enclosing Ball for a multivariate Gaussian distribution.

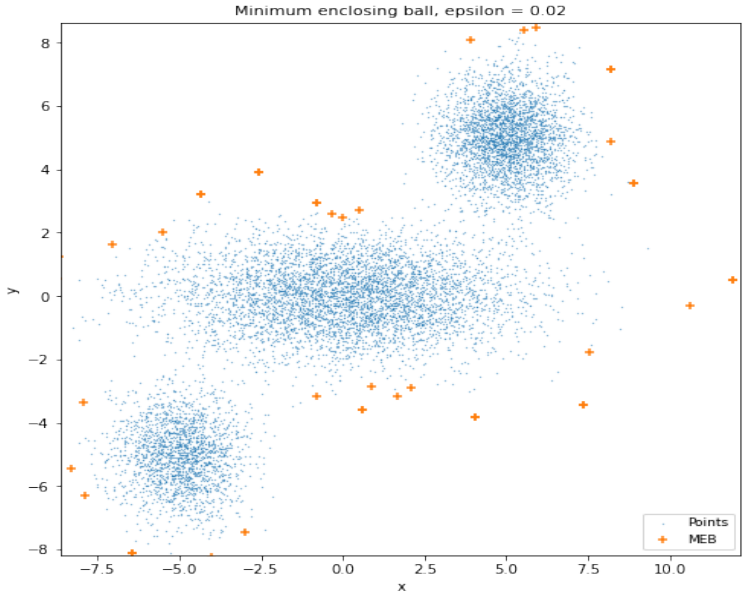

Figure 3: Minimum Enclosing Ball for a Multivariate Gaussian

2.3 k-center Clustering

Given a set $P = \{P_1, \ldots, P_n\}$ of n of points in $\mathbb{R}^d$, and an integer $k > 0$, the goal is to partition P into k disjoint subsets, $S_1, \ldots, S_k$ such that a cost function μ that measures the extent of a cluster is minimized.

$$\text{k-center objective} = \max_{1 \le i \le k} \mu(P_i)$$

The implementation of the k-center clustering algorithm first computes a greedy k-center clustering for the input set of points, $P \subset \mathbb{R}^d$. This greedy clustering is a 2-approximation for the optimal clustering. The cost of this clustering, r is used to then compute a grid of side length $\epsilon r/(5d)$. Then, all points in the input are snapped onto the grid, and a single representative point from each grid cell is selected to be a part of the ϵ additive coreset, which is returned by the algorithm.

The k-center clustering can then be computed using this coreset as an input. The implementation includes code to plot the clustering in 2D, and also compute the error distance between the original clustering and the coreset clustering.

The implementation was tested on both synthetic and real data. The following 2D plots show the coreset generation on 2012 GPS location data from Finland ($n = 13466$). For $\epsilon = 0.1$, the average clustering error over 50 iterations for the k-center algorithm on all data points was 3.29, and the error on the average clustering error on the coreset was 3.36.

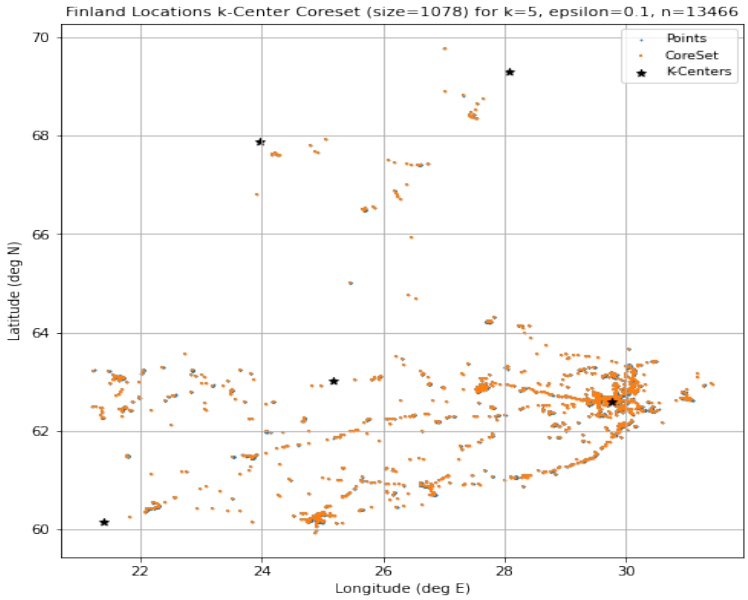

Figure 4: Finland 2012 GPS user locations Coreset points and original k-center clustering

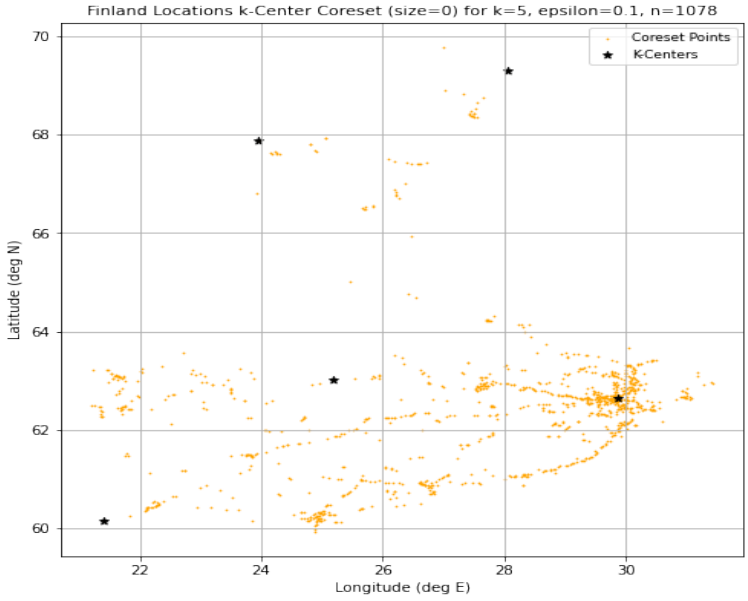

Figure 5: Finland 2012 GPS user locations Coreset k-center clustering

2.4 Gaussian Mixture Models (GMM)

2.4.1 Problem Setup

Assume that we observe input data $x_1, \ldots, x_N \subset \mathbb{R}^d$. The standard Gaussian Mixture Model (GMM) assumes that the x_i are independent and distributed according to a weighted average of k multivariate normal (MVN) distributions, with k fixed in advance. Specifically, we assume:

$$p(x_i|\theta) = \sum_{j=1}^{k} w_j \mathcal{N}(x_i|\mu_j, \Sigma_j), \;\; \theta = (w_1, \mu_1, \Sigma_1, \ldots, w_k, \mu_k, \Sigma_k)$$

where the *mixture weights* w_i satisfy $w_i \in [0, 1]$ and $\sum_{j=1}^{k} w_j = 1$. $\mathcal{N}(\cdot, \mu_j, \Sigma_j)$ is used to denote the MVN density with mean vector μ_j and covariance matrix Σ_j. The task in this model is to estimate the k mixture weights, means, and covariance matrices; that is, to estimate the parameter vector θ. The most common approach is to estimate θ via Maximum Likelihood Estimation (MLE), where the maximization of the log-likelihood is accomplished via the Expectation-Maximization (EM) algorithm.

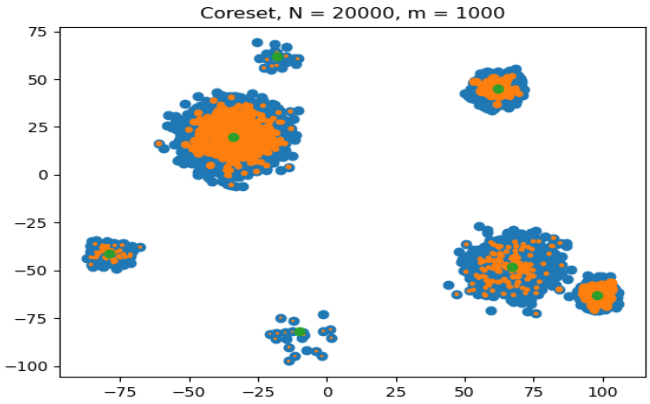

Figure 6: Simulated points from a GMM with k = 7 are in blue, with the sampled coreset points in orange sized according to the coreset weights. The true Gaussian means are in green.

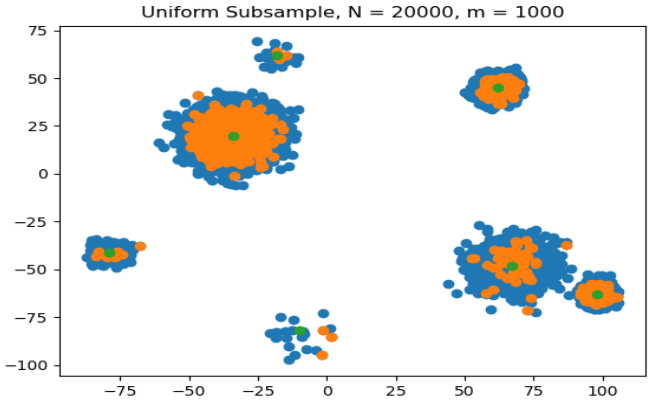

Figure 7: Similar to above, but the orange points were produced via uniform subsampling.

2.4.2 Coresets for GMM

Let $X = (x_1, \ldots, x_N)'$ denote the input data. The ultimate goal of the EM procedure is to approximate the maximum of the log-likelihood

$$\mathcal{L}(\theta; X) = \sum_{i=1}^{N} \log p(x_i|\theta)$$

To define a coreset (C, γ) for this problem it is thus natural to define the *weighted log-likelihood*

$$\mathcal{L}(\theta; C, \gamma) = \sum_{i=1}^{M} \gamma_i \log p(c_i|\theta), \;\; M := |C|$$

and require that $\mathcal{L}(\theta; C, \gamma) \approx \mathcal{L}(\theta; X)$ with high probability. We seek an algorithm that constructs such a coreset with $M << N$.

2.4.3 Coreset Construction Algorithm

This section provides a brief summary of the GMM coreset construction algorithm detailed in Lucic et al (2018). To motivate the algorithm, let us first consider a naive alternative: uniformly subsample from X to construct a smaller dataset and then run the EM algorithm on the subsample. This approach is problematic in that it can produce arbitrarily bad approximations to the log-likelihood; indeed, for imbalanced data small clusters may be under-sampled and thus a uniform subsample may erase important structures present in the original data. The coreset algorithm directly addresses this concern by biasing the sampling towards less dense regions of the input space. A brief outline of the algorithm steps are provided below.

1. Produce an approximate k-means solution via the k-means++ algorithm. Intuitively, this produces a rough approximation to the k Gaussian means. These k centers partition the dataset into k clusters, where each point is assigned to the nearest center.

2. Use these clusters to define sampling probabilities $q(x_1), \ldots, q(x_N)$. In particular, a point is more likely to be selected if it is far from a cluster center, in a cluster with high within-cluster variance, and a member of a small cluster. Thus, these sampling probabilities bias the sampling towards regions that would be most likely neglected by a uniform sampling scheme.

3. Sample M points $x_{i_1}, \ldots, x_{i_M}$ from X according to the probabilities $q(x_1), \ldots, q(x_N)$. Define the coreset weight for each point proportional to the inverse of its sampling weight.

The results of this coreset construction algorithm are presented in figure 6, and contrasted with the uniform subsampling approach shown in 7. We see that the former approach "reaches" into the less dense regions of the input space, sampling points in small clusters that are largely ignored by the uniform subsampling scheme.

2.4.4 Fitting GMM Model on the Coreset

Once a GMM coreset (C, γ) is constructed via the above algorithm, we seek to approximate θ by running the EM algorithm on this weighted subset of the data. Although the standard EM algorithm does not handle weighed input points, it can easily be generalized to the weighted case. Indeed, analogous update rules can be derived that take the point weights into account and the

modified algorithm can be shown to converge to a local maximum of the log-likelihood function (Lucic et al, 2018).

In general, the performance of this method varies with the characteristics of the input data. Lucic et al (2018) show promising theoretical and experimental results in various settings, with the coreset results consistently outperforming uniform subsampling. We present a specific example in figures 8 and 9. In these figures, we use the parameter estimates $\hat{\theta}_{\text{coreset}}$ and $\hat{\theta}_{\text{uniform}}$ to simulate GMM data. Good parameter estimates will produce data that closely resembles the original data (blue data points in figures 6 and 7). This highlights some of the benefits and pitfalls of the coreset approach. The coreset, as expected, provides a superior reconstruction of the small cluster at the bottom of the plot. However, the biased sampling scheme produces an erroneous cluster in the top left-hand corner, whereas the uniform subsample does not make this mistake. We emphasize that this is a specific example; the coreset guarantees are probablistic in nature and have superior performance on average. Nevertheless, this example illustrates some of the difficulties that may arise in practice.

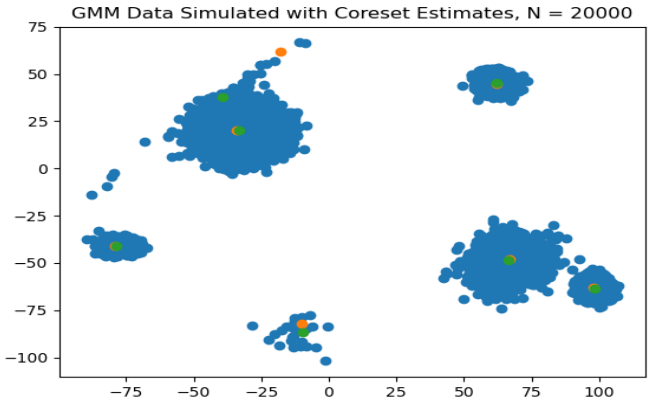

Figure 8: Estimate $\hat{\theta}$ of θ produced using the coreset. The blue data points are simulated using $\hat{\theta}$. Estimated means are given in green, and true means in orange. The smaller cluster mimics the original data, but the coreset makes a mistake identifying the top-left cluster.

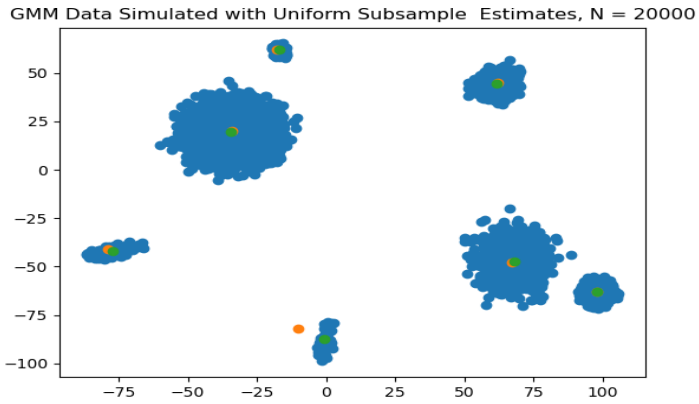

Figure 9: Analogous to above, but blue points are generated via $\hat{\theta}$ estimated on a uniform subsample. The uniform subsample does a poor job simulating the smaller cluster at the bottom, but correctly identifies the top-left cluster.

2.5 Streaming k-means/k-median

We implemented streaming algorithms from Har-Peled and Mazumdar (2004) for k-means and k-medians. This setting is motivated by the practical consideration where we might know what size of the coreset we would want beforehand because of memory constraints, or because we cannot fit the entire dataset into memory.

Given a set of points P, the objective of the k-means setting is to minimize

$$v_c(P) = \sum_{p \in P} w(p) d(p, C)^2$$

where C is our coreset, $w(p)$ is the weight associated with that particular point and $d(p, C)$ is the distance of point p from the coreset C. The objective of the k-medians setting is similar, to minimize

$$v_c(P) = \sum_{p \in P} w(p) d(p, C)$$

2.5.1 Streaming Algorithm

The algorithm they present in a streaming works as follows:

1. Partition point sequence into chunks $X_1, X_2, \ldots, X_n$.
2. Build a d-dimensional grid in the point space. (The point are in $\mathbb{R}^d$).
3. Associate each point in X_1 with the d-dimensional box it is in. Sample one 'representative' from the grid uniformly at random to be in the coreset C_1.
4. If too many items are in the coreset, double the side length of the box.
5. Take in the points in X_2, repeat steps 2-4 to build coreset C_2. Take a union of C_1 and C_2.
6. Continue until end of stream.

This process is possible because of what they call the Merge and Reduce properties of coreset:

1. **Merge property:** If Q is a α-corest for P and Q' is a β-coreset of P', then $Q \cup Q'$ is an $(\alpha\beta)$-coreset for $P \cup P'$.
2. **Reduce property:** If Q is a α-corest for P and R is a β-coreset of Q, then R is an $(\alpha\beta)$-coreset for P.

Each box in the grid sends one representative chosen uniformly at random to the coreset Qi, and the weight of representative is the sum of weights of points in the box. If too many points in coreset, double the side-length of the grid boxes.

2.5.2 Final notes and thoughts:

This is a note on the implementation that departs from the way the authors have presented on their work. The paper is structured by first explaining their k-means and k-medians algorithms, and then later in another section for streaming they talk about how to change their algorithm for the streaming case. However, in their construction of their coreset for the non-streaming case they set the length of the side based on a parameter R that takes either the value of $v_c(P)/(cn)$ or $\sqrt{v_c(P)/(cn)}$ for k-medians and k-means respectively. ($v_c(P)$ is as described above, n is the length of the stream, and c is a constant). This $v_c(P)$ is suggested to be approximated in a way that was possible for the non-streaming cases similar to k-center clustering and GMMs above, where k-means is run on the data first to get an approximation for a lower bound. However, in their discussion of streaming we did not find mention of how to approximate this, since the chunk we are observing may be idiosyncratic for the dataset and actually produce a value that is lower than the true $v_c(P)$. We chose to get around this by arbitrarily setting this R to 1, which results in the same computation for both the k-medians and k-means cases.

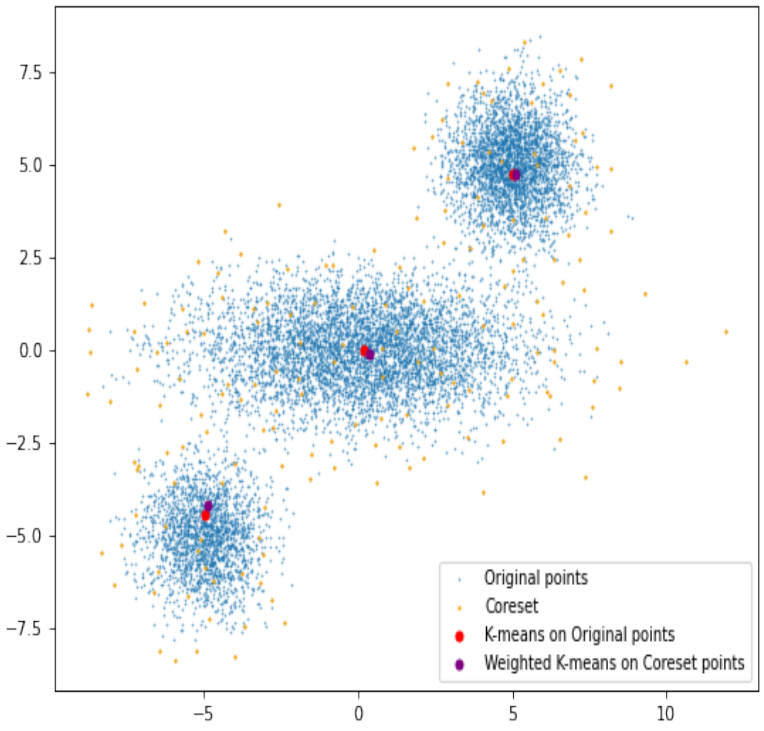

Figure 10: An example of the weighted k-means finding great approximations to the k-means run on the entire dataset.

3 References

1. Agarwal, Pankaj K., Har-Peled, Sariel, and Kasturi R. Varadarajan. "Geometric approximation via coresets." Combinatorial and computational geometry 52.1-30 (2005): 3.

2. Feldman, Dan, Matthew Faulkner, and Andreas Krause. "Scalable Training of Mixture Models via Coresets." NIPS. 2011.

3. Sariel Har-Peled, and Soham Mazumdar. "On coresets for k-means and k-median clustering." Proceedings of the thirty-sixth annual ACM symposium on Theory of computing. 2004.

4. Mario Lucic, Matthew Faulkner, Andreas Krause, and Dan Feldman. "Training Gaussian Mixture Models at Scale via Coresets." Journal of Machine Learning Research. 2018.