AutoCoreset: An Automatic Practical Coreset Construction Framework

Alaa Maalouf [1 *] **Murad Tukan** [2 *] **Vladimir Braverman** [3] **Daniela Rus** [1]

Abstract

A coreset is a tiny weighted subset of an input set, that closely resembles the loss function, with respect to a certain set of queries. Coresets became prevalent in machine learning as they have shown to be advantageous for many applications. While coreset research is an active research area, unfortunately, coresets are constructed in a problem-dependent manner, where for each problem, a new coreset construction algorithm is usually suggested, a process that may take time or may be hard for new researchers in the field. Even the generic frameworks require additional (problem-dependent) computations or proofs to be done by the user. Besides, many problems do not have (provable) small coresets, limiting their applicability. To this end, we suggest an automatic practical framework for constructing coresets, which requires (only) the input data and the desired cost function from the user, without the need for any other task-related computation to be done by the user. To do so, we reduce the problem of approximating a loss function to an instance of vector summation approximation, where the vectors we aim to sum are loss vectors of a specific subset of the queries, such that we aim to approximate the image of the function on this subset. We show that while this set is limited, the coreset is quite general. An extensive experimental study on various machine learning applications is also conducted. Finally, we provide a "plug and play" style implementation, proposing a user-friendly system that can be easily used to apply coresets for many problems. We believe that these contributions enable future research and easier use and applications of coresets.

[*]Equal contribution [1]CSAIL, MIT, Cambridge, USA. [2]DataHeroes, Israel. [3]Department of Computer science, Rice university. Correspondence to: Alaa Maalouf <alaam@mit.edu>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

1. Introduction and Motivation

In many machine learning (ML) problems, the input is usually a set $P = \{p_1, \cdots, p_n\}$ of n items, a (probably infinite) set of candidate solutions $\mathcal{X}$ called query set, and a loss function $f : P \times \mathcal{X} \to [0, \infty])$. The goal is to find a query (model, classifier) x^* that minimizes the sum $\sum_{i=1}^{n} f(p_i, x)$ over every query $x \in \mathcal{X}$. Notably, many of these optimization/learning tasks are typically challenging to approximate when the input is very large. Furthermore, in the era of big data, we usually aim towards maintaining a solution for streaming and/or distributed input data, while consuming small memory. Finally, even well-known problems with a close optimal solution, such as ridge regression and other classes of convex optimization involving Cross-validation methods or hyper-parameter tuning methods, must analyze under many restrictions several queries for various subsets of data, leading to a drastic increase in the running time.

Coresets. A common approach to solve such issues is to use data summarization techniques, namely *Coresets*, which got increasing attention over recent years (Bachem et al., 2018a;b; Bădoiu & Clarkson, 2008; Balcan et al., 2013; Braverman et al., 2019; Curtain et al., 2019; Feldman et al., 2010; 2014; Karnin & Liberty, 2019); see surveys in (Feldman, 2020; Munteanu & Schwiegelshohn, 2018; Phillips, 2016). A coreset, informally, is a tiny weighted subset of the input set P, roughly approximating the loss of P for every possible query $x \in \mathcal{X}$, up to a bound of $1 \pm \varepsilon$ factor ($0 \leq \varepsilon < 1$). The size of the coreset is often independent or close to logarithmic in the amount of the input points n, but polynomial in $1/\varepsilon$. Coresets are useful in ML as they significantly increase the efficiency of ML solvers. Specifically, employing conventional methods on the constructed coresets should approximate the optimal solution on the entire dataset, in orders of magnitude less expensive time and memory. Furthermore, by repeatedly running existing heuristics on the coreset in the time it takes to run them once on the original (large) dataset, heuristics that are already quick can be more accurate. Additionally, coresets can be maintained for distributed and streaming data.

So what's the problem? Obtaining non-trivial theoretical guarantees is frequently impossible in many contemporary machine learning problems due to either the target model being highly complex or since every input element $p \in P$ is

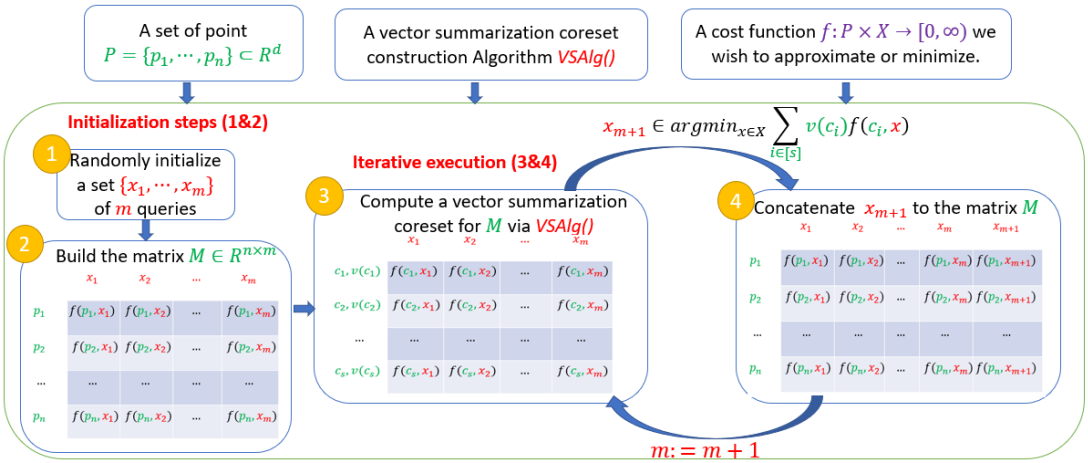


Figure 1. A flowchart illustrating our automatic coreset construction framework. Note that *VSAlg()* can be any algorithm from Table 1.

significant in the sense of high sensitivity; see (Tukan et al., 2020). Hence, generating a coreset becomes a highly challenging process, and the corresponding theoretical analysis occasionally falls short of recommending such approximations. As a result, designing a new coreset and demonstrating its accuracy for a new ML problem might take years, even for simple ones.

Another crucial issue with current theoretical frameworks is their lack of universality. Even the most general frameworks (e.g., (Feldman & Langberg, 2011; Langberg & Schulman, 2010) replace the problem of generating a coreset for an input set P of n points with n new optimization problems (one problem for each of the n input points $p \in P$) known as sensitivity bounding. Solving these may be more difficult than solving the original problem, where for every $p \in P$ we are required to bound its own sensitivity defined as $s(p) = \sup_{x \in \mathcal{X}} \frac{f(p,x)}{\sum_{q \in P} f(q,x)}$. As a result, distinct approximation strategies are often adapted to each task. Hence, the main disadvantage of such frameworks is that researchers provide papers solely for bounding the sensitivities with respect to a certain problem or a family of functions (Tukan et al., 2020; Maalouf et al., 2020), limiting the spread of coresets, as non-expert won't be able to suggest coresets for their desired task. These problems raise the following questions:

Is it possible to design an automatic and practically robust coreset construction framework (for any desired cost function and input dataset) that does not need sensitivity calculation or any other problem-dependent computation by the user? Can we provide some provable guarantees with respect to this framework?

1.1. Vision

Goal. Our goal is to provide a single algorithm that only receives the loss function we wish to compute a coreset for and the input dataset, then, it practically outputs a good coreset for the input dataset with respect to the given loss. This algorithm should be generic, efficient, and work practically well for many problems.

motivation. The main motivation behind this goal is (1) to increase the spread and use of coreset to a larger community that is not limited to coreset researchers or pioneers. (2) Additionally, to ease the use of coresets for many other applications that may be out of the scope of the coreset literature, and finally, to (3) easily apply coresets for new problems that do not have provable coresets. Theoretically speaking, it is indeed very hard to provide a "theoretical strong coreset" to any problem – for example, there exist lower bounds on the coreset sizes for different problems (Munteanu et al., 2018; Tukan et al., 2021). Thus we aimed at a practical result while providing weaker theoretical guarantees, with an extensive experimental study.

1.2. Our contribution

In this paper, we provide a coreset construction mechanism that answers both questions. Specifically:

(i) The first automatic practical coreset construction system that only needs to receive the loss function associated with the problem. Our coreset does not require any computation to be done by the user, not mathematical nor technical (without the need for sensitivities or any other task-related computation by the user). To the best of our knowledge, this is the first paper to suggest a **plug-and-play** style framework/compiler for coreset construction. We also provide a theoretical justification for using our framework.

(ii) An extensive empirical study on real-world datasets for various ML solvers of Scikit-Learn (Pedregosa et al.,

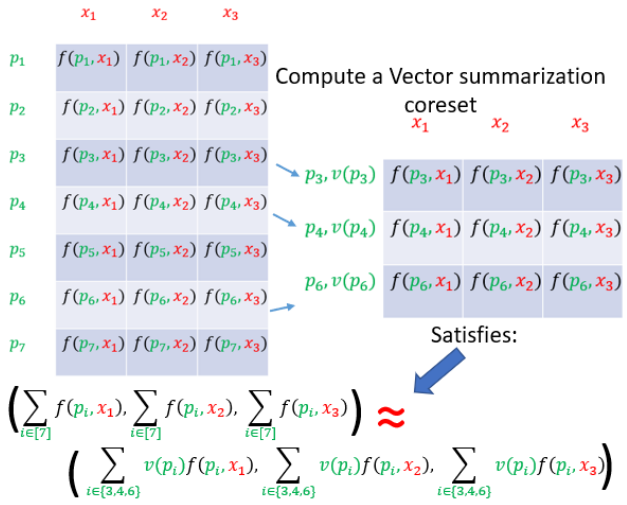


Figure 2. Illustration of a vector summarization coreset for an input matrix of 7 rows and 3 columns which represent the loss function concerning a set P of 7 input points, and set of queries x_1, x_2, x_3.

2011), including k-means, logistic regression, linear regression, and support vector machines (SVM), showing the effectiveness of our proposed system.

(iii) **AutoCoreset:** An open-source code implementation of our algorithm for reproducing our results and future research. For simplicity and ease of access, to obtain a coreset, the user only needs to plug in his desired loss function and the data into our system. We believe this system will popularize and expose the use of coresets to other scientific fields.

2. Setup Details

Given a set $P = \{p_1 \cdots, p_n\} \subseteq \mathbb{R}^d$ of n points[1] and a loss function $f : P \times \mathcal{X} \to [0, \infty)$ where $\mathcal{X}$ is a (possibly infinite) set of queries. In this paper, we develop an automatic coreset construction framework for any problem involving cost functions of the form $\sum_{p \in P} f(p, x)$, here $x \in \mathcal{X}$. Formally, we wish to find a small subset $\mathcal{I} \subseteq [n]$ and a weight function $v : \mathcal{I} \to [0, \infty)$ such that $\max_{x \in \mathcal{X}} \frac{\sum_{j \in \mathcal{I}} v(j) f(p_j, x)}{\sum_{i=1}^{n} f(p_i, x)} \in 1 + O(\varepsilon)$, for some small $\varepsilon \geq 0$.

2.1. Preliminaries

We now give our notations and used Definition.

Notations. For a pair of integers $n, m > 0$, we denote by $[n]$ the set $\{1, \cdots, n\}$, and by $\mathbb{R}^{n \times m}$ the set of every possible $n \times m$ real matrix. For a matrix $M \in \mathbb{R}^{n \times m}$ and a pair of integers $i \in [n], j \in [m]$, we use $M_{i,*}$ to denote its ith row (vector), $M_{*,j}$ to denote its jth column, and $M_{i,j}$ to denote the entry in the ith row and jth columns.

In what follows, we define a crucial component on which our system relies, namely, *vector summarization coreset*.

Definition 2.1 (Vector summarization coreset)**.** Let $M \in \mathbb{R}^{n \times m}$, $\mathcal{I} \subseteq [n]$, $v : \mathcal{I} \to [0, \infty)$ be a weight function, and let $\varepsilon > 0$. The tuple $(\mathcal{I}, v)$ is an vector summarization ε-coreset for M if $\left\| \sum_{i \in [n]} M_{i,*} - \sum_{j \in \mathcal{I}} v(j) M_{j,*} \right\|_2^2 \leq \varepsilon$.

Many papers suggested different algorithms for computing such coresets; in Table 1 summarizes some of these results, as we can use them all of them in our method.

3. *AutoCoreset*

A coreset aims to approximate the probability distribution induced upon the input data by the cost function. Hence, in order to approximate a given cost function, the coreset must contain points that can result in an approximated distribution to that of the full data.

Key idea. Loosely speaking, assume that for a given cost function f and a set $P = \{p_1, \cdots, p_n\} \subset \mathbb{R}^d$, we access an infinite matrix $\mathcal{M}^*(P, f)$ where the rows correspond to the n points of P, and each column corresponds to a query point from the infinite set of queries $\mathcal{X}$. Specifically, each row $i \in [n]$ is of infinite length representing the loss of each point $p_i \in P$ with respect to the infinite set of queries $\mathcal{X}$. A coreset in this context means finding a subset of the rows $\mathcal{I} \subseteq [n]$, and a weight function $v : \mathcal{I} \to [0, \infty]$, that satisfies the vector summarization coreset guarantee (see Definition 2.1), i.e.,

$$\left\| \sum_{i \in [n]} \mathcal{M}^*(P, f)_{i,*} - \sum_{j \in \mathcal{I}} v(j) \mathcal{M}^*(P, f)_{j,*} \right\|_2^2 \leq \varepsilon. \quad (1)$$

From such a coreset $\mathcal{I} \subseteq [n]$, the cost function can be approximated, since for every query $x \in \mathcal{X}$ (column in the matrix $\mathcal{M}^*(P, f)$), the weighted sum of losses over the coreset $\mathcal{I}$ approximate the original sum of losses of the whole data. While such a concept is admirable, having an access to such immense data is rather imaginative.

Recently (Maalouf et al., 2022) showed that for an input set of points P, and query space X that is defined as a family of sine wave functions, a coreset can be constructed. Specifically, it was shown that if the coreset approximates the loss of every query in a smaller set of queries on the input data, then it will also approximate the losses of the whole set of queries (sine waves). Thus, indeed, the sine wave that fits best the coreset approximates the sine wave that best fits

[1] if P is a set of labeled items, then $P = \{p_i = (p_i', y_i) | p_i' \in \mathbb{R}^{d-1}, y_i \in \mathbb{R}\}_{i=1}^{n}$

Algorithm 1 AUTOCORESET (P, f, τ, m, ζ)

input set of n points P, a loss function f, a coreset size τ, number of initial models m, and an stopping criterion ζ
output A coreset $(\mathcal{I}, v)$ such that

1: $\tilde{\mathcal{M}}(P, f) \leftarrow \vec{0}_{n \times m}$
2: **for** each $i \in [m]$ **do**
3: $x_i \leftarrow$ a randomized approximated solution involving P and f
4: **for** every $j \in [n]$ **do**
5: $\tilde{\mathcal{M}}(P, f)_{j,i} \leftarrow f(p_j, x_i)$
6: **end for**
7: **end for**
8: **repeat**
9: $(\mathcal{I}, v) \leftarrow$ coreset of m indices for vector summarization problem involving $\tilde{\mathcal{M}}(P, f)$ {See Definition 2.1}
10: $x^* \leftarrow \arg\min_{x \in \mathcal{X}} \sum_{i \in \mathcal{I}} v(i) f(p_i, x)$
11: $\tilde{\mathcal{M}}(P, f) \leftarrow \left[\tilde{\mathcal{M}}(P, f) \middle| \vec{0}_n\right]$
12: **for** every $i \in [n]$ **do**
13: $\tilde{\mathcal{M}}(P, f)_{i,m+1} \leftarrow f(p_i, x_C)$
14: **end for**
15: $m \leftarrow m + 1$
16: **until** ζ is satisfied
return $(\mathcal{I}, v)$

the entire data. Inspired by such a result, we aim towards constructing a sub-matrix $\tilde{\mathcal{M}}(P, f)$ of $\mathcal{M}^*(P, f)$ ($\tilde{\mathcal{M}}(P, f)$ contain a subset of the columns of $\mathcal{M}^*(P, f)$; see Figure 2 for illustration) such that constructing the coreset on $\tilde{\mathcal{M}}(P, f)$ (a weighted subset of the rows of $\tilde{\mathcal{M}}(P, f)$) will also yield a similar coreset to that of (1) on the $\mathcal{M}^*(P, f)$. But, how to build the sub-matrix $\tilde{\mathcal{M}}(P, f)$? how to choose the query set corresponding to the columns of $\tilde{\mathcal{M}}(P, f)$?

Table 1. Summary of known vector summarization coresets and their properties.

Method	Probability of failure	Approximation error	Coreset size $\lvert\mathcal{I}\rvert$	Construction time
Caratheodory (Maalouf et al., 2019) (Carathéodory, 1907)	0	0	$m+1$	$O(\min\{nm + \log^4(m), m^2n^2, nm^3\})$
Frank-Wolfe (Feldman et al., 2017) (Clarkson, 2010)	0	ε	$O(1/\varepsilon)$	$O(\min\{nd/\varepsilon\})$
Median of means tournament (Minsker, 2015)	δ	ε	$O(1/\varepsilon)$	$O(m \log^2(1/\delta) + m \log(1/\delta)/\varepsilon)$
Sensitivity sampling (Feldman & Langberg, 2011)	δ	ε	$O(\frac{1}{\varepsilon}(m + \log \frac{1}{\delta}))$	$O(nm)$
Uniform sampling	δ	ε	$O(\frac{1}{\varepsilon\delta})$	$O(1)$

3.1. A deeper look into *AutoCoreset*

We now give and explain our algorithm AUTOCORESET (see Algorithm 1), which aims to provide a parasitical coreset with similar guarantees.

Into the forging of our coresets. Let $m > 1$ be an integer. First, a matrix $\tilde{\mathcal{M}}(P, f)$ is generated to contain $n \times m$ zero entries, followed by generating a set $\mathcal{X}' = \{x_1, \ldots, x_m\}$ of m approximated solutions with respect to $\min_{x \in \mathcal{X}} \sum_{i=1}^{n} f(p_i, x)$ as depicted at Lines 1–7. If no such approximated solution exists, then the initialization may be also completely random. The (sub)matrix $\tilde{\mathcal{M}}(P, f)$ is now initialized, where for every $i \in [n]$, and $j \in [m]$, the entry $\tilde{\mathcal{M}}(P, f)_{i,j}$ in the ith row and jth column is equal to $f(p_i, x_j)$. While the properties associated with generated solutions at Line 3 hold with some probability, our framework is always guaranteed practically to generate a good coreset. This is due to the fact that these solutions are merely used as an initialization mechanism.

From this point on, a loop is invoked. First, using the current state of $\tilde{\mathcal{M}}(P, f)$, a vector summarization coreset $(\mathcal{I}, v)$ (see Definition 2.1) is generated with respect to the rows of $\tilde{\mathcal{M}}(P, f)$.

A coherent claim of our system is that any vector summarization coreset $\mathcal{I}$ for the rows of $\tilde{\mathcal{M}}(P, f)$, is directly mapped to coreset for P (using the same set of indexes and the same weight function) with respect to the query set $\mathcal{X}' \subset \mathcal{X}$ and the function f, where $\mathcal{X}'$ is the set of all queries that brought about the columns of $\tilde{\mathcal{M}}(P, f)$. More preciously, $\max_{x \in \mathcal{X}'} \frac{\sum_{j \in \mathcal{I}} v(j) f(p_j, x)}{\sum_{i=1}^{n} f(p_i, x)} \in 1 + O(\varepsilon)$; see Lemma 3.1.

Since the computed vector summarization coreset $\mathcal{I}$ is also a coreset with respect to f, P, and $\mathcal{X}'$, we can optimize f over the small coreset $\mathcal{I}$ to obtain a new query $x^* \in \mathcal{X}$ that gives an approximated solution to the full data (see Line 10). We then apply the loss f function and the new solution x^* on $p_1, \cdots, p_n$ to obtain the vector of losses $l = (f(p_1, x^*), \cdots, f(p_n, x^*))^T$, and concatenate such vector of loss values to $\tilde{\mathcal{M}}(P, f)$ as its last column. This aids in expanding the exposure of generated coreset to a wider spectrum of queries, leading towards a *strong coreset*. Observe that in the next iteration, when we compute a new coreset for the given set of queries, the coreset will approximate all of the previous ones (set of queries) and the new computed query/solution x^*.

This procedure is repeated until some stopping criterion ζ is invoked – we provide more details on the used ζ in Section 5. We refer the reader to Lines 10–15. Note that if we were able to run the above procedure infinitely while ensuring that at each iteration a new solution is computed, $\mathcal{M}^*(P, f)$ would have been generated, resulting in the "strong coreset" this system is leaning towards. To better grasp the idea of the framework, we provide a flowchart illustration at Figure 1.

The parameters τ, m, ζ. Our Algorithm initializes its matrix $\tilde{\mathcal{M}}(P, f)$ with respect to the losses of $m > 1$ different queries, and outputs a coreset of size $\tau > 1$, hence, the larger the m and τ the better the approximation, but the slower the time; See section 5 for more details. Regarding

ζ, it is the used stopping criterion, we provide full details regarding the used ζ in Section 5.

3.2. Weaker coresets are fine too

Our *AutoCoreset* system, while ambitiously aims towards holding a grasp over $\mathcal{M}^*(P,f)$, it finds a weaker version of the "strong coresets". Specifically speaking, it finds a coreset that attains approximation guarantees with respect to a subset of the query set $\mathcal{X}$. Theoretically speaking, the following lemma summarizes one aspect of the theoretical properties guaranteed by *AutoCoreset*.

Lemma 3.1 (Vector summarization coreset $\rightarrow$ "a weak coreset for any loss")**.** *Let $P = \{p_1, \cdots, p_n\} \subseteq \mathbb{R}^d$ be a set of n points as defined in Section 3.1, $\mathcal{X}' \subset \mathcal{X}$ be a set of queries, $f : P \times \mathcal{X} \rightarrow [0,\infty)$ be a loss function, and let $\tilde{\mathcal{M}}(P,f) \in \mathbb{R}^{n \times |\mathcal{X}'|}$ be the loss matrix defined with respect to $P, f, \mathcal{X}'$ as in Algorithm 1. Let $\tau \geq 1$ be an integer, and let $(\mathcal{I}, v)$ be a ε-vector summarization coreset concerning $\tilde{\mathcal{M}}(P,f)$ of size $|\mathcal{I}| = \tau$. Then, for every $x \in \mathcal{X}'$,*

$$\left| \sum_{i \in [n]} f\left(p_i, x\right) - \sum_{j \in \mathcal{I}} v(j) f\left(p_j, x\right) \right|^2 \leq \varepsilon.$$

Implications of Lemma 3.1. *AutoCoreset* guarantees theoretically that for a finite set of queries $\mathcal{X}'$, a coreset can be constructed supporting $\mathcal{X}'$. A key advantage here would be the ability to represent any query x such that its loss vector $(f(p_1,x), \cdots, f(p_n,x))$ lies inside the "convex hull" of the loss vectors of the query set $\mathcal{X}'$. Luckily, such a trait is supported by our system. Specifically speaking, for any query such that its corresponding loss vector ℓ with respect to f and P can be formulated as a convex combination of the columns of $\tilde{\mathcal{M}}(P,f)$, then a vector summarization coreset for the rows of $\tilde{\mathcal{M}}(P,f)$ is also a vector summarization to the rows of concatenating $\tilde{\mathcal{M}}(P,f)$ and the column vector ℓ. In what follows, we give the theoretical justification for the above claim.

Claim 3.2 (Weak Coreset with hidden abilities)**.** *Let $P = \{p_1, \cdots, p_n\} \subseteq \mathbb{R}^d$ be a set of n points as in Section 3.1, f be a loss function supported by* AutoCoreset*, and let m, τ, ζ be the defined number of initial solutions, sample size, and stopping criterion, respectively. Let $z \geq m$, $(\mathcal{I}, v)$ be the output of a call to* AutoCoreset (P, f, τ, m, ζ)*, and let $\tilde{\mathcal{M}}(P,f) \in \mathbb{R}^{d \times z}$ be the matrix of losses that was constructed throughout the running time of* AutoCoreset*; see Lines 1, 5, 11 13 at Algorithm 1. Then for any weight function $\alpha : [z] \rightarrow [0,1]$ where $\sum_{i=1}^{z} \alpha(i) = 1$, and any $x \in \mathcal{X}$ satisfying that for every $i \in [n]$, $f\left(p_i, x\right) = \sum_{k=1}^{z} \alpha(k) \tilde{\mathcal{M}}(P,f)_{i,k}$, we have*

$$\left| \sum_{i=1}^{n} f\left(p_i, x\right) - \sum_{j \in \mathcal{I}} v(j) f\left(p_j, x\right) \right|^2 \leq \varepsilon,$$

where $\varepsilon \geq 0$ is the approximation factor associated with generating a vector summarization coreset of m points.

The best of both worlds. Claim 3.2 states that even if it seems that our generated coreset only supports a handful of queries from $\mathcal{X}$, our coreset basically supports many more queries. The highlight of such a claim is that if the optimal solution for the objective function involves f and P, then our coreset becomes stronger in the sense of ensuring better quality even during the training/optimization process which involves both f and P. Such a claim is usually targeted via "Strong coresets" and mainly by "Weak coresets". *AutoCoreset* ensures a coreset that resides on the spectrum involving these coresets at its ends, i.e., generating a coreset from the best of both worlds – a coreset supporting the optimal solution that the user is aiming to solve using accelerated training via coresets while maintaining the provable approximation guarantees of strong coresets to some extent.

4. Size, Space, and Time Analysis

Time complexity. Let `VAlg` be the vector summarization algorithm used at Line 9 of Algorithm 1 (pick one from Table 1). Let $\varepsilon, \delta \in (0,1)$ be the desired vector summarization approximation error, and probability of failure, respectively. Now denote by

- $T(n,i,\varepsilon,\delta)$: the running time of `VAlg` on a matrix of n rows and i columns with respect to ε and δ.
- $S(n,i,\varepsilon,\delta)$: the size of the coreset computed by VAlg on a matrix of n rows and i columns with respect to ε and δ.
- $T_{sol}(n,d)$: the time required to compute a solution vector x^* for n points in the d dimensional space with respect to the problem at hand (e.g., the time required to compute the solution of linear regression is $O(nd^2)$).
- $T_{cost}(n,d)$: the time required to calculate the cost for n points in the d dimensional space on a single query with respect to the problem at hand (e.g., the time required to compute the cost of linear regression for n points in the d dimensional space given a solution vector x is $O(nd)$. "t": be the number of iterations of the algorithm.

At each iteration "i", Algorithm 1

1. applies `VAlg` on a matrix of n rows and i columns to obtain a coreset of size $S(n, i, \varepsilon, \delta)$. This step requires $T(n, i, \varepsilon, \delta)$ time.

2. Solves the problem on the coreset to obtain a new solution x^*. Requiring $T_{sol}(S(n, i, \varepsilon, \delta), d)$ time.

3. Calculates the cost of the n points with respect to x^*. Requiring $T_{cost}(n, d)$ time

Thus, for a single step i the running time is $T(n, i, \varepsilon, \delta) + T_{sol}(S(n, i, \varepsilon, \delta), d) + T_{cost}(n, d)$. Summing for t iterations:

$$\sum_{i=1}^{t}(T(n, i, \varepsilon, \delta) + T_{sol}(S(n, i, \varepsilon, \delta), d)) + tT_{cost}(n, d).$$

For example, in Linear regression and using the Sensitivity sampling as *VAlg*, an immediate bound for the running time is $O(t(nt + (t/\varepsilon + log(1/\delta)/\varepsilon)d^2 + nd))$.

Space complexity. First, note that the input data and the matrix of losses take $O(n(d + t))$ where t here denotes the number of iterations our coreset generation has taken. Recall the definitions of *VAlg*, ε, δ and $S(n, i, \varepsilon, \delta)$. We now denote by

- $Mem(\textit{VAlg}, \varepsilon, \delta, i)$ the amount of space needed by Valg to generate an ε-coreset with a success probability of at least $1 - \delta$.

- $Mem_{sol}(n, d)$ the space required to compute a solution vector x^* for n points in the d dimensional space with respect to the problem at hand (e.g., the space required to compute the solution of SVM is $O(n^2 + d)$.

The total space complexity is thus bounded by $O(n(d+t)+\max_{i\in[t]}(Mem(\textit{VAlg}, \varepsilon, \delta, i) + Mem_{sol}(S(n, i, \varepsilon, \delta)), d)$.

For example for SVM and using the Sensitivity sampling vector summarization, an immediate bound for the space complexity is $O(n(d + t) + (1/\varepsilon(t + \log(1/\delta)))^2)$.

Coreset size. The size of the constructed coreset is equal to the used vector summarization coreset size (See Table 1), and it depends on the approximation error ε, the probability of failure δ we wish to have, and the final number of approximated queries – columns of the query matrix.

In short – let ε be the desired approximation error and let δ be the probability of failure. Let t be the number of iterations required Algorithm 1. Denote by $S(n, i, \varepsilon, \delta)$ the size of the set computed by the used vector summarization algorithm on a matrix of n rows and i columns with respect to ε and δ (see Table 1 for examples). Then, the size of the coreset is $S(n, t, \varepsilon, \delta)$.

For example, using the Sensitivity sampling method (as the used vector summarization coreset), to approximate the currently given t queries after t iterations, with ε approximation error, and δ probability of failure, we get a coreset of size $O(t/\varepsilon + log(1/\delta)/\varepsilon)$.

From additive to multiplicative approximation error. Algorithm 1 can immediately be modified to compute a coreset that yields a multiplicative approximation as follows. Given the set P, the current set of queries $\mathcal{X}'$, and the loss f, define a new function $g(p, x) := \frac{f(p,x)}{\sqrt{\sum_{p\in P} f(p,x)}}$ for every pair of a query $x \in \mathcal{X}'$ and input data $p \in P$.

Now build the corresponding matrix $\tilde{\mathcal{M}}(P, g)$ (as done in Algorithm 1 for $f(p, x)$) instead of $\mathcal{M}(P, f)$, and run the exact same vector summarization coreset algorithm on it. Then, by Lemma 3.1, for every $x \in \mathcal{X}'$, $\left|\sum_{i\in[n]} g(p_i, x) - \sum_{j\in\mathcal{I}} v(j)g(p_j, x)\right|^2 \leq \varepsilon$, and by the definition of g we get that the result is a multiplicative coreset for the given set of queries as for every $x \in \mathcal{X}'$

$$\left|\sum_{i\in[n]} f(p_i, x) - \sum_{j\in\mathcal{I}} v(j)f(p_j, x)\right|^2 \leq \varepsilon \sum_{p\in P} f(p, x).$$

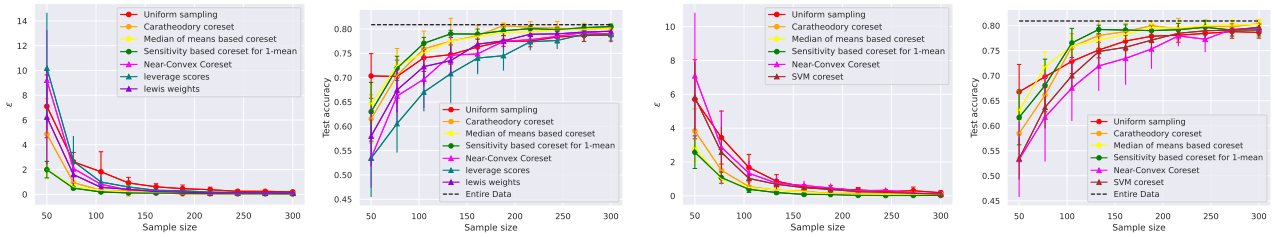
5. Experimental Study

In what follows, we first discuss the choices of different vector summarization coresets, and the used parameters in our experiments, followed by evaluating our coreset on real-world datasets, against other famous competing methods: Near Convex Coreset (Tukan et al., 2020), Lewis weights (Munteanu et al., 2018) and leverage scores (Munteanu et al., 2018) for logistic regression, Near Convex Coreset (Tukan et al., 2020) and optimization based coreset (Tukan et al., 2021) for support vector machines (SVM), SVD-based coreset (Maalouf et al., 2020) for linear regression, Bi criteria coreset (Braverman et al., 2021) for k-means, and uniform sampling in all of the experiments. We note that each experiment was conducted for 16 trials, we report both the mean and std for all of the presented metrics.

Software/Hardware. Our algorithms were implemented in Python 3.9 (Van Rossum & Drake, 2009) using "Numpy" (Oliphant, 2006), "Scipy" (Virtanen et al., 2020) and "Scikit-learn" (Pedregosa et al., 2011). Tests were performed on 2.59GHz i7-6500U (2 cores total) machine with 16GB RAM.

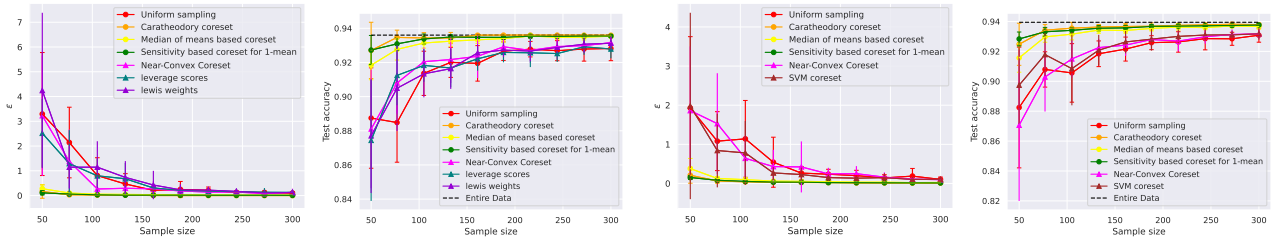
5.1. *AutoCoreset* parameters

Vector summarization coresets. There are many methods for computing such coresets, some of them are deterministic, i.e., with no probability of failure, and others work with



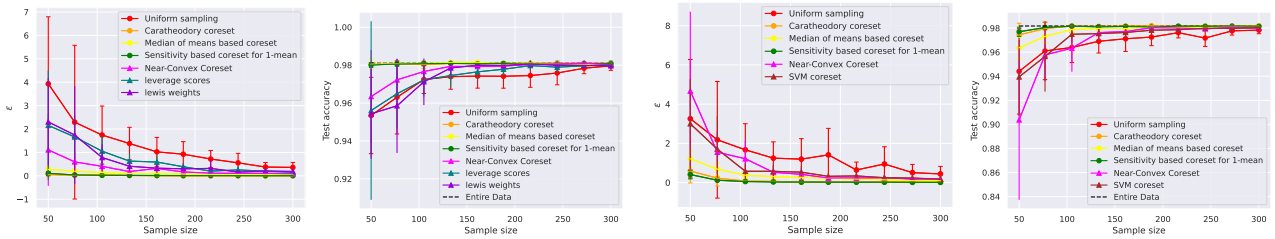
(a) Logistic regression (b) SVMs

Figure 3. Evaluation of our coresets against other competing methods on the Dataset (i).



(a) Logistic regression (b) SVMs

Figure 4. Evaluation of our coresets against other competing methods on the Dataset (ii).



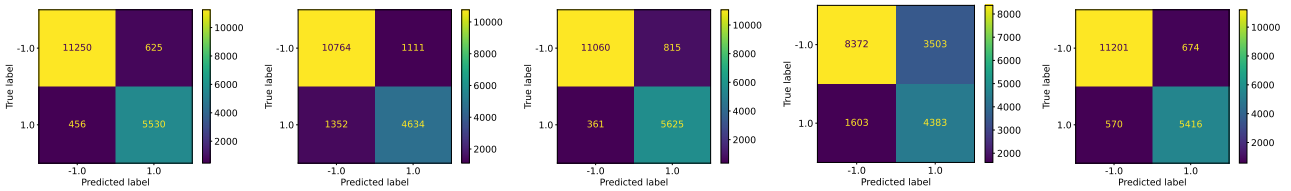
(a) Logistic regression (b) SVMs

Figure 5. Evaluation of our coresets against other competing methods on the Dataset (iii).

some probability $1-\delta$. On the other hand, some are accurate, i.e., $\varepsilon = 0$, and others yield an approximation error $\varepsilon > 0$. In Table 1 we summarize some of the common methods for computing such coresets, and their properties, such as size, running time, approximation error, and probability of failure. In our system, we implemented all of the given methods and compared them via extensive experiments.

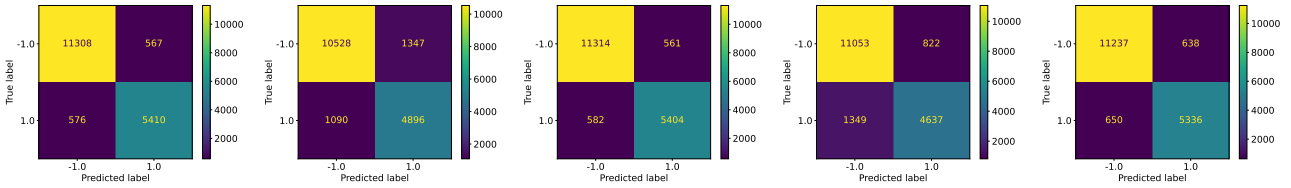
Setting the number of initial solutions m. Throughout our experiments, we have set the number of initial solutions to 10. The idea behind this is to expose AUTOCORESET to a number of solutions that is not too high nor too low. Hence, we ensure that the coreset is not too weak nor too dependent on the initial solutions.

Choosing a stopping criterion ζ. Inspired by the early-stopping mechanism of (Prechelt, 1998), we adopt a similar idea. We make use of a parameter, namely "patience", which was set to 7, to attempt an indication of the occurrence of saturation with respect to the exposure of our coreset paradigm to new queries; see more details at Section A. To correctly use this parameter, we use additional two parameters, one of which is a counter, while the other holds the optimal coreset that resulted in the smallest sum of the entries of the concatenated columns (see Line 13 at Algorithm 1). The counter will be reset to 0 once a new column is added such that its sum is lower than the smallest sum so far, and the optimal coreset will be updated. Otherwise, the counter will be increased. AUTOCORESET will keep running until the above counter reaches the "patience" parameter. In our experiments, we returned the optimal coreset since it led to better results. For completeness, we refer the reader to the appendix where we conduct an ablation study and check our results without taking the optimal coreset, i.e., in those results, we take the last coreset. Note that, in both sets of



(a) Full data (b) Uniform sampling (c) Carathédory coreset (d) Median of means coreset (e) Importance sampling based coreset

Figure 6. SVM confusion matrices with respect to our coresets against Uniform sampling and the entire data of Dataset (ii).



(a) Full data (b) Uniform sampling (c) Carathédory coreset (d) Median of means coreset (e) Importance sampling based coreset

Figure 7. Logistic regression confusion matrices with respect to our coresets against Uniform sampling and the entire data of Dataset (ii).

experiments, we outperform the competing methods.

Datasets. The following datasets were used throughout our experimentation. These datasets were taken from (Dua & Graff, 2017) and (Chang & Lin, 2011): (i) Credit card dataset (Yeh & Lien, 2009) composed of 30000 points with 24 features representing customers' default payments in Taiwan, (ii) Cod-RNA dataset (Uzilov et al., 2006): dataset containing 59535 points with 8 features, (iii) HTRU dataset (Lyon et al., 2016): Pulsar candidates collected during the HTRU survey containing 17898 each with 9 features, (iv) $3D$ Road Network (Guo et al., 2012): $3D$ road network with highly accurate elevation information from Denmark containing 434874 points each with 4 features, (v) Accelerometer dataset (Sampaio et al., 2019): an accelerometer data from vibrations of a cooler fan with weights on its blades containing 153000 points consisting each of 5 features, and (vi) Energy efficiency Data Set (Tsanas & Xifara, 2012): a dataset containing 768 points each of 8 features.

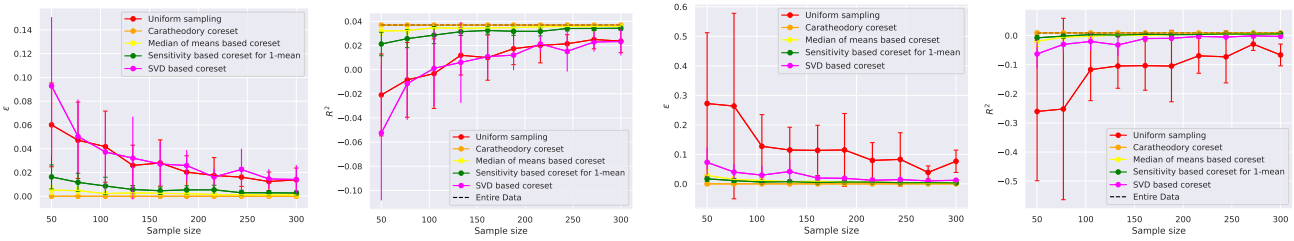
ML models. Throughout our entire set of experiments, we have relied on "Scikit-Learn" ML models.

Reported results. First, for each coreset $(\mathcal{I}, v)$ of an input data P and a loss function f, we compute the optimal solution on the coreset $x^*_{\mathcal{I}} \in \arg\min_{X\in\mathcal{X}} \sum_{i\in\mathcal{I}} v(i) f(p_i, x)$, and on the real data $x^*_P \in \arg\min_{x\in\mathcal{X}} \sum_{i\in[n]} f(p_i, x)$, and we report the optimal solution approximation error $\varepsilon = \left|\sum_{i\in[n]} f(p_i, x^*_{\mathcal{I}}) - \sum_{i\in[n]} f(p_i, x^*_P)\right|$. Secondly, we show for classification problems the test accuracy obtained when training on the coreset, while on regression problems we show an estimate of the coefficient of determination of the prediction R^2 (Ozer, 1985). Additional measures are reported for some problems; we discuss them in the following sections. The bars in our graphs reflect the standard deviation.

5.2. Traditional ML classification problems

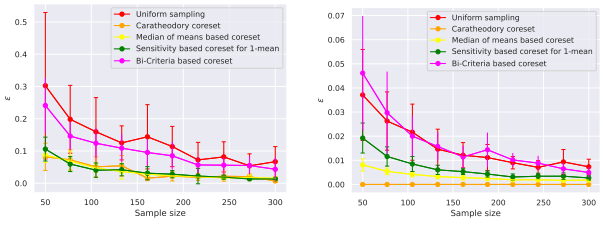
In what follows, we show our results when setting f to be the loss function of either the *Logistic* regression problem or the *SVMs* problem. In both experiments, since, some of the datasets were unbalanced, each sample coreset size has been split – small classes get a slightly larger portion of the sample size than simply taking $\eta\times$ sample size where η represents the class size percentage with respect to the total number of points, while larger classes get a portion of the sample size smaller than $\eta\times$ sample size.

Logistic regression. We have set the maximal number of iterations to 1000 (for the Scikit-Learn solver) while setting the regularization parameter to 1. Our system's approximation error was smaller by orders of magnitude, and the accuracy associated with the models trained using our coreset was better than the model trained on the competing methods; see Figure 3(a) and Figure 5(a). On the other hand, Figure 4(a) depicts a multiplicative gap of 30 with respect to the approximation error in comparison to the competing methods while simultaneously acceding by 5% accuracy gap over them. In addition, we present the confusion matrix for each of our coresets using *AutoCoreset*, and compare it to the confusion matrices with respect to the entire data and the uniform sampling coreset; See Figure 7. The confusion matrices aim towards explaining our advantage as



(a) Dataset (iv) (b) Dataset (v)

Figure 8. Evaluation of our coresets against other competitors concerning the linear regression problem.



(a) Dataset (iii), $k := 5$ (b) Dataset (vi), $k := 2$

Figure 9. Evaluation of our coresets against other competitors concerning the k-means problem.

our system outputs coresets that approximately maintain the structural properties of the confusion matrix of the entire data better than simply using uniform sampling, as our recall and accuracy are closer to their corresponding values when using the entire dataset.

SVMs. As for SVMs, we mainly focused on the linear kernel, while setting the regularization parameter to 1. Similarly to logistic regression, we outperform the competing methods both in accuracy and approximation error; see Figure 3(b) and Figure 5(b).

Discussion These results show that general frameworks that aim to handle a large family of functions without embedding some crucial information concerning the properties of the problem, usually tend to lose through the race towards smaller coresets sizes with small approximation errors. We thus show that while *AutoCoreset* is general in the reach of its applications, it also embeds the functional properties of the problem into higher consideration than that of (Tukan et al., 2020), and practically achieves robust results (smaller std).

5.3. Linear regression and k-means clustering.

In our experiments for linear regression, we observe a clear gap between each of our vector summarization coresets and the competing methods, leading towards outperforming the competing coreset for the task of fitting linear regression. In addition, we observe that the determination coefficient R^2 for our method is much closer to the determination coefficient R^2 when using the entire data. This indicates that our coresets lead to better learning and correlation between the input data and the corresponding outputs of the regression problem; see Figure 8. In addition, for k-means, our coresets outperform the competitors (see Figure 9), justifying their robustness across a wide range of applications.

6. Conclusions and Future Work

In this work, we proposed an automatic practical coreset construction framework that requires only two parameters: the input data and the loss function. Our system, namely *AutoCoreset*, results in small coresets with multiplicative approximation errors significantly smaller than traditional coreset constructions for various machine learning problems, as well as showing that the model learned on our coresets gained more information than the other coresets. While *AutoCoreset* is practical, we also show some desirable theoretical guarantees. We believe that *AutoCoreset* can be further enhanced and tuned to work in the context of Deep learning, e.g., subset selection for boosting training of deep neural networks. We leave this as future work.

Finally, we hope *AutoCoreset* will lay the foundation of practical frameworks for coresets, and hope it reaches the vast scientific community, aiding to achieve faster training with provable guarantees due to training on our coresets.

7. Acknowledgements

This research was supported in part by the AI2050 program at Schmidt Futures (Grant G-96422-63172), the United States Air Force Research Laboratory, and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000.

References

Arthur, D. and Vassilvitskii, S. K-means++ the advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 1027–1035, 2007.

Bachem, O., Lucic, M., and Krause, A. Scalable k-means clustering via lightweight coresets. In *KDD'18 Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1119–1127. ACM, 2018a.

Bachem, O., Lucic, M., and Lattanzi, S. One-shot coresets: The case of k-clustering. In Storkey, A. and Perez-Cruz, F. (eds.), *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pp. 784–792, Playa Blanca, Lanzarote, Canary Islands, 09–11 Apr 2018b. PMLR. URL http://proceedings.mlr.press/v84/bachem18a.html.

Bădoiu, M. and Clarkson, K. L. Optimal core-sets for balls. *Computational Geometry*, 40(1):14–22, 2008.

Balcan, M.-F. F., Ehrlich, S., and Liang, Y. Distributed k-means and k-median clustering on general topologies. In *Advances in Neural Information Processing Systems*, pp. 1995–2003, 2013.

Braverman, V., Jiang, S. H.-C., Krauthgamer, R., and Wu, X. Coresets for ordered weighted clustering. In Chaudhuri, K. and Salakhutdinov, R. (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 744–753, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL http://proceedings.mlr.press/v97/braverman19a.html.

Braverman, V., Feldman, D., Lang, H., Statman, A., and Zhou, S. Efficient coreset constructions via sensitivity sampling. In *Asian Conference on Machine Learning*, pp. 948–963. PMLR, 2021.

Carathéodory, C. Über den variabilitätsbereich der koeffizienten von potenzreihen, die gegebene werte nicht annehmen. *Mathematische Annalen*, 64(1):95–115, 1907.

Chang, C.-C. and Lin, C.-J. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

Clarkson, K. L. Coresets, sparse greedy approximation, and the frank-wolfe algorithm. *ACM Transactions on Algorithms (TALG)*, 6(4):63, 2010.

Curtain, R., Im, S., Moseley, B., Pruhs, K., and Samadian, A. On coresets for regularized loss minimization. *arXiv preprint arXiv:1905.10845*, 2019.

Dua, D. and Graff, C. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

Feldman, D. Core-sets: Updated survey. In *Sampling Techniques for Supervised or Unsupervised Tasks*, pp. 23–44. Springer, 2020.

Feldman, D. and Langberg, M. A unified framework for approximating and clustering data. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pp. 569–578, 2011.

Feldman, D., Monemizadeh, M., Sohler, C., and Woodruff, D. P. Coresets and sketches for high dimensional subspace approximation problems. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pp. 630–649. SIAM, 2010.

Feldman, D., Rossman, G., Volkov, M., and Rus, D. Coresets for k-segmentation of streaming data. In *NIPS*, 2014.

Feldman, D., Ozer, S., and Rus, D. Coresets for vector summarization with applications to network graphs. In *International Conference on Machine Learning*, pp. 1117–1125. PMLR, 2017.

Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., Liu, T., Wang, X., Wang, G., Cai, J., et al. Recent advances in convolutional neural networks. *Pattern recognition*, 77:354–377, 2018.

Guo, C., Ma, Y., Yang, B., Jensen, C. S., and Kaul, M. Ecomark: evaluating models of vehicular environmental impact. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, pp. 269–278, 2012.

Karnin, Z. and Liberty, E. Discrepancy, coresets, and sketches in machine learning. In *Conference on Learning Theory*, pp. 1975–1993, 2019.

Langberg, M. and Schulman, L. J. Universal ε-approximators for integrals. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pp. 598–607. SIAM, 2010.

Lyon, R. J., Stappers, B., Cooper, S., Brooke, J. M., and Knowles, J. D. Fifty years of pulsar candidate selection: from simple filters to a new principled real-time classification approach. *Monthly Notices of the Royal Astronomical Society*, 459(1):1104–1123, 2016.

Maalouf, A., Jubran, I., and Feldman, D. Fast and accurate least-mean-squares solvers. In *Advances in Neural Information Processing Systems*, pp. 8305–8316, 2019.

Maalouf, A., Statman, A., and Feldman, D. Tight sensitivity bounds for smaller coresets. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2051–2061, 2020.

Maalouf, A., Tukan, M., Price, E., Kane, D. M., and Feldman, D. Coresets for data discretization and sine wave fitting. In *International Conference on Artificial Intelligence and Statistics*, pp. 10622–10639. PMLR, 2022.

Minsker, S. Geometric median and robust estimation in banach spaces. *Bernoulli*, 21(4):2308–2335, 2015.

Munteanu, A. and Schwiegelshohn, C. Coresets-methods and history: A theoreticians design pattern for approximation and streaming algorithms. *KI-Künstliche Intelligenz*, 32(1):37–53, 2018.

Munteanu, A., Schwiegelshohn, C., Sohler, C., and Woodruff, D. P. On coresets for logistic regression. *arXiv preprint arXiv:1805.08571*, 2018.

Oliphant, T. E. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

Ozer, D. J. Correlation and the coefficient of determination. *Psychological bulletin*, 97(2):307, 1985.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Phillips, J. M. Coresets and sketches. *arXiv preprint arXiv:1601.00617*, 2016.

Prechelt, L. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pp. 55–69. Springer, 1998.

Prechelt, L. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pp. 55–69. Springer, 2002.

Sampaio, G. S., de Aguiar Vallim Filho, A. R., da Silva, L. S., and da Silva, L. A. Prediction of motor failure time using an artificial neural network. *Sensors*, 19(19): 4342, October 2019. doi: 10.3390/s19194342. URL [WebLink].

Tsanas, A. and Xifara, A. Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools. *Energy and buildings*, 49: 560–567, 2012.

Tukan, M., Maalouf, A., and Feldman, D. Coresets for near-convex functions. *Advances in Neural Information Processing Systems*, 33, 2020.

Tukan, M., Baykal, C., Feldman, D., and Rus, D. On coresets for support vector machines. *Theoretical Computer Science*, 890:171–191, 2021.

Uzilov, A. V., Keegan, J. M., and Mathews, D. H. Detection of non-coding rnas on the basis of predicted secondary structure formation free energy change. *BMC bioinformatics*, 7(1):173, 2006.

Van Rossum, G. and Drake, F. L. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Jarrod Millman, K., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C., Polat, İ., Feng, Y., Moore, E. W., Vand erPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and Contributors, S. . . SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 2020. doi: https://doi.org/10.1038/s41592-019-0686-2.

Yeh, I.-C. and Lien, C.-h. The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. *Expert systems with applications*, 36(2):2473–2480, 2009.

Zhou, W., Xu, C., Ge, T., McAuley, J., Xu, K., and Wei, F. Bert loses patience: Fast and robust inference with early exit. *Advances in Neural Information Processing Systems*, 33:18330–18341, 2020.

A. More details

More on the initialization technique. Initialization using different approximated solutions is a technique commonly used in optimization algorithms, including kmeans (Arthur & Vassilvitskii, 2007). The idea behind this technique is to start the optimization process from different starting points, or initializations, and to use the resulting approximate solutions to improve the overall optimization performance. This is because different initializations may result in different local optima, and by considering multiple initializations, the optimization algorithm may be able to find a better overall solution. In our method, we do not optimize the given approximate solution, but approximating several approximated solutions using our coreset practically moves the coresets towards approximating "good" various regions of the query set, where each of these regions contains a good solution on the dataset. While there is a possibility that the solutions found may be very similar, in practice, the technique tends to provide benefits in terms of improved optimization performance. Practically, we saw that uniform sampling is also sufficient to achieve very good coresets which approximate the optimal solution very well.

More on the stopping criteria. First of all, the intuition behind setting stopping criteria is derived from the theory of training models in deep learning. Specifically speaking, the early stopping technique in deep learning. While we could have set the number of iterations to a hard-coded scalar (e.g., 400), we would have either made a very weak coreset that has been exposed to not enough queries, or we would have extended the running time of the algorithm beyond the limits of being practical. The idea that we have used in the paper is to put a threshold on the number of times the minimal cost so far has not changed thus implying some sort of convergence. Notably and most importantly, the usage of such criteria is intensively justified practically in many experimental papers (see for example, (Prechelt, 2002; Zhou et al., 2020; Gu et al., 2018)) in deep/machine learning.

We also note that the user can use any stopping criterion and of course, the results will change depending on such a choice.

The construction of the query set. We aimed to obtain a coreset that supports a query set that can span a meaningful part of the entire query space. Intuitively speaking, we aim to have a coreset that approximates the loss of a query set containing (i) the optimal solution of the entire data or some fine approximation to it (see next paragraph for an intuitive explanation of how this should intuitively hold) and (ii) the optimal solution on this computed coreset, given a desired problem (e.g., logistic regression). With this in mind, solving the desired problem on our generated coreset will yield a coreset approximating the solution of the entire data up to $O(\varepsilon)$.

Hence, in the ith iteration of our algorithm, we add the solution optimizing the current coreset to the supported set of queries (e.g., optimal logistic regression solution for the current coreset).

Since the coreset is biased towards this solution, we have evaluated the quality of such a solution on the entire data and concatenated such a vector of losses to our matrix of losses (denoted by the matrix $\tilde{\mathcal{M}}$).

This, in turn, means that each time a new query is added to the supported set of queries, the coreset in the next iteration will be adapted to approximate every query in the query set and it will become more generalized, or in a sense a "stronger coreset".

With this in mind, we can initialize our support query set with approximated solutions to the problem (e.g. ε-approximations), so as to ensure a good initial coreset.

B. Proof of Our Theoretical Results

B.1. Proof of Lemma 3.1

Proof. First, observe that by construction of $\tilde{\mathcal{M}}(P, f)$, it holds that for every $x \in \mathcal{X}'$, and $j \in [n]$, there exists an integer $i \in [|\mathcal{X}'|]$ such that

$$\tilde{\mathcal{M}}(P, f)_{j,i} = f\left(p_j, x\right). \tag{2}$$

By Definition 2.1, the pair $(\mathcal{I}, v)$ satisfies that

$$\left\| \sum_{j=1}^{n} \tilde{\mathcal{M}}(P, f)_{j,*} - \sum_{\ell \in \mathcal{I}} v\left(\ell\right) \tilde{\mathcal{M}}(P, f)_{\ell,*} \right\|_2^2 \leq \varepsilon. \tag{3}$$

Note that (3) dictates that for every $k \in [|X'|]$, it holds that

$$\left| \sum_{j \in [n]} \tilde{\mathcal{M}}(P,f)_{j,k} - \sum_{\ell \in \mathcal{I}} v(\ell)\, \tilde{\mathcal{M}}(P,f)_{\ell,k} \right|^2 \leq \varepsilon. \tag{4}$$

Finally, combining (2) and (4) yields Lemma 3.1. □

B.2. Proof of Claim 3.2

Proof. For every $k \in [z]$, denote by x_k the query which corresponds to the kth column of $\tilde{\mathcal{M}}(P,f)$. The claim holds by the following derivations:

$$\begin{aligned}
&\left| \sum_{i=1}^{n} f(p_i,x) - \sum_{j \in \mathcal{I}} v(j) f(p_j,x) \right|^2 \\
&= \left| \sum_{i=1}^{n} \sum_{k=1}^{z} \alpha(k) f(p_i,x_k) - \sum_{j \in \mathcal{I}} v(j) \sum_{k=1}^{z} \alpha(k) f(p_i,x_k) \right|^2 \\
&= \left| \sum_{k=1}^{z} \alpha(k) \sum_{i=1}^{n} f(p_i,x_k) - \sum_{k=1}^{z} \alpha(k) \sum_{j \in \mathcal{I}} v(j) f(p_i,x_k) \right|^2 \\
&= \left| \sum_{k=1}^{z} \alpha(k) \left(\sum_{i=1}^{n} f(p_i,x_k) - \sum_{j \in \mathcal{I}} v(j) f(p_i,x_k) \right) \right|^2 \\
&\leq \left| \sum_{k=1}^{z} \alpha(k) \sqrt{\varepsilon} \right|^2 = \left| \sqrt{\varepsilon} \right|^2 \leq \varepsilon,
\end{aligned}$$

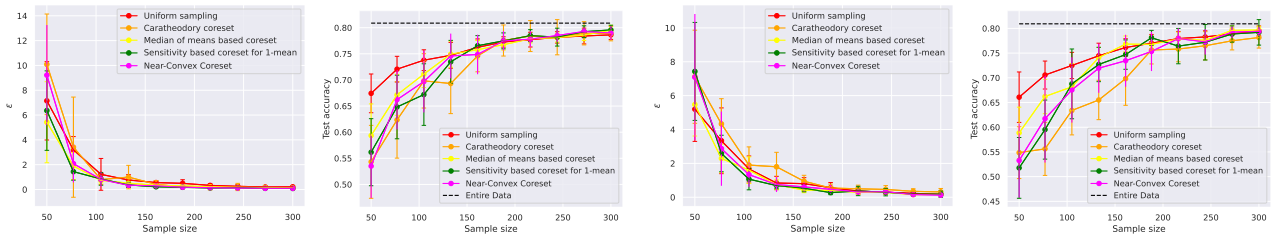
where the first equality hold by the definition of x, the second and thirds are simple rearrangements, the first inequality holds by Claim 3.2. □

C. Experimental Results

In this section, we dive into exploring the effect of the actions/parameters used in *AutoCore*.

C.1. Taking the last coreset

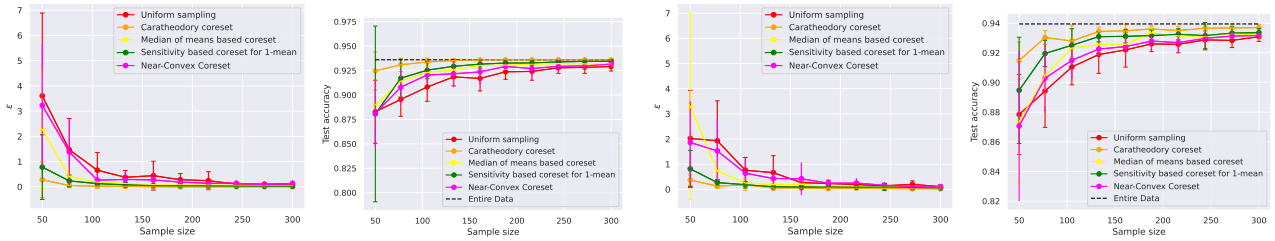
In what follows, we show the results of using the last coresets *AutoCore* has devised, i.e., as Algorithm 1 suggests.



(a) Logistic regression (b) SVMs

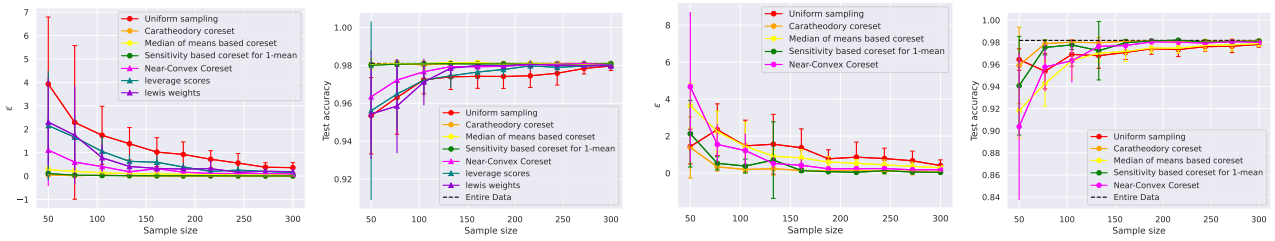
Figure 10. Evaluation of our coresets against Uniform sampling on the Dataset (i).

As depicted throughout Figures C.2–12, we observe that *AutoCore* output coresets that outperform the competing methods almost in all of our experiments. In some, we observe that the desired behavior of our coreset gets delayed (takes 2 to 3



(a) Logistic regression (b) SVMs

Figure 11. Evaluation of our coresets against Uniform sampling on the Dataset (ii).



(a) Logistic regression (b) SVMs

Figure 12. Evaluation of our coresets against Uniform sampling on the Dataset (iii).

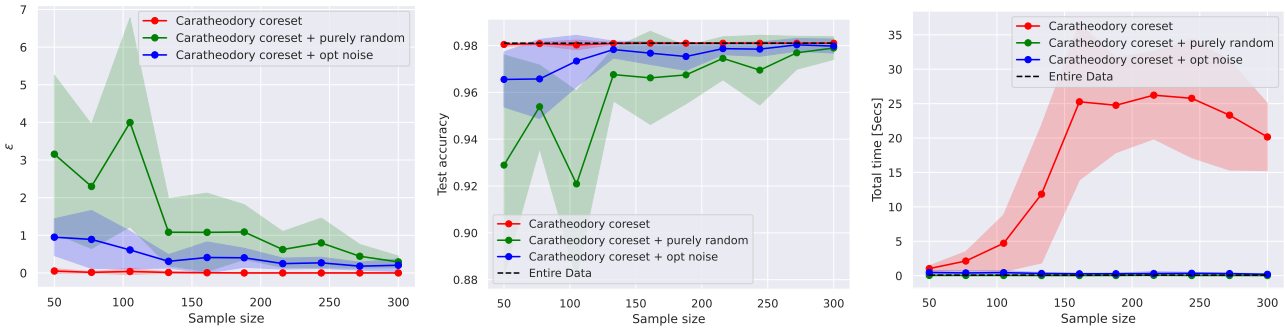
samples to outperform the rest of the competitors). This is due to the fact that taking such coresets means that the coreset is becoming more general, thus requiring a larger sample size to guarantee better approximation, one needs to sample more. Such behavior does not appear in our "optimal coresets" where we have taken the coreset with the optimal cost; see Figures 3– 5. The reason for this is that the optimal coreset has been exposed to fewer models/queries than the coreset that would be output by the plain *AutoCore*, and thus the need for a larger sample size for smaller approximation error becomes less demanding.

C.2. Exploration of different algorithms for choosing queries

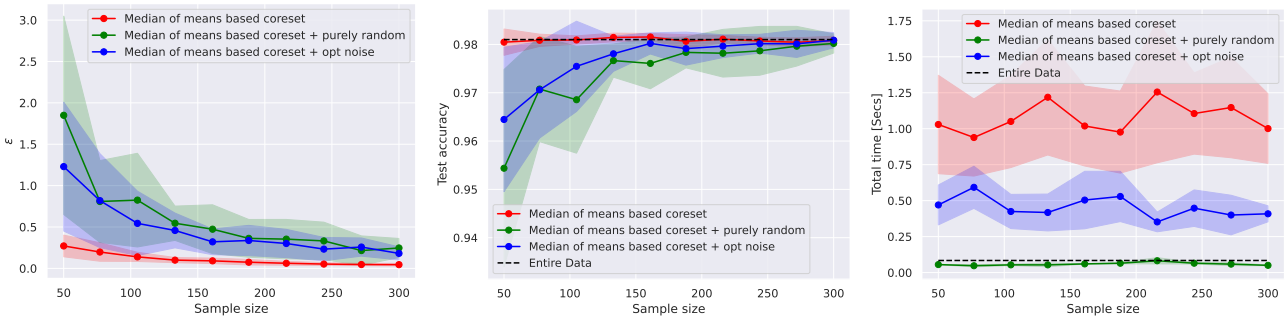
In what follows, we show the effect of different methods for choosing the next query for our practical coreset paradigm with respect to the logistic regression problem.

C.3. Experimenting with Cifar10 and TinyImageNet

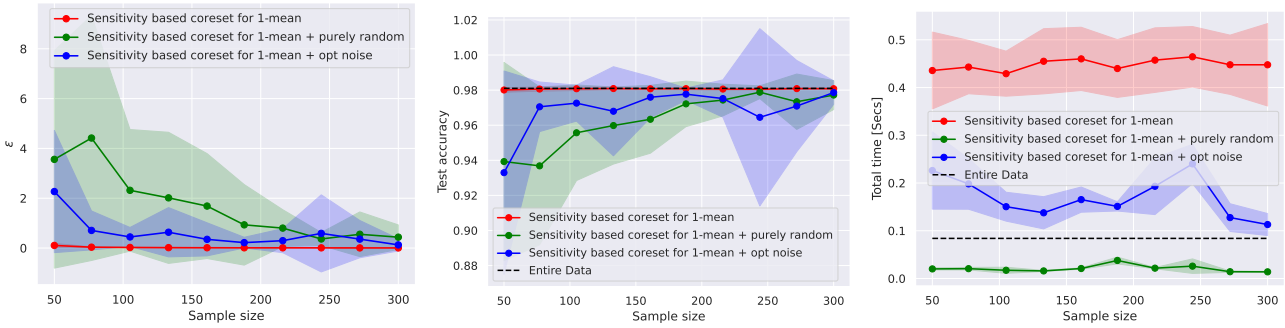
In what follows, we run our coreset paradigm on Cifar10 and TinyImageNet. For TinyImageNet data, we had to use the JL-lemma to reduce the dimensionality of the data. As seen from Figure C.3, our coreset construction technique yields better coresets than uniform sampling even for large-scale datasets, where our coreset can be better than uniform sampling by at max ≈ 1.5 times in terms of relative approximation error.



(a) Results when using Carathéodory as our coreset inner construction

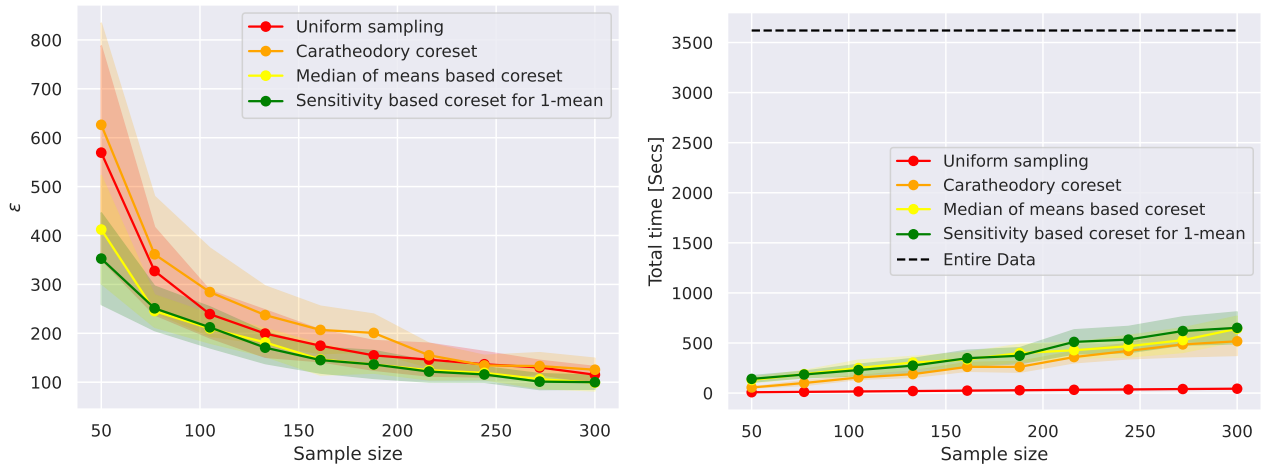


(b) Median of means

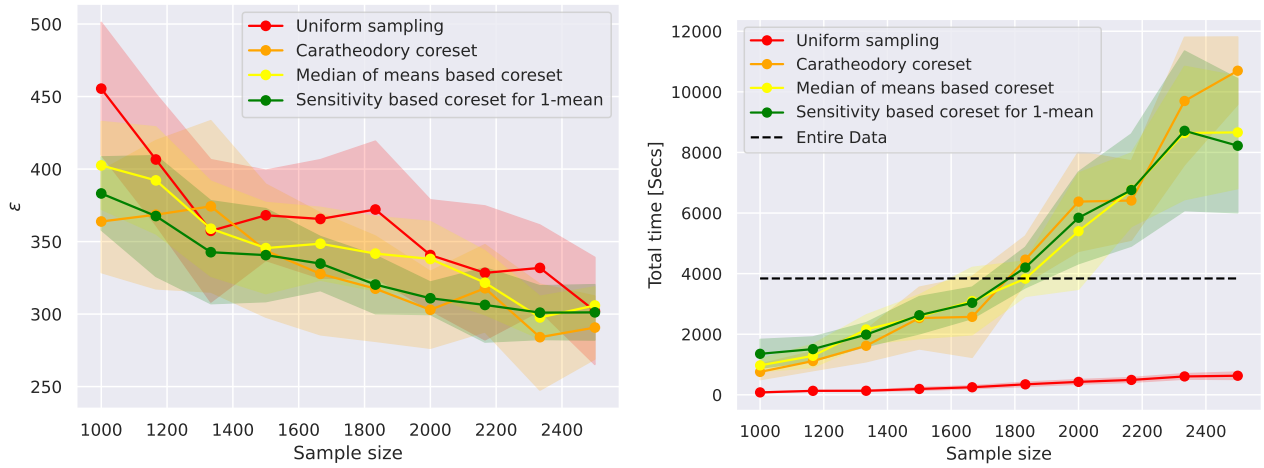


(c) Results when using Sensitivity for 1-means as our coreset inner construction

Figure 13. Evaluation of our coresets with different algorithms for choosing the next query.



(a) Cifar-10



(b) TinyImageNet

Figure 14. Evaluation of our coreset on large-scale datasets.