Fast and Accurate Least-Mean-Squares Solvers

Alaa Maalouf *
Alaamalouf12@gmail.com

Ibrahim Jubran*
ibrahim.jub@gmail.com

Dan Feldman
dannyf.post@gmail.com

The Robotics and Big Data Lab,
Department of Computer Science,
University of Haifa,
Haifa, Israel

Abstract

Least-mean squares (LMS) solvers such as Linear / Ridge / Lasso-Regression, SVD and Elastic-Net not only solve fundamental machine learning problems, but are also the building blocks in a variety of other methods, such as decision trees and matrix factorizations.

We suggest an algorithm that gets a finite set of n d-dimensional real vectors and returns a weighted subset of $d+1$ vectors whose sum is *exactly* the same. The proof in Caratheodory's Theorem (1907) computes such a subset in $O(n^2d^2)$ time and thus not used in practice. Our algorithm computes this subset in $O(nd)$ time, using $O(\log n)$ calls to Caratheodory's construction on small but "smart" subsets. This is based on a novel paradigm of fusion between different data summarization techniques, known as sketches and coresets.

As an example application, we show how it can be used to boost the performance of existing LMS solvers, such as those in scikit-learn library, up to x100. Generalization for streaming and distributed (big) data is trivial. Extensive experimental results and complete open source code are also provided.

1 Introduction and Motivation

Least-Mean-Squares (LMS) solvers are the family of fundamental optimization problems in machine learning and statistics that include linear regression, Principle Component Analysis (PCA), Singular Value Decomposition (SVD), Lasso and Ridge regression, Elastic net, and many more Golub und Reinsch (1971); Jolliffe (2011); Hoerl und Kennard (1970); Seber und Lee (2012); Zou und Hastie (2005); Tibshirani (1996); Safavian und Landgrebe (1991). See formal definition below. First closed form solutions for problems such as linear regression were published by e.g. Pearson Pearson (1900) around 1900 but were probably known before. Nevertheless, today they are still used extensively as building blocks in both academy and industry for normalization Liang u. a. (2013); Kang u. a. (2011); Afrabandpey u. a. (2016), spectral clustering Peng u. a. (2015), graph theory Zhang und Rohe (2018), prediction Copas (1983); Porco u. a. (2015), dimensionality reduction Laparra u. a. (2015), feature selection Gallagher u. a. (2017) and many more; see more examples in Golub und Van Loan (2012).

Least-Mean-Squares solver in this paper is an optimization problem that gets as input an $n \times d$ real matrix A, and another n-dimensional real vector b (possibly the zero vector). It aims to minimize the sum of squared distances from the rows (points) of A to some hyperplane that is represented by its normal or vector of d coefficients x, that is constrained to be in a given set $X \subseteq \mathbb{R}^d$:

$$\min_{x \in X} f(\|Ax - b\|_2) + g(x).$$

*These authors contributed equally to this work.

Here, g is called a *regularization term*. For example: in linear regression $X = \mathbb{R}^d$, $f(x) = x^2$ and $g(x) = 0$ for every $x \in X$. In Lasso $f(y) = y^2$ and $g(y) = \alpha \cdot \|x\|_1$ for every $y \in \mathbb{R}^d$ and $\alpha > 0$. Such LMS solvers can be computed via the covariance matrix $A^T A$. For example, the solution to linear regression of minimizing $\|Ax - b\|_2$ is $(A^T A)^{-1} A^T b$.

1.1 Related work

While there are many LMS solvers and implementations, there is always a trade-off between their accuracy and running time; see comparison table in Bauckhage (2015) with references therein. The reason is related to the fact that computing the covariance matrix of A can be done essentially in two ways: (i) summing the $d \times d$ outer product $a_i a_i^T$ of the ith row a_i^T of A over every i, $1 \leq i \leq n$. This is due to the fact that $A^T A = \sum_{i=1}^{n} a_i a_i^T$, or (ii) factorization of A, e.g. using SVD or the QR decomposition.

Numerical issues. Method (i) is easy to implement for streaming rows of A by maintaining only d^2 entries of the covariance matrix for the n vectors seen so far, or maintaining its inverse $(A^T A)^{-1}$ as explained e.g. in Golub und Van Loan (2012). This takes $O(d^2)$ for each vector insertion. However, every such addition may introduce another numerical error which accumulates over time. This error increases significantly when running the algorithms using 32 bit floating point representation, which is common for GPU computations. This solution is similar to maintaining the set of d rows of the matrix DV^T, where $A = UDV^T$ is the SVD of A, which is not a subset of the original input matrix A but has the same covariance matrix $A^T A = VD^2V$. A common problem is that to compute $(A^T A)^{-1}$ the matrix $A^T A$ must be invertible. This may not be the case due to numerical issues. In algorithms such as Lasso, the input cannot be a covariance matrix, but can be its Cholesky decomposition Bjorck (1967) VD. However, Cholesky decomposition can be applied only on positive-definite matrices, which is not the case even for small numerical errors that are added to $A^T A$. See Section 4 for more details and empirical evidence.

Running-time issues. Method (ii) above utilizes factorizations such as SVD, i.e., $A = UDV^T$ to compute the covariance matrix via $A^T A = VD^2V^T$ or the QR decomposition $A = QR$ to compute $A^T A = R^T Q^T Q R^T = R^T R$. This approach is known to be much more stable. However, it is much more time consuming: while in theory the running time is $O(nd^2)$ as in the first method, the constants that are hidden in the $O(\cdot)$ notation are significantly larger. Moreover, unlike Method (i), it is impossible to compute such factorizations exactly for streaming data Clarkson und Woodruff (2009).

Caratheodory's Theorem Carathéodory (1907) states that every point contained in the convex hull of n points in $\mathbb{R}^d$ can be represented as a convex combination of a subset of at most $d + 1$ points, which we call the *Caratheodory set*; see Section 2 and Fig. 1. This implies that we can maintain a weighted (scaled) set of $d^2 + 1$ points (rows) whose covariance matrix is the same as A, since $(1/n) \sum_i a_i a_i^T$ is the mean of n matrices and thus in the convex hull of their corresponding points in $\mathbb{R}^{d^2}$; see Algorithm 3. The fact that we maintain such a small sized subset of points instead of updating linear combinations of all the n points seen so far, significantly reduces the numerical errors as shown in Fig. 10b–10a. Unfortunately, computing this set from Caratheodory's Theorem takes $O(n^2 d^2)$ or $O(nd^3)$ time via $O(n)$ calls to an LMS solver. This fact makes it non-practical to use in an LMS solver, as we aim to do in this paper, and may explain the lack of source code for this algorithm on the web.

Approximations via Coresets and Sketches. In the recent decades numerous approximation and data summarization algorithms were suggested to *approximate* the covariance matrix or its singular values. This is by computing a small matrix S whose covariance $S^T S$ approximates, in some sense, the covariance matrix $A^T A$ of the input data A. The term *coreset* is usually used when S is a weighted (scaled) subset of rows from the n rows of the input matrix. The matrix S is sometimes called a *sketch* if each rows in S is a linear combination of few or all its rows, i.e. $S = WA$ for some matrix $W \in \mathbb{R}^{s \times n}$. However, those coresets and sketches usually yield $(1 + \varepsilon)$-multiplicative approximations for $\|Ax\|_2^2$ by $\|Sx\|_2^2$ where the matrix S is of $(d/\varepsilon)^{O(1)}$ rows and x may be any vector, or the smallest/largest singular vector of S or A; see lower bounds in Feldman u. a. (2010).

Moreover, a $(1+\varepsilon)$-approximation to $\|Ax\|_2^2$ by $\|Sx\|_2^2$ does not guarantee an approximation to the actual entries or eigenvectors of A by S that may be very different; see Drineas u. a. (2006).

Accurately handling big data. The algorithms in this paper return *accurate* coresets ($\varepsilon = 0$), which is less common in the literature, including computations of the exact covariance matrix $A^T A$. Such coresets can easily support infinite stream of input rows using memory that is *linear* in their size, and also support dynamic/distributed data in parallel. This is by the useful merge-and-reduce property of coresets that allow them to handle big data; see details e.g. in Agarwal u. a. (2004). Unlike traditional coresets that pay additional logarithmic multiplicative factors due to the usage of merge-reduce trees and increasing error, the suggested weighted subsets in this paper do not introduce additional error to the resulting compression since they preserve the result accurately. The actual numerical errors are measured in the experimental results, with analysis that explain the differences.

A main advantage of a coreset over a sketch is that it preserves sparsity of the input rows Feldman u. a. (2016), which usually reduces theoretical running time. Our experiments show, as expected, that coresets can also be used to significantly improve the numerical stability of existing algorithms, even if the running time is the same. Another advantage is that the same coreset can be used for parameter tuning over a large set of candidates. In addition to other reasons, this significantly reduced the running time of such algorithms in our experiments; see Section 4.

1.2 Our contribution

A natural question that follows from the previous section is: *can we maintain the optimal solution for LMS problems both accurately and fast?* We answer this question affirmably by suggesting:

(i) the first algorithm that computes the Caratheodory set of n input points in time that is linear in the input $O(nd)$ for asymptotically large n, and using only $O(\log n)$ calls to an LMS solver. This is by using a novel approach of coreset/skecthes fusion that is explained in the next section; see Algorithm 2 and Theorem 2.

(ii) an algorithm that maintains a ("coreset") matrix $S \in \mathbb{R}^{(d^2+1)\times d}$ such that: (a) its set of rows is a weighted subset of the matrix $A \in \mathbb{R}^{n\times d}$ whose rows are the input points, and (b) the covariance matrices of S and A are the same, i.e., $S^T S = A^T A$; see Algorithm 3 and Theorem 3.2.

(iii) example applications for boosting the performance of *existing* solvers by running them on the matrix S above or its variants for Linear/Ridge/Lasso Regressions and Elastic-net.

(iv) extensive experimental results on synthetic and real-world data for common LMS solvers of Scikit-learn library with either CPython or Intel's distribution. Either the running time or numerical stability is improved up to two orders of magnitude.

(v) Open code Maalouf u. a. (2019) for our algorithms that we hope will be used for the many other LMS solvers and future research as suggested in our Conclusion section; see Section 5.

1.3 Novel approach: Coresets meet Sketches

As explained in Section 1.1, the covariance matrix $A^T A$ of A itself can be considered as a sketch which is relatively less numerically stable to maintain (especially its inverse, as desired by e.g. linear regression). The Caratheodory set that corresponds to the outer products of the rows of A is a coreset whose covariance matrix is $A^T A$ and, as a weighted subset of the original rows, is more numerically stable but takes much more time to compute; see Theorem 2.2.

We thus suggest a meta-algorithm that combines these two approaches: sketches and coresets. It may be generalized to other, not-necessarily accurate ε-coresets and sketches ($\varepsilon > 0$); see Section 5.

The input to our meta-algorithm is 1) a set P of n items, 2) an integer k from 1 to n where n is maximum accuracy but longest running time, and 3) a pair of coreset and sketch construction schemes for the problem at hand.
The output is a coreset for the problem whose construction time is faster; see Fig. 1.

Step I: Compute a balanced partition $\{P_1, \cdots, P_k\}$ of the input set P into k clusters of roughly the same size. While the correctness holds for any such arbitrary partition (e.g. see Algorithm 3.1), to reduce numerical errors – a partition that minimizes the sum of loss with respect to the problem at hand would be optimal.

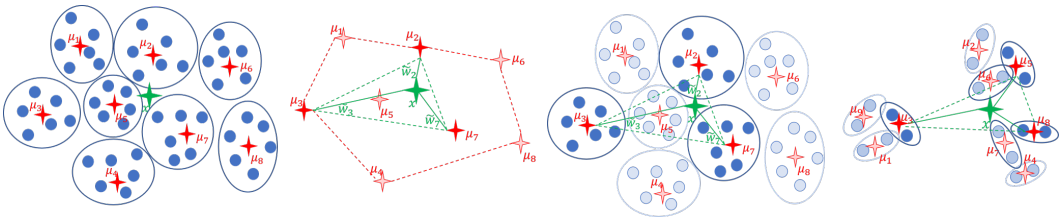


Figure 1: Overview of Algorithm 2 and the steps in Section 1.3. Images left to right: Steps I and II (Partition and sketch steps): A partition of the input weighted set of $n = 48$ points (in blue) into $k = 8$ equal clusters (in circles) whose corresponding means are $\mu, \ldots, \mu_8$ (in red). The mean of P (and these means) is x (in green). Step III (Coreset step): Caratheodory (sub)set of $d + 1 = 3$ points (bold red) with corresponding weights (in green) is computed only for these $k = 8 \ll n$ means. Step IV (Recover step): the Caratheodory set is replaced by its corresponding original points (dark blue). The remaining points in P (bright blue) are deleted. Step V (Recursive step): Previous steps are repeated until only $d + 1 = 3$ points remains. This takes $O(\log n)$ iterations for $k = \Theta(d)$.

Step II: Compute a sketch S_i for each cluster P_i, where $i \in \{1, \cdots, k\}$, using the input sketch scheme. This step does not return a subset of P as desired, and is usually numerically less stable.

Step III: Compute a coreset B for the union $S = S_1 \cup \cdots S_k$ of sketches from Step II, using the input coreset scheme. Note that B is not a subset (or coreset) of P.

Step IV: Compute the union C of clusters in $P_1, \cdots, P_k$ that correspond to the selected sketches in Step III, i.e. $C = \bigcup_{S_i \in B} P_i$. By definition, C is a coreset for the problem at hand.

Step V: Recursively compute a coreset for C until a sufficiently coreset is obtained. This step is used to reduce running time, without selecting k that is too small.

We then run an existing solver on the coreset C to obtain a faster accurate solution for P. Algorithm 2 and 3.1 are special cases of this meta-algorithm, where the sketch is simply the sum of a set of points/matrices, and the coreset is the existing (slow) implementation of the Caratheodory set from Theorem 2.2.

Paper organization. In Section 2 we give our notations, definitions and the current state-of-the-art result. Section 3 presents our main algorithms for efficient computation of the Caratheodory (core-)set and a subset that preserves the inputs covariance matrix, their theorems of correctness and proofs. Section 4 demonstrates the applications of those algorithms to common LMS solvers, with extensive experimental results on both real-world and synthetic data via the Scikit-learn library with either CPython or Intel's Python distributions. We conclude the paper with open problems and future work in Section 5.

2 Notation and Preliminaries

For integers $n, d \geq 1$, we denote by $\mathbb{R}^{n \times d}$ the set of $n \times d$ real matrices, and $[n] = \{1, \cdots, n\}$. To avoid abuse of notation, we use the big O notation where $O(\cdot)$ is a set Cormen u. a. (2009). A *weighted set* is a pair (P, u) where $P = \{p_1, \cdots, p_n\}$ is an ordered finite set in $\mathbb{R}^d$, and $u : P \to [0, \infty)$ is a positive *weights function*. A *linear system solver* is an algorithm that solves a system of n linear equations with d variables, i.e., return $x \in \mathbb{R}^d$ such that $Ax = b$ for a given $A \in \mathbb{R}^{n \times d}$ and $b \in \mathbb{R}^n$, assuming there is such a solution.

Given a point q inside the convex hull of a set of points P, Caratheodory's Theorem proves that there a subset of at most $d + 1$ points in P whose convex hull also contains q. This geometric definition can be formulated as follows.

Definition 2.1 (Caratheodory set)**.** *Let (P, u) be a weighted set of n points in $\mathbb{R}^d$ such that $\sum_{p \in P} u(p) = 1$. A weighted set (S, w) is called a* Caratheodory Set *for (P, u) if: (i) its size is $|S| \leq d + 1$, (ii) its weighted mean is the same, $\sum_{p \in S} w(p) \cdot p = \sum_{p \in P} u(p) \cdot p$, and (iii) its sum of weights $\sum_{p \in S} w(p) = 1$.*

Caratheodory's Theorem suggests a constructive proof for computing this set in $O(n^2 d^2)$ time Carathéodory (1907); Cook und Webster (1972). This is implemented in Algorithm 1, which takes as input a weighted set (P, u) such that $\sum_{p \in P} u(p) = 1$ and computes a Caratheodory set (S, w) for (P, u) in $O(n^2 d^2)$ time. However, as observed e.g. in Nasser u. a. (2015) it can be com-

puted only for the first $m = d + 1$ points, and then be updated point by point in $O(md^2) = O(d^3)$ time per point, to obtain $O(nd^3)$ overall time. This still takes $\Theta(n)$ calls to a linear system solver that solves $Ax = b$ for a given matrix A and vector b, each of $\Theta(d)$ rows and columns, in $O(d^3)$ time per call.

Theorem 2.2 (Carathéodory (1907))**.** *A Caratheodory set (S, w) can be computed for any weighted set (P, u) where $\sum_{p \in P} u(p) = 1$ in $t(n, d) \in O(1) \cdot \min\left\{n^2d^2, nd^3\right\}$ time.*

Algorithm 1 CARATHEODORY(P, u)

Input : A weighted set (P, u) of n points in $\mathbb{R}^d$.
Output: A Caratheodory set (S, w) for (P, u) in $O(n^2d^2)$ time.

1 **if** $n \leq d + 1$ **then**
2 | **return** (P, u)
3 **for** *every* $i \in \{2, \cdots, n\}$ **do**
4 | $a_i := p_i - p_1$
5 $A := (a_2 \mid \cdots \mid a_n)$ // $A \in \mathbb{R}^{d \times (n-1)}$
6 Compute $v = (v_2, \cdots, v_n)^T \neq 0$ such that $Av = 0$.
7 $v_1 := -\sum_{i=2}^{n} v_i$
8 $\alpha := \min\left\{\frac{u_i}{v_i} \mid i \in \{1, \cdots, n\} \text{ and } v_i > 0\right\}$
9 $w_i := (u_i - \alpha v_i)$ for every $i \in \{1, \cdots, n\}$ s.t. $w_i > 0$.
10 $S := \{p_i \mid w_i > 0 \text{ and } i \in \{1, \cdots, n\}\}$
if $|S| > d + 1$ **then**
11 | $(S, w) :=$ CARATHEODORY(S, w)
12 **return** (S, w)

3 Faster Caratheodory Set

In this section, we present our main algorithm that reduces the running time for computing a Caratheodory set from $O(\min\left\{n^2d^2, nd^3\right\})$ in Theorem 2.2 or Nasser u. a. (2015) to $O(nd)$ for sufficiently large n; see Theorem 3.1. A visual illustration of the corresponding Algorithm 2 is shown in Fig 1. We also present a second algorithm, called CARATHEODORY-MATRIX, which computes a small weighted subset of a the given input that has the same covariance matrix as the input data; see Algorithm 3.

Theorem 3.1 (Caratheodory-Set Booster)**.** *Let (P, w) be a weighted set of n points in $\mathbb{R}^d$, and $k \geq d + 2$ be an integer. Let (C, u) be the output of a call to* FAST-CARATHEODORY-SET(P, u, k)*; See Algorithm 2. Let $t(k, d)$ be the time it takes to compute a Caratheodory Set for k points in $\mathbb{R}^d$, as in Theorem 2.2. Then (C, u) is a Caratheodory set of (P, w) that can be computed in time*

$$O\left(nd + t(k, d) \cdot \frac{\log n}{\log(k/d)}\right).$$

Proof. We use the notation and variable names as defined in Algorithm 2.

Identify the input set $P = \{p_1, \cdots, p_n\}$ and the set C that is computed at Line 6 of Algorithm 2 as $C = \left\{c_1, \cdots, c_{|C|}\right\}$. We will first prove that the weighted set (C, w) computed in Lines 6–8 at the current (some) iteration is a Caratheodory set for (P, u), i.e., $\sum_{p \in P} u(p) \cdot p = \sum_{p \in C} w(p) \cdot p$, $\sum_{p \in P} u(p) = \sum_{p \in C} w(p)$ and $|C| \leq d + 1$.

For every $i \in \{1, \cdots, k\}$, let $U(P_i) = \sum_{p \in P_i} u(p)$.

Let $(\tilde{\mu}, \tilde{w})$ be the pair computed at the current iteration at Line 5. By the definition of CARATHEODORY, we have that the pair $(\tilde{\mu}, \tilde{w})$ computed at Line 5 is a Caratheodory set of the

weighted set $(\{\mu_1, \cdots, \mu_k\}, u')$, i.e., it satisfies that

$$\sum_{\mu_i \in \tilde{\mu}} \tilde{w}(\mu_i) = 1, \quad \sum_{\mu_i \in \tilde{\mu}} \tilde{w}(\mu_i)\mu_i = \sum_{i=1}^{k} U(P_i) \cdot \mu_i \text{ and } \quad |\tilde{\mu}| \leq d+1. \tag{1}$$

Observe that by the definition of μ_i for every $i \in \{1, \cdots, k\}$ at Line 4 we have that

$$\sum_{i=1}^{k} U(P_i) \cdot \mu_i = \sum_{i=1}^{k} U(P_i) \cdot \left(\frac{1}{U(P_i)} \cdot \sum_{p \in P_i} u(p) \cdot p \right) = \sum_{i=1}^{k} \sum_{p \in P_i} u(p)p = \sum_{p \in P} u(p)p. \tag{2}$$

We now have that

$$\begin{aligned} \sum_{p \in C} w(p)p &= \sum_{\mu_i \in \tilde{\mu}} \sum_{p \in P_i} \frac{\tilde{w}(\mu_i)u(p)}{U(P_i)} \cdot p = \sum_{\mu_i \in \tilde{\mu}} \tilde{w}(\mu_i) \sum_{p \in P_i} \frac{u(p)}{U(P_i)} p = \sum_{\mu_i \in \tilde{\mu}} \tilde{w}(\mu_i)\mu_i \\ &= \sum_{i=1}^{k} U(P_i) \cdot \mu_i = \sum_{p \in P} u(p)p. \end{aligned} \tag{3}$$

where the first equality holds by the definitions of C and w, the third equality holds by the definition of μ_i at Line 4, the fourth equality is by (1), and the last equality is by (2).

We also have that the new sum of weights is equal to

$$\sum_{p \in C} w(p) = \sum_{\mu_i \in \tilde{\mu}} \sum_{p \in P_i} \frac{\tilde{w}(\mu_i)u(p)}{U(P_i)} = \sum_{\mu_i \in \tilde{\mu}} \frac{\tilde{w}(\mu_i)}{U(P_i)} \cdot \sum_{p \in P_i} u(p) = \sum_{\mu_i \in \tilde{\mu}} \frac{\tilde{w}(\mu_i)}{U(P_i)} \cdot U(P_i) = \sum_{\mu_i \in \tilde{\mu}} \tilde{w}(\mu_i) = 1. \tag{4}$$

Combining (3) and (4) yields that the weighted (C, w) computed before the recursive call at Line 10 of the algorithm is a Caratheodory set for the weighted input set (P, u). Since at each iteration we either return such a Caratheodory set (C, w) at Line 10 or return the input weighted set (P, u) itself at Line 1, by induction we get that the output weighted set of a call to FAST-CARATHEODORY-SET(P, u, k) is a Caratheodory set for the original input (P, u).

By (1) we have that C contains at most $|C| \leq (d+1) \cdot \frac{n}{k} = n \cdot \frac{d+1}{k}$ points. Hence, there are at most $\log_{\frac{k}{d+1}}(n)$ recursive calls before the stopping condition at line 1 is met. The time complexity of each iteration is $n' + t(k, d)$ where $n' = |P| \cdot d$ is the number of points in the current iteration. Thus the total time complexity is

$$\sum_{i=1}^{\log(n)} \frac{nd}{2^{i-1}} + t(k,d) \leq 2nd + \log_{\frac{k}{d+1}}(n) \cdot t(k,d) \in O\left(nd + \frac{\log n}{\log(k/(d+1))} \cdot t(k,d) \right).$$

□

Theorem 3.2. *Let $A \in \mathbb{R}^{n \times d}$ be a matrix, and $k \geq d^2 + 2$ be an integer. Let $S \in \mathbb{R}^{(d^2+1) \times d}$ be the output of a call to* CARATHEODORY-MATRIX(A, k)*; see Algorithm 3. Let $t(k, d)$ be the computation time of* CARATHEODORY *given k point in $\mathbb{R}^{d^2}$. Then S satisfies that $A^T A = S^T S$. Furthermore, S can be computed in $O(nd^2 + t(k, d^2) \cdot \frac{\log n}{\log(k/d^2))})$ time.*

Proof. We use the notation and variable names as defined in Algorithm 3.

Since (C, w) at Line 5 of Algorithm 3 is the output of a call to FAST-CARATHEODORY-SET(P, u, k), by Theorem 3.1 we have that: (i) the weighted means of (C, w) and (P, u) are equal, i.e.,

$$\sum_{p \in P} u(p) \cdot p = \sum_{p \in C} w(p) \cdot p, \tag{5}$$

(ii) $|C| \leq d^2 + 1$ since $P \subseteq \mathbb{R}^{(d^2)}$, and (iii) C is computed in $O(nd^2 + \log_{\frac{k}{d^2+1}}(n) \cdot t(k, d^2))$ time.

Algorithm 2 FAST-CARATHEODORY-SET(P, u, k); see Theorem 3.1

Input: A set P of n points in $\mathbb{R}^d$, a (weight) function $u : P \to [0, \infty)$ such that $\sum_{p \in P} u(p) = 1$, and an integer (number of clusters) $k \in \{1, \cdots, n\}$ for the accuracy/speed trade-off.

Output: A Caratheodory set of (P, u); see Definition 2.1.

1 **if** $n \leq d + 1$ **then**
  return (P, u) `// n = |P| is already small`

2 $\{P_1, \cdots, P_k\} :=$ a partition of P into k disjoint subsets (clusters), each contains at most n/k points.

3 **for** *every* $i \in \{1, \cdots, k\}$ **do**
  $u'(\mu_i) := \sum_{p \in P_i} u(p)$ `// The weight of the ith cluster.`
4 $\mu_i := \frac{1}{u'(\mu_i)} \cdot \sum_{p \in P_i} u(p) \cdot p$ `// the weighted mean of` P_i

5 $(\tilde{\mu}, \tilde{w}) :=$ CARATHEODORY$(\{\mu_1, \cdots, \mu_k\}, u')$ `// see Algorithm 1 and Theorem 2.2.`

6 $C := \bigcup_{\mu_i \in \tilde{\mu}} P_i$

7 **for** *every* $\mu_i \in \tilde{\mu}$ *and* $p \in P_i$ **do**
8 $w(p) := \frac{\tilde{w}(\mu_i) u(p)}{\sum_{p \in P_i} u(p)}$ `// assign weight for each point in` C

9 $(C, w) :=$ FAST-CARATHEODORY-SET(C, w, k) `// recursive call`

10 **return** (C, w)

Algorithm 3 CARATHEODORY-MATRIX(A, k); see Theorem 3.2

Input : A matrix $A = (a_1 \mid \cdots \mid a_n)^T \in \mathbb{R}^{n \times d}$ and an integer $k \in \{1, \cdots, n\}$ that denotes accuracy/speed trade-off.

Output: A matrix $S \in \mathbb{R}^{(d^2+1) \times d}$ whose union of rows is a weighted subset of A, and $A^T A = S^T S$.

1 **for** *every* $i \in \{1 \cdots, n\}$ **do**
2 Set $p_i \in \mathbb{R}^{(d^2)}$ as the concatenation of the d^2 entries of $a_i a_i^T \in \mathbb{R}^{d \times d}$.
    `// The order of entries may be arbitrary but the same for all points.`
3 $u(p_i) := 1/n$

4 $P := \{p_i \mid i \in \{1, \cdots, n\}\}$ `//` P `is a set of` n `vectors in` $\mathbb{R}^{(d^2)}$.

5 $(C, w) :=$ FAST-CARATHEODORY-SET(P, u, k) `//` $C \subseteq P$ `and` $|C| = d^2 + 1$ `by Theorem 3.1`

6 $S :=$ a $(d^2 + 1) \times d$ matrix whose ith row is $\sqrt{n \cdot w(p_i)} \cdot a_i^T$ for every $p_i \in C$.

7 **return** S

Combining (5) with the fact that p_i is simply the concatenation of the entries of $a_i a_i^T$, we get that

$$\sum_{p_i \in P} u(p_i) a_i a_i^T = \sum_{p_i \in C} w(p_i) \cdot a_i a_i^T. \tag{6}$$

By the definition of S on Line 6, we have that

$$S^T S = \sum_{p_i \in C} (\sqrt{n \cdot w(p_i)} \cdot a_i)(\sqrt{n \cdot w(p_i)} \cdot a_i)^T = n \cdot \sum_{p_i \in C} w(p_i) \cdot a_i a_i^T. \tag{7}$$

We also have that

$$A^T A = \sum_{i=1}^{n} a_i a_i^T = n \cdot \sum_{p_i \in P} (1/n) a_i a_i^T = n \cdot \sum_{p_i \in P} u(p_i) a_i a_i^T, \tag{8}$$

where the second derivation holds since $u \equiv 1/n$.

Theorem 3.2 now holds by combining (6), (7) and (8). □

4 Experimental Results

In this section we apply our fast construction of the Carathoodory Set S from previous section to boost the running time of common LMS solvers in Table 1 by a factor of ten to hundreds, or to

Solver	Objective function	Python's Package	Example Python's solver
Linear Regression	$\|Ax-b\|_2^2$	`scipy.linalg`	`lstsq`(A,b)
Ridge Regression	$\|Ax-b\|_2^2+\alpha\|x\|_2^2$	`sklearn.linear_model`	`RidgeCV`$(A,b,\mathbb{A},m)$
Lasso Regression	$\frac{1}{2n}\|Ax-b\|_2^2+\alpha\|x\|_1$	`sklearn.linear_model`	`LassoCV`$(A,b,\mathbb{A},m)$
Elastic-Net Regression	$\frac{1}{2n}\|Ax-b\|_2^2+\rho\alpha\|x\|_2^2+\frac{(1-\rho)}{2}\alpha\|x\|_1$	`sklearn.linear_model`	`ElasticNetCV`$(A,b,\mathbb{A},\rho,m)$

Table 1: The four example solvers that were applied on the LMS-coreset in Algorithm 4. Each gets a matrix $A \in \mathbb{R}^{n\times d}$, a vector $b \in \mathbb{R}^d$ and aims to compute $x \in \mathbb{R}^d$ that minimizes the objective function. Additional regularization parameters include $\alpha > 0$ and $\rho \in [0,1]$. The Python's solvers use m-fold cross validation over every α in a given set $\mathbb{A} \subseteq [0,\infty)$.

improve their numerical accuracy by a similar factor to support, e.g. 32 bit floating point representation. This is by running the given solver as a black box on the small matrix C that is returned by Algorithms 5–8, which is based on S. That is, our algorithm does not compete with existing solvers but relies on them, which is why we called it "booster". Open code for our algorithms is provided Maalouf u. a. (2019).

From Caratheodory Matrix to LMS solvers. As stated in Theorem 3.2, Algorithm 3 gets an input matrix $A \in \mathbb{R}^{n\times d}$ and an integer $k > d+1$, and returns a matrix $S \in \mathbb{R}^{(d^2+1)\times d}$ of the same covariance $A^TA = S^TS$, where k is a parameter for setting the desired accuracy. To "learn" a given label vector $b \in \mathbb{R}^n$, Algorithm 4 partitions the matrix $A' = (A \mid b)$ into m partitions, computes using Algorithm 3 a subset for each partition that preserves its covariance matrix, and returns the union of subsets as a pair (C,y) where $C \in \mathbb{R}^{(m(d+1)^2+m)\times d}$ and $y \in \mathbb{R}^{m(d+1)^2+m}$. For $m=1$, it is easy to see that for every $x \in \mathbb{R}^d$,

$$\|Ax-b\| = \left\|A'(x \mid -1)^T\right\| = \left\|S(x \mid -1)^T\right\| = \left\|(C \mid y)(x \mid -1)^T\right\| = \|Cx-y\|, \qquad (9)$$

where the third and fourth equalities are by Theorem 3.2 and the construction of C, respectively. This enables us to replace the original pair (A,b) by the smaller pair (C,y) for the solvers in Table 1 as in Algorithms 5–8. A scaling factor β is also needed in Algorithms 7–8.

Cross validation. To select the value of the regularization term α, the existing Python solvers we used partition the rows of A into m folds (subsets) and run the solver $m \cdot |\mathbb{A}|$ times for every fold and $\alpha \in \mathbb{A}$ to select the desired α; see Kohavi u. a. (1995) for details. For consistency, Algorithm 4 computes a coreset for each of these m folds in Line 4 and concatenate them in Line 5. Thus, (9) holds similarly for $m > 1$.

The experiments. We evaluated the algorithms in Table 1 using the common Python's solvers in its right two columns. Most of these experiments were repeated twice: using the default CPython distribution Wikipedia contributors (2019a) and Intel's distribution LTD (2019) of Python. All the experiments were conducted on a standard Lenovo Z70 laptop with an Intel i7-5500U CPU @ 2.40GHZ and 16GB RAM. We used the 3 following real-world datasets in our experiments:

(i) 3D Road Network (North Jutland, Denmark) Data Set Kaul u. a. (2013). It contains 434874 records. We used the 2 attributes: (Web Mercaptor (Google format) longitude [real], Web Mercaptor (Google format) latitude [real]) to predict the attribute (Height in meters [real]).

(ii) Individual household electric power consumption Data Set dataset:power (2012). It contains 2075259 records. We used the 2 attributes: (global active power [kilowatt - real], global reactive power [kilowatt - real]) to predict the attribute (voltage [volt - real]).

(iii) House Sales in King County, USA dataset:sales (2015). It contains $21,600$ records. We used the following 8 attributes: (bedrooms [integer], sqft living [integer], sqft lot [integer], floors [integer], waterfront [boolean], sqft above [integer], sqft basement [integer], year built [integer]) to predict the (house price [integer]) attribute.

The synthetic data is an $n \times d$ matrix A and vector b of length n, both of random entries in $[0, 1000]$. As expected by the analysis, since our compression introduces no error to the computation accuracy, the actual values of the data had no affect on the results, unlike the size of the input which affects the computation time. Table 2 summarizes the experimental results.

4.1 Discussion

Why running time is faster? The number of rows in the reduced matrix C is $O(d^2)$, which is usually much smaller than the original matrix A. This also explains why some coresets (dashed red line) failed for small values of n in Fig. 2b,2c,4b and 4c. The construction of C takes $O(nd^2)$. Solving Linear Regression takes the same time, with or without the coreset. However, the constant are much smaller since the time for computing C becomes neglected for large values of n, as shown in Fig. 11. We emphasize that, unlike common coresets, there is *no accuracy loss* due to the use of our coreset, ignoring $\pm 10^{-15}$ additive errors/improvements. The improvement factor in running time due to our booster is in order of up to x10. The contribution of the coreset is much significant, already for smaller values of n, when it boosts other solvers that use cross validation for parameter tuning as explained above. In this case, the time complexity reduced by a factor of $m \cdot |\mathbb{A}|$ since the coreset is computed only once for each of the m folds, regardless of the size $|\mathbb{A}|$. In this case, the running time improvement is between x10–x100.
As shown in the graphs, the computations via Intel's Python distribution reduced the running times by 15-40%, with or without the booster, probably due to its tailored implementation for our hardware.

Why numerical stability is better? A sketch that simply sums the 1-rank matrices of outer products of rows in the input matrix $A' = (A \mid b)$ yields its covariance matrix $B = A'^T A'$. The Cholesky decomposition $B = L^T L$ then returns a small matrix $L \in \mathbb{R}^{d \times d}$ that can be plugged to the solvers, similarly to our coreset. This algorithm which we call SKETCH + CHOLESKY is so simple and there is no hope to improve its running time via our much more involved booster. Nevertheless, it is numerically unstable for the reasons that are explained in Section 1.1. In fact, on most of our experiments we could not even apply this technique at all using 32-bit floating point representation. This is because the resulting approximation to $A'^T A'$ was not a positive definite matrix as required by the Cholesky Decomposition, and we could not compute the matrix L at all. In case of success, the running time of the booster was slower by at most a factor of 2 but even in these cases numerical accuracy is improved up to orders of magnitude; See Fig. 10b–10a for histogram of errors using such 32-bit float representation which is especially common in GPUs for saving memory, running time and power Wikipedia contributors (2019b). For the special case of Linear regression, we can avoid Cholesky decomposition and compute the solution $(A^T A)^{-1} A^T b$ directly after maintaining such a sketch for $A^T A$ and $A^T b$. However, this sketch wich we call SKETCH + INVERSE still has large numerical issues compared to our coreset computation as shown in Fig. 10b–10a.

Algorithm 4 LMS-CORESET(A, b, m, k)

Input: A matrix $A \in \mathbb{R}^{n \times d}$, a vector $b \in \mathbb{R}^n$,
and a number (integer) m of cross-validation folds,
and an integer $k \in \{1, \cdots, n\}$ that denotes accuracy/speed trade-off.

Output: A matrix $C \in \mathbb{R}^{O(md^2) \times d}$ of weighted subset of rows in A, and a vector $y \in \mathbb{R}^d$.

$A' := (A \mid b)$ `// A matrix` $A' \in \mathbb{R}^{n \times (d+1)}$

$\{A'_1, \cdots, A'_m\} :=$ split the matrix A' into m block matrices, each of size $(\frac{n}{m}) \times (d+1)$

for *every* $i \in \{1, \cdots, m\}$ **do**

  $S_i :=$ CARATHEODORY-MATRIX(A'_i, k) `// see Algorithm 3`

$S := (S_1^T | \cdots | S_m^T)^T$ `//` concatenation of the m matrices into a single matrix of $m(d+1)^2 + m$ rows and $d+1$ columns

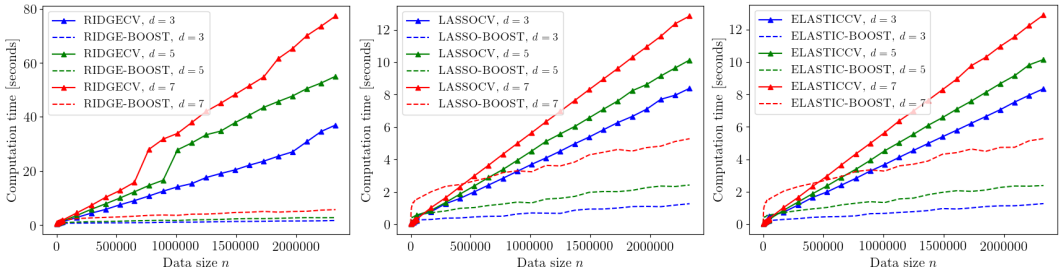
$C :=$ the first d columns of S.

$y :=$ the last column of S.

return (C, y)

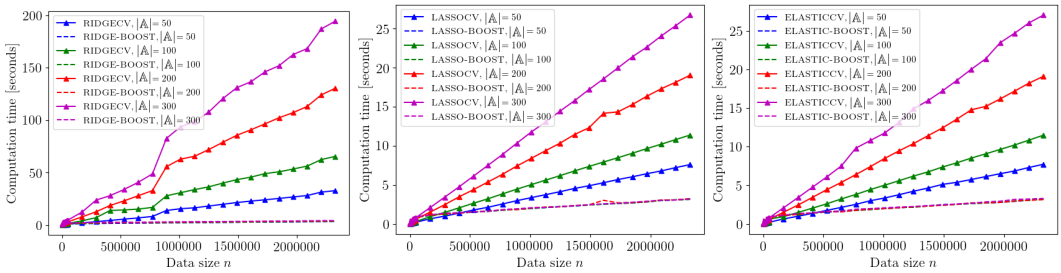
Figure	Algorithm's number	x/y Axes labels	Python Distribution	Dataset	Input Parameter
2	6–8	Size/Time for various d	CPython	Synthetic	$m = 3$, $\lvert\mathbb{A}\rvert = 100$
3	6–8	Size/Time for various $\lvert\mathbb{A}\rvert$	CPython	Synthetic	$m = 3, d = 7$
4	6–8	Size/Time for various d	Intel's	Synthetic	$m = 3$, $\lvert\mathbb{A}\rvert = 100$
5	6–8	Size/Time for various $\lvert\mathbb{A}\rvert$	Intel's	Synthetic	$m = 3, d = 7$
6	6–8	$\lvert\mathbb{A}\rvert$/Time	CPython	Datasets (i)–(ii)	$m = 3$, $\lvert\mathbb{A}\rvert = 100$
7	6–8	$\lvert\mathbb{A}\rvert$/Time	Intel's	Datasets (i)–(ii)	$m = 3$, $\lvert\mathbb{A}\rvert = 100$
8	6–8	Time/maximal $\lvert\mathbb{A}\rvert$ than is feasible	CPython	Datasets (i)–(ii)	$m = 3$
9	6–8	Time/maximal $\lvert\mathbb{A}\rvert$ than is feasible	Intel's	Datasets (i)–(ii)	$m = 3$
10	5	Error/Count Histogram + Size/Error	CPython	Datasets (i),(iii)	$m = 1$
11	5	Size/Time for various Distributions	CPython, Intel's	Synthetic	$m = 64, d = 15$

Table 2: Experiments. Our booster was applied on the common CPython Wikipedia contributors (2019a) and Intel's LTD (2019) distributions (implementations). The input matrix is $A \in \mathbb{R}^{n \times d}$ with label vector $b \in \mathbb{R}^n$, where n is "Data size". Cross validation uses m folds for evaluating each regularization term in $\mathbb{A}$. Number of clusters is chosen as $k = 2(d+1)^2 + 2$ in order to have $O(\log n)$ iterations in Algorithm 2, and $\rho = 0.5$ for Algorithm 8. Computation time includes the computation of the reduced input (C, y); See Section 3. The histogram graphs consist of bins that count the number of occurrences of a range *of errors.*



(a) *(b)* *(c)*

Figure 2: Time comparison on synthetic data using CPython distribution.



(a) *(b)* *(c)*

Figure 3: Time comparison on synthetic data using CPython distribution.

Algorithm 5 LINREG-BOOST(A, b, m, k)

$(C, y) :=$ LMS-CORESET(A, b, m, k)
$x^* :=$ `lstsq`(C, y)
return x^*

Algorithm 6 RIDGECV-BOOST$(A, b, \mathbb{A}, m, k)$

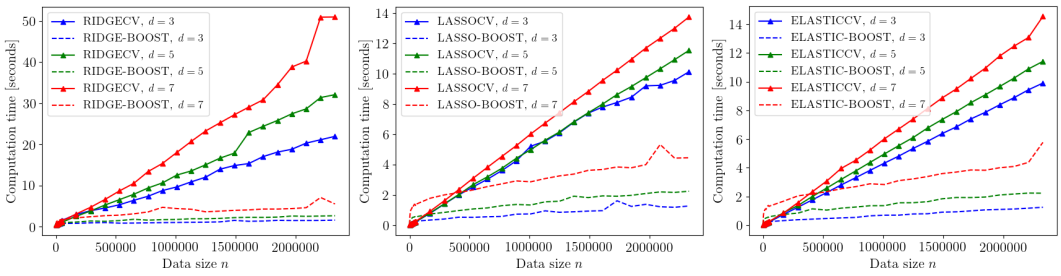
$(C, y) :=$ LMS-CORESET(A, b, m, k)
$(x, \alpha) :=$ `RidgeCV`$(C, y, \mathbb{A}, m)$
return (x, α)

Algorithm 7 LASSOCV-BOOST$(A, b, \mathbb{A}, m, k)$

$(C, y) :=$ LMS-CORESET(A, b, m, k)
$\beta := \sqrt{\left(m \cdot (d+1)^2 + m\right)/n}$
$(x, \alpha) :=$ `LassoCV`$(\beta \cdot C, \beta \cdot y, \mathbb{A}, m)$
return (x, α)

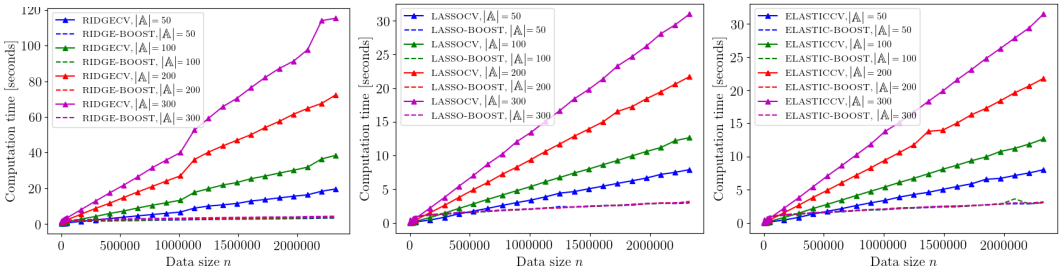
Algorithm 8 ELASTICCV-BOOST$(A, b, m, \mathbb{A}, \rho, k)$

$(C, y) :=$ LMS-CORESET(A, b, m, k)
$\beta := \sqrt{\left(m \cdot (d+1)^2 + m\right)/n}$
$(x, \alpha) :=$ `ElasticNetCV`$(\beta \cdot C, \beta \cdot y, \mathbb{A}, \rho, m)$
return (x, α)



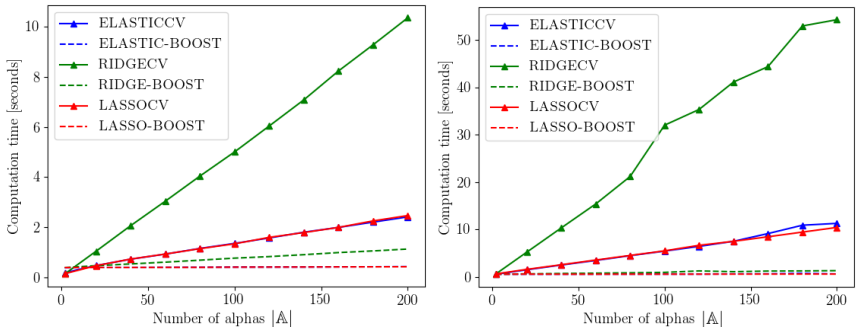
(a) *(b)* *(c)*

Figure 4: Time comparison on synthetic data using Intel's python distribution.



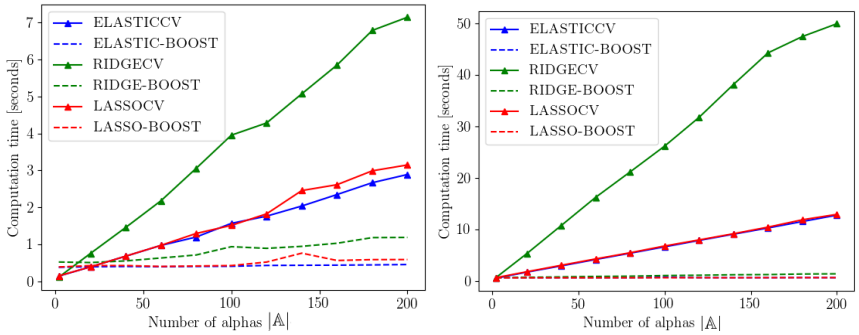
(a) *(b)* *(c)*

Figure 5: Time comparison on synthetic data using Intel's python distribution.



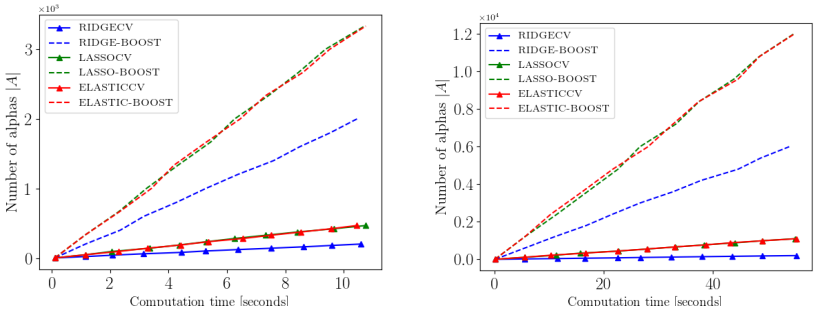
(a) Dataset (i). *(b) Dataset (ii).*

Figure 6: Time comparison on real world data using CPython distribution.



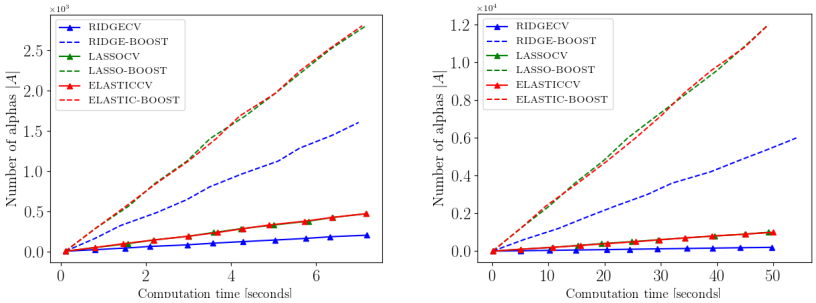
(a) Dataset (i). *(b) Dataset (ii).*

Figure 7: Time comparison on real world data using Intel's python distribution.



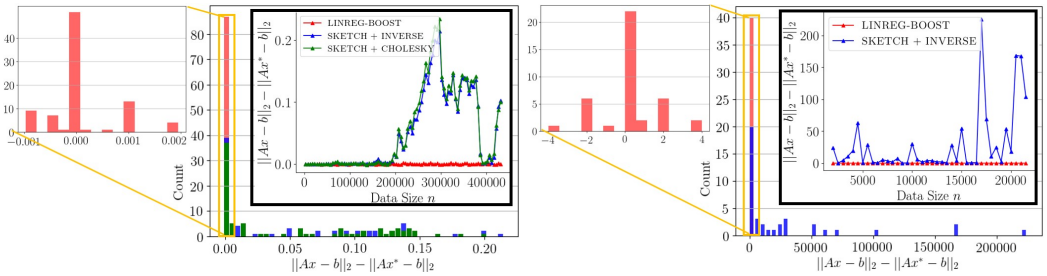
(a) Dataset (i). *(b) Dataset (ii).*

Figure 8: Number of alphas $|\mathbb{A}|$ that can be tested in a predefined amount of time using CPython distribution.



(a) Dataset (i). *(b) Dataset (ii).*

Figure 9: Number of alphas $|\mathbb{A}|$ that can be tested in a predefined amount of time using Intel's Python distribution.



(a) Dataset (i). *(b) Dataset (iii).*

Figure 10: Accuracy comparison using real word data. $x^* = \texttt{lstsq}(A, b)$. x was computed using the methods specified in the legend; see Section 4.1.

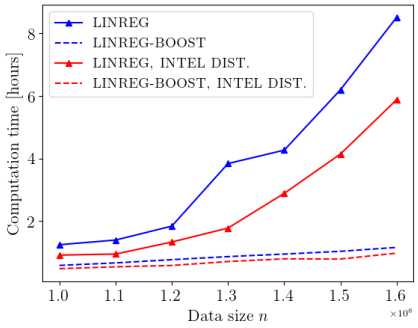


Figure 11: Time comparison using synthetic data.

5 Conclusion and Future Work

We presented a novel framework that combines sketches and coresets. As an example application, we proved that the set from the Caratheodory Theorem can be computed in $O(nd)$ overall time for sufficiently large n (for a set of n points in $\mathbb{R}^d$). This is instead of $O(n^2d^2)$ time as in the original theorem. We then generalized the result for a matrix S whose rows are a weighted subset of the input matrix and their covariance matrix is the same. Our experimental results section shows how to significantly boost the numerical stability or running time of existing LMS solvers by applying them on S. Future work includes: (a) applications of our framework to combine other sketch-coreset pairs e.g. as listed in Phillips (2016), (b) Experiments for streaming/distributed/GPU data and other potential applications such as for Deep Learning e.g. as part of the Stochastic Gradient Descent that uses the LMS adaptive filter Widrow u. a. (1977); Mandic (2004), and (c) experiments with higher dimensional data: we may compute each of the $O(d^2)$ entries in the covariance matrix by calling our algorithm with $d = 2$ and the corresponding pair of columns in the d columns of the input matrix.

References

[dataset:power 2012] : *Individual household electric power consumption Data Set* . `https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption`. 2012

[dataset:sales 2015] : *House Sales in King County, USA*. `https://www.kaggle.com/harlfoxem/housesalesprediction`. 2015

[Afrabandpey u. a. 2016] AFRABANDPEY, Homayun ; PELTOLA, Tomi ; KASKI, Samuel: Regression Analysis in Small-n-Large-p Using Interactive Prior Elicitation of Pairwise Similarities. In: *FILM 2016, NIPS Workshop on Future of Interactive Learning Machines*, 2016

[Agarwal u. a. 2004] AGARWAL, Pankaj K. ; HAR-PELED, Sariel ; VARADARAJAN, Kasturi R.: Approximating extent measures of points. In: *Journal of the ACM (JACM)* 51 (2004), Nr. 4, S. 606–635

[Bauckhage 2015] BAUCKHAGE, Christian: NumPy/SciPy Recipes for Data Science: Ordinary Least Squares Optimization. In: *researchgate. net, Mar* (2015)

[Bjorck 1967] BJORCK, Ake: Solving linear least squares problems by Gram-Schmidt orthogonalization. In: *BIT Numerical Mathematics* 7 (1967), Nr. 1, S. 1–21

[Carathéodory 1907] CARATHÉODORY, Constantin: Über den Variabilitätsbereich der Koeffizienten von Potenzreihen, die gegebene Werte nicht annehmen. In: *Mathematische Annalen* 64 (1907), Nr. 1, S. 95–115

[Clarkson und Woodruff 2009] CLARKSON, Kenneth L. ; WOODRUFF, David P.: Numerical linear algebra in the streaming model. In: *Proceedings of the forty-first annual ACM symposium on Theory of computing* ACM (Veranst.), 2009, S. 205–214

[Cook und Webster 1972] COOK, WD ; WEBSTER, RJ: Caratheodory's theorem. In: *Canadian Mathematical Bulletin* 15 (1972), Nr. 2, S. 293–293

[Copas 1983] COPAS, John B.: Regression, prediction and shrinkage. In: *Journal of the Royal Statistical Society: Series B (Methodological)* 45 (1983), Nr. 3, S. 311–335

[Cormen u. a. 2009] CORMEN, Thomas H. ; LEISERSON, Charles E. ; RIVEST, Ronald L. ; STEIN, Clifford: *Introduction to algorithms*. MIT press, 2009

[Drineas u. a. 2006] DRINEAS, Petros ; MAHONEY, Michael W. ; MUTHUKRISHNAN, Shan: Sampling algorithms for l 2 regression and applications. In: *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm* Society for Industrial and Applied Mathematics (Veranst.), 2006, S. 1127–1136

[Feldman u. a. 2010] FELDMAN, Dan ; MONEMIZADEH, Morteza ; SOHLER, Christian ; WOODRUFF, David P.: Coresets and sketches for high dimensional subspace approximation problems. In: *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms* Society for Industrial and Applied Mathematics (Veranst.), 2010, S. 630–649

[Feldman u. a. 2016] FELDMAN, Dan ; VOLKOV, Mikhail ; RUS, Daniela: Dimensionality Reduction of Massive Sparse Datasets Using Coresets. In: *Advances in neural information processing systems (NIPS)*, 2016

[Gallagher u. a. 2017] GALLAGHER, Neil ; ULRICH, Kyle R. ; TALBOT, Austin ; DZIRASA, Kafui ; CARIN, Lawrence ; CARLSON, David E.: Cross-spectral factor analysis. In: *Advances in Neural Information Processing Systems*, 2017, S. 6842–6852

[Golub und Reinsch 1971] GOLUB, Gene H. ; REINSCH, Christian: Singular value decomposition and least squares solutions. In: *Linear Algebra*. Springer, 1971, S. 134–151

[Golub und Van Loan 2012] GOLUB, Gene H. ; VAN LOAN, Charles F.: *Matrix computations*. Bd. 3. JHU press, 2012

[Hoerl und Kennard 1970] HOERL, Arthur E. ; KENNARD, Robert W.: Ridge regression: Biased estimation for nonorthogonal problems. In: *Technometrics* 12 (1970), Nr. 1, S. 55–67

[Jolliffe 2011] JOLLIFFE, Ian: *Principal component analysis*. Springer, 2011

[Kang u. a. 2011] KANG, Byung ; LIM, Woosang ; JUNG, Kyomin: Scalable kernel K-means via centroid approximation. In: *Proc. NIPS*, 2011

[Kaul u. a. 2013] KAUL, Manohar ; YANG, Bin ; JENSEN, Christian S.: Building accurate 3d spatial networks to enable next generation intelligent transportation systems. In: *2013 IEEE 14th International Conference on Mobile Data Management* Bd. 1 IEEE (Veranst.), 2013, S. 137–146

[Kohavi u. a. 1995] KOHAVI, Ron u. a.: A study of cross-validation and bootstrap for accuracy estimation and model selection. In: *Ijcai* Bd. 14 Montreal, Canada (Veranst.), 1995, S. 1137–1145

[Laparra u. a. 2015] LAPARRA, Valero ; MALO, Jesús ; CAMPS-VALLS, Gustau: Dimensionality reduction via regression in hyperspectral imagery. In: *IEEE Journal of Selected Topics in Signal Processing* 9 (2015), Nr. 6, S. 1026–1036

[Liang u. a. 2013] LIANG, Yingyu ; BALCAN, Maria-Florina ; KANCHANAPALLY, Vandana: Distributed PCA and k-means clustering. In: *The Big Learning Workshop at NIPS* Bd. 2013 Citeseer (Veranst.), 2013

[LTD 2019] LTD, Intel: *Accelerate Python* Performance*. `https://software.intel.com/en-us/distribution-for-python`. 2019

[Maalouf u. a. 2019] MAALOUF, Alaa ; JUBRAN, Ibrahim ; FELDMAN, Dan: *Open code for the algorithms in this paper*. 2019. – Open code will be provided upon the publication of this paper.

[Mandic 2004] MANDIC, Danilo P.: A generalized normalized gradient descent algorithm. In: *IEEE signal processing letters* 11 (2004), Nr. 2, S. 115–118

[Nasser u. a. 2015] NASSER, Soliman ; JUBRAN, Ibrahim ; FELDMAN, Dan: Coresets for Kinematic Data: From Theorems to Real-Time Systems. In: *arXiv preprint arXiv:1511.09120* (2015)

[Pearson 1900] PEARSON, Karl: X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50 (1900), Nr. 302, S. 157–175

[Peng u. a. 2015] PENG, Xi ; YI, Zhang ; TANG, Huajin: Robust subspace clustering via thresholding ridge regression. In: *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015

[Phillips 2016] PHILLIPS, Jeff M.: Coresets and sketches. In: *arXiv preprint arXiv:1601.00617* (2016)

[Porco u. a. 2015] PORCO, Aldo ; KALTENBRUNNER, Andreas ; GÓMEZ, Vicenç: Low-rank approximations for predicting voting behaviour. In: *Workshop on Networks in the Social and Information Sciences, NIPS*, 2015

[Safavian und Landgrebe 1991] SAFAVIAN, S R. ; LANDGREBE, David: A survey of decision tree classifier methodology. In: *IEEE transactions on systems, man, and cybernetics* 21 (1991), Nr. 3, S. 660–674

[Seber und Lee 2012] SEBER, George A. ; LEE, Alan J.: *Linear regression analysis*. Bd. 329. John Wiley & Sons, 2012

[Tibshirani 1996] TIBSHIRANI, Robert: Regression shrinkage and selection via the lasso. In: *Journal of the Royal Statistical Society: Series B (Methodological)* 58 (1996), Nr. 1, S. 267–288

[Widrow u. a. 1977] WIDROW, Bernard ; MCCOOL, John ; LARIMORE, Michael G. ; JOHNSON, C R.: Stationary and nonstationary learning characteristics of the LMS adaptive filter. In: *Aspects of Signal Processing*. Springer, 1977, S. 355–393

[Wikipedia contributors 2019a] WIKIPEDIA CONTRIBUTORS: *CPython — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=CPython&oldid=896388498`. 2019

[Wikipedia contributors 2019b] WIKIPEDIA CONTRIBUTORS: *List of Nvidia graphics processing units — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=List_of_Nvidia_graphics_processing_units&oldid=897973746`. 2019

[Zhang und Rohe 2018] ZHANG, Yilin ; ROHE, Karl: Understanding Regularized Spectral Clustering via Graph Conductance. In: *Advances in Neural Information Processing Systems*, 2018, S. 10631–10640

[Zou und Hastie 2005] ZOU, Hui ; HASTIE, Trevor: Regularization and variable selection via the elastic net. In: *Journal of the royal statistical society: series B (statistical methodology)* 67 (2005), Nr. 2, S. 301–320