Coresets for Minimum Enclosing Balls over Sliding Windows

Yanhao Wang
National University of Singapore
yanhao90@comp.nus.edu.sg

Yuchen Li
Singapore Management University
yuchenli@smu.edu.sg

Kian-Lee Tan
National University of Singapore
tankl@comp.nus.edu.sg

ABSTRACT

Coresets are important tools to generate concise summaries of massive datasets for approximate analysis. A coreset is a small subset of points extracted from the original point set such that certain geometric properties are preserved with provable guarantees. This paper investigates the problem of maintaining a coreset to preserve the minimum enclosing ball (MEB) for a sliding window of points that are continuously updated in a data stream. Although the problem has been extensively studied in batch and append-only streaming settings, no efficient sliding-window solution is available yet. In this work, we first introduce an algorithm, called AOMEB, to build a coreset for MEB in an append-only stream. AOMEB improves the practical performance of the state-of-the-art algorithm while having the same approximation ratio. Furthermore, using AOMEB as a building block, we propose two novel algorithms, namely SWMEB and SWMEB+, to maintain coresets for MEB over the sliding window with constant approximation ratios. The proposed algorithms also support coresets for MEB in a reproducing kernel Hilbert space (RKHS). Finally, extensive experiments on real-world and synthetic datasets demonstrate that SWMEB and SWMEB+ achieve speedups of up to four orders of magnitude over the state-of-the-art batch algorithm while providing coresets for MEB with rather small errors compared to the optimal ones.

CCS CONCEPTS

• **Information systems** → *Data stream mining*; • **Theory of computation** → *Computational geometry*; *Streaming algorithms*.

KEYWORDS

coresets; minimum enclosing ball; streaming algorithm

ACM Reference Format:
Yanhao Wang, Yuchen Li, and Kian-Lee Tan. 2019. Coresets for Minimum Enclosing Balls over Sliding Windows. In *The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '19), August 4–8, 2019, Anchorage, AK, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3292500.3330826

1 INTRODUCTION

Unprecedented growth of data poses significant challenges in designing algorithms that can scale to massive datasets. Algorithms with superlinear complexity often become infeasible on datasets with millions or billions of points. *Coresets* are effective approaches to tackling the challenges of big data analysis. A coreset is a small subset extracted from the original point set such that certain geometric properties are preserved with provable guarantees [3]. Instead of processing the original dataset, one can perform the computation on its coreset with little loss of accuracy. Various types of problems have been shown to be effective under coreset approximation, e.g., k-median and k-means clustering [3, 8, 20], non-negative matrix factorization (NMF) [17], kernel density estimation (KDE) [26, 35], and many others [6, 10, 21].

Coresets for minimum enclosing balls (MEB) [2, 4, 5, 22, 23, 33, 34] have received significant attention due to its wide applications in clustering [4, 5, 22], support vector machines [28], kernel regression [32], fuzzy inference [13], shape fitting [23], and approximate furthest neighbor search [25]. Given a set of points P, the minimum enclosing ball of P, denoted by $\mathsf{MEB}(P)$, is the smallest ball that contains all points in P. A subset $S \subset P$ is a μ-coreset for $\mathsf{MEB}(P)$ if the distance between the center $\mathbf{c}^*(S)$ of $\mathsf{MEB}(S)$ and any point in P is within $\mu \cdot r^*(S)$, where $r^*(S)$ is the radius of $\mathsf{MEB}(S)$ [4]. Existing studies [4, 22] show that there always exists a $(1+\varepsilon)$-coreset of size $O(\frac{1}{\varepsilon})$ $(0 < \varepsilon < 1)$ for the MEB of any point set, which is independent of the dataset size and dimension.

Most existing methods [4, 5, 22, 23, 33] of coresets for MEB focus on the batch setting and must keep all points in memory when constructing the coresets. In many applications, such as network monitoring, financial analysis, and sensor data mining, one needs to process data in the *streaming* model [24] where the input points arrive one at a time and cannot be stored entirely. There have been several methods [1, 2, 11, 34] to maintain coresets for MEB in data streams. The state-of-the-art streaming algorithm [2] can maintain a $(\sqrt{2}+\varepsilon)$-coreset for MEB with a single pass through the input dataset. However, these algorithms only consider the append-only scenario where new points are continuously added to, but old ones are never deleted from, the stream. Hence, they fail to capture recency in time-sensitive applications since the computation may be performed on outdated data. To meet the recency requirement, the *sliding window* model [8, 9, 15, 16] that only considers the most recent N points in the stream at any time is a popular approach for real-time analytics. One can trivially adapt append-only methods for the sliding window model but a complete coreset reconstruction is deemed inevitable once an expired point is deleted. To the best of our knowledge, there is no existing algorithm that can maintain coresets for MEB over the sliding window efficiently.

Our Results. In this paper, we investigate the problem of maintaining coresets for MEB in the sliding window model. In particular, our results are summarized as follows.

- In Section 3.1, we present the AOMEB algorithm to maintain a $(\sqrt{2}+\varepsilon)$-coreset of size $O(\frac{\log \theta}{\varepsilon^2})$ with $O(\frac{m \log \theta}{\varepsilon^3})$ computation time per point for the MEB of an append-only stream, where m

KDD '19, August 4–8, 2019, Anchorage, AK, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6201-6/19/08…$15.00
https://doi.org/10.1145/3292500.3330826

is the dimension of the points and θ is the ratio of the maximum and minimum distances between any two points in the input dataset. AOMEB shows better empirical performance than the algorithm in [2] while having the same approximation ratio.

- In Section 3.2, using AOMEB as a building block, we propose the SWMEB algorithm for coreset maintenance over a sliding window W_t of the most recent N points at time t. SWMEB divides W_t into equal-length partitions. On each partition, it maintains a sequence of indices where each index corresponds to an instance of AOMEB. Theoretically, SWMEB can return a $(\sqrt{2}+\varepsilon)$-coreset for $\mathsf{MEB}(W_t)$ with $O(\sqrt{N}\cdot\frac{m\log\theta}{\varepsilon^3})$ time and $O(\sqrt{N}\cdot\frac{\log\theta}{\varepsilon^2})$ space complexity, where N is the window size.
- In Section 3.3, we propose the SWMEB+ algorithm to improve upon SWMEB. SWMEB+ only maintains one sequence of indices, as well as the corresponding AOMEB instances, over W_t. By keeping fewer indices, SWMEB+ is more efficient than SWMEB in terms of time and space. Specifically, it only stores $O(\frac{\log^2\theta}{\varepsilon^3})$ points with $O(\frac{m\log^2\theta}{\varepsilon^4})$ processing time per point, both of which are independent of N. At the same time, it can still return a $(9.66+\varepsilon)$-coreset for $\mathsf{MEB}(W_t)$.
- In Section 3.4, we generalize our proposed algorithms to maintain coresets for MEB in a reproducing kernel Hilbert space (RKHS).
- In Section 4, we conduct extensive experiments on real-world and synthetic datasets to evaluate the performance of our proposed algorithms. The experimental results demonstrate that (1) AOMEB outperforms the state-of-the-art streaming algorithm [2] in terms of coreset quality and efficiency; (2) SWMEB and SWMEB+ can return coresets for MEB with rather small errors (mostly within 1%), which are competitive with AOMEB and other streaming algorithms; (3) SWMEB and SWMEB+ achieve 2 to 4 orders of magnitude speedups over batch algorithms while running between 10 and 150 times faster than AOMEB; (4) SWMEB+ further improves the efficiency of SWMEB by up to 14 times while providing coresets with similar or even better quality.

2 PRELIMINARIES AND RELATED WORK

Coresets for MEB. For two m-dimensional points $\mathbf{p}=(p_1,\ldots,p_m)$, $\mathbf{q}=(q_1,\ldots,q_m)$, the Euclidean distance between $\mathbf{p}$ and $\mathbf{q}$ is denoted by $d(\mathbf{p},\mathbf{q})=\sqrt{\sum_{i=1}^{m}(p_i-q_i)^2}$. An m-dimensional (closed) ball with center $\mathbf{c}$ and radius r is defined as $B(\mathbf{c},r)=\{\mathbf{p}\in\mathbb{R}^m : d(\mathbf{c},\mathbf{p})\le r\}$. We use $\mathbf{c}(B)$ and $r(B)$ to denote the center and radius of ball B. The μ-expansion of ball $B(\mathbf{c},r)$, denoted as $\mu\cdot B$, is a ball centered at $\mathbf{c}$ with radius $\mu\cdot r$, i.e., $\mu\cdot B=B(\mathbf{c},\mu\cdot r)$.

Given a set of n points $P=\{\mathbf{p}_1,\ldots,\mathbf{p}_n\}\subset\mathbb{R}^m$, the minimum enclosing ball of P, denoted as $\mathsf{MEB}(P)$, is the smallest ball that contains all points in P. The center and radius of $\mathsf{MEB}(P)$ are represented by $\mathbf{c}^*(P)$ and $r^*(P)$. For a parameter $\mu>1$, a ball B is a μ-approximate MEB of P if $P\subset B$ and $r(B)\le\mu\cdot r^*(P)$. A subset $S\subset P$ is a μ-coreset for $\mathsf{MEB}(P)$, or μ-$\mathsf{Coreset}(P)$ for brevity, if $P\subset\mu\cdot\mathsf{MEB}(S)$. Since $S\subseteq P$ and $r^*(S)\le r^*(P)$, $\mu\cdot\mathsf{MEB}(S)$ is always a μ-approximate MEB of P.

Sliding Window Model. This work focuses on maintaining coresets for MEB in append-only streaming and sliding window settings. For a sequence of (possibly infinite) points $P=\langle\mathbf{p}_1,\mathbf{p}_2,\ldots\rangle$ arriving continuously as a data stream where $\mathbf{p}_t$ is the t-th point, we first consider the problem of maintaining a μ-$\mathsf{Coreset}(P_t)$ for $P_t=\{\mathbf{p}_1,\ldots,\mathbf{p}_t\}$ at any time t.

Algorithm 1: CoreMEB [4]

```
Input  : A set of points $P = \{\mathbf{p}_1, \ldots, \mathbf{p}_n\}$, a parameter $\varepsilon \in (0, 1)$
Output : A coreset $S$ for $\mathsf{MEB}(P)$
1 $\mathbf{p}_a \leftarrow \mathrm{argmax}_{\mathbf{p}\in P}\, d(\mathbf{p}_1, \mathbf{p})$, $\mathbf{p}_b \leftarrow \mathrm{argmax}_{\mathbf{p}\in P}\, d(\mathbf{p}_a, \mathbf{p})$;
2 $S \leftarrow \{\mathbf{p}_a, \mathbf{p}_b\}$;
3 Initialize $B(\mathbf{c}, r)$ with $\mathbf{c} \leftarrow (\mathbf{p}_a + \mathbf{p}_b)/2$, $r \leftarrow d(\mathbf{p}_a, \mathbf{p}_b)/2$;
4 while $\exists \mathbf{p} \in P \setminus S : \mathbf{p} \notin (1 + \varepsilon) \cdot B$ do
5     $\mathbf{q} \leftarrow \mathrm{argmax}_{\mathbf{p}\in P\setminus S}\, d(\mathbf{c}, \mathbf{p})$, $S \leftarrow S \cup \{\mathbf{q}\}$;
6     Update $B$ such that $B(\mathbf{c}, r) = \mathsf{MEB}(S)$;
7 return $S$;
```

Furthermore, we consider the count-based sliding window[1] on the stream P: given a window size N, the sliding window [15] W_t at any time t always contains the latest N points, i.e., $W_t=\{\mathbf{p}_{t'},\ldots,\mathbf{p}_t\}$ where $t'=\max(1,t-N+1)$. We consider the problem of maintaining a μ-$\mathsf{Coreset}(W_t)$ for W_t at any time t.

Related Work. We review the literature on MEB computation and coresets for MEB. Gärtner [19] and Fischer et al. [18] propose two pivoting algorithms that resemble the simplex method of linear programming for computing exact MEBs. Both algorithms have an exponential complexity w.r.t. the dimension m and thus are not scalable for large datasets with high dimensions. Subsequently, a line of research work [4, 5, 22, 23, 33] studies the problem of building coresets to approximate MEBs. They propose efficient batch algorithms for constructing a $(1+\varepsilon)$-$\mathsf{Coreset}(P)$ of any point set P. The basic scheme used in these algorithms is presented in Algorithm 1. First of all, it selects the point $\mathbf{p}_a$ furthest from $\mathbf{p}_1$ and $\mathbf{p}_b$ furthest from $\mathbf{p}_a$ out of P, using $S=\{\mathbf{p}_a,\mathbf{p}_b\}$ as the initial coreset (Lines 1 & 2). The center $\mathbf{c}$ and radius r of $\mathsf{MEB}(S)$ can be computed from $\mathbf{p}_a$ and $\mathbf{p}_b$ directly (Line 3). Then, it iteratively picks the point $\mathbf{q}$ furthest from the current center $\mathbf{c}$, adds $\mathbf{q}$ to S, and updates $B(\mathbf{c},r)$ so that B is $\mathsf{MEB}(S)$, until no point in P is outside of the $(1+\varepsilon)$-expansion of B (Lines 4–6). Finally, it returns S as a coreset for $\mathsf{MEB}(P)$ (Line 7). Theoretically, Algorithm 1 terminates in $O(\frac{1}{\varepsilon})$ iterations and returns a $(1+\varepsilon)$-$\mathsf{Coreset}(P)$ of size $O(\frac{1}{\varepsilon})$ [4]. Compared with exact MEB solvers [18, 19], coreset-based approaches run in linear time w.r.t. the dataset size n and dimension m, and achieve better performance on high-dimensional data. Nevertheless, they must store all points in memory and process them in multiple passes, which are not suitable for data stream applications.

Several methods are proposed to approximate MEBs or coresets for MEB in streaming and dynamic settings. Agarwal et al. [1] and Chan [11] propose algorithms to build $(1+\varepsilon)$-coresets for MEB in append-only streams. Though working well in low dimensions, both algorithms become impractical for higher dimensions (i.e., $m>10$) due to $O(1/\varepsilon^{O(m)})$ complexity. Zarrabi-Zadeh and Chan [34] propose a 1.5-approximate algorithm to compute MEBs in append-only streams. Agarwal and Sharathkumar [2] design a data structure that can maintain $(\sqrt{2}+\varepsilon)$-coresets for MEB and $(1.37+\varepsilon)$-approximate MEBs over append-only streams. Chan and Pathak [12] propose

[1] In this paper, we focus on the count-based sliding window model. But our proposed approaches can be trivially extended to the time-based sliding window model [15].

Algorithm 2: AOMEB

```
  Input  : A set of points $P = \{\mathbf{p}_1, \dots, \mathbf{p}_n\}$, a parameter $\varepsilon_1 \in (0, 1)$
  Output : A coreset $S$ for MEB($P$)
1 $S_1 \leftarrow \{\mathbf{p}_1\}$ and initialize $B_1(\mathbf{c}_1, r_1)$ with $\mathbf{c}_1 \leftarrow \mathbf{p}_1, r_1 \leftarrow 0$;
2 for $t \leftarrow 2, \dots, n$ do
3     if $\mathbf{p}_t \notin (1 + \varepsilon_1) \cdot B_{t-1}$ then
4         $S_t \leftarrow S_{t-1} \cup \{\mathbf{p}_t\}$;
5         Update $B_{t-1}$ to $B_t(\mathbf{c}_t, r_t) = \mathrm{MEB}(S_t)$;
6     else
7         $S_t \leftarrow S_{t-1}$ and $B_t \leftarrow B_{t-1}$;
8 return $S \leftarrow S_n$;
```

a method for maintaining $(1.22 + \varepsilon)$-approximate MEBs in the dynamic setting, which supports the insertions and deletions of random points. To the best of our knowledge, none of the existing methods can maintain coresets for MEB over the sliding window efficiently. All of them have to store the entire window of points and recompute from scratch for every window slide, which is expensive in terms of time and space.

3 OUR ALGORITHMS

In this section we present our algorithms to maintain coresets for MEB. We first introduce a $(\sqrt{2} + \varepsilon)$-approximate append-only streaming algorithm, called AOMEB, in Section 3.1. Using AOMEB as a building block, we propose the SWMEB algorithm with the same $(\sqrt{2} + \varepsilon)$-approximation ratio in Section 3.2. Furthermore, we propose a more efficient SWMEB+ algorithm that retains a constant approximation ratio in Section 3.3.

3.1 The AOMEB Algorithm

The AOMEB algorithm is inspired by CoreMEB [4] (see Algorithm 1) to work in the append-only streaming model. Compared with CoreMEB, which can access the entire dataset and *optimally* select the furthest point into the coreset at each iteration, AOMEB is restricted to process the dataset in a single pass and determine whether to include a point into the coreset or discard it immediately after seeing it. Therefore, AOMEB adopts a *greedy* strategy for coreset maintenance: adding a new point to the coreset once it is outside of the MEB w.r.t. the current coreset.

The pseudo code of AOMEB is presented in Algorithm 2. First of all, it takes $S_1 = \{\mathbf{p}_1\}$ as the initial coreset with $B_1(\mathbf{p}_1, 0)$ as MEB(S_1) (Line 1). Then, it performs a one-pass scan over the point set, using the procedure in Lines 3–7 for each point $\mathbf{p}_t$: It first computes the distance between $\mathbf{p}_t$ and $\mathbf{c}_{t-1}$. If $\mathbf{p}_t \in (1 + \varepsilon_1) \cdot B_{t-1}$, no update is needed; otherwise, it adds $\mathbf{p}_t$ to the coreset S_{t-1} and updates B_{t-1} to MEB(S_t). Finally, after processing all points in P, it returns S_n as the coreset S for MEB(P) (Line 8).

Theoretical Analysis. Next, we provide an analysis of the approximation ratio and complexity of AOMEB. It is noted that the *greedy* strategy of AOMEB is also adopted by existing streaming algorithms, i.e., SSMEB [34] and blurred ball cover (BBC) [2]. Nevertheless, the update procedure is different: SSMEB uses a simple geometric method to enlarge the MEB such that both the previous MEB and the new point are contained while AOMEB and BBC recompute the MEB once the coreset is updated. As a result, AOMEB and BBC are less efficient than SSMEB but ensure a better approximation ratio. Compared with BBC, which keeps the "archives" of MEBs for previous coresets, AOMEB only maintains one MEB w.r.t. S_t at time t. Therefore, AOMEB is more efficient than BBC in practice. Next, we will prove that AOMEB has the same $(\sqrt{2} + \varepsilon)$-approximation as BBC. First of all, we present the *hemisphere property* [5] that forms the basis of our analysis.

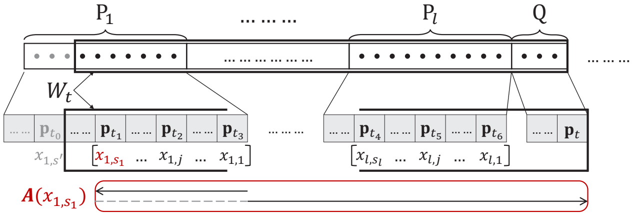
P$_1$
… … …
P$_l$
Q
… … … … … …
… … …
W_t
… … $\mathbf{p}_{t_0}$ … … $\mathbf{p}_{t_1}$ … … $\mathbf{p}_{t_2}$ … … $\mathbf{p}_{t_3}$
… … …
… … $\mathbf{p}_{t_4}$ … … $\mathbf{p}_{t_5}$ … … $\mathbf{p}_{t_6}$
… … $\mathbf{p}_t$
$x_{1,s'}$
$[x_{1,s_1}$ … $x_{1,j}$ … $x_{1,1}]$
$[x_{l,s_l}$ … $x_{l,j}$ … $x_{l,1}]$
$A(x_{1,s_1})$

Figure 1: An illustration of the SWMEB algorithm. Two arrows indicate the order in which the points in W_t are processed by $\mathcal{A}(x_{1,s_1})$.

Lemma 3.1 (Hemisphere Property [5]). *For a set of points $P \subset \mathbb{R}^m$, any closed half-space that contains $\mathbf{c}^*(P)$ must contain at least a point $\mathbf{p} \in P$ such that $d(\mathbf{c}^*(P), \mathbf{p}) = r^*(P)$.*

Based on Lemma 3.1, we can analyze the complexity and approximation ratio of AOMEB theoretically.

Theorem 3.2. *For any $\mathbf{p}_t \in S_n$, it holds that $r_t \geq (1 + \frac{\varepsilon_1^2}{8})r_{t-1}$.*

Theorem 3.3. *For any $\mathbf{p}_t \in P$, it holds that $\mathbf{p}_t \in (\sqrt{2} + \varepsilon_1) \cdot B_n$.*

The proofs of above theorems are in the extended version [31].

Theorem 3.3 indicates that AOMEB returns a $(\sqrt{2}+\varepsilon)$-Coreset(P) where $\varepsilon = O(\varepsilon_1)$ for an arbitrary point set P. According to Theorem 3.2, the radius of MEB(S_t) increases by $1 + O(\varepsilon^2)$ times whenever a new point is added to S_t. After processing $\mathbf{p}_1$ and $\mathbf{p}_2$, the coreset S_2 contains both points with $r_2 \geq d_{min}/2$ where $d_{min} = \min_{\mathbf{p},\mathbf{q} \in P \wedge \mathbf{p} \neq \mathbf{q}} d(\mathbf{p}, \mathbf{q})$. In addition, the radius r_n of S_n is bounded by $r^*(P) < d_{max}$ where $d_{max} = \max_{\mathbf{p},\mathbf{q} \in P} d(\mathbf{p}, \mathbf{q})$. Therefore, $r_n/r_2 = O(\theta)$ where $\theta = d_{max}/d_{min}$ and the size of S is $O(\frac{\log \theta}{\varepsilon^2})$. Finally, the update procedure for each point $\mathbf{p}_t$ spends $O(m)$ time to compute $d(\mathbf{c}_{t-1}, \mathbf{p}_t)$ and $O(\frac{m \log \theta}{\varepsilon^3})$ time to update B_t.

3.2 The SWMEB Algorithm

In this subsection, we present the SWMEB algorithm for coreset maintenance over the sliding window W_t. The basic idea is to adapt AOMEB for the sliding window model by keeping multiple AOMEB instances with different starting points over W_t. However, the key problem is to identify the appropriate indices, i.e., starting points, for these instances. A naive scheme, i.e., creating a set of indices that are evenly distributed over W_t, cannot give any approximation guarantee of coreset quality. Therefore, we design a partition-based scheme for index maintenance in SWMEB: dividing W_t into equal-length partitions and keeping a sequence of indices on each partition such that at least one instance can provide an approximate coreset for MEB(W_t) at any time t.

The procedure of SWMEB is illustrated in Figure 1. It divides W_t into $l = N/L$ partitions $\{P_1, \dots, P_l\}$ of equal length L. It keeps a

Algorithm 3: SWMEB

```
   Input  : A sequence of points $P = \langle \mathbf{p}_1, \mathbf{p}_2, \ldots \rangle$, the window size
            $N$, the partition size $L$, two parameters $\varepsilon_1, \varepsilon_2 \in (0, 1)$
   Output : A coreset $S_t$ for MEB($W_t$)
1  Initialize $l \leftarrow 0, Q \leftarrow \emptyset, X_0 \leftarrow \emptyset$;
2  for $t \leftarrow 1, 2, \ldots$ do
3      $Q \leftarrow Q \cup \{\mathbf{p}_t\}, X_t \leftarrow X_{t-1}$;
4      if $|Q| = L$ then
5          if $t \le N$ then
6              $l \leftarrow l + 1$, create a new partition $P_l \leftarrow Q$;
7          else
8              Drop $P_1$, shift $P_i$ (as well as the indices on $P_i$) to $P_{i-1}$
                 for $i \in [2, l]$, and create a new partition $P_l \leftarrow Q$;
9          Initialize an instance $\mathcal{A}$ of Algorithm 2, $r_l \leftarrow 0$, $s_l \leftarrow 0$;
10         for $t' \leftarrow t, \ldots, t - L + 1$ do
11             $\mathcal{A}$ processes $\mathbf{p}_{t'}$ with Line 3–7 of Algorithm 2 and
                 maintains a coreset $S[t, t']$ and its MEB $B[t, t']$;
12             if $r[t, t'] \ge (1 + \varepsilon_2) r_l$ then
13                 $r_l \leftarrow r[t, t'], s_l \leftarrow s_l + 1$;
14                 $x_{l,s_l} \leftarrow t', X_t \leftarrow X_t \cup \{x_{l,s_l}\}$;
15                 $\mathcal{A}(x_{l,s_l}) \leftarrow \mathcal{A}$ after processing $\mathbf{p}_{t'}$;
16         $Q, P_l \leftarrow \emptyset$;
17     if $x_{1,s_1} < t - N + 1$ then
18         $X_t \leftarrow X_t \setminus \langle x_{1,s_1} \rangle$, terminate $\mathcal{A}(x_{1,s_1})$, and $s_1 \leftarrow s_1 - 1$;
19     for $i \leftarrow 1, \ldots, l$ and $j \leftarrow 1, \ldots, s_i$ do
20         $\mathcal{A}(x_{i,j})$ processes $\mathbf{p}_t$ with Line 3–7 of Algorithm 2 and
             maintains a coreset $S[x_{i,j}, t]$ and its MEB $B[x_{i,j}, t]$;
21     return $S_t \leftarrow S[x_{1,s_1}, t]$ as the coreset for MEB($W_t$);
```

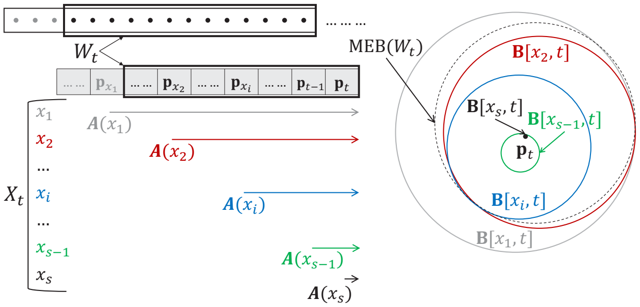

Figure 2: An illustration of the SWMEB+ algorithm.

sequence of s_i indices $\langle x_{i,1}, \ldots, x_{i,s_i} \rangle$ from the end to the beginning of each partition P_i. As W_t slides over time, old points in P_1 expire (colored in *grey*) while new points are temporarily stored in a buffer Q. The index x_{1,s_1} on P_1, which is the closest to the beginning of W_t, will be deleted once it expires. When the size of Q reaches L, it will delete P_1 and shift remaining partitions as all points in P_1 must have expired. Then, it creates a new partition P_l for the points in Q and the indices on P_l. Moreover, each index $x_{i,j}$ corresponds to an AOMEB instance $\mathcal{A}(x_{i,j})$ that processes $P[x_{i,j}, t] = \{\mathbf{p}_{x_{i,j}}, \ldots, \mathbf{p}_t\}$ at any time t. Specifically, $\mathcal{A}(x_{i,j})$ will process the points from the end of P_i to $\mathbf{p}_{x_{i,j}}$ when $x_{i,j}$ is created and then update for each point till $\mathbf{p}_t$. Finally, the coreset is always provided by $\mathcal{A}(x_{1,s_1})$.

The pseudo code of SWMEB is presented in Algorithm 3. For initialization, the latest partition ID l is set to 0 and the buffer Q as well as the indices X_0 are set to $\emptyset$ (Line 1). Then, it processes all points in the stream P one by one with the procedure of Lines 3–20, which can be separated into four phases as follows.

- **Phase 1 (Lines 3–8):** After adding a new point $\mathbf{p}_t$ to Q, it checks the size of Q. If $|Q| = L$, a new partition will be created for Q. When $t \le N$, it increases l by 1 and creates a new partition P_l. Otherwise, P_1 must have expired and thus is dropped. Then, the partitions $\{P_2, \ldots, P_l\}$ (and the indices on each partition) are shifted to $\{P_1, \ldots, P_{l-1}\}$ and a new partition P_l is created.
- **Phase 2 (Lines 9–16):** Next, it creates the indices and corresponding AOMEB instances on P_l. It runs an AOMEB instance $\mathcal{A}$ to process each point in P_l inversely from $\mathbf{p}_t$ to $\mathbf{p}_{t-L+1}$. Initially, the number of indices s_l on P_l and the radius r_l w.r.t. the latest index x_{l,s_l} are 0. We denote the coreset maintained by $\mathcal{A}$ after processing $\mathbf{p}_{t'}$ as $S[t, t']$. Then, MEB($S[t, t']$) is represented by $B[t, t']$ with radius $r[t, t']$. If $r[t, t'] \ge (1 + \varepsilon_2) r_l$, it will update r_l to $r[t, t']$, add a new index x_{l,s_l} to X_t, and use the snapshot of $\mathcal{A}$ after processing $\mathbf{p}_{t'}$ as $\mathcal{A}(x_{l,s_l})$. After the indices on P_l is created, Q will be reset for new incoming points.
- **Phase 3 (Lines 17–18):** It checks whether x_{1,s_1}, i.e., the earliest index on P_1, has expired. If so, it will delete x_{1,s_1} from X_t and terminate $\mathcal{A}(x_{1,s_1})$ accordingly.
- **Phase 4 (Lines 19–20):** For each index $x_{i,j} \in X_t$ with $i \in [1, l]$ and $j \in [1, s_i]$ at time t, it updates the corresponding AOMEB instance $\mathcal{A}(x_{i,j})$ by processing $\mathbf{p}_t$.

Finally, it always returns $S[x_{1,s_1}, t]$ from $\mathcal{A}(x_{1,s_1})$ as the coreset S_t for MEB(W_t) at time t (Line 21).

Theoretical Analysis. In the following, we will first prove the approximation ratio of S_t returned by SWMEB for MEB(W_t). Then, we will discuss the time and space complexity of SWMEB.

Theorem 3.4. *For any* $\mathbf{p} \in W_t$, *it holds that* $\mathbf{p} \in (\sqrt{2} + \varepsilon) \cdot \mathrm{MEB}(S_t)$ *where* $\varepsilon = O(\varepsilon_1 + \sqrt{\varepsilon_2})$.

The proof of Theorem 3.4 is in the extended version [31].

Theorem 3.4 shows that S_t returned by SWMEB is a $(\sqrt{2} + \varepsilon)$-Coreset(W_t) where $\varepsilon = O(\varepsilon_1 + \sqrt{\varepsilon_2})$ at any time t. To analyze the complexity of SWMEB, we first consider the number of indices in X_t. For each partition, SWMEB maintains $O(\frac{\log \theta}{\varepsilon_2})$ indices where $\theta = d_{max}/d_{min}$. Thus, X_t contains $O(\frac{l \log \theta}{\varepsilon_2})$ indices and the number of points stored by SWMEB is $O(\frac{l \log^2 \theta}{\varepsilon_1^2 \varepsilon_2} + L)$. Furthermore, the time of SWMEB to update a point $\mathbf{p}_t$ comprises (1) the time to maintain the instance w.r.t. each index in X_t for $\mathbf{p}_t$ and (2) the amortized time to create the indices for each partition. Overall, the time complexity of SWMEB to update each point is $O(\frac{l m \log^2 \theta}{\varepsilon_1^3 \varepsilon_2})$. As $l = N/L$, the number of points maintained by SWMEB is minimal when $L = \frac{\log \theta}{\varepsilon_1} \cdot \sqrt{\frac{N}{\varepsilon_2}}$. In this case, the number of points stored by SWMEB is $O(\sqrt{N} \cdot \frac{\log \theta}{\varepsilon^2})$ and the time complexity of SWMEB to update one point is $O(\sqrt{N} \cdot \frac{m \log \theta}{\varepsilon^3})$.

3.3 The SWMEB+ Algorithm

In this subsection we present the SWMEB+ algorithm that improves upon SWMEB in terms of time and space while still achieving a constant approximation ratio. The basic idea of SWMEB+

Algorithm 4: SWMEB+

```
Input  : A sequence of points $P = \langle \mathbf{p}_1, \mathbf{p}_2, \dots \rangle$, the window size
         $N$, two parameters $\varepsilon_1, \varepsilon_2 \in (0, 1)$
Output : A coreset $S_t$ for $\mathsf{MEB}(W_t)$
1  Initialize $s \leftarrow 0, X_0 \leftarrow \emptyset$;
2  for $t \leftarrow 1, 2, \dots$ do
3      $s \leftarrow s + 1, x_s \leftarrow t$, and $X_t \leftarrow X_{t-1} \cup \langle x_s \rangle$;
4      Initialize an instance $\mathcal{A}(x_s)$ of Algorithm 2;
5      while $x_2 < t - N + 1$ do
6          $X_t \leftarrow X_t \setminus \langle x_1 \rangle$, terminate $\mathcal{A}(x_1)$;
7          Shift the remaining indices in $X_t$, $s \leftarrow s - 1$;
8      for $i \leftarrow 1, \dots, s$ do
9          $\mathcal{A}(x_i)$ processes $\mathbf{p}_t$ with Line 3–7 of Algorithm 2,
             maintaining a coreset $S[x_i, t]$ and its MEB $B[x_i, t]$;
10     while $\exists i : 1 \le i \le s - 2 \wedge r[x_i, t] \le (1 + \varepsilon_2) r[x_{i+2}, t]$ do
11         $X_t \leftarrow X_t \setminus \langle x_{i+1} \rangle$, terminate $\mathcal{A}(x_{i+1})$;
12         Shift the remaining indices in $X_t$, $s \leftarrow s - 1$;
13     if $x_1 \ge t - N + 1$ then
14         return $S_t \leftarrow S[x_1, t]$ as the coreset for $\mathsf{MEB}(W_t)$;
15     else
16         return $S_t \leftarrow S[x_2, t]$ as the coreset for $\mathsf{MEB}(W_t)$;
```

is illustrated in Figure 2. Different from SWMEB, SWMEB+ only maintains a single sequence of s indices $X_t = \{x_1, \dots, x_s\}$ over W_t. Then, each index x_i also corresponds to an AOMEB instance $\mathcal{A}(x_i)$ that processes a substream of points from $\mathbf{p}_{x_i}$ to $\mathbf{p}_t$. We use $S[x_i, t]$ for the coreset returned by $\mathcal{A}(x_i)$ at time t and $B[x_i, t]$ centered at $\mathbf{c}[x_i, t]$ with radius $r[x_i, t]$ for $\mathsf{MEB}(S[x_i, t])$. Furthermore, SWMEB+ maintains the indices based on the radii of the MEBs. Specifically, given any $\varepsilon_2 > 0$, for three neighboring indices x_i, x_{i+1}, x_{i+2}, if $r[x_i, t] \le (1 + \varepsilon_2) r[x_{i+2}, t]$, then x_{i+2} is considered as a good approximation for x_i and thus x_{i+1} can be deleted. In this way, the radii of the MEBs gradually decreases from x_1 to x_s, with the ratios of any two neighboring indices close to $(1 + \varepsilon_2)$. Any window starting between x_i and x_{i+1} is approximated by $\mathcal{A}(x_{i+1})$. Finally, SWMEB+ keeps at most one expired index (and must be x_1) in X_t to track the upper bound for the radius $r^*(W_t)$ of $\mathsf{MEB}(W_t)$. The AOMEB instance corresponding to the first non-expired index (x_1 or x_2) provides the coreset for $\mathsf{MEB}(W_t)$.

The pseudo code of SWMEB+ is presented in Algorithm 4. In the initialization phase, X_0 and s are set to $\emptyset$ and 0 respectively (Line 1). Then, all points in P are processed one by one with the procedure of Lines 3–12, which includes four phases as follows.

- **Phase 1 (Lines 3–4):** Upon the arrival of $\mathbf{p}_t$ at time t, it creates a new index $x_s = t$ and adds x_s to X_t; accordingly, an AOMEB instance $\mathcal{A}(x_s)$ w.r.t. x_s is initialized to process the substream beginning at $\mathbf{p}_t$.
- **Phase 2 (Lines 5–7):** When there exists more than one expired index (i.e., earlier than the beginning of W_t), it deletes the first index x_1 and terminates $\mathcal{A}(x_1)$ until there is only one expired index. Note that it shifts the remaining indices after deletion to always guarantee x_i is the i-th index of X_t.
- **Phase 3 (Lines 8–9):** For each $x_i \in X_t$, it updates the instance $\mathcal{A}(x_i)$ for $\mathbf{p}_t$. The update procedure follows Line 3–7 of Algorithm 2. After the update, $\mathcal{A}(x_i)$ maintains a coreset $S[x_i, t]$ and its MEB $B[x_i, t]$ by processing a stream $P[x_i, t] = \langle \mathbf{p}_{x_i}, \dots, \mathbf{p}_t \rangle$.
- **Phase 4 (Lines 10–12):** It executes a scan of X_t from x_1 to x_t to delete the indices that can be approximated by their successors. For each $x_i \in X_t$ ($i \le s - 2$), it checks the radii $r[x_i, t]$ and $r[x_{i+2}, t]$ of $B[x_i, t]$ and $B[x_{i+2}, t]$. If $r[x_i, t] \le (1 + \varepsilon_2) r[x_{i+2}, t]$, then it deletes the index x_{i+1} from X_t, terminates $\mathcal{A}(x_{i+1})$, and shifts the remaining indices accordingly.

After performing the above procedure, it returns either $S[x_1, t]$ (when x_1 has not expired) or $S[x_2, t]$ (when x_1 has expired) as the coreset S_t for $\mathsf{MEB}(W_t)$ at time t.

Theoretical Analysis. The strategy of index maintenance based on the ratios of radii is inspired by *Smooth Histograms* [9] for estimating stream statistics over sliding windows. However, *Smooth Histograms* cannot be applied to our problem because it requires an oracle to provide a $(1 + \varepsilon)$-approximate function value in any append-only stream [16] but any practical solution (i.e., [2] and AOMEB) only gives a $(\sqrt{2}+\varepsilon)$-approximation for $r^*(P)$ of an append-only stream P. In addition, *Smooth Histograms* are also used for submodular maximization in the sliding window model [16, 29, 30]. Nevertheless, such an extension is still not applicable for our problem because the radius function $r^*(\cdot)$ is not submodular in the view of set functions, which is shown by a counter example in the extended version [31]. In the following, we will prove that SWMEB+ still has a constant approximation ratio by an analysis that is different from [9, 16, 29, 30].

Theorem 3.5. *For any* $\mathbf{p} \in W_t$*, it holds that* $\mathbf{p} \in (9.66 + \varepsilon) \cdot \mathsf{MEB}(S_t)$ *where* $\varepsilon = O(\sqrt{\varepsilon_1} + \sqrt{\varepsilon_2})$.

The proof of Theorem 3.5 is in the extended version [31].

According to Theorem 3.5, SWMEB+ can always return a $(9.66 + \varepsilon)$-Coreset(W_t) at any time t. In practice, since $\frac{r[x_1,t]}{r[x_2,t]}$ is bounded by $1+O(\varepsilon_2)$ in almost all cases, the approximation ratio of SWMEB+ can be improved to $(3.36 + \varepsilon)$ accordingly. Furthermore, for any $x_i \in X_t$ ($i \le s - 2$), either $r[x_{i+1}, t]$ or $r[x_{i+2}, t]$ is less than $(1 + \varepsilon_2) r[x_i, t]$. In addition, it holds that $r[x_1, t] \le d_{max}$ and $r[x_{s-1}, t] \ge 0.5 \cdot d_{min}$. Therefore, the number of indices in X_t is $O(\frac{\log \theta}{\varepsilon})$. Accordingly, the time complexity for SWMEB+ to update each point is $O(\frac{m \log^2 \theta}{\varepsilon^4})$ while the number of points stored by SWMEB+ is $O(\frac{\log^2 \theta}{\varepsilon^3})$, both of which are independent of N.

3.4 Generalization to Kernelized MEB

In real-world applications [13, 28, 32], it is required to compute the coreset for MEB in a reproducing kernel Hilbert space (RKHS) instead of Euclidean space. Given a symmetric positive definite kernel $k(\cdot, \cdot) : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$ and its associated feature mapping $\varphi(\cdot)$ where $k(\mathbf{p}, \mathbf{q}) = \langle \varphi(\mathbf{p}), \varphi(\mathbf{q}) \rangle$ for any $\mathbf{p}, \mathbf{q} \in \mathbb{R}^m$, the kernelized MEB of a set of points P is the smallest ball $B^*(\mathbf{c}^*, r^*)$ in the RKHS such that the maximum distance from $\mathbf{c}^*$ to $\varphi(\mathbf{p})$ is no greater than r^*, which can be formulated as follows:

$$\min_{\mathbf{c}, r} \; r^2 \quad \text{s.t.} \quad \big(\mathbf{c} - \varphi(\mathbf{p}_i)\big)' \big(\mathbf{c} - \varphi(\mathbf{p}_i)\big) \le r^2, \forall \mathbf{p}_i \in P \tag{1}$$

However, it is impractical to solve Problem 1 directly in the primal form due to the infinite dimensionality of RKHS. We transform Problem 1 into the dual form as follows:

$$\max_{\boldsymbol{\alpha}} \; \boldsymbol{\alpha}' \text{diag}(\mathbf{K}) - \boldsymbol{\alpha}' \mathbf{K} \boldsymbol{\alpha} \quad \text{s.t.} \quad \boldsymbol{\alpha} \ge \mathbf{0}, \boldsymbol{\alpha}' \mathbf{1} = 1 \tag{2}$$

where $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_n]'$ is the n-dimensional Lagrange multiplier vector, $\mathbf{0} = [0, \ldots, 0]'$ and $\mathbf{1} = [1, \ldots, 1]'$ are both n-dimensional vectors, $\mathbf{K} = [k(\mathbf{p}_i, \mathbf{p}_j)]_{i,j=1}^{n}$ is the $n \times n$ kernel matrix of P, and diag($\mathbf{K}$) is the diagonal of $\mathbf{K}$. Problem 2 is known to be a quadratic programming [7] (QP) problem. According to the KKT conditions [7], the kernelized MEB B^* of P, can be recovered from the Lagrange multiplier vector $\boldsymbol{\alpha}$ as follows:

$$\mathbf{c}^* = \sum_{i=1}^{n} \alpha_i \cdot \varphi(\mathbf{p}_i),\ (r^*)^2 = \boldsymbol{\alpha}'\text{diag}(\mathbf{K}) - \boldsymbol{\alpha}'\mathbf{K}\boldsymbol{\alpha} \tag{3}$$

Here the center $\mathbf{c}^*$ is represented implicitly by each point $\mathbf{p}_i$ in P and the corresponding α_i's. Then, the distance between $\mathbf{c}^*$ and $\varphi(\mathbf{q})$ for any $\mathbf{q} \in \mathbb{R}^m$ can be computed by:

$$d(\mathbf{c}^*, \varphi(\mathbf{q}))^2 = \sum_{i,j=1}^{n} \alpha_i \alpha_j k(\mathbf{p}_i, \mathbf{p}_j) + k(\mathbf{q}, \mathbf{q}) - 2\sum_{i=1}^{n} \alpha_i k(\mathbf{p}_i, \mathbf{q}) \tag{4}$$

Next, we introduce how to generalize AOMEB in Algorithm 2 to maintain coresets for kernelized MEB. First, to represent the center $\mathbf{c}_t$ and compute the distance from $\varphi(\mathbf{p}_t)$ to the center, it always keeps the Lagrange multiplier vector $\boldsymbol{\alpha}$. In Line 1, α_1 is initialized to 1 so that $\mathbf{c}_1 = \varphi(\mathbf{p}_1)$. Then, in Line 3, Equation 4 is used to compute the distance between $\mathbf{c}_{t-1}$ and $\varphi(\mathbf{p}_t)$. If $\mathbf{p}_t$ is added to S_t, it will re-optimize Problem 2 on S_t to adjust $\boldsymbol{\alpha}$ so that $B_t = \mathsf{MEB}(S_t)$ in Line 6. Specifically, the Frank-Wolfe algorithm [14, 33] is used to solve Problem 2 as it is efficient for QP problems with unit simplex constraints. SWMEB and SWMEB+ can also maintain coresets for kernelized MEB by using the generalized AOMEB instance in Algorithms 3 and 4.

Theoretically, the generalized algorithms have the same approximation ratios and coreset sizes as the original ones. However, the time complexity will increase by a factor of $\frac{1}{\varepsilon}$ because the time to compute the distance from $\mathbf{c}_{t-1}$ to $\varphi(\mathbf{p}_t)$ using Equation 4 is $O(\frac{m}{\varepsilon})$ instead of $O(m)$.

3.5 Discussion

For ease of presentation, we describe Algorithms 2–4 in the single-update-mode where the coreset is maintained for every new point. In practice, it is not required to update the coreset at such an intense rate. Here we discuss how to adapt these algorithms for the *mini-batch-mode*.

In the mini-batch-mode, given a batch size b, each update will add b new points while deleting the earliest b points at the same time. The adaptations of AOMEB in Algorithm 2 for the mini-batch-mode are as follows: (1) In Line 1, an initial coreset S_b is built for the first b points using CoreMEB in Algorithm 1. (2) In Lines 3–7, the coreset is updated for a batch of b points collectively. Specifically, it adds the points not contained in $(1 + \varepsilon_1) \cdot B_{t-b}$ in the batch to S_t and then updates B_t. To adapt SWMEB and SWMEB+ for the mini-batch-mode, each AOMEB instance should run in the mini-batch-mode as shown above. In addition, the indices and AOMEB instances in SWMEB and SWMEB+ are created and updated for batches instead of points. Note that the approximation ratio and coreset size will remain the same in the mini-batch-mode but the efficiency will be improved as fewer indices are created.

4 EXPERIMENTS

In this section we evaluate the empirical performance of our proposed algorithms on real-world and synthetic datasets. First of all, we will introduce the experimental setup in Section 4.1. Then, we will present the experimental results on effectiveness and efficiency in Section 4.2. Finally, the experimental results on scalability are presented in Section 4.3.

4.1 Experimental Setup

Algorithms. We compare the following eight algorithms for computing MEBs or building coresets for MEB in our experiments.

- **COMEB** [18]: a combinatorial algorithm for computing the *exact* MEB of a set of points in the Euclidean space. It has an exponential time complexity w.r.t. the dimension m. Moreover, it is not applicable to kernelized MEB.
- **CoreMEB** [4]: a batch algorithm for constructing a $(1 + \varepsilon)$-approximate coreset for the MEB of a set of points. The procedure is as described in Algorithm 1.
- **SSMEB** [34]: a 1.5-approximation algorithm for computing a MEB in an append-only stream. We adopt the method described in [27] to compute a kernelized MEB by SSMEB.
- **Blurred Ball Cover (BBC)** [2]: an append-only streaming algorithm to maintain a $(\sqrt{2} + \varepsilon)$-approximate coreset for MEB.
- **DyMEB** [12]: a 1.22-approximate dynamic algorithm for MEB computation. It keeps a data structure that permits to insert/delete random points without fully reconstructions.
- **AOMEB**: our append-only streaming algorithm presented in Section 3.1. It has the same $(\sqrt{2} + \varepsilon)$-approximation ratio as BBC.
- **SWMEB**: our first sliding-window algorithm presented in Section 3.2. It can maintain a $(\sqrt{2} + \varepsilon)$-coreset for MEB over the sliding window.
- **SWMEB+**: our second sliding-window algorithm presented in Section 3.3. It has higher efficiency than SWMEB at the expense of a worse approximation ratio.

We do not compare with the algorithms in [1, 11] because they cannot scale to the datasets with $m > 10$. In our experiments, all algorithms run in the mini-batch-mode with batch size $b = 100$. Furthermore, all algorithms, except for SWMEB and SWMEB+, cannot directly work in the sliding window model. The batch and append-only streaming algorithms store the entire sliding window and rerun from scratch for each update. DyMEB also stores the entire sliding window for tracking the expired point to delete. For each update, it must execute one deletion/insertion for every expired/arrival point in a mini-batch to maintain the coreset for MEB w.r.t. the up-to-date window. The implementations of our algorithms are available at https://github.com/yhwang1990/SW-MEB.

Datasets. The dataset statistics are listed in Table 3. We use 6 real-world datasets and 1 synthetic dataset for evaluation. All real-world datasets are downloaded from publicly available sources, e.g., UCI Machine Learning Repository[2], LIBSVM[3], SNAP[4], and TEXMAX[5]. The generation procedure of the Synthetic dataset is as follows. We first decide the dimension m. By default, we set $m = 50$. For testing

[2] https://archive.ics.uci.edu/ml/index.php

[3] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets

[4] http://snap.stanford.edu

[5] http://corpus-texmex.irisa.fr

Table 1: The average errors and update time of different algorithms for Euclidean MEB

Algorithm	average error							average update time (ms)						
	Census	CovType	GIST	Gowalla	HIGGS	SIFT	Synthetic	Census	CovType	GIST	Gowalla	HIGGS	SIFT	Synthetic
COMEB	0	0	0	0	0	0	0	1150.7	1914.8	3813.7	13.52	146.67	12861.8	2793.7
CoreMEB	6.03e-04	3.66e-04	4.32e-04	2.01e-04	4.59e-04	4.97e-04	4.49e-04	643.1	1275.1	2647.2	30.19	337.72	2789.1	1698.8
SSMEB	5.90e-02	1.56e-01	1.10e-01	7.19e-03	1.59e-01	1.61e-01	9.49e-02	43.75	44.11	262.8	24.55	45.66	74.16	54.46
BBC	1.59e-03	8.05e-03	4.47e-04	1.53e-03	1.33e-02	5.83e-03	2.05e-02	193.4	767.4	961.8	37.61	118.19	6807.2	1150.4
DyMEB	9.14e-04	3.27e-03	4.51e-04	1.98e-04	2.19e-03	4.34e-03	5.96e-03	1639.4	2725.9	7498.8	80.32	3483.2	9195.1	2786.3
AOMEB	9.55e-04	4.55e-03	4.55e-04	2.23e-04	4.73e-03	2.86e-03	1.14e-02	95.74	507.3	1755.5	17.82	125.48	1413.9	480.63
SWMEB	6.12e-04	8.36e-03	2.31e-04	2.57e-04	3.87e-03	2.94e-03	1.11e-02	2.129	25.08	127.77	0.1593	4.861	57.61	19.9
SWMEB+	8.98e-04	4.17e-03	1.81e-03	1.93e-04	9.67e-03	3.01e-03	1.52e-02	1.467	5.414	72.31	0.1887	2.592	14.37	5.679

Table 2: The average errors and update time of different algorithms for kernelized MEB

Algorithm	average error							average update time (ms)						
	Census	CovType	GIST	Gowalla	HIGGS	SIFT	Synthetic	Census	CovType	GIST	Gowalla	HIGGS	SIFT	Synthetic
CoreMEB	9.31e-05	9.16e-05	9.76e-05	7.97e-05	9.64e-05	9.40e-05	9.53e-05	75041.6	91571.8	36522	5813.1	87875.1	149386	76223.1
SSMEB	2.22e-01	1.73e-01	1.83e-01	2.26e-01	2.08e-01	1.67e-01	1.40e-01	208.67	251.12	2893.9	118.71	169.92	395.5	255.25
BBC	1.21e-03	1.55e-03	1.78e-05	1.15e-02	9.02e-04	1.60e-03	1.48e-03	14914.5	34728.1	704506	410.30	59789.6	148539	44383.9
DyMEB	1.62e-04	6.07e-05	1.18e-04	2.07e-03	1.57e-04	9.91e-05	4.26e-05	106093	256616	2484757	7142.1	345400	1333728	513587
AOMEB	5.63e-04	6.02e-04	1.62e-06	1.02e-04	6.08e-04	7.55e-04	4.59e-04	2002.1	5716.6	42961.2	253.87	6471.1	19163.7	10556.2
SWMEB	7.77e-04	4.71e-03	5.01e-03	1.19e-04	5.82e-03	3.12e-03	4.99e-03	146.09	545.9	4550.3	18.118	606.2	1705.7	1077.2
SWMEB+	5.86e-04	1.01e-03	4.85e-04	1.67e-04	2.10e-03	6.78e-04	1.61e-03	26.847	52.06	372.4	4.769	55.85	130.46	87.21

Table 3: Statistics of datasets

dataset	source	size	m	γ
Census	UCI	2,458,285	68	8984.58
CovType	LIBSVM	297,711	54	3.734
GIST	TEXMAX	1,000,000	960	4.0409
Gowalla	SNAP	6,442,892	2	7455.33
HIGGS	UCI	5,829,123	28	38.8496
SIFT	TEXMAX	1,000,000	128	298919.5
Synthetic	-	10,000,000	50	100.6

the scalability of different algorithms w.r.t. m, we vary m from 10 to 100 and from 1,000 to 10,000. Then, we generate a point by drawing the values of m dimensions from a normal distribution $\mathcal{N}(0,1)$ independently. For kernelized MEB, we adopt the Gaussian kernel $k(\mathbf{p}_i, \mathbf{p}_j) = \exp(-d(\mathbf{p}_i, \mathbf{p}_j)^2/\gamma)$ where $\gamma = \frac{1}{n^2}\sum_{i,j=1}^{n} d(\mathbf{p}_i, \mathbf{p}_j)^2$ is the kernel width. In practice, we sample 10,000 points randomly from each dataset to compute γ. The results are also listed in Table 3. Note that the values of γ vary with m ($\gamma \approx 2m$) on the Synthetic dataset. More details on datasets are provided in Appendix A.3.

In an experiment, all points in a dataset are processed sequentially by each algorithm in the same order as a stream and the results are recorded for every batch of points.

Parameter Setting. The default window size N is 10^5 in all experiments except the ones for testing the scalability w.r.t. N, where we vary N from 10^5 to 10^6. The parameter ε_1 in AOMEB, SWMEB, and SWMEB+ (as well as ε in CoreMEB, BBC, and DyMEB) is 10^{-3} for Euclidean MEB and 10^{-4} for kernelized MEB. Then, we use $\varepsilon_2 = 0.1$ in SWMEB and $\varepsilon_2 = \min(4^{i-1} \cdot \frac{\varepsilon_1}{10}, 0.1)$ for each $x_i \in X_t$ in SWMEB+. Finally, the partition size L in SWMEB is $\frac{N}{10}$. The procedure of parameter tuning is shown in Appendix A.5.

Environment. All experiments are conducted on a server running Ubuntu 16.04 with a 1.9GHz processor and 128 GB memory. The details on hardware and software configurations are in Appendices A.1 and A.2.

4.2 Effectiveness and Efficiency

The result quality of different algorithms is evaluated by *average error* computed as follows. For a sliding window W_t, each algorithm (except COMEB and SSMEB) returns a coreset S_t for $\mathsf{MEB}(W_t)$. The errors of coreset-based algorithms are acquired based on the definition of *coreset*: we compute the minimal λ' such that $W_t \subset \lambda' \cdot \mathsf{MEB}(S_t)$, and use $\varepsilon' = \frac{\lambda' \cdot r^*(S_t) - r^*(W_t)}{r^*(W_t)}$ for the relative error. Since SSMEB directly returns MEBs, we compute its relative error according to the definition of *approximate MEB*, i.e., $\varepsilon' = \frac{r(B'_t) - r^*(W_t)}{r^*(W_t)}$ where B'_t is the approximate MEB for W_t. It is noted that for kernelized MEB we use the radius of the MEB returned by CoreMEB when $\varepsilon = 10^{-9}$ (i.e., the relative error is within 10^{-9}) as $r^*(W_t)$ since exact MEBs are intractable in a RKHS [28]. On each dataset, we take 100 timestamps over the stream, obtain the result of each algorithm for the sliding window at each sampled timestamp, compute the relative errors of obtained results, and use the *average error* as the quality metric. The efficiency of different algorithms is evaluated by *average update time*. We record the CPU time of each algorithm to update every batch of 100 points and compute the average time per update over the entire dataset.

The *average errors* and *average update time* of different algorithms for Euclidean and kernelized MEBs are presented in Tables 1 and 2 respectively. Note that COMEB cannot be applied to kernelized MEB and thus is not included in Table 2.

In general, all algorithms except SSMEB can provide MEBs or coresets for MEB with rather small errors (at most 1.5%) in all cases. Furthermore, the update time for kernelized MEBs is one to two orders of magnitude longer than that for Euclidean MEB due to (1) higher time complexity for distance evaluation, (2) larger coresets caused by the infinite dimensionality of RKHS, and (3) smaller ε_1 (or ε) used. In terms of effectiveness, the error of COMEB for Euclidean MEB is always 0 because it can return exact results. Additionally, as CoreMEB guarantees a $(1+\varepsilon)$-approximation ratio theoretically,

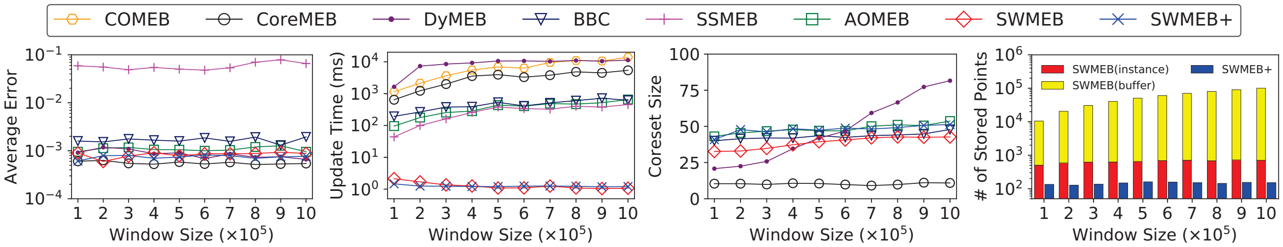

Figure 3: The performance for Euclidean MEB with varying N on the Census dataset.

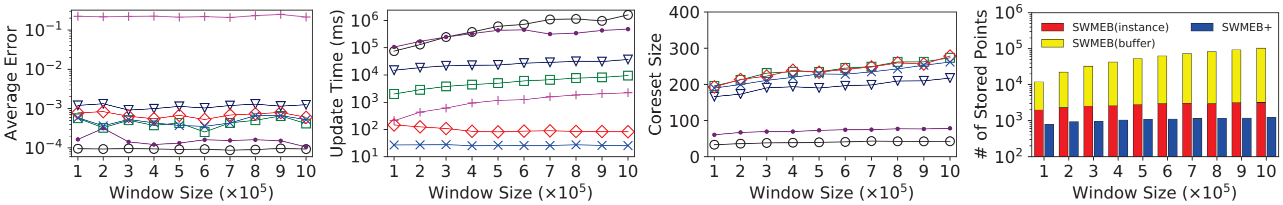

Figure 4: The performance for kernelized MEB with varying N on the Census dataset.

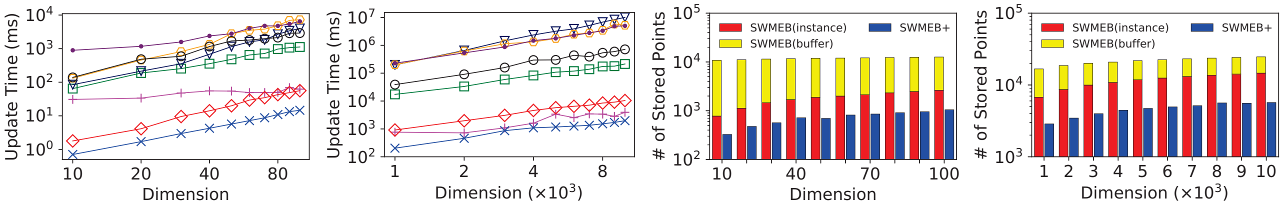

Figure 5: The performance for Euclidean MEB with varying m on the Synthetic dataset.

the errors of the coresets returned by CoreMEB are less than 10^{-3} for Euclidean MEB and 10^{-4} for kernelized MEB. With regard to the efficiency for Euclidean MEB, COMEB only runs faster than CoreMEB on low-dimensional datasets (i.e., Gowalla and HIGGS). CoreMEB can always outperform COMEB when $m \geq 30$. The reason is that COMEB has a higher time complexity than CoreMEB for its exponential dependency on m.

There are three append-only streaming algorithms in our experiments, namely BBC, SSMEB, and our proposed AOMEB. Among them, the result quality of SSMEB is not competitive with any other algorithms, though its update time is shorter than BBC and AOMEB. This is because the simple geometric method for update in SSMEB is efficient but largely inaccurate. The errors of BBC and AOMEB are slightly higher than those of CoreMEB while they are more efficient than CoreMEB. These experimental results are consistent with our theoretical analysis: AOMEB and BBC have a lower $(\sqrt{2}+\varepsilon)$-approximation ratio but only require a single-pass scan over the window. Finally, AOMEB can run up to 9 times faster than BBC while having similar or better coreset quality because AOMEB maintains fewer MEBs than BBC, which leads to a more efficient update procedure.

The dynamic algorithm, i.e., DyMEB, shows slightly better coreset quality than AOMEB and BBC but runs even slower than CoreMEB. There are two reasons for such observations: first, the data structure maintained by DyMEB contains all points in W_t for coreset construction, which naturally leads to good quality; second, the performance of DyMEB depends on the assumption that the probability of deleting any existing point is equal. However, the sliding window model always deletes the earliest point, which obviously violates this assumption. As a result, when expired points are deleted, DyMEB frequently calls for (partial) coreset reconstructions. In practice, the average update time of DyMEB even exceeds the time to build the coreset from scratch.

Finally, our sliding-window algorithms, namely SWMEB and SWMEB+, achieve two to four orders of magnitude speedups over CoreMEB across all datasets for both Euclidean and kernelized MEBs. In addition, they run 10 to 150 times faster than AOMEB. The reason for their superior efficiency is that they can maintain the coreset incrementally over the sliding window without rebuilding from scratch. In terms of effectiveness, the errors of SWMEB and SWMEB+ are slightly higher than those of AOMEB. Although both algorithms use AOMEB instances to provide the coresets, the index schemes inevitably cause quality losses since the points between the beginning of W_t (i.e., $t-N+1$) and the first non-expired index (i.e., x_{1,s_1} in SWMEB and x_2 in SWMEB+) are missing from the coresets. Lastly, SWMEB+ runs up to 14 times faster than SWMEB owing to the fact that SWMEB+ maintains fewer indices than SWMEB. At the same time, SWMEB+ can return coresets with nearly equal quality to SWMEB, which means that the radius-based index scheme of SWMEB+, though having a worse theoretical guarantee, is competitive with the partition-based scheme of SWMEB empirically.

4.3 Scalability

We compare the scalability of different algorithms with varying the window size N and dimension m. The performance for Euclidean and kernelized MEBs with varying N is shown in Figures 3 and 4.

Here we only present the results on the Census dataset due to space limitations. Additional results on other datasets can be found in the extended version [31].

First, we observe that the average errors of different algorithms basically remain stable w.r.t. N (around 10^{-3} for SWMEB and SWMEB+). The update time of all algorithms, except SWMEB and SWMEB+, increases with N because the number of points processed per update is equal to N. For SWMEB and SWMEB+, the number of indices in X_t hardly changes N. Moreover, the update frequency of any AOMEB instance decreases over time, when the coreset grows larger and fewer new points can be added. Since both algorithms contain "older" instances when N is larger, they take less time for each update on average. In addition, the coreset sizes of our algorithms are 3–5 times larger than that of CoreMEB because the *greedy* strategy used by AOMEB inevitably adds some redundant points to coresets. Despite this, the coreset sizes are at most $0.3\% \cdot N$. Moreover, the coreset size for kernelized MEBs is around 5 times larger then that for Euclidean MEBs due to the infinite dimensionality of RKHS. In terms of space, SWMEB stores up to 100 times more points than SWMEB+ since it not only maintains more AOMEB instances that SWMEB+ but also needs to keep a buffer Q, whose size is at most $10\% \cdot N$ in the experiments. Specifically, SWMEB stores at most $13\% \cdot N$ points while SWMEB+ only stores up to $2,000$ points, which barely changes with N.

The update time and space usage of different algorithms with varying the dimension m is shown in Figure 5. Here only the results for Euclidean MEBs are presented as the trend is similar for kernelized MEBs. As plotted in log-log scale, we can observe that the update time of all algorithms except SSMEB increases almost linearly with m. Nevertheless, SWMEB and SWMEB+ still demonstrate their superiority in efficiency on high-dimensional datasets. Furthermore, both algorithms store more points when m is larger because the coreset size grows with m. But even when $m = 10,000$ the ratios of stored points are at most 25% and 5% for SWMEB and SWMEB+ respectively, whereas any other algorithms require to store the entire window of points.

5 CONCLUSION

We studied the problem of maintaining a coreset for the MEB of a sliding window of points in this paper. Firstly, we proposed the AOMEB algorithm to maintain a $(\sqrt{2} + \varepsilon)$-coreset for MEB in an append-only stream. Then, based on AOMEB, we proposed two sliding-window algorithms, namely SWMEB and SWMEB+, for coreset maintenance with constant approximation ratios. We further generalized our proposed algorithms for kernelized MEBs. Empirically, SWMEB and SWMEB+ improved the efficiency of the state-of-the-art batch algorithm by up to four orders of magnitude while providing coresets with rather small errors compared to the optimal ones. For future work, we plan to explore the applications of coresets to various data mining and machine learning problems in the streaming or sliding window model.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their insightful comments that help us improve the quality and presentation of this paper. Yuchen Li is supported by the Singapore MOE Tier 1 grant MSS18C001.

REFERENCES

[1] Pankaj K. Agarwal, Sariel Har-Peled, and Kasturi R. Varadarajan. 2004. Approximating extent measures of points. *J. ACM* 51, 4 (2004), 606–635.

[2] Pankaj K. Agarwal and R. Sharathkumar. 2010. Streaming Algorithms for Extent Problems in High Dimensions. In *SODA*. 1481–1489.

[3] Olivier Bachem, Mario Lucic, and Andreas Krause. 2018. Scalable k-Means Clustering via Lightweight Coresets. In *KDD*. 1119–1127.

[4] Mihai Badoiu and Kenneth L. Clarkson. 2008. Optimal Core-sets for Balls. *Comput. Geom.* 40, 1 (2008), 14–22.

[5] Mihai Badoiu, Sariel Har-Peled, and Piotr Indyk. 2002. Approximate clustering via core-sets. In *STOC*. 250–257.

[6] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, and Ricard Gavaldà. 2011. Mining frequent closed graphs on evolving data streams. In *KDD*. 591–599.

[7] Stephen Boyd and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge university press.

[8] Vladimir Braverman, Harry Lang, Keith Levin, and Morteza Monemizadeh. 2016. Clustering Problems on Sliding Windows. In *SODA*. 1374–1390.

[9] Vladimir Braverman and Rafail Ostrovsky. 2007. Smooth Histograms for Sliding Windows. In *FOCS*. 283–293.

[10] Matteo Ceccarello, Andrea Pietracaprina, and Geppino Pucci. 2018. Fast Coreset-based Diversity Maximization under Matroid Constraints. In *WSDM*. 81–89.

[11] Timothy M. Chan. 2006. Faster core-set constructions and data-stream algorithms in fixed dimensions. *Comput. Geom.* 35, 1-2 (2006), 20–35.

[12] Timothy M. Chan and Vinayak Pathak. 2014. Streaming and dynamic algorithms for minimum enclosing balls in high dimensions. *Comput. Geom.* 47, 2 (2014), 240–247.

[13] Fu-Lai Chung, Zhaohong Deng, and Shitong Wang. 2009. From Minimum Enclosing Ball to Fast Fuzzy Inference System Training on Large Datasets. *IEEE Trans. Fuzzy Syst.* 17, 1 (2009), 173–184.

[14] Kenneth L. Clarkson. 2010. Coresets, Sparse Greedy Approximation, and the Frank-Wolfe Algorithm. *ACM Trans. Algorithms* 6, 4 (2010), 63:1–63:30.

[15] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 2002. Maintaining Stream Statistics over Sliding Windows. *SIAM J. Comput.* 31, 6 (2002), 1794–1813.

[16] Alessandro Epasto, Silvio Lattanzi, Sergei Vassilvitskii, and Morteza Zadimoghaddam. 2017. Submodular Optimization Over Sliding Windows. In *WWW*. 421–430.

[17] Dan Feldman and Tamir Tassa. 2015. More Constraints, Smaller Coresets: Constrained Matrix Approximation of Sparse Big Data. In *KDD*. 249–258.

[18] Kaspar Fischer, Bernd Gärtner, and Martin Kutz. 2003. Fast Smallest-Enclosing-Ball Computation in High Dimensions. In *ESA*. 630–641.

[19] Bernd Gärtner. 1999. Fast and Robust Smallest Enclosing Balls. In *ESA*. 325–338.

[20] Sariel Har-Peled and Soham Mazumdar. 2004. On coresets for k-means and k-median clustering. In *STOC*. 291–300.

[21] Jonathan H. Huggins, Trevor Campbell, and Tamara Broderick. 2016. Coresets for Scalable Bayesian Logistic Regression. In *NIPS*. 4080–4088.

[22] Piyush Kumar, Joseph S. B. Mitchell, and Emre Alper Yildirim. 2003. Approximate Minimum Enclosing Balls in High Dimensions using Core-Sets. *ACM J. Exp. Algorithmics* 8 (2003).

[23] Thomas Larsson and Linus Källberg. 2013. Fast and Robust Approximation of Smallest Enclosing Balls in Arbitrary Dimensions. *Comput. Graph. Forum* 32, 5 (2013), 93–101.

[24] S. Muthukrishnan. 2005. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science* 1, 2 (2005), 117–236.

[25] Rasmus Pagh, Francesco Silvestri, Johan Sivertsen, and Matthew Skala. 2015. Approximate Furthest Neighbor in High Dimensions. In *SISAP*. 3–14.

[26] Jeff M. Phillips and Wai Ming Tai. 2018. Near-Optimal Coresets of Kernel Density Estimates. In *SoCG*. 66:1–66:13.

[27] Piyush Rai, Hal Daumé III, and Suresh Venkatasubramanian. 2009. Streamed Learning: One-Pass SVMs. In *IJCAI*. 1211–1216.

[28] Ivor W. Tsang, James T. Kwok, and Pak-Ming Cheung. 2005. Core Vector Machines: Fast SVM Training on Very Large Data Sets. *J. Mach. Learn. Res.* 6 (2005), 363–392.

[29] Yanhao Wang, Qi Fan, Yuchen Li, and Kian-Lee Tan. 2017. Real-Time Influence Maximization on Dynamic Social Streams. *PVLDB* 10, 7 (2017), 805–816.

[30] Yanhao Wang, Yuchen Li, and Kian-Lee Tan. 2018. Efficient Representative Subset Selection over Sliding Windows. *IEEE Trans. Knowl. Data Eng.* (2018).

[31] Yanhao Wang, Yuchen Li, and Kian-Lee Tan. 2019. Coresets for Minimum Enclosing Balls over Sliding Windows. (2019). arXiv:1905.03718

[32] Xunkai Wei and Yinghong Li. 2008. Theoretical Analysis of a Rigid Coreset Minimum Enclosing Ball Algorithm for Kernel Regression Estimation. In *ISNN*. 741–752.

[33] Emre Alper Yildirim. 2008. Two Algorithms for the Minimum Enclosing Ball Problem. *SIAM J. Optim.* 19, 3 (2008), 1368–1391.

[34] Hamid Zarrabi-Zadeh and Timothy M. Chan. 2006. A Simple Streaming Algorithm for Minimum Enclosing Balls. In *CCCG*. 139–142.

[35] Yan Zheng and Jeff M. Phillips. 2017. Coresets for Kernel Regression. In *KDD*. 645–654.

A DETAILS ON EXPERIMENTAL SETUP

In this section, we present the details of our experimental setup for reproducibility.

A.1 Hardware Configuration

All the experiments are conducted on a server with the following specifications:

- **CPU:** Intel(R) Xeon(R) E7-4820 v3 @ 1.90GHz
- **Memory:** 128GB (8×16GB) RAM 2133MHz DDR4 memory
- **Hard Disk:** a 480GB SATA-III solid-state drive

A.2 Software Environment

The server runs Ubuntu GNU/Linux 16.04.3 LTS 64-bit with kernel v4.11.0-rc2. All the code is written in Java 8 only using the standard libraries. No third-party software/library is required. The version of JVM for compilation is Java HotSpot(TM) 64-Bit Server VM 18.3 (build 10.0.2+13). In the experiments, each instance is limited to use a single thread for computation to ensure the fairness of comparison. In addition, we use a JVM option "`-Xmx80000m`" to restrict the maximum heap size used by each instance. Note that the memory usage is much less than 80GB and our purpose is to guarantee that the bottleneck is not in memory and I/O.

A.3 Datasets

The statistics of the datasets used in our experiments are listed in Table 3. Here we briefly describe the real-world datasets we use and the preprocessing procedures.

- Census contains a one percent sample of the Public Use Microdata Samples person records drawn from the full 1990 census sample. It is downloaded from UCI Machine Learning Repository.
- CovType is a dataset for predicting forest cover type from cartographic variables. We download the dataset from LIBSVM. In the preprocessing, we only retain the points with class label "0".
- GIST is an image collection retrieved from TEXMAX.
- Gowalla is a collection of user check-ins over the period of February 2009 – October 2010 on gowalla.com. It is downloaded from SNAP. In the preprocessing, we extract the latitude and longitude of each check-in as a two-dimensional point and dispose other attributes.
- HIGGS is a dataset for distinguishing between a signal process which produces Higgs bosons and a background process which does not. It is downloaded from UCI Machine Learning Repository. In the preprocessing, we only retain the points with class label "1" (i.e., signal process).
- SIFT is an image collection retrieved from TEXMAX.

The procedure of generating Synthetic has been introduced in the main paper. We transform each dataset into a single file that stores the points in dense format: each point is represented by one line in the file and different dimensions of the point are split by a single space. After dataset preprocessing and format transformation, we shuffle all points of each dataset randomly. It is guaranteed that a dataset must be processed by all algorithms in the same order for the fairness of comparison.

A.4 Implementation Issues

The algorithms we compare in the experiments are as listed in Section 4.1. Here we discuss the implementations of these algorithms.

We use the Java code published by the authors, which is available at https://github.com/hbf/miniball, for the implementation of COMEB [18]. All the other algorithms are implemented by ourselves. First of all, the basic scheme of CoreMEB [4] is presented in Algorithm 1. Our practical implementations are based on two improved versions of CoreMEB, i.e., Figure 2 in [23] for Euclidean MEB and Algorithm 4.1 in [33] for kernelized MEB. They use the same scheme as shown in Algorithm 1 but have lower computational costs and quicker convergence rates. The implementation of SSMEB is based on Section 2 of [34], which is extended to kernelized MEB according to Section 4.2 of [27]. We implement the Blurred Ball Cover (BBC) algorithm according to Section 2 of [2]. In addition, DyMEB is implemented based on Algorithms 1–3 in [12]. Finally, our proposed methods, i.e., AOMEB, SWMEB and SWMEB+, are implemented according to Algorithms 2–4 in this paper.

A.5 Parameter Tuning

The procedure of parameter tuning is as follows. Generally, there are three parameters used in our experiments: ε_1 (or ε) in all algorithms except COMEB and SSMEB, ε_2 in SWMEB and SWMEB+, the partition size L in SWMEB.

First of all, we choose appropriate ε_1 for Euclidean and kernelized MEBs respectively. The parameter ε_1 determines the trade-off between coreset quality and efficiency. We test the effect of ε_1 by validating on $[10^{-1}, 10^{-2}, \ldots, 10^{-6}]$. For Euclidean MEB, the coreset quality hardly improves but the running time increases rapidly when $\varepsilon_1 < 10^{-3}$. Therefore, we set $\varepsilon_1 = 10^{-3}$ for Euclidean MEB. Using the same method, we set $\varepsilon_1 = 10^{-4}$ for kernelized MEB.

After choosing appropriate ε_1's, we further test the remaining parameters in SWMEB and SWMEB+. Specifically, ε_2 is also selected from $[10^{-1}, 10^{-2}, \ldots, 10^{-6}]$. In SWMEB, ε_2 affects the index construction on each partition. When $\varepsilon_2 \leq 0.01$, SWMEB suffers from high overhead because of containing too many indices. Thus, we use $\varepsilon_2 = 0.1$ for SWMEB. Additionally, to avoid the indices being too sparse, we restrict the maximal distance between any neighboring indices in the same partition to $\frac{L}{10}$. In SWMEB+, ε_2 adjusts the number of indices in X_t. Firstly, we observe that the coreset quality cannot be improved any more when $\varepsilon_2 < \frac{\varepsilon_1}{10}$. Secondly, we scale ε_2 by a factor of $\lambda > 1$ among indices, i.e., $\varepsilon_2 = \lambda^{i-1} \cdot \frac{\varepsilon_1}{10}$ for $x_i \in X_t$, to reduce the index size without seriously affecting the quality. We select λ from $\{2, 4, 8, 16\}$ and use $\lambda = 4$ for SWMEB+ since the quality seriously degrades when $\lambda > 4$. Thirdly, we set 0.1 as the upper bound of ε_2 to ensure the theoretical soundness. To sum up, we use $\varepsilon_2 = \min(4^{i-1} \cdot \frac{\varepsilon_1}{10}, 0.1)$ for each $x_i \in X_t$ in SWMEB+. Finally, the partition size L in SWMEB affects the balance between space and time complexity. If L is smaller, W_t will be divided into more partitions, which leads to more indices in X_t, but fewer points will be stored in the buffer Q, and vice versa. We try to select L from range $[\frac{N}{5}, \frac{N}{10}, \ldots]$. The results show that SWMEB cannot scale to large datasets when $L < \frac{N}{10}$ for too many indices in X_t. Therefore, the partition size L in SWMEB is set to $\frac{N}{10}$.