Submodular Optimization Over Streams with Inhomogeneous Decays

Junzhou Zhao, Shuo Shang, Pinghui Wang, John C.S. Lui, and Xiangliang Zhang

1King Abdullah University of Science and Technology, KSA
2Inception Institute of Artificial Intelligence, UAE
3Xi’an Jiaotong University, China
4The Chinese University of Hong Kong, Hong Kong
{junzhou.zhao, xiangliang.zhang}@kaust.edu.sa, jedi.shang@gmail.com, phwang@mail.xjtu.edu.cn, cslui@cse.cuhk.edu.hk

Abstract
Cardinality constrained submodular function maximization, which aims to select a subset of size at most \( k \) to maximize a monotone submodular utility function, is the key in many data mining and machine learning applications such as data summarization and maximum coverage problems. When data is given as a stream, streaming submodular optimization (SSO) techniques are desired. Existing SSO techniques can only apply to insertion-only streams where each element has an infinite lifespan, and sliding-window streams where each element has a same lifespan (i.e., window size). However, elements in some data streams may have arbitrary different lifespans, and this requires addressing SSO over streams with inhomogeneous-decays (SSO-ID). This work formulates the SSO-ID problem and presents three algorithms: \textsc{BasicStreaming} is a basic streaming algorithm that achieves an \((1/2 - \epsilon)\) approximation factor; \textsc{HistApprox} improves the efficiency significantly and achieves an \((1/3 - \epsilon)\) approximation factor; \textsc{HistStreaming} is a streaming version of \textsc{HistApprox} and uses heuristics to further improve the efficiency. Experiments conducted on real data demonstrate that \textsc{HistStreaming} can find high quality solutions and is up to two orders of magnitude faster than the naive \textsc{Greedy} algorithm.

1 Introduction
Selecting a subset of data to maximize some utility function under a cardinality constraint is a fundamental problem facing many data mining and machine learning applications. In myriad scenarios, ranging from data summarization (Mitrovic et al. 2018), to search results diversification (Agrawal et al. 2009), to feature selection (Brown et al. 2012), to coverage maximization (Cormode, Karloff, and Wirth 2010), utility functions commonly satisfy submodularity (Nemhauser, Wolsey, and Fisher 1978), which captures the diminishing returns property. It is therefore not surprising that submodular optimization has attracted a lot of interests in recent years (Krause and Golovin 2014).

Fig. 1: Insertion-only stream: each element has an infinite lifespan. Sliding-window stream: each element has a same initial lifespan. Our model: each element can have an arbitrary lifespan.

If data is given in advance, the \textsc{Greedy} algorithm can be applied to solve submodular optimization in a batch mode. However, today’s data could be generated continuously with no ending, and in some cases, data is produced so rapidly that it cannot even be stored in computer main memory, e.g., Twitter generates more than 8,000 tweets every second (Twitter 2018). Thus, it is crucial to design streaming algorithms where at any point of time the algorithm has access only to a small fraction of data. To this end, streaming submodular optimization (SSO) techniques have been developed for insertion-only streams where a subset is selected from all historical data (Badanidiyuru et al. 2014), and sliding-window streams where a subset is selected from the most recent data only (Epasto et al. 2017).

We notice that these two existing streaming settings, i.e., insertion-only stream and sliding-window stream, actually represent two extremes. In insertion-only streams, a subset is selected from all historical data elements which are treated as of equal importance, regardless of how outdated they are. This is often undesirable because the stale historical data is usually less important than fresh and recent data. While in sliding-window streams, a subset is selected from the most recent data only and historical data outside of the window is completely discarded. This is also sometimes undesirable because one may not wish to completely lose the entire history of past data and some historical data may be still important. As a result, SSO over insertion-only streams may find solutions that are not fresh; while SSO over sliding-window...
streams may find solutions that exclude historical important data or include many recent but valueless data. Can we design SSO techniques with a better streaming setting?

We observe that both insertion-only stream and sliding-window stream actually can be unified by introducing the concept of data lifespan, which is the amount of time an element participating in subset selection. As time advances, an element’s remaining lifespan decreases. When an element’s lifespan becomes zero, it is discarded and no longer participates in subset selection. Specifically, in insertion-only streams, each element has a finite lifespan and will always participate in subset selection after arrival. While in sliding-window streams, each element has a same initial lifespan (i.e., the window size), and hence participates in subset selection for a same amount of time (see Fig. 1).

We observe that in some real-world scenarios, it may be inappropriate to assume that each element in a data stream has a same lifespan. Let us consider the following scenario.

**Motivating Example.** Consider a news aggregation website such as Hacker News (HN 2018) where news submitted by users form a news stream. Interesting news may attract users and always participate in subset selection. While in sliding-window streams, each element has a same initial lifespan (i.e., the window size), and hence participates in subset selection for a same amount of time (see Fig. 1).

Therefore, besides timestamp of each data element, lifespan of each data element should also be considered in subset selection. Other similar scenarios include hot video selection from YouTube (where each video may have its own lifespan), and trending hashtag selection from Twitter (where each hashtag may have a different lifespan).

**Overview of Our Approach.** We propose to extend the two extreme streaming settings to a more general streaming setting where each element is allowed to have an arbitrary initial lifespan and thus each element can participate in subset selection for an arbitrary amount of time (see Fig. 1). We refer to this more general decaying mechanism as inhomogeneous decay, in contrast to the homogeneous decay adopted in sliding-window streams. This work presents three algorithms to address SSO over streams with inhomogeneous decays (SSO-ID). We first present a simple streaming algorithm, i.e., BASICSTREAMING. Then, we present HISTAPPROX to improve the efficiency significantly. Finally, we design a streaming version of HISTAPPROX, i.e., HISTSTREAMING. We theoretically show that our algorithms have constant approximation factors.

Our main contributions include:

- We propose a general inhomogeneous-decaying streaming model that allows each element to participate in subset selection for an arbitrary amount of time.
- We design three algorithms to address the SSO-ID problem with constant approximation factors.
- We conduct experiments on real data, and the results demonstrate that our method finds high quality solutions and is up to two orders of magnitude faster than GREEDY.

## 2 Problem Statement

**Data Stream.** A data stream comprises an unbounded sequence of elements arriving in chronological order, denoted by $\{v_1, v_2, \ldots\}$. Each element is from set $V$, called the ground set, and each element $v$ has a discrete timestamp $t_v \in \mathbb{N}$. It is possible that multiple data elements arriving at the same time. In addition, there may be other attributes associated with each element.

**Inhomogeneous Decay.** We propose an inhomogeneous-decaying data stream (IDS) model to enable inhomogeneous decays. For an element $v$ arrived at time $t_v$, it is assigned an initial lifespan $l(v, t_v) \in \mathbb{N}$ representing the maximum time span that the element will remain active. As time advances to $t \geq t_v$, the element’s remaining lifespan decreases to $l(v, t) = l(v, t_v) - (t - t_v).$ If $l(v, t') = 0$ at some time $t'$, $v$ is discarded. We will assume $l(v, t_v)$ is given as an input to our algorithm. At any time $t$, active elements in the stream form a set, denoted by $S_t \equiv \{v: v \in V \land t_v \leq t \land l(v, t) > 0\}$.

IDS model is general. If $l(v, t_v) = \infty, \forall v$, an IDS becomes an insertion-only stream. If $l(v, t_v) = W, \forall v$, an IDS becomes a sliding-window stream. If $l(v, t_v)$ follows a geometric distribution parameterized by $p$, i.e., $P(l(v, t_v) = l) = (1 - p)^l - 1 p$, it is equivalent of saying that an active element is discarded with probability $p$ at each time step.

To simplify notations, if time $t$ is clear from context, we will use $l_v$ to represent $l(v, t)$, i.e., the remaining lifespan (or just say “the lifespan”) of element $v$ at time $t$.

**Monotone Submodular Function** (Nemhauser, Wolsey, and Fisher 1978). A set function $f: 2^V \rightarrow \mathbb{R}_{\geq 0}$ is submodular if $f(S \cup \{s\}) - f(S) \geq f(T \cup \{s\}) - f(T)$, for all $S \subseteq T \subseteq V$ and $s \in V \setminus T$. $f$ is monotone (non-decreasing) if $f(S) \leq f(T)$ for all $S \subseteq T \subseteq V$. Without loss of generality, we assume $f$ is normalized, i.e., $f(\emptyset) = 0$.

Let $\delta(s|S) \equiv f(S \cup \{s\}) - f(S)$ denote the marginal gain of adding element $s$ to $S$. Then monotonicity is equivalent of saying that the marginal gain of every element is always non-negative, and submodularity is equivalent of saying that marginal gain $\delta(s|S)$ of element $s$ never increases as set $S$ grows bigger, aka the diminishing returns property.

**Streaming Submodular Optimization with Inhomogeneous Decays (SSO-ID).** Equipped with the above notations, we formulate the cardinality constrained SSO-ID problem as follows:

$$\text{OPT}_t \triangleq \max_S f(S), \quad \text{s.t.} \quad S \subseteq S_t \land |S| \leq k,$$

where $k$ is a given budget.

**Remark.** The SSO-ID problem is NP-hard, and active data $S_t$ is continuously evolving with outdated data being discarded and new data being added in at every time $t$, which further complicates the algorithm design. A naive algorithm to solve the SSO-ID problem is that, when $S_t$ is updated, we re-run GREEDY on $S_t$ from scratch, and this approach outputs a solution that is $(1 - 1/e)$-approximate. However, it needs $O(k|S_t|)$ utility function evaluations at each time step, which is unaffordable for large $S_t$. Our goal is to find faster algorithms with comparable approximation guarantees.
3 Algorithms

This section presents three algorithms to address the SSO-ID problem. Due to space limitation, the proofs of all theorems are included in the extended version of this paper.

3.1 Warm-up: The BasicStreaming Algorithm

In the literature, SIEVESTREAMING (Badanidiyuru et al. 2014) is designed to address SSO over insertion-only streams. We leverage SIEVESTREAMING as a basic building block to design a BasicStreaming algorithm. BasicStreaming is simple per se and may be inefficient, but offers opportunities for further improvement. This section assumes lifespan is upper bounded by \( L \), i.e., \( l_v \leq L, \forall v \).

We later remove this assumption in the following sections.

SIEVESTREAMING (Badanidiyuru et al. 2014) is a threshold-based streaming algorithm for solving cardinality constrained SSO over insertion-only streams. The high level idea is that, for each coming element, it is selected only if its gain w.r.t. a set is no less than a threshold. In its implementation, SIEVESTREAMING lazily maintains a set of \( \log_{1+p} 2k = O(\epsilon^{-1} \log k) \) thresholds and each is associated with a candidate set initialized empty. For each incoming element, its marginal gain w.r.t. each candidate set is computed; if the gain is no less than the corresponding threshold and the candidate set is not full, the element is added in the candidate set. At any time, a candidate set having the maximum utility is the current solution. SIEVESTREAMING achieves an \((1/2 - \epsilon)\) approximation guarantee.

Algorithm Description. We show how SIEVESTREAMING can be used to design a BasicStreaming algorithm to solve the SSO-ID problem. Let \( V_t \) denote a set of elements arrived at time \( t \). We partition \( V_t \) into (at most) \( L \) non-overlapping subsets, i.e., \( V_t = \bigcup_{i=1}^L V_i^{(t)} \) where \( V_i^{(t)} \) is the subset of elements with lifespan \( i \) at time \( t \). BasicStreaming maintains \( L \) SIEVESTREAMING instances, denoted by \( \{A_i^{(t)}\}_{i=1}^L \), and alternates a data update step and a time update step to process the arriving elements \( V_t \).

• Data Update. This step processes arriving data \( V_t \). Let instance \( A_i^{(t)} \) only process elements with lifespan no less than \( l \). In other words, elements in \( \bigcup_{i \geq l} V_i^{(t)} \) are fed to \( A_i^{(t)} \). After processing \( V_i \), \( A_i^{(t)} \) outputs the current solution.

• Time Update. This step prepares for processing the upcoming data in the next time step. We reset instance \( A_i^{(t)} \), i.e., empty its threshold set and each candidate set. Then we conduct a circular shift operation: \( A_1^{(t+1)} \leftarrow A_2^{(t)}, A_2^{(t+1)} \leftarrow A_3^{(t)}, \ldots, A_{L}^{(t+1)} \leftarrow A_1^{(t)} \).

BasicStreaming alternates the two steps and continuously processes data at each time step. We illustrate BasicStreaming in Fig. 2, with pseudo-code given in Alg. 1.

Analysis. BasicStreaming exhibits a feature that an instance gradually expires (and is reset) as data processed in it expires. Such a feature ensures that, at any time \( t \), \( A_i^{(t)} \) always processed all the data in \( S_t \). Because \( A_i^{(t)} \) is a SIEVESTREAMING instance, we immediately have the following conclusions.

![Fig. 2: BasicStreaming](image)

**Algorithm 1: BasicStreaming**

| Input: An IDS of data elements arriving over time |
| Output: A subset \( S_t \) at any time \( t \) |

1. Initialize \( L \) SIEVESTREAMING instances \( \{A_i^{(1)}\}_{i=1}^L \).
2. for \( t = 1, 2, \ldots \)
3.   for \( l = 1, \ldots, L \) do
4.     Feed \( A_i^{(t)} \) with data \( \bigcup_{i \geq l} V_i^{(t)} \); // data update
5.     \( S_t \) ← output of \( A_i^{(t)} \); // time update
6.   for \( l = 2, \ldots, L \) do \( A_{l-1}^{(t+1)} \leftarrow A_l^{(t)} \); // time update
7. Reset \( A_1^{(t)} \) and \( A_1^{(t+1)} \leftarrow A_1^{(t)} \); // time update

**Theorem 1.** BasicStreaming achieves an \((1/2 - \epsilon)\) approximation guarantee.

**Theorem 2.** BasicStreaming uses \( O(Lc^{-1} \log k) \) time to process each element, and \( O(Lk\epsilon^{-1} \log k) \) memory to store intermediate results (i.e., candidate sets).

Remark. As illustrated in Fig. 2, data with lifespan \( l \) will be fed to \( \{A_i^{(t)}\}_{i \leq l} \). Hence, elements with large lifespans will fan out to a large fraction of SIEVESTREAMING instances, and incur high CPU and memory usage, especially when \( L \) is large. This is the main bottleneck of BasicStreaming. On the other hand, elements with small lifespans only need to be fed to a few instances. Therefore, if data lifespans are mainly distributed over small values, e.g., power-law distributed, then BasicStreaming is still efficient.

3.2 HistApprox: Improving Efficiency

To address the bottleneck of BasicStreaming when processing data with a large lifespan, we design HistApprox in this section. HistApprox can significantly improve the efficiency of BasicStreaming but requires active data \( S_t \) to be stored in RAM

1\(^{1}\)

Strictly speaking, HistApprox is not a streaming algorithm. We later remove the assumption of storing \( S_t \) in RAM in the next section.

**Basic Idea**. If at any time, only a few instances are maintained and running in BasicStreaming, then both CPU time and memory usage will decrease. Our idea is hence to selectively maintain a subset of SIEVESTREAMING instances that can approximate the rest. Roughly speaking, this

1\(^{1}\)For example, if lifespan follows a geometric distribution, i.e., \( P(l_t = l) = (1-p)p^{l-1}, l = 1, 2, \ldots, \) and at most \( M \) elements arrive at a time, then \( |S_t| \leq \sum_{l=0}^{t-1} M p^l \leq \frac{M}{1-p} \). Hence, if RAM is larger than \( \frac{M}{1-p} \), \( S_t \) actually can be stored in RAM even as \( t \to \infty \).
idea can be thought of as using a histogram to approximate a curve. Specifically, let \( g_t(l) \) denote the value of output of \( A_l(t) \) at time \( t \). For very large \( L \), we can think of \( \{g_t(l)\}_{l \geq 1} \) as a “curve” (e.g., the dashed curve in Fig. 3). Our idea is to pick a few instances as active instances and construct a histogram to approximate this curve, as illustrated in Fig. 3.

![Fig. 3: Approximate \( \{g_t(l)\}_{l \geq 1} \) by \( \{g_t(l)\}_{l \in x_t} \).](image)

The challenge is that, as new data keeps arriving, the curve is changing; hence, we need to update the histogram accordingly to make sure that the histogram always well approximates the curve. Let \( x_t \equiv \{x_t^1, x_t^2, \ldots \} \) index a set of active instances at time \( t \), where each index \( x_t^i \geq 1 \). In the follows, we describe the \( x_t \) updating method, i.e., \( \text{HISTAPPROX} \), and the method guarantees that the maintained histogram satisfies our requirement.

**Algorithm Description.** \( \text{HISTAPPROX} \) consists of two steps: (1) updating indices; (2) removing redundant indices.

* **Updating Indices.** The algorithm starts with an empty index set, i.e., \( x_1 = \emptyset \). At time \( t \), consider a set of newly arrived elements \( V_t(t_l) \) with lifespan \( t \). These elements will increase the curve before \( t \) (because data \( V_t(t_l) \) will be fed to \( \{A_l(t_i)\}_{i \leq l} \), see Fig. 2). There are three cases based on the position of \( l \), as illustrated in Fig. 3.

  - **Case 1.** If \( l \in x_t \), we simply feed \( V_t(t_l) \) to \( \{A_l(t_i)\}_{i \in x_t, i \leq l} \).

  - **Case 2.** If \( l \notin x_t \) and \( l \) has no successor in \( x_t \), we create a new instance \( A_l(t_l) \) and feed \( V_t(t_l) \) to \( \{A_l(t_i)\}_{i \in x_t, i \leq l} \).

  - **Case 3.** If \( l \notin x_t \) and \( l \) has a successor \( l_2 \in x_t \). Let \( A_l(t_2) \) be a copy of \( A_l(t_1) \), then we feed \( V_t(t_2) \) to \( \{A_l(t_i)\}_{i \in x_t, i \leq l} \). Note that \( A_l(t_2) \) needs to process all data with lifespan \( \geq l \) at time \( t \). Because \( A_l(t_2) \) has processed all data with lifespan \( \geq l_2 \), we still need to feed \( A_l(t_1) \) with historical data s.t. their lifespan \( \in [l, l_2] \). That is the reason we need \( S_t \) to be stored in RAM.

  Above scheme guarantees that each \( A_l(t_i), l \in x_t \) processed all the data with lifespan \( \geq l \) at time \( t \). The detailed pseudo-code is given in procedure `Process` of Alg. 2.

* **Removing Redundant Indices.** Intuitively, if the outputs of two instances are close to each other, it is not necessary to keep both of them. We need the following definition to quantify redundancy.

**Definition 1** (\( \epsilon \)-redundancy). At time \( t \), consider two instances \( A_{l_1}(t) \) and \( A_{l_2}(t) \) with \( i < l \). We say \( A_{l_1}(t) \) is \( \epsilon \)-redundant if their exists \( j > l \) such that \( g_t(j) \geq (1 - \epsilon)g_t(i) \).

The above definition simply states that, since \( A_{l_1}(t) \) and \( A_{l_2}(t) \) are already close with each other, then instances between them are redundant. In \( \text{HISTAPPROX} \), we regularly check the output of each instance and terminate those redundant ones, as described in `ReduceRedundancy` of Alg. 2.

**Analysis.** Notice that indices \( x_t \in x_t \) and \( x_t + 1 \in x_{t-1} \) are actually the same index (if they both exist) but appear at different time. In general, we say \( x^t \in x^t \) is an ancestor of \( x \in x_t \) if \( x^t \leq t \). The follows, let \( x^t \) denote \( x^t \)'s ancestor at time \( t^t \). First, \( \text{HISTAPPROX} \) maintains a histogram satisfying the following property.

**Lemma 1.** For two consecutive indices \( x_t, x_{t+1} \in x_t \) at any time \( t \), one of the following two cases holds:

\( \text{C1} \) \( S_t \) contains no data with lifespan \( \in [x_t, x_{t+1}] \).
\( \text{C2} \) \( g_t(x_t^j) \geq (1 - \epsilon)g_t(x_t^i) \) for some time \( t' \leq t \), and from time \( t' \) to \( t \), there is no data with lifespan between the two indices arrived (exclusive).

Histogram with property C2 is known as a smooth histogram (Braverman and Ostrovsky 2007). Smooth histogram together with the submodularity of \( f \) are sufficient to ensure a constant factor approximation guarantee of \( g_t(x_t) \).

**Theorem 3.** \( \text{HISTAPPROX} \) is \((1/3 - \epsilon)\)-approximate, i.e., at any time \( t \), \( g_t(x_t) \geq (1/3 - \epsilon)\text{OPT}_t \).

**Theorem 4.** \( \text{HISTAPPROX} \) uses \( O(\epsilon^{-2} \log^2 k) \) time to process each coming element and \( O(k \epsilon^{-2} \log^2 k) \) memory to store intermediate results and \( |S_t| \) memory to store \( S_t \).

**Remark.** Because we use a histogram to approximate a curve, \( \text{HISTAPPROX} \) has a weaker approximation guarantee than \( \text{BASISTREAMING} \). In experiments, we observe that...
3.3 **HISTSTREAMING: A Heuristic Streaming Algorithm**

Based on HISTAPPROX, this section presents a streaming algorithm HISTSTREAMING, which uses heuristics to further improve the efficiency of HISTAPPROX. HISTSTREAMING no longer requires storing active data $S_t$ in memory.

**Basic Idea.** If we do not need to process the historical data in HISTAPPROX (Line 14), then there is no need to store $S_t$. What if $A_i^{(t)}$ does not process historical data? Because $A_i^{(t)}$ does not process all the data with lifespan $\geq i$ in $S_t$, there will be a bias between its actual output $\hat{g}_i(l)$ and expected output $g_i(l)$. We only need to worry about the case $\hat{g}_i(l) < g_i(l)$, as the other case $\hat{g}_i(l) \geq g_i(l)$ means that without processing historical data, $A_i^{(t)}$ finds even better solutions (which may rarely happen in practice but indeed possible). In the follows, we apply two useful heuristics to design HISTSTREAMING, and show that historical data can be ignored due to its insignificance and submodularity of objective function.

**Effects of historical data.** Intuitively, if historical data is insignificant, then a SIEVESTREAMING instance may not need to process it at all, and can still output quality guaranteed solutions. We notice that, in HISTAPPROX, a newly created instance $A_i^{(t)}$ essentially needs to process three substreams: (1) elements arrived before $t$ with lifespan $\leq l_2$ (Line 13); (2) unprocessed historical elements with lifespan $\in [l, l_2)$ (Line 14); (3) newly arrived elements $V_t$ (Line 16). Denote these three substreams by $S_1$, $S_2$ and $S_3$, respectively. We state a useful lemma below.

**Lemma 2.** Let $S_1||S_2||S_3$ denote the concatenation of three substreams $S_1$, $S_2$, $S_3$. Let $A(S)$ denote the output value of applying SIEVESTREAMING algorithm $A$ on stream $S$. If $A(S_1) \geq \alpha A(S_1||S_2)$ for $0 < \alpha < 1$, then $A(S_1||S_3) \geq (1/4 - \epsilon)\alpha\text{OPT}$ where OPT is the value of an optimal solution in stream $S_1||S_2||S_3$.

**Algorithm Description.** We only need to slightly modify Process and ReduceRedundancy (see Alg. 3).

---

### Alg. 3: HISTSTREAMING

<table>
<thead>
<tr>
<th>Procedure Process ($V_t^{(t)}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $\delta_l \leftarrow 0$;</td>
</tr>
<tr>
<td>2. if $l \notin x_i$ then</td>
</tr>
<tr>
<td>3. $\delta_l \leftarrow 0$;</td>
</tr>
<tr>
<td>4. $\delta_l \leftarrow 0$;</td>
</tr>
<tr>
<td>5. $\delta_l \leftarrow 0$;</td>
</tr>
<tr>
<td>6. Find $i, j \in x_t$ s.t. $l \in (i, j)$ and $\delta_{ij}$ is recorded,</td>
</tr>
<tr>
<td>7. then let $\delta_l \leftarrow \delta_{ij}$;</td>
</tr>
<tr>
<td>8. $\delta_{ij}$ $\leftarrow \delta_{ij}$;</td>
</tr>
<tr>
<td>9. $\delta_{ij}$ $\leftarrow \delta_{ij}$;</td>
</tr>
<tr>
<td>10. $\delta_{ij}$ $\leftarrow \delta_{ij}$;</td>
</tr>
<tr>
<td>11. $\delta_{ij}$ $\leftarrow \delta_{ij}$;</td>
</tr>
<tr>
<td>12. $\delta_{ij}$ $\leftarrow \delta_{ij}$;</td>
</tr>
</tbody>
</table>

redundant because $A(S_1||S_2)$ does not increase much upon $A(S_1)$. Hence, it makes sense to assume that historical data $S_2$ is insignificant, and by Lemma 2, $S_2$ can be ignored.

![Fig. 4: At time $t'$, data with lifespan $l_0'$ arrives and forms a redundant instance, which is removed. At time $t > t'$, data with lifespan $l$ arrives and $A_i^{(t)}$ is created. Data at $l_0$ becomes the unprocessed historical data. We thus say that unprocessed historical data is caused by the deletion of redundant instance at previous time.](image-url)
Remark. HistStreaming uses heuristics to further improve the efficiency of HistApprox, and no longer needs to store $S_t$ in memory. In experiments, we observe that HistStreaming can find high quality solutions.

4 Experiments

In this section, we construct several maximum coverage problems to evaluate the performance of our methods. We use real world and public available datasets. Note that the optimization problems defined on these datasets may seem to be simplistic, as our main purpose is to validate the performance of proposed algorithms, and hence we want to keep the problem settings as simple and clear as possible.

4.1 Datasets

DBLP. We construct a representative author selection problem on the DBLP dataset (DBLP 2018), which records the meta information of about 3 million papers, including 1.8 million authors and 5,079 conferences from 1936 to 2018. We say that an author represents a conference if the author published papers in the conference. Our goal is to maintain a small set of $k$ authors that jointly represent the maximum number of distinct conferences at any time. We filter out authors that published less than 10 papers and sort the remaining 188,383 authors by their first publication date to form an author stream. On this dataset, an author’s lifespan could be defined as the time period between its first and last publication dates.

StackExchange. We construct a hot question selection problem on the math.stackexchange.com website (StackExchange 2018). The dataset records about 1.3 million questions with 152 thousand commenters from 7/2010 to 6/2018. We say a question is hot if it attracts many commenters to comment. Our goal is select a small set of $k$ questions that jointly attract the maximum number of distinct commenters at any time. The questions are ordered by the post date, and the lifespan of a question can be defined as the time interval length between its first and last comment time.

4.2 Settings

Benchmarks. We consider the following two methods as benchmarks.

- **Greedy.** We re-run Greedy on the active data $S_t$ at each time $t$, and apply the lazy evaluation trick (Minoux 1978) to further improve its efficiency. Greedy will serve as an upper bound.

- **Random.** We randomly pick $k$ elements from active data $S_t$ at each time $t$. Random will serve as a lower bound.

Efficiency Measure. When evaluating algorithm efficiency, we follow the previous work (Badanidiyuru et al. 2014) and record the number of utility function evaluations, i.e., the number of oracle calls. The advantage of this measure is that it is independent of the concrete algorithm implementation and platform.

Lifespan Generating. In order to test the algorithm performance under different lifespan distributions, we also consider generating data lifespans by sampling from a geometric distribution, i.e., $P(l = 1) = (1 - p)^{l-1}p, l = 1, 2, \ldots$. Here $0 < p < 1$ controls the skewness of geometric distribution, i.e., larger $p$ implies that a data element is more likely to have a small lifespan.

4.3 Results

Analyzing BasicStreaming. Before comparing the performance of our algorithms with benchmarks, let us first study the properties of BasicStreaming, as it is the basis of HistApprox and HistStreaming. We mainly analyze how lifespan distribution affects the performance of BasicStreaming. To this end, we generate lifespans from $Geo(p)$ with varying $p$, and truncate the lifespan at $L = 1,000$. We run the three proposed algorithms for 100 time steps and maintain a set with cardinality $k = 10$ at every time step. We set $\epsilon = 0.1$. The solution value and number of oracle calls (at time $t = 100$) are depicted in Figs. 5 and 6, respectively.

![Fig. 5: Solution value comparison (higher is better)](image1)

![Fig. 6: Oracle calls comparison (lower is better)](image2)

Figure 5 states that the outputs of the three methods are always close with each other under different lifespan distributions, i.e., they always output similar quality solutions. Fig. 6 states that BasicStreaming requires much more oracle calls than the other two methods, indicating that BasicStreaming is less efficient than the other two methods. We also observe that, as $p$ increases (hence more data elements tend to have small lifespans), the number of oracle calls of BasicStreaming decreases. This confirms our previous analysis that BasicStreaming is efficient when
most data elements have small lifespans. We also observe that HistAPPROX and HistSTREAMING are not quite sensitive to lifetime distribution, and they are much more efficient than HistSTREAMING. In addition, we observe that HistSTREAMING is slightly faster than HistAPPROX even though HistSTREAMING uses smaller RAM.

This experiment demonstrates that BasicSTREAMING, HistAPPROX, and HistSTREAMING find solutions with similar quality, but HistAPPROX and HistSTREAMING are much more efficient than BasicSTREAMING.

**Performance Over Time.** In the next experiment, we focus on analyzing the performance of HistSTREAMING. We fix the lifespan distribution to be Geo(0.001) with \( L = 10,000 \), and run each method for 5,000 time steps to maintain a set with cardinality \( k = 10 \). Figs. 7 and 8 depict the solution value and ratio of the number of oracle calls (w.r.t. Greedy), respectively.

![Fig. 7: Solution value over time (higher is better)](image)

![Fig. 8: Oracle calls ratio over time (lower is better)](image)

Figure 7 shows that Greedy and Random always find the best and worst solutions, respectively, which is expected. HistSTREAMING finds solutions that are close to Greedy. Small \( \epsilon \) can further improve the solution quality. In Fig. 8, we show the ratio of cumulative number of oracle calls between HistSTREAMING and Greedy. It is clear to see that HistSTREAMING uses quite a small number of oracle calls comparing with Greedy. Larger \( \epsilon \) further improves efficiency, and for \( \epsilon = 0.2 \) the speedup of HistSTREAMING could be up to two orders of magnitude faster than Greedy.

This experiment demonstrates that HistSTREAMING finds solutions with quality close to Greedy and is much more efficient than Greedy. \( \epsilon \) can trade off between solution quality and computational efficiency.

![Performance under Different Budget](image)

**Performance under Different Budget** \( k \). Finally, we conduct experiments to study the performance of HistSTREAMING under different budget \( k \). Here, we choose the lifespan distribution as the same as the previous experiment, and set \( \epsilon = 0.2 \). We run HistSTREAMING and Greedy for 1000 time steps and compute the ratios of solution value and number of oracle calls between HistSTREAMING and Greedy. The results are depicted in Fig. 9.

![Fig. 9: Ratios under different budget](image)

In general, using different budgets, HistSTREAMING always finds solutions that are close to Greedy, i.e., larger than 80%; but uses very few oracle calls, i.e., less than 10%. Hence, we conclude that HistSTREAMING finds solutions with similar quality to Greedy, but is much efficient than Greedy, under different budgets.

## 5 Related Work

**Cardinality Constrained Submodular Function Maximization.** Submodular optimization lies at the core of many data mining and machine learning applications. Because the objectives in many optimization problems have a diminishing returns property, which can be captured by submodularity. In the past few years, submodular optimization has been applied to a wide variety of scenarios, including sensor placement (Krause, Singh, and Guestrin 2008), outbreak detection (Leskovec et al. 2007), search result diversification (Agrawal et al. 2009), feature selection (Brown et al. 2012), data summarization (Mirzasoleiman et al. 2015; Mitrovic et al. 2018), influence maximization (Kempe, Kleinberg, and Tardos 2003), just name a few. The Greedy algorithm (Nemhauser, Wolsey, and Fisher 1978) plays as a silver bullet in solving the cardinality constrained submodular maximization problem. Improving the efficiency of Greedy algorithm has also gained a lot of interests, such as lazy evaluation (Minoux 1978), disk-based optimization (Cornode, Karloff, and Wirth 2010), distributed computation (Epasto, Mirrokni, and Zadimoghaddam 2017; Kumar et al. 2013), sampling (Mirzasoleiman et al. 2015), etc.

**Streaming Submodular Optimization (SSO).** SSO is another way to improve the efficiency of solving submodular optimization problems, and are gaining interests in recent years due to the rise of big data and high-speed streams that an algorithm can only access a small fraction of the data at a time point. Kumar et al. (2013) design streaming algorithms that need to traverse the streaming data for a
few rounds which is suitable for the MapReduce framework. Badanidiyuru et al. (2014) then design the SIEVE STREAMING algorithm which is the first one round streaming algorithm for insertion-only streams. SIEVE STREAMING is adopted as the basic building block in our algorithms. SSO over sliding-window streams has recently been studied by Chen et al. (2016) and Epasto et al. (2017) respectively, that both leverage smooth histograms (Braverman and Ostrovsky 2007). Our algorithms actually can be viewed as a generalization of these existing methods, and our SSO techniques apply for streams with inhomogeneous decays.

Streaming Models. The sliding-window streaming model is proposed by Datar et al. (2002). Cohen et al. (2006) later extend the sliding-window model to general time-decaying model for the purpose of approximating summation aggregates in data streams (e.g., count the number of 1’s in a 01 stream). Cormode et al. (2009) consider the similar estimation problem by designing time-decaying sketches. These studies have inspired us to propose the IDS model.

6 Conclusion

When a data stream consists of elements with different lifespans, existing SSO techniques become inapplicable. This work formulates the SSO-ID problem, and presents three new SSO techniques to address the SSO-ID problem. BASICSTREAMING is simple and achieves an $1/2 - \epsilon$ approximation factor, but it may be inefficient. HISTAPPROX improves the efficiency of BASICSTREAMING significantly and achieves an $1/3 - \epsilon$ approximation factor, but it requires additional memory to store active data. HISTSTREAMING uses heuristics to further improve the efficiency of HISTAPPROX, and no longer requires storing active data in memory. In practice, if memory is not a problem, we suggest using HISTAPPROX as it has a provable approximation guarantee; otherwise, HISTSTREAMING is also a good choice.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments and suggestions to help us improve this paper. This work is financially supported by the King Abdullah University of Science and Technology (KAUST) Sensor Initiative, Saudi Arabia. The work of John C.S. Lui was supported in part by the GRF Funding 14208816.

References


Cormode, G.; Karloff, H.; and Wirth, A. 2010. Set cover algorithms for very large datasets. In CIKM.


Epasto, A.; Mirrokni, V.; and Zadimoghaddam, M. 2017. Bicriteria distributed submodular maximization in a few rounds. In SPAA.


Kumar, R.; Moseley, B.; Vassilvitskii, S.; and Vattani, A. 2013. Fast greedy algorithms in MapReduce and streaming. In SPAA.


Proof of Lemma 1

Proof. If \( x'_i \) became the successor of \( x'_i \) due to the removal of indices between them at some most recent time \( t' \leq t \), then procedure ReduceRedundancy in Alg. 2 guarantees that \( g_i(x'_i) \geq (1 - \epsilon)g_i(x'_i) \) after the removal at time \( t' \). From time \( t' \) to \( t \), it is also impossible to have data elements with lifespans between the two indices. Otherwise we will meet a contradiction: either these elements form redundant SIEVESTREAMING instances again thus \( t' \) is not the most recent time as claimed, or these elements form non-redundant SIEVESTREAMING instances thus \( x_i \) and \( x_{i+1} \) cannot be consecutive at time \( t \). We thus get C2.

Otherwise \( x'_{i+1} \) became the successor of \( x'_i \) when one of them is inserted in the histogram at some time \( t' \leq t \). Without lose of generality, let us assume \( x'_{i+1} \) is inserted after \( x'_i \) at time \( t' \). If elements with lifespans between the two indices arrive from time \( t' \) to \( t \), these elements must form redundant SIEVESTREAMING instances. We still get C2. Or, there is no element with lifespan between the two indices at all, i.e., \( S_t \) contains no data with lifespan between \( x_i \) and \( x_{i+1} \). We thus get C1. This completes the proof.

\[ \square \]

Proof of Theorem 3

Proof. If \( x_1 = 1 \) at time \( t \), then \( A_1^{(t)} \) exists. By the property of SIEVESTREAMING, we conclude that

\[ g_i(x_1) = g_i(1) \geq \frac{1}{2} \omega \text{OPT}_i. \]

Otherwise we have \( x_1 > 1 \) at time \( t \). If \( S_t \) contains elements with lifespan less than \( x_1 \), then \( A_1^{(t)} \) does not process all of the elements in \( S_t \), thus incurs a loss of solution quality. Our goal is to bound this loss.

Let \( x_0 \) denote the most recent expired predecessor of \( x_1 \) at time \( t \), and let \( t' < t \) denote the last time \( x_0 \)'s ancestor \( x'_0 \) was still alive, i.e., \( x'_0 = 1 \) at time \( t' \) (cf. Fig. 10). For ease of presentation, we commonly refer to \( x_0 \) and \( x_0 \)'s ancestors as the left index, and refer to \( x_1 \) and \( x_1 \)'s ancestors as the right index. Obviously, in time interval \( (t', t] \), no element with lifespan less than the right index arrives; otherwise, these elements would create new indices before the right index; then \( x_1 \) will not be the first index at time \( t \), or \( x_0 \) is not the most recent expired predecessor of \( x_1 \) at time \( t \).

\[ \text{Fig. 10: Indices relations at time } t'' \leq t' < t \text{. For ease of presentation, } x_0 \text{ (resp. } x_1 \text{) and its ancestors will be simply referred to as the left (right) index.} \]

Notice that \( x'_0 \) and \( x'_1 \) are two consecutive indices at time \( t' \). By Lemma 1, we have two cases.

- **If C1 holds.** In this case, \( S_t \) contains no element with lifespan between \( x'_0 \) and \( x'_1 \). Because there is also no element with lifespan less than the right index from time \( t' \) to \( t \), then \( S_t \) has no element with lifespan less than \( x_1 \) at time \( t \). Therefore, \( A_1^{(t)} \) processed all of the elements in \( S_t \). By the property of SIEVESTREAMING, we still have

\[ g_i(x_1) = g_i(1) \geq \frac{1}{2} \omega \text{OPT}_i. \]

- **If C2 holds.** In this case, there exists some time \( t'' \leq t' \) s.t. \( g_i(x'_0) \geq (1 - \epsilon)g_i(x'_0) \) holds (cf. Fig. 10), and from time \( t'' \) to \( t' \), no element with lifespan between the two indices arrived (however \( S_t \) may have elements with lifespan less than \( x_1 \) at time \( t \) and these elements arrived before time \( t'' \)). Notice that elements with lifespan no larger than the left index all expired after time \( t' \) and they do not affect the solution at time \( t \); therefore, we can safely ignore these elements in our analysis and only care elements with lifespan less than the right index arrived in interval \( [t'', t] \). Notice that these elements are only inserted on the right side of the right index.

In other words, at time \( t'' \), the output values of the two instances satisfy \( g_i(x'_0) \geq (1 - \epsilon)g_i(x'_0) \); from time \( t'' \) to \( t \), the two instances are fed with same elements. Such a scenario has been studied in the sliding-window case (Epasto et al. 2017). By submodularity of \( f \) and suffix-monotonicity\(^4\) of the SIEVESTREAMING algorithm, the following lemma guarantees that \( g_i(x_1) \) is close to \( \text{OPT}_i \).

**Lemma 3.** Consider a cardinality constrained monotone submodular function maximization problem. Let \( A(S) \) denote the output value of applying the SIEVESTREAMING algorithm on stream \( S \). Let \( S \parallel S' \) denote the concatenation of two streams \( S \) and \( S' \). If \( A(S_2) \geq (1 - \epsilon)A(S_1) \) for \( S_2 \subseteq S_1 \) (i.e., each element in stream \( S_2 \) is also an element in stream \( S_1 \)), then \( A(S_2 || S) \geq (1/3 - \epsilon)\text{OPT} \) for all \( S \), where \( \text{OPT} \) is the value of an optimal solution in stream \( S_1 || S \).

\(^4\)That is, feeding more elements to a SIEVESTREAMING algorithm cannot decrease its output value.
In our scenario, at time $t''$ the two instances satisfy $g_{i''}(x_{i''}) \ge (1 - \epsilon)g_i(x_i)$ and $A_{x_i}$’s input elements is a subset of $A_{x_{i''}}$’s input elements. After time $t''$, the two instances are fed with same elements. Hence, $g_i(x_i) \ge (1/3 - \epsilon)\text{OPT}_i$.

Combining above results, we conclude that HISTAPPROX guarantees a $(1/3 - \epsilon)$ approximation factor.

\[ \square \]

**Proof of Theorem 4**

\textbf{Proof.} At any time $t$, because $g_t(x_{i+2}) < (1 - \epsilon)g_t(x_i)$, and $g_t(l) \in [\Delta, k\Delta]$ where $\Delta \triangleq \max\{f(v) : v \in V\}$, then the size of index set $x_i$ is upper bounded by $O(\log(k - 1)) = O(\epsilon^{-1} \log k)$. For each data element, in the worst case, we need to update $|x_i|$ SIEVESTREAMING instances, and each SIEVESTREAMING instance has update time $O(\epsilon^{-1} \log k)$. In addition, procedure ReduceRedundancy has complexity is $O(\epsilon^{-2} \log^2 k)$. Thus the total time complexity is $O(\epsilon^{-2} \log^2 k)$.

For memory usage, because HISTAPPROX maintains $|x_i|$ SIEVESTREAMING instances, and each instance uses memory $O(kc^{-1} \log k)$), Thus the total memory used by HISTAPPROX is $O(kc^{-2} \log^2 k))$. \[ \square \]

**Proof of Lemma 2**

We state a more general conclusion than Lemma 2. This conclusion will be useful in the later discussion.

\textbf{Lemma 4.} Let $S_1 \parallel S_2 \parallel S_3$ denote the concatenation of three streams $S_1, S_2, S_3$. Let $A$ denote an insertion-only SSO algorithm with approximation factor $c$. Let $A(S)$ denote the output value of applying algorithm $A$ on stream $S$. Assume $A$ is suffix-monotone, i.e., $A(S \parallel S') \ge A(S)$, $\forall S'$. Let OPT denote the value of an optimal solution in stream $S_1 \parallel S_2 \parallel S_3$. Then we have the following results:

1. If $A(S_2) \ge (1 - \epsilon)A(S_1 \parallel S_2)$, then $A(S_2 \parallel S_3) \ge \frac{c}{2}(1 - \epsilon)\text{OPT}$.\footnote{Adapted from Lemma 1 in (Chen, Nguyen, and Zhang 2016).}

2. If $A(S_1) \ge (1 - \epsilon)A(S_1 \parallel S_2)$, then $A(S_1 \parallel S_3) \ge \frac{c}{2}(1 - \epsilon)\text{OPT}$.

\textbf{Proof.} Let $O_{123}, O_{12}, O_{13}, O_{23},$ and $O_3$ denote the optimal solutions in streams $S_1 \parallel S_2 \parallel S_3, S_1 \parallel S_2, S_1 \parallel S_3, S_2 \parallel S_3,$ and $S_3$, respectively.

To prove (1), by the property of algorithm $A$, we have

\[
A(S_2 \parallel S_3) \ge c f(O_{123}).
\]

We also have

\[
A(S_2 \parallel S_3) \ge A(S_2) \\
\ge (1 - \epsilon)A(S_1 \parallel S_2) \\
\ge c(1 - \epsilon)f(O_{12}) + c f(O_{23})
\]

Combining above two relations, we have

\[
2A(S_2 \parallel S_3) \ge c(1 - \epsilon)f(O_{12}) + c f(O_{23}) \\
\ge c(1 - \epsilon)[f(O_{12}) + f(O_{23})] \\
\ge c(1 - \epsilon)[f(O_{12}) + f(O_{3})] \\
\ge c(1 - \epsilon)[f(O_{123} \cap O_{12}) + f(O_{123} \cap O_3)] \\
\ge c(1 - \epsilon)f(O_{123}) \\
= c(1 - \epsilon)\text{OPT}
\]

where the last inequality holds due to the submodularity of $f$.\footnote{For two sets $A, B \subseteq V$ and a submodular function $f : 2^V \rightarrow \mathbb{R}$, it holds that $f(A) + f(B) \ge f(A \cup B) + f(A \cap B)$.} We hence obtain

\[
A(S_2 \parallel S_3) \ge \frac{c}{2}(1 - \epsilon)f(O_{123}).
\]

To prove (2), similarly, by the property of algorithm $A$, we have

\[
A(S_1 \parallel S_3) \ge c f(O_{13}).
\]

We also have

\[
A(S_1 \parallel S_3) \ge A(S_1) \\
\ge (1 - \epsilon)A(S_1 \parallel S_2) \\
\ge c(1 - \epsilon)f(O_{12})
\]
Combining above two relations, we have
\[
2 \mathcal{A}(S_1\|S_3) \geq c(1-\epsilon)f(O_{12}) + cf(O_{13}) \\
\geq c(1-\epsilon)[f(O_{12}) + f(O_{13})] \\
\geq c(1-\epsilon)[f(O_{12}) + f(O_3)] \\
\geq c(1-\epsilon)[f(O_{123} \cap O_{12}) + f(O_{123} \cap O_3)] \\
\geq c(1-\epsilon)f(O_{123}) \\
= c(1-\epsilon)\text{OPT}.
\]

We hence obtain
\[
\mathcal{A}(S_1\|S_3) \geq \frac{c}{2}(1-\epsilon)f(O_{123}).
\]

If \(\mathcal{A}\) is the SIEVE\textsc{Streaming} algorithm, then \(c = 1/2 - \epsilon\). In (2), replace the condition \(\mathcal{A}(S_1) \geq (1-\epsilon)\mathcal{A}(S_1\|S_2)\) by \(\mathcal{A}(S_1) \geq \alpha\mathcal{A}(S_1\|S_2)\), then we immediately obtain Lemma 2.

Conclusion (1) of Lemma 4 is a generalization of Lemma 3. Lemma 3 is specific to SIEVE\textsc{Streaming} and the proof of Lemma 3 leverages a specific property of the SIEVE\textsc{Streaming} algorithm. Hence, a better bound is obtained.

Note that conclusions (1) and (2) in Lemma 4 can also be unified.

**Lemma 5.** Let \(S\|S'\) denote the concatenation of two streams \(S\) and \(S'\). Let \(\mathcal{A}\) denote an insertion-only SSO algorithm with approximation factor \(c\). Let \(\mathcal{A}(S)\) denote the output value of applying algorithm \(\mathcal{A}\) on stream \(S\). Assume \(\mathcal{A}\) is suffix-monotone, i.e., \(\mathcal{A}(S\|S') \geq \mathcal{A}(S), \forall S'\). If two streams \(S_1, S_2\) have relation \(S_2 \subseteq S_1\), i.e., each element in stream \(S_2\) is also an element in stream \(S_1\), and \(\mathcal{A}(S_2) \geq (1-\epsilon)\mathcal{A}(S_1)\), then \(\mathcal{A}(S_2\|S) \geq \frac{c}{2}(1-\epsilon)\text{OPT}\) for all stream \(S\), where \(\text{OPT}\) is the value of an optimal solution in stream \(S_1\|S\).

**A Note On Hist\textsc{Approx} and Hist\textsc{Streaming}**

Hist\textsc{Approx} maintains a histogram satisfying Lemma 1, which ensures two consecutive indices \(x_0\) and \(x_1\) to satisfy either Case 1 or Case 2.

Case 1 is trivial. It simply states that there is no data with lifespan between \(x_0\) and \(x_1\). This property is used in the proof to show that \(\mathcal{A}_x^{(i)}\) processes all the elements in \(S_t\). Hence, \(g_t(x_1) \geq (1/2 - \epsilon)\text{OPT}_t\).

Case 2 states that, \(g_t(x_1') \geq (1-\epsilon)g_t(x_0')\) at some time \(t'\), and from \(t'\) to \(t\), \(\mathcal{A}_{x_0}\) and \(\mathcal{A}_{x_1}\) are fed with same elements. Lemma 3 then guarantees that their output values will remain close with each other after time \(t'\), and \(g_t(x_1) \geq (1/3 - \epsilon)\text{OPT}_t \geq (1/3 - \epsilon)g_t(x_0)\).

As we discussed in the proof of Lemma 4, conclusion (1) of Lemma 4 generalizes Lemma 3. Hence, if \(\mathcal{A}_{x_0}\) and \(\mathcal{A}_{x_1}\) in Hist\textsc{Approx} could find solutions with constant approximation factors, then in Case 2, their output values will be still close with each other, and \(g_t(x_1)\) has a constant approximation factor to \(\text{OPT}_t\).

Hist\textsc{Streaming} algorithm essentially leverages above observation. Hist\textsc{Streaming} ignores the insignificant historical data and conclusion (2) of Lemma 4 guarantees that each instance in Hist\textsc{Streaming} still has a constant approximation factor. Then when we were in Case 2, conclusion (1) of Lemma 4 will guarantee that \(g_t(x_1)\) has a constant approximation factor to \(\text{OPT}_t\).