Submodular Optimization over Streams with Inhomogeneous Decays

Junzhou Zhao,[1] Shuo Shang,[2] Pinghui Wang,[3] John C.S. Lui,[4] Xiangliang Zhang[1]*
[1]King Abdullah University of Science and Technology, KSA
[2]Inception Institute of Artificial Intelligence, UAE
[3]Xi'an Jiaotong University, China
[4]The 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: BASIC-STREAMING is a basic streaming algorithm that achieves an $(1/2-\epsilon)$ approximation factor; HISTAPPROX improves the efficiency significantly and achieves an $(1/3-\epsilon)$ approximation factor; HISTSTREAMING is a streaming version of HISTAPPROX and uses heuristics to further improve the efficiency. Experiments conducted on real data demonstrate that HISTSTREAMING can find high quality solutions and is up to two orders of magnitude faster than the naive GREEDY algorithm.

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).

If data is given in advance, the 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 (Internet Live Stats 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).

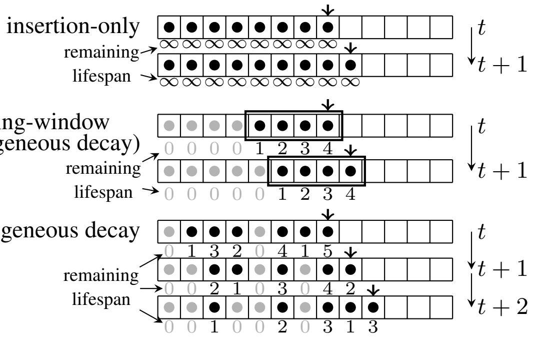


Figure 1: **Insertion-only stream**: each element has an infinite lifespan. **Sliding-window stream**: each element has a same initial lifespan. **Our model**: each element can have an arbitrary lifespan.

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

*Shuo Shang and Xiangliang Zhang are the corresponding authors.

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 an infinite 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 (Y Combinator 2018) where news submitted by users form a news stream. Interesting news may attract users to keep clicking and commenting and thus survive for a long time; while boring news may only survive for one or two days (Leskovec, Backstrom, and Kleinberg 2009). In news recommendation tasks, we should select a subset of news from *current alive news* rather than the most recent news.

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.

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 $l(v,t) \triangleq l(v,t_v) - (t - t_v)$. 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 $\mathcal{S}_t \triangleq \{v \colon v \in V \wedge t_v \leq t \wedge l(v,t) > 0\}$.

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 $f \colon 2^V \mapsto \mathbb{R}_{\geq 0}$ is submodular if $f(S \cup \{s\}) - f(S) \geq f(T \cup \{s\}) - f(T)$, for all $S \subseteq T \subseteq V$ and $s \in V \backslash T$. 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 $\delta(s|S) \triangleq f(S \cup \{s\}) - f(S)$ 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:

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

where k is a given budget.

Remark. The SSO-ID problem is NP-hard, and active data $\mathcal{S}_t$ 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 $\mathcal{S}_t$ is updated, we re-run GREEDY on $\mathcal{S}_t$ from scratch, and this approach outputs a solution that is $(1 - 1/e)$-approximate. However, it needs $O(k|\mathcal{S}_t|)$ utility function evaluations at each time step, which is unaffordable for large $\mathcal{S}_t$. Our goal is to find faster algorithms with comparable approximation guarantees.

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.

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 $\log_{1+\epsilon} 2k = O(\epsilon^{-1}\log k)$ thresholds and each is associated with a candidate set initialized empty. For each coming 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., $V_t = \cup_{l=1}^{L} V_l^{(t)}$ where $V_l^{(t)}$ is the subset of elements with lifespan l at time t. BASICSTREAMING maintains L SIEVESTREAMING instances, denoted by $\{\mathcal{A}_l^{(t)}\}_{l=1}^{L}$, 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 $\mathcal{A}_l^{(t)}$ only process elements with lifespan no less than l. In other words, elements in $\cup_{i\geq l} V_i^{(t)}$ are fed to $\mathcal{A}_l^{(t)}$. After processing V_t, $\mathcal{A}_1^{(t)}$ outputs the current solution.

• **Time Update.** This step prepares for processing the upcoming data in the next time step. We reset instance $\mathcal{A}_1^{(t)}$, i.e., empty its threshold set and each candidate set. Then we conduct a *circular shift* operation: $\mathcal{A}_1^{(t+1)} \leftarrow \mathcal{A}_2^{(t)}, \mathcal{A}_2^{(t+1)} \leftarrow \mathcal{A}_3^{(t)}, \ldots, \mathcal{A}_L^{(t+1)} \leftarrow \mathcal{A}_1^{(t)}$.

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, $\mathcal{A}_1^{(t)}$ ***always processed all the data in*** $\mathcal{S}_t$. Because $\mathcal{A}_1^{(t)}$ is a SIEVESTREAMING instance, we immediately have the following conclusions.

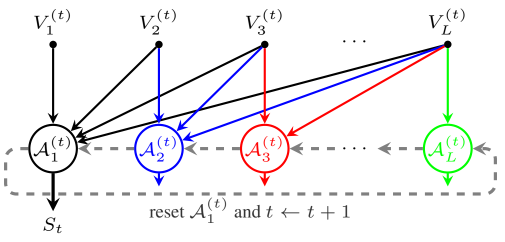


Figure 2: BASICSTREAMING. Solid lines denote data update, and dashed lines denote time update.

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 $\{\mathcal{A}_l^{(1)}\}_{l=1}^{L}$;
2 for $t = 1, 2, \ldots$ do
3     for $l = 1, \ldots, L$ do Feed $\mathcal{A}_l^{(t)}$ with data $\cup_{i\geq l} V_i^{(t)}$;
4     $S_t \leftarrow$ output of $\mathcal{A}_1^{(t)}$;
5     for $l = 2, \ldots, L$ do $\mathcal{A}_{l-1}^{(t+1)} \leftarrow \mathcal{A}_l^{(t)}$;
6     Reset $\mathcal{A}_1^{(t)}$ and $\mathcal{A}_L^{(t+1)} \leftarrow \mathcal{A}_1^{(t)}$;
```

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

Theorem 2. BASICSTREAMING *uses* $O(L\epsilon^{-1}\log k)$ *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 $\{\mathcal{A}_i^{(t)}\}_{i\leq l}$. 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.

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 $\mathcal{S}_t$ to be stored in RAM[1]. Strictly speaking, HISTAPPROX is not a streaming algorithm. We later remove the assumption of storing $\mathcal{S}_t$ 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]For example, if lifespan follows a geometric distribution, i.e., $P(l_v = l) = (1-p)p^{l-1}, l = 1, 2, \ldots$, and at most M elements arrive at a time, then $|\mathcal{S}_t| \leq \sum_{a=0}^{t-1} Mp^a \leq \frac{M}{1-p}$. Hence, if RAM is larger than $\frac{M}{1-p}$, $\mathcal{S}_t$ 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 $\mathcal{A}_l^{(t)}$ 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.

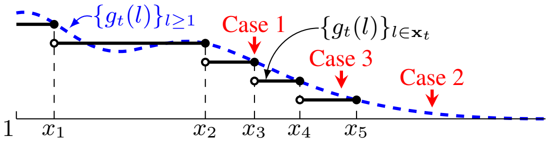


Figure 3: Approximate $\{g_t(l)\}_{l\geq 1}$ by $\{g_t(l)\}_{l\in \mathbf{x}_t}$.

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 $\mathbf{x}_t \triangleq \{x_1^{(t)}, x_2^{(t)}, \ldots\}$ index a set of active instances at time t, where each index $x_i^{(t)} \geq 1$.[2] In the follows, we describe the $\mathbf{x}_t$ updating method, i.e., HISTAPPROX, and the method guarantees that the maintained histogram satisfies our requirement.

Algorithm Description. HISTAPPROX consists of two steps: (1) updating indices; (2) removing redundant indices.

• **Updating Indices.** The algorithm starts with an empty index set, i.e., $\mathbf{x}_1 = \emptyset$. At time t, consider a set of newly arrived elements $V_l^{(t)}$ with lifespan l. These elements will increase the curve before l (because data $V_l^{(t)}$ will be fed to $\{\mathcal{A}_i^{(t)}\}_{i\leq l}$, see Fig. 2). There are three cases based on the position of l, as illustrated in Fig. 3.

Case 1. If $l \in \mathbf{x}_t$, we simply feed $V_l^{(t)}$ to $\{\mathcal{A}_i^{(t)}\}_{i\in\mathbf{x}_t \wedge i\leq l}$.

Case 2. If $l \notin \mathbf{x}_t$ and l has no successor in $\mathbf{x}_t$, we create a new instance $\mathcal{A}_l^{(t)}$ and feed $V_l^{(t)}$ to $\{\mathcal{A}_i^{(t)}\}_{i\in\mathbf{x}_t \wedge i\leq l}$.

Case 3. If $l \notin \mathbf{x}_t$ and l has a successor $l_2 \in \mathbf{x}_t$. Let $\mathcal{A}_l^{(t)}$ be a copy of $\mathcal{A}_{l_2}^{(t)}$, then we feed $V_l^{(t)}$ to $\{\mathcal{A}_i^{(t)}\}_{i\in\mathbf{x}_t \wedge i\leq l}$. Note that $\mathcal{A}_l^{(t)}$ needs to process all data with lifespan $\geq l$ at time t. Because $\mathcal{A}_{l_2}^{(t)}$ has processed all data with lifespan $\geq l_2$, we still need to feed $\mathcal{A}_l^{(t)}$ with historical data s.t. their lifespan $\in [l, l_2)$. That is the reason we need $\mathcal{S}_t$ to be stored in RAM.

Above scheme guarantees that each $\mathcal{A}_l^{(t)}, l \in \mathbf{x}_t$ 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 (ϵ-redundancy)**.** *At time t, consider two instances $\mathcal{A}_i^{(t)}$ and $\mathcal{A}_l^{(t)}$ with $i < l$. We say $\mathcal{A}_l^{(t)}$ is ϵ-redundant if their exists $j > l$ such that $g_t(j) \geq (1-\epsilon)g_t(i)$.*

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

[2]Superscript t will be omitted if time t is clear from context.

Algorithm 2: HISTAPPROX

```
Input: An IDS of data elements arriving over time
Output: A subset $S_t$ at any time $t$
$\mathbf{x}_1 \leftarrow \emptyset$;
for $t = 1, 2, \ldots$ do                                  // $V_t = \cup_l V_l^{(t)}$
    foreach $V_l^{(t)} \neq \emptyset$ do Process($V_l^{(t)}$);     // data update
    $S_t \leftarrow$ output of $\mathcal{A}_{x_1}^{(t)}$;
    if $x_1 = 1$ then Kill $\mathcal{A}_1^{(t)}$, $\mathbf{x}_t \leftarrow \mathbf{x}_t \backslash \{1\}$;      // time update
    for $i = 1, \ldots, |\mathbf{x}_t|$ do
        $\mathcal{A}_{x_i-1}^{(t+1)} \leftarrow \mathcal{A}_{x_i}^{(t)}$, $x_i^{(t+1)} \leftarrow x_i^{(t)} - 1$;
Procedure Process($V_l^{(t)}$)
    if $l \notin \mathbf{x}_t$ then
        if $l$ has no successor in $\mathbf{x}_t$ then      // Case 2 in Fig. 3
            $\mathcal{A}_l^{(t)} \leftarrow$ new instance;
        else // let $l_2$ denote the successor of $l$
            $\mathcal{A}_l^{(t)} \leftarrow$ a copy of $\mathcal{A}_{l_2}^{(t)}$;        // Case 3 in Fig. 3
            Feed $\mathcal{A}_l^{(t)}$ with historical data elements s.t. their
              lifespans $\in [l, l_2)$;
        $\mathbf{x}_t \leftarrow \mathbf{x}_t \cup \{l\}$;
    foreach $i \in \mathbf{x}_t$ and $i \leq l$ do Feed $\mathcal{A}_i^{(t)}$ with $V_l^{(t)}$;
    ReduceRedundancy();
Procedure ReduceRedundancy()
    foreach $i \in \mathbf{x}_t$ do
        Find the largest $j > i$ in $\mathbf{x}_t$ s.t. $g_t(j) \geq (1-\epsilon)g_t(i)$;
        Delete each index $l \in \mathbf{x}_t$ s.t. $i < l < j$ and kill $\mathcal{A}_l^{(t)}$;
```

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

Lemma 1. *For two consecutive indices $x_i, x_{i+1} \in \mathbf{x}_t$ at any time t, one of the following two cases holds:*

C1 *$\mathcal{S}_t$ contains no data with lifespan $\in (x_i, x_{i+1})$.*

C2 *$g_{t'}(x'_{i+1}) \geq (1-\epsilon)g_{t'}(x'_i)$ at 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 $g_t(x_1)$.

Theorem 3. HISTAPPROX *is $(1/3-\epsilon)$-approximate, i.e., at any time t, $g_t(x_1) \geq (1/3-\epsilon)\mathrm{OPT}_t$.*

Theorem 4. 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 $|\mathcal{S}_t|$ memory to store $\mathcal{S}_t$.*

Remark. Because we use a histogram to approximate a curve, HISTAPPROX has a weaker approximation guarantee than BASICSTREAMING. In experiments, we observe that

HISTAPPROX finds solutions with quality very close to BASICSTREAMING and is much faster. The main drawback of HISTAPPROX is that active data $\mathcal{S}_t$ needs to be stored in RAM to ensure each $\mathcal{A}_l^{(t)}$'s output is accurate. If $\mathcal{S}_t$ is larger than RAM capacity, then HISTAPPROX is inapplicable. We address this limitation in the following section.

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 $\mathcal{S}_t$ 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 $\mathcal{S}_t$. What if $\mathcal{A}_l^{(t)}$ does not process historical data? Because $\mathcal{A}_l^{(t)}$ does not process all the data with lifespan $\geq l$ in $\mathcal{S}_t$, there will be a bias between its actual output $\hat{g}_t(l)$ and expected output $g_t(l)$. We only need to worry about the case $\hat{g}_t(l) < g_t(l)$, as the other case $\hat{g}_t(l) \geq g_t(l)$ means that without processing historical data, $\mathcal{A}_l^{(t)}$ 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 $\mathcal{A}_l^{(t)}$ essentially needs to process three substreams: (1) elements arrived before t with lifespan $\leq l_2$ (Line 13)[3]; (2) unprocessed historical elements with lifespan $\in [l, l_2)$ (Line 14); (3) newly arrived elements V_l (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 $\mathcal{A}(S)$ denote the output value of applying* SIEVESTREAMING *algorithm $\mathcal{A}$ on stream S. If $\mathcal{A}(S_1) \geq \alpha\mathcal{A}(S_1 \| S_2)$ for $0 < \alpha < 1$, then $\mathcal{A}(S_1 \| S_3) \geq (1/4 - \epsilon)\alpha\mathrm{OPT}$ where OPT is the value of an optimal solution in stream $S_1 \| S_2 \| S_3$.*

Lemma 2 states that, if historical data S_2 is insignificant, i.e., $\mathcal{A}(S_1) \geq \alpha\mathcal{A}(S_1 \| S_2)$ for $0 < \alpha < 1$ (the closer α is to 1, the less significant S_2 is), then an instance does not need to process S_2 and still finds quality guaranteed solutions. This will further ensure that HISTAPPROX finds quality guaranteed solutions (more explanation on this point can be found in the extended version of this paper). Although it is intractable to theoretically show that historical data S_2 is indeed insignificant, intuitively, as unprocessed historical data S_2 is caused by the deletion of redundant instances (consider the example given in Fig. 4). These instances are redundant because $\mathcal{A}(S_1 \| S_2)$ does not increase much upon $\mathcal{A}(S_1)$. Hence, it makes sense to assume that historical data S_2 is insignificant, and by Lemma 2, S_2 can be ignored.

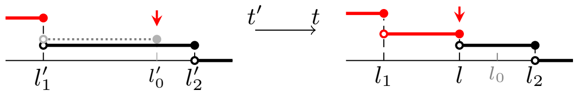


Figure 4: At time t', data with lifespan l'_0 arrives and forms a redundant instance, which is removed. At time $t > t'$, data with lifespan l arrives and $\mathcal{A}_l^{(t)}$ is created. Data at l_0 becomes the unprocessed historical data. We thus say that unprocessed historical data is caused by the deletion of redundant instance at previous time.

Protecting non-redundant instances. To further ensure the solution quality of HISTSTREAMING, we introduce another heuristic to protect non-redundant instances.

Because $g_t(l)$ is unknown, to avoid removing instances that are actually not redundant, we give each instance $\mathcal{A}_l^{(t)}$ an amount of value, denoted by δ_l, as compensation for not processing historical data, i.e., $g_t(l)$ may be as large as $\hat{g}_t(l) + \delta_l$. This allows us to represent $g_t(l)$ by an interval $[\underline{g}_t(l), \overline{g}_t(l)]$ where $\underline{g}_t(l) \triangleq \hat{g}_t(l)$ and $\overline{g}_t(l) \triangleq \hat{g}_t(l) + \delta_l$. As $\underline{g}_t(j) \geq (1-\epsilon)\overline{g}_t(i)$ implies $g_t(j) \geq (1-\epsilon)g_t(i)$, the condition in Line 20 of HISTAPPROX is replaced by $\underline{g}_t(j) \geq (1-\epsilon)\overline{g}_t(i)$.

We want δ_l to be related to the amount of historical data that $\mathcal{A}_l^{(t)}$ does not process. Recall the example in Fig. 4. Unprocessed historical data is always fed to l's predecessor instance whenever redundant instances are removed in the interval l belonging to. Also notice that $g_{t'}(l'_2) \geq (1-\epsilon)g_{t'}(l'_1)$ holds after the removal of redundant instances. Hence, the contribution of unprocessed historical data can be estimated to be at most $\epsilon g_{t'}(l'_1)$. In general, if some redundant indices are removed in interval (i, j) at time t, we set $\delta_l = \epsilon\overline{g}_t(i)$ for index l that is later created in the interval (i, j).

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

Remark. HISTSTREAMING uses heuristics to further improve the efficiency of HISTAPPROX, and no longer needs to store $\mathcal{S}_t$ in memory. In experiments, we observe that HISTSTREAMING can find high quality solutions.

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.

Datasets

DBLP. We construct a *representative author selection* problem on the DBLP dataset (DBLP 2018), which records the

[3]This substream is actually processed by $\mathcal{A}_l^{(t)}$'s successor $\mathcal{A}_{l_2}^{(t)}$, and note that $\mathcal{A}_l^{(t)}$ is copied from $\mathcal{A}_{l_2}^{(t)}$.

Algorithm 3: HISTSTREAMING

```
1  Procedure Process($V_l^{(t)}$)
2      if $l \notin \mathbf{x}_t$ then
3          $\delta_l \leftarrow 0$;
4          ...
           // If $l$ has a successor $l_2$
5          $\mathcal{A}_l^{(t)} \leftarrow$ a copy of $\mathcal{A}_{l_2}^{(t)}$;
6          Find $i, j \in \mathbf{x}_t$ s.t. $l \in (i, j)$ and $\delta_{ij}$ is recorded, then
             let $\delta_l \leftarrow \delta_{ij}$;
7      ...
8  Procedure ReduceRedundancy()
9      foreach $i \in \mathbf{x}_t$ do
10         Find the largest $j > i$ in $\mathbf{x}_t$ s.t. $\underline{g}_t(j) \geq (1-\epsilon)\overline{g}_t(i)$;
11         Delete each index $l \in \mathbf{x}_t$ s.t. $i < l < j$ and kill $\mathcal{A}_l^{(t)}$;
           // Record the amount of unprocessed data in $(i, j)$
12         $\delta_{ij} \leftarrow \epsilon\overline{g}_t(i)$;
```

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 (Stack Exchange 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 post time and last comment time.

Settings

Benchmarks. We consider the following two methods as benchmarks.

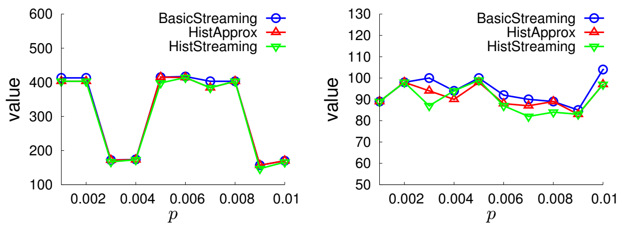
- **GREEDY**. We re-run GREEDY on the active data $\mathcal{S}_t$ at each time t, and apply the *lazy evaluation* trick (Minoux 1978) to further improve its efficiency. GREEDY will serve as an upper bound.
- **RANDOM**. We randomly pick k elements from active data $\mathcal{S}_t$ at each time t. RANDOM 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., $P(l_e = l) = (1-p)^{l-1}p, l = 1, 2, \ldots$. 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.

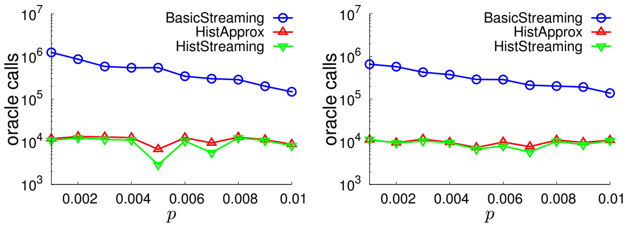
Results

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.



(a) DBLP (b) StackExchange

Figure 5: Solution value comparison (higher is better)



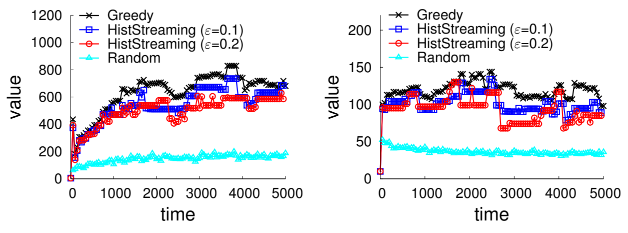
(a) DBLP (b) StackExchange

Figure 6: Oracle calls comparison (lower is better)

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 BASICSTREAMING. In addition, we observe that HISTSTREAMING is slightly faster than HISTAPPROX even though HISTSTREAMING uses smaller RAM.

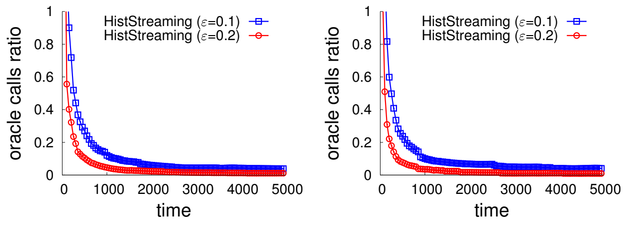
This experiment demonstrates that BASICSTREAMING, HISTAPPROX, and HISTSTREAMING find solutions with similar quality, but HISTAPPROX and HISTSTREAMING are much more efficient than BASICSTREAMING.

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.



(a) DBLP (b) StackExchange

Figure 7: Solution value over time (higher is better)



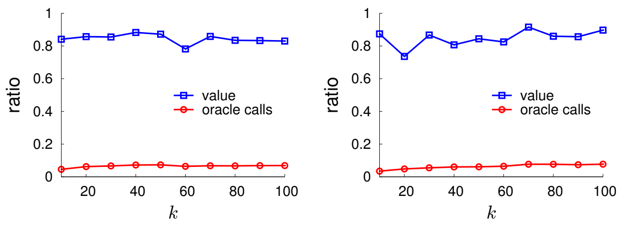
(a) DBLP (b) StackExchange

Figure 8: Oracle calls ratio over time (lower is better)

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 ϵ 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 ϵ 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. ϵ can trade off between solution quality and computational efficiency.

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.



(a) DBLP (b) StackExchange

Figure 9: Ratios under different budget k

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.

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 (Cormode, 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 SIEVESTREAMING algorithm which is the first one round streaming algorithm for insertion-only streams. SIEVESTREAMING 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.

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

Agrawal, R.; Gollapudi, S.; Halverson, A.; and Ieong, S. 2009. Diversifying search results. In *WSDM*, WSDM.

Badanidiyuru, A.; Mirzasoleiman, B.; Karbasi, A.; and Krause, A. 2014. Streaming submodular maximization: Massive data summarization on the fly. In *KDD*.

Braverman, V., and Ostrovsky, R. 2007. Smooth histograms for sliding windows. In *FOCS*.

Brown, G.; Pocock, A.; Zhao, M.-J.; and Luján, M. 2012. Conditional likelihood maximisation: A unifying framework for information theoretic feature selection. *JMLR* 13:27–66.

Chen, J.; Nguyen, H. L.; and Zhang, Q. 2016. Submodular maximization over sliding windows. In *arXiv:1611.00129*.

Cohen, E., and Strauss, M. J. 2006. Maintaining time-decaying stream aggregates. *Journal of Algorithms* 59:19–36.

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

Cormode, G.; Tirthapura, S.; and Xu, B. 2009. Time-decaying sketches for robust aggregation of sensor data. *SIAM Journal on Computing* 39(4):1309–1339.

Datar, M.; Gionis, A.; Indyk, P.; and Motwani, R. 2002. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing* 31(6):1794–1813.

DBLP. 2018. Computer science bibliography. http://dblp.dagstuhl.de/.

Epasto, A.; Lattanzi, S.; Vassilvitskii, S.; and Zadimoghaddam, M. 2017. Submodular optimization over sliding windows. In *WWW*.

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

Internet Live Stats. 2018. Tweets per second. http://www.internetlivestats.com/one-second.

Kempe, D.; Kleinberg, J.; and Tardos, E. 2003. Maximizing the spread of influence through a social network. In *KDD*.

Krause, A., and Golovin, D. 2014. Submodular function maximization. In *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press.

Krause, A.; Singh, A.; and Guestrin, C. 2008. Near-optimal sensor placements in Gaussian processes: Theory, efficient algorithms and empirical studies. *JMLR* 9:235–284.

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

Leskovec, J.; Backstrom, L.; and Kleinberg, J. 2009. Meme-tracking and the dynamics of the news cycle. In *KDD*.

Leskovec, J.; Krause, A.; Guestrin, C.; Faloutsos, C.; VanBriesen, J.; and Glance, N. 2007. Cost-effective outbreak detection in networks. In *KDD*.

Minoux, M. 1978. Accelerated greedy algorithms for maximizing submodular set functions. *Optimization Techniques* 7:234–243.

Mirzasoleiman, B.; Badanidiyuru, A.; Karbasi, A.; Vondrak, J.; and Krause, A. 2015. Lazier than lazy greedy. In *AAAI*.

Mitrovic, M.; Kazemi, E.; Zadimoghaddam, M.; and Karbasi, A. 2018. Data summarization at scale: A two-stage submodular approach. In *ICML*.

Nemhauser, G.; Wolsey, L.; and Fisher, M. 1978. An analysis of approximations for maximizing submodular set functions - I. *Mathematical Programming* 14:265–294.

Stack Exchange. 2018. Stack Exchange data dump. https://archive.org/details/stackexchange.

Y Combintor. 2018. Hacker News. https://news.ycombinator.com/newest.