Streaming Submodular Maximization: Massive Data Summarization on the Fly

Ashwinkumar Badanidiyuru
Cornell Unversity
ashwinkumarbv@gmail.com

Baharan Mirzasoleiman
ETH Zurich
baharanm@inf.ethz.ch

Amin Karbasi
ETH Zurich
Yale University
amin.karbasi@gmail.com

Andreas Krause
ETH Zurich
krausea@ethz.ch

ABSTRACT

How can one summarize a massive data set "on the fly", i.e., without even having seen it in its entirety? In this paper, we address the problem of extracting representative elements from a large stream of data. I.e., we would like to select a subset of say k data points from the stream that are most representative according to some objective function. Many natural notions of "representativeness" satisfy submodularity, an intuitive notion of diminishing returns. Thus, such problems can be reduced to maximizing a submodular set function subject to a cardinality constraint. Classical approaches to submodular maximization require full access to the data set. We develop the first efficient streaming algorithm with constant factor $1/2 - \varepsilon$ approximation guarantee to the optimum solution, requiring only a single pass through the data, and memory independent of data size. In our experiments, we extensively evaluate the effectiveness of our approach on several applications, including training large-scale kernel methods and exemplar-based clustering, on millions of data points. We observe that our streaming method, while achieving practically the same utility value, runs about 100 times faster than previous work.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database applications—*Data mining*

Keywords

Submodular functions; Streaming algorithms

1. INTRODUCTION

The unprecedented growth in modern datasets – coming from different sources and modalities such as images, videos, sensor data, social networks, etc. – demands novel techniques that extract useful information from massive data, while still remaining computationally tractable. One compelling approach that has gained a lot of interest in recent years is *data summarization*: selecting representative subsets of manageable size out of large data sets. Applications range from exemplar-based clustering [8], to document [23, 7] and corpus summarization [33], to recommender systems [10, 9], just to name a few. A systematic way for data summarization, used in all the aforementioned applications, is to turn the problem into selecting a subset of data elements optimizing a utility function that quantifies "representativeness" of the selected set. Often-times, these objective functions satisfy *submodularity*, an intuitive notion of diminishing returns (c.f., [27]), stating that selecting any given element earlier helps more than selecting it later. Thus, many problems in data summarization require maximizing submodular set functions subject to cardinality constraints [14, 18], and big data means we have to solve this problem at scale.

Submodularity is a property of set functions with deep theoretical and practical consequences. The seminal result of Nemhauser et al. [27], that has been of great importance in data mining, is that a simple greedy algorithm produces solutions competitive with the optimal (intractable) solution. This greedy algorithm starts with the empty set, and iteratively locates the element with maximal marginal benefit (increasing the utility the most over the elements picked so far). This greedy algorithm (and other standard algorithms for submodular optimization), however, unfortunately requires *random access* to the data. Hence, while it can easily be applied if the data fits in main memory, it is impractical for data residing on disk, or arriving over time at a fast pace.

In many domains, data volumes are increasing faster than the ability of individual computers to store them in main memory. In some cases, data may be produced so rapidly that it cannot even be stored. Thus, it becomes of crucial importance to process the data in a *streaming* fashion where at any point of time the algorithm has access only to a small fraction of data stored in primary memory. This approach not only avoids the need for vast amounts of random-access memory but also provides predictions in a timely manner based on the data seen so far, facilitating real-time analytics.

In this paper, we provide a simple streaming protocol, called SIEVE-STREAMING, for monotone submodular function maximization, subject to the constraint that at most k points are selected. It requires only a *single pass* over the data, in *arbitrary* order, and provides a *constant factor* $1/2 - \varepsilon$ approximation to the optimum solution, for any $\varepsilon > 0$. At the same time, it only requires $O((k \log k)/\varepsilon)$

KDD'14, August 24–27, 2014, New York, NY, USA.

ACM 978-1-4503-2956-9/14/08 ...$15.00.
http://dx.doi.org/10.1145/2623330.2623637.

memory (i.e., independent of the data set size), and processes data points with $O((\log k)/\varepsilon)$ update time. To the best of our knowledge, it is the first streaming protocol that provides such strong theoretical guarantees if nothing but monotone submodularity is assumed. Our experimental results demonstrate the effectiveness of our approach on several submodular maximization problems. We show that for problems such as exemplar-based clustering and active set selection in nonparametric learning, our approach leads to streaming solutions that provide competitive utility when compared with those obtained via classical methods, at a dramatically reduced fraction of the computational cost (about 1% in both the exemplar based clustering and active set selection applications).

2. BACKGROUND AND RELATED WORK

Over the recent years, submodular optimization has been identified as a powerful tool for numerous data mining and machine learning applications including viral marketing [17], network monitoring [22], news article recommendation [10], nonparametric learning [14, 29], document and corpus summarization [23, 7, 33], network inference [30], and Determinantal Point Processes [13]. A problem of key importance in all these applications is to maximize a monotone submodular function subject to a cardinality constraint (i.e., a bound on the number k of elements that can be selected). See [18] for a survey on submodular maximization.

Classical approaches for cardinality-constrained submodular optimization, such as the celebrated greedy algorithm of Nemhauser et al. [27], or its accelerated variants [24, 22, 3] require random access to the data. Once the size of the dataset increases beyond the memory capacity (typical in many modern datasets) or the data is arriving incrementally over time, neither the greedy algorithm, nor its accelerated versions can be used.

Scaling up: distributed algorithms.

One possible approach to scale up submodular optimization is to distribute data to several machines, and seek parallel computation methods. In particular, Mirzasoleiman et al. [25] in parallel to Kumar et al. [20] devised distributed algorithms for maximizing submodular functions, under some additional assumptions on the objective function: Lipschitz continuity [25] or bounded spread of the non-zero marginal gains [20]. Prior to [25] and [20], specific instances of distributed submodular maximization, that often arise in large-scale graph mining problems, have been studied. In particular, Chierichetti et al. [6] and later Blelloch et al. [5] addressed the MAX-COVER problem and provided a constant approximation to the centralized algorithm. Lattanzi et al. [21] addressed more general graph problems by introducing the idea of *filtering*: reduce the size of the input in a distributed fashion so that the resulting, much smaller, problem instance can be solved on a single machine. Our streaming method Sieve-Streaming employs a similar filtering idea.

Streaming algorithms for submodular maximization.

Another natural approach to scale up submodular optimization, explored in this paper, is to use streaming algorithms. In fact, in applications where data arrives at a pace that does not allow even storing it, this is the only viable option. The first approach, Stream-Greedy, for submodular maximization on data streams is presented by [14]. However, their approach makes strong assumptions about the way the data stream is generated, and unless their assumptions are met, it is fairly easy to construct examples (and we demonstrate one in this paper) where the performance of their algorithm degrades quickly when compared to the optimum solution. Furthermore, the update time (computational cost to process one data point) of their approach is $\Omega(k)$, which is prohibitive for large k. We compare against their approach in this paper.

The work of [20] claims a multi-pass and a single-pass streaming algorithm. The claimed guarantees for the single pass algorithm depend on the maximum increase in the objective any element can offer (Thm. 27, [20]), while the multi-pass algorithm has a memory requirement depending on the data size n (Thm. 28, [20]). Our algorithm Sieve-Streaming lazily tracks the maximum valued element, enabling a single pass streaming algorithm which *does not depend* on the maximum increase in the objective any element can offer. We cannot empirically compare against [20] as the details of both algorithms are omitted.

There is further related work on the *submodular secretary problem* [15, 4]. While also processing elements in a stream, these approaches are different in two important ways: (i) they work in the stronger model where they must either commit to or permanently discard newly arriving elements; (ii) they require *random* arrival of elements, and have a worse approximation ratio (≤ 0.1 vs. $1/2-\epsilon$). If elements arrive in arbitrary order, performance can degrade arbitrarily. Some approaches also require large (i.e., $O(n)$) memory [15].

In this paper, we provide the first streaming algorithm for cardinality-constrained submodular maximization with 1) constant factor approximation guarantees, that 2) makes no assumptions on the data stream, 3) requires only a single pass, 4) only $O(k \log k)$ memory and 5) only $O(\log k)$ update time, assuming 6) nothing but monotone submodularity.

3. STREAMING SUBMODULAR MAX

We consider the problem of selecting subsets out of a large data set of size n, indexed by V (called ground set). Our goal is to maximize a non-negative set function $f : 2^V \rightarrow \mathbb{R}_+$, where, for $S \subseteq V$, $f(S)$ quantifies the utility of set S, capturing, e.g., how well S represents V according to some objective. We will discuss concrete instances of functions f in Section 4. A set function f is naturally associated with a *discrete derivative*, also called the *marginal gain*,

$$\triangle_f (e|S) \doteq f(S \cup \{e\}) - f(S), \tag{1}$$

where $S \subseteq V$ and $e \in V$, which quantifies the increase in utility obtained when adding e to set S. f is called *monotone* iff for all e and S it holds that $\triangle_f(e|S) \geq 0$. Further, f is *submodular* iff for all $A \subseteq B \subseteq V$ and $e \in V \setminus B$ the following diminishing returns condition holds:

$$\triangle_f (e|A) \geq \triangle_f(e|B). \tag{2}$$

That means, adding an element e in context of a set A helps at least as much as adding e in context of a superset B of A. Throughout this paper, we focus on such monotone submodular functions. For now, we adopt the common assumption that f is given in terms of a value oracle (a black box) that computes $f(S)$ for any $S \subseteq V$. In Section 6, we will discuss the setting where $f(S)$ itself depends on the entire data set V, and not just the selected subset S. Submodular functions

contain a large class of functions that naturally arise in data mining and machine learning applications (c.f., [17, 22, 10, 14, 9, 23, 7, 33, 30, 18]).

Cardinality-constrained submodular maximization.

The focus of this paper is on maximizing a monotone submodular function subject to a cardinality constraint, i.e.,

$$\max_{S \subseteq V} f(S) \quad \text{s.t.} \quad |S| \leq k. \tag{3}$$

We will denote by S^* the subset of size at most k that achieves the above maximization, i.e., the optimal solution, with value $\texttt{OPT} = f(S^*)$. Unfortunately, problem (3) is NP-hard, for many classes of submodular functions [11]. However, a seminal result by Nemhauser et al. [27] shows that a simple greedy algorithm is highly effective. It starts with the empty set $S_0 = \emptyset$, and at each iteration i over the whole dataset, it chooses an element $e \in V$ maximizing (1), i.e.,

$$S_i = S_{i-1} \cup \{\arg\max_{e \in V} \triangle_f(e|S_{i-1})\}. \tag{4}$$

Let S^g denote this greedy solution of size at most k. Nemhauser et al. prove that $f(S^g) \geq (1-1/e)\texttt{OPT}$, i.e., the greedy algorithm obtains a $(1-1/e) \approx 0.63$ approximation. For several classes of monotone submodular functions, it is known that $(1-1/e)$ is the best approximation guarantee that one can hope for [26, 11, 19]. Moreover, the greedy algorithm can be accelerated using lazy evaluations [24, 22].

Submodular optimization over data streams.

In many today's data mining and machine learning applications, running the standard greedy algorithm or its variants [24, 22] is computationally prohibitive: either the data set does not fit in main memory on a single computer, precluding random access, or the data itself arrives in a stream (e.g., activity logs, video streams), possibly in a pace that precludes storage. Hence, in such applications, we seek methods that can process quickly arriving data in a timely manner. *Streaming algorithms* with a limited memory available to them (much less than the ground set) and limited processing time per item [12] are practical solutions in such scenarios. They access only a small fraction of data at any point in time and provide approximate solutions.

More formally, in context of streaming submodular maximization, we assume that the ground set $V = \{e_1, \ldots, e_n\}$ is ordered (in some arbitrary manner, w.l.o.g., the natural order $1, 2, \ldots, n$), and any streaming algorithm must process V in the given order. At each iteration t, the algorithm may maintain a memory $M_t \subset V$ of points, and must be ready to output a candidate feasible solution $S_t \subset M_t$ of size at most $|S_t| \leq k$. Whenever a new point arrives from the stream, the algorithm may elect to remember it (i.e., insert it into its memory). However, if the memory exceeds a specified capacity bound, it must discard elements before accepting new ones. The performance of a streaming algorithm is measured by four basic parameters:

- the *number of passes* the algorithm needs to make over the data stream,
- the *memory* required by the algorithm (i.e., $\max_t |M_t|$),
- the *running time* of the algorithm, in particular the number of oracle queries (evaluations of f) made,
- the *approximation ratio*, i.e., $f(S_T)/\texttt{OPT}$ where S_T is the final solution produced by the algorithm[1].

[1]Note that T can be bigger than n, if the algorithm makes multiple passes over the data.

Towards Streaming Submodular Maximization.

The standard greedy algorithm (4) requires access to all elements of the ground set and hence cannot be directly applied in the streaming setting. A naive way to implement it in a streaming fashion, when the data is static and does not increase over time, is to pass k times over the ground set and at each iteration select an element with the maximum marginal gain. This naive approach will provide a $(1-1/e)$ approximation at the price of passing many (i.e, k) times over the data, using $O(k)$ memory, $O(nk)$ function evaluations. Note that, if the data size increases over time (e.g., new elements are added to a log, video is being recorded, etc.) we can no longer implement this naive approach. Moreover, the accelerated versions of the greedy algorithm do not provide any benefit in the streaming setting as the full ground set is not available for random access.

An alternative approach is to keep a memory of the best elements seen so far. For instance, we can start from the empty set $S_0 = \emptyset$. As long as no more than k elements $e_1, e_2, \ldots, e_t$ have arrived (or no more than k elements are read from the ground set), we keep all of them, i.e., $S_t = S_{t-1} \cup \{e_t\}$, for $t \leq k$. Then for each new data point e_t, where $t > k$, we check whether switching it with an element in S_{t-1} will increase the value of the utility function f. If so, we switch it with the one that maximizes the utility. Formally, if there exists an $e \in S_{t-1}$ such that $f(S_{t-1} \cup \{e_t\} \setminus \{e\}) > f(S_{t-1})$, then we swap e and e_t, setting $S_t = S_{t-1} \cup \{e_t\} \setminus \{e\}$. This greedy approach is the essence of Stream-Greedy of [14]. However, unless strong conditions are met, the performance of Stream-Greedy degrades arbitrarily with k (see Appendix).

Very recently, the existence of another streaming algorithm – Greedy-Scaling – was claimed in [20]. As neither the algorithm nor its proof were described in the paper, we were unable to identify its running time in theory and its performance in practice. Based on their claim, if nothing but monotone submodularity is assumed, Greedy-Scaling has to pass over the dataset $O(1/\delta)$ times in order to provide a solution with $\delta/2$ approximation guarantee. Moreover, the required memory also increases with data as $O(kn^{\delta} \log n)$. With a stronger assumption, namely that all (non-zero) marginals are bounded between 1 and Δ, the existence of a one-pass streaming algorithm with approximation guarantee $1/2-\epsilon$, and memory $k/\epsilon \log(n\Delta)$ is claimed. Note that the above requirement on the bounded spread of the nonzero marginal gains is rather strong and does not hold in certain applications, such as the objective in (6) (here, Δ may increase exponentially in k).

In this paper, we devise an algorithm– Sieve-Streaming – that, for any $\varepsilon > 0$, within only one pass over the data stream, using only $O(k \log(k)/\varepsilon)$ memory, running time of at most $O(n \log(k)/\varepsilon)$ produces a $1/2-\varepsilon$ approximate solution to (3). So, while the approximation guarantee is slightly worse compared to the classical greedy algorithm, a single pass suffices, and the running time is dramatically improved. Moreover, ε serves as a tuning parameter for trading accuracy and cost.

4. APPLICATIONS

We now discuss two concrete applications, with their submodular objective functions f, where the size of the datasets or the nature of the problem often requires a streaming solution. We report experimental results in Section 7. Note that

	# passes	approx. guarantee	memory	update time
Standard Greedy [27]	$O(k)$	$(1-1/e)$	$O(k)$	$O(k)$
GREEDY-SCALING [20]	$O(1/\delta)$	$\delta/2$	$kn^{\delta}\log n$	?
STREAM-GREEDY [14]	multiple	$(1/2-\varepsilon)$	$O(k)$	$O(k)$
SIEVE-STREAMING	1	$(1/2-\varepsilon)$	$O(k\log(k)/\varepsilon)$	$O(\log(k)/\varepsilon)$

Table 1: Comparisons between the existing streaming methods in terms of number of passes over the data, required memory, update time per new element and approximation guarantee. For GREEDY-SCALING we report here the performance guarantees claimed in [20]. However, as the details of the streaming algorithm are not presented in [20] we were unable to identify the update time and compare with them in our experiments. For STREAM-GREEDY no upper bound on the number of rounds is provided in [14]. The memory and update times are not explicitly tied to ε.

many more data mining applications have been identified to rely on submodular optimization (e.g., [17, 22, 10, 9, 23, 33, 30]), which can potentially benefit from our work. Providing a comprehensive survey is beyond the scope of this article.

4.1 Exemplar Based Clustering

We start with a classical data mining application. Suppose we wish to select a set of exemplars, that best represent a massive data set. One approach for finding such exemplars is solving the k-medoid problem [16], which aims to minimize the sum of pairwise dissimilarities between exemplars and elements of the dataset. More precisely, let us assume that for the data set V we are given a distance function $d : V \times V \to \mathbb{R}$ such that $d(\cdot,\cdot)$ encodes dissimilarity between elements of the underlying set V. Then, the k-medoid loss function can be defined as follows:

$$L(S) = \frac{1}{|V|}\sum_{e\in V}\min_{v\in S} d(e,v).$$

By introducing an auxiliary element e_0 (e.g., $= \mathbf{0}$, the all zero vector) we can turn L into a monotone submodular function [14]:

$$f(S) = L(\{e_0\}) - L(S\cup\{e_0\}). \tag{5}$$

In words, f measures the decrease in the loss associated with the set S versus the loss associated with just the auxiliary element. It is easy to see that for suitable choice of e_0, maximizing f is equivalent to minimizing L. Hence, the standard greedy algorithm provides a very good solution. But the problem becomes computationally challenging when we have a large data set and we wish to extract a small subset S of exemplars. Our streaming solution SIEVE-STREAMING addresses this challenge.

Note that in contrast to classical clustering algorithms (such as k-means), the submodularity-based approach is very general: It does not require *any* properties of the distance function d, except nonnegativity (i.e., $d(\cdot,\cdot) \geq 0$). In particular, d is not necessarily assumed to be symmetric, nor obey the triangle inequality.

4.2 Large-scale Nonparametric Learning

Besides extracting representative elements for sake of explorative data analysis, data summarization is a powerful technique for speeding up learning algorithms.

As a concrete example, consider kernel machines [31] (such as kernelized SVMs/logistic regression, Gaussian processes, etc.), which are powerful non-parametric learning techniques. The key idea in these approaches is to reduce non-linear problems such as classification, regression, clustering etc. to linear problems – for which good algorithms are available – in a, typically high-dimensional, transformed space. Crucially, the data set $V = \{e_1,\ldots,e_n\}$ is represented in this transformed space only implicitly via a kernel matrix

$$K_{V,V} = \begin{pmatrix} \mathcal{K}_{e_1,e_1} & \ldots & \mathcal{K}_{e_1,e_n} \\ \vdots & & \vdots \\ \mathcal{K}_{e_n,e_1} & \ldots & \mathcal{K}_{e_n,e_n} \end{pmatrix}.$$

Hereby $\mathcal{K}_{e_i,e_j}$ is the similarity of item i and j measured via a symmetric positive definite kernel function. For example, a commonly used kernel function in practice where elements of the ground set V are embedded in a Euclidean space is the squared exponential kernel

$$\mathcal{K}_{e_i,e_j} = \exp(-|e_i - e_j|_2^2/h^2).$$

Many different kernel functions are available for modeling various types of data beyond Euclidean vectors, such as sequences, sets, graphs etc. Unfortunately, when scaling to large data sets, even representing the kernel matrix $K_{V,V}$ requires space quadratic in n. Moreover, solving the learning problems (such as kernelized SVMs, Gaussian processes, etc.) typically has cost $\Omega(n^2)$ (e.g., $O(n^3)$ for Gaussian process regression).

Thus, a common approach to scale kernel methods to large data sets is to perform *active set selection* (c.f., [28, 32]), i.e., select a small, representative subset $S \subseteq V$, and only work with the kernel matrix $K_{S,S}$ restricted to this subset. The key challenge is to select such a representative set S.

One prominent procedure that is often used in practice is the Informative Vector Machine (IVM) [32], which aims to select a set S maximizing the following objective function

$$f(S) = \frac{1}{2}\log\det(\mathbf{I} + \sigma^{-2}\Sigma_{S,S}), \tag{6}$$

where σ is a regularization parameter. Thus, sets maximizing $f(S)$ maximize the log-determinant $\mathbf{I} + \sigma^{-2}\Sigma_{S,S}$, and therefore capture diversity of the selected elements S. It can be shown that f is monotone submodular [32]. Note that similar objective functions arise when performing MAP inference in Determinantal Point Processes, powerful probabilistic models for data summarization [13].

When the size of the ground set is small, standard greedy algorithms (akin to (4)) provide good solutions. Note that the objective function f only depends on the selected elements (i.e., the cost of evaluating f does not depend on the size of V). For massive data sets, however, classical greedy algorithms do not scale, and we must resort to streaming.

In Section 7, we will show how SIEVE-STREAMING can choose near-optimal subsets out of a data set of 45 million vectors (user visits from Yahoo! Front Page) by only accessing a small portion of the dataset. Note that in many

nonparametric learning applications, data naturally arrives over time. For instance, the Yahoo! Front Page is visited by thousands of people every hour. It is then advantageous, or sometimes the only way, to make predictions with kernel methods by choosing the active set on the fly.

5. THE SIEVE-STREAMING ALGORITHM

We now present our main contribution, the SIEVE-STREAMING algorithm for streaming submodular maximization. Our approach builds on three key ideas: 1) a way of simulating the (intractable) optimum algorithm via thresholding, 2) guessing the threshold based on the maximum singleton element, and 3) lazily running the algorithm for different thresholds when the maximum singleton element is updated. As our final algorithm is a careful mixture of these ideas, we showcase each of them by making certain assumptions and then removing each assumption to get the final algorithm.

5.1 Knowing OPT helps

The key reason why the classical greedy algorithm for submodular maximization works, is that at every iteration, an element is identified that reduces the "gap" to the optimal solution by a significant amount. More formally, it can be seen that, if S_i is the set of the first i elements picked by the greedy algorithm (4), then the marginal value $\triangle_f(e_{i+1}|S_i)$ of the next element e_{i+1} added is at least $(\texttt{OPT} - f(S_i))/k$. Thus, our main strategy for developing our streaming algorithm is identify elements with similarly high marginal value. The challenge in the streaming setting is that, when we receive the next element from the stream, we must immediately decide whether it has "sufficient" marginal value. This will require us to compare it to OPT in some way which gives the intuition that knowing OPT should help.

With the above intuition, we could try to pick the first element with marginal value $\texttt{OPT}/k$. This specific attempt does not work for instances that contain a single element with marginal value just above $\texttt{OPT}/k$ towards the end of the stream, and where the rest of elements with marginal value just below $\texttt{OPT}/k$ appear towards the beginning of the stream. Our algorithm would have then rejected these elements with marginal value just below $\texttt{OPT}/k$ and can never get their value. But we can immediately observe that if we had instead lowered our threshold from $\texttt{OPT}/k$ to some $\beta\texttt{OPT}/k$, we could have still gotten these lower valued elements while still making sure that we get the high valued elements when β is reasonably large. Below, we use $\beta = 1/2$.

Our algorithm will be based on the above intuition. Suppose we know OPT up to a constant factor α, i.e., we have a value v such that $\texttt{OPT} \geq v \geq \alpha \cdot \texttt{OPT}$ for some $0 \leq \alpha \leq 1$. The algorithm starts with setting $S = \emptyset$ and then, after observing each element, it adds it to S if the marginal value is at least $(v/2 - f(S))/(k - |S|)$ and we are still below the cardinality constraint. Thus, it "sieves" out elements with large marginal value. The pseudocode is given in algorithm 1

PROPOSITION 5.1. *Assuming input v to algorithm 1 satisfies $\texttt{OPT} \geq v \geq \alpha\ \texttt{OPT}$, the algorithm satisfies the following properties*

- *It outputs a set S such that $|S| \leq k$ and $f(S) \geq \frac{\alpha}{2}\texttt{OPT}$*
- *It does 1 pass over the data set, stores at most k elements and has $O(1)$ update time per element.*

Algorithm 1 SIEVE-STREAMING-KNOW-OPT-VAL

Input: v such that $\texttt{OPT} \geq v \geq \alpha\ \texttt{OPT}$
1: $S = \emptyset$
2: **for** $i = 1$ to n **do**
3: **if** $\triangle_f(e_i \mid S) \geq \frac{v/2 - f(S)}{k - |S|}$ and $|S_v| < k$ **then**
4: $S := S \cup \{e_i\}$
5: **return** S

5.2 Knowing $\max_{e \in V} f(\{e\})$ is enough

Algorithm 1 requires that we know (a good approximation) to the value of the optimal solution OPT. However, obtaining this value is a kind of chicken and egg problem where we have to estimate OPT to get the solution and use the solution to estimate OPT. The crucial insight is that, in order to get a very crude estimate on OPT, it is enough to know the maximum value of any singleton element $m = \max_{e \in V} f(\{e\})$. From submodularity, we have that

$$m \leq \texttt{OPT} \leq k \cdot m.$$

This estimate is not too useful yet; if we apply Proposition 5.1 directly with $v = km$ and $\alpha = 1/k$, we only obtain the guarantee that the solution will obtain a value of $\texttt{OPT}/2k$.

Fortunately, once we get this crude upper bound $k \cdot m$ on OPT, we can immediately refine it. In fact, consider the following set

$$O = \{(1+\epsilon)^i | i \in \mathbb{Z}, m \leq (1+\epsilon)^i \leq k \cdot m\}.$$

At least one of the thresholds $v \in O$ should be a pretty good estimate of OPT, i.e there should exist at least some $v \in O$ such that $(1-\epsilon)\texttt{OPT} \leq v \leq \texttt{OPT}$. That means, we could run Algorithm 1 once for each value $v \in O$, requiring multiple passes over the data. In fact, instead of using multiple passes, a single pass is enough: We simply run several copies of Algorithm 1 in parallel, producing one candidate solution for each threshold $v \in O$. As final output, we return the best solution obtained.

More formally, the algorithm proceeds as follows. It assumes that the value $m = \max_{e \in V} f(\{e\})$ is given at the beginning of the algorithm. The algorithm discretizes the range $[m, k \cdot m]$ to get the set O. Since the algorithm does not know which value among O is a good estimate for OPT, it simulates Algorithm 1 for each of these values v: Formally it starts with a set S_v for each $v \in O$ and after observing each element, it adds to every S_v for which it has a marginal value of at least $(v/2 - f(S_v))/(k - |S_v|)$ and S_v is below the cardinality constraint. Note that $|O| = \mathcal{O}((\log k)/\epsilon)$, i.e., we only need to keep track of $\mathcal{O}((\log k)/\epsilon)$ many sets S_v of size at most k each, bounding the size of the memory $M = \cup_{v \in O} S_v$ by $\mathcal{O}((k \log k)/\epsilon)$. Moreover, the update time is $\mathcal{O}((\log k)/\epsilon)$ per element. The pseudocode is given in Algorithm 2.

PROPOSITION 5.2. *Assuming input m to Algorithm 2 satisfies $m = \max_{e \in V} f(\{e\})$, the algorithm satisfies the following properties*

- *It outputs a set S such that $|S| \leq k$ and $f(S) \geq \left(\frac{1}{2} - \epsilon\right)\texttt{OPT}$*
- *It does 1 pass over the data set, stores at most $O\left(\frac{k \log k}{\epsilon}\right)$ elements and has $O\left(\frac{\log k}{\epsilon}\right)$ update time per element.*

Algorithm 2 SIEVE-STREAMING-KNOW-MAX-VAL

Input: $m = \max_{e \in V} f(\{e\})$
1: $O = \{(1+\epsilon)^i | i \in \mathbb{Z}, m \leq (1+\epsilon)^i \leq k \cdot m\}$
2: For each $v \in O, S_v := \emptyset$
3: **for** $i = 1$ to n **do**
4: **for** $v \in O$ **do**
5: **if** $\triangle_f(e_i \mid S_v) \geq \frac{v/2 - f(S_v)}{k - |S_v|}$ and $|S_v| < k$ **then**
6: $S_v := S_v \cup \{e_i\}$
7: **return** $\operatorname{argmax}_{v \in O_n} f(S_v)$

5.3 Lazy updates: The final algorithm

While Algorithm 2 successfully removed the unrealistic requirement of knowing OPT, obtaining the maximum value m of all singletons still requires one pass over the full data set, resulting in a two-pass algorithm.

It turns out that it is in fact possible to estimate m on the fly, within a *single* pass over the data. We will need two ideas to achieve this. The first natural idea is to maintain an auxiliary variable m which holds the current maximum singleton element after observing each element e_i and lazily instantiate the thresholds $v = (1+\epsilon)^i, m \leq (1+\epsilon)^i \leq k \cdot m$. Unfortunately, this idea alone does not yet work: This is because when we instantiate a threshold v we could potentially have already seen elements with marginal value $v/(2k)$ that we should have taken for the solution corresponding to S_v.

The second idea is to instead instantiate thresholds for an *increased range* $v = (1+\epsilon)^i, m \leq (1+\epsilon)^i \leq 2 \cdot k \cdot m$. It can be seen that when a threshold v is instantiated from this expanded set O, every element with marginal value $v/(2k)$ to S_v will appear on or after v is instantiated.

We now state the algorithm formally. It maintains an auxiliary variable m that holds the current maximum singleton element after observing each element e_i. Whenever m gets updated, the algorithm lazily instantiates the set O_i and deletes all thresholds outside O_i. Then it includes the element e_i into every S_v for $v \in O_i$ if e_i has the marginal value $(v/2 - f(S_v))/(k - |S_v|)$ to S_v. Finally, it outputs the best solution among S_v. We call the resulting algorithm SIEVE-STREAMING, and present its pseudocode in Algorithm 3, as well as an illustration in Figure 1.

Algorithm 3 SIEVE-STREAMING

1: $O = \{(1+\epsilon)^i | i \in \mathbb{Z}\}$
2: For each $v \in O, S_v := \emptyset$ (maintain the sets only for the necessary v's lazily)
3: $m := 0$
4: **for** $i = 1$ to n **do**
5: $m := \max(m, f(\{e_i\}))$
6: $O_i = \{(1+\epsilon)^i | m \leq (1+\epsilon)^i \leq 2 \cdot k \cdot m\}$
7: Delete all S_v such that $v \notin O_i$.
8: **for** $v \in O_i$ **do**
9: **if** $\triangle_f(e_i \mid S_v) \geq \frac{v/2 - f(S_v)}{k - |S_v|}$ and $|S_v| < k$ **then**
10: $S_v := S_v \cup \{e_i\}$
11: **return** $\operatorname{argmax}_{v \in O_n} f(S_v)$

THEOREM 5.3. SIEVE-STREAMING *(Algorithm 3) satisfies the following properties*

- *It outputs a set S such that $|S| \leq k$ and $f(S) \geq \left(\frac{1}{2} - \epsilon\right) OPT$*

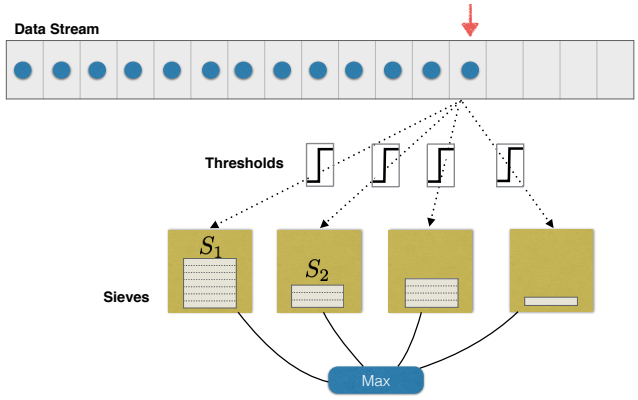


Figure 1: Illustration of SIEVE-STREAMING. Data arrives in any order. The marginal gain of any new data point is computed with respect to all of the sieves. If it exceeds the specific threshold of any sieve that does not yet meet the cardinality constraint, the point will be added. Otherwise it will be discarded. SIEVE-STREAMING ensures that the number of sieves is bounded. Moreover, it can provide statistics about the data accumulated at any time by returning the elements of the sieve with maximum utility.

- *It does 1 pass over the data set, stores at most $O\left(\frac{k \log k}{\epsilon}\right)$ elements and has $O\left(\frac{\log k}{\epsilon}\right)$ update time per element.*

Note that the total computation cost of SIEVE-STREAMING is $O\left(\frac{n \log k}{\epsilon}\right)$. This is in contrast to the cost of $O(nk)$ for the classical greedy algorithm. Thus, not only does SIEVE-STREAMING require only a single pass through the data, it also offers an accuracy–performance tradeoff by providing the tuning parameter ϵ. We empirically evaluate this tradeoff in our experiments in Section 7. Further note that when executing SIEVE-STREAMING, some sets S_v do not "fill up" (i.e., meet the cardinality constraint). The empirical performance can be improved – without sacrificing any guarantees – by maintaining a reservoir (cf., Sec. 6) of $O(k \log k)$ random elements, and augment the non-full sets S_v upon termination by greedily adding elements from this reservoir.

6. BEYOND THE BLACK-BOX

In the previous sections, we have effectively assumed a so called *black-box* model for the function evaluations: given any set S, our algorithm SIEVE-STREAMING can evaluate $f(S)$ independently of the ground set V. I.e., the black-box implementing f only needs access to the selected elements S, but not the full data stream V. In several practical settings, however, this assumption is violated, meaning that the utility function f depends on the entire dataset. For instance, in order to evaluate (5) we need to know the loss function over the entire data set. Fortunately, many such functions (including (5)) share an important characteristic; they are *additively decomposable* [25] over the ground set V. That means, they can be written as

$$f(S) = \frac{1}{|V|} \sum_{e \in V} f_e(S), \tag{7}$$

where $f_e(S)$ is a non-negative submodular function. Decomposability requires that there is a separate monotone submodular function associated with every data point $e \in V$

and the value of $f(S)$ is nothing but the average of $f_e(S)$. For the remaining of this section, we assume that the functions $f_e(\cdot)$ can all be evaluated without accessing the full ground set. We define the evaluation of the utility function f restricted to a subset $W \subseteq V$ as follows:

$$f_W(S) = \frac{1}{|W|} \sum_{e \in W} f_e(S).$$

Hence $f_W(S)$ is the empirical average of f w.r.t. to set W. Note that as long as W is large enough and its elements are randomly chosen, the value of the empirical mean $f_W(S)$ will be a very good approximation of the true mean $f(S)$.

PROPOSITION 6.1. *Assume that all of $f_e(S)$ are bounded and w.l.o.g. $|f_e(S)| \leq 1$. Moreover, let W be uniformly sampled from V. Then by Hoeffding's inequality we have*

$$\Pr(|f_W(S) - f(S)| > \xi) \leq 2 \exp\left(\frac{-|W|\xi^2}{2}\right).$$

There are at most $|V|^k$ sets of size at most k. Hence, in order to have the RHS $\leq \delta$ for *any* set S of size at most k we simply (using the union bound) need to ensure

$$|W| \geq \frac{2\log(2/\delta) + 2k\log(|V|)}{\xi^2}. \quad (8)$$

As long as we know how to sample uniformly at random from a data stream, we can ensure that for decomposable functions defined earlier, our estimate is close (within the error margin of ξ) to the correct value. To sample randomly, we can use a *reservoir sampling* technique [35]. It creates a reservoir array of size $|W|$ and populates it with the first $|W|$ items of V. It then iterates through the remaining elements of the ground set until it is exhausted. At the i-th element where $i > |W|$, the algorithm generates a random number j between 1 and i. If j is at most $|W|$, the j-th element of the reservoir array is replaced with the ith element of V. It can be shown that upon finishing the data stream, each item in V has equal probability of being chosen for the reservoir.

Now we can devise a two round streaming algorithm, which in the first round applies reservoir sampling to sample an evaluation set W uniformly at random. In the second round, it simply runs SIEVE-STREAMING and evaluates the utility function f only with respect to the reservoir W.

Algorithm 4 SIEVE-STREAMING + Reservoir Sampling

1: Go through the data and find a reservoir W of size (8)
2: Run SIEVE-STREAMING by only evaluating $f_W(\cdot)$

THEOREM 6.2. *Suppose* SIEVE-STREAMING *uses a validation set W of size $|W| \geq \frac{2\log(2/\delta)k^2 + 2k^3\log(|V|)}{\varepsilon^2}$. Then with probability $1-\delta$, the output of Alg 4 will be a set S of size at most k such that*

$$f(S) \geq (\frac{1}{2} - \varepsilon)(\texttt{OPT} - \varepsilon).$$

This result shows how SIEVE-STREAMING can be applied to decomposable submodular functions if we can afford enough memory to store a sufficiently large evaluation set. Note that the above approach naturally suggests a heuristic one-pass algorithm for decomposable functions: Take the first $\frac{2\log(2/\delta)k^2 + 2k^3\log(|V|)}{\varepsilon^2}$ samples as validation set and run the greedy algorithm on it to produce a candidate solution. Then process the remaining stream, updating the validation set via reservoir sampling, and applying SIEVE-STREAMING, using the current validation set in order to approximately evaluate f. We report our experimental results for the exemplar-based clustering application using an evaluation set W.

7. EXPERIMENTS

In this section, we address the following questions:

- How good is the solution provided by SIEVE-STREAMING compared to the existing streaming algorithms?
- What computational benefits do we gain when using SIEVE-STREAMING on large datasets?
- How good is the performance of SIEVE-STREAMING on decomposable utility functions?

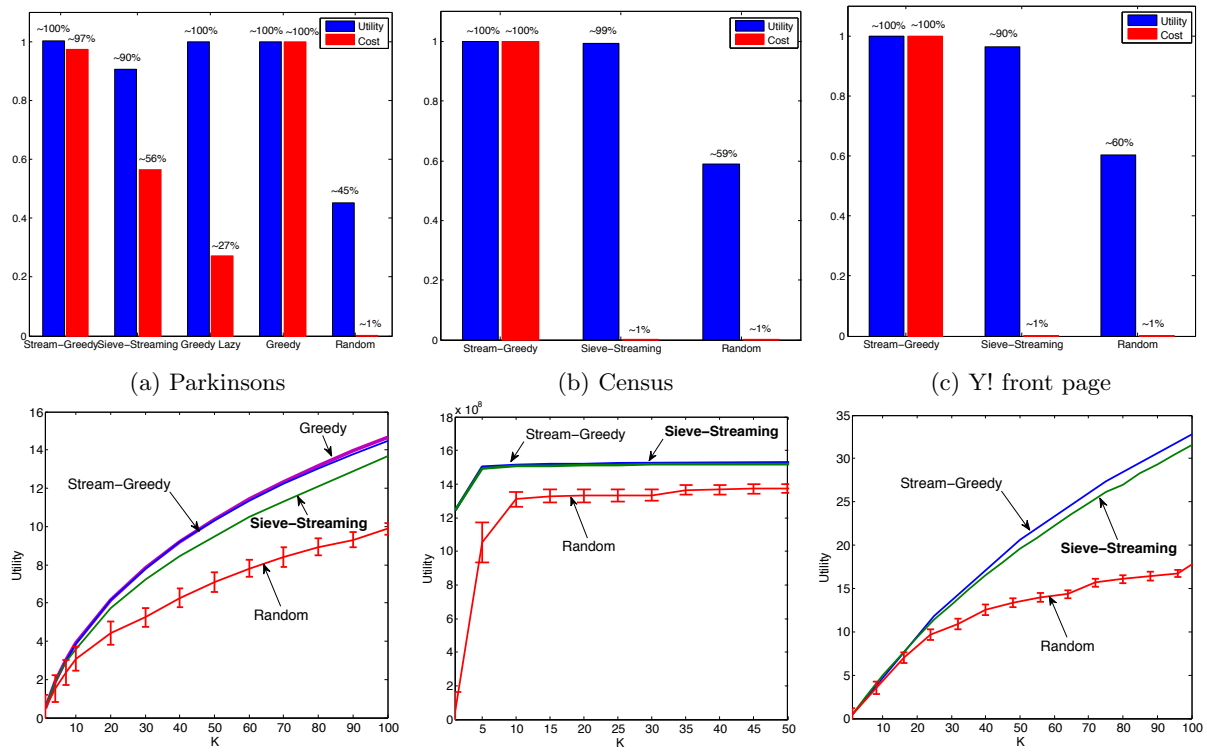
To this end, we run SIEVE-STREAMING on the two data mining applications we described in Section 4: exemplar-based clustering and active set selection for nonparametric learning. For both applications we report experiments on large datasets with millions of data points. Throughout this section we consider the following benchmarks:

- *random selection*: the output is k randomly selected data points from V.
- *standard greedy*: the output is the k data points selected by algorithm (4). This algorithm is not applicable in the streaming setting.
- *lazy greedy*: the output produced by the accelerated greedy method [24]. This algorithm is not applicable in the streaming setting.
- STREAM-GREEDY: The output is the k data points provided by STREAM-GREEDY as described in Section 3

In all of our experiments, we stop the streaming algorithms if the utility function does not improve significantly (relative improvement of at least γ for some small value $\gamma > 0$). In all of our experiment, we chose $\gamma = 10^{-5}$. This way, we can compare the performance of different algorithms in terms of computational efforts in a fair way. Throughout this section, we measure the computational cost in terms of the number of function evaluations used (more precisely, number of oracle queries). The advantage of this measure is that it is independent of the concrete implementation and platform. However, to demonstrate that the results remain almost identical, we also report the actual wall clock time for the exemplar-based clustering application. The random selection policy has the lowest computational cost among the streaming algorithms we consider in this paper. In fact, in terms of function evaluations its cost is one; at the end of the sampling process the selected set is evaluated once. To implement the random selection policy we can employ the reservoir sampling technique discussed earlier. On the other end of the spectrum, we have the standard greedy algorithm which makes k passes over the ground set, providing typically the best solution in terms of utility. Since it is computationally prohibitive we cannot run it for the large-scale datasets. However, we also report results on a smaller data set, where we compare against this algorithm.

7.1 Active Set Selection

For the active set selection objective described in Section 4.2, we chose a Gaussian kernel with $h = 0.75$ and



(a) Parkinsons (b) Census (c) Y! front page

(d) Parkinsons (e) Census (f) Y! front page

Figure 2: Performance comparison. a), b) and c) show the utility obtained, along with the computational cost, by the algorithms on 3 different datasets. SIEVE-STREAMING achieves the major fraction of the utility with orders of magnitude less computation time. d), e) f) demonstrate the performance of all the algorithms for different values of k on the same datasets. In all of them SIEVE-STREAMING performs close to the (much more computationally expensive) STREAM-GREEDY benchmark.

$\sigma = 1$. For the small-scale experiments, we used the *Parkinsons Telemonitoring* dataset [34] consisting of 5,875 biomedical voice measurements with 22 attributes from people with early-stage Parkinson's disease. We normalized the vectors to zero mean and unit variance. Fig. 2a compares the performance of SIEVE-STREAMING to the benchmarks for a fixed active set size $k = 20$. The computational costs of all algorithms, as well as their acquired utilities, are normalized to those of the standard greedy. As we can see, SIEVE-STREAMING provides a solution close to that of the standard greedy algorithm with a much reduced computational cost. For different values of k, Fig. 2d shows the performance of all the benchmarks. Again, SIEVE-STREAMING operates close to STREAM-GREEDY and (lazy) greedy.

For our large-scale scale experiment, we used the *Yahoo! Webscope* data set consisting of 45,811,883 user visits from the Featured Tab of the Today Module on the Yahoo! Front Page [2]. For each visit, the user is associated with a feature vector of dimension six. Fig. 2c compares the performance of SIEVE-STREAMING to the benchmarks for a fixed active set size $k = 100$. Since the size of the data set is large, we cannot run the standard (or lazy) greedy algorithm. As a consequence, computational costs and utilities are normalized to those of the STREAM-GREEDY benchmark. For such a large dataset, the benefit of using SIEVE-STREAMING is much more pronounced. As we see, SIEVE-STREAMING provides a solution close to that of STREAM-GREEDY while having several orders of magnitude lower computational cost. The performance of all algorithms (expect the standard and lazy greedy) for different values of k is shown in Fig. 2f.

7.2 Exemplar-Based Clustering

Our exemplar-based clustering experiment involves SIEVE-STREAMING applied to the clustering utility function (described in Section 4.1) with the squared Euclidean distance, namely, $d(x, x') = \|x - x'\|^2$. We run the streaming algorithms on the *Census1990* dataset [1]. It consists of 2,458,285 data points with 68 attributes. We compare the performance of SIEVE-STREAMING to the benchmarks as well as the classical online k-means algorithm (where we snap each mean to the nearest exemplar in a second pass to obtain a subset $S \subseteq V$). Again, the size of the dataset is too large to run the standard greedy algorithm. As discussed in Section 6, the clustering utility function depends on the whole data set. However, it is decomposable, thus an evaluation set W can be employed to estimate the utility of any set S based on the data seen so far. For our experiments, we used reservoir sampling with $|W| = 1/10|V|$.

Fig 2b shows the performance of SIEVE-STREAMING compared to the benchmarks for a fixed active set size $k = 5$. The computational costs of all algorithms, as well as their acquired utilities, are normalized to those of STREAM-GREEDY. We did not add the performance of online k-means to this figure as online k-means is not based on submodular function maximization. Hence, it does not query the clustering utility function. As we observe again, SIEVE-STREAMING provides a solution close to that of STREAM-GREEDY with substantially lower computational cost. For different values of k, Fig. 2e shows the performance of all the benchmarks. To compare the computational cost of online k-means with the benchmarks that utilize the clustering utility function, we measure the wall clock times of all the methods. This is

reported in Table 2. We see again that the utility of SIEVE-STREAMING is comparable to STREAM-GREEDY and online k-means with much lower wall clock time.

8. CONCLUSIONS

We have developed the first efficient streaming algorithm–SIEVE-STREAMING–for cardinality-constrained submodular maximization. SIEVE-STREAMING provides a constant factor $1/2 - \varepsilon$ approximation guarantee to the optimum solution and requires only a single pass through the data and memory independent of the data size. In contrast to previous work, which makes strong assumptions on the data stream V or on the utility function (e.g., bounded spread of marginal gains, or Lipschitz continuity), we assumed nothing but monotonicity and submodularity. We have also demonstrated the effectiveness of our approach through extensive large scale experiments. As shown in Section 7, SIEVE-STREAMING reaches the major fraction of the utility function with much (often several orders of magnitude) less computational cost. This property of SIEVE-STREAMING makes it an appealing and sometimes the only viable method for solving very large scale or streaming applications. Given the importance of submodular optimization to numerous data mining and machine learning applications, we believe our results provide an important step towards addressing such problems at scale.

Acknowledgments. This research was supported in part by NSF AF-0910940, SNF 200021-137971, DARPA MSEE FA8650-11-1-7156, ERC StG 307036, a Microsoft Faculty Fellowship and an ETH Fellowship.

9. REFERENCES

[1] Census1990, UCI machine learning repository, 2010.

[2] Yahoo! academic relations. r6a, yahoo! front page today module user click log dataset, version 1.0, 2012.

[3] A. Badanidiyuru and J. Vondrák. Fast algorithms for maximizing submodular functions. In *SODA*, 2014.

[4] M. Bateni, M. Hajiaghayi, and M. Zadimoghaddam. Submodular secretary problem and extensions. In *APPROX-RANDOM*, pages 39–52, 2010.

[5] G. E. Blelloch, R. Peng, and K. Tangwongsan. Linear-work greedy parallel approximate set cover and variants. In *SPAA*, 2011.

[6] F. Chierichetti, R. Kumar, and A. Tomkins. Max-cover in map-reduce. In *WWW*, 2010.

[7] A. Dasgupta, R. Kumar, and S. Ravi. Summarization through submodularity and dispersion. In *ACL*, 2013.

[8] D. Dueck and B. J. Frey. Non-metric affinity propagation for unsupervised image categorization. In *ICCV*, 2007.

[9] K. El-Arini and C. Guestrin. Beyond keyword search: Discovering relevant scientific literature. In *KDD*, 2011.

[10] K. El-Arini, G. Veda, D. Shahaf, and C. Guestrin. Turning down the noise in the blogosphere. In *KDD*, 2009.

[11] U. Feige. A threshold of ln n for approximating set cover. *Journal of the ACM*, 1998.

[12] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: a review. *SIGMOD Record*, 34(2):18–26, 2005.

[13] J. Gillenwater, A. Kulesza, and B. Taskar. Near-optimal map inference for determinantal point processes. In *NIPS*, pages 2744–2752, 2012.

[14] R. Gomes and A. Krause. Budgeted nonparametric learning from data streams. In *ICML*, 2010.

[15] A. Gupta, A. Roth, G. Schoenebeck, and K. Talwar. Constrained non-monotone submodular maximization: Offline and secretary algorithms. In *WINE*, 2010.

[16] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*, volume 344. Wiley-Interscience, 2009.

[17] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the ninth ACM SIGKDD*, 2003.

[18] A. Krause and D. Golovin. Submodular function maximization. In *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press, 2013.

[19] A. Krause and C. Guestrin. Near-optimal nonmyopic value of information in graphical models. In *UAI*, 2005.

[20] R. Kumar, B. Moseley, S. Vassilvitskii, and A. Vattani. Fast greedy algorithms in mapreduce and streaming. In *SPAA*, 2013.

[21] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *SPAA*, 2011.

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

[23] H. Lin and J. Bilmes. A class of submodular functions for document summarization. In *NAACL/HLT*, 2011.

[24] M. Minoux. Accelerated greedy algorithms for maximizing submodular set functions. *Optimization Techniques, LNCS*, pages 234–243, 1978.

[25] B. Mirzasoleiman, A. Karbasi, R. Sarkar, and A. Krause. Distributed submodular maximization: Identifying representative elements in massive data. In *Neural Information Processing Systems (NIPS)*, 2013.

[26] G. L. Nemhauser and L. A. Wolsey. Best algorithms for approximating the maximum of a submodular set function. *Math. Oper. Research*, 1978.

[27] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions - I. *Mathematical Programming*, 1978.

[28] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. 2006.

[29] C. Reed and Z. Ghahramini. Scaling the indian buffet process via submodular maximization. In *ICML*, 2013.

[30] M. G. Rodriguez, J. Leskovec, and A. Krause. Inferring networks of diffusion and influence. *ACM Transactions on Knowledge Discovery from Data*, 5(4):21:1–21:37, 2012.

[31] B. Schölkopf and A. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA, 2001.

[32] M. Seeger. Greedy forward selection in the informative vector machine. Technical report, University of California, Berkeley, 2004.

	Sieve-Streaming	online k-means	Stream-Greedy
wall clock time	4376 s	11905 s	77160 s
utility	$152 \cdot 10^6$	$152 \cdot 10^6$	$153 \cdot 10^6$

Table 2: Performance of Sieve-Streaming with respect to Stream-Greedy and online k-means. To compare the computational cost, we report the wall clock time in second. In this experiment we used $k = 100$.

[33] R. Sipos, A. Swaminathan, P. Shivaswamy, and T. Joachims. Temporal corpus summarization using submodular word coverage. In *CIKM*, 2012.

[34] A. Tsanas, M. A. Little, P. E. McSharry, and L. O. Ramig. Enhanced classical dysphonia measures and sparse regression for telemonitoring of parkinson's disease progression. In *ICASSP*, 2010.

[35] J. Vitter. Random sampling with a reservoir. *ACM Trans. Mathematical Software*, 11(1):37–57, 1985.

APPENDIX

Proof of Proposition 5.1. We will first prove by induction that after adding $|S|$ elements to S the solution will satisfy the following inequality

$$f(S) \geq \frac{v|S|}{2k} \tag{9}$$

The proof is by induction. The base case $f(\emptyset) \geq 0$ is easy to see. Assume by induction that equation 9 holds for set S and we add element e. Then we know by the condition of the algorithm that $f(S \cup \{e\}) - f(S) \geq \frac{v/2 - f(S)}{k - |S|}$. Simplifying we get $f(S \cup \{e\}) \geq f(S)(1 - \frac{1}{k-|S|}) + \frac{v}{2(k-|S|)}$. Substituting equation 9 into this equation and simplifying we get the desired result $f(S \cup \{e\}) \geq \frac{v(|S|+1)}{2k}$.

The rest of the proof is quite simple. There are two cases.

- Case 1: At the end of the algorithm S has k elements. Then from equation 9 we get $f(S) \geq \frac{v}{2k} \cdot k = \frac{v}{2} \geq \frac{\alpha}{2}\,\texttt{OPT}$.
- Case 2: At the end of the algorithm $|S| < k$. Then let A^* be the optimal solution with $A^* - S = \{a_1, a_2, \ldots, a_l\}$ and $A^* \cap S = \{a_{l+1}, a_{l+2}, \ldots, a_k\}$. Let for each element $a_j \in A^* - S$, it be rejected from including into S when the current solution was $S_j \subseteq S$. Then we have the two inequalities that $f_{S_j}(a_j) \leq \frac{v/2 - f(S_j)}{k - |S_j|}$ and that $f(S_j) \geq \frac{v|S_j|}{2k}$. Combining the two we get $f_{S_j}(a_j) \leq \frac{v}{2k}$ which implies that $f_S(a_j) \leq \frac{v}{2k}$. Let A_j denote $\{a_1, a_2, \ldots, a_j\}$.

$$\begin{aligned} f(S \cup A^*) - f(S) &= \sum_{j=1}^{l} f(S \cup A_j) - f(S \cup A_{j-1}) \\ &\leq \sum_{j=1}^{l} f(S_j \cup \{a_j\}) - f(S_j) \\ &\leq \sum_{j=1}^{l} \frac{v}{2k} \leq \frac{v}{2} \leq \frac{\texttt{OPT}}{2} \\ \Rightarrow \texttt{OPT} - f(S) \leq \frac{\texttt{OPT}}{2} &\Rightarrow f(S) \geq \frac{1}{2}\texttt{OPT} \end{aligned}$$

Here the first inequality is due to submodularity, second is by definition of $A^* - S$.

□

Proof of Proposition 5.2. The proof directly follows from Theorem 5.1 and the fact that there exists a $v \in O$ such that $(1 - \epsilon)\texttt{OPT} \leq v \leq \texttt{OPT}$. □

Proof of Theorem 5.3. First observe that when a threshold v is instantiated any element with marginal value at least $v/(2k)$ to S_v appears on or after v is instantiated. This is because if such an element e_i appeared before v was instantiated then $v \in O_i$ and would have been instantiated when e_i appeared. Then the proof directly follows from Theorem 5.2 and the fact that Algorithm 3 is the same as Algorithm 2 with a lazy implementation. □

Proof of Theorem 6.2. The proof closely follows the proof of the algorithm when we evaluate the submodular function exactly from Proposition 5.1, 5.2 and Theorem 5.3. From Proposition 6.1 we get that for any set S, $|f_W(S) - f(S)| \leq \frac{\epsilon}{k}$. Hence, as we are taking k elements, the error for the solution after taking each element adds up and we get $f(S) \geq (1/2 - \epsilon)\texttt{OPT} - \frac{\epsilon}{k} \cdot k$ which is the desired result. □

Bad example for Stream-Greedy

We show that there exists a family of problem instances of streaming submodular maximization under a k cardinality constraint, where it holds for the solution S produced by Stream-Greedy after one pass through the data, that $f(S)/\texttt{OPT} \leq 1/k$. For this claim, take the weighted coverage of a collection of sets: Fix a set X and a collection V of subsets of X. Then for a subcollection $S \subseteq V$, the monotone submodular function f is defined as

$$f_{\text{cov}}(S) = \sum_{x \in \cup_{v \in S} v} w(x).$$

Here, $w(\cdot)$ is the weight function that assigns positive weights to any element from X. Now, let us assume that X is the set of natural numbers and elements arrive as follows:

$$\begin{aligned} &\{1\}, \{2\}, \ldots, \{k\}, \{1, 2, \ldots, k\} \\ &\{k+1\}, \{k+2\}, \ldots, \{2k\}, \{k+1, k+2, \ldots, 2k\} \ldots \\ &\{k^2{+}1\}, \{k^2{+}2\}, \ldots, \{k^2{+}k\}, \{k^2{+}1, k^2{+}2, \ldots, k^2{+}k\} \end{aligned}$$

Let $\epsilon << 1$. We define the weights as

$$\begin{aligned} w(1) = w(2) = \cdots = w(k) &= 1 \\ w(k+1) = w(k+2) = \cdots = w(2k) &= 1 + \epsilon \\ &\vdots \\ w(k^2+1) = w(k^2+2) = \cdots = w(k^2+k) &= 1 + k\epsilon \end{aligned}$$

Then it is clear that Stream-Greedy skips the non-singleton sets as they do not provide any benefit. In contrast, the optimum solution only consists of those sets. Now, after observing $O(k^2)$ elements from the above data stream, the ratio between the solution provided by Stream-Greedy and the optimum algorithm decays as $O(1/k)$.