Summarizing Edge-Device Data via Core Items

Damjan Gjurovski, Jan Heidemann, and Sebastian Michel

TU Kaiserslautern (TUK), Kaiserslautern, Germany
{gjurovski,j_heideman18,michel}@cs.uni-kl.de

Abstract. In this work, we consider the problem of summarizing a data stream through an item-based summary using core items. We consider an IoT setting, where computing such summaries at the edge devices instead of emitting the whole data stream can drastically reduce the network traffic and speed up further processing. Core items of a data stream are the items with the highest values for a given monotone submodular utility function. To create stream summaries, we propose the SoftSieving approach for parallel processing with low memory consumption and fast execution time while attaining acceptable utility gain. Through extensive experiments with real-world datasets, we show the suitability of our approach and its superiority over state-of-the-art competitors.

1 Introduction

The amount of data that is continuously generated by different applications and devices, like social networks and deployed sensing devices, is increasing at an unprecedented speed. Processing such a vast amount of data is still commonly done at centralized compute clusters where high computational power and network bandwidth are available for deep analytical tasks. The required data transfer from originating sources to such a centralized instance creates significant network traffic. This is especially visible when considering data sources on edge devices that collect and transmit data in real-time. We aim to minimize network traffic to gather data in centralized locations and, simultaneously, to speed up subsequent data processing. We propose to do so not by pushing the entire analytical processing to the edge devices —that often have limited compute power— but by compacting the stream using item-based summaries that represent the original data in an optimal way given an objective (utility) function.

As a motivating example, consider the task of determining outliers in camera surveillance footage to detect wildfire outbreaks in a widespread national park. When the camera footage has high resolution and high frame rate, and there are multiple cameras at multiple locations, it is easy to imagine how the amount of data to be transferred and further processing become difficult to manage, specifically in areas with only rudimentary wireless network coverage. A summary with only the most useful frames can drastically reduce the amount of data while keeping the important information to be further evaluated.

The most obvious and straightforward solution to summarize data are random sampling techniques [2, 27]. However, while they are understandable and

easy to implement, they can result in important information being lost [8]. To overcome this problem, one solution is to use a **utility function**, which can measure the informativeness [4], representativeness [3, 30], coverage [13], etc., of a selected data subset. With a utility function, the information gain for any new data point can be calculated and data points can accordingly be added to the summary. Typically, such functions belong to the class of *monotone submodular functions*, which means they are *non-decreasing*, and possess the *diminishing returns property*. Following Zhao et al. [29], we refer to these functions as *core item functions*, and the items identified by them are called *core items*.

As the problem of finding an optimal subset according to one of the functions is NP-hard [10], the main focus has been on finding good approximation algorithms [3, 6, 7, 15]. In data stream settings, the basis for most state-of-the-art approaches is the Sieve-Streaming algorithm, introduced by Badanidiyuru et al. [3], where a data stream summarization is performed by maximizing a submodular set function subject to a cardinality constraint. However, when considering stream processing at the edge, Sieve-Streaming and the related variants Sieve-Streaming++ [15] and ThreeSieves [6] exhibit limitations. We develop our approach by considering the existing algorithms and addressing their limitations.

The related research investigating submodular function maximization typically focuses on text data [9, 18, 19] which enables the usage of natural coverage functions. However, such data is rarely encountered in edge applications. Additionally, the datasets are often not realistic representations of data generated on the edge but rather come from machine learning and bring diverse data points for classification purposes [3, 6, 7, 15]. Furthermore, most evaluations are performed on setups with high processing power, not representative of edge devices.

Problem Statement, Contributions, and Outline In our work, data arrives in the form of data stream D on k edge devices. To avoid transferring every item e_i from D through the network, we aim at computing item-based summaries using core items directly at the edge devices. The core items will be computed as the items that have the highest value for a given monotone submodular utility function f. The task of computing the core items will be done in parallel on the $k-1$ edge devices such that 1 edge device will be responsible for consolidating and generating the global core item set.

In this paper, we make the following **contributions**.

- We introduce an approach for fast core items computation in a data stream at the edge, which we call *SoftSieving*.
- We have tailored Sieve-Streaming [3], Sieve-Streaming++ [15], and ThreeSieves [6] to be applicable inside an Apache Storm topology [1].
- We performed extensive experiments by using two real-world datasets measuring the processing times, latency, and utility values of the approaches.

The remainder of the paper is organized as follows. Section 2 discussed related work, followed by background information on data stream summarization and submodular functions in Section 3. Section 4 reviews the shortcomings of existing approaches and presents our approach. Section 5 reports on the result of the experimental evaluation before Section 6 concludes the paper.

2 Related Work

Although different notions of *core sets* exist in related work on data streams [13, 21], the terminology in this work is based on the one used by Zhao et al. [29]. The core items are representative items chosen from a data stream. The problem of *core items tracking* is to continuously maintain core items in a streaming setting. Their approach is based on probabilistic decay. Since this work deals with a setting where all items within a window are equally important, these approaches are not considered in the comparisons.

Sieve-Streaming [3] is a popular algorithm for submodular function maximization of data streams, and the basis for our approach. The authors formalize the problem of summarizing a data stream to the maximization of a submodular set function subject to a cardinality constraint. The algorithm, unlike some previous approaches [17, 23], uses a single pass over the data. This approach achieves the best possible approximation for this setting [11].

Sieve-Streaming++ [15] is motivated by the observation that Sieve-Streaming unnecessarily maintains sieves that cannot achieve the best utility value over all sieves anymore. These sieves can be safely discarded and replaced by sieves based on the new lower bound.

Dynamic Sieve Streaming [7], unlike the previous approaches, is explicitly aimed at a distributed IoT setting. It proves improved upper and lower bounds to be used for the active set utility function. However, the experiments were not done in an IoT-representative setting but on a machine with higher processing power and memory size. Moreover, the improvements are specific to active set, and we aim to compare approaches on multiple utility functions. Thus, this approach will not be included in our experiments.

The two previous algorithms do not weaken the theoretical guarantees of Sieve-Streaming. In contrast, ThreeSieves [6] ignores the theoretical worst case and aims for better practical performance. ThreeSieves maintains only one threshold, drastically reducing memory cost, and dynamically changes that threshold.

The related research on approaches that focus on submodular function optimization is vast [9, 18, 19, 23, 29]. However, these approaches will not be explored in more detail, as they either do not offer significant improvement over the mentioned approaches or do not apply to our edge streaming setting.

3 Preliminaries

3.1 Apache Storm

Apache Storm [1] is a popular real-time, distributed stream processing framework. Data processing in Storm is done through *topologies*. A topology consists of two types of components, *spouts* and *bolts*. Data flows through them in the form of tuple streams. Tuples are first emitted by the spouts. Bolts receive and process the tuples by executing an arbitrary function and can send as output a new tuple to another bolt for more complex operations. Apache Storm can

require too many resources that might not be available in typical edge devices. Thus, in our work, we use EdgeWise [12]. In Apache Storm, each operation is assigned to a thread and scheduling is handled by the operating system, which is not aware of congestion inside the topology. On the edge, this can lead to high latency and backpressure. In EdgeWise, there is a congestion-aware scheduler assigning the operations to threads from a fixed-size worker pool. Queue lengths are balanced and backpressure is minimized.

3.2 Stream Summarization

To form the summaries, we consider utility functions which can measure the informativeness, representativeness, coverage, etc., of the selected subset.
Utility Functions: To mathematically determine the quality of a summary, subsets of the data set need to be assigned a function value and then compared. This is the purpose of a utility function. For any new data item, the new value of the utility function can be used to determine whether to add it to the summary or not. A utility function f is any function $2^D \Rightarrow \mathbb{R}_{\geq 0}$ that assigns all subsets a nonnegative value [29]. The objective is to find the subset of size at most K with the best utility value. Formally put: $OPT = S^* = \underset{S \subseteq D, |S| \leq K}{\arg\max} f(S)$.
Submodular Functions: Suitable utility functions often belong to the class of *monotone submodular functions*. They have a diminishing returns property, i.e., $f(\{e\} \cup A) - f(A) \geq f(\{e\} \cup B) - f(B)$ for $A \subseteq B$. Additionally, they are monotone, so $f(\{e\} \cup A) - f(A) \geq 0$ for all e and A. The best approximation ratio for OPT in existing algorithms is $\mathcal{O}(\frac{1}{2} - \epsilon)$, which is also the theoretical maximum approximation ratio for the streaming setting [11].

Using the submodular functions as utility functions, we create data summaries, i.e., sets of core items. For a utility function f, the *marginal gain* for adding item e to the core item set S can be calculated as $\Delta_f(e \mid S) = f(S \cup \{e\}) - f(S)$. We assume $f(\emptyset) = 0$. In this work, we consider two different monotone submodular utility functions based on *active sets* and *exemplar-based clustering*.
Active set stems from *Gaussian Processes* (GP), which are used in nonparametric regression [28]. In GP the active set is used for efficiency and one way to choose an active set is the Informative Vector Machine (IVM) [16]. IVM is monotonic and submodular, as shown by Seeger [24]. The **K-medoids** problem aims to build clusters around exemplars from the set of data points [14]. To compute distances, it requires a nonnegative distance function. However, when working with data streams a problem arises since the distance to all data points needs to be known. Fortunately, the function is *additively decomposable* [22]. Thus, in our setting, we can continuously generate samples needed for the computation.

4 Algorithms for Core Item Sets Computation

In the following, we present our proposed approach by first analyzing the existing algorithms and identifying their limitations. The Sieve-Streaming algorithm [3]

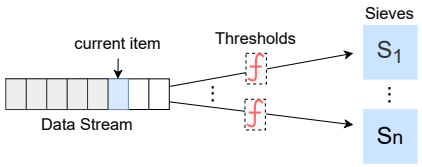


Fig. 1: Computing sieves in a data stream

and its variants, Sieve-Streaming++ [15] and ThreeSieves [6], provide solutions of high quality for data stream summarization. However, when considering data stream summarization on edge devices, we have identified that all of these approaches have certain limitations and can be further improved.

The *Sieve-Streaming* algorithm manages multiple sieves with different thresholds in parallel, as shown in Fig. 1. For a new data item, if it satisfies the marginal gain, it will be added to any sieve for which it exceeds the specific threshold. In other words, each sieve filters items based on its threshold. At least one of the sieves is expected to have a fitting threshold and produce a good core item set. The elements from the sieve with the best utility value are returned as a result. The thresholds are approximated using the maximum singleton value $m = \max_{e \in D} f(\{e\})$, i.e., the maximum value of the submodular utility function f for any single item e. The lower bound is m since that value has already been reached. If the summary has size K, the best case would be K items increasing the function value by m, resulting in upper bound of $K \cdot m$. The number of sieves depends on the parameter $\epsilon > 0$, which also influences the quality of the result.

If Sieve-Streaming is implemented in an Apache Storm topology without modifications, no parallelism is possible since the best utility value over all sieves is needed for the output. This will lead to increased processing load and increased latencies. To overcome this, we split the algorithm into a part where sieves are processing tuples and finding core items and a part where the core item sets of the different sieves are collected and compared.

When analyzing the Sieve-Streaming algorithm and topology, several limitations are immediately observable. First, the maximum singleton value m can be updated for every new entry, which also means that the number of sieves can change when this update occurs. Hence, in the proposed topology, there will be additional communication overhead between the bolts responsible for computing sieves and the collector bolt, especially when the maximum singleton value m changes frequently. Furthermore, the large number of created sieves can directly lead to problems with both storage and computation times.

Although *Sieve-Streaming++* [15] improves this approach by modifying the lower bound m for the optimal solution OPT as the algorithm is running, still the same limitations are present. The number of sieves remains a problem for low ϵ. Additionally, since the Storm topology remains the same, the problem of additional communication overhead when m changes frequently persists.

ThreeSieves [6] maintains only one single sieve and decreases its threshold over time. The rules for adding an item to a sieve are the same as in Sieve-

Streaming, however, when an item e is not added, a counter t is incremented by one. If the counter reaches the value of parameter T, the threshold value is decreased to the next-biggest estimate. This results in a threshold that is lowered more and more over time. The implementation of ThreeSieves in a Storm topology needs only one bolt. Although this eliminates the unnecessary communication overhead, it leads to an obvious limitation, i.e., lack of parallelization. This single bolt is a bottleneck, leading to increased processing latencies. Additionally, the algorithm performance depends heavily on the choice of parameter T. If it is too small, the threshold will get too low and the summary will be filled quickly, resulting in lower utility. If it is too large, useful items can be missed.

4.1 SoftSieving

Considering the limitations of the existing approaches, we build our new approach called *SoftSieving* by keeping the following design goals in mind.

- Use fewer sieves while keeping a high solution quality.
- Facilitate parallelization as much as possible.
- Make the processing as fast as possible by apt assignment of computations.

The first design goal steers in the direction of ThreeSieves [6] since its dynamic sieve can achieve high utility for many settings. The question then becomes how to increase the number of sieves. There should be bolt instances running in parallel that are responsible for computing their own sieves. Each instance should be independent of the others to minimize the communication overhead. Thus, there are two general methods that we can take.

1. Split the stream over n bolts and have all of them use the same threshold(s).
2. The n bolt instances process all tuples, each with different threshold(s).

Since one of our design goals is to minimize the number of sieves, it is natural to consider the first method. However, we cannot expect much gain from having additional sieves in every bolt. This comes directly from the ThreeSieves algorithm since the solution quality is already high while using only one sieve. The sieve is built starting from the highest possible threshold and lowering the threshold over time. As a result, high-utility items at the beginning of the data stream can be wrongly discarded. To better include these items, we can employ the reverse strategy by starting at the lowest threshold and increasing it. In this way, when the core item set of the sieve is filled with low utility items, it will be difficult to replace them later with higher utility items. Thus, we will include sieves that start from lower thresholds but still decrease them. The number of sieves depends on how much the thresholding calculations throttle the topology. Considering the first design goal, we will prefer a small, fixed number of sieves. Fixing the number of sieves prevents a rapidly increasing number of sieves and will enable fast inner-bolt processing. Since the data stream is split, we would require a collector bolt responsible for building the core item sets. However, when adding a new item, every sieve bolt would need the current set of core items for their sieves at all times. Thus, this will result in an extensive amount of updates.

Algorithm 1 SoftSieving

1: **for** $i = 1, ..., n$ **do**
2: $S_i \leftarrow \emptyset$, $t_i \leftarrow 0$, $b_i \leftarrow 0$
3: **while** $(e \leftarrow D.next()) != null$ **do**
4: $m \leftarrow \max(m, f(\{e\}))$, $O \leftarrow \{(1+\epsilon)^i \mid m \leq (1+\epsilon)^i \leq K \cdot m\}$
5: $V_n \leftarrow n$ equidistant samples from O, and $\max(O)$
6: Delete all S_i such that $v_i \notin O$ **and** Create new S_i all new $v_i \in O$
7: **for** $(S_i, v_i) \mid i = 1, \ldots, n$ and $v_i = V_n[i]$ **do**
8: **if** $\Delta_f(e \mid S_i) \geq \frac{\frac{v_i}{2} - f(S_i)}{K - |S_i|}$ **then**
9: **if** $|S_i| < K$ **then**
10: $S_i \leftarrow S_i \cup \{e\}$
11: **else if** $b_i < K$ **then**
12: $e_s \leftarrow \arg\min_{e \in S_i} \Delta_f(e \mid S_i\{e\})$
13: **if** $\Delta_f(e \mid S_i \backslash \{e_s\}) > \Delta_f(S_i)$ **then**
14: $S_i \leftarrow S_i \cup \{e\} \backslash \{e_s\}$
15: $b_i \leftarrow b_i + 1$
16: $t_i \leftarrow 0$
17: **else**
18: $t_i \leftarrow t_i + 1$
19: **if** $t_i \geq T$ **then**
20: $v_i \leftarrow$ next lowest threshold from O
21: $O \leftarrow O \backslash v_{next}$, $t_i \leftarrow 0$
22: **return** $\arg\max_{S_i \mid i=1,\ldots,n} f(S)$

To avoid numerous updates, we propose the usage of preemption [5]. Preemption means that once an item is added to the summary, it does not necessarily stay there but instead can be replaced by newer items at any point in time. The first K items are always added to the summary. Further items are swapped if they improve the solution by more than a defined threshold. If the summary is not fixed, we do not need to update the sieve bolts. They can continuously send the core items to the collector, which then checks if they improve the global solution when using swapping. The collector does not need to process as many items as the sieve bolts but still checking for every item in the summary quickly becomes infeasible with increasing K. Thus, the marginal gain of every item in the summary is stored. If the summary is full, new items are compared only to the item with the lowest gain. Consequently, we avoid the updates and fulfill our third design goal. To avoid excessive load in the collector and longer computations in the sieve bolts, the summaries will be restricted to size K.

Following these design decisions, as next, we present the **SoftSieving algorithm** as shown in Algorithm 1. In Lines (1–2), we initialize the algorithm. O is the set of thresholds, satisfying the defined lower and upper bounds. Compared to existing algorithms, not all thresholds are chosen from the set O, nor the maximum is only used. Instead, V_n gets n equidistant samples from O (Line 5). For the items from the data stream, we repeat steps 3 to 21. Similarly to the related approaches, we update m and the sieves (Lines 4 – 6). Next, we add items to the

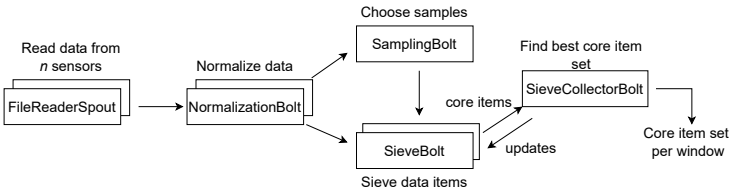


Fig. 2: SoftSieving storm topology

sieves (Lines 8–16) such that an item will be added if Lines 8 and 9 are satisfied. The amount by which adding e to S_i increases the utility function must be big enough such that sieve S_i can still reach the approximated optimal value v_i, i.e., increase by $v_i - f(S_i)$ after adding $K - |S_i|$ items to the summary. To account for some items in the summary influencing the utility value more that others, $\frac{v_i}{2}$ is used instead of v_i. If a sieve is full, we compare the next K items above the threshold against e_s, the item with the lowest utility gain from the core item set S_i (Lines 12–13). If item e offers higher utility, e and e_s are swapped. If an item is not added and Line 19 is satisfied, the sieve-thresholds are decreased (Lines 17–21). Finally, the sieve with the highest utility is returned (Line 22).

The actual Apache Storm **topology** responsible for realizing the SoftSieving algorithm is depicted in Fig. 2. The topology consists of five main components. First, the multiple instances of the *FileReaderSpout* are responsible for emitting the data from the data stream. Since for some data streams the data needs to be normalized, we introduce the *NormalizationBolt*. As explained in Section 3.2, we might need samples for computing the utility functions. For that reason, we introduce the *SamplingBolt*. The *SieveBolt* is responsible for maintaining and updating the sieves. The *SieveCollectorBolt* receives the core items from the *SieveBolts* and updates the core item set with the best utility. Although there can be several *SieveBolts*, there can be only one instance of the *SieveCollectorBolt*.

Theoretical Analysis We will now analyze the time and space cost as well as the utility for SoftSieving, with a focus on the Storm implementation. For m bolt instances and n sieves per bolt, SoftSieving stores $\mathcal{O}(n \cdot (m+1) \cdot K)$ items. Each bolt instance has its own n summaries with up to K items.

For the time analysis, consider a single item e. For the bolt it is assigned to, it takes n evaluations of the utility function, which take time T_K for a summary of size K. If instead the core item set and an additional K items have been sent, no further evaluations are performed. In the collector bolt, there are two possibilities. If the global summary S for item e is not full yet, e is added to S in $\mathcal{O}(1)$ and the new utility value is calculated, taking T_K time. Otherwise, a swap is considered, which takes T_K time for the utility evaluation of $S \backslash \{e_{min}\} \cup \{e\}$. Overall, the collector receives a maximum of $2K$ core items from each summary, i.e., a total of $m \cdot n \cdot 2K$. For a data stream of size N split between m bolt tasks, the time complexity is $\mathcal{O}(\frac{N}{m} \cdot (n \cdot T_K) + 2nmK \cdot T_K)$. Considering tumbling windows of size w, we get $\mathcal{O}(\frac{N}{m} \cdot (n \cdot T_K) + 2nwmK \cdot T_K)$. When considering

the updates of the maximum singleton value, since there is no upper bound on the number of updates, this increases to $\mathcal{O}(\frac{N}{m} \cdot (n \cdot T_K) + N \cdot T_K)$, or since $m, n << N$, $\mathcal{O}(N \cdot T_K)$. In practice, there will be significantly faster runtimes since the updates will not happen for every data item. For other approaches, the number of queries per element is often used [3, 6, 15], which is $\mathcal{O}(n)$ here since we always use n sieves.

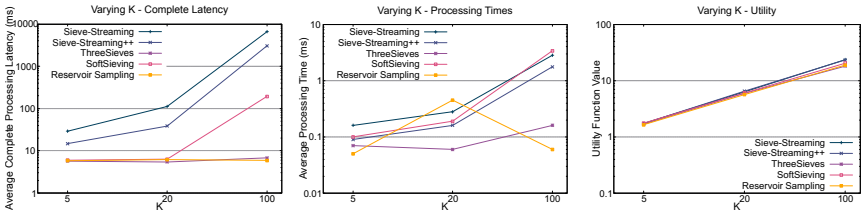
Regarding the quality, we can consider the result in ThreeSieves [6] as a lower bound since we always include the sieve with threshold $K \cdot m$. The authors prove that the solution S achieves an approximation to the optimal value OPT of $(1-\epsilon)(1-\frac{1}{\exp(1)})OPT$ with probability $(1-\alpha)^K$. When using n sieves, we expect to reach the optimal threshold v^* for one of them after $\frac{|O|}{2n} \cdot T$ items, instead of $\frac{|O|}{2} \cdot T$ for one sieve. This follows from choosing the thresholds equidistant from O, thus partition O into partitions of size $\frac{|O|}{n}$. In their proof, the probability of $(1-\alpha)^K$ comes from $P(v_1 = v_1^*, ..., v_K = v_K^*) > (1 - \frac{-\ln(\alpha)}{T})^K$, where, v_i^* are the thresholds of the sieve and $v_i = \Delta(S \mid e_i)$ are the marginal gains achieved by the greedy algorithm (for details see [6], appendix). For SoftSieving, any of the n sieves can achieve these values, increasing the probability. We can approximate it with $1-((1-\alpha)^K)^n$ for n sieves, but recall that the n thresholds are not chosen at random, but to be equidistant. The swapping yields no such improved theoretical guarantees since we try to swap with the item with the lowest marginal gain. This is done for performance reasons but can result in swapping new item e only with its most similar item from the core item set, bringing minimal improvement.

5 Experimental Evaluation

The proposed topology and the considered competitors are implemented in EdgeWise [12]. The experiments were performed on a Raspberry Pi 4 with a 1.5 Ghz Quad-Core-processor and 8 GB of main memory. The Raspberry Pi runs an Apache Storm Cluster and ZooKeeper, where the topologies were submitted. We carried out experiments with the goal of answering the following questions.

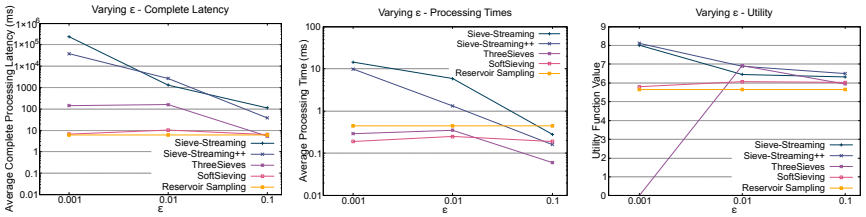
1. Can we find summaries in a reasonable timeframe which is preferable to sending all data items to the core without summarization?
2. How well can the approaches handle increasing load from the IoT-device(s)?
3. Does the SoftSieving approach offer significantly better processing times or solution quality compared to existing algorithms?

Thus, we measure the total latency over the topologies (questions 1 and 3). We measure the processing times in the SieveBolts (questions 2 and 3) and the utility for all approaches (question 3). We perform experiments with varying summary sizes $K \in \{5, 20, 100\}$, window sizes $w \in \{1000, 10000, 100000\}$, and the parameter impacting the number of sieves $\varepsilon \in \{0.1, 0.01, 0.001\}$. For processing times and total latency, the average over all tuples is taken, the utility is the average value over all windows.



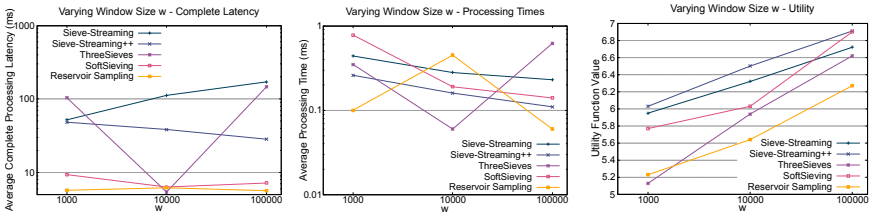
(a) Total processing latency (b) Local processing time (c) Utility function value

Fig. 3: Varying summary size ($K = 5$, $K = 20$, $K = 100$) - log scaled



(a) Total processing latency (b) Local processing time (c) Utility function value

Fig. 4: Varying number of sieves ($\varepsilon = 0.001$, $\varepsilon = 0.01$, $\varepsilon = 0.1$) - 4a, 4b log-scaled



(a) Total processing latency (b) Local processing time (c) Utility function value

Fig. 5: Varying window size ($w = 1000$, $w = 10000$, $w = 100000$)

Competitors: We considered the Sieve-Streaming [3], Sieve-Streaming++ [15], ThreeSieves [6], and Reservoir Sampling [27]. All the competitors were implemented in a Storm topology. All sieve-based algorithms are executed with parameters $K = 20, w = 10000, \varepsilon = 0.1$, unless otherwise specified. ThreeSieves and SoftSieving additionally use $T = 100$ as a standard parameter and SoftSieving uses $n = 4$. The utility functions used are Active Set and k-medoids (Section 3.2). The topologies are configured to have 10 SieveBolts and one SieveCollectorBolt. The NormalizationBolts have 5 instances. The SamplingBolt uses one instance.

Datasets: The approaches were evaluated on two real-world datasets, specifically chosen to simulate a real IoT scenario. The first dataset is a *telemetry*

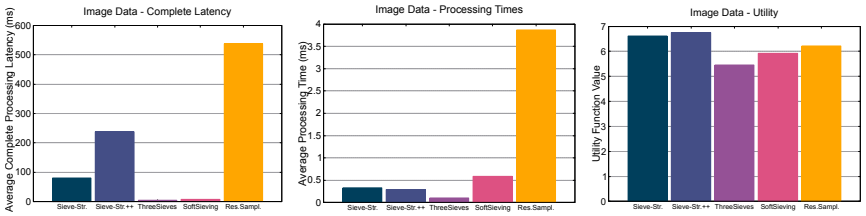
dataset from Stafford [26]. The data is collected from three IoT-devices, reading environmental sensor data in regular intervals. The measurements include temperature, humidity, CO, liquid petroleum gas, smoke, light, and motion, having 405184 entries. The distance between two items was implemented as sum of the euclidean distances of all measurements. The data is normalized. The second *images dataset* is an RGB-D dataset [20, 25]. It contains image frames from three Kinect sensors in an university hall. The images in the dataset are stored as 8 bits, 3 channels PPM images with 640x480 pixels. We preprocess the data and transform it into feature vectors of size 804. Distances are calculated as sum of the euclidean distances of all feature vector entries. Since the feature vector is normalized the NormalizationBolt is not necessary.

First, we performed experiments on the telemetry dataset. When one parameter is varied, the other remain on their standard values.

Varying Summary Size (K): Considering the total latency (Fig. 3a), all approaches are affected by the summary size, but Sieve-Streaming and Sieve-Streaming++ perform the worst, with SoftSieving being up to an order of magnitude below them. Reservoir Sampling stays consistent since the sampling calculations are not affected by K. ThreeSieves is also barely affected, which means that the increase in computation cost is not notable when using only one sieve. The processing time of the SieveBolt (Fig. 3b) increases as well. Reservoir Sampling, as it is not affected by the summary size, does not show a correlation to K. Fig. 3c depicts the average utilities where a higher value corresponds to a better approach. Clearly the results differ more for larger K. SoftSieving lies between Sieve-Streaming and ThreeSieves, but it constantly outperforms ThreeSieves. As expected, increasing the summary size leads to increased latency and processing time, while increasing the utility. Sieve-Streaming and Sieve-Streaming++ are the most dependent on K having latencies orders of magnitude larger compared to SoftSieving. ThreeSieves and Reservoir Sampling are barely affected by K and produce the best latencies. However, they produce the lowest utility.

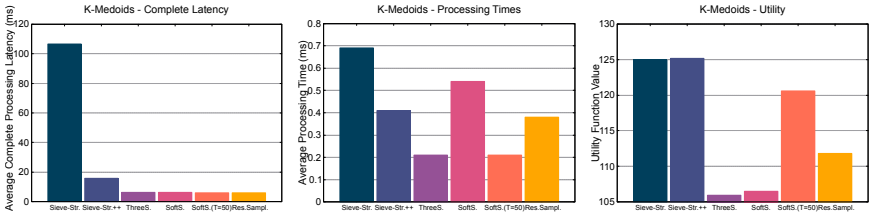
Varying Number of Sieves (ε): By increasing ε, the number of sieves decreases and with that the latency for all sieve-based approaches (Fig. 4a). This is most notable for Sieve-Streaming and Sieve-Streaming++. SoftSieving achieves lower total latency than ThreeSieves, which is a direct consequence of the improved parallelization. The local processing times follow the same trend (Fig. 4b). Sieve-Streaming and Sieve-Streaming++ have high processing times for low ε. The utility values in Fig. 4c show the dependence of Sieve-Streaming and Sieve-Streaming++ on the number of sieves. Their utility decreases when ε increases and fewer sieves are used. SoftSieving shows a slight increase with increasing ε, while ThreeSieves has lower utility for both $\varepsilon = 0.001$ and $\varepsilon = 0.1$. The number of sieves does not change for ThreeSieves, but a lower ε means that, with constant T, it takes longer to decrease the threshold to a suitable value. The extreme case is shown for $\varepsilon = 0.001$, where no items were added to the core item set since the threshold remained too high.

Varying Window Size (w): Different window sizes show how quickly the approaches build a high-quality summary and how much they improve on their



(a) Total processing latency (b) Local processing time (c) Utility function value

Fig. 6: Image dataset (standard parameters)



(a) Total processing latency (b) Local processing time (c) Utility function value

Fig. 7: K-Medoids utility function (standard parameters)

summaries over time. The total latency stays consistent for most approaches when varying w, with the exception of ThreeSieves for $w = 10000$ (Fig. 5a). The local processing times (Fig. 5b) of Sieve-Streaming/++ and SoftSieving show a decrease when increasing w. The cause for this can be the increase in the number of sieves that have completed their summary and no longer require processing over the window. For the utility (Fig. 5c), there is an increase for larger w for all approaches. Interestingly, this includes reservoir sampling. This may indicate that for our dataset, the data changes over time, and evenly distributed sampling naturally includes these changes. SoftSieving achieves higher utility as w increases such that it reaches the utility of Sieve-Streaming and Sieve-Streaming++ for $w = 100000$ while constantly outperforming ThreeSieves.

Image Dataset: We evaluate on the images dataset using the standard parameters and the active set utility function. For the complete processing latency (Fig. 6a), the approaches show a similar distribution to the sensor data, with the exception of Reservoir Sampling. When looking at the local processing times (Fig. 6b) it is apparent that the cause for this is the slow sampling in the respective bolt. The utility values (Fig. 6c) show SoftSieving being lower than Reservoir Sampling, with ThreeSieves being the lowest by a large margin. This shows the impact of tuning T, or parameters like ε. Consequently, they are surpassed by random sampling in utility since they are too restrictive with their thresholds.

K-Medoids Utility Function: We evaluated the k-medoids utility function on the telemetry dataset with the standard parameters. All topologies include a

SamplingBolt, collecting sample sets of size 50 to use for the per window utility calculations. The complete processing latency (Fig. 7a) is low for all approaches except Sieve-Streaming. This is because the effects of having more sieves, and therefore more utility function evaluations, are stronger with k-medoids. The local processing times (Fig. 7b) of SoftSieving are higher than ThreeSieves. Since the total latency remains low, this indicates effective partitioning of the data stream. SoftSieving with $T = 50$ was included to highlight the importance of T. The utility of SoftSieving (Fig. 7c), although higher than ThreeSieves, is low compared to the other approaches. However, the high utility for $T = 50$ shows that this is a result of the choice of T. In conclusion, the results are comparable to active sets, but the choice of T can greatly influence our approach.

6 Conclusion

We investigated the problem of continuously extracting core-item–based summaries from a data stream. Specifically, we looked at an edge setting, where finding such summaries can save network load and speed up centralized applications that depend on the edge data. We proposed a new algorithm for data stream summarization using core items, called SoftSieving. It uses a fixed, low number of dynamic sieves and enables parallelized processing. The summaries are soft, meaning that core items can be swapped for ones with greater utility gain. We compared the performance to the state-of-the-art sieve-based algorithms in an extensive experimental evaluation and showed that our approach achieves acceptable balance between fast processing and high utility.

Acknowledgments This work has been partially funded by the German Federal Ministry of Education and Research under grant number 28DE113C18 (DigiVine). The responsibility for the content of this publication lies with the authors.

References

1. Apache storm. `https://storm.apache.org/` (2011), accessed: 2022-04-09
2. Aggarwal, C.C.: On biased reservoir sampling in the presence of stream evolution. In: VLDB. pp. 607–618. (2006)
3. Badanidiyuru, A., Mirzasoleiman, B., Karbasi, A., Krause, A.: Streaming submodular maximization: Massive data summarization on the fly. In: KDD (2014)
4. Brown, G., Pocock, A.C., Zhao, M., Luján, M.: Conditional likelihood maximisation: A unifying framework for information theoretic feature selection. J. Mach. Learn. Res. **13**, 27–66 (2012)
5. Buchbinder, N., Feldman, M., Schwartz, R.: Online submodular maximization with preemption. In: SIAM. pp. 1202–1216 (2014)
6. Buschjäger, S., Honysz, P.J., Pfahler, L., Morik, K.: Very fast streaming submodular function maximization. In: ECML PKDD. pp. 151–166. Springer (2021)
7. Buschjäger, S., Morik, K., Schmidt, M.: Summary extraction on data streams in embedded systems. In: IOTSTREAMING@ PKDD/ECML (2017)

8. Chen, J., Zhang, Q.: Distinct sampling on streaming data with near-duplicates. In: PODS. pp. 369–382. (2018)
9. Dasgupta, A., Kumar, R., Ravi, S.: Summarization through submodularity and dispersion. In: ACL (1). pp. 1014–1022 (2013)
10. Feige, U.: A threshold of ln n for approximating set cover. Journal of the ACM (JACM) **45**(4), 634–652 (1998)
11. Feldman, M., Norouzi-Fard, A., Svensson, O., Zenklusen, R.: The one-way communication complexity of submodular maximization with applications to streaming and robustness. In: SIGACT. pp. 1363–1374 (2020)
12. Fu, X., Ghaffar, T., Davis, J.C., Lee, D.: {EdgeWise}: A better stream processing engine for the edge. In: USENIX ATC. pp. 929–946 (2019)
13. Indyk, P., Mahabadi, S., Mahdian, M., Mirrokni, V.S.: Composable core-sets for diversity and coverage maximization. In: PODS. pp 100–108 (2014)
14. Kaufman, L., Rousseeuw, P.J.: Partitioning around medoids (program pam). Finding groups in data: an introduction to cluster analysis. Wiley Series in Probability and Statistics. **344**, 68–125 (1990)
15. Kazemi, E., Mitrovic, M., Zadimoghaddam, M., Lattanzi, S., Karbasi, A.: Submodular streaming in all its glory: Tight approximation, minimum memory and low adaptive complexity. In: ICML. pp. 3311–3320 (2019)
16. Lawrence, N., Seeger, M., Herbrich, R.: Fast sparse gaussian process methods: The informative vector machine. In: NIPS.**15** (2002)
17. Leskovec, J., Krause, A., Guestrin, C., Faloutsos, C., VanBriesen, J., Glance, N.: Cost-effective outbreak detection in networks. In: SIGKDD. pp. 420–429 (2007)
18. Lin, H., Bilmes, J.: Multi-document summarization via budgeted maximization of submodular functions. In: HLT-NAACL. pp. 912–920 (2010)
19. Lin, H., Bilmes, J.: A class of submodular functions for document summarization. In: HLT. pp. 510–520 (2011)
20. Luber, M., Spinello, L., Arras, K.: People tracking in rgb-d data with on-line boosted target models. In: IROS. pp. 3844–3849 (2011).
21. Mirrokni, V.S., Zadimoghaddam, M.: Randomized composable core-sets for distributed submodular maximization. In: STOC. pp. 153–162. (2015)
22. Mirzasoleiman, B., Karbasi, A., Sarkar, R., Krause, A.: Distributed submodular maximization: Identifying representative elements in massive data. In: NIPS.**26** (2013)
23. Nemhauser, G.L., Wolsey, L.A., Fisher, M.L.: An analysis of approximations for maximizing submodular set functions—i. Mathematical programming **14**(1), 265–294 (1978)
24. Seeger, M.: Greedy forward selection in the informative vector machine. Technical report, UC Berkeley (2004)
25. Spinello, L., Arras, K.O.: People detection in rgb-d data. In: RSJ. pp. 3838–3843 (2011).
26. Stafford, G.A.: Environmental sensor telemetry data. `https://www.kaggle.com/datasets/garystafford/environmental-sensor-data-132k` (2020), accessed: 2022-04-28
27. Vitter, J.S.: Random sampling with a reservoir. ACM TOMS. **11**(1), 37–57 (1985)
28. Williams, C.K., Rasmussen, C.E.: Gaussian processes for machine learning, vol. 2. MIT press Cambridge, MA (2006)
29. Zhao, J., Wang, P., Tao, J., Zhang, S., Lui, J.C.: Continuously tracking core items in data streams with probabilistic decays. In: ICDE. pp. 769–780 (2020).
30. Zhuang, H., Rahman, R., Hu, X., Guo, T., Hui, P., Aberer, K.: Data summarization with social contexts. In: CIKM. pp. 397–406. ACM (2016)