4-2018

A sliding-window framework for representative subset selection

Yanhao WANG

Yuchen LI
Singapore Management University, yuchenli@smu.edu.sg

Kian-Lee TAN

2018 IEEE 34th International Conference on Data Engineering

A Sliding-Window Framework for Representative Subset Selection

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

Yuchen Li
School of Information Systems
Singapore Management University
yuchenli@smu.edu.sg

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

***Abstract*—Representative subset selection (RSS) is an important tool for users to draw insights from massive datasets. A common approach is to model RSS as the submodular maximization problem because the utility of extracted representatives often satisfies the "diminishing returns" property. To capture the data recency issue and support different types of constraints in real-world problems, we formulate RSS as maximizing a submodular function subject to a d-knapsack constraint (SMDK) over sliding windows. Then, we propose a novel KnapWindow framework for SMDK. Theoretically, KnapWindow is $\frac{1-\varepsilon}{1+d}$-approximate for SMDK and achieves sublinear complexity. Finally, we evaluate the efficiency and effectiveness of KnapWindow on real-world datasets. The results show that it achieves up to 120x speedups over the batch baseline with at least 94% utility assurance.**

***Index Terms*—Data summarization; submodular maximization; data stream; sliding window; approximation algorithm**

I. Introduction

In the big data era, massive data is being continuously generated by a wide range of applications, e.g., social media, network traffic, and sensors. A critical task of immense importance is to retain only a small portion of valuable information for users to draw insights about massive datasets. A common approach is *representative subset selection* (RSS) [1]–[3] which extracts a concise set of representative elements from the source dataset. In practice, the representativeness of elements can be measured by information coverage [3], information entropy [1], [2], and so on. Since these notions of representativeness often satisfy the "diminishing returns" property, RSS can be modeled as the submodular maximization (SM) problem with certain budget constraints. In addition, developing efficient RSS algorithms over data streams for real-time analytics [1], [2] has also gained increasing research attention recently. However, there are mainly two drawbacks that limit the deployment of existing RSS algorithms in many real-world applications.

First, most of the streaming RSS algorithms can only work with a cardinality constraint, i.e., selecting k elements as representatives, and fail to support more complex constraints. In many real-world problems, RSS has more general constraints than cardinality [4], [5]. For example, to extract representatives from social data, the *number*, *length*, and *influence scores* of social posts are all considered as constraints [5]. Generally, the above constraints are practical examples of *knapsack constraints*. However, existing algorithms that only support a cardinality constraint cannot provide solutions with quality assurance for RSS in more general scenarios.

Second, existing techniques cannot effectively support dynamic RSS over sliding windows. In many cases, data elements arrive as a *stream* with high velocity [6]–[9]. Moreover, recent elements are more important and valuable than earlier ones in a data stream, where the sliding window [10] model is a natural way to capture such an essence. Although several RSS algorithms [1], [5], [11] have been developed for the append-only streaming model, RSS over sliding windows is still largely unexplored and, to the best of our knowledge, there is only one existing method [12] for this problem. Not surprisingly, it is also specific for one cardinality constraint.

Our Contribution. To address the limitations of existing techniques, it requires a framework for RSS that (1) supports general monotone submodular utility functions, (2) works with multiple knapsack constraints, and (3) maintains a set of representatives for a massive dataset over sliding windows efficiently. In this paper, we formulate RSS as maximizing a submodular function with a d-knapsack constraint (SMDK) and propose a novel KnapWindow framework to address SMDK over sliding windows.

We first propose the KnapStream algorithm for SMDK in append-only streams. KnapStream estimates the optimal utility value from observed elements and maintains a sequence of *candidates* for different estimations. Each candidate derives a unique *threshold* based on the estimation and includes a subset of observed elements whose marginal utility gains reach the threshold while the d-knapsack constraint is satisfied. After processing the stream, the candidate with the maximum utility is returned as the solution. KnapStream is $\frac{1-\varepsilon}{1+d}$-approximate for SMDK and achieves sublinear time and space complexity. It improves upon the state-of-the-art approximation factor of $\frac{1}{1+2d} - \varepsilon$ for SMDK in append-only streams [5]. Then, we design the KnapWindow framework for SMDK over sliding windows. KnapWindow maintains a sequence of s checkpoints $\{x_1, \ldots, x_s\} \subseteq [t', t]$ over the sliding window at any time t (t' is the start of the sliding window at time t). Each checkpoint x_i corresponds to a KnapStream instance on processing a substream of elements from time x_i to t. The first checkpoint x_1 always provides the solution at time t after post-processing. Theoretically, KnapWindow has the same approximation factor as KnapStream. To the best of

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00127

our knowledge, it is the first algorithm for SMDK in the sliding window model. Finally, we evaluate the efficiency and effectiveness of KNAPWINDOW compared with several baselines on two real-world datasets. The results show that it achieves up to 120x speedups over the batch baseline with at least 94% utility assurance.

II. PROBLEM FORMULATION

Data Streams. A data stream comprises an unbounded sequence of elements $V = \{v_1, v_2, \ldots\}$ and $v_t \in V$ is the t-th element of the stream. We consider the elements in V arrive one at a time in an *arbitrary* order. It only permits *one pass* over the stream and elements must be processed in the arrival order. Specifically, we focus on the sequence-based *sliding window* [10] model for data streams. Let W be the size of the sliding window. At any time t, the *active window* A_t always contains the W most recent elements (called active elements) in the stream[1], i.e., $A_t = \{v_{t'}, \ldots, v_t\}$ where $t' = \max(1, t - W + 1)$.

RSS over Sliding Windows. RSS selects a set of representative elements from the ground set according to a utility function with some budget constraint. In this paper, we target the class of *nonnegative monotone submodular* functions adopted in a wide range of RSS problems [1]–[3], [12].

Given a ground set V, we consider a set function $f : 2^V \rightarrow \mathbb{R}_{\geq 0}$ that maps any subset of elements to a nonnegative *utility* value. For a set $S \subseteq V$ and an element $v \in V \setminus S$, the *marginal gain* of $f(\cdot)$ is defined by $\Delta_f(v|S) \triangleq f(S \cup \{v\}) - f(S)$. Then, we define the *monotonicity* and *submodularity* of $f(\cdot)$ based on its marginal gain.

Definition 1 (Monotonicity & Submodularity)**.** *A set function $f(\cdot)$ is monotone iff $\Delta_f(v|S) \geq 0$ for any $S \subseteq V$ and $v \in V \setminus S$. $f(\cdot)$ is submodular iff $\Delta_f(v|S) \geq \Delta_f(v|T)$ for any $S \subseteq T \subseteq V$ and $v \in V \setminus T$.*

Intuitively, monotonicity means adding more elements does not decrease the utility value. Submodularity captures the "diminishing returns" property that the marginal gain of adding any new element decreases as a set grows larger.

A *knapsack* is defined by a cost function $c : V \rightarrow \mathbb{R}^+$ that assigns a positive cost to each element in V. Let $c(v)$ denote the cost of $v \in V$. The cost $c(S)$ of a set $S \subseteq V$ is the sum of the costs of its members, i.e., $c(S) = \sum_{v \in S} c(v)$. Given a budget b, we say S satisfies the *knapsack* constraint iff $c(S) \leq b$. W.l.o.g., we normalize the budget to 1 and the costs of elements to $(0, 1)$. A d-knapsack constraint ξ is defined by d cost functions $c_1(\cdot), \ldots, c_d(\cdot)$. Formally, we define $\xi = \{S \subseteq V : c_j(S) \leq 1, \forall j \in [d]\}$. We say S satisfies the d-knapsack constraint iff $S \in \xi$.

Then, we formulate RSS as an optimization problem of maximizing a utility function $f(\cdot)$ subject to a d-knapsack constraint ξ (SMDK) over the sliding window. At every time t, RSS provides a subset S_t of elements that (1) only contains active elements, (2) satisfies the constraint ξ, and (3) maximizes the utility function $f(\cdot)$. Formally,

$$\text{maximize} \quad f(S_t) \quad \text{s.t.} \quad S_t \subseteq A_t \wedge S_t \in \xi \tag{1}$$

We use $S_t^* = \text{argmax}_{S_t \subseteq A_t \wedge S_t \in \xi} f(S_t)$ to denote the optimal solution of SMDK at time t.

According to the definition of the d-knapsack constraint, a cardinality constraint with a budget k is a special case of a 1-knapsack constraint when $c(v) = \frac{1}{k}, \forall v \in V$. As SM with a cardinality constraint is NP-hard, SMDK is also NP-hard. A naïve approach to SMDK over the sliding window is to store all elements in A_t and run an algorithm for SMDK in the batch setting from scratch for every window slide. A typical batch algorithm for SMDK is COSTEFFECTGREEDY [4], which is designed for SM with 1-knapsack constraints but can be trivially extended to SMDK. It initializes $S_{UC} = \varnothing$ and $S_{CE} = \varnothing$. Then, it iteratively adds $v_{uc}^* = \text{argmax}_{v \in A_t \setminus S_{UC}} \Delta_f(v|S_{UC})$ to S_{UC} until $S_{UC} \cup \{v\} \notin \xi, \forall v \in A_t \setminus S_{UC}$. At the same time, it also iteratively adds $v_{ce}^* = \text{argmax}_{v \in A_t \setminus S_{CE}} \frac{\Delta_f(v|S_{CE})}{\delta(v)}$, where $\delta(v) = \max_{j \in [d]} c_j(v)$, to S_{CE} until $S_{CE} \cup \{v\} \notin \xi$, $\forall v \in A_t \setminus S_{CE}$. Finally, it returns the one with higher utility between S_{UC} and S_{CE} as the solution for SMDK at time t. However, COSTEFFECTGREEDY is undesirable for stream processing because it requires multiple passes over all active elements and invokes a large number of evaluations of $f(\cdot)$ for every window slide. Therefore, it calls for a more efficient method for RSS over sliding windows.

III. THE KNAPWINDOW FRAMEWORK

Overview. We propose an efficient KNAPWINDOW framework for SMDK over sliding windows in this section. KNAPWINDOW maintains a sequence of *checkpoints* $X_t = \{x_1, \ldots, x_s\}$ over the sliding window at any time t. At each checkpoint x_i, an instance of KNAPSTREAM is invoked for an append-only substream from v_{x_i} to v_t at time t. KNAPSTREAM tracks the optimal utility `OPT` by a sequence of estimations. Each *candidate* is maintained independently over the substream with a unique *threshold* derived from an estimation for `OPT`. To retrieve the solution for SMDK at time t, KNAPWINDOW returns the result of KNAPSTREAM corresponding to x_1 and KNAPSTREAM selects the candidate with the maximum utility among all candidates.

The KnapStream Algorithm maintains a solution for SMDK in an append-only stream $V_{x,y} = \{v_x, \ldots, v_y\}$. Basically, it uses the threshold-based framework [1], [11] for streaming SM. The mechanism of KNAPSTREAM depends on estimating the optimal utility value `OPT`. Although `OPT` cannot be exactly determined unless P=NP, KNAPSTREAM can track the lower and upper bounds of `OPT` from observed elements and maintain a sequence of candidates with different estimations for `OPT` within the range. Each candidate derives a unique threshold for marginal gains according to its estimation for `OPT`. When an element arrives, a candidate decides whether to include it based on the marginal gain of the element and its threshold. After processing the stream, the candidate with the maximal utility is returned as the result.

[1]We only discuss the sequence-based sliding window in this paper. But the proposed algorithms can naturally support the time-based sliding window.

Algorithm 1 KNAPSTREAM

Input: Stream $V_{x,y} = \{v_x, \ldots, v_y\}$, parameter λ
Output: Solution $S_{x,y}$ for SMDK w.r.t. $V_{x,y}$
1: $\Phi = \{(1+\lambda)^l | l \in \mathbb{Z}\}$
2: **for all** $\phi \in \Phi$ **do** $S_\phi \leftarrow \emptyset$
3: Initialize $m \leftarrow 0$, $M \leftarrow 0$ and $v_{max} \leftarrow nil$
4: **for** $t \leftarrow x, \ldots, y$ **do**
5: **if** $f(\{v_t\}) > f(\{v_{max}\})$ **then** $v_{max} \leftarrow v_t$
6: **if** $M < f(\{v_t\})/\gamma_t$ **then** $M \leftarrow f(\{v_t\})/\gamma_t, m \leftarrow f(\{v_t\})$
7: $\Phi = \{(1+\lambda)^l | l \in \mathbb{Z}, m \leq (1+\lambda)^l \leq M \cdot (1+d)\}$
8: Delete all S_ϕ such that $\phi \notin \Phi$
9: **for all** $\phi \in \Phi$ **do**
10: **if** $\Delta_f(v_t|S_\phi) \geq \frac{\delta_t \cdot \phi}{1+d} \wedge S_\phi \cup \{v_t\} \in \xi$ **then**
11: $S_\phi \leftarrow S_\phi \cup \{v_t\}$
12: $S_{max} \leftarrow \text{argmax}_{\phi \in \Phi} f(S_\phi)$
13: **return** $S_{x,y} \leftarrow \text{argmax}(f(S_{max}), f(\{v_{max}\}))$

Although following a similar scheme, KNAPSTREAM is different from the algorithms of [1] and [11] from two aspects: (1) the criterion for including an element considers not only the marginal gain of adding it but also its costs, i.e., it checks the cost-effectiveness of adding the element in each knapsack and includes it only when its cost-effectivenesses reaches the threshold in d knapsacks; (2) the singleton element with the largest utility is also considered as a candidate.

The pseudo-code of KNAPSTREAM is given in Alg. 1. It maintains three auxiliary variables: v_{max} stores the element with the maximum utility among observed elements at time t; m and M track the lower and upper bounds of OPT respectively (Lines 5–6). The sequence of candidates is maintained according to the up-to-date m and M (Lines 7–8). Then, each candidate checks whether to include v_t independently. For each $\phi \in \Phi$, if the marginal gain $\Delta_f(v_t|S_\phi)$ reaches $\frac{\delta_t \cdot \phi}{1+d}$ where $\delta_t = \max_{\forall j \in [d]} c_{tj}$ and the d-knapsack constraint is still satisfied after adding v_t into S_ϕ, v_t will be included into S_ϕ (Lines 9–11). Finally, after processing all elements in the stream, it first finds S_{max} with the maximum utility among the candidates, then compares S_{max} with $\{v_{max}\}$. The one with the higher utility is returned as the solution $S_{x,y}$ for SMDK w.r.t. the stream $V_{x,y}$ (Lines 12 and 13).

The theoretical results for KNAPSTREAM are shown in Theorems 1 and 2. We assume the cost of any element is bounded by γ and δ, i.e., $0 < \gamma \leq c_{tj} \leq \delta < 1, \forall t \in [n], \forall j \in [d]$. It is noted that γ and δ are not needed to be known in advance.

Theorem 1. *$S_{x,y}$ returned by* KNAPSTREAM *satisfies that $f(S_{x,y}) \geq \frac{1-\varepsilon}{1+d} \cdot f(S^*_{x,y})$ where $\varepsilon = \lambda + \delta$ and $S^*_{x,y}$ is the optimal solution for SMDK in $V_{x,y}$.*

Theorem 2. KNAPSTREAM *requires only one pass over $V_{x,y}$, buffers $O(\frac{\log(d \cdot \gamma^{-1})}{\gamma \cdot \lambda})$ elements, and has $O(\frac{\log(d \cdot \gamma^{-1})}{\lambda})$ update complexity per element.*

The KnapWindow Framework adapts KNAPSTREAM to SMDK in the sliding window model. It maintains a sequence of *checkpoints* and KNAPSTREAM instances over the sliding window. At time t, KNAPWINDOW maintains a sequence of s checkpoints $X_t = \{x_1, \ldots, x_s\}$ where $t' \leq x_1 < \ldots < x_s \leq t$. At each checkpoint x_i, a KNAPSTREAM instance $\mathcal{H}(x_i)$ is maintained by processing a substream from element v_{x_i} to the up-to-date element v_t. When $t > W$, if the first checkpoint x_1 expires ($x_1 < t'$ where $t' = max(1, t - W + 1)$), it will be deleted from checkpoints. Its corresponding KNAPSTREAM instance $\mathcal{H}(x_1)$ will be terminated as well. In addition, it does not set up a checkpoint for any arrival element for efficiency issue. Given an interval L, it only creates a checkpoint and initiates a KNAPSTREAM instance for every L elements. To provide the solution for SMDK in A_t, it uses the result of $\mathcal{H}(x_1)$. But it is noted that elements from $v_{t'}$ to v_{x_1-1} have not been seen by $\mathcal{H}(x_1)$ yet. Thus, it feeds the unseen elements to $\mathcal{H}(x_1)$ before returning the final solution.

Algorithm 2 KNAPWINDOW

Input: Stream $V = \{v_1, v_2, \ldots\}$, window size W, interval L
Output: Solution S_t for SMDK at time t
1: Initialize $s \leftarrow 0$, $X_0 \leftarrow \emptyset$
2: **for** $t \leftarrow 1, 2, \ldots$ **do**
3: **if** $t \in \{x : x = j \cdot L, j \in \mathbb{N}\}$ **then**
4: $s \leftarrow s + 1$, $x_s \leftarrow t$, and $X_t \leftarrow X_{t-L} \cup \{x_s\}$
5: Initiate KNAPSTREAM instance $\mathcal{H}(x_s)$
6: **while** $t > W \wedge x_1 < t'$ **do**
7: $X_t \leftarrow X_t \setminus \{x_1\}$ and terminate $\mathcal{H}(x_1)$
8: Shift the remaining checkpoints, $s \leftarrow s - 1$
9: **for** $i \leftarrow 1, \ldots, s$ **do**
10: $\mathcal{H}(x_i)$ processes v_t according to Lines 5–11 of Alg. 1
11: *// post-processing at time t*
12: $\mathcal{H}(x_1)$ processes elements from $v_{t'}$ to v_{x_1-1} according to Lines 5–11 of Alg. 1
13: **return** $S_t \leftarrow$ the solution of $\mathcal{H}(x_1)$

The pseudo-code of KNAPWINDOW is presented in Alg. 2. The sequence of checkpoints is initialized to $X_0 = \varnothing$. A checkpoint $x_s = t$ is created and added to X_t at each time $t = L, 2L, \ldots$. A KNAPSTREAM instance $\mathcal{H}(x_s)$ is invoked accordingly (Lines 3–5). Then, it deletes all expired checkpoints from X_t (Lines 6–8). Subsequently, each checkpoint processes v_t and updates the result independently. This procedure follows Lines 5–11 of Alg. 1. To provide the solution S_t for SMDK at time t, $\mathcal{H}(x_1)$ post-processes the elements from $v_{t'}$ to v_{x_1-1} (Line 12). Finally, the solution of $\mathcal{H}(x_1)$ after post-processing is returned as S_t (Line 13).

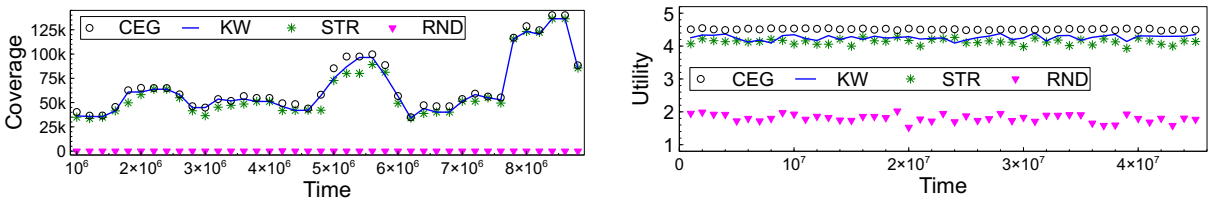
The approximation ratio and complexity of KNAPWINDOW are shown in Theorems 3 and 4.

Theorem 3. *S_t returned by* KNAPWINDOW *at time t satisfies that $f(S_t) \geq \frac{1-\varepsilon}{1+d} \cdot f(S^*_t)$ where $\varepsilon = \lambda + \delta$ and S^*_t is the optimal solution for SMDK at time t.*

Theorem 4. KNAPWINDOW *scans only one pass over the stream V, maintains $s = \lceil \frac{W}{L} \rceil$ checkpoints for A_t, buffers $O(\frac{s \cdot \log(d \cdot \gamma^{-1})}{\gamma \cdot \lambda})$ elements, updates an element in $O(\frac{s \cdot \log(d \cdot \gamma^{-1})}{\lambda})$ time, and takes $O(\frac{L \cdot \log(d \cdot \gamma^{-1})}{\lambda})$ time for post-processing.*

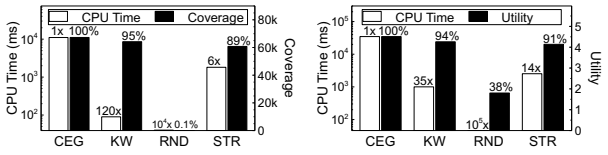
IV. EXPERIMENTS

We use two real-world datasets, **Twitter** and **Yahoo! Webscope**, in the experiments. **Twitter** is collected by the stream-



(a) Twitter (b) Yahoo! Webscope

Fig. 1. The utility of compared approaches over time.



(a) Twitter (b) Yahoo! Webscope

Fig. 2. The overall experimental results.

ing API containing about 8.9M tweets and 6.9M users. Selecting representative tweets is modeled as the *maximum coverage* problem with a 2-knapsack constraint. A tweet is represented as a set of users who retweet it and the goal is to select a set of tweets to cover the maximum number of users. We define a 2-knapsack constraint to limit the *number* and *total length* of selected tweets. **Yahoo! Webscope** is downloaded from https://webscope.sandbox.yahoo.com/. It consists of about 46M five-dimensional feature vectors. Selecting representative feature vectors is modeled as the *active set selection* problem with a 1-knapsack constraint. The utility for a set of vectors is measured by the IVM [1] function to maximize the information entropy. We define a 1-knapsack constraint to limit the *number* of selected vectors.

We compare the KNAPWINDOW (KW) framework with COSTEFFECTGREEDY [4] (CEG) for SMDK in the batch setting, the algorithm in [5] for SMDK in append-only streams (STR), and selecting a set of random elements from the active window (RND). All compared approaches are implemented in Java 8 and the experiments are conducted on a server running Ubuntu 16.04 with an Intel Xeon E7-4820 1.9GHz processor and 128 GB memory.

In the experiments, we feed the elements of both datasets to compared approaches in ascending order of timestamp and retrieve the representatives selected by each approach for every window slide. The window size W is 1M, the interval L for neighboring checkpoints is 100K, and the number of elements for each window slide is $1\% \times W$. In addition, the average cost of elements is 0.1 and λ in Alg. 1 is set to 0.1.

We present the *average CPU time* to process each window slide and *average utility* of the solutions returned by compared approaches in Fig. 2. First, KW achieves speedups of 120x and 35x over CEG on two datasets respectively. At the same time, the average utility of solutions provided by KW is at least 94% of those returned by CEG. Second, KW not only runs faster than STR but also provides solutions with higher utilities. Third, not surprisingly, RND cannot provide any meaningful solutions on both datasets. In Fig. 1, we show the *utilities* of the solutions returned by compared approaches from $t = W$ to the end of the stream. We can see the solutions of CEG always have the highest utilities among compared approaches. KW shows equivalent or better solution utilities than STR in most cases. The experimental results show that KW significantly improves the efficiency of CEG for SMDK over sliding windows with a trivial loss in utility.

V. CONCLUSION

In this paper, we studied the *representative subset selection* (RSS) problem in data streams. First, we formulated dynamic RSS as maximizing a submodular function subject to a d-knapsack constraint (SMDK) over sliding windows. We then proposed the KNAPWINDOW framework for this problem. Theoretically, KNAPWINDOW provided solutions for SMDK over sliding windows with an approximation factor of $\frac{1-\varepsilon}{1+d}$. Finally, the experimental results showed the efficiency and effectiveness of KNAPWINDOW compared with baselines.

REFERENCES

[1] A. Badanidiyuru, B. Mirzasoleiman, A. Karbasi, and A. Krause, "Streaming submodular maximization: Massive data summarization on the fly," in *KDD*, 2014, pp. 671–680.
[2] A. Krause and R. G. Gomes, "Budgeted nonparametric learning from data streams," in *ICML*, 2010, pp. 391–398.
[3] H. Zhuang, R. Rahman, X. Hu, T. Guo, P. Hui, and K. Aberer, "Data summarization with social contexts," in *CIKM*, 2016, pp. 397–406.
[4] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. Van Briesen, and N. Glance, "Cost-effective outbreak detection in networks," in *KDD*, 2007, pp. 420–429.
[5] Q. Yu, E. L. Xu, and S. Cui, "Submodular maximization with multi-knapsack constraints and its applications in scientific literature recommendations," in *IEEE GlobalSIP*, 2016, pp. 1295–1299.
[6] Y. Li, Z. Bao, G. Li, and K. Tan, "Real time personalized search on social networks," in *ICDE*, 2015, pp. 639–650.
[7] Y. Li, D. Zhang, Z. Lan, and K. Tan, "Context-aware advertisement recommendation for high-speed social news feeding," in *ICDE*, 2016, pp. 505–516.
[8] Y. Wang, Q. Fan, Y. Li, and K. Tan, "Real-time influence maximization on dynamic social streams," *PVLDB*, vol. 10, no. 7, pp. 805–816, 2017.
[9] D. Zhang, Y. Li, J. Fan, L. Gao, F. Shen, and H. T. Shen, "Processing long queries against short text: Top-k advertisement matching in news stream applications," *ACM TOIS*, vol. 35, no. 3, pp. 28:1–28:27, 2017.
[10] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM J. Comput.*, vol. 31, no. 6, pp. 1794–1813, 2002.
[11] R. Kumar, B. Moseley, S. Vassilvitskii, and A. Vattani, "Fast greedy algorithms in mapreduce and streaming," in *SPAA*, 2013, pp. 1–10.
[12] A. Epasto, S. Lattanzi, S. Vassilvitskii, and M. Zadimoghaddam, "Submodular optimization over sliding windows," in *WWW*, 2017, pp. 421–430.