A Framework for Summarizing and Analyzing Twitter Feeds

Xintian Yang
The Ohio State University
Columbus, OH
yangxin@cse.ohio-state.edu

Amol Ghoting
IBM T. J. Watson Research Center
Yorktown Heights, NY
aghoting@us.ibm.com

Yiye Ruan
The Ohio State University
Columbus, OH
ruan@cse.ohio-state.edu

Srinivasan Parthasarathy
The Ohio State University
Columbus, OH
srini@cse.ohio-state.edu

ABSTRACT

The firehose of data generated by users on social networking and microblogging sites such as Facebook and Twitter is enormous. Real-time analytics on such data is challenging with most current efforts largely focusing on the efficient querying and retrieval of data produced recently. In this paper, we present a dynamic pattern driven approach to summarize data produced by Twitter feeds. We develop a novel approach to maintain an in-memory summary while retaining sufficient information to facilitate a range of user-specific and topic-specific temporal analytics. We empirically compare our approach with several state-of-the-art pattern summarization approaches along the axes of storage cost, query accuracy, query flexibility, and efficiency using real data from Twitter. We find that the proposed approach is not only scalable but also outperforms existing approaches by a large margin.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data Mining

Keywords

Data Summarization, Analytics, Twitter

1. INTRODUCTION

Microblogging, a lightweight and easy form of communication within social networks such as Facebook, Google+ and Twitter, has become ubiquitous in its use with over 4 billion mobile devices worldwide of which over 1 billion support smart services. An increasing number of organizations and agencies are turning to extract and analyze useful nuggets of information from such services to aid in functions as diverse as emergency response, viral marketing, disease outbreaks, and predicting movie box office success. A fundamental challenge for effective human-computer interaction (querying and analytics) is the scale of the data involved. Twitter for instance has over 200 million users (and growing) and several hundred million tweets per day. Supporting interactive querying and analytics requires novel approaches for filtering and summarizing such data.

Given the diverse nature of applications, a number of queries may be of interest. Queries such as: *What are the currently trending topics?; What did a specific user tweet about today or yesterday?* are straightforward to support since one only needs to maintain recent data to answer such queries. However, often times users and organizations, are interested in capturing high level trends about both current and past activity – particularly highly trending past activity to understand the evolution of user interests and topic trending. For example complex queries of the following form would be of interest: *What topics were people talking about in a specific time interval (2 weeks in the past)?; How has a particular topic evolved across multiple time intervals?; How have a user's or a group's tweets, or topics they tweet on changed over time?* Answers to such questions may enable organizations to understand questions related to the lineage of topic evolution as well as to better understand user interests, their influence, and possibly build a model of trust for specific users and groups.

Answering such queries in real-time is challenging simply because of the scale of the data that is produced – the memory footprint will grow linearly with time and it can easily overwhelm the capacity of even the most powerful computer systems. In this study, we aim to build a summary of microblogging data, focusing on Twitter feeds, that can fit in a limited memory budget and can help to answer complex queries. In our view the desiderata for such a framework include: 1) efficient, incremental summary construction (ideally using a single pass and at pace with data influx rate); 2) budgeted memory which grows at most logarithmically with data influx; and 3) support for complex querying and analytics with low reconstruction error (particularly on more recent data, or on highly trending data). Ideally, we would like to be able to answer queries about the topics in a time interval in the past, and evolutionary events related to specific topics across multiple time intervals.

A novel framework (Figure 1) is proposed to address this desiderata. The elements of our framework include: a) SPUR, a batch summarization and compression algorithm that relies on a novel notion of pattern utility and ranking

KDD'12, August 12–16, 2012, Beijing, China.

which can be incrementally updated; b) D-SPUR, a dynamic variant of SPUR that accordingly merges summaries and maintains pyramidal time frames that grows logarithmically while enabling querying at multiple temporal contexts; and c) TED-SPUR, a topic and event based analytics tool to support complex querying on dynamic data.

We compare the effectiveness of various SPUR variants against state-of-the-art pattern summarization algorithms on a large corpus of Twitter data along the axes of compressability, reconstruction error, efficiency and flexibility in querying. We find that the SPUR variants are up to two orders of magnitude faster and can produce summaries with much lower reconstruction errors than extant approaches. Furthermore, maintaining temporal information in D-SPUR enables the approximate reconstruction of original data over arbitrary time intervals facilitating novel complex queries. We further demonstrate the efficacy of TED-SPUR, in analyzing temporal topic evolution and capturing real-world behavioral events.

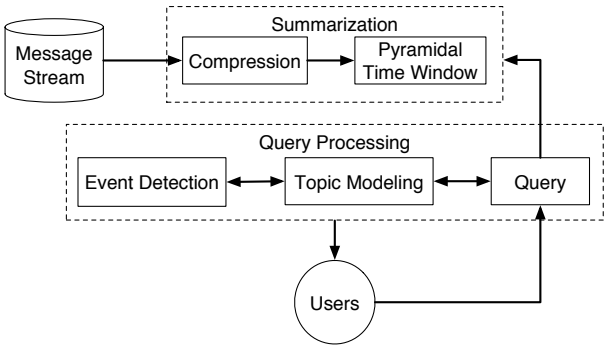


Figure 1: Overview of Summarization via Pattern Utility and Ranking (SPUR) Framework

2. STREAM SUMMARIZATION

In this section, we introduce the summarization component of our proposed stream processing framework.

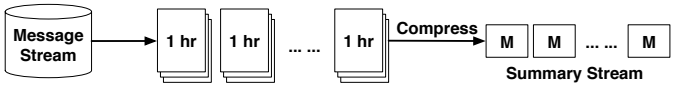


Figure 2: Division and compression of message stream

Given the input message stream with proper word stemming and stop-word removal performed, we divide it into approximately equal-sized batches, e.g. one hour per batch (the first arrow in Figure 2). To compress each batch of messages into a summary object which can fit in a constant memory budget M (the second arrow in Figure 2), we describe our SPUR algorithm in Section 2.1. Then in Section 2.2 we discuss the D-SPUR algorithm which ensures the summary size grows logarithmically with time.

2.1 SPUR

We develop a novel algorithm called SPUR (Summarization via Pattern Utility and Ranking) to summarize a batch of transactions with low compression ratio and high quality in a highly scalable fashion. Our basic idea of compressing a batch of tweets is to replace individual words with frequently used phrases that can cover the original content of the tweets. Consider the example in Figure 3 (left), each column represents a word and each row represents a tweet. The original tweets need 24 words to be stored in memory. However, if we use the frequent phrases as patterns to represent the tweets, we can save 10 of 24 words (Figure 3 right). Our approach represents each tweet as a transaction of words and a batch as a set of transactions. We cast the challenge of finding frequent phrases as a frequent itemset mining problem. To compress a batch of tweets into a summary with memory budget M, we aim to reduce the storage size by covering the transactions with frequent patterns. There are three main challenges one need to address: compressability, scalability and quality (of compressed summary).

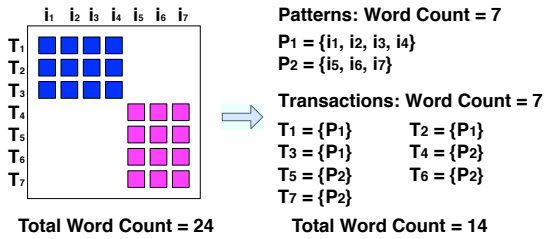


Figure 3: A batch of tweets compressed to a summary

Algorithm 1 provides the pseudo code of SPUR. The algorithm receives a batch of transactions B, a memory budget M, a support threshold σ and a false positive rate f as input. It outputs a summary which can fit in the memory budget M with false positive rate lower than f. First, our algorithm mines frequent itemsets above the support threshold σ as candidate patterns (line 1). We use LCM [13] for our purposes. Second, we define a function to capture the utility of each pattern in terms of compressing the transactions (line 3). Third, we rank all the candidates by their utility values and insert them into a priority queue (lines 2 – 4). Finally, we iteratively select the top ranked pattern to cover the items in transactions until we reach the memory budget M or the top pattern is not cost effective for compression any more (lines 5 – 13).

Algorithm 1 SPUR(B, M, σ, f)

```
1:  P ← MineFrequentPatterns(B, σ); Q ← φ; size ← 0;
2:  for all p ∈ P do
3:      p.utility = Utility(p, f); Q.insert(p);
4:  end for
5:  while size < M do
6:      p ← Q.top;
7:      if p.utility ≥ 0 then
8:          Replace(B, p); {Replace items using p}
9:          UpdateRank(Q, p); size = size + p.cost;
10:     else
11:         break;
12:     end if
13: end while
```

A coverage of transaction $T_i \in B$ using pattern p will replace the items in $T_i \cap p$ with a pointer to p. Two types of errors will be introduced: false negative errors are the items in the original data but are not covered by any pattern; false positive errors are items not belonging to a transaction but are introduced by a pattern coverage, i.e. the items in $p \setminus T_i$. False negative/positive rate is the ratio of false negative/positive errors over the total number of items in the transactions. In Algorithm 1, f is a threshold to control false positive rate; and the false negative rate is controlled by σ and M together where σ decides the infrequent items

dropped by frequent pattern mining and M controls the selection of frequent items in the summary, thus indirectly determines the frequent items that will be dropped. Next, we will first introduce the definition of pattern utility.

Pattern Utility and Ranking: We define the utility of a pattern p covering a transaction T_i as:

$$u(p, T_i) = |T_i \cap p| - 1 - |p \setminus T_i| \quad (1)$$

where $|T_i \cap p|$ captures the storage saved by p, 1 is the storage cost of a pointer to p in the compressed representation of T_i and $|p \setminus T_i|$ is the amount of false positive errors. Here $|T_i \cap p| - 1$ records the total savings in storage space whereas $|p \setminus T_i|$ penalizes false positive errors [1].

There is also a cost of space for storing pattern p, because we need to record the actual items in p. But this is not a cost for a coverage with an individual transaction, but the cost for a set of transactions. Given a set of transactions $C \subseteq B$, the total utility of covering all transactions in C with p is defined as the sum of the utilities on all transactions in C less the cost of storing pattern p:

$$U(p, C) = \sum_{T_i \in C} u(p, T_i) - |p| \quad (2)$$

The compression utility $Utility(p)$ of pattern p is the maximum value of $U(p, C)$ among all $C \subseteq B$. The *coverage set* $C(p)$ of p is defined as the set of transactions that yields this maximum value. So,

$$Utility(p) = \max_{C \subseteq B} U(p, C) \qquad C(p) = \arg\max_{C \subseteq B} U(p, C) \quad (3)$$

For example, in Figure 3, the best compression using pattern p_1 is to cover transactions T_1, T_2 and T_3. So $C(p_1) = \{T_1, T_2, T_3\}$, $u(p_1, T_i) = |T_i \cap p_1| - 1 - |p_1 \setminus T_i| = 3$ for $i = 1$ to 3 and $Utility(p_1) = \sum_{i=1}^{3} u(p_1, T_i) - |p_1| = 5$. Our compression algorithm needs to find the value of $Utility(p)$ for each pattern p and the transactions in $C(p)$ with the constraint that the false positive rate should be below a threshold f.

Let's first consider a simple case where no false positive errors are allowed in a coverage. In this case p can only cover transactions that contain p. Suppose pattern p with support $\sigma(p)$ and length $l(p)$, if we use $sup(p) = \{T | p \subseteq T\}$ to represent the set of all transactions that contain p, then $C(p) = sup(p)$. Therefore,

$$\begin{aligned} Utility_{no\_fp}(p) &= \sum_{T_i \in sup(p)} (|T_i \cap p| - 1 - |p \setminus T_i|) - |p| \\ &= \sum_{T_i \in sup(p)} (l(p) - 1) - l(p) \\ &= l(p) \cdot \sigma(p) - \sigma(p) - l(p) \end{aligned}$$

Next, let's increase the complexity of the pattern coverage problem by allowing false positive errors but no limitation of the false positive rate. Each transaction T_i will add $u(p, T_i)$ to $Utility(p)$. So the value of $Utility(p)$ is at maximum when all T_i's with $u(p, T_i) \geq 0$ are in p's coverage set $C(p)$. We can rewrite the definition of $u(p, T_i)$ as:

$$\begin{aligned} u(p, T_i) &= |T_i \cap p| - 1 - (|p| - |T_i \cap p|) \\ &= 2 \cdot |T_i \cap p| - 1 - l(p) \end{aligned}$$

[1] A pattern coverage will not introduce false negative errors, because the support threshold in frequent pattern mining decides the infrequent items we dropped.

So the set of transactions that can maximize p's utility is $C(p) = \{T \ s.t. \ |T \cap p| \geq (l(p) + 1)/2\}$. It is inefficient if we intersect all possible pattern and transaction pairs. The following theorem provides a faster way to find $C(p)$ by only considering p and its sub-patterns' supporting transactions.

THEOREM 1. *Suppose* $C'(p) = \{\bigcup sup(p_i) \,|\, p_i \subseteq p$ *and* $l(p_i) \geq (l(p) + 1)/2\}$, *where* $sup(p_i)$ *represents all transactions containing* p_i, *then* $C(p) = C'(p)$.

PROOF. First show $C(p) \subseteq C'(p)$.

$\forall T_i \in C(p)$, $|T_i \cap p| \geq (l(p) + 1)/2$. Let $p' = T_i \cap p$, then $p' \subseteq p$ and $l(p') \geq (l(p) + 1)/2$. $p' \subseteq T_i \Rightarrow T_i \in sup(p') \subseteq C'(p)$. So $C(p) \subseteq C'(p)$.

Then show $C'(p) \subseteq C(p)$.

$\forall T_i \in C'(p)$, $\exists p_i$ s.t. $T_i \in sup(p_i)$, $p_i \subseteq p$ and $|p_i| \geq (l(p)+1)/2$. Then $p_i \subseteq T_i \cap p$ and $|T_i \cap p| \geq |p_i| \geq (l(p)+1)/2 \Rightarrow T_i \in C(p)$. So $C'(p) \subseteq C(p)$. □

Theorem 1 shows that the utility value of a pattern p can be maximized by covering all transactions that contain sub-patterns of p with more than half of the length of p. We essentially replace p's sub-patterns with p in those transactions. This process will reduce the storage size by introducing false positive items. However, we cannot control the upper bound of the false positive rate in this strategy. Next, we will discuss how we can guarantee that the false positive rate in the summary is below a threshold f.

Algorithm 2 Utility(p, f)

```
1: p.utility ← l(p)·σ(p) − σ(p) − l(p); p.coverage_set ← sup(p);
2: p.replaced_patterns ← {p}; area ← l(p)·σ(p); fp_error ← 0; {Safe
   to add transactions without false positive}
3: sub(p) ← all sub-patterns of p;
4: sort sub(p) by pattern length from long to short;
5: for all p_i ∈ sub(p) do
6:    if |p_i| ≥ (|p| + 1)/2 then
7:       transactions ← sup(p_i) \ p.coverage_set; {First find the new
         transactions that can potentially be covered.}
8:       new_area ← area + l(p) · |transactions|;
9:       new_error ← fp_error + (l(p) − l(p_i)) · |transactions|;
10:      if new_error / new_area ≤ f then
11:         p.utility ← p.utility + |transactions| · (2·l(p_i) − l(p) − 1);
            {Update p's utility}
12:         p.coverage_set ← p.coverage_set ∪ transactions;
13:         p.replaced_patterns ← p.replaced_patterns ∪{p_i};
14:         area ← new_area; fp_error ← new_error;
15:      else
16:         break;
17:      end if
18:   else
19:      break;
20:   end if
21: end for
```

Suppose p_i is a sub-pattern of p, if we replace p_i with p in the transactions that contain p_i but not p, the false positive rate is $1 - l(p_i)/l(p)$. Therefore, longer sub-patterns will introduce lower false positive rate. To control the false positive rate below a threshold f, Algorithm 2 first sorts the sub-patterns of p from long to short (line 3 – 4) and keeps replacing the sub-patterns in this order until the false positive rate is higher than f (line 5 – 21). With this greedy strategy, Algorithm 2 can find the maximum utility of a pattern below a false positive rate threshold. Algorithm 2 also generates the transactions in the coverage (p.coverage_set) as well as the sub-patterns that are replaced by p (p.replaced_patterns). Note: we define the utility of singleton patterns as 0. Instead of making a new pattern with only a single item and storing pointers to it, we can directly store the item id in

each transaction and it will cost the same amount of memory as the original data.

The SPUR algorithm calls Algorithm 2 to initialize the ranking of pattern utilities. However, the ranking can change dynamically during the compression iterations. Existing approaches either use a static approximation (e.g. Krimp [11]) or if dynamic, need multiple passes of the data. Hence, they do not fit in the setting of summarizing data streams. Next, we will show how our utility function can be dynamically and efficiently updated without accessing the original data.

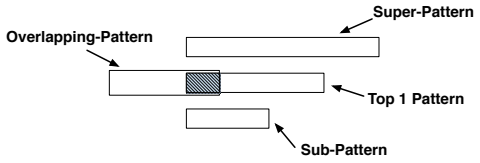


Figure 4: Dynamic adjustment of pattern utility

Compression with Dynamic Ranking Adjustment There are three categories of patterns whose utility values will be affected by a top 1 pattern p: p's super-patterns, sub-patterns and overlapping patterns (Figure 4). Algorithm 3 elaborates how we penalize the utilities of these three types of patterns when p is selected. For an affected pattern p_a, we first find the transactions in the intersection of p_a's and p's coverage sets (line 2). The effective coverage area of p_a will be changed by including p in these transactions, because the common items of p and p_a are already covered by p. Based on the type of p_a, we can penalize its utility value by the total area covered by p already. For a super-pattern p_{super}, the area covered by p is $l(p)$ per transaction (line 4) and $l(p_{overlap} \cap p)$ for a overlapping pattern $p_{overlap}$ (line 12). For a sub-pattern p_{sub}, the area covered by p is $l(p_{sub})$ per transaction. But since the transactions covered by p are not in p_{sub}'s coverage set any more, the space of the pointers to p_{sub} in those transactions are saved. So each transaction is penalized by $l(p_{sub}) - 1$ (line 6). We can see our algorithm only needs to deduct a value of area from a pattern's utility without scanning the items in the transactions.

Algorithm 3 UpdateRank(Q, p)

```
1:  for all p_a ∈ p's sub-/super-/overlapping set do
2:      covered_set ← p_a.coverage_set ∩ p.coverage_set;
3:      if p_a is p's super-pattern then
4:          p_a.utility ← p_a.utility - l(p) · |covered_set|;
5:      else if p_a is p's sub-pattern then
6:          p_a.utility ← p_a.utility - (l(p_a) − 1) · |covered_set|;
7:          p_a.coverage_set ← p_a.coverage_set \ covered_set;
8:          if p_a.coverage_set is empty then
9:              Q.remove(p_a); {p_a has been replaced by p.}
10:         end if
11:     else
12:         p_a.utility ← p_a.utility - l(p_a ∩ p) · |covered_set|;
13:     end if
14: end for
```

There are several implementation and performance related issues worth mentioned: First, our algorithm needs to find the sub-/super-/overlapping-pattern relationships among all patterns. However, once these relationships are established, they are reused by the many iterations of the SPUR algorithm. Second, the overlapping-patterns with p are found by taking the union of all super-patterns of p's sub-patterns. Third, we use a max heap to maintain the candidate pattern queue dynamically.

2.2 D-SPUR

We now present D-SPUR, the dynamic version of SPUR. In D-SPUR, we enhance and modify the pyramidal time window suggested by Aggarwal *at al.* [1] for clustering in data streams. Our enhancements center on the fact that we need to find an effective way to manage the stream of summary objects produced by SPUR while limiting the growth of memory footprint and reconstruction error (especially on recent and trending data). D-SPUR summarizes dynamic message streams by maintaining the pyramidal time window in Figure 5.

The input to the pyramidal time window is a stream of summary objects, each with memory size M. (right half of Figure 2). A level of the time window can hold two summary objects. Figure 5 demonstrates how the summary objects are inserted into the time window. Initially, the time window is empty, so Summary 1 and 2 can be directly inserted into Level 1 of the time window (Figure 5(a), (b) and (c)). When Summary 3 is ready, Level 1 of the time window is full (Figure 5(c)). We will *merge* the summary objects on the filled level, expand the time window with one more level, and insert the merged summary objects into the expanded level. In Figure 5(d), Summary 1 and 2 are merged and moved to Level 2, and Sumary 3 is inserted into Level 1. Similarly, Summary 4 is inserted into Level 1 (Figure 5(e)). When Summary 5 is ready, we first merge Summary 3 and 4 into a new summary object Summary 3-4 and insert it into Level 2 of the time window and place Summary 5 on Level 1 (Figure 5(f)). Note that the product of the merging operation (e.g. Summary 1-2 and Summary 3-4) must also fit in the constant memory budget M.

Using the pyramidal time window, the total memory footprint will grow logarithmically. More importantly, historical data is more compressed than recent data, so the summary is more accurate in recent time intervals. Next, we discuss the key operations of maintaining the time window: merging two summaries and maintaining time information.

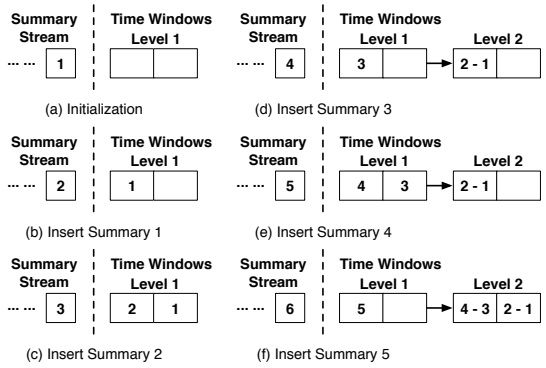


Figure 5: Maintaining pyramidal time window

Merging Two Summary Objects Given two summary objects S_1 and S_2, the corresponding patterns and the transactions represented by the patterns are $S_1 = (P_1, T_1)$ and $S_2 = (P_2, T_2)$. Both S_1 and S_2 are produced by the SPUR algorithm and can fit in a constant memory budget M. When merging S_1 and S_2 in the time window (Figure 5), we want to merge them into a new summary $S = (P, T)$ that can also fit in M. We first combine the pattern sets P_1 and P_2 to a new pattern set P' by removing the duplicate patterns, representing all the transactions in T_1 and T_2 with a unified

alphabet P'. Then we merge the transaction sets T_1 and T_2 into a new transaction set T' to get a new summary (P', T').

However, we cannot guarantee that (P', T') can fit in the memory budget M. More compression is needed to meet the budget. We rely on the output of SPUR to perform the compression. Remember the SPUR algorithm outputs a utility value for each pattern. This value measures the compression performance of a pattern. We use a greedy strategy to reduce the memory space of (P', T'). We sort the pattern utility values from low to high. We start with dropping the low utility patterns and removing them from the transactions they cover. This process will stop when the total size of the patterns and transactions left is below M. We output the final merged summary $S = (P, T)$ as our result. D-SPUR uses this merging algorithm to maintain the pyramidal time window in Figure 5. After merging two summaries, however, the time information of the messages is lost since we cannot distinguish the messages between two batches. We will address this problem next.

Maintaining Time It is necessary to maintain time information for the transactions in a summary, otherwise the summary cannot effectively answer a query about an arbitrary time interval. For example, if the state for the summary is as shown in Figure 6, and we want to retrieve messages in batches 3 to 6, the query cannot be answered if no time information is stored with each transaction as the messages in batches 1 to 4 and 5 to 8 are all merged together.

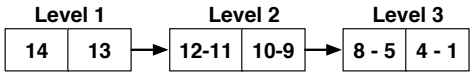


Figure 6: An example of the pyramidal time window

When we compress individual batches to summaries with SPUR, the transactions in the summary are represented by the patterns selected by SPUR. It is likely that two similar but non-identical transactions will be represented by the same set of patterns and they will essentially be identical in the summary. Instead of storing them separately, we store distinct transactions in the summary and associate a count with each transaction to indicate how many times a transaction appeared in a batch.

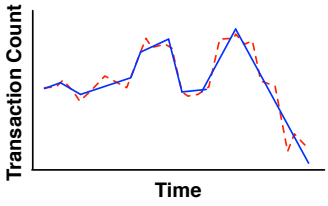


Figure 7: An example of the time series of a transaction

When D-SPUR merges summaries of two adjacent batches, the merging operation combines their transaction sets. If two transactions contain the same set of patterns, they must be from two different batches, because within a batch, we only keep distinct transactions. Instead of summing the count of these two transactions, we could concatenate their counts in time order and form a time series with two points. As D-SPUR combines more summaries, we concatenate more points to each transaction. A time series that spans batches (i.e. the red dashed line in Figure 7) is therefore formed for each transaction, enabling reconstruction of the exact count in any time interval.

However, this time series will grow linearly with time, which will violate the memory budget constraint. We therefore approximate (compress) the time series using a constant number of linear regression lines (blue line in Figure 7). Whenever the number of regression lines exceed a constant k after merging summary objects, D-SPUR will compress the two series for all transactions. We employ methods proposed by Palpanas *et al.* [9] for the purpose of compressing the time series. Their methods can find the optimal approximation when concatenating two adjacent time series.

3. LIVE ANALYTICS WITH TED-SPUR

Next, we present the query processing component of our framework, an analytical tool *TED-SPUR* (Topic and Event detection with D-SPUR), to support complex queries on dynamic data. One typical example of analytical query on text data is to discover topics [3, 10, 12]. Under the dynamic setting of message streams, it is also important to model the evolutionary behaviors [2] of topics. With the in-memory summary produced by D-SPUR, we can approximately reconstruct the original messages to perform those two tasks on arbitrary time interval(s).

We employ an implementation of non-negative matrix factorization (NMF) algorithm [12] to find the topics from the tweets within a query time interval, as NMF has been shown to handle sparse input well. The main idea is to factor a $d \times w$ *document-word* matrix into two non-negative matrices, the first of which a $d \times k$ *document-topic* matrix and the second a $k \times w$ *topic-word* matrix.

Another interesting task of analyzing tweets stream is to discover the evolution of the topics. For this task, we define a series of events to capture the *appearance/disappearance* of a topic, the *growth/ shrinkage/ continuation* of topic popularity and the *merging/ split/ transformation* of topic content. Given two time intervals I_1 and I_2, assume I_2 is after but not necessarily succeeding I_1. Let the mined topic sets from both intervals be T_1 and T_2 respectively. Each topic is further represented as a word distribution z and a support value *sup* indicating how many times it appears in the time interval. We use asymmetric KL-divergence [5] to capture the difference between two topics with regard to their content. The formalization of topic events is as follows:

Appearance: A topic z *appears* in T_2 iff there is no topic z' in T_1 such that $D_{KL}(z, z') < \gamma$, where parameter γ measures the closeness of two topics. The intuition here is that if we cannot find a topic z' in T_1 which is close enough to topic z, we will consider topic z as a novel topic in T_2.

Disappearance: A topic z *disappears* in T_1 iff there is no topic z' in T_2 such that $D_{KL}(z, z') < \gamma$.

Growth: For two topics (z, sup) in T_2 and (z', sup') in T_1, if $D_{KL}(z, z') \leq \delta$, and $sup/sup' \geq 1 + \epsilon$ where $0 < \delta < \gamma$ and $\epsilon > 0$, then topic z *grows* from topic z'. To explain, we find a topic z in T_2 whose content is similar enough to topic z' in T_1 while it is also more frequent than z'. Here the similarity threshold δ should be much lower than γ.

Shrinkage: For two topics (z, sup) in T_2 and (z', sup') in T_1, if $D_{KL}(z, z') \leq \delta$, and $sup/sup' \leq 1 - \epsilon$ where $0 < \delta < \gamma$ and $\epsilon > 0$, then topic z' *shrinks* to topic z.

Continuation: For two topics (z, sup) in T_2 and (z', sup') in T_1, if $D_{KL}(z, z') \leq \delta$, and $1 - \epsilon < sup/sup' < 1 + \epsilon$ where

$0 < \delta < \gamma$ and $\epsilon > 0$, then there is a *continuation* from z' to z. To put it informally, if we can find two topics in T_1 and T_2 with similar enough contents and their strengths are also similar, then we regard the topic in T_2 as the continuation of the topic in T_1.

Merging: Given a topic z in T_2, and a subset of topics $\{z'\}$ from T_1 such that each z' satisfies $\delta < D_{KL}(z, z') < \gamma$. For any pair of topics z'_1 and z'_2 in $\{z'\}$, let the weighted sum of distribution be $Z = \frac{z'_1 \times sup'_1 + z'_2 \times sup'_2}{sup'_1 + sup'_2}$. If $D_{KL}(z, Z) \leq \delta$, then we say z is *merged* from z'_1 and z'_2. The *merging* event deals with topics which can neither be classified as *growth/ shrinkage/ continuation* from a previous topic, nor a newly appeared topic. The explanation is that if the content of a topic is similar enough to that of the weighted combination of two topics from previous time interval, then we consider the new topic to be *merged* from the previous two topics. Note that usually the resultant z's from NMF are divergent from each other, making it unlikely that multiple pairs in $\{z'\}$ merge into the same z.

Split: Given a topic z in T_1, and a set of topics $\{z'\}$ from T_2, such that z' satisfies $\delta < D_{KL}(z', z) < \gamma$. For any pair of topics z'_1 and z'_2 in $\{z'\}$ with the same definition of Z as above, if $D_{KL}(Z, z) \leq \delta$, then z is *split* into z'_1 and z'_2.

Transformation: If a topic z in T_2 is not involved in any of the seven aforementioned events, z is said to be *transformed* from topics $\{z'\}$ in T_1, such that each z' satisfies $\delta < D_{KL}(z, z') < \gamma$. The underlying intuition is that if a topic can find topics in the previous time interval which are partly similar to it and are not involved in any other event, then there is a content transformation to this topic.

By issuing multiple topic modeling queries over subsequent time windows and applying the above event detection algorithms, one can find evolutionary events over time.

4. EXPERIMENTAL RESULTS

Next, we present results for an extensive set of experiments we conducted to evaluate our stream summarization methods and advanced query processing capabilities. We considered several other algorithms as candidates for our baseline. Methods that use a probabilistic models to summarize patterns [15] are not efficient enough to handle the data influx rate we would like to handle. We therefore omit comparisons with these methods. We compare *SPUR* with the following algorithms: a) CDB [16] that uses rectangles to cover a transactional database b) RPMine [17] that tries to cluster the patterns and use the cluster centers to cover the remaining patterns. c) Krimp [11] that generates a static ranking of all patterns first and then summarizes the data using the top ranked patterns. Neither CDB nor RPMine can compress streaming data. Therefore, the *D-SPUR* algorithm is compared only with StreamKrimp [14], the streaming version of Krimp.

4.1 Dataset and Setup

We gathered 2100 hours of Twitter message streams from June to September in 2010[2]. The total data size is 5.9GB. For our evaluation, we partition the message stream into 1-hour batches. There are 100,000 messages per batch and 8 words per message on average after stop word removal and word stemming.

[2]As provided by Twitter, it is a 15% random sample of all messages.

All experiments were performed on a desktop machine with dual boot Red Hat Linux 6 and Windows 7 operating systems. The machine is equipped with an Intel i7 3.4GHz CPU and 16GB of main memory. Except for Krimp and StreamKrimp for which only Windows binaries are available, all algorithms were executed under Linux. All algorithms were implemented in C++.

4.2 Batch Compression With SPUR

In this section we present performance results for compressing a batch of messages to a summary object. Note that we only present results for batch summarization and not for the summary merging procedure. The SPUR algorithm is compared with the three baseline algorithms (**CDB, RPMine and Krimp**). The windows executable for Krimp and the source code for CDB and RPMine were obtained from the authors' websites. As the baseline algorithms cannot compress to a target memory budget, we relax the memory budget constraint in our algorithm by summarizing a batch until the utility of the top pattern is negative. We present results for processing the first 100 batches since they are sufficiently representative of the entire data. Our comparative study focuses on three aspects: execution time, false positive rate (i.e. compression quality) and compression ratio.

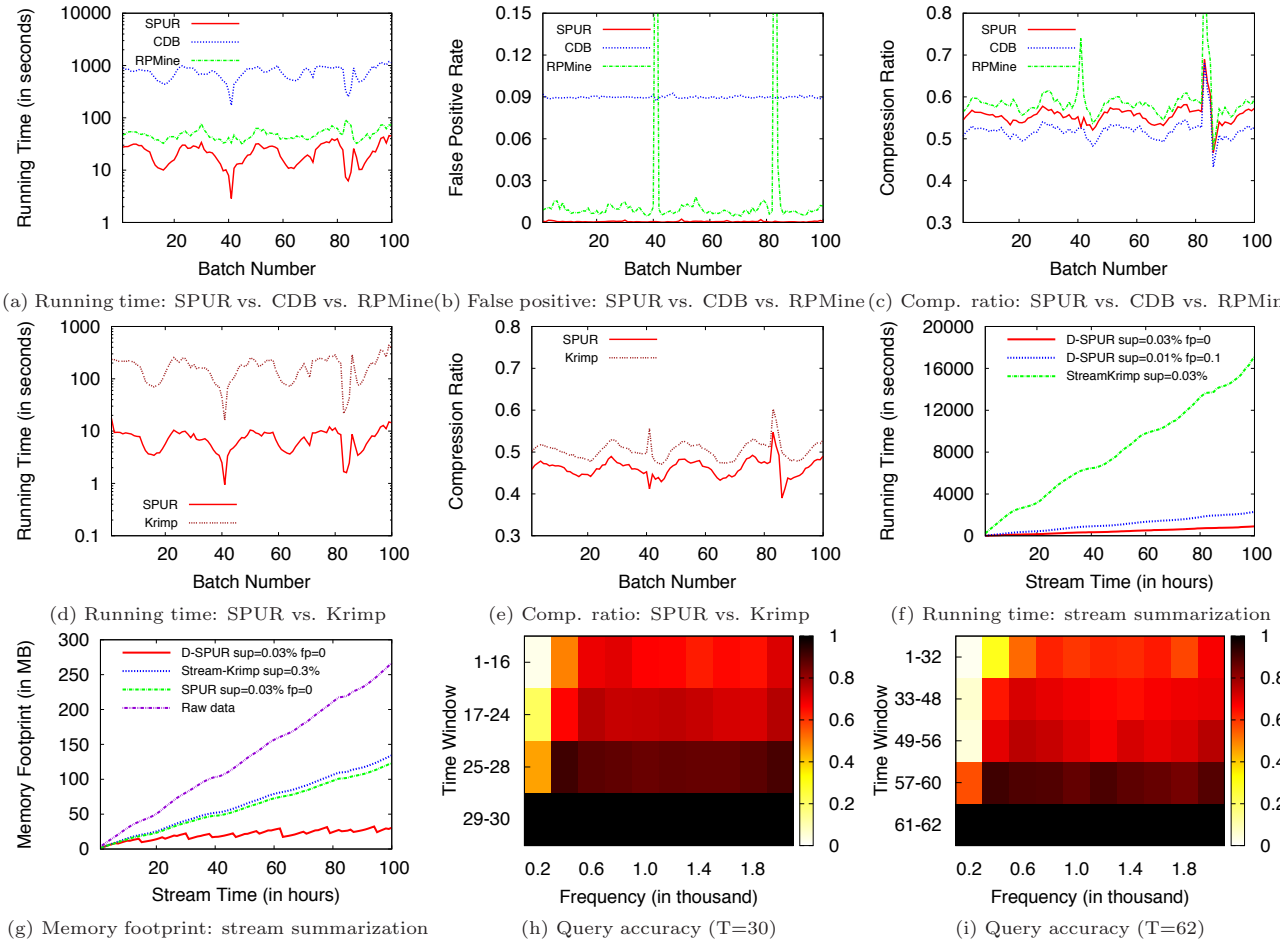
4.2.1 Comparison with CDB and RPMine

We first compare SPUR with CDB and RPMine as they all support the trade off between false positive rate and compression ratio. Support level is set to 0.01% for all three methods. We set a low support level because we want the summary to cover as many topical words as possible. We set the maximum false positive rate for SPUR and CDB to 0.1. For RPMine, the pattern cluster tightness parameter is also set to 0.1.

Figure 8a presents the running times for the three methods in log scale. It is easy to see that our method is significantly more efficient – it is at least one order of magnitude faster than RPMine, and two orders of magnitude faster than CDB. SPUR is able to process an hours worth of data in less than one minute, lending itself to our requirement of being able to process the stream in real-time.

We next evaluate the algorithms' false positive rate. Ideally, we want to introduce as few false positives as possible when reducing the data size. We therefore measure the *actual* false positive rate each method exhibits when given the same parameters for support threshold and maximum false positive rate. Figure 8b presents the false positive rate for each batch of the stream. Again, the performance of SPUR is solid – its false positive rate never exceeds 0.005. In contrast, CDB constantly suffers from a higher false positive rate, which is close to the maximum specified value. We also find that RPMine is susceptible to changes in data size, as suggested by the two spikes at batches 41 and 83. With relatively small data size for both batches, false positive rates reach 0.365 and 0.399 respectively.

As for the compression ratio (the lower the better), our method performs midway between CDB and RPMine with a range of 50% to 60% (see Figure 8c). Again note the relatively high compression ratio for RPMine for batches 41 and 83. We conjecture that RPMine tends to cluster singleton patterns with longer patterns for these cases, leading to less reduction in size and a higher false positive rate. Note



(a) Running time: SPUR vs. CDB vs. RPMine (b) False positive: SPUR vs. CDB vs. RPMine (c) Comp. ratio: SPUR vs. CDB vs. RPMine

(d) Running time: SPUR vs. Krimp (e) Comp. ratio: SPUR vs. Krimp (f) Running time: stream summarization

(g) Memory footprint: stream summarization (h) Query accuracy (T=30) (i) Query accuracy (T=62)

Figure 8: Experiment results for batch compression and stream summarization

that the compression quality for our method is more robust against changes in data size, which is a favorable property.

All algorithms achieve exactly the same false negative rate as this only depends on the support level that is set to be same for all algorithms.

4.2.2 *Comparison with Krimp*

For this set of experiments, we compare Krimp with SPUR. We compare KRIMP separately from CDB and RPMine for two reasons. First, it does not allow for false positives during compression. Second, its running time is very large when support level is as low as 0.01%. Therefore, we set the support level to 0.03% in the following set of experiments. For our method, we set the maximum false positive rate to 0.0.

The running times and compression ratios for the two algorithms are shown in 8d and 8e. Our method is always at least 10 times faster than Krimp, and it also bests Krimp in terms of compression ratio.

4.3 Stream Summarization with D-SPUR

In this section, we present the performance of the D-SPUR algorithm on summarizing the Twitter message stream. There is no streaming version for CDB and RPMine, thus we only compared D-SPUR with StreamKrimp at support level = 0.03% and false positive rate = 0. For D-SPUR, the memory budget M is set to 1.2 MB as we find it yields low false negative error rate in the SPUR summarization step. We use 5 linear segments to approximate the time series of the transactions. This configuration leads to a compression ratio of approximately 50% for each batch. We evaluate the algorithms in three aspects: execution time, memory footprint, and quality of the summary.

Execution time: Figure 8f shows the running times for D-SPUR and StreamKrimp. The x-axis represents the batch number and the y-axis shows the time different stream summarization algorithms take to summarize the input stream. We can see D-SPUR is much faster than StreamKrimp at the support level of 0.03%. Even at a lower support level of 0.01%, D-SPUR only takes approximately 40 minutes to summarize 100 hours worth of Twitter messages. Our method can thus process the data faster than it arrives, which is desirable for real-time stream processing.

Memory footprint: As for memory consumption, we plot the storage size of the raw stream together with the memory footprint for different summarization algorithms in Figure 8g. We can see that the memory footprint for D-SPUR grows very slowly – asymptotically it grows logarithmically in the size of the input. Furthermore, the size of D-SPUR's summary is 8 times smaller than the raw data and 3 times

smaller than the summary produced by StreamKrimp. This is because the merging of two summaries in D-SPUR is based on the pyramidal time window. If one does not use the pyramidal summarization scheme, the memory footprint grows linearly (see the green line in Figure 8g). Also note that StreamKrimp has an even higher memory footprint than our SPUR algorithm. When a new batch of transactions streams in, the StreamKrimp algorithm first tries to compress the transactions with the codetable used for the previous batches. The old codetable is not always suitable for compressing the new batch and often it is forced to rebuild a new codetable for the new batch. In effect, it ends up maintaining separate summaries for individual batches.

Quality: We evaluate the summary quality by querying the summary with a randomly generated query workload. The query is in the format of a keyword w and a timestamp t. w and t are sampled uniformly at random from all words above the support threshold and all time stamps. We do not consider infrequent words because they are dropped by the frequent pattern mining algorithm and can never be retrieved by any frequent pattern based summarization algorithms. All messages containing keyword w at time t are reconstructed and returned as the query result. We represent the result as a multiset of words, and compute the Jaccard similarity between the query result and the original messages that contain w at time t to measure query accuracy. We expect the query accuracy to be high for both *recent* data and *frequent* keywords.

We present the distribution of query accuracy over time and word frequencies. Figure 8i presents the distribution of query accuracy over time and word frequencies where system time T is 62. The y-axis represents the query time stamps aligned with the pyramidal time window at T and x-axis represents the frequency of the query keywords. The color represents the accuracy at a specific time and word frequency, the darker the better. We can see on recent data, e.g. T = 61 – 62, the accuracy is high on all words; on historical data, e.g. T = 1 – 32, the accuracy is higher on frequent words than infrequent words; the overall accuracy on the recent data is higher than the historical data; and the accuracy on frequent words is generally higher than infrequent words.

Figure 8h shows the distribution of query accuracy when we rewind the system time T to 30. We observe the same trend as the accuracy on recent data for frequent words is higher than that on historical data for infrequent words. Compared with the summary at system time 62 (Figure 8i), we can see the overall query accuracy for batch 1 to 30 is higher, except for the infrequent words for batch 1 to 16. At system time 62, our pyramidal time window compresses the historical data (batch 1 to 30) more, so query accuracy on those batches is low. But at system time 30, the summary can keep more information and is more accurate.

4.4 Analytical Query with TED-SPUR

In this section, we present qualitative evaluations of our algorithm by presenting experimental results for answering topic modeling and event detection queries with our summary on the Twitter data. For these experiments, we apply D-SPUR on subsets of the 2100-hour dataset to answer topic modeling and event detection queries. The data is divided into 6-hour batches with approximately 700K messages per batch. Support threshold is set to 0.01%, false positive rate to 0.1 and M to 20 MB. For the topic modeling algorithm, we use $k = 50$ as the number of topics.

Event Type	Topic ID
Appear	21, 23, 24, 25, 28, 29, 30
Disappear	13, 14, 15, 18, 19, 20
Continue	12 → 22
Merge	(11, 16) → 26
Transform	17 → 27

Table 1: Event detection from the topics in Table 2b and 2c

Topic modeling: We build a summary of data from June 11th to June 26th. All messages generated between 5pm, June 12th and 5pm, June 13th and match the query "world cup" are retrieved from the summary. This time interval is the second day after the World Cup in South Africa began. The top 10 topics returned by NMF are listed in Table 2a, with each topic represented by the top 5 words of the topic's word distribution. There are several topics detected from our summary that are very related to the keywords "world cup". For example, "south africa" indicates the country where the World Cup was held; "vuvuzela" is a handy horn used by South African fans in the World Cup stadiums, and there was a popular debate about whether "vuvuzela" should be banned, because it made an annoying noise. Note that the query time interval is also one day before the game between England and USA. The topic modeling algorithm detected a topic mentioning this game with the keywords "ENG" and "USA". The detected topics show that our summary can approximately reconstruct the messages to support topic modeling queries.

Event Detection: Next, we give an example of an event detection query. A summary is built for data from June 26th to July 11th, and the "world cup" query is run on two time intervals: 5pm, July 3rd to 5pm, July 4th; 5pm, July 4th to 5pm July 5th. We first perform topic modeling on the retrieved messages. The resultant topics are listed in Table 2b and 2c. Note that the first time interval is the day after the quarter-final game between Netherlands and Brazil, and the day before the quarter-final game of Argentina against Germany. From Table 2b, we can see some related keywords of these two games, for example, *NED*, *BRA*, *ARG* and *GER* are abbreviations for the teams, and *Maradona* is the coach of Argentina, *Messi* and *Klose* are the players of Argentina and Germany respectively. In this time interval, we detected topics which discuss the NED vs. BRA game on the day before and predict the ARG vs. GER game. Similarly, the second time interval is the day after Germany beat ARG in the quarter-final and before the semifinal began. Table 2c lists the topics in the second time interval. We also detect topics which discuss previous day's ARG vs. GER game, such as the keywords ARG, GER, Messi and Klose. There are also messages which predict that Germany can enter the final because they defeated Argentina, and messages that predict Spain (ESP) can enter the semi-final.

Next, we apply the event detection algorithms introduced in Section 3 on the topics of the two time intervals. We set the parameters in the algorithm to $\delta = 0.5$, $\gamma = 0.5$ and $\epsilon = 0.3$. The detected events are presented in Table 1. The numbers in the table are topic ids in Table 2b and 2c. Among the events, the newly emerged topics about predicting the semifinal and final games are detected as appear; the topic about the NED vs. BRA game in the day before is categorized as disappear; two same topics about live TV coverage from the two time intervals are detected as continue event; the two topics about the ARG vs. GER game in the first

ID	Words
1	***south***, ***africa***, group, play, now
2	nowplay, yeah, people, twitter, follow
3	watch, big, play, fan, go
4	match, play, win, team, group
5	group, now, ***ENG***, go, ***USA***
6	live, home, real, watch, game
7	***vuvuzela***, watch, annoy, people, game
8	twitter, people, please, org, bit
9	***vuvuzela***, stadium, health, people, ***fifa***
10	bit, photo, show, people, match

(a) 5pm Jun 12th to 5pm Jun 13th.

ID	Words
11	***Messi***, ***German***, today, ***Klose***, goal
12	watch, game, show, tv, live
13	***Maradona***, start, say, people, match
14	fox, soccer, kickoff, down, ***ARG***
15	fan, tv, espn, score, twitter
16	***ARG***, ***GER***, match, goal, ***Klose***
17	***ARG***, out, match, wow, go
18	thank, god, please, today, give
19	tweetphoto, party, nice, look, wow
20	***NED***, ***BRA***, match, play, fly

(b) 5pm Jul 3rd to 5pm Jul 4th.

ID	Words
21	***semifinal***, tv, match, ***ESP***, go
22	watch, game, show, tv, live
23	***german***, ***final***, ***Klose***, play, wow
24	www, home, look, win, play
25	tinyurl, home, play, game, fan
26	***ARG***, ***GER***, ***Messi***, goal, ***Klose***
27	***ARG***, die, game, team, play
28	auction, dasar, demi, goal, otw
29	www, live, show, please, home
30	haha, yeah, watch, gua, cool

(c) 5pm Jul 4th to 5pm Jul 5th.

Table 2: Topics related to "world cup"

time interval are merged to a single topic in the second time interval after Germany defeated Argentina; finally, a thread of topic about Argentina is transformed to a similar topic in the second time interval. We can see that our event detection algorithm can detect evolutionary events which are aligned with events in the real world. These events can help users understand the dynamics of the topics in Twitter messages.

5. RELATED WORK

Since the content generated by social media is available as a stream of timestamped messages, dynamic studies have been conducted to track topics on the stream [7, 6], and find temporal patterns [18]. However, all of these studies assume unlimited amount of memory to store the data. We are interested in summarizing the data stream with a limited memory budget and serving new analytical tasks with the summary. Existing work can also benefit from the summary from approximately reconstructing the original data by querying the summary.

We treat each social media post as a transaction of words and use frequent patterns to summarize. The CDB [16] algorithm uses rectangles to cover a transactional database; RPMine [17] first tries to cluster the patterns and use the cluster centers to cover the remaining patterns. Methods that use a probabilistic model of the patterns [15] are slow and not capable of processing streams that arrive at a fast pace. All the above algorithms do not support mining on data streams. Krimp [11] follows the MDL principle to summarize the data with codetables, which is essentially a static ranking of patterns. StreamKrimp [14], the streaming version of Krimp, can dynamically adjust the codetable for streaming data. Traditional methods of mining frequent patterns on streams [4, 8] focus on either counting item or pattern frequencies rather than summarizing the data.

6. CONCLUSIONS

We proposed an efficient stream summarization framework, which can incrementally build summaries of Twitter message streams with one-pass over the data. We developed a novel algorithm to compress twitter messages with low compression ratio, high quality and fast running time. The memory footprint of our stream summarization algorithm grows approximately logarithmically with time. Our summary allows one to issue queries to retrieve messages over arbitrary time intervals. The original messages can be approximately reconstructed to support topic modeling algorithms. We also defined a suite of events on topics from two time intervals. An event detection algorithm is proposed to find evolutionary events between two time intervals.

Acknowledgments: This material is based upon work supported by the National Science Foundation under Grant No. SoCS-1111118 and IIS-0917070. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

7. REFERENCES

[1] C. Aggarwal, J. Han, J. Wang, and P. Yu. A framework for clustering evolving data streams. In *VLDB '03*.

[2] S. Asur, S. Parthasarathy, and D. Ucar. An event-based framework for characterizing the evolutionary behavior of interaction graphs. In *KDD '07*.

[3] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, 2003.

[4] T. Calders, N. Dexters, and B. Goethals. Mining frequent items in a stream using flexible windows. *Intell. Data Anal.*, 12(3):293–304, 2008.

[5] Kullback, S. and Leibler, R. A. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[6] J. Leskovec, L. Backstrom, and J. Kleinberg. Meme-tracking and the dynamics of the news cycle. In *KDD '09*.

[7] J. Lin, R. Snow, and W. Morgan. Smoothing techniques for adaptive online language models: topic tracking in tweet streams. In *KDD '11*.

[8] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB '02*.

[9] T. Palpanas, M. Vlachos, E. Keogh, D. Gunopulos, and W. Truppel. Online amnesic approximation of streaming time series. In *ICDE '04*.

[10] D. Ramage, S. Dumais, and D. Liebling. Characterizing microblogs with topic models. In *ICWSM*, 2010.

[11] A. Siebes, J. Vreeken, and M. van Leeuwen. Item sets that compress. In *SDM '06*.

[12] V. Sindhwani, A. Ghoting, E. Ting, and R. Lawrence. Extracting insights from social media with large-scale matrix approximations. *IBM J. Res. Dev.*, 55:527–539, Sept. 2011.

[13] T. Uno, M. Kiyomi, and H. Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *ICDM '04 FIMI Workshop*, 2004.

[14] M. van Leeuwen and A. Siebes. Streamkrimp: Detecting change in data streams. In *ECML/PKDD '08*.

[15] C. Wang and S. Parthasarathy. Summarizing itemset patterns using probabilistic models. In *KDD '06*.

[16] Y. Xiang, R. Jin, D. Fuhry, and F. F. Dragan. Succinct summarization of transactional databases: an overlapped hyperrectangle scheme. In *KDD '08*.

[17] D. Xin, J. Han, X. Yan, and H. Cheng. Mining compressed frequent-pattern sets. In *VLDB '05*.

[18] J. Yang and J. Leskovec. Patterns of temporal variation in online media. In *WSDM '11*.