Noname manuscript No.
(will be inserted by the editor)

Mining Compressing Sequential Patterns in Streams

Hoang Thanh Lam[♯] · **Toon Calders**[♯] ·
Jie Yang[♯] · **Fabian Mörchen**[§]
and Dmitriy Fradkin[♭]

Received: date / Accepted: date

Abstract Mining patterns that compress the data well was shown to be an effective approach for extracting meaningful patterns and solving the redundancy issue in frequent pattern mining. Most of the existing work in the literature however, consider mining compressing patterns from a static database of itemsets or sequences. These approaches require multiple passes through the data and do not scale up with the size of data streams. In this paper, we study the problem of mining compressing sequential patterns from a data stream. We propose an approximate algorithm that needs only a single pass through the data and efficiently extracts a meaningful and non-redundant set of sequential patterns. Experiments on three synthetic and three real-world large-scale datasets show that our approach extracts meaningful compressing patterns with similar quality to the state-of-the-art multi-pass algorithms proposed for static databases of sequences. Moreover, our approach scales linearly with the size of data streams while all the existing algorithms do not.

Keywords Data Stream · Compressing pattern · Complexity

1 Introduction

Descriptive pattern mining aims at finding important structures hidden in data and providing a concise summary of important patterns or events in data. It answers questions posed by users during a data exploration process

[♯] Department of Mathematics and Computer Science
TU Eindhoven, Eindhoven, the Netherlands
E-mail: [t.l.hoang, t.calders,]@tue.nl, j.yang.1@student.tue.nl
[§] Amazon.com Inc
Seattle, WA, USA
E-mail: moerchen@amazon.com
[♭] Siemens Corporate Research
A division of Siemens Corporation in Princeton, USA
E-mail: dmitriy.fradkin@siemens.com

Pattern	Support	Pattern	Support
algorithm algorithm	0.376	method method	0.250
learn learn	0.362	algorithm result	0.247
learn algorithm	0.356	Data set	0.244
algorithm learn	0.288	learn learn learn	0.241
data data	0.284	learn problem	0.239
learn data	0.263	learn method	0.229
model model	0.260	algorithm data	0.229
problem problem	0.258	learn set	0.228
learn result	0.255	problem learn	0.227
problem algorithm	0.251	algorithm algorithm algorithm	0.222

Fig. 1 The set of the most frequent closed patterns in the abstracts of the Journal of Machine Learning Research articles. The set is very redundant and contains a lot of uninteresting patterns.

such as "what are the interesting patterns in the data?" and "what do these interesting patterns look like and how are they related?". The answers to these questions help people gain insights into the properties of the data, which in turn enables them to make business decisions. For example, given a stream of tweets on Twitter[1], people may be interested in what hot topics Twitter users are talking about at a given moment. Or given a stream of queries issued by users of a search engine, one may be interested in what kind of information people are looking for at a given time.

Mining frequent patterns is an important research topic in data mining. It has been shown that frequent pattern mining helps finding interesting association rules, or can be useful for classification and clustering tasks when the extracted patterns are used as features [1,2]. However, in descriptive data mining the pattern frequency is not a reliable measure. In fact, it is often the case that highly frequent patterns are just a combination of very frequent yet independent items. For example, in [3,4] we showed that the set of frequent sequential patterns extracted from a set of 787 abstracts of the Journal of Machine Learning Research articles contained many sequences that were multiple repetitions of the same frequent word such as "algorithm", "machine", "learn" etc. (see Figure 1 for an example). These uninteresting patterns do not provide any further insight into the important structures of the data given prior knowledge about individual item frequencies. Moreover, due to the anti-monotonicity of the frequency measure, if an itemset is frequent, all of its subsets are frequent as well which leads to the redundancy issue in the set of frequent patterns [5–7].

There are many approaches that address the aforementioned issues in the literature. One of the most successful approaches is based on data compression which looks for the set of patterns that compresses the data most. The main idea is based on the *Minimum Description Length Principle* (MDL) [18] stating that the best model describing data is the one that together with the description of the model, it compresses the data most. The MDL principle has

[1] www.twitter.com

been successfully applied to solve the redundancy issue in pattern mining and to return meaningful patterns [7,19].

So far most of the work focussed on mining compressing patterns from static datasets or from modestly-sized data. In practice, however databases are often very large. In some applications, data instances arrive continuously with high speed in a streaming fashion. In both cases, the algorithm must ideally scale linearly with the size of the data and be able to quickly handle fast data stream updates. In the streaming case, the main challenge is that whole data cannot be kept in memory and hence the algorithm has to be single-pass. None of the approaches described in the literature scales up to the arbitrarily large data or obey the single-pass constraint.

In this work, we study the problem of mining compressing patterns in a data stream where events arrive in batches for instance like stream of tweets. We first introduce a novel encoding scheme that encodes sequence data with the help of patterns. Different from the encodings being used in recent work [3, 4,20], the new encoding is online which enables us to design online algorithms for efficiently mining compressing patterns from a data stream. We prove that there is a simple algorithm using the proposed online encoding scheme and achieving a near optimal compression ratio for data streams generated by an independent and identical distributed source, i.e. the same assumption that guarantees the optimality of the *Huffman* encoding in the offline case [22].

Subsequently, we formulate the problem of mining compressing patterns from a data stream. Generally, the data compression problem is NP-complete [24]. Under the streaming context with the additional single pass constraint, we propose a heuristic algorithm to solve the problem. The proposed algorithm scales linearly with the size of data. In the experiments with three synthetic and three real-life large-scale datasets, the proposed algorithm was able to extract meaningful patterns from the streams while being much more scalable than the-state-of-the-art algorithms.

This paper is organized as follows. Section 2 discusses related work including two state-of-the-art algorithms we will compare with. An encoding scheme of sequence data with the help of a set of patterns is described in section 3. The problem formulation is introduced in section 4. The algorithm and the analysis are described in section 5 and 6. Finally, section 7 demonstrates the effectiveness of the proposed approach in an extensive experimental study before the conclusions and future work part introduced in section 8.

2 Related work

We classify the existing work into three main categories and discuss each of them in the following subsections.

2.1 Concise summary of the set of frequent patterns:

Mining closed patterns is one of the first approaches [5] addressing the redundancy issue in pattern mining. A frequent pattern is closed if there is no frequent super-pattern having the same frequency. The set of frequent closed patterns typically has much lower cardinality than the set of all frequent patterns.

Alternatively, another approach finding a concise representation of the set of frequent patterns is proposed by mining all non-derivable patterns [8]. It finds a concise set of patterns such that together with the support frequency information the entire set of frequent patterns can be derived.

The aforementioned methods are lossless approaches in which the set of all frequent patterns can be reconstructed from its concise representation. Alternative approaches were proposed to find a lossy concise representation of the set of frequent patterns by mining maximal frequent patterns [10]. Belonging to this category also include other lossy methods using clustering [9] or condense the pattern set in post-processing [11].

Approaches of this type were shown to be very effective in reducing the size of the pattern set. However, in the worst case, the pattern set can still be exponential in the number of items. Moreover, the set of frequent closed patterns may still contain many uninteresting patterns that are combinations of frequent items that are independent of each other [7].

2.2 Hypothesis testing based approaches

Hypothesis testing based approaches first assume that data are generated by a null model. The observed pattern frequency in the data is compared to the expected frequency of the pattern given the assumption that the data are generated by the null model. If the observed frequency significantly deviates from the expectation of the pattern frequency in the null model, the pattern is considered statistically significant. This approach helps to remove uninteresting patterns that are explained by the null model alone.

Swap randomization [6] was one of the first hypothesis testing based approaches proposed for testing the significance of data mining results. Similar approaches are also proposed for sequence data in which Markov models are usually chosen as a null model [12,26]. In the field of significant subgraph mining, the idea of using hypothesis testing to filter out insignificant frequent subgraphs were proposed in which the null model is similar to swap randomization for generating random graphs preserving the degree distribution [13].

Depending on the expectation of data miners about the type of patterns, different null models are chosen. In the case when there is no particular preference about a specific null model, the maximum entropy model with constraints on pattern frequency can be used as a null model [16,14,15]. Beside assuming a fixed null model, in [17], the authors proposed an approach iteratively updating the null model and succinctly building the set of patterns to summarize the pattern sets effectively.

The hypothesis testing based approaches were shown to be very effective in filtering out uninteresting patterns. Especially, the iterative mining approach was also able to solve the redundancy issue. These approaches on the one hand provide us with a flexibility through explicit choice of the null models to only retain the patterns that we consider as unexpected. However, on the other hand, the hypothesis testing based approaches have a disadvantage that the significance of patterns is a subjective score with respect to a given null model. Therefore, the mining results are highly dependent on the choice of the null model. In many cases, choosing the right null model is not a trivial task.

2.3 Minimum description length approaches

In the MDL-based approaches, an encoding scheme is defined to compress data by a set of patterns. The choice of encodings implicitly defines a probability distribution on the data. This is in contrast to hypothesis testing based approaches in which the null models are chosen explicitly. According to the MDL principle [18] the set of most compacted patterns that compresses the data most is considered as the best set of patterns. This approach was shown to be very effective in solving the redundancy issue in pattern mining [7].

The SubDue system [19] is the first work exploiting the MDL principle for mining a non-redundant set of frequent subgraphs. In the field of frequent itemset mining, the well-known Krimp algorithm [7] was shown to be very good at solving the redundancy issue and at finding meaningful patterns.

The MDL principle was first applied for mining compressing patterns in sequence data in [3,4] and in [20]. The GoKrimp algorithm in the former work solved the redundancy issue effectively. However, the first version of the GoKrimp algorithm [3] used an encoding scheme that does not punish large gaps between events in a pattern. In an extended version of the GoKrimp algorithm [4] this issue is solved by introducing gaps into the encoding scheme based on Elias codes. Besides, a dependency test technique is proposed to filter out meaningless patterns. Meanwhile, in the latter work the SQS algorithm proposed a clever encoding scheme punishing large gaps. In doing so, the SQS was able to solve the redundancy issue effectively. At the same time it was able to return meaningful patterns based solely on the MDL principle.

However, a disadvantage of the encoding defined by the SQS algorithm is that it does not allow encoding of overlapping patterns. Situations where patterns in sequences overlap are common in practice, e.g. message logs produced by different independent components of a machine, network logs through a router etc. Moreover, neither the GoKrimp algorithm nor the SQS algorithm were intended for mining compressing patterns in data streams. The encodings proposed for these algorithms are *offline encodings*. Under the streaming context, an offline encoding does not work because of the following reasons:

1. Complete usage information is not available at the moment of encoding because we don't know the incoming part of the stream
2. When the data size becomes large, the dictionary size usually grows indefinitely beyond the memory limit. Temporally, part of the dictionary must be evicted. In the latter steps, when an evicted word enters the dictionary again we loose the historical usage of the word completely.
3. Handling updates for the offline encoding is expensive. In fact, whenever the usage of the word is updated, all the words in the dictionary must be updated accordingly. On one hand, this operation is expensive, on the other hand, it is impossible to update the compression size correctly for the case that part of the dictionary has been evicted.

In contrast to these approaches, the Zips algorithm proposed in this work inherits the advantages of both state-of-the-art algorithms. It defines a new *online encoding* scheme that allows to encode overlapping patterns. It does not need any dependency test to filter out meaningless patterns and more importantly, under reasonable assumptions, it provably scales linearly with the size of the stream making it the first work in this topic being able to work efficiently on very large datasets.

Our work is tightly related to the *Lempel-Ziv*'s data compression algorithm [22]. However, since our goal is to mine interesting patterns instead of compression, the main differences between our algorithm and data compression algorithms are:

1. Data compression algorithms do not aim to a set of patterns because they only focus on data compression.
2. Encodings of data compression algorithms do not consider important patterns with gaps. The *Lempel-Ziv* compression algorithms only exploit repeated strings (consecutive subsequences) to compress the data while in descriptive data mining we are mostly interested in patterns interleaved with noise events and other patterns.

3 Data stream encoding

In this work, we assume that events in a data stream arrive in batches. This assumption covers broad types of data streams such as tweets, web-access sequences, search engine query logs, etc. This section discusses online encodings that compress a data stream by a set of patterns. For education reasons, we start with the simplest case where only singletons are used to encode the data. The generalized encoding for the case with non-singletons are described in the subsequent subsections.

3.1 Online encoding using singletons:

We discuss an online encoding that uses only singletons to compress the data. Since this encoding does not exploit any pattern for compress the data, we consider the representation of the data in this encoding as an uncompressed form of that data. Let $\sum = \{a_1, a_2, \cdots, a_n\}$ be an alphabet containing a set of characters a_i, the online data encoding problem can be formulated as follows:

Definition 1 (Online Data Encoding Problem) Let A denote a sender and let B denote a receiver. A and B communicate over some network, where sending information is expensive. A observes a data stream $S_t = b_1 b_2 \cdots b_t$. Upon observing a character b_t, A needs to compress the character and transmit it to the receiver B, who must be able to uncompress it. Since sending information on the network is expensive, the goal of A and B is to compress the stream as much as possible to save up the network bandwidth.

In the offline scenario, i.e. when S_t is finite and given in advance, one possible solution is to first calculate the frequency of every item a (denoted as $f(a)$) of the alphabet in the sequence S_t. Then assign each item a a codeword with length proportional to its entropy, i.e. $-\log f(a)$. It has been shown that when the stream is independent and identically distributed (i.i.d) this encoding, known as the *Huffman* code in the literature, is optimal [22]. However, in the streaming scenario, the frequency of every item a is unknown and the codeword of a must be assigned at the time a arrives and B must know that codeword to decode the compressed item.

We propose a simple solution for Problem 1 as follows. First, in our proposed encoding we need codewords for natural numbers. This work uses the *Elias Delta* code [21] denoted $E(n)$ to encode the number n. The Elias was chosen because it was provable as near optimal when the upper bound on n is unknown in advance. The length of the codeword $E(n)$ is $\lfloor \log_2 n \rfloor + 2\lfloor \log_2 (\lfloor \log_2 n \rfloor + 1) \rfloor + 1$ bits.

A first notifies B of the size of the alphabet by sending $E(|\sum|)$ to B. Then it sends B the dictionary containing all characters of the alphabet $\sum$ in the lexicographical order. Every character in the dictionary is encoded by a binary string with length $\lceil \log_2 |\sum| \rceil$. Finally, when a new character in the stream arrives A sends the codeword of the gap between the current and the most recent occurrence of the character. When B receives the codeword of the gap it decodes the gap and uses that information to refer to the most recent occurrence of the character which has been already decoded in the previous step. Since the given encoding uses a reference to the most recent occurrence of a word to encode its current occurrence we call this encoding the *reference encoding* scheme. We call the sequence of encoded gaps sent by A to B the *reference stream.*

Example 1 Figure 2 shows an example of a reference encoding scheme. A first sends B the alphabet in lexicographical order. When each item of the stream arrives A sends B the codeword of the gap to its most recent occurrence. For instance, A sends $E(3)$ to encode the first occurrence of b and sends $E(1)$ to encode the next occurrence of b. The complete reference stream that A sends B is $E(3)E(1)E(6)E(5)E(2)E(4)$.

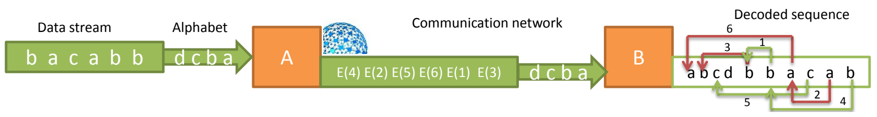


Fig. 2 A first sends B the alphabet *abcd* then it sends the codewords of gaps between consecutive occurrences of a character. B decodes the gaps and uses them to refer to the characters in part of the stream having been already decoded. The reference stream is $E(3)E(1)E(6)E(5)E(2)E(4)$.

Let O be a *reference encoding*; denote $L^O(S_t)$ as the length of the data including the length of the alphabet. The average number of bits per character is calculated as $\frac{L^O(S_t)}{t}$. The following theorem shows that when the data stream is generated by an *i.i.d* source, i.e. the same assumption guaranteeing the optimality of the *Huffman* code, the *reference encoding* scheme approximates the optimal solution by a constant factor with probability 1.

Theorem 1 (Near Optimality) *Given an i.i.d data stream S_t, let $H(P)$ denote the entropy of the distribution of the characters in the stream. If the Elias Delta code is used to encode natural numbers then:*

$$Pr\left(\lim_{t\mapsto\infty}\frac{L^O(S_t)}{t} \leq H(P) + \log_2(H(P)+1) + 1\right) = 1$$

Proof For every character $a_i \in \sum$, let $f_i(t)$ be the frequency of a_i in the stream S_t at time point t. Denote $C_i(t)$ as the total cost (in the number of bits) for encoding the occurrences of a_i. Therefore, the description length of the stream can be represented as:

$$L^O(S_t) = \sum_{i=1}^{n} C_i(t)$$

$$\Rightarrow \frac{L^O(S_t)}{t} = \frac{\sum_{i=1}^{n} C_i(t)}{t}$$

$$\Rightarrow \frac{L^O(S_t)}{t} = \sum_{i=1}^{n} \frac{C_i(t)}{f_i(t)} * \frac{f_i(t)}{t}$$

Given a character a_i denote $p_i > 0$ as the probability that a_i occurs in the stream. Denote G_i as the gap between two consecutive occurrences of a_i in the stream. Since the stream is i.i.d, G_i is distributed according to the *geometric distribution* with parameter p_i, i.e $Pr(G_i = k) = (1-p_i)^{k-1}p_i$. Recall that the expectation of G_i is $E[G_i] = \frac{1}{p_i}$.

According to the law of large numbers $Pr\left(\lim_{t\mapsto\infty}\frac{f_i(t)}{t} = p_i\right) = 1$. Moreover, recall that $C_i(t)$ is the sum of the gaps' codewords lengths and the initial cost for encoding the character a_i in the alphabet. When $t \mapsto \infty$, the frequency of a_i also goes to infinity because $p_i > 0$, therefore, by *the law of large number*: $Pr\left(\lim_{t\mapsto\infty}\frac{C_i(t)}{f_i(t)} = E[C(G_i)]\right) = 1$.

When the Elias Delta code is used to encode the gap, we have:

$$C(G_i) = \lfloor \log_2 G_i \rfloor + 2\lfloor \log_2 (\lfloor \log_2 G_i \rfloor + 1) \rfloor + 1$$
$$\Rightarrow C(G_i) \leq \log_2 G_i + 2\log_2 (\log_2 G_i + 1) + 1$$
$$\Rightarrow E[C(G_i)] \leq E[\log_2 G_i] + 2E[\log_2 (\log_2 G_i + 1)] + 1$$

Since log is a concave function, by the *Jenssen's inequality* :

$$E[C(G_i)] \leq \log_2 E[G_i] + 2\log_2 (\log_2 E[G_i] + 1) + 1$$

We further imply that:

$$Pr\left(\lim_{t\mapsto\infty}\frac{L^O(S_t)}{t} \leq \sum_i p_i \log_2 E[G_i] + 2p_i \log_2 (\log_2 E[G_i] + 1) + p_i\right) = 1$$

$$\Rightarrow Pr\left(\lim_{t\mapsto\infty}\frac{L^O(S_t)}{t} \leq \sum_{i=1}^{n} p_i \log_2 E[G_i] + 2\log_2 (\sum_{i=1}^{n} p_i \log_2 E[G_i] + 1) + 1\right) = 1$$

$$\Rightarrow Pr\left(\lim_{t\mapsto\infty}\frac{L^O(S_t)}{t} \leq H(P) + \log_2(H(P) + 1) + 1\right) = 1$$

from which the lemma is proved.□

It has been shown that in expectation the lower bound of the average number of bits per character of any encoding scheme is $H(P)$ [22]. Therefore, a corollary of Theorem 1 is that the *reference encoding* approximates the optimal solution by a constant factor $\alpha = 2$ plus one extra bit.

In the proof of Theorem 1 we can also notice that the gaps between two consecutive occurrences of a character represent the usage of the character in the offline encoding because in expectation the gap is proportional to the entropy of the character, i.e. $-\log p_i$. This property is very important because it provides us with a lot of conveniences in designing an effective algorithm to find compressing patterns in a data stream. In particular, since gaps can be calculated instantly without the knowledge about the whole data stream, using reference encoding we can solve all the aforementioned issues of the offline encodings discussed earlier in this section:

1. Using the reference encoding we don't need complete information about the stream to encode the observed part of the stream. The near optimality of the reference encoding in the i.i.d case is ensured by Theorem 1.
2. Under the presence of word evictions (to be discussed in the algorithm part), when an evicted word enters the dictionary again, we just need to follow that word for a while to get the recent gaps information with no need to care about the historical usage of the word before the eviction.
3. Updates can be handled more efficiently even under the presence of eviction because part of the stream that has been compressed remains unchanged.

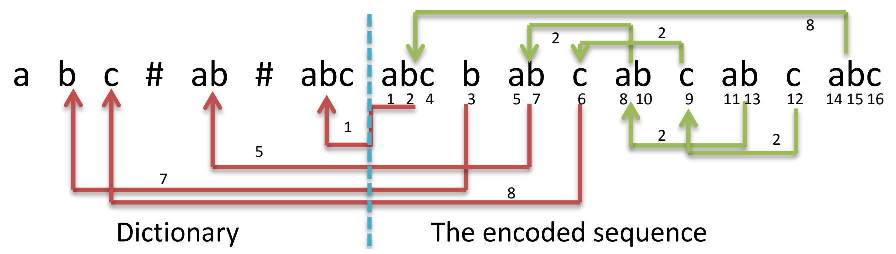


Fig. 3 An example of encoding the sequence $S = abbcacbacbacbabc$ with a reference encoding. The arrows represent the references between two consecutive encoded occurrences of a word which represent as a reference stream $E(1)E(7)E(5)E(8)E(2)E(2)E(2)E(2)E(8)$. Having the reference stream we can reconstruct the stream but not in the same order of event occurrence. Therefore, we need a gap stream indicating the gaps between consecutive characters in each encoded non-singletons. For example, the gap stream is $E(1)E(2)E(2)E(2)E(2)E(1)E(1)$.

3.2 Online encoding with non-singletons

The *reference encoding* can be easily extended for the case that uses singleton together with non-singleton patterns to encode a data stream. Let $\mathfrak{S} = S_1S_2\cdots S_t$ denote a stream of sequences where each S_i is a sequence of events. Let D be a dictionary containing all characters of the alphabet and some non-singletons. *Reference encodings* use D to compress a data stream $\mathfrak{S}$ by replacing instances of words in the dictionary by references to the most recent occurrences of the words. If the words are non-singletons, beside the references, gaps between characters of the encoded words must be stored together with the references. Therefore, in a reference encoding, beside the reference stream we also have a *gap stream*.

Similar to the case with non-singleton, first we need to encode the dictionary D (now contains both singleton and non-singleton). We add a special symbol $\sharp$ to the alphabet. The binary representation of the dictionary starts with the codeword of the size of the dictionary. It is followed by the codewords of all the characters in the alphabet each with length $\lceil \log_2 |D| \rceil$. The representations of every non-singleton follow right after that. The binary representation of a non-singleton contains codewords of the characters of the non-singleton. Non-singletons are separated from each other by the special character $\sharp$.

Example 2 (Dictionary representation) The dictionary $D = \{a, b, c, \sharp, ab, abc\}$ can be represented as follows $E(6)C(a)C(b)C(c)C(\sharp)C(a)C(b)C(\sharp)C(a)C(b)C(c)$. The representation starts with $E(6)$ indicating the size of D. It follows by the codewords of all the characters and the binary representation of the non-singletons separated by $\sharp$.

Having the binary representation of the dictionary, A first sends that representation to B. After that A send the encoding of the actual stream with the reference stream and the gap stream. The following example show how to encode a sequence with a reference encoding.

Example 3 (Reference encoding) Given the dictionary $D = \{a, b, c, \sharp, ab, abc\}$ and a sequence $S = abbcacbacbacbabc$. S can be encoded by a reference encoding using the dictionary D as shown in Figure 3 (the numbers below the character stream denote the positions of the character in the original stream). The reference stream is $E(1)E(7)E(5)E(8)E(2)E(2)E(2)E(2)E(8)$, where $E(1)$ is the reference of the first abc to the position of abc in the dictionary, $E(7)$ is the reference of the following b to the position of b in the dictionary and so on. The gap stream is $E(1)E(2)E(2)E(2)E(2)E(1)E(1)$, where for instance, the first codewords $E(1)E(2)$ indicate the gaps between a and b, b and c in the first occurrence of abc at position $(1, 2, 4)$ in the stream. For non-singleton, there is no gap information representing in the gap stream.

Example 4 (Decoding) In Figure 3, the sequence can be decoded as follows. Reading the first codeword of the reference stream, i.e. $E(1)$, the decoder refers one step back to get abc. There will be two gaps (between a, b and b, c) so the decoder reads the next two codewords from the gap stream, i.e. $E(1)$, and $E(2)$. Knowing the gaps it can infer the positions of abc. In this case, the positions are 1,2 and 4. Subsequently, the decoder reads the next codeword from the reference stream, i.e. $E(7)$, it refers seven steps back and decode the current reference as b. There is no gap because the word is a singleton, the position of b in the stream corresponds to the earliest position that has not been occupied by any decoded character, i.e. 3. The decoder continues decode the other references of the stream in the same way.

Different from the singleton case, there might be a lot of different reference encodings for a stream given a dictionary. Each reference encoding incurs different description lengths. Finding an optimal dictionary and an optimal reference encoding is the main problem we solve in this paper.

4 Problem definition

Given a data stream $\mathfrak{S}$ and a dictionary D denote $L^C_D(\mathfrak{S})$ as the description length of the data (including the cost to store the dictionary) in the encoding C. The problem of mining compressing sequential patterns in data stream can be formulated as follows:

Definition 2 (Compressing patterns mining) Given a stream of sequences $\mathfrak{S}$, find a dictionary D and an encoding C such that $L^C_D(\mathfrak{S})$ is minimized.

Generally, the problem of finding the optimal lossless compressed form of a sequence is NP-complete [24]. In this work, Problem 2 is similar to the data compression problem but with additional constraint on the number of passes through data. Therefore, in next section we discuss a heuristic algorithm inspired by the idea of the *Lempel-Ziv*'s data compression algorithm [22] to solve this problem.

Algorithm 1 Zips(S)

```
1: Input: Event stream $\mathfrak{S} = S_1 S_2 \cdots$
2: Output: Dictionary $D$
3: $D \longleftarrow \emptyset$
4: for $t = 1$ to $\infty$ do
5:     while $S_t \neq \epsilon$ do
6:         $w = encode(S_t)$
7:         $w^* = extend(w)$
8:         $update(w^*)$
9:     end while
10: end for
11: Return $D$
```

5 Algorithms

In this section, we discuss an algorithm for finding a good set of compressing patterns from a data stream. Our algorithm has the following essential properties for a streaming application:

1. Single pass: only one pass through the data.
2. Memory-efficient: since the data stream grows indefinitely, the streaming algorithms create a memory efficient summary of the stream.
3. Fast and scalable: summary update operation is fast to catch up with high-speed stream and scales up to the size of the stream.

We call our algorithm Zips as for *Zip a stream*. There are three important subproblems that Zips will solve. The first problem concerns how to grow a dictionary of promising candidate patterns for encoding the stream. Since the memory is limited, the second problem is how to keep a small set of important candidates and evict from the dictionary unpromising candidates. The last problem is that having a dictionary how to encode the next sequence effectively with existing words in the dictionary.

The pseudo-code depicted in Algorithm 1 shows how Zips work. It has three subroutines each of them solves one of the three subproblems:

1. Compress a sequence given a dictionary: for every new sequence S_t in the stream, Zips first uses the subroutine $encode(S_t)$ to find the word w in the dictionary which gives the most compression benefit when it is used to encode the uncompressed part of the sequence. Subsequently, the instance corresponds to the best chosen word is removed from S_t. Detail about how to choose w is discussed in subsection 5.1.
2. Grow a dictionary: Zips uses the subroutine $extend(w)$ to extend the word w returned by the subroutine $encode(S_t)$. Word extensions are discussed in subsection 5.2.
3. Dictionary update: the new extension is added to the dictionary. When the dictionary size exceeds the memory limit, a space-saving algorithm is used to evict unpromising words from the dictionary (subsection 5.3).

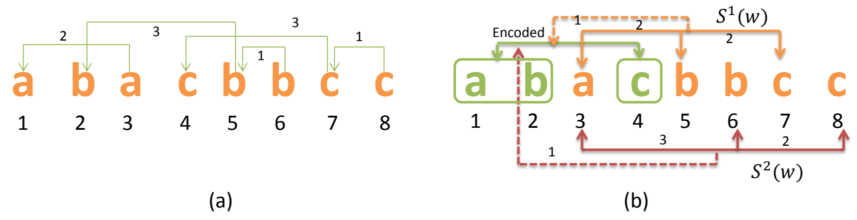


Fig. 4 An illustration of how compression benefit is calculated: (a) The sequence S in the uncompressed form. (b) two instances of $w = abc$: $S^1(w)$ and $S^2(w)$ and their references to the most recent encoded instance of w highlighted by the green color.

These steps are iteratively repeated as long as S_t is not encoded completely. When compression of the sequence finishes, Zips continues to compress the next in a similar way.

5.1 Compress a sequence:

Let S be a sequence, consider a dictionary word $w = a_1 a_2 \cdots a_k$, let $S(w)$ denote an instance of w in S. Let $g_2, g_3, \cdots, g_k$ be the gaps between the consecutive characters in $S(w)$. We denote the gap between the current occurrence and the most recent encoded occurrence of w by g. Let $\overline{g_i}$ $i = 1, \cdots, k$ be the gap between the current and the most recent occurrence of a_i. Therefore, to calculate the compression benefit we subtract the size of encoding $S(w)$ and the cost of encoding the gap to the last encoded occurrence of $S(w)$ from the size of encoding each singleton:

$$B(S(w)) = \sum_{i=1}^{k} |E(\overline{g_i})| - |E(g)| - \sum_{i=2}^{k} |E(g_i)| \qquad (1)$$

Example 5 (Compression benefit) Figure 4.a shows a sequence S in the uncompressed form and Figure 4.b shows the current form of S. Assume that the instance of $w = abc$ at positions $1, 2, 4$ is already compressed. Consider two instances of abc in the uncompressed part of S:

1. $S^1(w) = (a, 3)(b, 5)(c, 7)$: the cost to replace this instance by a pointer is equal to the sum of the cost to encode the reference to the previous encoded instance of abc $|E(1)|$ plus the cost of gaps $|E(2)| + |E(2)|$. The cost of representing this instance in an uncompressed form is $|E(2)| + |E(3)| + |E(3)|$. Therefore the compression benefit of using this instance to encode the sequence is $B(S^1(w)) = |E(2)| + |E(3)| + |E(3)| - |E(1)| - |E(2)| - |E(2)| = 3$ bits.
2. $S^2(w) = (a, 3)(b, 6)(c, 8)$: the compression benefit of using $S^2(w)$ to encode the sequence is calculated in a similar way: $B(S^2(w)) = |E(2)| + |E(1)| + |E(1)| - |E(1)| - |E(3)| - |E(2)| = -3$ bits.

In order to ensure that every symbol of the sequence is encoded, the next instance considered for encoding has to start at the first non-encoded symbol of the sequence. There maybe many instances of w in S that start with the first non-encoded symbol of S, denote $S^*(w) = argmax_{S(w)} B(S(w))$ as the one that results in the maximum compression benefit. We call $S^*(w)$ the *best match* of w in S. Given a dictionary, the encoding function depicted in Algorithm 2 first goes through the dictionary and finds the best match starting at the next uncompressed character of every dictionary word in the sequence S (line 4). Among all the best matches, it greedily chooses the one that results in the maximum compression benefit (line 6).

For any given dictionary word $w = a_1 a_2 \cdots a_k$, the most important subroutine of Algorithm 2 is to find the best match $S^*(w)$. This problem can be solved by creating a directed acyclic graph $G(V, E)$ as follows:

1. Initially, V contains a start node s and an end node e
2. For every occurrence of a_i at position p in S, add a vertex (a_i, p) to V
3. Connect s with the node (a_1, p) by a directed edge and add to that edge a weight value equal to $|E(\overline{g}_1)| - |E(g)|$.
4. Connect every vertex (a_k, p) with e by a directed edge with weight 0.
5. For all $q > p$ connect (a_i, p) to (a_{i+1}, q) by a directed edge with weight value $|E(\overline{g}_{i+1})| - |E(q - p)|$

Theorem 2 (The best match and the maximum path) *The best match $S^*(w)$ corresponds to the directed path from s to e with the maximum sum of the weight values along the path.*

The proof of theorem 2 is trivial since any instance of w in S corresponds to a directed path in the directed acyclic graph and vice versa. The sum of the weights along a directed path is equal to the benefit of using the corresponding instance of w to encode the sequence. Finding the directed path with maximum weight sum in a directed graph is a well-known problem in graph theory. That problem can be solved by a simple dynamic programming algorithm in linear time of the size of the graph, i.e. $O(|S|^2)$[25].

Example 6 (Find the best match in a graph) Figure 5 shows the directed acyclic graph created from the instances of *abc* in the uncompressed part of S shown in Figure 4.b. The best match of *abc* corresponding to the path $s(a, 3)(b, 5)(c, 7)e$ with the maximum sum of weights equal to 3 bits.

Algorithm 2 encode(S)

```
1: Input: a sequence $S$ and dictionary $D = w_1 w_2 \cdots w_N$
2: Output: the word $w$ that starts at the first non-encoded symbol gives the most additional compression benefit
3: for $i = 1$ to $N$ do
4:     $S^*(w_i) = bestmatch(S, w_i)$
5: end for
6: $max = argmax_i B(S^*(w_i))$
7: Return $w_{max}$
```

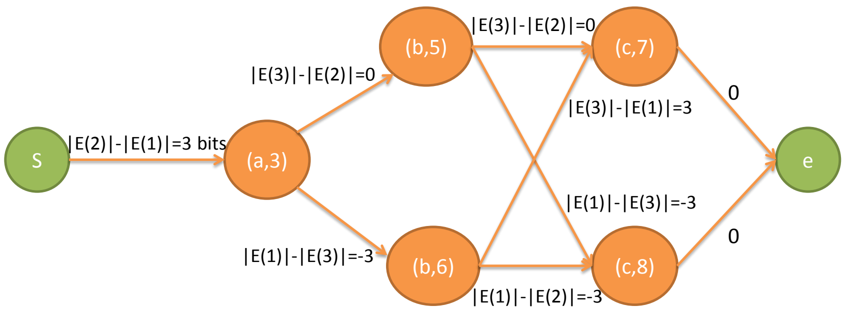


Fig. 5 A directed acyclic graph created from the instances of *abc* in the uncompressed part of S shown in Figure 4.b

It is important to notice that in order to evaluate the compression benefit of a dictionary word, Equation 1 only requires bookkeeping the position of the most recent encoded instance of the word. This is in contrast to the offline encodings used in recent work [3,4,20] in which the bookkeeping of the word usage and the gaps cost is a must. When a new instance of a word is replaced by a pointer, the relative usage and the codewords of all dictionary words also change. As a result, the compression benefit needs to be recalculated by a pass through the dictionary. This operation is an expensive task when the dictionary size is unbounded.

5.2 Dictionary extension:

Initially, the dictionary contains all singletons; it is iteratively expanded with the locally best words. In each step, when the best match of a dictionary word has been found, Zips extends the best match with one extra character and adds this extension to the dictionary. There are different options to choose the character for extension. In this work, Zips chooses the next uncompressed character right after the word. The choice is inspired by the same extension method suggested by the *Lempel-Ziv* compression algorithms.

Moreover, there is another reason behind our choice. When w is encoded for the first time, the reference to the previous encoded instance of w is undefined although the word has been already added to the dictionary. Under that circumstance, we have to differentiate between two cases: either a reference to an extension or to an encoded word. To achieve this goal one extra flag bit is added to every reference. When the flag bit is equal to 1, the reference refers to an extension of an encoded word. Otherwise, it refers to an encoded word. When the former case happens by extending the word with the character right after it, the decoder always knows where to find the last character of the extension. When a word is added to a dictionary all of its prefixes have been already added to the dictionary. This property enables us to store the dictionary by using a prefix tree.

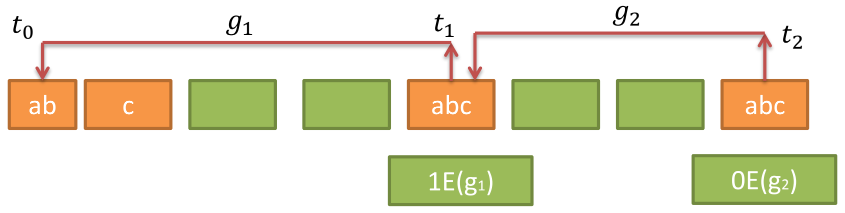


Fig. 6 An example of word is extended and encoded the first time and the second time. One extra flag bit is added to every reference to differentiate two cases.

Example 7 Figure 6 shows the first moment t_0 when the word $w = abc$ is added to the dictionary and two other moments t_1 and t_2 when it is used to encode the stream. At t_1, the flag bit 1 is used to notify the decoder that the gap g_1 is a reference to an extension of an encoded word, while at t_2 the decoder understands that the gap g_2 is a reference to the previous encoded instance of w.

Due to dictionary extensions, ambiguous reference may happen. For instance, a potential case of ambiguous reference called *reference loop* is discussed in the following example:

Example 8 (reference loops) At time point t_0 the word $w = ab$ at positions p_1p_2 is extended by c at position p_3. Later on at another time point $t_1 > t_0$, another instance of abc at $q_1q_2q_3$ $(q_1 > p_1)$ refers back to the instance abc at $p_1p_2p_3$. At time point $t_2 > t_1$, another instance of abc at $r_1r_2p_3$ $(r_1 > q_1)$ refers back to the instance abc at $q_1q_2q_3$. In this case, c at p_3 and c at q_3 refers to each other forming a reference loop.

Reference loops result in ambiguity when we decode the sequence. Therefore, when we look for the next best matches, in order to avoid reference loops, we always check if the new match incurs a loop. The match that incurs a loop is not considered for encoding the sequence. Checks for ambiguity can be done efficiently by creating paths of references. Vertices of a reference path are events in the sequence. Edge between two consecutive vertices of the path corresponds to a reference between the associated events. Since the total sizes of all the paths is at most the sequence length, its is cheap to store the paths for a bounded size sequence. Moreover, updating and checking if a path is a loop can be done in $O(1)$ if vertices of the path are stored in a hashmap.

Example 9 (reference paths) The references paths of the encoding in Example 8 are: $(a, r_1) \mapsto (a, q_1) \mapsto (a, p_1)$, $(b, r_2) \mapsto (b, q_2) \mapsto (b, p_2)$ and $(c, p_3) \mapsto (c, q_3) \mapsto (c, p_3)$. The last path is a loop.

5.3 Dictionary update:

The new extension is added to the dictionary. When the dictionary exceeds the memory limit, the *space-saving* algorithm is used to evict unpromising

Algorithm 3 update(w^*)

```
1: Input: a word $w^*$ and dictionary $D = \{w_1, w_2, \cdots, w_N\}$
2: Output: the dictionary $D$
3: $m \leftarrow |i : w_i$ is a non-singleton$|$
4: $v = argmin_i w_i[1]$ and $v$ is non-singleton at a leave of the prefix-tree
5: if $m > M$ and $w^* \notin D$ then
6:     $D = D \setminus \{v\}$
7:     $w^*[1] = w^*[2] = v[1]$
8:     $D = D \bigcup \{w^*\}$
9: else if $w^* \notin D$ then
10:    $w^*[1] = w^*[2] = 0$
11:    $D = D \bigcup \{w^*\}$
12: else
13:    add additional compression benefit to $w^*[1]$
14: end if
15: Return $D$
```

words from the dictionary. The space-saving algorithm [23] is a well-known method proposed for finding the most frequent items in a stream of items given a budget on the maximum number of counters it can keep in the stream summary. In this work, we propose a similar space-saving algorithm to keep the number of non-singleton words in the dictionary at below a predefined number M while it can be able to return the set of compressing patterns with high accuracy.

The algorithm works as follows, for every non-singleton word w it maintains a counter with two fields. The first field denoted as $w[1]$ contains an over-estimate of the compression benefit of w. The second field denoted as $w[2]$ contains the compression benefit of the word with least compression benefit in the dictionary at the moment that w is inserted into the dictionary.

Every time when a word w is chosen by Algorithm 2 to encode its best match in the sequence S_t, the compression benefit of the word is updated. The word w is then extended to w^* with an extra character by the extension subroutine. In its turn, Algorithm 3 checks if the dictionary already contains w^*. If the dictionary does not contains w^* and it is full with M non-singleton words, the least compressing word v resident at a leaf of the dictionary prefix-tree is removed from the tree. Subsequently, the word w^* is inserted into the tree and its compression benefit can be over-estimated as $w[1] = w[2] = v[1]$. The first counter of every word is always greater than the true compression benefit of the words. This property ensures that the new emerging word is not removed very quickly because its accumulated compression benefit is dominated by long lasting words in the dictionary. For any word w, the difference between $w[2]$ and $w[1]$ is the actual compression benefit of w since the moment that w is inserted into the dictionary. At anytime point when we need to find the most compressing patterns, we compare the value of $w[2] - w[1]$ and select those with highest $w[2] - w[1]$. In section 7 we show empirical results with different datasets that this algorithm is very effective in finding the most compressing patterns with high accuracy even with limited memory.

Datasets	# Sequences	# Events	Alphabet size	Ground-truth
Parallel	10000	1000000	25	Yes
Noise	10000	1000000	1025	Yes
Plant	1000	100000	1050	Yes
JMLR	787	75646	3846	No
Tweets	900417	8008552	452264	No
AOL	10122004	21080479	616145	No

Fig. 7 Datasets.

6 Algorithm analysis

Memory consumption: the algorithm needs to store the whole dictionary. The size of the dictionary is proportional to $O(|\sum|+M)$ where M is the maximum number of non-singletons in the dictionary. When the size of the alphabet is bounded and M is chosen as a constant, the memory consumption of Zips is constant too.

Computation complexity: the complexity of the dynamic programming algorithm for calculating the best match in a sequence S is $O(|S|^2)$. The maximum number of iterations to encode a sequence is $O(|S|)$. Therefore, the encoding function takes $O(M|S|^3)$ in the worst case. If M and $|S|$ are upper-bounded by a constant, the cost of encoding takes O(1). The extension also takes $O(1)$ with the assumption that M and $|S|$ are upper-bounded by a constant. Therefore, the complexity of the Zips algorithm is linear in the size of the stream. This fact is empirically verified in the section 7.

7 Experiments

We perform experiments with three synthetic datasets with ground truth and three large-scale real-world datasets. Our implementation of the Zips algorithm in C++ together with the datasets are available for download at our project website[2]. All the experiments were carried out on a machine with 16 processor cores, 2 Ghz, 12 GB memory, 194 GB local disk, Fedora 14 / 64-bit. As baseline algorithms, we choose the GoKrimp algorithm [3,4] and the SQS algorithm [20] for comparison in terms of running time, scalability, and interpretability of the set of patterns.

7.1 Data

We use six different datasets to evaluate the performance of the Zips algorithm. A summary of five datasets is presented in Figure 7. Details about the creation of these datasets from raw data are as follows:

[2] `www.win.tue.nl/~lamthuy/zips.html`

1. **Parallel** [3,4]: is a synthetic dataset which mimics a typical situation in practice where the data stream is generated by five independent parallel processes. Each process P_i $(i = 1, 2, \cdots, 5)$ generates one event from the set of events $\{A_i, B_i, C_i, D_i, E_i\}$ in that order. In each step, the generator chooses one of five processes uniformly at random and generates an event by using that process until the stream length is 1000000. For this dataset, we know the ground truth since all the sequences containing a mixture of events from different parallel processes are not the right patterns.
2. **Noise** [3,4]: is a synthetic dataset generated in the same way as the generation of the parallel dataset but with additional noise. A noise source generates independent events from a noise alphabet with 1000 distinct noise events and randomly mixes noise events with parallel data. The amount of noise is 20% of the data. All the subsequences containing mixtures of the events from different sources are considered wrong patterns.
3. **Plant**: is a synthetic dataset generated in the same way as the generation of the plant10 and plant50 dataset used in [20]. The plant10 and plant50 are small so we generate a larger one with ten patterns each with 5 events occurs 100 times at random positions in a sequence with length 100000 generated by 1000 independent noise event types.
4. **JMLR**: contains 787 abstracts of articles in the Journal of Machine Learning Research. English words are stemmed and stop words are removed. JMLR is small but it is considered as a benchmark dataset in the recent work [3,4,20]. The dataset is chosen also because the set of extracted patterns can be easily interpreted.
5. **Tweets**: contains over 1270000 tweets from 1250 different twitter accounts[3]. All tweets are ordered ascending by timestamp, English words are stemmed and stop words are removed. After preprocessing, the dataset contains over 900000 tweets. Similar to the JMLR dataset, this dataset is chosen because the set of extracted patterns can be easily interpreted.
6. **AOL**: contains over 25 million queries given by users of the AOL search engine[4]. All queries are ordered ascending by timestamp, English words are stemmed and stop words are removed. Duplicate queries by the same users in a session are removed. The final dataset after preprocessing contains more than 10 million queries.

7.2 Running time and Scalability

Figure 8 plots the running time and the average update time per sequence of the Zips algorithm in two dataset Tweets and AOL when the data stream size (the number of sequences) increases. Three different lines in each subplot correspond to different maximum dictionary size settings $M = 1000$, $M = 5000$ and $M = 10000$ respectively. The results show that the Zips algorithm scales linearly with the size of the stream. The average update time per

[3] `http://user.informatik.uni-goettingen.de/~txu/cuckoo/dataset.html`
[4] http://gregsadetsky.com/aol-data/

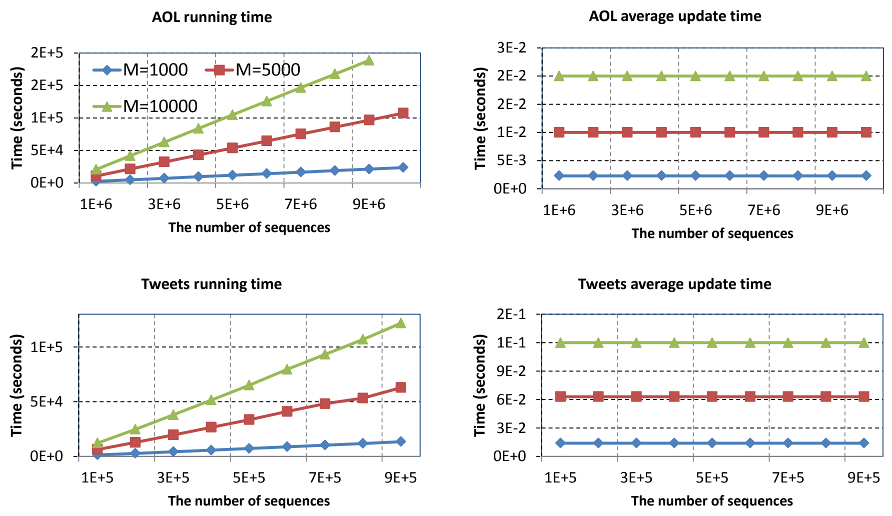


Fig. 8 The running time and the average update time per sequence of the Zips algorithm in two datasets Tweets and AOL when the stream size increases. Zips scales linearly with the size of the stream.

sequence is constant given a maximum dictionary size setting. For example, when $M = 10000$, Zips can handle one update in about 20-100 milliseconds.

Figure 9 shows the running time in y-axis of the Zips algorithm against the stream size in x-axis in two datasets Tweets and AOL when the maximum dictionary size is set to 1000. In the same figure, the running time of the baseline algorithms GoKrimp and SQS are also shown. There are some missing points in the results corresponding to the SQS algorithm because we set a deadline of ten hours for an algorithm to get the results corresponding to a point. The missing points corresponding to the cases when the SQS program did not finish in time.

In the log-log scale, three running time lines resemble a straight line. This result shows that the running time of Zips, GoKrimp and SQS are the power of data size, i.e. $T \sim \alpha|S|^\beta$. Using linear fitting functions in log-log scale we found that with the Zips algorithm $\beta = 1.02$ and $\beta = 1.01$ for the AOL and the Tweets datasets respectively, i.e. Zips scales linearly with the data size. Meanwhile, for the SQS algorithm the corresponding exponents are $\beta = 1.91$ and $\beta = 2.2$ and for the GoKrimp algorithm the exponents are $\beta = 2.28$ and $\beta = 2.01$. Therefore, both GoKrimp and SQS do not scale linearly with the data size and hence they are not suitable for data stream applications.

7.3 Real-world dataset

In this subsection, we discuss the interpretability of the patterns with three real-world datasets. All three datasets are text so it is easy to interpret the meanings of the set of patterns.

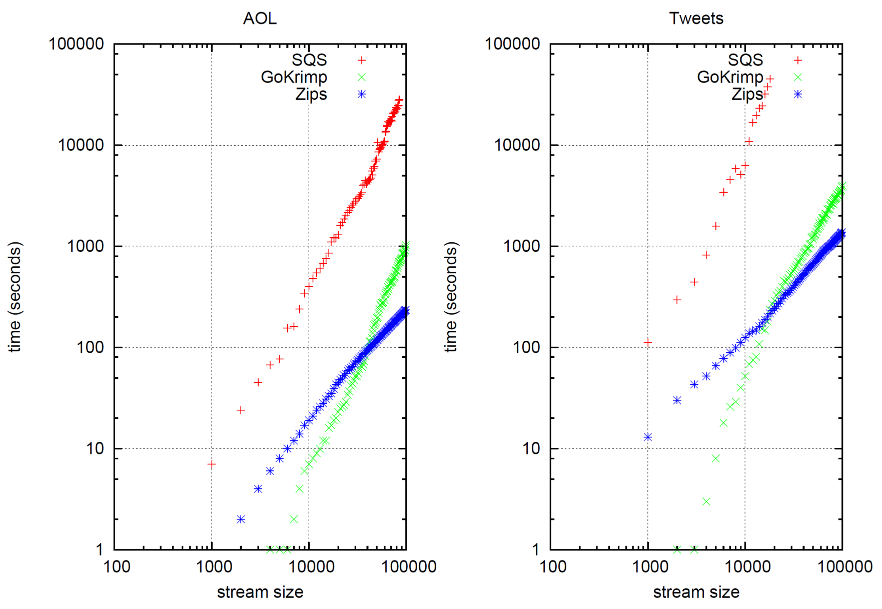
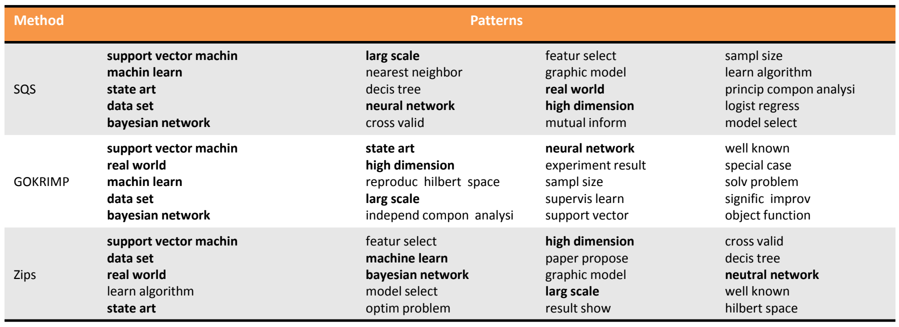


Fig. 9 Running time (x-axis) against the data size (y-axis) of three algorithms in log-log scales. The Zips algorithm scales linearly with the data size while the GoKrimp and the SQS algorithm scales quadratically with the data size.



Method	Patterns			
SQS	**support vector machin**	**larg scale**	featur select	sampl size
	machin learn	nearest neighbor	graphic model	learn algorithm
	state art	decis tree	**real world**	princip compon analysi
	data set	**neural network**	**high dimension**	logist regress
	bayesian network	cross valid	mutual inform	model select
GOKRIMP	**support vector machin**	**state art**	**neural network**	well known
	real world	**high dimension**	experiment result	special case
	machin learn	reproduc hilbert space	sampl size	solv problem
	data set	**larg scale**	supervis learn	signific improv
	bayesian network	independ compon analysi	support vector	object function
Zips	**support vector machin**	featur select	**high dimension**	cross valid
	data set	**machine learn**	paper propose	decis tree
	real world	**bayesian network**	graphic model	**neutral network**
	learn algorithm	model select	**larg scale**	well known
	state art	optim problem	result show	hilbert space

Fig. 10 The first 20 patterns extracted from the JMLR dataset by two baseline algorithms GoKrimp and SQS and the Zips algorithm. Common patterns discovered by all the three algorithms are bold.

7.3.1 JMLR

In Figure 10, we show the first 20 patterns extracted by two baseline algorithms GoKrimp and SQS and the Zips algorithm from the JMLR dataset. Three lists are slightly different but the important patterns such as "support vector machine", "data set", "machine learn", "bayesian network" or "state art" were discovered by all of the three algorithms. This experiment confirms that the Zips algorithm was able to find important patterns that are consistent with the results of state-of-the-art algorithms.

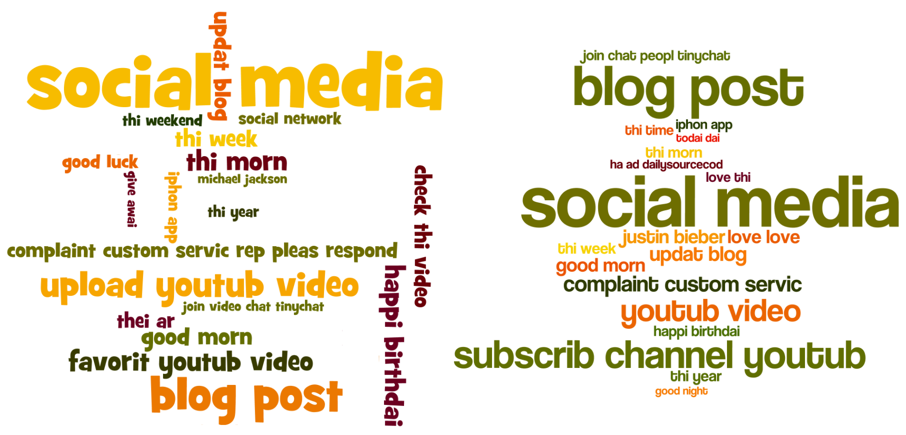


Fig. 11 The top 20 most compressing patterns extracted by Zips (left) and by GoKrimp (right) from the Tweets dataset.

7.3.2 Tweets

Since the tweet dataset is large, we schedule the programs so that they terminate their running after two weeks. The SQS algorithm was not able to finish its running before the deadline while GoKrimp finished running after three days and Zips finished running after 35 hours. The set of patterns extracted by the Zips algorithm and the GoKrimp algorithm are shown in Figure 11. Patterns are visualized by the wordcloud tool in R such that more important patterns are represented as larger words. In both algorithms, the sets of patterns are very similar. The result shows the daily discussions of the 1250 twitter accounts about the topics regarding "social media", "Blog post', about "youtube video", about "iphone apps", about greetings such as "happy birthday", "good morning" and "good night", about custom service complaint etc.

7.3.3 AOL

The AOL is a very large dataset. We scheduled the programs so that they terminate their running after two weeks. Both the SQS algorithm and the GoKrimp algorithm did not finish running before the deadline so we don't know the set of patterns extracted by these algorithms. Zips finished running after 58 hours. The result shows how people in the US used the AOL search engine in 2006. Figure 12 shows that the users of the AOL search engine (mostly from the US) were mostly interested in looking for information about "real estate", "high school", "community college", "credit union", "credit card" and "cell phone". Especially, the queries regarding locations in the US such as "los angel", "south and north Carolina", "York city" are also popular.

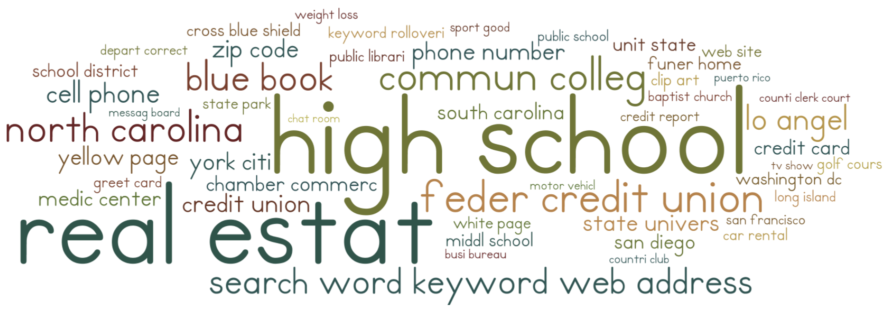


Fig. 12 Top 50 most compressing patterns extracted by Zips from the AOL dataset.

7.4 Synthetic dataset with known ground truths

In this subsection, we show the results with three synthetic datasets, i.e. the parallel dataset, the noise dataset and the plant dataset. For the parallel and the noise dataset, all sequences containing a mixture of events generated by different processes or containing at least one noise event are considered as wrong patterns. True patterns are sequences containing events generated by only one process. For the plant dataset, true patterns are ten sequences with exactly 5 events.

We get the first ten patterns extracted by each algorithm and calculate the precision and recall at K. Precision at K is calculated as the fraction of the number of right patterns in the first K patterns selected be each algorithm. While the recall is measured as the faction of the number of types of true patterns in the the first K patterns selected be each algorithms. For instance in the parallel dataset, if the set of the first 10 patterns contains only events from the set $\{A_i, B_i, C_i, D_i, E_i\}$ for a given i then the precision at $K = 10$ is 100% while the recall at $K = 10$ is 20%. The precision measures the accuracy of the set of patterns and the recall measures the diversity of the set of patterns.

Figure 13 shows the precision and recall at K for $K = 1, 2, \cdots, 10$. For the parallel and the noise dataset, an interesting result is that all the three algorithms are good at dealing with noise events, none of them return patterns that contain noise events. In term of precision GoKrimp and SQS were able to return all true patterns while the precision of the Zips algorithm is high with small K and the precision starts decreasing as K increases. At $K = 10$ the precision of the Zips algorithm is about about 65% in both datasets.

The SQS algorithm uses an encoding scheme that does not allow interleaving patterns so it returns only one among 5 different pattern types of patterns. Therefore, the recall of the SQS algorithm is low in contrast to the Gokrimp and the Zips algorithms where the recall is very high. The plant dataset is an ideal case when gaps between events of patterns are rare and patterns are not overlapping. In such case, three algorithms return a perfect result.

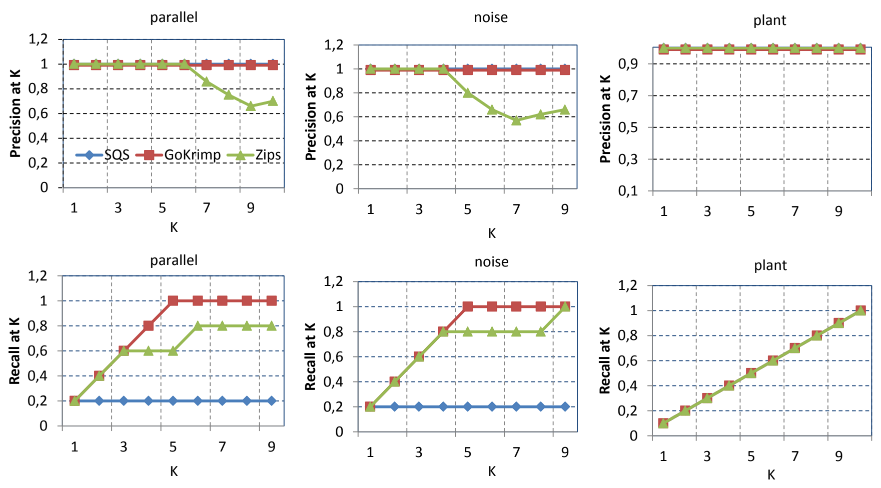


Fig. 13 The precision and recall at K of three algorithms SQS, GoKrimp and Zips in three synthetic datasets.

7.5 On the effects of the space-saving technique

The space saving technique requires the Zips algorithm to set the parameter M in advance, i.e. the maximum number of non-singleton dictionary words. As we have discussed in subsection 7.2, the update time is proportional to the value of M. In this subsection, we empirically show that when M is set to a reasonable value the top patterns extracted by the Zips algorithm is very similar to the top patterns when M is set to infinity.

It is important to notice that the case when M is set to ∞ is equivalent to the case when space saving algorithm is not applied. In this subsection, we will compare the results of the cases when $M = \infty$ and the results of the cases when M is increased from 1000 to 20000. When $M = \infty$, for the JMLR dataset, the Zips algorithm scaled up to the size of the entire dataset. However, since the Tweets and the AOL datasets are very large, the Zips algorithm was not able to scale up the size of these datasets. Therefore, for these datasets we report the results for only the first 100000 sequences.

First, we used the Zips algorithm to extract the first 100 patterns from three datasets Tweets, AOL and JMLR when M was set from 1000 to 20000. Then the similarity between two lists L_1 and L_2 with 100 elements each is calculated as $\frac{L_1 \bigcap L_2}{100}$. Figure 14 shows the similarity between the top-100 lists when M was set from 1000 to 20000 and the top-100 lists when $M = \infty$.

In the figure, we can see that the similarity of the top-K lists increases when the M increases. When $M = 20000$ the similarity reaches very high value and being close to 1.0. In the AOL dataset, the sequences are shorter so the similarity reaches high values even for small value of M.

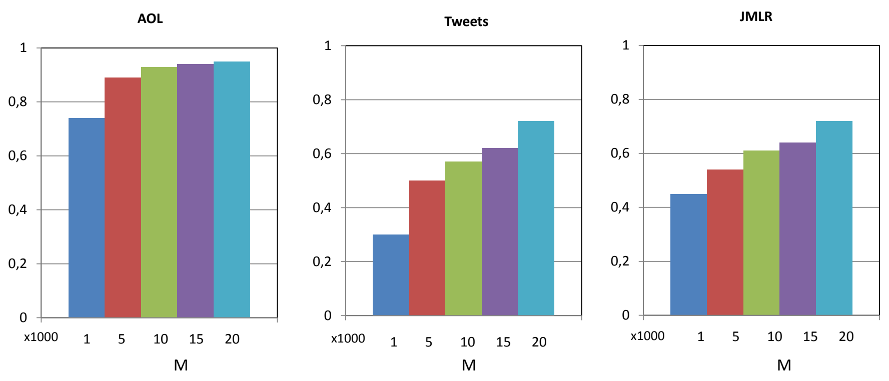


Fig. 14 Similarity between the lists of the first 100 patterns extracted by the Zips algorithm when M was set to $1000 - 20000$ and M was set to ∞.

8 Conclusions and future work

In this paper we studied the problem of mining compressing patterns from a data stream. A new encoding scheme for sequence is proposed. The new encoding is convenient for streaming applications because it allows encoding the data in an online manner. Because the problem of mining the best set of patterns with respect to the given encoding is shown to be unsolvable under the streaming context, we propose a heuristic solution that solves the mining compressing problem effectively. In the experiments with three synthetic datasets with ground-truths the proposed algorithm was able to extract the most compressing patterns with high accuracy. Meanwhile, in the experiments with three real-world datasets it can find patterns that are similar to the-state-of-the-art algorithms extract from these datasets. More importantly, the proposed algorithm was able to scale linearly with the size of the stream while the-state-of-the-art algorithms were not.

There are several options to extend the current work. One of the most promising future work is to study the problem of mining compressing patterns for different kinds of data stream such as a stream of graphs. In that problem a new encoding scheme must be defined but the idea of growing a dictionary incrementally and the space saving technique can be reapplied for those problems.

Acknowledgements The work is part of the project *Mining Complex Patterns in Stream* (COMPASS) supported by the Netherlands Organisation for Scientific Research (NWO). Part of the work was done when the first author was visiting Siemens Corporate Research center, a division of Siemens corporation in Princeton, NJ, USA.

References

1. Hong Cheng, Xifeng Yan, Jiawei Han, Philip S. Yu: Direct Discriminative Pattern Mining for Effective Classification. ICDE 2008: 169-178

2. Björn Bringmann, Siegfried Nijssen, Albrecht Zimmermann: Pattern-Based Classification: A Unifying Perspective. CoRR abs/1111.6191 (2011)
3. Hoang Thanh Lam, Fabian Moerchen, Dmitriy Fradkin, Toon Calders: Mining Compressing Sequential Patterns. SDM 2012: 319-330
4. Hoang Thanh Lam, Fabian Moerchen, Dmitriy Fradkin, Toon Calders: Mining Compressing Sequential Patterns. Accepted for publish in Statistical Analysis and Data Mining, A Journal of American Statistical Association, Wiley.
5. Nicolas Pasquier, Yves Bastide, Rafik Taouil, Lotfi Lakhal: Discovering Frequent Closed Itemsets for Association Rules. ICDT 1999: 398-416
6. Aristides Gionis, Heikki Mannila, Taneli Mielikäinen, Panayiotis Tsaparas: Assessing data mining results via swap randomization. TKDD 1(3) (2007)
7. Jilles Vreeken, Matthijs van Leeuwen, Arno Siebes: Krimp: mining itemsets that compress. Data Min. Knowl. Discov. 23(1): 169-214 (2011)
8. Toon Calders, Bart Goethals: Mining All Non-derivable Frequent Itemsets. PKDD 2002: 74-85
9. Dong Xin, Jiawei Han, Xifeng Yan, Hong Cheng: Mining Compressed Frequent-Pattern Sets. VLDB 2005: 709-720
10. Roberto J. Bayardo Jr.: Efficiently Mining Long Patterns from Databases. SIGMOD Conference 1998: 85-93
11. Jian Pei, Guozhu Dong, Wei Zou, Jiawei Han: On Computing Condensed Frequent Pattern Bases. ICDM 2002: 378-385
12. Robert Gwadera, Mikhail J. Atallah, Wojciech Szpankowski: Markov Models for Identification of Significant Episodes. SDM 2005
13. Milo R, Shen-Orr S, Itzkovitz S, Kashtan N, Chklovskii D, Alon U. Network motifs: simple building blocks of complex network. Science 2002
14. Tijl De Bie: Maximum entropy models and subjective interestingness: an application to tiles in binary databases. Data Min. Knowl. Discov. 23(3): 407-446 (2011)
15. Tijl De Bie, Kleanthis-Nikolaos Kontonasios, Eirini Spyropoulou: A framework for mining interesting pattern sets. SIGKDD Explorations 12(2): 92-100 (2010)
16. Nikolaj Tatti, Jilles Vreeken: Comparing apples and oranges: measuring differences between exploratory data mining results. Data Min. Knowl. Discov. 25(2): 173-207 (2012)
17. Michael Mampaey, Nikolaj Tatti, Jilles Vreeken: Tell me what i need to know: succinctly summarizing data with itemsets. KDD 2011: 573-581
18. Peter D. Grünwald The Minimum Description Length Principle MIT Press 2007
19. L. B. Holder, D. J. Cook and S. Djoko. Substructure Discovery in the SUBDUE System. In Proceedings of the AAAI Workhop on Knowledge Discovery in Databases, pages 169-180, 1994.
20. Nikolaj Tatti, Jilles Vreeken: The long and the short of it: summarising event sequences with serial episodes. KDD 2012: 462-470
21. Ian H. Witten, Alistair Moffat and Timothy C. Bell Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition. The Morgan Kaufmann Series in Multimedia Information and Systems. 1999
22. Thomas M. Cover and Joy A. Thomas. Elements of information theory. Second edition. Wiley Chapter 13.
23. Ahmed Metwally, Divyakant Agrawal, Amr El Abbadi: Efficient Computation of Frequent and Top-k Elements in Data Streams. ICDT 2005: 398-412
24. James A. Storer. Data compression via textual substitution Journal of the ACM (JACM) 1982
25. Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill
26. Nuno Castro, Paulo J. Azevedo: Significant motifs in time series. Statistical Analysis and Data Mining 5(1): 35-53 (2012)