# The Long and the Short of It: Summarising Event Sequences with Serial Episodes

Nikolaj Tatti Jilles Vreeken

Department of Mathematics and Computer Science
Universiteit Antwerpen
{firstname.lastname}@ua.ac.be

## ABSTRACT

An ideal outcome of pattern mining is a small set of informative patterns, containing no redundancy or noise, that identifies the key structure of the data at hand. Standard frequent pattern miners do not achieve this goal, as due to the pattern explosion typically very large numbers of highly redundant patterns are returned.

We pursue the ideal for sequential data, by employing a *pattern set* mining approach—an approach where, instead of ranking patterns individually, we consider results as a whole. Pattern set mining has been successfully applied to transactional data, but has been surprisingly understudied for sequential data.

In this paper, we employ the MDL principle to identify the set of sequential patterns that summarises the data best. In particular, we formalise how to encode sequential data using sets of serial episodes, and use the encoded length as a quality score. As search strategy, we propose two approaches: the first algorithm selects a good pattern set from a large candidate set, while the second is a parameter-free any-time algorithm that mines pattern sets directly from the data. Experimentation on synthetic and real data demonstrates we efficiently discover small sets of informative patterns.

## Categories and Subject Descriptors

H.2.8 [**Database management**]: Database applications–*Data mining*

## General Terms

Theory, Algorithms, Experimentation

## Keywords

serial episodes, event sequence, pattern mining, pattern set mining

## 1. INTRODUCTION

Suppose we are analysing an event sequence database, and are interested in its most important patterns. Traditionally, we would apply a frequent pattern miner, and mine all patterns that occur at least so-many times. However, due to the well-known pattern explosion we would then quickly be buried in huge amounts of highly redundant patterns, such that analysing the patterns becomes the problem, as opposed to the solution.

In this paper we therefore adopt a different approach. Instead of considering patterns individually, which is where the explosion stems from, we are after the *set of patterns* that summarises the data best. Desired properties of such a summary include that it should be small, generalise the data well, and be non-redundant. To this end, we employ the Minimum Description Length principle [5], which identifies the best set of patterns as that set by which we can describe the data most succinctly.

This approach has been shown to be highly successful for summarising transaction data [23], where the discovered patterns provide insight, as well as high performance in a wide range of data mining tasks, including clustering, missing value estimation, and anomaly detection [17, 22, 23]. Sequence data, however, poses additional challenges compared to itemsets. For starters, the order of events is important, and we have to take gaps in patterns into account. As such, encoding the data given a cover, finding a good cover given a set of patterns, as well as finding good sets of patterns, is much more complicated for sequences than for itemsets.

We are not the first to consider summarising sequential data. Existing methods, however, are different in that they require a single pattern to generate a full sequence [12], do not consider gaps [2, 12], or do not punish gaps in patterns [9]. In short, none of these methods take the full expressiveness of episodes into account.

As we identify the best model by best lossless compression, and we consider strings as data, standard compression algorithms such as Lempel-Ziv and Huffman coding are related [16]. While general purpose compressors can provide top-notch compression, they do not result interpretable models. In our case, compression is not the goal, but a means: in order to summarise the data well, we are after those serial episodes that describe it most succinctly. We discuss related work in closer detail in Section 5.

In this paper, we introduce a statistically well-founded approach for succinctly summarising event sequences, or SQS for short—pronounced as '*squeeze*'. We formalise how to encode a sequence dataset given a set of episodes, and using MDL identify the best set as the set that describes the data most succinctly. To optimise this score, we give an efficient heuristic to determine which pattern best describes what part of your data. To find good sets of patterns, we introduce two heuristics: SQS-CANDIDATES filters a given candidate collection, and SQS-SEARCH is a parameter-free any-time algorithm that efficiently mines models directly from data.

Experiments on real and synthetic data show SQS efficiently discovers high-quality models that summarise the data well, correctly identify key patterns. The number of returned patterns stays small, up to a few hundred— most importantly, though, the returned mod-

*KDD'12,* August 12–16, 2012, Beijing, China.
Copyright 2012 ACM 978-1-4503-1462-6 /12/08 ...$15.00.

els do not show redundancy, and none of the patterns are polluted by frequent, yet unrelated, events.

Altogether, the long and the short of it is that SQS mines small sets of the most important, non-redundant, serial episodes that together succinctly describe the data at hand.

# 2. MDL FOR EVENT SEQUENCES

In this section we formally introduce the problem we consider.

## 2.1 Preliminaries and Notation

As data type we consider *event sequences*. A sequence database $D$ over an event alphabet $\Omega$ consists of $|D|$ sequences $S \in D$. Every $S \in D$ is a sequence of $|S|$ events $e \in \Omega$, i.e. $S \in \Omega^{|S|}$. We write $S[i]$ to mean the $i$th event in $S$ and $S[i, j]$ to mean a subsequence $S[i] \cdots S[j]$. We denote by $||D||$ the sum of the lengths of all $S_i \in D$, i.e. $||D|| = \sum_{S_i \in D} |S_i|$. In this work, we do not explicitly consider time stamps, however we can extend our framework to time stamped events.

The support of an event $e$ in a sequence $S$ is simply the number of occurrences of $e$ in $S$, i.e. $supp(e \mid S) = |\{i \in S | i = e\}|$. The support of $e$ in a database $D$ is defined as $supp(e \mid D) = \sum_{S \in D} supp(e \mid S)$.

As patterns we consider serial episodes. A serial episode $X$ is a sequence of events and we say that a sequence $S$ contains $X$ if there is a subsequence in $S$ equal to $X$. Note that we are allowing gap events between the events of $X$. A singleton pattern is a single event $e \in \Omega$.

All logarithms in this paper are to base 2, and we employ the usual convention of $0 \log 0 = 0$.

## 2.2 MDL, a brief introduction

The Minimum Description Length principle (MDL) [5] is a practical version of Kolmogorov Complexity [11]. Both embrace the slogan *Induction by Compression*. For MDL, this can be roughly described as follows.

Given a set of models $\mathcal{M}$, the best model $M \in \mathcal{M}$ is the one that minimises $L(M) + L(D \mid M)$, in which $L(M)$ is the length in bits of the description of $M$, and $L(D \mid M)$ is the length of the data when encoded with model $M$.

This is called two-part MDL, or *crude* MDL—as opposed to *refined* MDL, where model and data are encoded together [5]. We use two-part MDL because we are specifically interested in the model: the patterns that give the best description. Further, although refined MDL has stronger theoretical foundations, it cannot be computed except for some special cases. Note that MDL requires the compression to be *lossless* in order to allow for fair comparison between different $M \in \mathcal{M}$, and that we are only concerned with code lengths, not actual code words.

To use MDL, we have to define what our models $\mathcal{M}$ are, how a $M \in \mathcal{M}$ describes a database, and how we encode these in bits.

## 2.3 MDL for Event Sequences

As models we consider *code tables*. A code table is essentially a look-up table, or dictionary, between patterns and associated codes. A code table has four columns, of which the first column contains patterns, the second column consists of codes for identifying these patterns, and the two right-most columns contain pattern-dependent codes for identifying gaps or the absence thereof within an embedding of a pattern. To ensure any sequence over $\Omega$ can be encoded by a code table, we require that all the singleton events in the alphabet, $X \in \Omega$, are included in a code table $CT$.

To refer to the different codes in $CT$, we write $code_p(X \mid CT)$ when we refer to the code corresponding to a pattern $X$, as stored in the second column of a code table $CT$. Similarly, we write $code_g(X \mid CT)$ and $code_n(X \mid CT)$ to resp. refer to the codes stored in the third and fourth column, which indicate whether or not the next symbol is part of a gap in the usage of pattern $X$. For readability, we do not write $CT$ wherever clear from context.

Our next step is to explain our encoding scheme. As we will see later, there are typically very many ways of encoding a database, apart from using only singletons. Hence, for clarity, we will first explain our encoding scheme by considering how to *decode* an already encoded database, and postpone finding a good encoding, or *cover*, as well as how to find a good code table to Sections 3 and 4.

### *Decoding a Database*

An encoded database consists of two code streams, $C_p$ and $C_g$, that follow from the cover $C$ chosen to encode the database. The first code stream, the *pattern-stream*, denoted by $C_p$, is a list of $|C_p|$ codes, $code_p(\cdot)$, for patterns $X \in CT$ corresponding to those patterns chosen by 'cover' algorithm. For example, $code_p(a) code_p(b)$ $code_p(c)$ encodes the sequence '$abc$'.

Serial episodes are not simple subsequences, however, as they allow for gaps. That is, a pattern $de$ specifies that event $d$, after possibly some other events, is followed by event $e$. As such, this pattern occurs both in sequence '$de$' as well as in sequence '$dfe$'. In the former there is no gap between the two events, and in the latter there is a gap of length one, in which event $f$ occurs.

As such, only when we read the code for a singleton pattern $X$ we can directly unambiguously append $X$ to the sequence $S_k$ we are decoding—there can be no gap in a singleton pattern. When $X$ is a non-singleton pattern, on the other hand, we may only append the first symbol $x_1$ of $X$ to $S_k$; before appending event $x_2$ to $S_k$, we first need to know whether there is a gap between the two events in this usage of $X$, and if so, what event(s) occur in the gap. This is what the second code stream is for. This stream, the *gap-stream*, denoted by $C_g$, is a list of codes from the third and fourth column of $CT$, indicating whether gaps occur when decoding patterns.

Given the gap-stream, we can determine whether the next event of $S_k$ may be read from the current pattern $X$, or we have to read a singleton pattern to fill the gap. Starting with an empty sequence for $S_k$, and assuming that we know its final length, we read the code for the first pattern $X$, $code_p(X)$, from $C_p$, and append the first event $x_1$ of $X$ to the sequence we are decoding. If $X$ is a non-singleton pattern, we read from $C_g$ whether the next event is a gap-event, or not. If we read the gap-code $code_g(X)$ from $C_g$ there is indeed a gap, after which we read from $C_p$ the $code(Y)$ for the (singleton) event $Y \in CT$ associated with this gap—and append it to $S_k$. We then read again from $C_g$ whether there is another gap, etc, until we encounter the no-gap code $code_n(X)$ in $C_g$ indicating we should append the next symbol $x_j$ of $X$ to $S_k$. Whenever we are finished decoding pattern $X$, we read the code for the next pattern $X$ from $C_p$, until we have read as many events as $S_k$ should be long, after which we continue decoding $S_{k+1} \in D$ until $C_p$ is depleted and all $S \in D$ are reconstructed.

Consider the toy example given as Fig. 1. One possible encoding would be to use only singletons, meaning that gap stream is empty. Another encoding is to use patterns. For example, to encode '$abdc$', we first give the code for $abc$ in the pattern stream, then a no-gap code (white) in $C_g$ to indicate $b$, then a gap code (black) in $C_g$, next the code for $d$ in $C_p$, and we finish with a no-gap code in $C_g$.

### *Calculating Encoded Lengths*

Given the above decoding scheme we know what codes we can expect to read where and when, and hence can now formalise how to

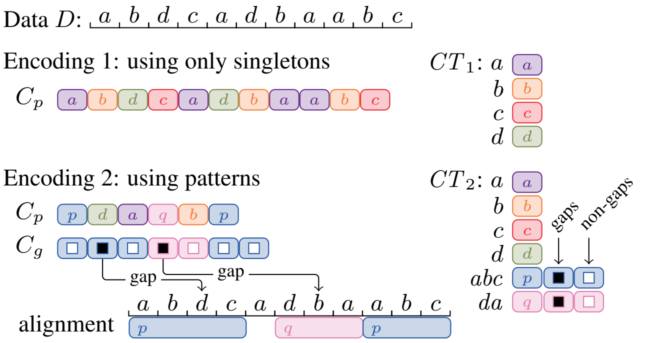

**Figure 1: Toy example of two possible encodings. The first encoding uses only singletons. The second encoding uses singletons and two patterns, namely,** $abc$ **and** $da$

calculate the lengths of these codes, as well as the encoded lengths of the code table and database.

We start with $L(code_p(X))$, the lengths of pattern codes in $C_p$, which we can look up in the second column of $CT$. By Shannon Entropy [3] we know that the length, in bits, of the optimal prefix-free code for an event $X$ is $-\log \Pr(X)$, where $\Pr(X)$ is the probability of $X$. Let us write $usage(X)$ for how often $code_p(X)$ occurs in $C_p$. That is, $usage(X) = |\{Y \in C_p \mid Y = code_p(X)\}|$. Then, the probability of $code_p(X)$ in $C_p$ is its relative occurrence in $C_p$. So, we have

$$L(code_p(X) \mid CT) = -\log \left( \frac{usage(X)}{\sum_{Y \in CT} usage(Y)} \right) \quad .$$

Similarly, the lengths of the codes for indicating the presence or absence of a gap in the usage of a pattern $X$, resp. $L(code_g(X))$ and $L(code_n(X))$, should be dependent on their relative frequency. Let us write $gaps(X)$ to refer to the number of gap events within the usages of a pattern $X$ in the cover of $D$. We then resp. have

$$fills(X) = usage(X)(|X| - 1) \quad ,$$

for the number of non-gap codes in the usage of a pattern $X$, and

$$L(code_g(X) \mid CT) = -\log \left( \frac{gaps(X)}{gaps(X) + fills(X)} \right) \quad ,$$

$$L(code_n(X) \mid CT) = -\log \left( \frac{fills(X)}{gaps(X) + fills(X)} \right) \quad ,$$

resp. for the length of a gap and a non-gap code of a pattern $X$.

We say a code table $CT$ is code-optimal for a cover $C$ of a database $D$ if all the codes in $CT$ are of the length according to their respective usage frequencies in $C_p$ and $C_g$ as defined above.

From the lengths of the individual codes, the encoded length of the code streams now follows straightforwardly, with resp.

$$L(C_p \mid CT) = \sum_{X \in CT} usage(X) L(code_p(X))$$

for the encoded size of the pattern-stream, and

$$\begin{aligned} L(C_g \mid CT) \quad = \sum_{\substack{X \in CT \\ |X|>1}} \Big( & gaps(X) L(code_g(X)) + \\ & fills(X) L(code_n(X)) \Big) \end{aligned}$$

for the encoded size of the gap-stream.

Combining the above, we define $L(D \mid CT)$, the encoded size of a database $D$ given a code table $CT$ and a cover $C$, as

$$\begin{aligned} L(D \mid CT) \quad = \quad & L_{\mathbb{N}}(|D|) + \sum_{S \in D} L_{\mathbb{N}}(|S|) + \\ & L(C_p \mid CT) + L(C_g \mid CT) \quad , \end{aligned}$$

where $|D|$ is the number of sequences in $D$, and $|S|$ is the length of a sequence $S \in D$. To encode these values, for which we have no prior knowledge, we employ the MDL optimal Universal code for integers [5, 15]. For this encoding, $L_{\mathbb{N}}$, the number of bits required to encode an integer $n \geq 1$, is defined as

$$L_{\mathbb{N}}(n) = \log^*(n) + \log(c_0) \quad ,$$

where $\log^*$ is defined as $\log^*(n) = \log(n) + \log\log(n) + \cdots$, where only the positive terms are included in the sum. To make $L_{\mathbb{N}}$ a valid encoding, $c_0$ is chosen as $c_0 = \sum_{J \geq 1} 2^{-L_{\mathbb{N}}(j)} \approx 2.865064$ such that the Kraft inequality is satisfied.

Next we discuss how to calculate $L(CT)$, the encoded size of a code table $CT$. To ensure lossless compression, we need to encode the number of entries, for which we employ $L_{\mathbb{N}}$ as defined above. For later use, and to avoid bias by large or small alphabets, we encode the number of singletons, $|\Omega|$, and the number of non-singleton entries, $|CT \setminus \Omega|$, separately. We disregard any non-singleton pattern with $usage(X) = 0$, as it is not used in describing the data, and has no valid (or infinite length) pattern code.

For the size of the left-hand side column, note that the simplest valid code table consists only of the single events. This code table we refer to as the *standard code table*, or $ST$. We encode the patterns in the left-hand side column using the pattern codes of $ST$. This allows us to decode up to the names of events.

As singletons cannot have gaps, the usage of a singleton $Y$ given $ST$ is simply the support of $Y$ in $D$. Hence, the code length of $Y$ in $ST$ is defined as $L(code_p(Y) \mid ST) = -\log \frac{supp(Y \mid D)}{||D||}$. Before we can use these codes, we must transmit these supports. We transmit these using a data-to-model code [21], an index over a canonically ordered enumeration of all possibilities; here, the number of ways $||D||$ events can be distributed over $|\Omega|$ labels, where none of the bins may be empty, as $supp(Y \mid D) > 0$. The number of such possibilities is given by $\binom{||D||-1}{|\Omega|-1}$, and by taking a log we have the number of bits required to identify the right set of values. Note that $||D||$ is known from $L(D \mid CT)$. In general, for the number of bits for an index of a number composition, the number of combinations of summing to $m$ with $n$ non-zero terms, we have $L_U(m, n) = \log \binom{m-1}{n-1}$, where for $m = 0$, and $n = 0$, we define $L_U(m, n) = 0$.

Combined, this gives us the information required to reconstruct the left-hand side of $CT$ for the singletons, as well as the information needed to decode the non-singleton patterns of $CT$. For a pattern $X$, the number of bits in the left-hand column is the length of $X$, $|X|$, as encoded by $L_{\mathbb{N}}$, and the sum of the singleton codes

$$L_{\mathbb{N}}(|X|) + \sum_{x_i \in X} L(code(x_i) \mid ST) \quad .$$

Next, we encode the second column. To *avoid* bias, we treat the singletons and non-singleton entries of $CT$ differently. Let us write $\mathcal{P}$ to refer to the non-singleton patterns in $CT$, i.e. $\mathcal{P} = CT \setminus \Omega$. For the elements of $\mathcal{P}$, we first encode the sum of their usages, denoted by $usage(\mathcal{P})$, and use a data-to-model code like above to identify the correct set of individual usages. With these values, and the singleton supports we know from $ST$, we can reconstruct the usages of the singletons in $CT$, and hence reconstruct the pattern codes associated with each pattern in $CT$.

This leaves us the gap-codes for the non-singleton entries of $CT$. For reconstructing these, we need to know $gaps(X)$, which we encode using $L_{\mathbb{N}}$. The number of non-gaps then follows from the length of a pattern $X$ and its usage. As such, we can determine $code_g(X)$ and $code_n(X)$ exactly.

Putting this all together, we have $L(CT \mid C, D)$, the encoded size in bits of a code table $CT$ for a cover $C$ of a database $D$, as

$$\begin{aligned} L(CT \mid C) = & L_{\mathbb{N}}(|\Omega|) + L_U(||D||, |\Omega|) + \\ & L_{\mathbb{N}}(|\mathcal{P}| + 1) + L_{\mathbb{N}}(usage(\mathcal{P}) + 1) + \\ & L_U(usage(\mathcal{P}), |\mathcal{P}|) + \sum_{X \in \mathcal{P}} L(X, CT) \quad , \end{aligned}$$

where $L(X, CT)$, the number of bits for encoding the events, length, and the number of gaps of patterns $X$ in $CT$, is

$$\begin{aligned} & L(X, CT) \\ & \quad = L_{\mathbb{N}}(|X|) + L_{\mathbb{N}}(gaps(X) + 1) + \sum_{x \in X} L(code_p(x \mid ST)) \quad . \end{aligned}$$

By MDL, we can then define the optimal set of serial episodes for a given sequence database as the set for which the optimal cover and associated optimal code table minimises the total encoded size

$$L(CT, D) = L(CT \mid C) + L(D \mid CT) \ .$$

More formally, we define the problem as follows.

**Minimal Code Table Problem** *Let $\Omega$ be a set of events and let $D$ be a sequence database over $\Omega$, find the minimal set of serial episodes $\mathcal{P}$ such that for the optimal cover $C$ of $D$ using $\mathcal{P}$ and $\Omega$, the total encoded cost $L(CT, D)$ is minimal, where $CT$ is the code-optimal code table for $C$.*

Clearly, this problem entails a rather large search space. First of all, given a set of patterns, there are many different ways to cover a database. Second, there are very many sets of serial episodes $\mathcal{P}$ we can consider, namely all possible subsets of the collection of serial episodes that occur in $D$. However, neither the full problem, or these sub-problems, exhibit trivial structure that we can exploit for fast search, e.g. (weak) monotonicity.

We hence break the **Minimal Code Table Problem** into two subproblems. First, in the next section we discuss how to optimise the cover of a sequence *given* a set of episodes. Then, in Section 4, we will discuss how to mine high quality code tables.

# 3. COVERING A STRING

Encoding, or covering, a sequence is more difficult than decoding one. The reason is simple: when decoding there is no ambiguity, while when encoding there are many choices, i.e. what pattern to encode a symbol with. In other words, given a set of episodes, there are many valid ways to cover a sequence, where by our problem definition we are after the cover $C$ that minimises $L(CT, D)$.

Due to lack of space, we provide the proofs in Appendix A.

## 3.1 Minimal windows

Assume we are decoding a sequence $S_k \in D$. Assume we decode the beginning of a pattern $X$ at $S_k[i]$ and that the last symbol belonging to this instance of $X$ is, say, $S_k[j]$. We say that $S_k[i, j]$ is an *active* window for $X$. Let $\mathcal{P}$ be the set of non-singleton patterns used by the encoding. We define an *alignment* $A$ to be the set of all active windows for all non-singleton patterns $X \in \mathcal{P}$ as

$$A = \{(i, j, X, k) \mid S_k[i, j] \text{ is an active window for } X, S_k \in D\} \ .$$

An alignment corresponding to the second encoding given in Figure 1 is $\{(1, 4, abc, 1), (6, 8, da, 1), (9, 11, abc, 1)\}$.

Note that an alignment $A$ does not uniquely define the cover of the sequence, as it does not take into account how the intermediate symbols (if any) within the active windows of a pattern $X$ are encoded. However, an alignment $A$ for a sequence database $D$ does define an equivalence class over covers of the same encoded length. In fact, given a sequence database $D$ and an alignment $A$, we can determine the number of bits our encoding scheme would require for such a cover. To see this, let $X$ be a pattern and let $W = \{(i, j, X, k) \in A\}$, then

$$usage(X) = |W| \text{ and } gaps(X) = gaps(W) \quad , \qquad (1)$$

where

$$gaps(W) = \sum_{(i,j,X,k) \in W} j - i - |X| - 1 \quad . \qquad (2)$$

The remaining symbols are encoded as singleton patterns. Hence, the usage of a singleton is equal to

$$usage(s) = supp(s \mid D) - \sum_{s \in X} usage(X) \quad . \qquad (3)$$

Given an alignment $A$ for $D$, we can trivially construct a valid cover $C$ for $D$, simply by following $A$ and greedily covering $S_k$ with pattern symbols if possible, and singletons otherwise. That is, if for a symbol $S_k[i]$ we have, by $A$, the choice for covering it as a gap or non-gap of a pattern $X$, we choose non-gap.

Then, from either $C$, or directly from $A$, we can derive the associated code-optimal code table $CT$. Given an alignment $A$, let us write $CT(A)$ for this code table. Wherever clear from context, we will write $L(D \mid A)$ to mean $L(D \mid CT(A))$, and similarly $L(D, A)$ as shorthand for $L(D, CT(A))$.

Our next step is to show what kind of windows can occur in the optimal alignment. We say that $W = S[i, j]$ is a minimal window of a pattern $X$ if $W$ contains $X$ but no other proper sub-windows of $W$ contain $X$. For example, in Figure 1 $S[6, 8]$ is a minimal window for $da$ but $S[6, 9]$ is not.

PROPOSITION 1. *Let $A$ be an alignment producing an optimal encoded length. Then all active windows in $A$ are minimal windows.*

Proposition 1 says that we need to only study minimal windows. Let $\mathcal{F}$ be a set of episodes and let $X \in \mathcal{F}$. Since an event $S_k[i]$ can be a starting point to only one minimal window of $X$, there are only $\|D\|$ minimal windows of $X$ in $D$, at most. Consequently, the number of minimal windows we need to investigate is bounded by $\|D\| \, |\mathcal{F}|$. Moreover, we can use FINDWINDOWS in [20] to discover all the minimal windows for a pattern $X$ in $O(|X| \, \|D\|)$ time.

## 3.2 Finding optimal alignment

Discovering an optimal alignment is non-trivial due to the complex relation between code lengths and the alignment. However, if we fix the alignment, Eqs. 1–3 give us the codes optimising $L(D \mid A)$. In this section we will show the converse, that if we fix the codes, we can easily find the alignment optimising $L(D \mid A)$. In order to do that let $w = (i, j, X, k)$ be a minimal window for a pattern $X$. We define the gain to be

$$\begin{aligned} gain(w) = & - L(code_p(X)) - (j - i - |X|) L(code_g(X)) \\ & - (|X| - 1) L(code_n(X)) + \sum_{x \in X} L(code_p(x)) \quad . \end{aligned}$$

PROPOSITION 2. *Let $D$ be a dataset and $A$ be an alignment. Then the length of encoding $D$ is equal to*

$$L(D \mid A) = const - \sum_{w \in A} gain(w) \quad ,$$

*where const does not depend on $A$.*

This proposition suggests that if we fix the code lengths we need to maximise the total gain. In order for an alignment to be valid, the windows must be disjoint. Hence, given a set of $W$, consisting of all minimal windows of the given patterns, we need to find a subset $O \subseteq W$ of disjoint windows maximising the gain.

Assume that $W$ is ordered by the first index of each window. For a window $w$, define $next(w)$ to be the next disjoint window in $W$. Let $o(w)$ be the optimal total gain of $w$ and its subsequent windows. Let $v$ be the next window of $w$, then the optimal total gain is $o(w) = \min(o(v), gain(w) + o(next(w)))$. This gives us a simple dynamic program, ALIGN, given as Algorithm 1.

**Algorithm 1:** ALIGN($W$)

```
input  : minimal windows $W$ sorted by the first event
output : mutually disjoint subset of $W$ having the optimal gain
$o(N+1) \leftarrow 0$; $opt(N+1) \leftarrow$ none;
foreach $i = N, \ldots, 1$ do
    $c \leftarrow 0$;
    if $next(i)$ then $c \leftarrow o(next(i))$;
    if $gain(w_i) + c > o(i+1)$ then
        $o(i) \leftarrow gain(w_i) + c$; $opt(i) \leftarrow i$;
    else
        $o(i) \leftarrow o(i+1)$; $opt(i) \leftarrow opt(i+1)$;
$O \leftarrow$ optimal alignment (obtained by iterating $opt$ and $next$);
return $O$;
```

We can now use ALIGN iteratively. Given the codes we find the optimal alignment and derive the optimal codes given the new alignment. We repeat this until convergence, which gives us a heuristic approximation to the optimal alignment $A^*$ for $D$ using patterns $\mathcal{P}$. As initial values, we use the number of minimal windows as usage and set gap code length to be 1 bit. The pseudo code of SQS, which stands for Summarising event seQuenceS, is given as Algorithm 2.

**Algorithm 2:** SQS($D, \mathcal{P}$). Summarising event seQuenceS

```
input  : Database of sequences $D$, set of patterns $\mathcal{P}$
output : Alignment $A$
foreach $s \in \Omega$ do $usage(s) \leftarrow supp(s \mid D)$;
foreach $X \in \mathcal{P}, |X| > 1$ do
    $W_X \leftarrow$ FINDWINDOWS($X, D$);
    $usage(X) \leftarrow |W_X|$; $gaps(X) \leftarrow |X| - 1$;
$W \leftarrow$ merge sort $\{W_X\}_{X \in \mathcal{F}}$ based on first event;
while changes do
    compute gain for each $w \in W$;
    $A \leftarrow$ ALIGN($W$);
    recompute usage and gaps from $A$ (Eqs. 1–3);
return $A$;
```

The computational complexity of single iteration comes down to the computational complexity of ALIGN($W$), which is $O(|W|) \subseteq O(|\mathcal{P}|\,\|D\|)$. Also note that $next$ is precomputed before calling ALIGN and this can be also computed with a single scan, taking $O(|W|)$ steps. Note that the encoded length improves at every iteration, and as there are only finite number of alignments, SQS will converge to a local optimum in finite time. In practice, the number of iterations is small—in the experiments typically less than 10.

# 4. MINING CODE TABLES

With the above, we both know how to score the quality of a pattern set, as well as how to heuristically optimise the alignment of a pattern set. This leaves us with the problem of finding good sets of patterns. In this section we give two algorithms to do so.

## 4.1 Filtering Candidates

Our first algorithm, SQS-CANDIDATES, assumes that we have a (large) set of candidate patterns $\mathcal{F}$. In practice, we assume the user obtains this set of patterns using a frequent pattern miner, although any set of patterns over $\Omega$ will do. From this set $\mathcal{F}$ we then select that set of patterns $\mathcal{P} \subseteq \mathcal{F}$ such that the optimal alignment $A$ and associated code table $CT$ minimises $L(D, CT)$.

For notational brevity, we simply write $L(D, \mathcal{P})$ as shorthand for the total encoded size $L(D, CT)$ obtained by the code table $CT$ containing a set of patterns $\mathcal{P}$ and singletons $\Omega$, and being code-optimal to the alignment $A$ as found by SQS.

We begin by sorting the candidates $X \in \mathcal{F}$ by $L(D, \{X\})$ from lowest to highest. After sorting, we iteratively greedily test each pattern $X \in \mathcal{F}$. If adding $X$ to $\mathcal{P}$ improves the score, i.e. fewer bits are needed, we keep $X$ in $\mathcal{P}$, otherwise it is permanently removed. The pseudo-code for SQS-CANDIDATES is given as Algorithm 3.

**Algorithm 3:** SQS-CANDIDATES($\mathcal{F}, D$)

```
input  : candidate patterns $\mathcal{F}$
output : set of non-singleton patterns $\mathcal{P}$ that heuristically
         minimise the Minimal Code Table Problem
order patterns $X \in \mathcal{F}$ based on $L(D, \{X\})$;
$\mathcal{P} \leftarrow \emptyset$;
foreach $X \in \mathcal{F}$ in order do
    if $L(D, \mathcal{P} \cup X) < L(D, \mathcal{P})$ then
        $\mathcal{P} \leftarrow$ PRUNE($\mathcal{P} \cup X, D,$ false);
$\mathcal{P} \leftarrow$ PRUNE($\mathcal{P}, D,$ true);
order patterns $X \in G$ by $L(D, \mathcal{P}) - L(D, \mathcal{P} \setminus X)$;
return $\mathcal{P}$;
```

During the search we iteratively update the code table Hence, it may be that over time, previously included patterns start to harm compression once their role in covering the sequence is taken over by new, more specific, patterns. As such, they become redundant, and should be removed from $\mathcal{P}$.

To this end, we prune redundant patterns (see Algorithm 4) after each successful addition. During pruning, we iteratively consider each pattern $Y \in \mathcal{P}$ in order of insertion. If $\mathcal{P} \setminus X$ improves the total encoded size, we remove $X$ from $\mathcal{P}$. As testing every pattern in $\mathcal{P}$ at every successful addition may become rather time-consuming, we use a simple heuristic: if the total gain of the windows of $X$ is higher than the cost of $X$ in the code table we do not test $X$.

After SQS-CANDIDATES considered every pattern of $\mathcal{F}$, we run one final round of pruning without this heuristic. Finally, we order the patterns in $\mathcal{P}$ by $L(D, \mathcal{P}) - L(D, \mathcal{P} \setminus X)$. That is, by the impact on the total encoded length when removing $X$ from $\mathcal{P}$. This order tells us which patterns in $\mathcal{P}$ are most important.

Let us consider the execution time needed by SQS-CANDIDATES. Ordering patterns can be done in $O(|\mathcal{F}|\,\|D\|)$ time. Computing $L(D, \mathcal{P} \cup X)$ can be done in $O(|\mathcal{P}|\,\|D\|) \subseteq O(|\mathcal{F}|\,\|D\|)$ time.

**Algorithm 4:** Prune$(\mathcal{P}, D, \mathit{full})$

```
input  : pattern set $\mathcal{P}$, database $D$, boolean variable $\mathit{full}$,
         **false** for heuristic scan, **true** for complete scan
output : pruned pattern set $\mathcal{P}$;
foreach $X \in \mathcal{P}$ do
    $CT \leftarrow$ code table corresponding to Sqs$(D, \mathcal{F})$;
    $CT' \leftarrow$ code table obtained from $CT$ by deleting $X$;
    $g \leftarrow \sum_{w=(i,j,X,k) \in A} gain(w)$;
    if $\mathit{full}$ or $g < L(CT) - L(CT')$ then
        if $L(D, \mathcal{P} \setminus X) < L(D, \mathcal{P})$ then $\mathcal{P} \leftarrow \mathcal{P} \setminus X$;
return $\mathcal{P}$;
```

Pruning can be done in $O(|\mathcal{P}|^2 \|D\|) \subseteq O(|\mathcal{F}|^2 \|D\|)$. Combined, this gives us a total time complexity of $O(|\mathcal{F}|^3 \|D\|)$. In practice, the algorithm is much faster, however, as first, due to MDL the code tables remain small, and hence $|\mathcal{P}| \ll |\mathcal{F}|$, second, the execution time of Sqs is typically faster than $O(|\mathcal{P}| \|D\|)$, and third, the pruning heuristic further reduces the computational burden.

## 4.2 Directly Mining Good Code Tables

The Sqs-Candidates algorithm requires a collection of candidate patterns to be materialised, which in practice can be troublesome; the well-known pattern explosion may prevent patterns to be mined at as low thresholds as desired. In this section we propose an alternative strategy for discovering good code tables directly from data. Instead of filtering a pre-mined candidate set, we now discover candidates on the fly, considering only patterns that we expect to optimise the score given the current alignment.

To illustrate the general idea, consider that we have a current set of patterns $\mathcal{P}$. We iteratively find patterns of form $XY$, where $X, Y \in \mathcal{P} \cup \Omega$ producing the lowest $L(D, \mathcal{P} \cup \{XY\})$. We add $XY$ to $\mathcal{P}$ and continue until no gain is possible. Unfortunately, as testing each combination takes $O((|\mathcal{P}| + |\Omega|)^2(|\mathcal{P}| + 1) \|D\|)$ time, we cannot do this exhaustively and exactly within reasonable time.

Hence, we resort to heuristics.

To guarantee the fast discovery of good candidates, we design a heuristic algorithm that, given a pattern $P$, will find a pattern $PQ$ of high expected gain in only $O(|\mathcal{P}| + |\Omega| + \|D\|)$ time.

Our first step is to demonstrate that if we take $N$ active windows of $P$, and $N$ active windows of $Q$, and convert them into $N$ active windows of $PQ$, the difference in total encoded length can be computed in constant time.

Proposition 3. *Fix a database $D$ and an alignment $A$. Let $P$ and $Q$ be two patterns. Let $V = \{v_1, \ldots, v_N\}$ and $W = \{w_1, \ldots, w_N\}$ be two set of candidate windows for $P$ and $Q$, respectively. Assume that either $P$ ($Q$) is a singleton or each $v_i$ ($w_i$) occurs in $A$. Assume that $v_i$ and $w_i$ occur in the same sequence and write $v_i = (a_i, b_i, P, k_i)$ and $w_i = (c_i, d_i, Q, k_i)$. Assume that $b_i < c_i$. Write $R = PQ$ and let*

$$U = \{(a_1, d_1, R, k_1), \ldots, (a_N, d_N, R, k_N)\} \quad .$$

*Assume that $U$ has no overlapping windows and has no overlapping windows with $A \setminus (V \cup W)$. Then the difference*

$$L(D, A \cup U \setminus (V \cup W)) - L(D, A)$$

*depends only $N$, $gaps(V)$, $gaps(W)$, and $gaps(U)$ and can be computed in constant time from these values.*

The conditions given in Proposition 3 are needed so that $A \setminus (V \cup W)$ is a proper alignment. We denote the aforementioned difference by $\mathit{diff}(V, W, U; A, D)$. Note that this difference partly depends on $A$ and $D$. However, since we keep these fixed in the proposition they only contribute constant terms. Further note that $U$ should not overlap with $A \setminus (V \cup W)$. We will address this limitation later. We should point out that in practice we do not keep lists of $U$, $V$, and $W$, but instead exploit the gap counts and number of windows, as this is sufficient for computing the difference.

Now that we have a way of computing the gain of using windows for $PQ$, we need to know which windows to use in the alignment. The following proposition suggest that we should pick the windows with the shortest length.

Proposition 4. *Let $D$ be a database and $A$ be an alignment. Let $v = (i, j, X, k) \in A$. Assume that there exists a window $S_l[a, b]$ containing $X$ such that $w = (a, b, X, l)$ does not overlap with any window in $A$ and $b - a < j - i$. Then $A$ is not an optimal alignment.*

This proposition gives us an outline of the heuristic. We start enumerating minimal windows of $PQ$ from shortest to largest. At each step we compute the score using Proposition 3, and among these scores we the pick optimal one.

We cannot guarantee linear time if we consider each $Q$ individually. Instead, we scan for all candidates simultaneously. In addition, to guarantee linear time we consider only active windows of $P$ and $Q$, and do not consider singletons occurring in the gaps. The scan starts by finding all the active windows (ignoring singletons in gaps) of $P$. We then continue by scanning the patterns occurring after each $P$. We interleave the scans in such a way that the new minimal windows are ordered, from shortest to longest. We stop the scan after we find next occurrence of $P$ or the end of the sequence.

There are two constraints that we need to take into account. When enumerating minimal windows of $PQ$ we need to make sure that we can add them to the alignment. That is, a new minimal window cannot intersect with other new minimal windows, and the only windows it may intersect in the alignment are the two windows from which it was constructed. The first constraint can only happen when $Q = P$, in which case we simply check if the adjacent scans have already used these two instances of $P$ for creating a minimal window for pattern $PP$. To guarantee the second constraint, we need to delete the intersecting windows from the alignment. We estimate the effect of deleting $w$ by adding $gain(w)$ (computed from the current alignment) to the score. The pseudo-code for calculating this estimate is given as Algorithm 5.

Proposition 5. Estimate$(P, \emptyset, D)$ *returns a pattern with optimal score.*

Next, let us consider the computational complexity of this approach. The initialisation in Estimate can be done in $O(|\Omega| + |\mathcal{P}| + \|D\|)$, where $\mathcal{P}$ are the current non-singleton patterns. After selecting the next window, each step in the main loop can be done in constant time. The only non-trivial step is picking the next smallest window. However, since the window lengths are integers smaller or equal than $\|D\|$, we can store the candidates into an array of lists, say $N_d$, where $N_d$ contains the windows of length $d$. Finding the next window may take more than a constant time since we need to find the next non-empty list $N_d$ but such search may only contribute $\|D\|$ checks in total. Since we stop after encountering $P$, every event is visited only twice at maximum, hence the running time for Estimate is $O(|\Omega| + |\mathcal{P}| + \|D\|)$.

The actual search algorithm, Sqs-Search, calls Estimate for each pattern $P$. The algorithm, given as Algorithm 6, continues by sorting the obtained patterns based on their estimated scores and attempts to add them into encoding in the same fashion as in

**Algorithm 5:** Estimate$(P, A, D)$. Heuristic for finding a pattern $X$ used by the current encoding with a low $L(D, A \cup PX)$

```
input  : database $D$, current alignment $A$, pattern $P \in CT$
output : pattern $PX$ with $X \in CT$ and a low $L(D, A \cup PX)$
foreach $X \in CT$ do $V_X \leftarrow \emptyset$; $W_X \leftarrow \emptyset$; $U_X \leftarrow \emptyset$; $d_X \leftarrow 0$;
$T \leftarrow \emptyset$;
foreach occurrence $v$ of $P$ in the encoding (ignoring gaps) do
    $(a, b, P, k) \leftarrow v$;
    $d \leftarrow$ the end index of the active window following $v$;
    $t \leftarrow (v, d, 0)$; $l(t) \leftarrow d - a$;
    add $t$ into $T$;
while $T$ is not empty do
    $t \leftarrow \arg\min_{u \in T} l(u)$;
    $(v, d, s) \leftarrow t$; $a \leftarrow$ first index of $v$;
    $w = (c, d, X, k) \leftarrow$ active window of a pattern ending at $d$;
    if $X = P$ and (event at $a$ or $d$ is marked) then
        delete $t$ from $T$;
        continue;
    if $S_k[a, d]$ is a minimal window of $PX$ then
        add $v$ into $V_X$;
        add $w$ into $W_X$;
        add $(a, d, PX, k)$ into $U_X$;
        $d_X \leftarrow \min(\mathit{diff}(V, W, U; A) + s, d_X)$;
        if $|X| > 1$ then $s \leftarrow s + \mathit{gain}(w)$;
        if $X = P$ then
            mark the events at $a$ and $d$;
            delete $t$ from $T$;
            continue;
    if $w$ is the last window in the sequence then
        delete $t$ from $T$;
    else
        $d \leftarrow$ the end index of the active window following $w$;
        update $t$ to $(v, d, s)$ and $l(t)$ to $d - a$;
return $PX$ with the lowest $d_X$;
```

Sqs-Candidates. After each successful addition of pattern $X$, we scan for the gap events occurring in the active windows of $X$, and test patterns obtained from $X$ by adding a gap event as intermediate event. The scan can be done in $O(\|D\|)$ time, and in theory we may end up testing $|\Omega|\,(|X| - 1)$ patterns. In practice, the number is much smaller since accepted patterns typically have small gaps. If any of these patterns in successfully added we repeat this procedure in a recursive fashion. In practice, testing $X$ is relatively fast, and the total computational complexity is dominated by Estimate.

## 5. RELATED WORK

Discovering frequent sequential patterns is an active research topic. Unlike for itemsets, there are several definitions for frequent sequential patterns. The first approach counts the number of sequences containing a pattern [24]. In such setup, having one long sequence do not make sense. In the second approach we count multiple occurrences within a sequence. This can be done by sliding a window [13] or counting disjoint minimal windows [10].

Mining general episodes, patterns where the order of events are specified by a DAG is surprisingly hard. For example, testing whether a sequence contains a pattern is **NP**-complete [19]. Consequently, research has focused on mining subclasses of episodes, such as, episodes with unique labels [1, 14], and strict episodes [20].

**Algorithm 6:** Sqs-Search$(D)$

```
input  : database $D$
output : significant patterns $\mathcal{P}$
$\mathcal{P} \leftarrow \emptyset$; $A \leftarrow$ Sqs$(D, \emptyset)$;
while changes do
    $\mathcal{F} \leftarrow \emptyset$;
    foreach $P \in CT$ do add Estimate$(P, A, D)$ to $\mathcal{F}$;
    foreach $X \in \mathcal{F}$ ordered by the estimate do
        if $L(D, \mathcal{P} \cup X) < L(D, \mathcal{P})$ then
            $\mathcal{P} \leftarrow$ Prune$(\mathcal{P} \cup X, D,$ false$)$;
        if $X$ is added then test recursively $X$ augmented with
            events occurring in the gaps;
$\mathcal{P} \leftarrow$ Prune$(\mathcal{P}, D,$ true$)$;
order patterns $X \in G$ by $L(D, \mathcal{P}) - L(D, \mathcal{P} \setminus X)$;
return $\mathcal{P}$;
```

Discovering statistically significant sequential patterns is a surprisingly understudied topic. One reason is that unlike for itemsets, computing an expected frequency under a null-hypothesis is very complex. Using independence assumption as a null-hypothesis has been suggested in [7, 18] and a Markov-chain model has been suggested in [6]. In [1] the authors use information theory-based measure to determine which edges to include in a general episode.

Summarising sequences using segmentation is a well-studied topic. The goal in segmentation is to divide the sequence in large segments of homogenous regions whereas our goal is to find a set of compact patterns that occur significantly often. For an overview in segmentation, see [4], and for a segmentation tool see [8].

Mannila and Meek [12] regard general episodes, as generative models for sequences. Their model generates short sequences by selecting a subset of events from an episode and select a random order compatible with the episode. They do not allow gaps and only one pattern is responsible for generating a single sequence. This is not feasible for our setup, where we may have long sequences and many patterns occurring in one sequence.

Sqs draws inspiration from the Krimp [23] and Slim [17] algorithms. Krimp pioneered the use of MDL for identifying good pattern sets; specifically, mining sets of itemsets that describe a transaction database well. As serial episodes are much more expressive than itemsets, we here need a much more elaborate encoding scheme, and in particular, a non-trivial approach for covering the data. For mining the patterns, Sqs-Candidates shares the greedy selection over an ordered set of candidates.

Smets and Vreeken recently gave the Slim algorithm [17] for directly mining Krimp code tables from data. With Sqs-Search we adopt a strategy that resembles Slim, by considering joins $XY$ of $X, Y \in CT$, and estimating the gain of adding $XY$ to $CT$. Whereas Slim iteratively searches for the best addition, for efficiency, Sqs-Search adopts a batch-wise strategy.

Lam et al. introduced GoKrimp [9] for mining sets of serial episodes. As opposed to the MDL principle, they use fixed length codes, and do not punish gaps within patterns—by which their goal is essentially to cover the sequence with as few patterns as possible, which is different from our goal of finding patterns that succinctly summarise the data. Bathoorn et al. [2] also cover greedily, and do not consider gaps at all.

## 6. EXPERIMENTS

We implemented our algorithms in C++, and provide the source code for research purposes, together with the considered datasets, as well as the generator for the synthetic data.[1] As candidates for Sqs-Candidates, we mined frequent serial episodes [10, 20] using disjoint minimal windows of maximal length 15, with minimal support thresholds as low as feasible—i.e. at the point where the number of patterns starts to explode. All experiments were executed single-threaded on a six-core Intel Xeon machine with 12GB of memory, running Linux.

In our experiments we consider both synthetic and real data. Table 1 shows the base statistics per dataset, i.e. number of distinct events, number of sequences, total number of events per database, and the total encoded length by the most simple code table $ST$.

*Synthetic Data.* First, we consider the synthetic *Indep*, *Plants-10*, and *Plants-50* datasets. Each consists of a single sequence of 10 000 events over an alphabet of 1 000. In the former, all events are independent, whereas in the latter two we planted resp. 10 and 50 patterns of 5 events 10 times each, with 10% probability of having a gap between consecutive events, but are independent otherwise.

Table 1 shows the results given by Sqs-Candidates and Sqs-Search. For the *Indep* dataset, while over 9 000 episodes occur at least 2 times, both methods correctly identify the data does not contain significant structure. Similar for *Plants-10* both methods correctly return the 10 planted patterns. *Plants-50* has a very high density of pattern symbols (25%), and hence poses a harder challenge. Sqs-Candidates and Sqs-Search identify resp. 47 and 46 patterns exactly, the remainder consisting of fragments of correct patterns. The imperfections are due to patterns being partly overwritten during the generation of the data.

*Real Data.* For the experiments on real data, in order to interpret the patterns, we consider text data. The events are the stemmed words from the text, with stop words removed. *Addresses* contains speeches of American presidents, *JMLR* consists of abstracts of papers from the Journal of Machine Learning Research website,[2] whereas *Moby* contains the novel 'Moby Dick' by Herman Melville.

Let us first consider the number of returned patterns, as shown in Table 1. We see that for all datasets small numbers of patterns are returned, in the order of 100s, two orders of magnitude less than the number of frequent patterns Sqs-Candidates considers.

When we consider the gains in compressed size, i.e. $\Delta L = L(D, ST) - L(D, CT)$, we see these few patterns in fact describe a lot of structure of the data; recall that 1 bit of gain corresponds to an increase of factor 2 in likelihood. We note Sqs-Search slightly outperforms Sqs-Candidates, which is due to the former being able to consider candidates of lower support without suffering from the pattern explosion.

The largest $\Delta L$ is recorded for *JMLR*, with almost 30k bits. This is not surprising, as the type of text, abstracts of machine learning papers, has a relatively small vocabulary—the use of which is quite structured, with many key phrases and combinations of words.

Table 2 depicts the top-10 most compressing patterns for *JMLR*, as found by Sqs-Search. Here, as $\Delta L$ we give the increase in bits the pattern would be removed from $CT$. Clearly, key machine learning concepts are identified, and importantly, the patterns are neither redundant, nor polluted with common words. In fact, in none of the $CT$s patterns incorrectly combine frequent events.

Further examples of patterns reported for *JMLR* include '*non neg matrix factor*', '*isotrop log concav distribut*', and '*reproduc[ing] kernel Hilbert space*'. For the presidential *Addresses*, we unsurprisingly see '*unit[ed] stat[es]*' and '*fellow citizen[s]*' as the top-2 patterns. An example of a pattern with many gaps (5.2 gap events, on average), we find the rather current '*economi[c] public expenditur[e]*'. From the *Moby Dick* novel we find the main antagonist's species, '*sperm whale*', and name, '*moby dick*', as well as the phrase '*seven hundr[ed] seventy seventh*' which occurs 6 times.

**Table 2: JMLR data. Top-10 patterns by Sqs-Search**

| patterns | $\Delta L$ | patterns | $\Delta L$ |
|---|---|---|---|
| 1. supp. vector machine | 850 | 6. large scale | 329 |
| 2. machine learning | 646 | 7. nearest neighbor | 322 |
| 3. state [of the] art | 480 | 8. decision tree | 293 |
| 4. data set | 446 | 9. neural network | 289 |
| 5. Bayesian network | 374 | 10. cross validation | 279 |

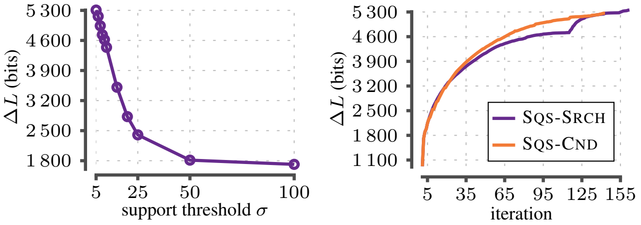

**Figure 2: *Addresses* dataset, gain in compression. (left) varying support thresholds for Sqs-Candidates. (right) Sqs-Candidates and Sqs-Search per accepted candidate.**

Next, we investigate our search strategies. First, in the left-hand plot of Fig. 2, for Sqs-Candidates on the *Moby* data, we show the gain in compression for different support thresholds. It shows that lower thresholds, i.e richer candidate sets, allow for (much) better models—though by the pattern explosion, mining candidates at low $\sigma$ can be infeasible.

Second, in the right-hand plot, we compare Sqs-Candidates and Sqs-Search, showing the gain in bits over $ST$ per candidate accepted into $CT$. It shows both search processes are efficient, considering patterns that strongly aid compression first. The slight dip of Sqs-Search around iteration 100 is due to its batch-wise search. At the expense of extra computation, an iterative search for the best estimated addition, like Slim [17] may find better models.

In these experiments, using these support thresholds, mining the candidates took up to 4 minutes, after which Sqs-Candidates took up to 15 minutes to order and filter the candidates. Sqs-Search resp. took 10, 18, and 91 minutes. As *Moby* has a large alphabet and is one long sequence, Sqs-Search has to consider many possible pattern co-occurrences.

## 7. DISCUSSION

The experiments show both Sqs-Candidates and Sqs-Search return high-quality models. By using synthetic data we showed that Sqs reveals the true patterns without redundancy, while further not picking up on spurious structure. Analysis of the results on the text data experiments show that key phrases are identified—combinations of words that may be interspersed with 'random' words in the data. Importantly, for all of the datasets, no noisy or redundant patterns are returned. As expected, the more structure a dataset exhibits, the better the attained compression.

With Sqs-Candidates we allow the user the freedom to provide a set of candidate serial episodes. In general, lower thresholds correspond to more candidates, more candidates correspond to a larger search space, and hence better models. Sqs-Search on the other hand, besides an any-time algorithm, is parameter-free, as it generates and tests patterns that are estimated to improve the score.

[1] http://adrem.ua.ac.be/sqs/

[2] http://jmlr.csail.mit.edu/

**Table 1: Basic statistics and results per dataset**

| | | | | | SQS-CANDIDATES | | | | SQS-SEARCH | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Dataset** | $\lvert\Omega\rvert$ | $\lvert D\rvert$ | $\lVert D\rVert$ | $L(D, ST)$ | $\sigma$ | $\lvert\mathcal{F}\rvert$ | $\lvert\mathcal{G}\rvert$ | $L(D, CT)$ | $\lvert\mathcal{G}\rvert$ | $L(D, CT)$ |
| Indep | 1 000 | 1 | 10 000 | 103 630 | 2 | 9 094 | 0 | 103 630 | 0 | 103 630 |
| Plants-10 | 1 000 | 1 | 10 000 | 103 340 | 2 | 11 957 | 10 | 100 629 | 10 | 100 629 |
| Plants-50 | 1 000 | 1 | 10 000 | 102 630 | 2 | 25 484 | 50 | 91 706 | 52 | 91 707 |
| Addresses | 5 295 | 56 | 62 066 | 685 593 | 5 | 15 506 | 138 | 680 287 | 155 | 680 236 |
| JMLR | 3 846 | 788 | 75 646 | 772 112 | 5 | 40 879 | 563 | 742 966 | 580 | 742 953 |
| Moby | 10 277 | 1 | 105 719 | 1 250 149 | 5 | 22 559 | 215 | 1 240 667 | 231 | 1 240 566 |

Both algorithms are fast, our prototype implementations taking only few minutes on the data here considered. The algorithms have many opportunities for parallelisation: candidates can be estimated or evaluated individually, as can the scanning for minimal windows.

MDL does not provide a free lunch. First of all, although highly desirable, it is not trivial to bound the score. While for Kolmogorov complexity we know this is incomputable, for our models we have no proof one way or another. Furthermore, although MDL gives a principled way to construct an encoding, this involves many choices that determine what structure is rewarded. As such, we do not claim our encoding is suited for all goals, nor that it cannot be improved.

Future work includes the extension of SQS for parallel and general episodes—which surpass serial episodes in expressiveness. Although seemingly opposed to MDL (why describe the same thing twice?) allowing patterns to overlap may provide more succinct summarisations. Last, but not least, we are interested in applying the SQS code tables for clustering and anomaly detection.

## 8. CONCLUSION

In this paper we employed the MDL principle to mine sets of sequential patterns that summarise the data well. In particular, we formalised how to encode sequential data using set of patterns, and use the encoded length as a quality measure. As search strategy for good models, we adopt two approaches. The first algorithm, SQS-CANDIDATES, selects a good pattern set from a large candidate set, while SQS-SEARCH is a parameter-free any-time algorithm that discovers good pattern sets directly from the data. Experimentation on synthetic and real data showed both methods to efficiently discover small, non-redundant sets of informative patterns.

And that's the long and the short of it.

## Acknowledgements

Nikolaj Tatti and Jilles Vreeken are supported by Post-Doctoral Fellowships of the Research Foundation – Flanders (FWO).

## 9. REFERENCES

[1] A. Achar, S. Laxman, R. Viswanathan, and P. S. Sastry. Discovering injective episodes with general partial orders. *Data Min. Knowl. Disc.*, 2011.

[2] R. Bathoorn, A. Koopman, and A. Siebes. Reducing the frequent pattern set. In *ICDM-Workshop*, pages 1–5, 2006.

[3] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience New York, 2006.

[4] S. Dzeroski, B. Goethals, and P. Panov, editors. *Inductive Databases and Constraint-Based Data Mining*. Springer, 2010.

[5] P. Grünwald. *The Minimum Description Length Principle*. MIT Press, 2007.

[6] R. Gwadera, M. J. Atallah, and W. Szpankowski. Markov models for identification of significant episodes. In *SDM*, pages 404–414, 2005.

[7] R. Gwadera, M. J. Atallah, and W. Szpankowski. Reliable detection of episodes in event sequences. *Knowl. Inf. Sys.*, 7(4):415–437, 2005.

[8] J. Kiernan and E. Terzi. EventSummarizer: a tool for summarizing large event sequences. In *EDBT*, pages 1136–1139, 2009.

[9] H. T. Lam, F. Mörchen, D. Fradkin, and T. Calders. Mining compressing sequential patterns. In *SDM*, 2012.

[10] S. Laxman, P. S. Sastry, and K. P. Unnikrishnan. A fast algorithm for finding frequent episodes in event streams. In *KDD*, pages 410–419, 2007.

[11] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer, 1993.

[12] H. Mannila and C. Meek. Global partial orders from sequential data. In *KDD*, pages 161–168, 2000.

[13] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Disc.*, 1(3):259–289, 1997.

[14] J. Pei, H. Wang, J. Liu, K. Wang, J. Wang, and P. S. Yu. Discovering frequent closed partial orders from strings. *IEEE TKDE*, 18(11):1467–1481, 2006.

[15] J. Rissanen. Modeling by shortest data description. *Annals Stat.*, 11(2):416–431, 1983.

[16] D. Salomon and G. Motta. *Handbook of Data Compression*. Springer, 2009.

[17] K. Smets and J. Vreeken. SLIM: Directly mining descriptive patterns. In *SDM*, pages 1–12. SIAM, 2012.

[18] N. Tatti. Significance of episodes based on minimal windows. In *ICDM*, pages 513–522, 2009.

[19] N. Tatti and B. Cule. Mining closed episodes with simultaneous events. In *KDD*, pages 1172–1180, 2011.

[20] N. Tatti and B. Cule. Mining closed strict episodes. *Data Min. Knowl. Disc.*, 2011.

[21] N. Vereshchagin and P. Vitanyi. Kolmogorov's structure functions and model selection. *IEEE TIT*, 50(12):3265– 3290, 2004.

[22] J. Vreeken and A. Siebes. Filling in the blanks: Krimp minimisation for missing data. In *ICDM*, pages 1067–1072, 2008.

[23] J. Vreeken, M. van Leeuwen, and A. Siebes. KRIMP: Mining itemsets that compress. *Data Min. Knowl. Disc.*, 23(1):169–214, 2011.

[24] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. *ICDE*, 0:79, 2004.

# APPENDIX

## A. PROOFS

PROOF OF PROPOSITION 1. Assume opposite: there is a window $(i, j, X, k) \in A$ such that $W = S_k[i, j]$ is not an minimal window for $X$. Let $S_k[a, b]$ be a minimal window of $X$ in $W$. Let $A'$ be an alignment in which we replace $(i, j, X, k)$ with $(a, b, X, k)$. Note that $usage(Y)$ remains constant for any pattern $Y$. In addition, $gaps(Y)$ remains also constant for any pattern $Y \neq X$. Since $b - a < j - i$, we see that $gaps(X) < gaps(Y)$. A straightforward computation shows that $L(CT, D)$ is a monotonic function of $gaps(X)$. Hence, the encoding length of $A'$ is lower than of $A$, which contradicts the optimality of $A$. □

PROOF OF PROPOSITION 2. Let

$$
\begin{aligned}
&const = \\
&\quad L_{\mathbb{N}}(|D|) + \sum_{S \in D} L_{\mathbb{N}}(|S|) + \sum_{s \in \Sigma} supp(s \mid D) L(code_p(s)) \quad .
\end{aligned}
$$

The first term in the definition of $gain$ will introduce the correct number of usages of non-singleton patterns. The second and the third terms correspond to the length of the gap stream. Finally, since for $s \in \Omega$,

$$
\begin{aligned}
usage(s) &= supp(s \mid D) - \sum_{s \in X} usage(X) \\
&= supp(s \mid D) - \sum_{s \in X} |\{(i, j, X, k) \in A\}| \quad ,
\end{aligned}
$$

the fourth term will correctly reduce the singleton usages. □

PROOF OF PROPOSITION 3. We will assume that $P \neq Q$, the treatment for the case $P = Q$ is almost equivalent. Let $A' = A \cup A \setminus (V \cup W)$. The usages in $A'$ remain the same except for $P$, $Q$, and $R$: Usages for $P$ and $Q$ are reduced by $N$ and usage of $R$ is increased by $N$. In addition, the total usage $usage(CT(A')) = usage(CT(A)) - N$ is reduced by $N$.

To compute the difference in the pattern stream $C_p$ we first compute the difference between the code lengths for patterns $P$, $Q$, and $R$ using new usages but old total usage. We have $usage(CT(A'))$ with incorrect total usage. To compensate the difference in total usage we add

$$
usage(CT(A'))(\log usage(CT(A')) - \log usage(CT(A))) \quad .
$$

The gaps $gaps(P)$ and $gaps(Q)$ are decreased by $gaps(V)$ and $gaps(W)$ under the new encoding. Also, $gaps(R)$ is increased by $gaps(U)$. The remaining gaps remain the same. Consequently we can compute the difference in the gap stream $C_g$ in constant time. Hence, we can compute the difference $L(D \mid A') - L(D \mid A)$ in constant time.

Encoding the code table will change since it depends on total usage of non-singleton patterns. In addition, we may delete $P$ or $Q$ from the code table if their usage counts go to zero (or add $R$ it its usage count was 0). We see from the definition of $L(CT)$ that in total 6 terms may change. Consequently, we can compute the difference $L(CT(A')) - L(CT(A))$ in constant time. □

PROOF OF PROPOSITION 4. Let $A' = A \cup \{w\} \setminus \{v\}$. The usage counts in $A$ and in $A'$ are the same. Thus $L(C_p \mid CT(A')) = L(C_p \mid CT(A))$. The gaps also remain constant except for $X$, in which case, the $gaps(X)$ is reduced by $i - j - (b - a)$. A straightforward calculation implies that $L(C_p \mid CT(A')) < L(C_p \mid CT(A))$ and $L(X, CT(A')) < L(X, CT(A))$. This implies that $L(D, A') < L(D, A)$. □

PROOF OF PROPOSITION 5. Let $PX$ be the optimal pattern. Since alignment is empty, we do not need to compensate for overlapping windows and the encoding lengths we are computing are accurate. The algorithm enumerates windows from smallest to largest. We can use Proposition 4 to see that there will be a point where $U_X$ will contain the optimal alignment, yielding a correct optimal $d_X$. Consequently, ESTIMATE will return $PX$. □