Claims to work on multi sensor multi variate time series, but no details of the pattern finding algo given. C++ implementation is available.

Keeping it Short and Simple: Summarising Complex Event Sequences with Multivariate Patterns

Roel Bertens
Department of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands
R.Bertens@uu.nl

Jilles Vreeken
Max Planck Institute for Informatics and
Saarland University
Saarbrücken, Germany
jilles@mpi-inf.mpg.de

Arno Siebes
Department of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands
A.P.J.M.Siebes@uu.nl

ABSTRACT

We study how to obtain concise descriptions of discrete multivariate sequential data. In particular, how to do so in terms of rich multivariate sequential patterns that can capture potentially highly interesting (cor)relations between sequences. To this end we allow our pattern language to span over the domains (alphabets) of all sequences, allow patterns to overlap temporally, as well as allow for gaps in their occurrences.

We formalise our goal by the Minimum Description Length principle, by which our objective is to discover the set of patterns that provides the most succinct description of the data. To discover high-quality pattern sets directly from data, we introduce DITTO, a highly efficient algorithm that approximates the ideal result very well.

Experiments show that DITTO correctly discovers the patterns planted in synthetic data. Moreover, it scales favourably with the length of the data, the number of attributes, the alphabet sizes. On real data, ranging from sensor networks to annotated text, DITTO discovers easily interpretable summaries that provide clear insight in both the univariate and multivariate structure.

1. INTRODUCTION

Most real sequential data is multivariate, as when we measure data over time, we typically do so over multiple attributes. Examples include sensors in a network, frequency bands in seismic or ECG data, transaction records, and annotated text. In this paper we investigate how to succinctly summarise such data in terms of those patterns that are most characteristic for the data.

To capture these characteristics, our pattern language needs to be rich, i.e., patterns may span multiple aligned sequences (attributes) in which no order between these attributes is assumed and occurrences may contain gaps. For example, if we consider a sensor network, a pattern could be a characteristic set of values for multiple attributes at one point in time, or, more complex, a specific value for one sensor, temporally followed by certain readings of one or more other sensors. That is, patterns that are able to identify associations and correlations between one or multiple attributes.

Having such a rich pattern language has as immediate consequence that the well-known frequent pattern explosion hits particularly hard. Even for trivial data sets one easily finds enormous numbers of frequent patterns, even at modest minimal frequency thresholds [27]; the simplest synthetic dataset we consider contains only five true patterns, yet the *lower bound* on the number of frequent patterns is more than a billion.

Clearly, returning collections of such magnitude is useless, as they cannot be inspected in any meaningful way. Rather, we want a small set of such patterns that collectively describe the data well. To find such sets, we employ the Minimum Description Length (MDL) principle [8]. This approach has a proven track record in the domain of transaction data mining [32, 24] and has also been successfully applied to, e.g. sequential data [28, 10]. Because complex multivariate sequential data is ubiquitous and correlation between multiple sequences can give much insight to domain experts, here we take this research further by defining a framework to summarise this data, while able to efficiently deal with the enormous space of frequent patterns. Note that, all real-valued time series can easily be discretised to fit our framework, for example using SAX [13].

The humongous number of frequent patterns makes it intractable to discover a small set of characteristic patterns by post-processing the set of all frequent patterns; all the more because the MDL objective function on pattern sets is neither monotone nor sub-modular. Hence we introduce a heuristic algorithm, called DITTO,[1] that mines characteristic sets of patterns directly from the data. In a nutshell, DITTO mines good models by iteratively adding those patterns to the model that maximally reduce redundancy. That is, it iteratively considers the current description of the data and searches for patterns that most frequently occur one after the other, and considers the union of such pairs as candidates to add to the model. As such, DITTO focusses on exactly those patterns that are likely to improve our score, pruning large parts of the search space, and hence only needs seconds to mine high-quality summaries.

Our extensive empirical evaluation of DITTO shows it ably discovers succinct summaries that contain the key patterns of the data at hand, and, importantly, that it does not pick up on noise. An example of what we discover includes the following. When applied to the novel Moby Dick by Herman Melville, on the topic of whaling, the summary DITTO discovers consists of meaningful non-trivial linguistic patterns including "⟨noun⟩ of the ⟨noun⟩", e.g. "Man of the World", and "the ⟨noun⟩ of", as used in "the ship of (somebody)". These summaries hence give clear insight in the writing style of the author(s). Besides giving direct insight, the summaries we discover have high downstream potential. One could, for example, use them for comparative analysis [4]. For example, for text identifying similarities and differences between authors, for sensor networks detecting and describing concept drift over time, and characterising differences between patients. Such analysis is left for future work, as we first have to be able to efficiently discover high quality pattern-based summaries from multivariate sequential data. That is what we focus on in this paper.

[1] The ditto mark (") is used in lists to indicate that an item is repeated, i.e., a multivariate pattern.

The remainder of this paper is organised as follows. We first cover preliminaries and introduce notation in Section 2. In Section 3 we formally introduce the problem. Section 4 gives the details of the DITTO algorithm. We discuss related work in Section 5, and empirically evaluate our score in Section 6. We round up with discussion and conclusions in Sections 7 and 8, respectively.

2. PRELIMINARIES AND NOTATION

As data type we consider databases D of $|D|$ multivariate event sequences $S \in D$, all over attributes A. We assume that the set of attributes is indexed, such that A_i refers to the i^{th} attribute of D.

A multivariate, or *complex*, event sequence S is a bag of $|A|$ univariate event sequences, $S = \{S_1, \ldots, S_{|A|}\}$. An event sequence $S_i \in \Omega_i^n$ is simply a sequence of n events drawn from discrete domain Ω_i, which we will refer to as its *alphabet*. An event is hence simply an attribute–value pair. That is, the j^{th} event of S^i corresponds to the value of attribute A_i at time j. We will write Ω for the union over these attribute alphabets, i.e. $\Omega = \cup_{i \in |A|} \Omega_i$.

By $||S||$ we indicate the number of events in a multivariate sequence S, and by $t(S)$ the length of the sequence, i.e. the number of time steps. We will refer to the set of events at a single time step j as a *multi-event*, writing $S[j]$ for the j^{th} multi-event of S. To project a multivariate event sequence S onto an individual attribute, we write S^j for the univariate event sequence on the j^{th} attribute, and analogously define D^j. For completeness, we define $||D|| = \sum_{S \in D} ||S||$ for the total number of events, and $t(D) = \sum_S t(S)$ for the total number of time steps in D.

Our framework fits both categorical and transaction (item set) data. With categorical data the number of events at each time step is equal to the number of attributes. For simplicity, w.l.o.g., we consider categorical data in the remainder.

As patterns we consider partial orders of multi-events, i.e. sequences of multi-events, allowing for gaps between time steps in their occurrences. By $t(X)$ we denote the *length* of a pattern X, i.e. the number of time steps for which a pattern X defines a value. In addition, we write $||X||$ to denote the *size* of a pattern X, the total number of values it defines, i.e. $\sum_{X[i] \in X} \sum_{x \in X[i]} 1$. For example, in Figure 1 it holds that $||X|| = 4$ and $t(X) = 3$.

We say a pattern X is *present* in a sequence $S \in D$ of the data when there exists an interval $[t_{start}, \cdots, t_{end}]$ in which all multi-events $X[i] \in X$ are contained in the order specified by X, allowing for gaps in time. That is, $\forall_{X[i] \in X} \exists_{j \in [start,end]} X[i] \subseteq S[j]$, and $k \geq j + 1$ for $X[i] \subseteq S[j]$ and $X[i+1] \subseteq S[k]$. A singleton pattern is a single event $e \in \Omega$ and $\mathcal{P}$ is the set of all non-singleton patterns. A minimal window for a pattern is an interval in the data in which a pattern occurs which can not be shortened while still containing the whole pattern [26]. In the remainder of this paper when we use the term pattern occurrence we always expect it to be a minimal window. Further, we use $occs(X, S)$ for the disjoint set of all occurrences of X in S. Figure 1 shows examples of two categorical patterns occurring both with and without gaps.

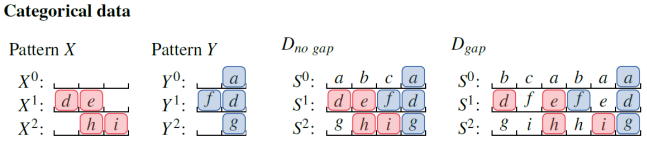

Figure 1: Patterns X and Y occurring with (D_{gap}) and without ($D_{no\ gap}$) gaps in the data.

Our method operates on categorical data. To find patterns in continuous real-valued time series we first have to discretise it. SAX [13] is a celebrated approach for doing so, and we will use it in our experiments – though we note that any discretisation scheme can be employed. This ordinal data can either be represented by absolute values per time step, or by relative values that represent the difference between each pair of subsequent values. These relative values describe the changes in the data rather than the exact values – by which different types of patterns can be discovered.

3. THEORY

In this section we give a brief introduction to the MDL principle, we define our objective function, and analyse the complexity of the optimization problem.

3.1 MDL Introduction

The Minimum Description Length principle (MDL) [8] is a method for inductive inference that aims to find the best model for the data. It is based on the insight that any regularity in the data can be used to compress the data and the more we can compress, the more regularity we have found. More formally, given a set of models $\mathcal{M}$, the best model $M \in \mathcal{M}$ is the one that minimises

$$L(M) + L(D \mid M)$$

where $L(M)$ is the length of the description of M, and $L(D \mid M)$ is the length of the description of the data.

To use MDL, we have to define how our model looks and how we can use it to describe the data. That is, we need to be able to encode and decode the data using our model.

3.2 MDL for Multivariate Sequential Data

As models for our data we consider code tables (CT) [32, 28]; these are simple four-column look-up tables (or, dictionaries) between patterns on the left hand side, and associated codes on the right hand side. In this paper we consider three types of codes, i.e. pattern codes, gap codes and fill codes. We will explain the use of these below, in Section 3.2.2 and Example 2.

Loosely speaking, whenever we read a pattern code $code_p(X \mid CT)$ we know we can start to decode the events of pattern X. A fill code $code_f(X \mid CT)$ tells us we can decode the next time step of pattern X, whereas a gap code $code_g(X \mid CT)$ tells us there is a gap in this occurrence of pattern X. To fill such a gap we read the next pattern code. Note that, our approach thus allows for patterns to interleave. For readability, we do not write CT wherever clear from context. To make sure we can encode any multivariate event sequence over Ω, we require that a code table at least contains all singleton events. Next, we will describe how to use these code tables to cover and encode a dataset. Further, we define how to compute $L(CT)$ and $L(D \mid CT)$ and we conclude with our formal problem definition.

3.2.1 Covering

A cover determines where we use each pattern when encoding a dataset and also where gaps in patterns occur. More formally, a cover $\mathcal{C}$ defines the set of all pattern occurrences that are used to cover the complete dataset. Therefore, if one of the events from pattern occurrence o is already covered by another pattern occurrence $o' \in \mathcal{C}$ we say that the occurrence o *overlaps* with the current cover $\mathcal{C}$, i.e. $\mathcal{C} \cap o \neq \emptyset$, which we do not allow.

As we are after the minimal description, we want to use optimal codes. Clearly, the more often a code is used, the shorter it should be. This, however, creates a non-trivial connection between how often a pattern is used to describe part of the data, and its code length. This, in turn, makes the complete optimisation process, and in particular

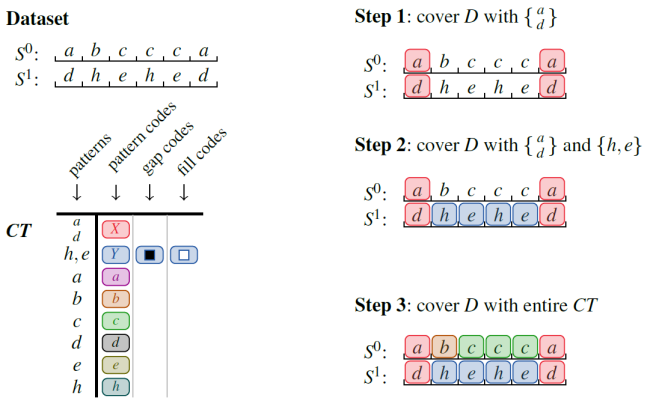

Figure 2: Sample data D, code table CT, and cover $\mathcal{C}$.

that of finding a good cover of the data given a set of patterns non-trivial. For the univariate case, we showed [28] that when we know the optimal code lengths we can determine the optimal cover by dynamic programming, and that we can approximate the optimal cover by iteratively optimising the code lengths and the cover. Although computationally expensive, for univariate data this strategy works well. For multivariate data, naively seen the search space grows enormously. More importantly, it is not clear how this strategy can be generalised, as now the usages of two patterns are allowed to temporally overlap as long as they do not cover the same events. We therefore take a more general and faster greedy strategy.

To cover a dataset we need the set of patterns from the first column of a code table CT, which we will refer to as PS (Patterns and Singletons). In Algorithm 1 we specify our COVER algorithm, which iterates through PS in a pre-specified order that we will detail later. For each pattern X the algorithm greedily covers all occurrences that do not overlap with already covered data, and have fewer than $t(X)$ gaps. To bound the number and size of gaps we only allow occurrences where the number of gap positions is smaller than the length of the pattern itself. This avoids gap codes becoming cheaper than fill codes, by which occurrences with more gaps would be preferred over compact occurrences. This process continues until all data is covered. In Figure 2 we show in 3 steps how a PS is used to cover the example dataset.

Example 1. In Algorithm 1, we start with an empty cover, corresponding to an uncovered dataset as in the top left of Fig 2. Next, we cover the data using the first pattern from the PS (line 1). For each occurrence of the pattern $\{\frac{a}{d}\}$ (line 2), we add it to our cover (line 4) when it does not overlap previously added elements and has a limited number of gaps (line 3). In Step 1 of Fig 2 we show the result of covering the data with the pattern $\{\frac{a}{d}\}$. We continue to do the same for the next pattern $\{h, e\}$ in CT. This gives us the cover as shown in Step 2. With only the singleton patterns left to consider in CT there is only one way to complete our cover by using them for all so far uncovered events. Now all events in the dataset are covered, thus we can break out of our loop (line 6) and return the final cover of Step 3 in Fig. 2 (line 7).

3.2.2 Encoding

A cover $\mathcal{C}$ specifies how the data will be encoded. Conceptually, we can decompose it into a pattern code stream and a gap code stream. The pattern stream C_p contains a sequence of pattern codes, $code_p(X)$ for pattern $X \in CT$, used to describe the data, whereas the gap code stream C_g consists of codes $code_g(X)$ and $code_f(X)$

Algorithm 1 The COVER Algorithm

Input: A sequence S, a set of patterns PS, and $\mathcal{C} = \emptyset$
Output: A cover $\mathcal{C}$ of $S \in D$ using PS

1: **for** each $X \in PS$ in order **do** // see Sec. 4 for order
2: **for all** $o \in occs(X, S)$ **do** // all occurrences of X
3: **if** $\mathcal{C} \cap o = \emptyset$ **and** $t(o) < 2t(X)$ **then**
4: $\mathcal{C} \leftarrow \mathcal{C} \cup o$
5: **if** $\mathcal{C}$ completely covers S **then**
6: **break**
7: **return** $\mathcal{C}$

indicating whether and where a pattern instance contains gaps or not.

To encode a dataset we traverse our dataset from left-to-right and top-to-bottom and each time we encounter a new pattern in our cover we add the code for this pattern to C_p. When moving to the next multi-event in the dataset we add for each pattern, that we already encountered but not yet completely passed by, a gap or fill code to C_g. We choose a gap code for a pattern if the current multi-event does not contain any event from the pattern or a fill code if it does. We are finished when we have encoded all multi-events in the data.

Example 2. In Figure 3 we show how a cover is translated to an encoding. To encode the first multi-event we first add $code_p(a)$ and then $code_p(Y)$ to C_p. For the second multi-event we add $code_p(X)$ to C_p and $code_g(Y)$ to C_g, because event e from pattern Y does not yet occur in this multi-event. For the third multi-event we add $code_p(a)$ to C_p, and $code_g(X)$ and $code_f(Y)$ to C_g. Code $code_g(X)$ marks the gap for pattern X and $code_f(Y)$ indicates that event e from pattern Y does occur in this multi-event. The fourth multi-event results in the addition of $code_p(Y)$ to C_p and $code_f(X)$ to C_g, where the latter marks the presence of event b from pattern X in this multi-event. For the last multi-event we add $code_p(a)$ to C_p and $code_f(Y)$ to C_g, which completes the encoding.

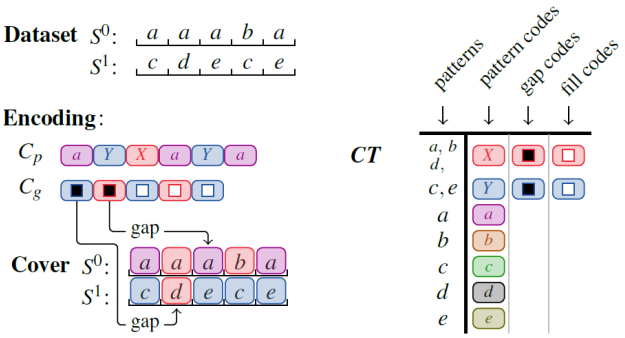

Figure 3: The encoding of the dataset given a cover and CT. See Example 2 for more details.

Using C_p and C_g we can compute the actual codes to construct the code table CT corresponding to PS used to cover the data. We will encode the data using optimal prefix codes [7], the length of which we can compute by Shannon entropy. In our case this means that the length of an optimal pattern code for pattern X is the negative log-likelihood of a pattern in the cover [8], that is

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

where $usage(X)$ is the number of times a pattern X is used to

describe the data, i.e. $usage(X) = |\{Y \in C_p | Y = code_p(X)\}|$.

Gap and fill code lengths are computed similarly, corresponding to the negative log-likelihood of these codes within the usages of the corresponding pattern. That is, we have

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

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

where $gaps(X)$ and $fills(X)$ are the number of times a gap, resp. fill-code of pattern X is used in the cover of the data, i.e. $gaps(X) = |\{Y \in C_g \mid Y = code_g(X)\}|$ and analogue for $fills(X)$.

3.2.3 Encoded Length of Data Given Code Table

Now that we have determined the cover and encoding scheme, we can formalise the calculation of the encoded length of the data given the CT. This encoded length is the sum of the encoded length of following terms: the pattern stream, the gap stream, the number of attributes, the number of sequences and the length of each sequence. Formally, we have

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

where $L_{\mathbb{N}}$ is the MDL optimal Universal code for integers [8] and the encoded length of the pattern and gap stream are simply the code lengths of all codes in the streams,

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

$$L(C_g \mid CT) = \sum_{X \in CT}^{t(X)>1} \big[gaps(X) L(code_g(X)) + fills(X) L(code_f(X))\big] \quad .$$

3.2.4 Encoded Length of Code Table

The encoded length of the code table consists of the following parts. For each attribute A_j we encode the number of singletons and their support in D^j. Then we encode the number of non-singleton patterns in the code table, the sum of their usages, and then using a strong number composition their individual usages. Last, we encode the patterns themselves using $L(X \in CT)$. We hence have

$$L(CT \mid \mathcal{C}) = \sum_{j \in |A|} \left(L_{\mathbb{N}}(|\Omega_j|) + \log\binom{|D^j|}{|\Omega_j|}\right) + L_{\mathbb{N}}(|\mathcal{P}| + 1) + L_{\mathbb{N}}(usage(\mathcal{P}) + 1) + \log\binom{usage(\mathcal{P})}{|\mathcal{P}|} + \sum_{X \in \mathcal{P}} L(X \in CT) \quad .$$

For the encoded length of a non-singleton pattern $X \in CT$ we have

$$L(X \in CT) = L_{\mathbb{N}}(t(X)) + L_{\mathbb{N}}(gaps(X) + 1) + \sum_{t(X)} \log(|A|) + \sum_{x \in X} L(code_p(x \mid ST)) \quad ,$$

where we first encode its length, and its total number of gaps – note that we can derive the total number of fills for this pattern from these two values. As $L_{\mathbb{N}}$ is defined for integers $z \geq 1$, we apply a $+1$ shift wherever z can be zero [8]. Then, per time step, we encode for how many attributes the pattern defines a value, and what these values are using the singleton-only, or Standard Code Table (ST). For the encoded length of an event given the ST we have

$$L(code_p(x \mid ST)) = -\log\left(\frac{support(x \mid D)}{||D||}\right) \quad ,$$

which is simply the negative log-likelihood of the event under an independence assumption.

3.3 Formal Problem Definition

Loosely speaking, our goal is to find the most succinct description of the data. By MDL, we define the optimal set of patterns for the given data as the set for which the optimal cover and associated optimal code table minimise the total encoded length. As such we have the following problem definition.

Minimal Pattern Set Problem *Let Ω be a set of events and let D be a multivariate sequential dataset over Ω, find the minimal set of multivariate sequential patterns $\mathcal{P}$ and cover $\mathcal{C}$ of D using $\mathcal{P}$ and Ω, such that the encoded length $L(CT, D) = L(CT \mid \mathcal{C}) + L(D \mid CT)$ is minimal, where CT is the code-optimal code table for $\mathcal{C}$.*

Let us consider how complex this problem is. Firstly, the number of possible pattern sets (with a maximum pattern length of n) is

$$\sum_{k=1}^{2^{|\Omega|^n} - |\Omega| - 1} \binom{2^{|\Omega|^n} - |\Omega| - 1}{k} \quad .$$

Secondly, to use a pattern set to cover the data we also need to specify the order of the patterns in the set. That is, we need to find the optimal order for the elements in the pattern set to find the one that minimises the total encoded length. The total number of ways to cover the dataset using one of the possible ordered pattern sets is

$$\sum_{k=1}^{2^{|\Omega|^n} - |\Omega| - 1} \binom{2^{|\Omega|^n} - |\Omega| - 1}{k} \times (k + |\Omega|)! \quad .$$

Moreover, unfortunately, it does not show submodular structure nor (weak) (anti-)monotonicity properties by which we would be able to prune large parts of it. Hence, we resort to heuristics.

4. THE DITTO ALGORITHM

In this section we present DITTO, an efficient algorithm to heuristically approximate the MDL optimal summary of the data. In particular, it avoids enumerating all multivariate patterns, let alone all possible subsets of those. Instead, it considers a small but highly promising part of this search space by iterative bottom-up search for those patterns that maximally improve the description. More specifically, we build on the idea of SLIM [24] and SQS [28]. That is, as candidates to add to our model, we only consider the most promising combinations of already chosen patterns – as identified by their estimated gain in compression.

We give the pseudo code of DITTO as Algorithm 2. We start with singleton code table ST (line 1) and a set of candidate patterns of all pairwise combinations of singletons (line 2). We then iteratively add patterns from the candidate set to code table CT in **Candidate Order**: $\downarrow$ $estimated\ gain(X)$, $\downarrow$ $support(X \mid D)$, $\downarrow$ $||X||$, $\downarrow$ $L(X \mid ST)$ and $\uparrow$ lexicographically (line 3). This order prefers the most promising candidates in terms of compression gain. When a new pattern improves the total encoded length $L(D, CT)$ we keep it, otherwise we discard it (line 4). After acceptance of a new pattern we prune (line 5) CT and recursively test whether to add variations (line 6) of the accepted pattern in combination with

Algorithm 2 The DITTO Algorithm

Input: The dataset D and singleton code table ST
Output: An approximation to the **Minimal Pattern Set Problem**

1: $CT \leftarrow ST$
2: $Cand \leftarrow CT \times CT$
3: **for** $X \in Cand$ in **Candidate Order do**
4: **if** $L(D, CT \oplus X) < L(D, CT)$ **then**
5: $CT \leftarrow \text{PRUNE}(D, CT \oplus X)$
6: $CT \leftarrow \text{VARIATIONS}(D, X, CT)$
7: $Cand \leftarrow CT \times CT$
8: **return** CT

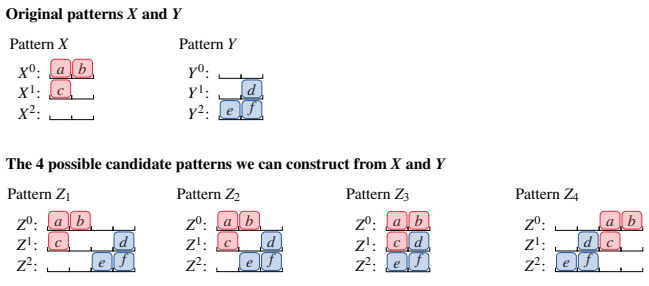

Figure 4: The 4 possible candidate patterns constructed from different alignments of X and Y.

its gap events. When all variations are tested recursively, we update the candidate set by combining $CT \times CT$ (line 7).

We give the details of each of these steps below, as well as explain how to gain efficiency through smart caching, and discuss the computational complexity of DITTO.

4.1 Covering

As detailed in the previous sections, to compute $L(D, CT)$, we first need to cover the data with CT. The COVER algorithm covers the data using the patterns as they are ordered in CT. To find the optimal cover we need to identify the cover order that leads to the smallest encoded length of the data. We do so greedily, by considering the pattern set in a fixed order. As our goal is to compress the data, we prefer patterns that cover many events with just a short code length. We hence define the **Cover Order** as follows: $\downarrow ||X||$, $\downarrow support(X \mid D)$, $\downarrow L(X \mid ST)$ and $\uparrow$ lexicographically. It follows the intuition that we give preference to larger and more frequent patterns, for which we expect a higher compression gain, to cover the data first, as these maximise the likelihood of the cover.

4.2 Candidates and Estimation

Conceptually, at every iteration the set of candidates from which we can choose, consists of the Cartesian product of the code table with itself. Two patterns X and Y can be combined to form new candidate patterns. Each alignment of X and Y without gaps in either X, Y and the resulting pattern, forms a new candidate, with the exception of alignments in which X and Y overlap. See Figure 4 for an example.

Selecting the candidate with the highest gain is very expensive – we would need to cover the whole data for every candidate. Instead, we select the candidate with the highest *estimated* gain – which can be done much more efficiently. Intuitively, based on the estimated gain ($\Delta L'$) we only consider candidate patterns in a lazy fashion based on their usage and do not consider patterns with a lower usage than the current best candidate. For notational brevity, for a pattern $X \in CT$ we use $x = usage(X)$. Further, let s be the total usage count of all patterns in CT, i.e. $s = \sum_{X \in CT} x$. For $CT' = CT \oplus Z$, we use x' and s' similarly. We estimate z, the usage of Z, optimistically as the minimum of the usages of X and Y – and as $x/2$ when $X = Y$ (because the usage of XX can not be higher). Formally, our gain estimate $\Delta L'(CT \oplus Z, D)$ of the true gain $\Delta L(CT \oplus Z, D)$ for adding pattern $Z = X \cup Y$ to CT is as follows,

$$\begin{aligned}
\Delta L'(CT',\ D) =\ & \Delta L'(CT' \mid D) + \Delta L'(D \mid CT') \quad , \\
\Delta L'(CT' \mid D) =\ & -L_{\mathbb{N}}(|Z|) - \sum_{l(Z)} \log(|A|) \\
& - \sum_{Z[i] \in Z} \sum_{z \in Z[i]} L(code_p(z \mid ST)) \quad , \\
\Delta L'(D \mid CT') =\ & s \log s - s' \log s' + z \log z - x \log x \\
& + x' \log x' - y \log y + y' \log y' \quad .
\end{aligned}$$

That is, the estimated gain of adding pattern Z to CT thus consists of the estimated gain in the size of the data, minus the increase in the size of CT. Note that, $\Delta L'$ is an estimate and for simplicity we ignore the effects of adding pattern Z to code table CT on the pattern and (no-)gap usages of patterns other than X and Y.

4.3 Pruning

After the acceptance of a new pattern in our code table other patterns may have become redundant as their role may have been overtaken by the newer pattern. Therefore, each time a pattern X is successfully added to the code table, we consider removing those $Y \in CT$ for which the usage decreased and hence the pattern code length increased. Algorithm 3 describes how a code table is pruned.

Algorithm 3 The PRUNE Algorithm

Input: The dataset D and a code table CT
Output: A pruned code table

1: $Cand \leftarrow X \in CT$ with decreased usage
2: **for** $X \in Cand$ in **Prune Order do**
3: **if** $L(D, CT \setminus X) < L(D, CT)$ **then**
4: $CT \leftarrow CT \setminus X$
5: $Cand \leftarrow Cand \cup \{Y \in CT \mid \text{usage decreased}\}$
6: $Cand \leftarrow Cand \setminus X$
7: **return** CT

4.4 Generating Pattern Variations

To efficiently discover a large and diverse set of promising patterns – without breadth-first-search, which takes long to find large patterns, and without depth-first-search, which would be prohibitively costly – we consider *variations* of each accepted pattern to the code table. That is, when a pattern leads to a gain in compression, we consider all ways by which we can extend it using events that occur in the gaps of its usages. This way we consider a rich set of candidates, plus speed up the search as we are automatically directed to patterns that actually exist in the data. Algorithm 4 outputs a code table possibly containing variations of the lastly added pattern Y.

For example, consider the dataset $\{a, b, a, b, c, a, c, a\}$ where pattern $\{a, a\}$ occurs twice with a gap of length one. After adding pattern $\{a, a\}$ to CT we consider the patterns $\{a, b, a\}$ and $\{a, c, a\}$ for addition to CT.

Algorithm 4 The VARIATIONS Algorithm

Input: The dataset D, a pattern Y and a code table CT
Output: A code table possibly containing variations of Y

```
Cand ← Y × gap events(Y)
for X ∈ Cand do
    if L(D, CT ⊕ X) < L(D, CT) then
        CT ← PRUNE(D, CT ⊕ X)
        CT ← VARIATIONS(D, X, CT)
    Cand ← Cand \ X
return CT
```

4.5 Faster Search through Caching

The space of all possible multivariate patterns is extremely rich. Moreover, in practice many candidate patterns will not lead to any gain in compression. In particular, those that occur only very infrequently in the data are unlikely to provide us any (or much) gain in compression. We thus can increase the efficiency of DITTO by allowing the user to impose a minimum support threshold for candidates. That is, only patterns X will be evaluated if they occur at least σ times in the data. To avoid time and again re-evaluating candidates of which we already know that they are infrequent, we cache these in a tree-based data structure. Only *materialised* infrequent pattern are added to the tree. Future candidates are only materialised when none of its subsets are present in the tree, as by the a priori principle we know it can not be frequent [15].

Even though this tree can theoretically grow very large, in practise it stays relatively small because we only consider a small part of the candidate space. That is, we only combine patterns we know to be frequent to form new candidate patterns. In practice, we found that DITTO only has to cache up to a few thousand candidates. Using this tree we see speed ups in computation of 2 to 4 times, while also memory consumption is strongly reduced. For some datasets the difference is even bigger, up to an order of magnitude.

In this work we only consider keeping track of infrequent candidates. Note, however, that at the expense of some optimality in the search additional efficiency can be gained by also storing *rejected* candidates in the tree. In both theory and practice, however, candidates rejected in one iteration may lead to compression gain later in the process [24].

4.6 Complexity

The time complexity of DITTO has a simple upper bound as follows. In the worst-case we cover the data for each frequent pattern from the set of all frequent patterns $\mathcal{F}$ in each iteration. Covering takes $O(|CT| \times ||D||)$ and the number of iterations is worst-case $O(|\mathcal{F}|)$. Together, the worst-case time complexity is

$$O(|\mathcal{F}|^2 \times |CT| \times ||D||) \quad .$$

From the experiments in Section 6, however, we will learn that this estimate is rather pessimistic. In practice the code table stays small ($|CT| \ll |\mathcal{F}|$), we only consider a subset of all frequent patterns and we do this greedily. In practice the runtime of DITTO therefore stays in the order of seconds to minutes.

5. RELATED WORK

The first to use MDL to summarise transaction data were Siebes et al. [22], resulting in KRIMP [32]. It shifted focus from the long-standing goal of mining collections of patterns that describe the set of all frequent patterns, e.g. closed [20] frequent patterns, to a *pattern set* that describes the data.

Similar to the transaction data domain, summarisation of sequential data also developed from frequent pattern mining. For sequential data, the key types of patterns studied are frequent subsequences [21], and frequent episodes [1, 16]. As with all traditional pattern mining approaches, redundancy is also a key problem when mining sequential patterns [33, 9]. To this end, Tatti and Vreeken [28] proposed to instead approximate the MDL-optimal summarisation of event sequence data using serial episodes. Their method SQS deals with many challenges inherent to this type of data, such as the importance of the order of events and the possibility for patterns to allow gaps in their occurrences – aspects we build upon and extend. Other methods exist, but either do not consider [2, 14] or do not punish gaps [10, 11] with optimal codes. None of these methods consider, or are easily extended to multivariate sequential data.

One of the first to consider multivariate sequential patterns, by searching for multi-stream dependencies, were Oates et al. [19]. Tatti and Cule [27] formalised how to mine the set of closed frequent patterns from multivariate sequential data, where patterns allow simultaneous events. In [35] the mining of high utility sequential patterns is studied, where they allow simultaneous events. Chen et al. [5] and Moerchen et al. [18] study mining interval-based patterns, where frequency is determined by how often univariate patterns co-occur within a given interval. All these methods are traditional pattern mining techniques in the sense that they return all patterns that pass an interestingness threshold.

Whereas traditional pattern mining techniques often only consider discrete data, there does exist extensive literature on mining patterns in continuous valued time series. These patterns are often called 'motifs' [12, 6]. For computational reasons, many of these methods first discretise the data [13]. Most motif discovery algorithms consider only univariate data. Example proposals for motif discovery in a multivariate setting include that by Tanaka et al. [25], who first transform the data into one dimension before the pattern discovery process and do not consider gaps, and by [17], who do not allow patterns to temporally overlap even if they span different dimensions and do not consider variable-length motifs. More recently Vespier et al. [30], mine characteristic multi-scale motifs in sensor-based time series but aim at mining all motifs, not a succinct summary.

To the best of our knowledge there are no methods yet to summarise *multivariate* sequential data, other than regarding each attribute separately or with restrictions on the pattern language [3]. In this work we introduce DITTO to discover important sequential associations between attributes by mining succinct summaries using rich multivariate patterns.

6. EXPERIMENTS

We implemented DITTO in C++ and generated our synthetic data and patterns using Python. We make our code available for research purposes.[2] All experiments were conducted on a 2.6 GHz system with 64 GB of memory. For our experiments on real data we always set the minimum support threshold as low as feasible, unless domain knowledge suggests otherwise.

We evaluate DITTO on a wide range of synthetic and real world data. As discussed in Sec. 5, there exist no direct competitors to DITTO. Traditional pattern mining and motif discovery methods 'simply' mine all patterns satisfying some constraints. For summarising sequential data, most existing methods consider univariate data [28, 10]. The only summarisation approach for multivariate sequential data considers the special case where attributes are ordered (e.g. frequency bands) [3], whereas we consider multivariate sequential data in general. We empirically compare DITTO to SQS [28].

[2] http://eda.mmci.uni-saarland.de/ditto/

We do so by applying SQS to each univariate sequence $S^j \in D$, combining these results into one pattern set.

6.1 Synthetic Data

To validate whether DITTO correctly identifies true multivariate sequential patterns from a dataset, we first consider synthetic data. In particular, we generate random data in which we plant a number of randomly generated patterns of different characteristics. Clearly, ideally the true patterns are recovered. Moreover, ideally no other, spurious patterns that are only due to noise are returned. To this end we perform an extensive set of experiments varying the number of events, the number of attributes and the alphabet size of the dataset, and the number, frequency and size of the planted patterns.

Data Generation

As noted in Table 1, for each experiment we generated $t(D)$ random multi-events on $|A|$ attributes (i.e. a total of $||D||$ events) with an alphabet size per attribute of $|\Omega_i|$. Further, after the data generation $|\mathcal{P}|$ patterns are planted, where each pattern X has a size $||X||$, a 5% chance on a gap between subsequent multi-events, and a support such that each pattern spans $support\%$ of all events in the dataset. An example of an insertion of a pattern in a random dataset that does not lead to an actual occurrence of that pattern is when due to the gap chance the minimal window of the pattern contains too many gaps. We do not allow patterns to overwrite each other during generation, as this makes evaluation much more complicated – i.e. it becomes unclear whether not recovering a pattern is an artefact of the search or of the data generation process. Further, only for experiments with 50 attributes, we prevented that pattern occurrences interleave and did not allow an event to be used in more than one pattern to assure that the planted patterns are actually present in the data. This restriction is justified because we are merely testing whether our algorithm is able to find multivariate patterns in multivariate data and without it patterns will easily crowd each other because of the high number of attributes.

Evaluation

We evaluate the quality of the pattern set discovered by considering how close they represent the planted patterns. In particular, following [34] we consider both exact ($=$) and subset ($\subset$) matches. Exact indicates that the pattern exactly corresponds with a planted pattern, whereas subset implies that it is only part of a planted pattern. In addition, we consider how well the planted patterns are recovered; we report how many of the events of the ground truth pattern set $\mathcal{P}$ we can cover with the discovered patterns. The higher this ratio, the better this result. Last, we consider the gain in compression of the discovered model over the initial, Standard Code Table. The higher this number, the better – the best score is attained when we recover all patterns exactly, and no further noise.

Results

We first consider the traditional approach of mining all (closed) frequent multivariate patterns. We do so using the implementation of Tatti and Cule [27]. We use a minimal support of 90% of the lower-support planted pattern. This choice is made to ensure that even when not all insertions of a pattern result in an actual occurrences, it can still be discovered. For the most simple synthetic dataset we consider, corresponding to the first row of Table 1, this takes a few days, finally reporting a *lower bound* of 14 092 944 394 frequent patterns, and returning 6 865 closed frequent patterns – hardly a summary, knowing there are only 5 true patterns. In the remainder we therefore do not consider traditional pattern mining.

Next, we consider DITTO and SQS. We report the results in Table 1. On the right hand side of the table we see that DITTO recovers all planted patterns, and does not report a single spurious pattern (!). In all cases it recovers the ground truth model, and obtains the best possible gain in compression. Next to the exactly identified planted pattern sometimes it also identifies some subsets of the planted patterns. This is a result of the data generation, i.e. subsets are sometimes included in the code table when planted occurrences contain too many gaps to be covered with the exact pattern. The patterns SQS discovers, on the other hand, are only small univariate fragments of the ground truth, recovering roughly only 10% to 30% of the ground truth. The near-zero gains in compression corroborate it is not able to detect much structure.

Regarding runtime and scalability, DITTO scales very favourably. Although SQS is faster it considers only the much simpler case of univariate data and patterns. DITTO requires seconds, up to a few minutes for the largest data, even for very low minimal support thresholds. Analysing the runtime of DITTO in more detail show how well its heuristics work; most time is spent on computing the minimal windows for candidates, of which up to ten thousand are materialised. Only for a few hundred of these a full cover of the data is required to evaluate their contribution to the model. Smart implementation for computing the minimal windows of all candidates in one pass will hence likely speed up DITTO tremendously.

6.2 Real World Data

As case studies we consider 4 datasets. An ECG sensor dataset, a structural integrity sensor dataset of a Dutch bridge, the text of the novel Moby Dick tagged with part-of-speech tags, and a multilingual text of an European Parliament resolution. See Table 2 for their characteristics.

ECG.

To investigate whether DITTO can find meaningful patterns on real-valued time series we consider a well-known ECG dataset.[3] For ease of presentation, and as our main goal is to show that DITTO can discover multivariate patterns, we consider only two sensors. As preprocessing steps we applied 3 transformations: we subsampled the data, we transformed it from absolute to relative, and we discretised it using SAX [13]. For the subsampling we replaced each 5 subsequent values with their average, thus creating a sequence 5 times shorter. Thereafter, we transformed the absolute data into relative data by replacing each value by the difference of its successor and its own value. Lastly, we discretised each attribute into 3 intervals using SAX. Using a minimum support of 10, within 360 seconds DITTO discovers a code table containing 11 non-singleton patterns that capture the main structure of the data. In Figure 5 we plotted 2 occurrences of the top ordered pattern of this code table. This is a multivariate pattern which shows a very characteristic repeating structure in the data comprising a longer flat line ending with a peak on both attributes simultaneously. Showing the power of our pattern language, note that the pattern does not define any values on the second attribute for the first and second-last time steps (indicated with arrows in Figure 5), while it does on the first attribute. This flexibility allows us to use this multivariate pattern to describe a larger part of the data.

Like for the synthetic data, we also ran SQS. Again we see that as it cannot reward multivariate structure it does not obtain competitive gains in compression. Close inspection shows it returns many small patterns, identifying consecutive values.

[3] ECG – `physionet.org/physiobank/database/stdb`

Table 1: DITTO discovers all planted patterns on all synthetic datasets, without picking up on noise. Given are the base statistics of the synthetic datasets, and the results of SQS and DITTO. For SQS and DITTO we give the number of exactly recovered patterns ($=$) and the number of discovered patterns that are subsets of planted patterns ($\subset$). Further, we report how much of the ground truth is recovered ($R\%$), as well as the gain in compression over the singleton-only model ($L\%$), for both higher is better. Last, we give the runtime in seconds.

Data				**Planted Patterns**			**SQS**					**DITTO**				
$\|\|D\|\|$	$t(D)$	$\|A\|$	$\|\Omega_i\|$	$\|\mathcal{P}\|$	$\|\|X\|\|$	*support*	$=$	$\subset$	$R\%$	$\Delta L\%$	time	$=$	$\subset$	$R\%$	$\Delta L\%$	time
100 000	10 000	10	100	5	3–7	1%	0	1	9.5	0.1	3	5	0	100.0	3.0	2
100 000	10 000	10	100	5	**5**	**10%**	0	0	0	0	3	5	5	100.0	31.6	31
100 000	10 000	10	**1 000**	5	3–7	1%	0	1	8.0	0	12	5	0	100.0	2.9	4
100 000	10 000	10	100	**20**	3–7	1%	0	9	19.5	0.9	4	20	0	100.0	12.9	15
500 000	10 000	**50**	100	5	3–7	1%	0	0	0	0	15	5	1	100.0	3.1	41
500 000	**50 000**	10	100	5	3–7	1%	0	1	10.0	0.2	4	5	0	100.0	3.1	28
1 000 000	**100 000**	10	100	5	3–7	1%	0	4	31.8	0.3	4	5	14	100.0	3.3	374

Table 2: Results on 4 real datasets. We give the basic statistics, as well as the number of patterns and relative gain in compression for both SQS and DITTO. The high compression gains show that DITTO discovers much more relevant structure than SQS.

Data						**SQS**			**DITTO**		
Dataset	$t(D)$	$\|D\|$	$\|A\|$	$\|\Omega\|$	*support*	$\|\mathcal{P}\|$	$\Delta L\%$	time (s)	$\|\mathcal{P}\|$	$\Delta L\%$	time (s)
ECG	2 999	1	2	6	10	57	38.8	1	11	**75.3**	360
Bridge	5 000	1	2	10	100	21	58.8	1	22	**76.3**	325
Moby Dick	2 248	103	2	887	5	20	1.7	3	79	**14.3**	102
Text	5 960	115	3	4 250	10	35	1.6	12	51	**2.2**	136

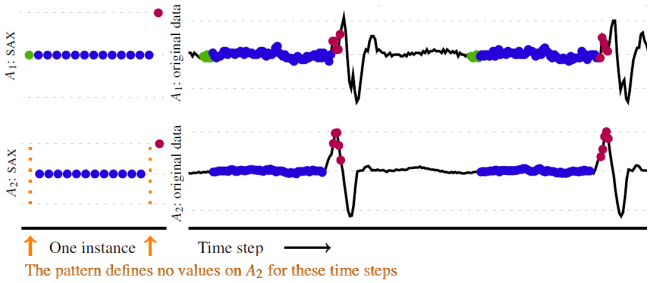

Figure 5: The top ordered pattern for the ECG data in the code table (left) and its first 2 occurrences in the data (right). The first time step of an occurrence is marked in green, subsequent ones in blue, and the last in red.

Bridge.

Next we consider the setting of monitoring the structural integrity of a bridge in the Netherlands.[4] Amongst the collected sensor data are the strain, vibration and temperature. We selected 2 strain sensors (1 Hz) on consecutive pillars of the bridge and decomposed the signals into low, medium and high frequency components using techniques from [29]. We used the medium frequency components, after preprocessing, as our dataset. As preprocessing, we transformed the absolute values into relative values by replacing each value by the difference of its successor and its own value. We then discretised each z-normalised attribute into 5 intervals using SAX [13].

For a support threshold of 100, it takes DITTO 325 seconds to discover a code table with 22 non-singleton patterns. Although only one more than SQS at the same threshold, the patterns DITTO discovers are more descriptive. That is, they are multivariate and larger, leading to a much higher gain in compression. Moreover, the patterns it discovers correctly show the correlation between the attributes, whereas the patterns SQS discovers only identify univariate patterns.

Moby Dick.

For more interpretable results, we next evaluated on text data. In particular we considered the first chapter of the book Moby Dick, written by Herman Melville,[5] aligning the text with part-of-speech tags.[6][7] That is, one attribute comprises a stream of the words used in the book; each sentence is regarded as a sequence. The other attribute contains the tags that identify the type and function of each of these words. For example,

attribute 1:	VB	PRP	NNP
attribute 2:	Call	me	Ishmael

for which we will further use the following notation, where each time step is enclosed by curly brackets and the symbols for different attributes within a time step are divided by a comma: {VB, Call}{PRP, me}{NNP, Ishmael}. A short description for the part-of-speech tags in this example can be found in Table 4.

With a support threshold of 5, it takes DITTO 102 seconds to discover 79 non-singleton patterns. After studying the resulting pattern set we found that the identified patterns show highly intuitive multivariate structure. The highest ordered patterns together with examples of text fragments that match these patterns are shown in Table 3.

[4] Bridge – `infrawatch.liacs.nl`

[5] `www.gutenberg.org`

[6] http://nlp.stanford.edu/software/tagger.shtml

[7] https://gate.ac.uk/wiki/twitter-postagger.html

Table 3: The highest ordered patterns in the code table for the Moby Dick dataset, together with example fragments of from the text which correspond to the patterns.

Pattern	Example text fragments
{TO, to}{VB}{DT, a}{NN}	to get a broom, to buy (him) a coat
{DT, the}{JJ}{NN}	the green fields, the poor poet
{DT, a}{JJ}{NN}	a mazy way, a hollow trunk
{DT, the}{JJ}{NNS}	the wild conceits, the old (sea-)captains
{PRP, i}{RB}{VB}	I quietly take, I always go
{EX, there}{VBZ, is}	there is

Table 4: A short description of the part-of-speech tags used in the examples in this paper for the Moby Dick experiment.

Tag	Explanation
DT	Determiner
EX	Existential *there*
JJ	Adjective
NN	Noun, singular or mass
NNP	Proper noun, singular
NNS	Noun, plural
PRP	Personal pronoun
RB	Adverb
TO	*to*
VB	Verb, base form
VBZ	Verb, 3rd person singular present

Given the modest compression gain that SQS obtains, see Table 2, it is clear there is not much structure in each of the attributes separately; DITTO, however, is able to find a significant amount of multivariate structure.

Multilingual Text.

As a final experiment, to further corroborate whether DITTO discovers meaningful and interpretable multivariate patterns, we consider mining patterns between the same text in different languages. To this end we collected a text in English, French and German from the European Parliament register of documents.[8] In this data we expect frequent combinations of words within one (or more) language(s), as well as (and of more interest), multivariate patterns in the form of translations between the languages. As a preprocessing step all text data was stemmed and stop words were removed. To keep the different languages aligned we regarded every paragraph as a subsequence and padded shorter aligned subsequences with sentinel events which are ignored by DITTO. This ensures that the difference in length of the sentences in different languages will not lead to very big misalignments.

For a support threshold of 10, DITTO takes 136 seconds to discover 51 non-singleton patterns. The highest ordered pattern, i.e. the one that aids compression most, is a translation pattern; it identifies the correct relation between the French word *relève*, the German phrase *stellt fest dass* and the English word *note*. Other high ordered patterns are the English *EUR (x) million* and the German *(x) Millionen EUR*, and the words *parliament*, *Parlament* and *parlement* in English, German and French, respectively.

The modest compression gain of DITTO over SQS, see Table 2, indicates this data is not very structured, neither univariately, nor multivariately. One of the reasons being the different order of words between different languages which results in very large gaps between translation patterns.

[8]Text – `www.europarl.europa.eu/RegistreWeb`

7. DISCUSSION

Overall, the experiments show that DITTO works well in practice. In particular, the experiments on synthetic data show that DITTO accurately discovers planted patterns in random data for a wide variety of data and patterns dimensions. That is, DITTO discovers the planted patterns regardless of their support, their size, and the number of planted patterns – without discovering any spurious patterns. DITTO also performed well on real data – efficiently discovering characteristic multivariate patterns.

The results on the annotated text are particularly interesting; they give clear insight in non-trivial linguistic constructs, characterising the style of writing. Besides giving direct insight, these summaries have high downstream potential. One could, for example, use them for comparative analysis [4]. For example, for text identifying similarities and differences between authors, for sensor networks detecting and describing concept drift over time, and characterising differences between patients.

Although DITTO performs very well in practice, we see ways to improve over it. Firstly, MDL is not a magic wand. That is, while our score performs rather well in practice, we carefully constructed it to reward structure in the form of multivariate patterns. It will be interesting to see how our score and algorithms can be adapted to work directly on real-valued data. Secondly, it is worth investigating whether our current encoding can be refined, e.g. using prequential codes [4]. A strong point of our approach is that we allow for noise in the form of gaps in patterns. We postulate that we can further reduce the amount of redundancy in the discovered pattern sets by allowing noise *in the occurrences* of a pattern, as well as when we allow *overlap* between the occurrences of patterns in a cover. For both cases, however, it is not immediately clear how to adapt the score accordingly, and even more so, how to maintain the efficiency of the cover and search algorithms.

Last, but not least, we are interested in applying DITTO on vast time series. To accomodate, the first step would be to investigate parallelisation; the search algorithm is trivially parallelisable, as candidates can be generated and estimated in parallel, as is the covering of the data. More interesting is to investigate more efficient candidate generation schemes, in particular top-k mining, or lazy materialization of candidates.

Previous work has shown that MDL-based methods work particularly well for a wide range of data mining problems, including classification [32, 11] and outlier detection [23]. It will make for interesting future work to investigate how well DITTO solves such problems for multivariate event sequences. Perhaps the most promising direction of further study is that of causal inference [31].

8. CONCLUSION

We studied the problem of mining interesting patterns from multivariate sequential data. We approached the problem from a pattern set mining perspective, by MDL identifying the optimal set of patterns as those that together describe the data most succinctly. We proposed the DITTO algorithm for efficiently discovering high-quality patterns sets from data.

Experiments show that DITTO discovers patterns planted in synthetic data with high accuracy. Moreover, it scales favourably with the length of the data, the number of attributes, and alphabet sizes. For real data, it discovers easily interpretable summaries that provide clear insight in the associations of the data.

As future work, building upon our results on the part-of-speech tagged text data, we are collaborating with colleagues from the

linguistics department to apply DITTO for analysis of semantically annotated text and for inferring patterns in morphologically rich languages.

Acknowledgments

The authors wish to thank Kathrin Grosse for her help with and analysis of the part-of-speech tagged text experiments. Roel Bertens and Arno Siebes are supported by the Dutch national program COMMIT. Jilles Vreeken is supported by the Cluster of Excellence "Multimodal Computing and Interaction" within the Excellence Initiative of the German Federal Government.

9. REFERENCES

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, pages 3–14, Los Alamitos, CA, USA, 1995. IEEE Computer Society.

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

[3] R. Bertens and A. Siebes. Characterising seismic data. In *SDM*, pages 884–892. SIAM, 2014.

[4] K. Budhathoki and J. Vreeken. The difference and the norm – characterising similarities and differences between databases. In *ECML PKDD*. Springer, 2015.

[5] Y.-C. Chen, J.-C. Jiang, W.-C. Peng, and S.-Y. Lee. An efficient algorithm for mining time interval-based patterns in large database. In *CIKM*, pages 49–58. ACM, 2010.

[6] B. Chiu, E. Keogh, and S. Lonardi. Probabilistic discovery of time series motifs. In *KDD*, pages 493–498. ACM, 2003.

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

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

[9] Y. He, J. Wang, and L. Zhou. Efficient incremental mining of frequent sequence generators. In *DASFAA*, pages 168–182, 2011.

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

[11] H. T. Lam, F. Mörchen, D. Fradkin, and T. Calders. Mining compressing sequential patterns. *Statistical Analysis and Data Mining*, 7(1):34–52, 2014.

[12] J. Lin, E. Keogh, P. Patel, and S. Lonardi. Finding motifs in time series. In *KDD-TDM*, pages 53–68, 2002.

[13] J. Lin, E. Keogh, L. Wei, and S. Lonardi. Experiencing sax: A novel symbolic representation of time series. *Data Min. Knowl. Disc.*, 15(2):107–144, Oct. 2007.

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

[15] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *KDD*, pages 181–192, 1994.

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

[17] D. Minnen, C. Isbell, I. Essa, and T. Starner. Detecting subdimensional motifs: An efficient algorithm for generalized multivariate pattern discovery. In *ICDM*, pages 601–606. IEEE, 2007.

[18] F. Mörchen and A. Ultsch. Efficient mining of understandable patterns from multivariate interval time series. *Data Min. Knowl. Disc.*, 15(2):181–215, 2007.

[19] T. Oates and P. R. Cohen. Searching for structure in multiple streams of data. In *ICML*, pages 346–354, 1996.

[20] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT*, pages 398–416. ACM, 1999.

[21] J. Pei, J. Han, B. Mortazaviasl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE TKDE*, 16(11):1424–1440, 2004.

[22] A. Siebes, J. Vreeken, and M. van Leeuwen. Item sets that compress. In *SDM*, pages 393–404. SIAM, 2006.

[23] K. Smets and J. Vreeken. The odd one out: Identifying and characterising anomalies. In *SDM*, pages 804–815. SIAM, 2011.

[24] K. Smets and J. Vreeken. SLIM: Directly mining descriptive patterns. In *SDM*, pages 236–247. SIAM, 2012.

[25] Y. Tanaka, K. Iwamoto, and K. Uehara. Discovery of time-series motif from multi-dimensional data based on mdl principle. *Mach. Learn.*, 58(2–3):269–300, 2005.

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

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

[28] N. Tatti and J. Vreeken. The long and the short of it: Summarizing event sequences with serial episodes. In *KDD*, pages 462–470. ACM, 2012.

[29] U. Vespier, A. J. Knobbe, S. Nijssen, and J. Vanschoren. MDL-based analysis of time series at multiple time-scales. In *ECML PKDD*, pages 371–386. Springer, 2012.

[30] U. Vespier, S. Nijssen, and A. Knobbe. Mining characteristic multi-scale motifs in sensor-based time series. In *CIKM*, pages 2393–2398. ACM, 2013.

[31] J. Vreeken. Causal inference by direction of information. In *SDM*. SIAM, 2015.

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

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

[34] G. Webb and J. Vreeken. Efficient discovery of the most interesting associations. *ACM TKDD*, 8(3):1–31, 2014.

[35] C.-W. Wu, Y.-F. Lin, P. S. Yu, and V. S. Tseng. Mining high utility episodes in complex event sequences. In *KDD*, pages 536–544. ACM, 2013.