Summarising Event Sequences using Serial Episodes and an Ontology

Kathrin Grosse[1] and Jilles Vreeken[2]

[1] CISPA, Saarland University, Germany
[2] Max-Planck Institute for Informatics and Saarland University, Germany

Abstract. The ideal result in pattern mining is a small set of patterns that identify the key structure of the data. In this paper we propose NemO, an efficient method for summarising discrete sequences in terms sequential patterns over an ontology. That is, NemO can detect patterns between not only the surface level, but also between the categories the events in the data are part of, and returns a small set of non-redundant patterns that together describe the data well. We formalize the problem in terms of the Minimum Description Length (MDL) principle, and propose an efficient and effective heuristic to mine good models directly from data. Extensive empirical evaluation shows that NemO performs very well in practice, recovering the ground truth in synthetic data, and finds meaningful conceptual and grammatical patterns in text data.

1 Introduction

Data often exhibits structure beyond what is apparent on the surface level, that is, beyond the actual observed values. As a toy example, consider both '*cat meows*' and '*dog barks*'. Whereas these sentences show no syntactic pattern, if we know that '*cat*' and '*dog*' are both nouns, and '*meows*' and '*barks*' are verbs, they are clear examples of a pattern *noun–verb*. If we know that the former two are '*pets*', and the latter two are cases of '*make noise*', a more specific pattern would be *pet–make noise*. In this paper we consider the problem of summarizing event sequences in terms of sequential patterns over an ontology, that is, to discover a small set of non-redundant patterns that describe the data well.

Our empirical evaluation shows that our method indeed discovers relevant patterns. Many of them are intuitive, such as *he–Verb* or *the–Noun*. More complex patterns from the Lord of the Rings novels, for example, was *he–Verb-Conjunction–he*, and matches indirect speech. Other patterns such as *the–Adjective–Noun–and*, from the same novel, match enumerations.

In particular, we formalize the problem in terms of the Minimum Description Length (MDL) principle [13,7]. While pattern based summarisation is already a hard problem [16], we show that using an ontology only increases the complexity: patterns are not only combinations of different entities, but also between different possible generalizations. To approximate the ideal result, we propose the NemO algorithm, an efficient heuristic that can mine good models directly from data. Through extensive empirical evaluation we show that NemO can recover the

In: P. Cellier, T. Charnois, A. Hotho, S. Matwin, M.-F. Moens, Y. Toussaint (Eds.): Proceedings of DMNLP, Workshop at ECML/PKDD, Skopje, Macedonia, 2017.

ground truth from synthetic data, as well as discover meaningful patterns from text data including speeches and novels.

2 Preliminaries

In this section, we introduce the notation used throughout the paper and further give a brief primer on MDL.

2.1 Notation

Our approach works on **discrete sequential data** $D \in \Sigma^l$ of length l over an alphabet Σ. To address an item at position i, we will write D_i. These items are part of an **ontology**, a directed graph $O = G(V, E)$ where V is a set of vertices and E are directed edges between those vertices. The root node v_Ω is the most general node. The most specific nodes without outgoing edges are called leafs.

To indicate a path or trail from node a to b we write T_{ab}. There are thus edges $e_1, .., e_n$ such that e_1 starts at a and e_n ends at b. There is always a path from a node to itself, T_{aa}, and the number of edges on this trace is specified as $|T_{nm}|$. Whenever several paths are possible, we will use the cheapest. The cost of the edges will be explained below. Coming back to v_Ω, we can say that $\forall v \in V$ $T_{(v_\Omega, v)}$, e.g. we can reach each node starting from the root.

Whenever we want to refer to an arbitrary item from the data D or a node from the ontology O, we will refer to it as entity or word $w \in V$. Consider that an entity w that describes some part of the data D need not be part of the data itself, since it can be the ancestor of several items in the data. Consider that a_1 and a_2 could both be descendants from A, e.g. T_{Aa_1} and T_{Aa_2}. Then it could be the case that $A \in O$, but $A \notin D$. By combining several entities, we can form a sequential **pattern** P of l events, i.e. $P \in V^l$. To refer to the i^{th} entity in P, we write $w_i \in P$. If we want to cover the data with a pattern, we need to know where it matches the data. Whenever there is a path from $w_1 \in P$ to D_i, a part of the pattern *matches* an entity in the data. If we can match all parts of the pattern in the right order, we have found a **window** $D[i, j]$. There are thus l data entities $D_1 \dots D_l \in D[i, j]$ such that $1 < 2 \dots < l$, $l = |P|$ and $\forall x \in 1 \dots l$ $T_{D_x w_x}$ where $w_x \in P$.

In particular, we consider **serial episodes** as patterns. That is, the occurrences of a pattern P are allowed to contain one or multiple **gaps**. As a consequence, it can be the case that $|P| \leq j - i$. Further the definition from the window is actually independent from P, so a window might contain none or up to several patterns.

Coming back to our example where A is an ancestor of a and given pattern $P = AB$ and data $D = abcab$, there are several possibilities to match the pattern. To encounter this issue, we introduce the notion of a **minimal window**, which does not contain any other window. A window $D[i, j]$ is minimal if $\neg\exists\ D[k, m]$ where $k \geq i$ and $m < j$ or $k < i$ and $m \leq j$ and both match P.

We can now define the **support** for a pattern by simply counting the number of all minimal windows in the data. The Support $S(D, P)$ of a pattern P is $|\{D[i, j])|$ window is minimal and matches $P\}|$, i.e. cardinality of the set of all minimal windows of P in the data D.

2.2 MDL

The Minimum Description Length (MDL) principle [13,7] is a practical version of Kolmogorov complexity. Both can be summarised by the slogan 'Induction by Compression'. The MDL principle states that the best model $M \in \mathcal{M}$ for a dataset D is the model that provides the best lossless compression. Formally, we optimize $L(D, M) = L(M) + L(D \mid M)$, where $L(M)$ is the complexity of the model and $L(D|M)$ is the complexity of the data given the model. To use MDL in practice, we need to define our class of models $\mathcal{M}$, how to describe a model M in bits, and how to describe the data D in bits given a model.

3 Theory

Before we can formally define the problem we consider we have to specify both $L(M)$, the encoded cost of a model, and $L(D \mid M)$, the encoded cost of the data given a model. We do this in turn.

3.1 Encoding a Model

Before we can define how to encode a pattern, and therewith a model consisting of patterns, we have to define how to reach an entity w in the ontology from a starting node v_s in O. From there onward, we have to identify which edges to follow. As each edge has a normalized frequency assigned, we have a probability distribution over the nodes we can reach from a given node. Shannon entropy [7] states that the length of the optimal prefix-free code for a probability p is simply $-\log(p)$, hence we can compute it easily. This means we formally have

$$L(w \mid O, v_s) = |T_{vt}| + 1 - \sum_{e_{nm} \in T_{vt}} \log \frac{fr_n(e_{nm})}{\sum\limits_{e_{no} \in E} fr_n(e_{no})} \quad ,$$

where the first part is the number of bits required to indicate whether we have already arrived, or not, at our destination node. In total, this sums up to the number of edges plus one. We then compute the logarithm of the normalized frequency of the individual given frequencies ($fr(e)$). Knowing how to encode a path in the ontology, we can define how to encode a pattern.

Encoding a Pattern. To encode a pattern, we first encode its length, and then one by one the paths to the entities $w \in X$ starting from the root node

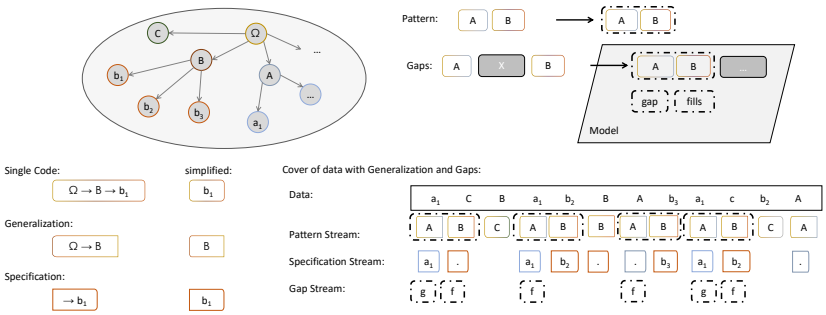

Fig. 1. An Ontology, generalized and specified singletons on the left. On the right hand side, there is a model and the representation of the data using the model.

of the ontology. As depicted in Fig. 1, the word need not be leaf and can be generalized. The encoded length of a pattern X hence is

$$L(X \mid O) = L_{\mathbb{N}}(|X|) + \sum_{w \in X} L(w \mid O, v_\Omega) \quad ,$$

where we first encode the length of the pattern using the MDL optimal Universal code for integers [14]. This is defined as $L_{\mathbb{N}}(n \geq 1) = \log^*(n) + \log(c_0)$, where $\log^*$ includes only positive terms and is defined as $\log^*(n) = \log(n) + \log(\log(n)) + \ldots$. To obtain a valid encoding, i.e. one that fulfills the Kraft inequality, c_0 is set to 2.865064.

Since we allow pattern occurrences to have gaps, to ensure lossless decoding we have to encode when these happen. We do this with *fill* and *gap* codes per pattern. A fill code means that we can simply decode the current entity of the pattern, and advance to the next. A gap code means that we may not yet decode the current entity of the pattern, and have to instead read which entity comes next from the pattern stream (details below, an example is given in Fig. 1). To obtain optimal codes, we again consider the probabilities that gaps and fills are used. Hence, we define $usage(\mathrm{X})$ as the times the pattern is actually used in the data.

To obtain the fill code, consider that we can only use a fills code for all but the first word in the pattern, yielding $fills(X) = usage(X) \times (|X| - 1)$. With $gaps(X)$, we denote the total number of gaps in how we use P to describe the data. We thus define the codes for gaps and fills based on the particular probabilities,

$$c_f(X) = -\log \frac{fills(X)}{gap(X) + fills(X)} \quad \text{and} \quad c_g(X) = -\log \frac{gap(X)}{gap(X) + fills(X)} \quad .$$

Encoding a Model. A model, $M = S \cup P$, which is a set of patterns P and singletons S. Due to their different nature, we will treat singletons and patterns in the model differently.

We will start to encode which words from our ontology are actually used as singletons in the data, and refer to this number by $|O_D|$. We encode this number again using the MDL optimal universal integer code, adding one in case the data is empty. To further know which words we actually use in the data, we will use a so-called data-to-model code [16]. We represent using this code all possibilities to distribute $|O_D|$ words over $|O|$ buckets, and then refer to the one representing the actual word set, thus $L_U(|O|, |O_D|) = \log \binom{|O|}{|O_D|}$. Additionally, we need to determine the support for each of the singletons to be able to use their codes. At this point, we will also use the data-to-model code. However, what we are now encoding are the possibilities to distribute the patterns over the data, and refer to the one actually representing the distribution. $L_U(|D|-1, |W|-1) = \log \binom{|D|-1}{|W|-1}$ where we treat the words as distinguishable and we define each of the $|D|$ bins to be non-empty. Finally, we also define $L(0, 0) = 0$.

We thus obtain as constant length for the singleton model $L(M_S \mid O) = L_{\mathbb{N}}(|O_D| + 1) + L_U(|O|, |O_D|) + L_U(|D| - 1, |O_D| - 1)$. Given some data and an ontology constructed from this data, however, this ontology will not be changed during the mining process. Realizing that this encoding factor is instead constant, we will omit the details here.

Next, we encode the patterns in the model. We first make use of the previously introduced encoding of a pattern. We then add the number of gaps and obtain

$$L(P \mid O) = \sum_{X \in P} L(X \mid O) + L_{\mathbb{N}}(\mathit{gaps}(X) \quad .$$

Additionally, we need to know the support for the pattern to determine its code. We will proceed here analogously to the singletons, and encode how many patterns there are in total, how often they are used and then use the previously defined data-to-model code to transmit which distribution represents the data.

Further, by encoding the gaps for each pattern, we can compute their fills. We thus define the length of our model for the patterns by

$$L(M_P \mid O) = L(P \mid O) + L_{\mathbb{N}}(|P| + 1) + L_{\mathbb{N}}(\mathit{usage}(P) + 1) + L_U(\mathit{usage}(P), |P|) \; .$$

The length of our final model is then simply combining the model of the singletons and the patterns, $L(M \mid O) = L(M_S \mid O) + L(M_P \mid O)$. We have now found a succinct way to represent the patterns and thus our model. Let us now consider how to use these patterns to determine the length of the data.

3.2 Length of the Data given a Model

Given a model, we now need to specify how to encode the data D in bits given a model M. To do so without loss, we need three code streams. The first, the pattern stream C_P, contains codes corresponding to patterns and singletons. As a pattern may generalize the observed data, we need the specification stream C_S to reconstruct the actual entities. Last, but not least, as occurrences of

patterns may include gaps, we need the gap stream C_G to identify when we may decode the next entity of a pattern. An example of the three streams is given in Fig. 1. We call a lossless, complete representation of the data given the model an alignment. The length of this alignment is obtained by adding up all the codewords needed to represent the data.

The length of the pattern stream therewith is simply the length of the concatenation of codes used to describe the data,

$$L(C_P \mid M, O) = \sum_{X \in M} \mathit{usage}(X) L(c(X) \mid M) \ .$$

Analogue, the length of the specification stream is simply the length of all specifications from the entities in the patterns to the entities in the data,

$$L(C_S \mid C_P, M, O) = \sum_{w_i \in D} L(w_i \mid O, v_s \in C_P) \ .$$

Here v_s is the starting point which we get from the pattern stream, and w_i is the entity given in the data. Finally, for the gap stream we again simply concatenate all the gap and fill codes corresponding,

$$L(C_G \mid M, O) = \sum_{X \in P, |X|>1} \mathit{fills}(X) c_f(X) + \mathit{gaps}(X) c_g(X) \quad .$$

The length of the alignment, and thus the length of the data given the model, is then the length of all three streams together: $L(D \mid M, O) = L(C_P \mid M, O) + L(C_G \mid M, O) + L(C_S \mid C_P, M, O)$. Hence, we also know the length of our data given the alignment, the patterns and the ontology.

3.3 Formal Problem Statement

Our aim is to know the total length of our data D when encoded using some model M and ontology O. This is formalized as $L(D, M, O) = L(M \mid O) + L(D \mid M, O)$, taking into account the length of the model and the length of the data represented using the model. Resuming, the problem is formalized as finding the model for which we get the shortest description. More formal,

Minimal Ontology-based Coding Problem *Given a sequential dataset D over an alphabet Σ and an ontology O (where $\Sigma \subset O$), find the smallest model $M \in \mathcal{M}$ and best cover for D such that $L(D, M, O)$ is minimal.*

Discovering the optimal model for given data D and ontology O is hard. Given an ontology O and a maximum pattern length of n, there are $|O|^n + |O|^{n-1} + |O|^{n-2} + \cdots + |O|^2 + |O|$ different patterns. Considering arbitrarily large subsets of these patterns, and the different ways they can be ordered, ideally we would have to evaluate $|O| \times 1! + |O|^2 \times 2! + \cdots + |O|^{n-1} \times (n-1)! + |O|^n \times n!$ different models. There are exponentially many alignments for a given model [16], and as the quality of an alignment depends on the code lengths, which again depend on the alignment there is no structure we can exploit to efficiently discover the optimal alignment, nor the optimal model.

Hence, we resort to heuristics.

4 Algorithm

In this section, we present the necessary algorithms to compute the models and alignments formalized in the previous section. We will start by explaining how to cover a stream and then present our algorithm mining good code-tables. Finally we explain how we evaluate a refined model.

4.1 Covering a Stream

Given a model, we need to compute $L(D \mid M, O)$, i.e. we need to find an optimal alignment. We follow the general idea of Tatti and Vreeken [16], and go backwards through the data and compute for each pattern starting at the current position which one is the best. We then obtain the optimal alignment by considering the pattern with highest gain as first one and then iterating over the next best pattern.

This yields the optimal alignment given the current codes, which may be different from those that the globally optimal alignment would assign; whenever referred to as optimal alignment, we actually mean locally optimal alignment. As a basis, we first define a gain function that we optimize. We use the previously introduced window as $D[i,j]$ from position i till j in the data and define its length as

$$L(D[i,j],P) = \sum_{d_k,k=i}^{d_j} c'(d_k) + |X-1|\mathit{fills}(X) + (j-i-|X| \times \mathit{gaps}(X)) \quad .$$

We simplified the notation here, $c'(d_k)$ actually means

$$c'(d_k) = \begin{cases} c(w) \text{ at } d_k & \text{if } w \notin X \\ (\frac{1}{|X|}c(X)) + L(w \mid O, v_s \in w \in X) \text{ at } d_k & \text{if } w \in X \end{cases}$$

In other words, we combine the pattern that spans the given window, its specification as well as all other words contained in the gaps with their singleton codes. This includes as well how many gaps and fills we need. To define a gain, the baseline of the data is $L_0(W(\_,1,|D|)) = \sum_{w_i \in D} L(w_i \mid v_\Omega \in O))$, where every position is encoded as singleton. For a given window $D[i,j]$ and pattern X we define the gain as $\mathit{gain}(D[i,j],X) = L_0(\_,i,j) - L(X,i,j)$.

Given a set of non-overlapping windows, we can calculate the gain for the whole data by simply summing up over the different parts. The gain of the same pattern might however differ for different positions or windows in the data, since the specification and gaps might be different. Further the length of the specification can be replaced by $\frac{1}{|X|}\pi(X)$, giving preference to different properties of the pattern. We will denote this gain as $\mathit{gain}^*(D[i,j],X)$. In the following, we will always write $\mathit{gain}^*(D[i,j],X)$, independent of the use of heuristics.

Additionally, we need function $\mathit{next}(D[i,j])$ which returns the next (optimal) disjoint window from the current one. We also define $\mathit{gain}^*(\mathit{next}(D[i,j]))$ recursively in the sense that we refer not only to the gain of the window referred

Algorithm 1: Discovering the optimal alignment

Data: ordered set of windows $\mathcal{W}$
Result: alignment A for given weights

```
curopt ← 0
opt ← p_1, p_2, ... p_|D|
for i = |D| to 1 do
    for D[i, j] ∈ W do
        if gain*(D[i, j]) + gain*(next(D[i, j])) > curopt then
            opt_j ← W_i
            curopt ← gain*(D[i, j]) + gain*(next(D[i, j]))
            next(D[i, j]) ← opt_e
A ← alignment build using curopt and next()
return A
```

to by $next(D[i,j])$, called $D[i',j']$, but also to all best next windows. Thus, $gain^*(next(D[i,j])) = gain^*(next(D[i,j])) + gain^*(next(next(D[i,j])))\ldots$.

Starting with the best pattern in the beginning of the data, we iterate over $next()$ to obtain our alignment. To do so, however, we first need to run Algorithm 1 first to instantiate *next* and to know the optimal pattern in the beginning.

We give an example in Fig. 2. We start at the end of the stream, move forwards and determine for each position the window up to this position with the highest gain. Best patterns for each starting position in Fig. 2 are marked by a star. The best option need not be a window that actually starts in this position, but might start at a previous location. Yet, a previous/following pattern might yield a higher gain. This case is marked in Fig. 2 with a gray star.

Additionally, for each pattern, we also determine $next()$ to be the best pattern for the position where the patterns ends. In Fig. 2, to obtain the optimal alignment, we just use all patterns marked with a black star. When reaching the beginning of the data, we have determined the optimal $next()$ for each pattern, and we also know the first pattern with the highest gain. Traversing the patterns calling $next()$ for the first, best pattern yields our alignment.

4.2 Mining Good Models

Given an alignment, we can define new code lengths based on the frequencies in the given alignment. Different code lengths, however, influence again the alignment itself. We thus alternate these two steps, computing the alignment and updating the codes, until convergence. At this point, the question of convergence may arise. Since a frequently used pattern will receive a shorter code, it is even more likely to be picked in the next round. As a consequence, it is even more likely to be picked, and so on. For rarely used patterns, the reverse holds.

This results in an improvement of the code length at each iteration. Hence, the code length decreases monotonically and is bounded, since it can not be

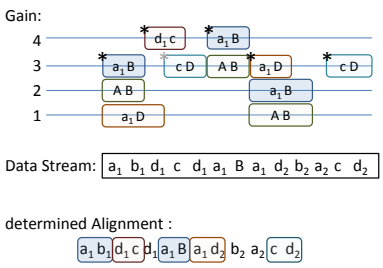

Fig. 2. Discovering Good Alignments. Given a set of patterns and their codes, we can use dynamic programming to discover the locally optimal alignment (marked with stars).

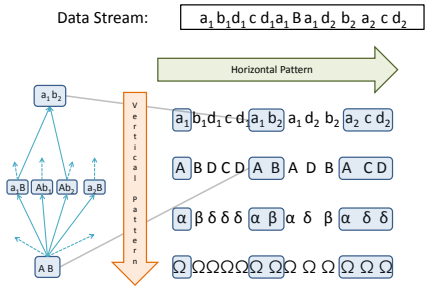

Fig. 3. Refining a pattern. Starting from a pattern AB there are many possible ways how to refine (left), both horizontally by adding entities, or vertically by making entities more or less specific.

arbitrarily short. Additionally, since there are only finitely many alignments, the algorithm converges and halts.

Refining a Model. This includes discovering new patterns, which is a complex task, as also illustrated in Fig. 3. A pattern can be either generalized or specified, e.g. we can change it vertically, or we can combine several patterns into a bigger one, which corresponds to a horizontal change. We will deal with this problem how to find new patterns later and for now assume that we have an oracle that returns promising refinements for our model.

We will first define how to evaluate patterns found. Only adding patterns will increasingly harm our model, hence we will discuss pruning it next. Since we will in both cases rely heavily on the computation of the alignment to quantify the use of a pattern, we will afterwards discuss how we can speed up the computation in this case, before we finally discuss how the model can be refined in detail.

Scoring Refinements Given some promising patterns to enhance our model, how do we decide whether we add them or not? Since we optimize $L(D, M)$, we can actually use this as a heuristic.

We test how each pattern X affects the length of the encoded data. Straightforwardly, we can build the alignment for the model without the pattern, M and $M \cup X$ and compare the two lengths. After generating a set of promising candidates, we thus add them one by one to the model and compare whether they improve $L(D, M)$. If they do, we add them to the model.

Possibly, only the combination of two patterns enhances the model, but each on its own does not. To take this into account, we can as well add all pattern at once, and check them one by one if $L(D, M) \geq L(D, M \backslash X)$ is better or equal, where equal most likely indicates that the pattern is not used.

In $L(D, M)$ we also consider $L(M)$. Patterns that are part of the model and where useful in the beginning might however be replaced by other more complex patterns, and start to harm the size of model. We thus also need to consider pruning, not only enhancing the model.

Pruning a model The model grows as it is refined. Additionally, patterns might be used less, or not at all, as they are replaced. As a consequence, we have to prune our model and exclude patterns that harm the encoded length of model and data, $L(D, M)$. Yet not all of the patterns might be refined, and are thus worth testing. We will thus only consider patterns which encoding length and thus usage changed in the model. If the model improves without them, we delete the pattern from the model and add it to the set of rejected patterns.

Speeding Up When computing the effect of adding or pruning a single pattern in the model, we do not have to recompute the whole alignment. Since large parts of the alignment will not change at all, we will briefly state how to optimize runtime for this task.

Given an old alignment and a pattern X, we first compute the set of Windows that relate to X, denoted by $\mathit{related}(X, M, W)$. This means either the windows where X is used, or, in case of a specialized/generalized pattern, where other instances of this pattern are used. In case of composed patterns, the set also contains Windows that contain parts of X. We then walk through this set of windows. In case we encounter windows containing X itself, we want to prune, and just delete the corresponding window. In the other case, given it is possible to insert a window with X, we just add it. In both case we propagate the changes and update all optimal values and return the optimal alignment build by iterating over *curopt* and *next*(), as before.

We can then compute the gain by simply comparing the length using the previously assigned patterns, say X, the newly assigned X' and further consider all patterns that where additionally changed as Y and their replacements Y',

$$\mathit{gain}(D[i,j], X') = L(X, i, j) - L(X', i', j') + \sum_{Y' \in A} \mathit{gain}(D[k,l], Y') \quad ,$$

where we recursively use *gain*() which returns the difference of using Y' in place of Y. If the replacement is a specification, the sum might be zero, since no other pattens are affected. If we refine horizontally, in contrast, it is very likely that we affect other patterns. The total gain over the whole alignment for X',

$$\mathit{gain}(X', D) = \sum_{i,j \text{ where } X'} \mathit{gain}(D[i,j], X') \quad ,$$

is then just the sum of the single parts where we replaced something by X. We do not need to take into account the other parts of the alignment that we did not change since they would rule out each other anyway.

Finding New Patterns An example is given in Fig. 3, where the refinement of a_1 is shown. In each iteration, we refine the pattern only by one step, either vertically or horizontally. For each pattern, we form a tree containing all possible

Algorithm 2: NemO

```
   Data: Data D, Ontology O
   Result: Model M
1  M ← singletons(O, D)
2  A ← align(D, M)
3  R ← ∅
4  while L(D, M′) < L(D, M) do
5  |  M′ ← M
6  |  P_new ← generate(M, A, R)
7  |  for X ∈ P_new ordered by gain(X, D) do
8  |  |  if L(D, M) ≥ L(D, M ∪ X) then
9  |  |  |  M ← M ∪ X
10 |  (M, R) ← prune(M, D, A, R)
11 |  (M, A) ← align(M, D)
12 return M
```

refinements/generalizations and their frequencies, where we expand the most frequent ones first. Before we start, we prune all combinations from our trees that were already rejected. Each time a new pattern is requested, we return the most promising combination that we have not yet investigated, where ties are broken arbitrarily. Given an updated alignment, we also update the frequencies and further prune accordingly.

4.3 NemO

We now present NemO,[3] for summarizing event sequences with serial episodes over an ontology. The previously defined algorithm are combined as formalized in Algorithm 2. Given the data D and an ontology O, we first compute a singleton model (line 1), and an alignment for this singleton model M (line 2). We then alternate refining (i.e. adding patterns and pruning, lines 6–10) model M, and re-computing the alignment for the refined model (line 11), until $L(D, M)$ does not decrease anymore (line 4).

4.4 Computational Complexity

Given the algorithm, we now briefly analyze the runtime of NemO. To find new patterns we combine each pattern in the model with all nodes from the ontology, yielding $O(|M||O|)$. To find optimal windows for each pattern we go once through the data, thus $O(D)$. To obtain an alignment we further add all found windows W, $O(|W|)$. To prune, we recompute the alignment once for each pattern, hence $O(|M||D|)$. Iteratively computing an alignment and updating codes in i iterations takes $O(i|M| + |W|)$. We repeat the whole procedure for n

[3] lit. *no one*, relating as joke to the long phase this project did not have a name.

iterations. The total complexity is thus $O(n \times (|M||O||D||W| + |M||D| + (i|M| + |W|)))$, and hence very high. In practice, however, we observe the number of iterations is typically small and the time until termination only some minutes.

5 Related Work

In this section, we will discuss other approaches that are related to NemO. A common way to find a patterns from data is frequent pattern mining [15,10,2]. Some of these approaches use ontologies, however focus on scalability by taking advantage of a Map-Reduce environment [2]. Others aim at very efficient ways to represent the ontology, in order to localize parent nodes quicker [9]. A somewhat different idea is to use several ontologies and generalize only one of them at a time [12]. By only allowing one generalization level for all patterns in one ontology, the search space is further restricted and the algorithms faster. Since the results sets are typically large, some approaches improve quality of the mined pattern sets by using heuristics or area under the curve [8,5,6]. In contrast to all these approaches, however, our method does not compute frequent patterns, but patterns that summarize the data best.

We use MDL to identify the best summarisation. Krimp was the first approach to use MDL to mine small sets of patterns from transaction data [17], and was later extended to sequential data [1]. Tatti and Vreeken proposed Sqs to summarise sequential data using serial episodes, allowing for gaps in occurrences, which increases the ability to deal with noise. Recently, Bhattacharyya and Vreeken proposed Squish, which additionally allows for interleaving patterns [4]. Wu et al.[18] presented a language specific algorithm. Neither can, however, take external information in the form of an ontology into account to summarise the data in *generalised* terms.

Bertens et al. [3] proposed Ditto, which extends Sqs to multivariate sequences. The models that Ditto discovers can contain patterns that are either identify structure in both single, as well as over multiple sequences. If we were to annotate every entity in the input data with all entities on the path to the root node of the ontology, we could theoretically use Ditto to discover generalised patterns; without knowing it explicitly, and the huge redundancy in the data, however, the patterns it would discover would rather be an attempt to reconstruct the ontology than a good summarisation of the input data.

6 Experiments

In this section, we describe experiments on both synthetic and language data and show that NemO yields promising results. We implemented NemO in Java, and make the implementation available for research purposes.[4] All experiments were ran on a Linux server with Intel Xeon E5-2643v2 processors with 64GB ram.

[4] `http://eda.mmci.uni-saarland.de/nemo/`

Data			Model					
#patterns	support	$\lvert\Sigma\rvert$	$=$	$\subset$	$\cup$	not	L%	iterations
0	0	17	0	0	0	1	19.6%	2
5	5	17	0	3	0	0	19.7%	2
5	10	17	2	4	1	0	20.3%	3
10	5	17	0	4	0	0	19.8%	2
10	10	17	5	5	0	0	21.1%	3

Table 1. Experiments on synthetic data, the ontology contains 57 elements. We plant either five or ten patterns either five or ten times and vary the size of the data. $=$ indicates a full match, $\subset$ a subset match, $\cup$ a concatenation of two patterns and *not* a pattern that was not planted. $L\%$ is the length ratio of the data compressed with the refined model compared to the original model, thus higher is better. Iterations until convergence are also given.

We start with the synthetic data, where we generate data using ten finite automatons (one for each planted pattern) to vary entities and thus require generalization to identify them. As real world data, we run NemO on the three Lord of the Rings books, Romeo and Juliet, Moby Dick and the addresses data set. In these experiments, we rely on wordnet[5] [11] and further tag our data using the Stanford POS-Tagger[6] to obtain the necessary ontology.

6.1 Synthetic data

We report our results on synthetic data of length 2 500 in Table 1. The computation time for each of this experiments is typically less than a minute. We observe that on the data generated without patterns, NemO discovers that using a model with a generalised singleton allows for better compression; this is an artifact of the generation process. We observe that patterns with a stable structure that only vary entities are almost always found. In the case of long patterns, combined by 4 or more entities, the algorithm sometimes finds both parts, but does not join them. We observe that for planted patterns that vary in length, the algorithm struggles to find them and often captures only a part.

We rerun this experiments for length 1 000 and 5 000 and observe similar results. We do, however, observe that the shorter length increases the performance of the algorithm, there are more perfect matches. This is intuitive, since there is a better ratio between structure and noise.

6.2 Text Data

We now describe the experiments on language data. Before we discuss the patterns found, we will quickly comment on the runtime and the convergence of

[5] `http://wordnet.princeton.edu/`, accessed March 2016.
[6] `http://nlp.stanford.edu/software/tagger.shtml`, v3.6.0 from September 2015.

Data		Ontology		Model				
Name	$\|D\|$	$\|v \in O\|$	|leaves|	$\|S\|$	$\|P\|$	L%	time	it
Romeo and Juliet	6 281	2 075	1 448	8	19	9.3%	< 1	2
Addresses: Obama	34 740	4 843	3 553	8	47	11.6%	< 1	2
Addresses: Bush	4 6151	5 716	4 274	9	35	10.9%	< 1	2
Lord of the Rings, 3	136 667	8 510	6 672	10	50	11%	9	6
Lord of the Rings, 2	154 410	9 186	7 225	10	104	11.7%	7	4
Lord of the Rings, 1	178 580	9 804	7 795	10	62	11.5%	5	4
Moby Dick	216 908	19 435	16 044	8	98	9.9%	12	4

Table 2. Experiments on real data. $|D|$ is the umber of entities in the data, S the set of singletons, P the set of all composed patterns. The gain when compressing with the refined model and not the empty model is $L\%$; thus higher is better. Runtime is given in minutes and *it* refers to the number of iterations needed till convergence.

the algorithm. In general, we observe that the first iteration yields the largest decrease of encoded length. In general, 2 to at most 6 iterations are needed till convergence. We further observe that in contrast to the synthetic data, we improve the encoded length only by on average 10%. Convergence, gain in encoding and number of found patterns relate in a non-trivial way to the complexity of the data.

Before we discuss some mined patterns in detail, we will quickly report on the ontologies. The average depth of the ontology constructed for text data is 2.2 to 2.65, where the first layer is always composed by 17 nodes, corresponding to the 17 categories of the tagger. The number of leaves is strongly dependent on the complexity of the data, and varies between 4 000 and 20 000.

Important Singletons. As the empty model describes every entity in the data with a path from the root node in the ontology, NemO can identify interesting generalised singletons. Examples include *Foreign word*, *Interjection*[7], in Addresses, *that* and another being *travel*[8] in Lord of the Rings, and *as* in Moby Dick.

Lord of the Rings. Looking closer at the non-singletons discovered in the Lord of the Rings books, we find grammatically correct patterns, such as *they Verb* or *he Verb* (as opposed to for example the speeches of the presidents, where *I Verb* or *we Verb* prevail). However, also more complex patterns are found. We present the following examples from the first book of the Lord of the Rings, where examples of matches sentences are given emphasized. Take into account that when analyzing the data, we ignore punctuation.

– **he Verb Conjunction he**: *He said that he [did not think Bilbo was dead.]*; *He suspects, but he [does not know – not yet.]*

[7] Exclamations such as oh! ah! ouch! and similar.

[8] Generalized from come, speed or walk in wordnet, however also directly used. This word is found in all three novels, and thus an accurate description of their content.

- **the Adjective Noun and**: *[... and] the young Hobbits, and [Boromir.]*; *[They dreaded] the dark hours, and [kept watch in pairs by night ...]*
- **Determiner _ _ Conjunction _ _ and**: *[Like a dream] the vision shifted and went back, and [he saw the trees again.]*, *[...] the language was that of Elven-song and [spoke of things little known on Middle-earth.]*

Not shown here, but also in the model are patterns matching simple grammatical structures, such as *the Adjective Noun*, *to Verb* or *he/it/they Verb*, as well as indirect speech and enumerations.

Additionally, we observe that _ is a guaranteed gap. We observe that such patterns containing _ allow to capture properties which are not obvious to humans, since they may differ a lot in meaning and seemingly in structure.

Moby Dick. Last, we consider the novel Moby Dick by Herman Melville. Example patterns we discover include

- **Pronoun Verb _ _ _ Noun**: *[And yesterday] I talked the same to Starbuck [there, ...]:[...,] I took my heavy bearskin jacket[, ...]*
- **the Adjective Noun of**: *[... one of] the old settlers of [Nantucket]*; *[... and fumbling in] the huge pockets of [his ..]*

Many patterns start with *Pronoun* and then contain varying other parts. They differ in length between six and eight. Further, nine patterns start with *he*, ranging in length from three to six, also mutating in content. Additionally, there are 18 patterns beginning with either *a* or *the* and then also differing in parts. Two examples for these patterns are presented above. Finally, we have 12 patterns which are composed of two parts, all representing intuitive English structure, such as *he Verb*, *a Adjective*, *the noun*, *in determiner*, and so on.

7 Conclusion

We considered the problem of summarizing event sequences with serial episodes over an ontology. We formalized the problem in terms of the Minimum Description Length principle, and proposed NemO, an efficient algorithm to mine good summaries directly from data.

Through extensive empirical evaluation we confirmed that NemO yields promising results in practice. On synthetic data, it recovers the ground truth and on text data we observe meaningful patterns that are beyond the surface. Of course the quality of the data influences the results we are able to achieve. Since tagging is an open research question, and current taggers might wrongly tag a word, further work in this area will alleviate this. Also the ontology used influences the results. Promising directions in future work include leveraging all parent nodes provided by wordnet, evaluating different ontologies for a single text, up to mining good ontologies directly from data.

Although the computation complexity of the summarisation problem is very high, NemO discovers good models in only minutes. It will be interesting to see how to leverage ideas from Lash [2] to scale up NemO to large collections

of text. Finally, it will be interesting to incorporate ideas from SQUISH [4], as NEMO will gain modelling power with choicisode and interleaving patterns.

Acknowledgements

The authors are supported by the Cluster of Excellence "Multimodal Computing and Interaction" within the Excellence Initiative of the German Federal Government. This work was supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0753).

References

1. R. Bathoorn, A. Koopman, and A. Siebes. Reducing the frequent pattern set. In *ICDM-Workshop*, pages 55–59. IEEE, 2006.
2. K. Beedkar and R. Gemulla. Lash: Large-scale sequence mining with hierarchies. In *SIGMOD*, pages 491–503. ACM, 2015.
3. R. Bertens, J. Vreeken, and A. Siebes. Keeping it short and simple: Summarising complex event sequences with multivariate patterns. In *KDD*, pages 735–744, 2016.
4. A. Bhattacharyya and J. Vreeken. Efficiently summarising event sequences with rich interleaving patterns. In *SDM*. SIAM, 2017.
5. B. Bringmann and A. Zimmermann. One in a million: picking the right patterns. *Knowl. Inf. Sys.*, 18(1):61–81, 2009.
6. F. Geerts, B. Goethals, and T. Mielikäinen. Tiling databases. In *DS*, pages 278–289, 2004.
7. P. Grünwald. *The Minimum Description Length Principle*. MIT Press, 2007.
8. A. J. Knobbe and E. K. Ho. Pattern teams. In *PKDD*, volume 4213, pages 577–584. Springer, 2006.
9. Y. Li, L. Sun, J. Yin, W. Bao, and M. Gu. Multi-level weighted sequential pattern mining based on prime encoding. *JDCTA*, 4(9):8–16, 2010.
10. H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Disc.*, 1(3):259–289, 1997.
11. G. Miller. WordNet: A Lexical Database for English. *CACM*, 38(11):39–41, 1995.
12. M. Plantevit, A. Laurent, D. Laurent, M. Teisseire, and Y. W. Choong. Mining multidimensional and multilevel sequential patterns. *TKDD*, 4(1), 2010.
13. J. Rissanen. Modeling by shortest data description. *Automatica*, 14(1):465–471, 1978.
14. J. Rissanen. A universal prior for integers and estimation by minimum description length. *Annals Stat.*, 11(2):416–431, 1983.
15. R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *EDBT*, March 1996.
16. 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.
17. J. Vreeken, M. van Leeuwen, and A. Siebes. KRIMP: Mining itemsets that compress. *Data Min. Knowl. Disc.*, 23(1):169–214, 2011.
18. K. Wu, J. Yu, H. Wang, and F. Cheng. Unsupervised text pattern learning using minimum description length. In *IUCS*, pages 161–166, 2010.