SUMMARISING EVENT SEQUENCES WITH SERIAL EPISODES

Jilles Vreeken and Nikolaj Tatti

Department of Mathematics and Computer Science,
University of Antwerp, Belgium,
{firstname.lastname}@ua.ac.be

ABSTRACT

The discovery of patterns is an important aspect of data mining. Data mining is the field of research concerned with the extraction of useful insight from large databases. The process of finding patterns in data is called pattern mining. A pattern can be any type of regularity in the data, such as, e.g., items are typically sold together, or events that often happen in close vicinity. An ideal outcome of pattern mining is a small set of patterns, containing no redundancy or noise, that identifies the key structure of the data.

We pursue this ideal for sequential data, employing a *pattern set* mining approach. We employ the MDL principle to identify the best set of sequential patterns, and propose two approaches for mining good pattern sets: the first algorithm selects a good pattern set from a large candidate set, while the second is a parameter-free any-time algorithm that mines pattern sets directly from the data. Experimentation on synthetic and real data demonstrates we efficiently discover small sets of informative patterns.

1. INTRODUCTION

Suppose we have an event sequence database, and are interested in its most important patterns. Traditionally, we would apply frequent pattern mining, and mine all patterns that occur at least so-many times. For non-trivial thresholds, however, by the pattern explosion we would then be buried in huge amounts of highly redundant patterns—making the patterns the problem instead of the solution.

We therefore adopt a different approach. Instead of considering patterns individually, which is where the explosion stems from, we are after the *set of patterns* that summarises the data best. Desired properties of such a summary include that it should be small, generalise the data well, and be non-redundant. To this end, we employ the Minimum Description Length principle [1], by which we can identify the best set of patterns as the set by which we can describe the data most succinctly.

This approach has been shown to be highly successful for transaction data [2], where the discovered patterns provide insight, as well as high performance in a wide range of data mining tasks, including clustering, missing value estimation, and anomaly detection.

Sequence data, however, poses additional challenges over binary data. For starters, event orders are important, and we have to take gaps in patterns into account. As such, encoding the data given a cover, finding a good cover given a set of patterns, as well as finding good sets of patterns, are all much more complicated for sequence data.

As we identify the best model by compression, and consider strings as data, standard compression approaches are related. However, although general purpose compressors provide top-notch compression, they do not result interpretable models. In our case, compression is not the goal, but a means for identifying those patterns that together describe the data most succinctly.

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

In this extended abstract we give a quick overview of SQS, only sketching the encoding and algorithms, and only report on some highlights of the empirical evaluation. For more detail, we refer the reader to [3].

2. MDL FOR EVENT SEQUENCES

As data type we consider *event sequences*. A sequence database D over an event alphabet Ω consists of $|D|$ sequences $S \in D$. Every $S \in D$ is a sequence of $|S|$ events $e \in \Omega$, i.e. $S \in \Omega^{|S|}$. We write $S[i]$ to mean the ith event in S and $S[i, j]$ to mean a subsequence $S[i] \cdots S[j]$. We denote by $||D||$ the sum of the lengths of all $S_i \in D$, i.e. $||D|| = \sum_{S_i \in D} |S_i|$. The support of an event e in S is its occurrences in S, i.e. $supp(e \mid S) = |\{i \in S | i = e\}|$, and the support of e in a database D is defined as $supp(e \mid D) = \sum_{S \in D} supp(e \mid S)$.

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

As models we consider *code tables*. A code table has four columns, one for patterns, one for pattern codes, and the latter two contain codes for indicating presence/absence

of a gap within a pattern. To ensure any sequence over Ω can be encoded by a code table, we require that all the singleton events in the alphabet, $X \in \Omega$, are included in a code table CT.

Encoding a Database

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

For $L(code_p(X))$, the lengths of pattern codes in C_p, as stored in the second column of CT, we use optimal prefix codes. Let us write $usage(X)$ for how often $code_p(X)$ occurs in C_p. That is, $usage(X) = |\{Y \in C_p \mid Y = code_p(X)\}|$. Then, the probability of $code_p(X)$ in C_p is its relative occurrence in C_p. So, we have

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

Serial episodes allow for gaps—only when we read the code for a singleton pattern X we can unambiguously append X to the decoded data. When X is a non-singleton pattern, we may only append the first symbol x_1, as before writing event x_2 of X, we need to know whether or not one or more gap events occur in between.

This is what C_g, the gap code stream, encodes. It is a list of optimal prefix codes for gap occurrences/absences within pattern embeddings. These code lengths, $L(code_g(X))$ and $L(code_n(X))$, are dependent on their relative frequency. Let us write $gaps(X)$ to refer to the number of gap events within the usage of pattern X in the cover of D. We then resp. have $fills(X) = usage(X)(|X| - 1)$, for the number of non-gaps in the usage of pattern X, and

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

for the length of a gap code within a pattern X, and analogue for $L(code_n(X) \mid CT)$.

Combining the above, we straightforwardly arrive at $L(C_p \mid CT) = \sum_{X \in CT} usage(X)L(code_p(X))$ for the encoded length of the pattern-stream, and analogously have

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

for C_g. We can then define $L(D \mid CT)$, the length of a database D given code table CT and cover C as

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

where $|D|$ is the number of sequences in D, and $|S|$ is the length of a sequence $S \in D$. To encode these values, we use $L_{\mathbb{N}}$, Rissanen's universal code for integers [4].

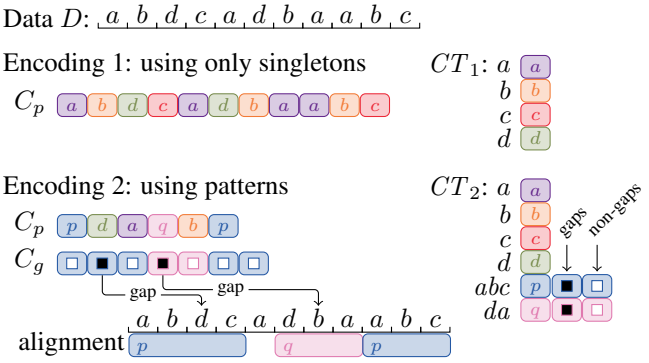

Figure 1. Toy example of two possible encodings. The first encoding uses only singletons. The second encoding uses singletons and two patterns, namely, abc and da

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

Encoding a Code Table

Next we discuss how to calculate $L(CT)$, the encoded length of a code table CT. We encode its number of entries using $L_{\mathbb{N}}$. For later use, and to avoid bias by large or small alphabets, we encode the number of singletons, $|\Omega|$, and the number of non-singleton entries, $|CT \setminus \Omega|$, separately. We disregard any non-singleton pattern with $usage(X) = 0$, as it is not used for describing the data.

The simplest valid code table consists of only singletons. We refer to this as the *standard code table*, or ST. We encode the patterns in the left-hand side column using ST, which allows us to decode up to the names of events.

The usage of $Y \in ST$ is the support of Y in D. Hence, the code length of Y in ST is defined as $L(code_p(Y) \mid ST) = -\log \frac{supp(Y|D)}{||D||}$. Before we can use these codes, the recipient needs these supports. We transmit these by the index of a number composition, the number of combinations of summing to m with n, non-zero, terms. The length in bits of such an index is $L_U(m,n) = \log \binom{m-1}{n-1}$, where for $m = 0$, and $n = 0$, we define $L_U(m,n) = 0$.

We can now reconstruct the first column of CT. To encode a pattern $X \in CT$, the number of bits is the length of X, $|X|$, and the sum of the singleton codes, i.e. $L_{\mathbb{N}}(|X|) + \sum_{x_i \in X} L(code(x_i) \mid ST)$.

Next, we encode the second column. To avoid bias, we treat the singletons and non-singleton entries of CT differently. Let us write $\mathcal{P}$ to refer to the non-singleton patterns in CT, i.e. $\mathcal{P} = CT \setminus \Omega$. For the elements of $\mathcal{P}$, we first encode the sum of their usages, denoted by $usage(\mathcal{P})$, and use L_U identify the individual usages. Together with ST, we can reconstruct all usages in CT.

This leaves the gap-codes of CT, for which we encode $gaps(X)$ using $L_{\mathbb{N}}$. The number of non-gaps then follows

from the length of a pattern X and its usage.

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

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

where $L(X, CT)$, the encoded length for the events, length, and the number of gaps of a pattern X in CT, is

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

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

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

More formally, we define the problem as follows.

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

This problem entails a large search space. First of all, there are many different ways to cover a database given a set of patterns. Second, there are many sets of serial episodes $\mathcal{P}$ we can consider. However, neither of these problems exhibits trivial structure that we can exploit for fast search, e.g. (weak) monotonicity.

3. COVERING A STRING

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

Assume we are decoding a sequence $S_k \in D$. Assume we decode the beginning of a pattern X at $S_k[i]$ and that the last symbol belonging to this instance of X is, say, $S_k[j]$. We say that $S_k[i, j]$ is an *active* window for X. Moreover, we can use FINDWINDOWS in [5] to discover all minimal windows for a pattern X in $O(|X|||D||)$.

Let $\mathcal{P}$ be the set of non-singleton patterns used by the encoding. We define an *alignment* A to be the set of all active windows for all non-singleton patterns $X \in \mathcal{P}$: $A = \{(i, j, X, k) \mid S_k[i, j]$ is an active window for $X, S_k \in D\}$. An alignment corresponding to the second encoding given in Figure 1 is $\{(1, 4, abc, 1), (6, 8, da, 1), (9, 11, abc, 1)\}$.

Note that an alignment A does not uniquely define the cover of the sequence, as it does not take into account how the intermediate symbols (if any) within the active windows of a pattern X are encoded. However, an alignment A for a sequence database D does define an equivalence class over covers of the same encoded length. In fact, given a sequence database D and an alignment A, we can determine the number of bits our encoding scheme would require, as we can distill the $usage(X)$ and $gaps(X)$ from A. As such, given an alignment A for D, we can trivially construct a valid cover C for D, simply by following A and greedily covering S_k with pattern symbols if possible, and singletons otherwise. Likewise, we can derive the associated code-optimal code table CT for A.

In [3] we show that given a code table CT, we can find the alignment of D that minimises the encoded length using the code lengths in CT. With the above, we can then calculate the optimal codes for this new alignment. By iterating these steps, we can heuristically approximate the optimal cover of D given a set of patterns $\mathcal{P}$.

4. MINING CODE TABLES

With the above, we can score the quality of a pattern set, and heuristically optimise the alignment of a pattern set. This leaves us with the problem of finding good sets of patterns. We sketch our two algorithms to do so.

4.1. Filtering Candidates

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

We sort candidates $\mathcal{F}$ ascending by $L(D, \{X\})$. We then iteratively greedily test each pattern $X \in \mathcal{F}$. If adding X to $\mathcal{P}$ improves the score, we keep X in $\mathcal{P}$, otherwise it is permanently removed.

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

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

4.2. Directly Mining Good Code Tables

The SQS-CANDIDATES algorithm requires a collection of candidate patterns to be materialised, which in practice can be troublesome; the well-known pattern explosion may prevent patterns to be mined at as low thresholds as desired.

We therefore propose an alternative strategy, that discovers good code tables directly from data. Instead of filtering a pre-mined candidate set, we now discover candidates on the fly, considering only patterns that we expect to optimise the score given the current alignment.

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

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

In [3], we show that if we take N active windows of P, and N active windows of Q, and convert them into N active windows of PQ, the difference in total encoded length can be calculated in constant time—as we know which N active windows to use: those with shortest length.

This gives the outline of SQS-SEARCH. We enumerate minimal windows of PQ from shortest to largest. At each step we compute the score using Proposition 3 of [3], and among these scores we the pick optimal one. We can do this in linear time by considering the active windows of P ascending on length, ignoring all singleton gap elements, and counting all elements occurring right after P.

To save on computation, we do not iteratively consider the estimated optimal PQ, but instead iteratively compute and rank all PQ on estimated gain, consider these in turn, and recompute once the candidate pool is depleted. Like for SQS-CANDIDATES, we apply pruning after each accepted candidate, as well as at the end of the search.

5. EXPERIMENTS

We here give a quick taste of the results obtained with SQS, and refer the interested reader to the full publication for further empirical evaluation [3].

Synthetic Data. First, we consider the synthetic *Indep*, *P10*, and *P50* datasets. Each consists of a single sequence of 10 000 events over an alphabet of 1 000. In the former, all events are independent, whereas in the latter two we planted resp. 10 and 50 patterns of 5 events 10 times each, with 10% probability of having a gap between consecutive events, but are independent otherwise.

For the *Indep* dataset, though 9 000+ episodes occur at least twice, both methods correctly identify it does not contain significant structure. For *P10* both methods return the 10 patterns. *P50* has a very high density of pattern symbols (25%). SQS-CANDIDATES and SQS-SEARCH resp. find 47 and 46 patterns exactly, plus fragments, due to partial overwrites during generation, of the others.

Real Data. In order to interpret the patterns, we consider 788 abstracts of papers from the Journal of Machine Learning Research website. The events are the stemmed words from the text, with stop words removed. We obtain compression of about 30 000 bits, with 563 and 580 patterns respectively, more than two orders of magnitude less than the number of candidates for SQS-CANDIDATES.

Table 1. JMLR data. Top-10 patterns by SQS-SEARCH

patterns	ΔL	patterns	ΔL
1. supp. vec. mach.	850	6. large scale	329
2. machine learn.	646	7. near. neighbor	322
3. state [of the] art	480	8. dec. tree	293
4. data set	446	9. neural netw.	289
5. Bayesian netw.	374	10. cross val.	279

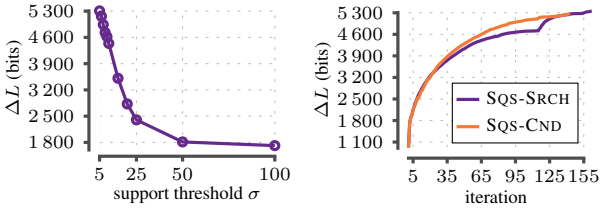

Figure 2. *Addresses* dataset, ΔL. (left) varying support thresholds for SQS-CANDIDATES. (right) SQS-CANDIDATES and SQS-SEARCH per accepted candidate.

Table 1 depicts the top-10 patterns most aiding compression, as found by SQS-SEARCH. ΔL is the increase in bits the pattern would be removed from CT. The left-hand plot of Fig. 2, for SQS-CANDIDATES, shows the gain in compression for different support thresholds. Lower thresholds, i.e richer candidate sets, allow for better models. In the right-hand plot, we compare SQS-CANDIDATES and SQS-SEARCH, showing the gain in bits over ST per candidate accepted into CT. Both search processes consider patterns aiding compression strongly first. The slight dip of SQS-SEARCH is by its batch-wise search.

6. CONCLUSION

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

7. REFERENCES

[1] Jorma Rissanen, "Modeling by shortest data description," *Automatica*, vol. 14, no. 1, pp. 465–471, 1978.

[2] Jilles Vreeken, Matthijs van Leeuwen, and Arno Siebes, "KRIMP: Mining itemsets that compress," *Data Min. Knowl. Disc.*, vol. 23, no. 1, pp. 169–214, 2011.

[3] Nikolaj Tatti and Jilles Vreeken, "The long and the short of it: Summarising event sequences with serial episodes," in *KDD*, 2012.

[4] Jorma Rissanen, "Modeling by shortest data description," *Annals Stat.*, vol. 11, no. 2, pp. 416–431, 1983.

[5] Nikolaj Tatti and Boris Cule, "Mining closed strict episodes," *Data Min. Knowl. Disc.*, 2011.