Discovering Sequential Patterns with Predictable Inter-Event Delays

Joscha Cüppers[1], Paul Krieger[2], Jilles Vreeken[1]

[1] CISPA Helmholtz Center for Information Security.
[2] Saarland University
joscha.cueppers@cispa.de, s8pakrie@stud.uni-saarland.de, vreeken@cispa.de

Abstract

Summarizing sequential data with serial episodes allows non-trivial insight into the data generating process. Existing methods penalize gaps in pattern occurrences equally, regardless of where in the pattern these occur. This results in a strong bias against patterns with long inter-event delays, and in addition that regularity in terms of delays is not rewarded or discovered—even though both aspects provide key insight.

In this paper we tackle both these problems by explicitly modeling inter-event delay distributions. That is, we are not only interested in discovering the patterns, but also in describing how many times steps typically occur between their individual events. We formalize the problem in terms of the Minimum Description Length principle, by which we say the best set of patterns is the one that compresses the data best. The resulting optimization problem does not lend itself to exact optimization, and hence we propose HOPPER to heuristically mine high quality patterns. Extensive experiments show that HOPPER efficiently recovers the ground truth, discovers meaningful patterns from real-world data, and outperforms existing methods in discovering long-delay patterns.

Introduction

Summarizing event sequences is one of the key problems in data mining. Most existing methods do so in terms of serial episodes and allow for gaps (Tatti and Vreeken 2012) and interleaving (Bhattacharyya and Vreeken 2017) of pattern occurrences. By penalizing every gap equally regardless of where in a pattern it occurs, these methods have a strong bias against long inter-event delays, whereas methods that do not penalize gaps (Fowkes and Sutton 2016) are prone to discover spurious dependencies. What both of these classes lack is a pattern to be able to specify *when* the next symbol is to be expected.

To illustrate, let us consider a toy example of a single event sequence of all national holidays of a given country over the span of multiple years. As is usual, some holidays are 'fixed' as they always occur on the same date every year, and others depend on the lunar cycle and hence 'move' around. Existing methods have no trouble finding holidays that occur right after each another, e.g. *1st Christmas Day* right before *2nd Christmas Day*, struggle with long delays, such as *Whit Monday* happening 49 days after *Easter Monday*, and outright fail when the relationship is 'far' and 'loose' such as *Easter* occurring between 82 to 114 days after *New Year's*. In this paper, we present a method that can find and describe all these types of dependencies and delays.

To do so, we propose to explicitly model the distributions of inter-event delays in pattern occurrences. That is, as patterns we do not just consider serial episodes, but also discrete distributions that model the number of time-steps between subsequent events of a pattern. This allows us to discover patterns like *New Year* $\xrightarrow{82-114}$ *Easter Monday* $\xrightarrow{49}$ *Whit Monday*, which specify there is a uniformly distributed delay of 82 to 114 days between *New Year's* and *Easter Monday*, and a fixed delay of 49 days until *Whit Monday*.

We define the problem of mining a succinct and non-redundant set of sequential patterns in terms of the Minimal Description Length Principle (MDL) (Grünwald 2007), by which we are after that model that compresses the data best. Simply put, unlike existing methods we do not plainly prefer patterns with 'compact' occurrences but rather those for which the inter-event delays are reliably predictable, no matter if these delays are short or long. This way we can automatically determine which discrete-valued distribution best characterizes the inter-event delays. In practice, we consider Uniform, Gaussian, Geometric, or Poisson distributions, but this set can be trivially extended.

The resulting problem does not lend itself for exact search, which is why we propose the effective HOPPER algorithm to efficiently discover good pattern sets in practice. Starting from just the singletons, HOPPER considers combinations of current patterns as candidates, uses an optimistic estimate to prune out unpromising candidates, explores both short and far dependencies, assigns the best-fitting delay distributions, and greedily chooses the candidate that improves the score most.

Through extensive evaluation, we show that HOPPER works well in practice. On synthetic data we demonstrate that unlike the state-of-the-art, we recover the ground truth well both in terms of patterns and delay distributions even in challenging settings where patterns include delays of hundreds of time steps. On real-world data, we show that HOPPER discovers easily interpretable patterns with meaningful delay distributions. We make all code, synthetic data, and real-world datasets available in the supplementary material.

Preliminaries

In this section, we discuss preliminaries and introduce the notation we use throughout the paper.

Notation

As data D we consider a set of $|D|$ event sequences $S \in D$ each drawn from a finite alphabet Ω of discrete events $e \in \Omega$, i.e. $S \in \Omega^{|S|}$. We write $S[i]$ to refer to the i^{th} event in S, and $||D||$ to denote the total number of events in D.

As patterns we consider serial episodes. A serial episode p is also a sequence drawn over Ω, i.e. $p \in \Omega^{|p|}$. We write $p[i]$ for the i^{th} event in p. We will model the inter-event delays between a subsequent pair of events $p[i]$ and $p[i+1]$ using discrete delay distribution $\pi_{p,i}(\cdot \mid \Theta_{p,i})$. Whenever clear from context we simply write $\pi_{p,i}(\cdot)$.

Finally, a window w^S is an ordered set of indices into S. Two windows a^S and b^S are *in conflict* iff they contain the same index, formally iff $|a^S \cap b^S| > 0$. A window w^S is said to *match* a pattern p if they identify the same events in the same order, i.e. when $\forall_{i \in [1,|p|]} S[w^S[i]] = p[i]$ and $\forall_{i \in [1,|p|-1]} \pi_{p,i}(w^S[i+1] - w^S[i]) > 0$, if p matches we write w_p^S. Whenever S is clear from context, we simply write w_p.

All logarithms are base 2 and we define $0 \log(0) = 0$.

Minimum Description Length

The Minimum Description Length (MDL) principle (Grünwald 2007) is a computable and statistically well-founded model selection principle based on Kolmogorov Complexity (Li and Vitányi 1993). For a given model class $\mathcal{M}$, it identifies the best model $M \in \mathcal{M}$ as the one minimizing $L(M) + L(D|M)$, where $L(M)$ is the length of model M and $L(D|M)$ the length of the data D given M.

This is known as two-part, or *crude* MDL, in contrast to one-part, or refined MDL (Grünwald 2007), which is not computable for arbitrary model classes. We use two-part MDL because we are particularly interested in the model. In MDL we are never concerned with materialized codes, we only care about code lengths. To use MDL we have to define a model class $\mathcal{M}$, and code length functions for the model and data given the model. We present these next.

MDL for Patterns with Predictable Delays

In this section we formally define the problem.

Decoding the Database

Before we define how to encode a sequence database using patterns with delay distributions we give the intuition, by explaining how to decode a database from a given cover. A cover C is a description of the data in terms of the patterns p in model M. Formally, a cover is defined as a tuple (C_p, C_d), where pattern stream C_p describes which pattern (windows) are used in what order, and delay stream C_d consists of the inter-event delays within those windows. Next we explain how to decode a cover C to reconstruct the encoded data.

In Figure 1 we show a toy example. We show a sequence S, a model M, and two covers of S using M.

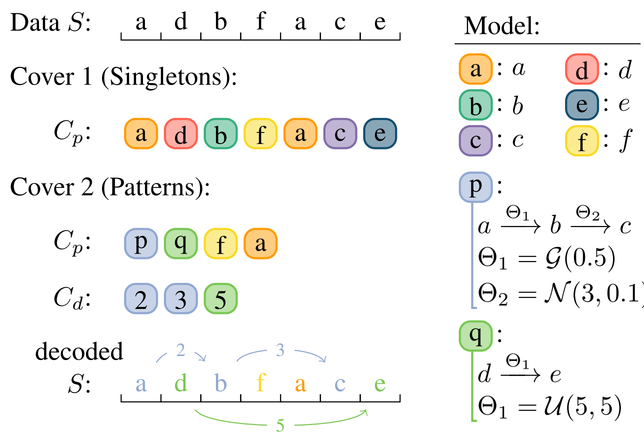


Figure 1: Toy example showing two possible encodings of the same data. Cover 1 uses only singletons, Cover 2 additionally uses two patterns, (p) and (q). A cover consist of the pattern stream C_p encoding the patterns, and the delay stream C_d encoding the inter-event delays. The first gap of pattern (p) is modeled with a geometric distribution, and the second with a normal distribution. The one gap of (q) is modeled by a uniform distribution.

We first consider Cover 1. We start by reading the first code from the pattern stream C_p. This is an (a) which we look up in M and find it encodes event 'a'. We write this to $S[0]$. We iterate reading and writing until S is decoded.

Next, we consider Cover 2. We again read the first code from C_p, which is now a (p). We look up that this stands for pattern p. We write its first symbol, a, to $S[0]$. To know where in S we should write 'b' we read a code from the delay stream C_d. We read a 2, which means we write 'b' to $S[0+2]$. We continue until we have decoded this instance of pattern p, and then read the next symbol from C_p. This is a (q). We start decoding it from the first empty position in S. We iterate until S is fully decoded.

Calculating the encoding cost

Now we know how to decode a sequence, we formally define how to compute the encoded sizes of the data and model.

Encoding the data To describe the data without loss, we need in addition to the pattern and delay streams, to know the number and length of sequences in D. We hence have

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

where we encode the numbers using the MDL-optimal encoding for integers $z \geq 1$ (Rissanen 1983). It is defined as $L_{\mathbb{N}}(z) = \log^* z + \log c_0$ where $\log^* z$ is the expansion $\log z + \log \log z + \cdots$ where we only include the positive terms. To ensure this is valid encoding, i.e. one that satisfies the Kraft inequality, we set $c_0 = 2.865064$ (Rissanen 1983).

To encode the pattern stream C_p and the delay stream C_d, we use prefix codes, which are codes that are proportional in length to their probability. For the pattern stream we have,

$$L(C_p) = \sum_{p \in M} -usg(p) \log \left(\frac{usg(p)}{\sum_{q \in M} usg(q)} \right) ,$$

where usg is the empirical frequency of pattern code $\boxed{p}$ in the pattern stream C_p. We encode the delay stream C_d similarly, encoding the inter-event delays d_j between events $p[i]$ and $p[i+1]$ of every instance of a pattern p using the corresponding delay distribution $\pi_{p,i}(d_j)$. We hence have

$$L(C_d) = \sum_{p \in M} \sum_{i=1}^{|p|-1} \sum_{j=1}^{usg(p)} -\log \pi_{p,i}(d_j) .$$

Encoding the Model As models we consider sets of patterns M that always include all singletons. We refer to the model that only consists of the singletons as the null model.

For the encoded length of a model we have

$$L(M) = L_{\mathbb{N}}(|\Omega|) + \log \binom{||D|| - 1}{|\Omega| - 1} + L_{\mathbb{N}}(|P| + 1)$$
$$+ L_{\mathbb{N}}(usg(P)) + \log \binom{usg(P) - 1}{|P| - 1} + \sum_{p \in P} L(p) ,$$

where we first encode the size of the alphabet Ω and the supports $supp(e|D)$ of each singleton event. The latter we do using a so-called data-to-model code — an index over an enumeration of all possible ways to distribute $||D||$ events over alphabet Ω (Vereshchagin and Vitányi 2004). Next, we encode the number $|P|$ of non-singleton patterns $p \in M$ and their combined usage by $L_{\mathbb{N}}()$, and then their individual usages by a data-to-model code. Finally, we encode the non-singleton patterns.

To do so we need to specify how many, and which events a pattern consists of, as well as identify and parameterize its delay distributions. To reward similarities in delay behavior, we allow a distribution to be used for multiple inter-event gaps. As a default, we equip every pattern with one Geometric delay distribution. Formally, the encoded length of a non-singleton pattern $p \in M$ is

$$L(p) = L_{\mathbb{N}}(|p|) + \log(|p| - 1) + \log \binom{|p| - 1}{k}$$
$$- \sum_{e \in p} \log \left(\frac{supp(e|D)}{||D||} \right) + \sum_{\Theta \in p} L(\Theta) ,$$

where we encode the number of events of p, then its number of delay distributions, k, and finally where in the pattern these are used. We encode the events of the pattern using prefix codes based on the supports of events e in D.

To encode a delay distribution $\pi(\cdot \mid \Theta)$ it suffices to encode Θ. For the non-default delay distributions we first encode its type out of the set $\Psi = \{Geometric, Poisson, Uniform, Normal\}$ of discrete probability functions under consideration, for which we need $-\log|\Psi|$ bits. We then encode the parameter values $\theta \in \Theta$. We use $L_{\mathbb{N}}(\theta)$ if $\theta \in \mathbb{N}$, and $L_{\mathbb{R}}(\theta)$ if $\theta \in \mathbb{R}$. We have $L_{\mathbb{R}}(\theta) = L_{\mathbb{N}}(d) + L_{\mathbb{N}}(\lceil \theta \cdot 10^d \rceil) + 1$ as the number of bits needed to encode a real number up to user-set precision p (Marx and Vreeken 2019). It does so by shifting θ by d digits, such that $\theta \cdot 10^d \geq 10^p$.

The Problem, Formally

With the above, we can now formally state the problem.

The Predictable Sequential Delay Problem *Given a sequence database D over an alphabet Ω, find the smallest pattern set P and cover C such that the total encoded size, $L(M, D) = L(M) + L(D|M)$ is minimal.*

Considering the complexity of this problem, even when we ignore delay distributions there already exist super-exponential many possible patterns, exponentially many patterns sets over those, as well as, given a pattern set there exist exponentially many covers (Bhattacharyya and Vreeken 2017). Worst of all the search space does not exhibit any structure such as (weak-)monotonicity or submodularity that we can exploit. We hence resort to heuristics.

The HOPPER Algorithm

Now we have formally defined the problem and know how to score a model we need a way to mine good models. We break the problem into two parts, finding a good cover given a model, and finding a good model, and discuss these in turn.

Finding Good Covers

Given a model, we are after that description of the data that minimizes $L(D \mid M)$. To compute $L(D \mid M)$, we need a cover C. A cover consists of a set of windows, and hence we first need to find a set of good windows.

Finding good windows Mining all possible windows for a pattern p can result in an exponential blow-up. To ensure tractability, we limit ourselves to the 100 windows per starting event with the most likely delays. To avoid wasting time on windows we will never use because they will be too costly, we restrict our search to those for which the delays fall within the 99.7% confidence-interval of the respective probability distribution. For a normal distribution, that corresponds to three standard deviations from the mean. In practice, it is extremely unlikely that we would like to include any of the not considered windows in cover C, hence these restrictions have a negligible to no effect on the results.

Selecting a good cover Armed with a set of candidate windows, we next explain how to select a set C of these that together form a good cover. Ideally, we would like to select that cover C that minimizes $L(D \mid M)$. Finding the optimal cover, however would require testing exponentially many combinations, which would, in turn, result in unfeasible runtime; we hence do it greedily. For a greedy approach we need a way to select the next window for addition. Generally speaking, we prefer long patterns with likely delays. Based on this intuition, we assign each window w_p a score $s(w_p)$. At each step we select the window w_p with the highest $s(w_p)$. If a window conflicts with a previously selected window, we skip it and proceed. We add windows until all events of D are covered. To ensure there always exist a valid cover we always include all singleton windows.

As we prefer long patterns with likely deltas, our window score trades of pattern length $(|p|c)$ against the cost of the individual delays. Formally, we have

$$s(w_p) = |p|c - \sum_{k=1}^{|p|-1} -\log \pi_{p,k}(w_p[k+1] - w_p[k])$$

where c is the average code cost of a singleton event under the null model, that is

$$c = \frac{\sum_{e \in \Omega} -supp(e|D) \log(supp(e|D)/||D||)}{||D||} \quad .$$

Mining Good Models

Now that we know how to find a good cover given a set of patterns, we explain how to discover a high-quality pattern set. Since there are super-exponentially many possible solutions, we again take a greedy approach. The general idea is that we use a pattern-growth strategy in which we iteratively combine existing patterns into new longer patterns. Before we explain our method in detail, we explain how we build a pattern candidate given two existing patterns and how to estimate the gain of such a candidate.

Estimating Candidate Gains Computing the total encoded length $L(M \oplus p', D)$ for when we add a new pattern p' to M is costly as this requires covering the data, which in turn requires finding good windows of p'. To avoid doing so for all candidates, we propose to instead use an optimistic estimator to discard those candidates for which we estimate no gain. Specifically, we want to estimate how many bits we will *gain* if we were to add pattern p' to the model.

To do so, we estimate the usage of p'. As we will explain below, every candidate p' is constructed by concatenating two existing patterns $p_1, p_2 \in M$. Assuming that p' will be used maximally, we have an optimistic estimate of its usage as $usg(p') = \min(usg(p_1), usg(p_2))$, or, if $p_1 = p_2$ as $usg(p') = usg(p_1)/2$. We estimate the change in model cost by adding p' by assuming all occurrences of the least frequent parent pattern are now covered by p'. Combined the estimated gain is,

$$\Delta\overline{L}(M \oplus p') = -\hat{L}(p') + L(\operatorname*{argmin}_{p \in \{p_1, p_2\}} usg(p))$$

where $\hat{L}(p')$ is the cost of p' omitting the delay distribution between p_1 and p_2. We estimate $\Delta L(D \mid M \oplus p')$ as

$$\begin{aligned}\Delta\overline{L}(D \mid M \oplus p') = s\log(s) - s'\log(s') + z\log(z) - \\ x\log(x) + x'\log(x') - y\log(y) + y'\log(y')\end{aligned}$$

where s is the sum of all usages, $s = \sum_{p \in M} usg(p)$, and, for readability, we shorten $usg(p')$ to z, $usg(p_1)$ to x, $usg(p_2)$ to y and write x', y', s' for the "updated" usages, that is $x' = x - z, y' = y - z$ and $s' = s - z$.

As we do not have any information about the delays between p_1 and p_2 we assume these are encoded for free. Putting the above together gives us an optimistic estimate of the total encoded cost when adding pattern p' to M as

$$\Delta\overline{L}(D, M \oplus p') = \Delta\overline{L}(M \oplus p') + \Delta\overline{L}(D \mid M \oplus p') \; .$$

Wherever clear from context, we simply write $\Delta\overline{L}(p')$.

Algorithm 1: OPTIMIZEALIGNMENT

Input : pattern candidate p', alignment A
Output: estimated $gain^*$, optimized alignment A^*

```
gain* ← −∞
while ΔL̄_A(p') > gain* do
    gain* ← ΔL̄_A(p')
    A* ← A
    drop all delays d with minimal frequency from A
return gain*, A*
```

Estimating Candidate Occurrences When we want to evaluate a candidate pattern p', constructed from patterns p_1 and p_2, we have to determine its occurrence windows. A simple and crude way to determine candidate windows is by mapping every occurrence of p_1 to the nearest next occurence of p_2. We call this procedure ALIGNNEXT. It is particularly good for finding a mapping with the shortest possible delays, but will not do well when delays are relatively long. For this, the ALIGNFAR algorithm by Cüppers, Kalofolias, and Vreeken (2022) provides a better solution. In a nutshell, it efficiently discover that mapping A that minimizes the variance in delays. By a much larger search space it is naturally more susceptible to noise.

As a result, both strategies can give a good starting points, but neither will likely give an alignment that optimizes our MDL score. We propose to greedily optimize these mappings using an optimistic estimate. We first observe that given a mapping, we can trivially compute the delays, on which we can then fit a distribution. We do so for all distributions $\pi \in \Psi$ and choose that $\pi^*_{p'}(\cdot \mid \Theta^*)$ that minimizes the cost of encoding the delays. Second, we observe that a mapping also allows us to better estimate the usage of p' as the number of mapped occurrences of p_1 and p_2. This gives a gain estimate under alignment A as

$$\Delta\overline{L}_A(p') = -L(p') + \Delta\overline{L}(D \mid M \oplus p') + \sum_{d \in A} \log \pi(d|\Theta^*).$$

We now use this estimate to identify and remove those mappings with the lowest delay probability (i.e. those with minimal frequency) until $\Delta\overline{L}_A(p')$ no longer increases. We give the pseudocode as Algorithm 1.

Mining Good Pattern Sets Next, we explain how we use the gain estimation and cover strategy to mine good pattern sets P. We give the pseudo-code for our method, HOPPER, as Algorithm 2. The key idea is to use a bottom-up approach and iteratively combine previously found patterns into longer ones.

We iteratively consider the Cartesian product of patterns $p_1, p_2 \in M$ as candidates. We evaluate these in order of potential gain. Events and patterns that occur frequently have the largest potential to compress the data, therefore we consider these combinations first. Specifically, we evaluate combinations of p_1 and p_2 in order of how many events they together currently cover (line 2).

Given a pattern candidate $p' = p_1 \oplus p_2$, we use our optimistic estimator to determine if we expect it to provide any

Algorithm 2: HOPPER

Input : sequence database D, alphabet Ω
Output: model M

1 $CT \leftarrow \Omega$; $\quad Cand \leftarrow CT \times CT$;
2 **forall** $p_1, p_2 \in Cand$ **do** ordered descending on $|p_1|usg(p_1) + |p_2|usg(p_2)$
3   **if** $\Delta\overline{L}(p_1 \oplus p_2) > 0$ **:**
4         $gain, p' \leftarrow$ ALIGNCANDIDATE(p_1, p_2)
5   **if** $gain > 0 \land L(D, M) > L(D, M \oplus p')$ **:**
6             $p' \leftarrow$ FILLGAPS$(p', |p_1|)$
7             $M \leftarrow M \oplus p'$
8             $M \leftarrow$ PRUNE(M)
9             $Cand \leftarrow Cand \cup \{M \times p', (p_1, p_2)\}$
10 $M \leftarrow$ PRUNEINSIGNIFICANT(M)
11 **return** M

gain in compression. If not, we move on to the next candidate. If we do estimate a gain based on usage of p_1 and p_2 alone, we proceed and optimize the alignment of occurrences of p_1 and p_2 to those of occurrences of p'. We do so using ALIGNCANDIDATE, for which we give the pseudocode in the supplementary. In a nutshell, it returns the best optimized result out of ALIGNNEXTand ALIGNFAR.

If the alignment leads to an estimated gain, we compute our score exactly (l. 5) and if the score improves we are safe to add p' to our model. We do so after we consider augmentations of p' with events that occur between p_1 and p_2 (FILLGAPS, line 6) such that we further improve the score. Adding a new pattern to M can make previously added patterns redundant, e.g. when all occurrences of p_1 are now covered by p'. We prune all patterns for which the score improves when we remove them from M (PRUNE). Finally, we create new candidates based on the just added pattern, and add (p_1, p_2) back to the candidate set, as we might want to build a different pattern from it in a later iteration.

Before returning the final pattern set, we reconsider all patterns in the model and only keep those that give us a significant gain (Bloem and de Rooij 2020; Grünwald 2007) in compression. We provide further details on the pattern mining procedure in the supplementary.

As we consider the most promising candidates first, the more candidates we evaluate to have no gain, the more unlikely it becomes we will find a candidate that will provide any substantial gain. To avoid evaluating all of those unnecessarily, we propose an early stopping criterion by considering up to $|\Omega|^2/100$, but at least $1\,000$, unsuccessful candidates in a row. As our score is bounded from below by 0, we know that Hopper will eventually converge.

Related Work

Mining sequential patterns from event sequences has a rich history. Traditional sequential pattern miners focus on finding all frequent patterns (Agrawal and Srikant 1995; Laxman, Sastry, and Unnikrishnan 2007), these suffer from exponentially many patterns, making interpretation hard to impossible. Closed episodes (Yan, Han, and Afshar 2003; Wang and Han 2004) partially solve this, but are highly sensitive to noise. More recently, research focus shifted to mining patterns with a frequency that is significant with respect to some null hypothesis (Low-Kam et al. 2013; Petitjean et al. 2016; Tonon and Vandin 2019; Jenkins, Walzer-Goldfeld, and Riondato 2022). While this alleviates, it does not solve the pattern explosion.

Pattern set mining solves the pattern explosion by asking for a small and non-redundant set of patterns that generalizes the data well, as instead of asking for *all* patterns that satisfy some individual criterion. There exist different approaches to how to score a pattern set. ISM (Fowkes and Sutton 2016) takes a probabilistic Bayesian approach, unlike us they do not model gaps. SQS (Tatti and Vreeken 2012) is an example of a method that employs the Minimum Description Length principle to identify the best set of serial episodes, which are sequential patterns that allow for gaps. SQUISH (Bhattacharyya and Vreeken 2017) builds upon SQS and additionally allows interleaved and nested patterns. However, SQS and SQUISH, are not capable of finding patterns with long inter-event delays and penalize each individual gap uniformly, regardless where in the pattern it occurs.

Existing methods that enrich patterns with delays can be categorized into two groups, methods that discover frequent patterns that satisfy some user set delay constrains (Yoshida et al. 2000; Giannotti et al. 2006; Dauxais et al. 2017; Cram, Mathern, and Mille 2012), and methods that discovers delay information from the data (Yen and Lee 2013; Nanni and Rigotti 2007). The latter, in contrast to our method, only consider the minimal delay between events, do not work on a single long sequence, and mine all frequent patterns, and hence also suffer from the pattern explosion.

Existing pattern set miners that do model the inter-event delay solve different problems. Galbrun et al. (2018) propose to mine *periodic* patterns, which are patterns that continuously appear throughout the data with near-exact delays. It is therewith well-suited for the holidays example in the introduction, but less so for discovering patterns that only appear more locally. OMEN (Cüppers, Kalofolias, and Vreeken 2022) does discover local patterns and delay distributions, but does so in a supervised setup between a pattern and a target attribute of interest. As such, each of the above methods consider part of the problem we study here, but none address it directly: we aim to discover a small set of sequential patterns where the delays between subsequent events in a pattern are modelled with a probability distribution.

Experiments

In this section we empirically evaluate HOPPER on synthetic and real-world data. We implement HOPPER in Python and provide the source code along with the synthetic data and the real-world data in the supplementary.[1] We compare HOPPER to SKOPUS (Petitjean et al. 2016) as a representative statistically significant sequential pattern miner, SQS (Tatti and Vreeken 2012), SQUISH (Bhattacharyya and Vreeken 2017) and ISM (Fowkes and Sutton 2016) as representatives of the

[1] eda.rg.cispa.io/prj/hopper

general class of pattern set miners, and to PPM (Galbrun et al. 2018) as a representative of the periodic pattern miners. For all, we use the implementation by the authors.

HOPPER considers delays up to a user set *max delay*, for all experiments we set it to 200. SKOPUS only works on a set of sequences, when the dataset consists of one sequence, we split the sequence into 100 equally long sequences. We parametrize SKOPUS to report the top 10 patterns of at most length 10, which corresponds to the ground-truth value in our synthetic experiments. PPM only accepts a single sequence as input, to make it applicable on databases of multiple sequences, we concatenate these into one long sequence. We give the full setup description in the supplementary.

Synthetic Data

To evaluate how well HOPPER recovers patterns with known ground, we consider synthetic data. To this end, we generate data as follows. For each synthetic configuration we generate 20 independent datasets. For each dataset we sample uniform at random one sequence of length 10000 over an alphabet of 500 events, we plant 10 unique patterns, uniformly, at random locations while avoiding collisions. The frequency of planted patterns, length and delay distributions between events we vary per experiment.

As evaluation we consider standard F1 score. Where, to reward partial discovery, we weight a reported pattern p_r by the relative edit distance to the planted pattern p_p, that is, $w(p_r, p_p) = \max(1 - lev(p_r, p_p)/|p_p|, 0)$ where lev is the Levenshtein edit distance. Since we do not want to reward redundant discoveries we cap the total reward to one per planted pattern. To illustrate, consider the example where we plant one pattern *abcd*, and discover two patterns, *ab* and *cd*. We value both as 0.5. As we can technically reconstruct the generating pattern we hence have a recall of one, but, as we have to do so using two rather than one pattern, we have a precision of 0.5. This way we reward partial discoveries, which is especially relevant for methods that are designed to pick up events that occur close to one another, but might miss the full pattern if it includes a long delay. We provide additional details on the evaluation in the supplementary.

Sanity Check We start with a sanity check, where we run HOPPER on 20 data sets without structure, generated uniformly at random. It correctly does not report any patterns.

Delay Distributions Next, we test how well HOPPER can recover patterns for varying numbers of delay distributions. We consider the case of no delay distributions up to a pattern including a delay distribution between every subsequent pair of events. We plant 10 unique patterns of length 10 and in total 200 pattern occurrences, that is, on expectation 20 instances per pattern. As delay distribution, we plant Uniform distributions with a delay of between 10 to 20 time steps.

We present the results in the first panel of Fig. 2. We observe that HOPPER performs on par when there are no delay distributions and outperforms the state of the art when we increase their number. We find that SQUISH performs on par with SQS in our experiments and to avoid clutter from here onward postpone its results to the supplementary.

Low Frequency Next, we evaluate performance with low-frequency patterns, we decrease the frequency of the total number of planted patterns. We consider the same setting as above, where we set the number of distributions to four and decrease the total number of planted patterns from 200 to 100, that is, on expectation, from 20 to 10 per pattern. We show the results in the 2nd panel of Fig. 2. We observe that HOPPER outperforms all other methods, ultimately reducing to the performance of SQS in the low-frequency domain.

Long Delays Next, we investigate how robust HOPPER is to long delays, to this end we plant 10 patterns at 200 locations. We plant patterns of length 3, with Normal distributed inter-event delays, with a standard deviation of one, and increase the mean stepwise from 1 to 180. We present the results in the third panel of Fig. 2. We observe HOPPER is very robust against long delays: even with an expected delay of 180 between the individual events it achieves a very high F1 score. In contrast, its competitors do not fare well; SQS and SKOPUS perform well initially but then quickly deteriorate.

High Variance Finally, we evaluate HOPPER under increasing variance of inter-event delays. To this end we plant 400 occurrences of 10 patterns of length 3, with Normally distributed delays with mean 50 and varying the standard deviations. We show the results in the last panel of Fig. 2.

We observe that HOPPER gets near perfect results for lower variance and high F1 score until a standard deviation of 7 at which point 95% percent of the probability mass is distributed over a range of 28 timestamps. In general, we observe that the higher the frequency, the more robust we are against higher variance. We can see that SKOPUS is consistent under increasing variance. This is probably due to the fact that SKOPUS does not care about the distance between events only about the order in which they occur.

Real World Results

Next, we evaluate Hopper on real-world data. We use eight datasets that together span a wide range of use-cases. We consider a dataset of all national *Holidays* in a European country over a century, the playlist a local *Radio* station recorded over a month, the *Lifelog*[2] of all activities of one person recorded in over seven years, the MIDI data of hundred Bach *Chorales* (Dua and Graff 2017), all commits to the *Samba* project for over ten years (Galbrun et al. 2018), the *Rolling Mill* production log of steel manufacturing plant (Wiegand, Klakow, and Vreeken 2021), the discretized muscle activations of professional ice *Skating* riders (Moerchen and Fradkin 2010), and finally, three text datasets the Gutenberg project, resp. *Romeo* and Juliet by Shakespeare, A *Room* with a View by E.M. Forster, and The Great *Gatsby* by F. Scott Fitzgerald. We give the total number of events per dataset in Table 1 and further statistics in the supplementary.

We run HOPPER, SQS, ISM, PPM, and SKOPUS on all datasets. We report the number of patterns ($|P|$), the average expected distance between the first and last event ($\mathbb{E}(w_p[|p|] - w_p[0])$) and for HOPPER, the number of discovered delay distributions ($\#\Theta$). In the interest of space

[2]https://quantifiedawesome.com/

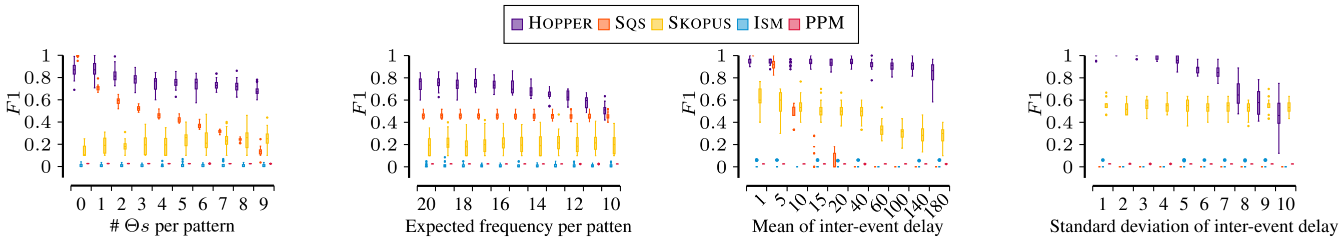


Figure 2: [Higher is better] F1 scores for recovering patterns from synthetic data. From left to right, we evaluate for varying numbers of inter-event distributions, expected frequency of a pattern, mean inter-event delay, resp. different standard deviations for normally distributed delays. We see that HOPPER performs on par with SQS when inter-event delays are few and simply structured, and outperforms the competition with a large margin whenever their structure is more complicated.

		HOPPER			SQS		PPM	
Dataset	$\|\|D\|\|$	$\|P\|$	$\#\Theta$	$\mathbb{E}(w)$	$\|P\|$	$\mathbb{E}(w)$	$\|P\|$	$\mathbb{E}(w)$
Holidays	37k	1	7	393	3	19.2	14	51.6
Radio	16k	22	43	48	15	5.8	587	71.9
Lifelog	40k	37	68	129	58	3.9	1.6k	119.1
Samba	29k	40	101	110	221	2.7	1.4k	17.1
Chorales	7k	56	57	4.7	114	2.6	433	2.6
Rolling	54k	237	489	7.4	470	5.0	3.6k	181.9
Skating	26k	86	160	9.1	160	4.0	1.4k	55.5
Romeo	37k	254	284	12.9	254	2.8	2.3k	332.7
Room	87k	565	610	3.1	701	2.5	–	–
Gatsby	64k	439	488	7.3	519	2.6	4.7k	641.9

Table 1: Results on real-world data. For HOPPER, SQS, and PPM we report the number of discovered patterns ($|P|$) and the average expected distance between the first and last symbol of a pattern ($\mathbb{E}(w)$). For HOPPER we additionally give total number of discovered inter-event distributions ($\#\Theta$).

we postpone the results of ISM and SKOPUS, along with the metrics runtime and average events per pattern to the supplementary. HOPPER terminates within seconds to hours, depending on the dataset. We find that while HOPPER and SQS discover similar numbers of patterns, those that HOPPER discovers reveal much longer range dependencies and, in general, include more events. PPM results in an order of magnitude more patterns, most of which are singletons. Next we look at the results for *Holidays* and *Radio* in more detail.

On the *Holidays* dataset, HOPPER finds a single pattern, *May 1st* $\xrightarrow{155}$ *National Holiday* $\xrightarrow{83}$ *1st Christmas Day* $\xrightarrow{1}$ *2nd Christmas Day* $\xrightarrow{6}$ *New Year* $\xrightarrow{80-112}$ *Good Friday* $\xrightarrow{3}$ *Easter Monday* $\xrightarrow{49}$ *Whit Monday*, where all delay distributions are uniform. The pattern precisely describes all fixed and all lunar-calendar dependent holidays within the year. In contrast, the competing methods only find fractions of this pattern, such as *1st Christmas Day, 2nd Christmas Day*. We show the results for all methods in the supplementary.

The *Radio* dataset includes all the songs played, as well as the ad slots and news segments, for a local radio station over the course of a month. On this data, HOPPER discovers the pattern *Jingle* $\xrightarrow{0}$ *Ads* $\xrightarrow{0}$ *News* $\xrightarrow{0}$ *Jingle* $\xrightarrow{\mathcal{U}(3,5)}$ *Jingle* where the 0-gaps correspond to geometric distributions with $p = 1$ and the last inter-event delay is a uniform distribution. Other methods find comparable or parts of this patterns, but none give the immediate insight that the first four events follow directly after one another and the last *Jingle* plays between 3 to 5 events after the previous.

More importantly, unlike other methods, HOPPER also picks up patterns such as *Solo Para* $\xrightarrow{\mathcal{G}(0.02)}$ *As It Was* $\xrightarrow{\mathcal{N}(48,25)}$ *I Believe* $\xrightarrow{\mathcal{P}(24)}$ *Anyone for You* that confirm our suspicion that radio stations often play the same sequence of particularly popular songs interspersed with less-well-known songs. No other method finds any comparable patterns. HOPPER discovers much longer patterns than its competitors. Whereas most competitors find patterns of length 2, SQS patterns of at most 4 events, HOPPER discovers patterns of up to 7 events long. Together, this illustrates that HOPPER finds patterns that are not only more detailed in terms of the delay structure, but also in which events they describe.

Conclusion

We consider the problem of summarizing sequential data with a small set of patterns with inter-event delays. We formalized the problem in terms of the Minimum Description Length principle and presented the greedy HOPPER algorithm. On synthetic data we saw that our method recovers the ground truth well and is robust against high delays and variance. On real-world data we observed that HOPPER finds meaningful patterns that go beyond what state of the art methods can capture. While methods that only consider the order of events, can in theory find patterns with long delays, they often do not do this in practice.

We introduce a more powerful pattern language that enables us to discover new structure in data. This comes with the trade-off, of a much larger search space and, in theory, makes us more susceptible to noise, however the experiments have shown that this is not a problem in practice. HOPPER achieves a high F1 score on all experiments in Fig. 2, despite these having 80% or more noise.

Currently, we model the delay between subsequent events in a pattern. In practice, some events may depend on some event earlier in the pattern. We see it as an interesting direction for future work to extend our pattern language to include rule-like dependencies.

References

Agrawal, R.; and Srikant, R. 1995. Mining sequential patterns. In *ICDE*, 3–14. Los Alamitos, CA, USA: IEEE Computer Society.

Bertens, R.; Vreeken, J.; and Siebes, A. 2016. Keeping it Short and Simple: Summarising Complex Event Sequences with Multivariate Patterns. In *KDD*, 735–744.

Bhattacharyya, A.; and Vreeken, J. 2017. Squish: Efficiently Summarising Event Sequences with Rich Interleaving Patterns. In *SDM*, 11.

Bloem, P.; and de Rooij, S. 2020. Large-Scale Network Motif Analysis Using Compression. *DAMI*, 34: 1421–1453.

Cram, D.; Mathern, B.; and Mille, A. 2012. A Complete Chronicle Discovery Approach: Application to Activity Analysis. *Expert Systems*, 29(4): 321–346.

Cüppers, J.; Kalofolias, J.; and Vreeken, J. 2022. Omen: Discovering Sequential Patterns with Reliable Prediction Delays. *KAIS*, 64(4): 1013–1045.

Dauxais, Y.; Guyet, T.; Gross-Amblard, D.; and Happe, A. 2017. Discriminant Chronicles Mining: Application to Care Pathways Analytics. In *AIME*, 234–244. Springer.

Dua, D.; and Graff, C. 2017. UCI Machine Learning Repository.

Edmonds, J.; and Karp, R. M. 1972. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM*, 19(2): 248–264.

Fowkes, J.; and Sutton, C. 2016. A Subsequence Interleaving Model for Sequential Pattern Mining. In *KDD*.

Galbrun, E.; Cellier, P.; Tatti, N.; Termier, A.; and Crémilleux, B. 2018. Mining Periodic Patterns with a MDL Criterion. In *ECMLPKDD18*, 535–551. Springer.

Giannotti, F.; Nanni, M.; Pedreschi, D.; and Pinelli, F. 2006. Mining Sequences with Temporal Annotations. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, 593–597. Dijon France: ACM. ISBN 978-1-59593-108-5.

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

Jenkins, S.; Walzer-Goldfeld, S.; and Riondato, M. 2022. SPEck: Mining Statistically-Significant Sequential Patterns Efficiently with Exact Sampling. *Data Min Knowl Disc*, 36(4): 1575–1599.

Laxman, S.; Sastry, P. S.; and Unnikrishnan, K. P. 2007. A Fast Algorithm for Finding Frequent Episodes in Event Streams. In *KDD07*, 410–419. ACM.

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

Low-Kam, C.; Raissi, C.; Kaytoue, M.; and Pei, J. 2013. Mining Statistically Significant Sequential Patterns. In *ICDM*, 488–497. Dallas, TX, USA: IEEE. ISBN 978-0-7695-5108-1.

Marx, A.; and Vreeken, J. 2019. Telling Cause from Effect by Local and Global Regression. *KAIS*, 60: 1277–1305.

Moerchen, F.; and Fradkin, D. 2010. Robust Mining of Time Intervals with Semi-Interval Partial Order Patterns. In *SDM*, 315–326.

Nanni, M.; and Rigotti, C. 2007. Extracting Trees of Quantitative Serial Episodes. In Džeroski, S.; and Struyf, J., eds., *Knowledge Discovery in Inductive Databases*, volume 4747, 170–188. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-75548-7.

Petitjean, F.; Li, T.; Tatti, N.; and Webb, G. I. 2016. Skopus: Mining Top-k Sequential Patterns under Leverage. *DAMI*, 30(5): 1086–1111.

Rissanen, J. 1983. A Universal Prior for Integers and Estimation by Minimum Description Length. *Annals Stat.*, 11(2): 416–431.

Tatti, N.; and Vreeken, J. 2012. The Long and the Short of It: Summarizing Event Sequences with Serial Episodes. In *KDD*, 462–470. ACM.

Tonon, A.; and Vandin, F. 2019. Permutation Strategies for Mining Significant Sequential Patterns. In *ICDM*, 1330–1335. Beijing, China: IEEE. ISBN 978-1-72814-604-1.

Vereshchagin, N. K.; and Vitányi, P. M. B. 2004. Kolmogorov's Structure Functions and Model Selection. *IEEE Transactions on Information Theory*, 50(12): 3265–3290.

Wang, J.; and Han, J. 2004. BIDE: Efficient Mining of Frequent Closed Sequences. In *ICDE*, 79–90.

Wiegand, B.; Klakow, D.; and Vreeken, J. 2021. Mining Easily Understandable Models from Complex Event Logs. In *SDM*, 10.

Yan, X.; Han, J.; and Afshar, R. 2003. CloSpan: Mining: Closed Sequential Patterns in Large Datasets. In *SDM*, 166–177. SIAM.

Yen, S.-J.; and Lee, Y.-S. 2013. Mining Non-Redundant Time-Gap Sequential Patterns. *Applied Intelligence*, 39(4): 727–738.

Yoshida, M.; Iizuka, T.; Shiohara, H.; and Ishiguro, M. 2000. Mining Sequential Patterns Including Time Intervals. In Dasarathy, B. V., ed., *AeroSense 2000*, 213–220. Orlando, FL.

Algorithm

In this section, we provide the derivation of the gain estimation as well as further details about the search algorithm.

Estimating Candidate Gain

In this subsection, we provide the derivation of

$$\Delta\overline{L}(D \mid M \oplus p') = s\log(s) - s'\log(s') + z\log(z) - x\log(x) + x'\log(x') - y\log(y) + y'\log(y')$$

where z is the assumed usage of p' and s the sum of all usages, $s = \sum_{p \in M} usg(p)$, for readability, we shorten $usg(p_1)$ to x, $usg(p_2)$ to y and write x', y', s' for the "updated" usages, that is $x' = x - z$, $y' = y - z$ and $s' = s - z$ (Bertens, Vreeken, and Siebes 2016).

We want to compute the difference in encoding cost induced by adding pattern p' with assumed usage z to model M,

$$\Delta\overline{L}(D \mid M \oplus p') = L(D \mid M) - L(D \mid M \oplus p') \quad .$$

As we do not have any information about the delays between p_1 and p_2 we assume these are encoded for free, this makes it a more optimistic estimation. As the constant costs will not affect the difference we do not consider them here, therefore we estimate the change in the pattern stream as,

$$\begin{aligned}
&\approx L(C_p \mid M) - L(C_p \mid M \oplus p') \\
&= \sum_{p \in M} -usg(p)\log\left(\frac{usg(p)}{s}\right) - \sum_{p \in M \oplus p'} -usg(p)\log\left(\frac{usg(p)}{s'}\right) \\
&= \left(\sum_{p \in M \setminus \{p_1, p_2\}} -usg(p)\log\left(\frac{usg(p)}{s}\right)\right) - x\log\left(\frac{x}{s}\right) - y\log\left(\frac{y}{s}\right) \\
&\quad - \left(\sum_{p \in M \oplus p' \setminus \{p_1, p_2, p'\}} -usg(p)\log\left(\frac{usg(p)}{s'}\right)\right) \\
&\quad + x'\log\left(\frac{x'}{s'}\right) + y'\log\left(\frac{y'}{s'}\right) + z\log\left(\frac{z}{s'}\right)
\end{aligned}$$

$$\text{since} \sum_{p \in M \setminus \{p_1, p_2\}} usg(p) = s - x - y \quad \text{and} \sum_{p \in M \setminus \{p_1, p_2, p'\}} usg(p) = s' - x' - y' - z$$

$$\begin{aligned}
&= (s - x - y)\log(s) + \cancel{\sum_{p \in M \setminus \{p_1, p_2\}} -usg(p)\log(usg(p))} - x\log\left(\frac{x}{s}\right) - y\log\left(\frac{y}{s}\right) \\
&\quad - (s' - x' - y' - z)\log(s') - \cancel{\sum_{p \in M \oplus p' \setminus \{p_1, p_2, p'\}} -usg(p)\log(usg(p))} \\
&\quad + x'\log\left(\frac{x'}{s'}\right) + y'\log\left(\frac{y'}{s'}\right) + z\log\left(\frac{z}{s'}\right) \\
&= (s - x - y)\log(s) - x\log\left(\frac{x}{s}\right) - y\log\left(\frac{y}{s}\right) \\
&\quad - (s' - x' - y' - z)\log(s') + x'\log\left(\frac{x'}{s'}\right) + y'\log\left(\frac{y'}{s'}\right) + z\log\left(\frac{z}{s'}\right) \\
&= s\log(s) \cancel{- x\log(s)} \cancel{- y\log(s)} - x\log(x) \cancel{+ x\log(s)} - y\log(y) \cancel{+ y\log(s)} \\
&\quad - s'\log(s') \cancel{+ x'\log(s')} \cancel{+ y'\log(s')} \cancel{+ z\log(s')} \\
&\quad + x'\log(x') \cancel{- x'\log(s')} + y'\log(y') \cancel{- y'\log(s')} + z\log(z) \cancel{- z\log(s')} \\
&= s\log(s) - x\log(x) - y\log(y) \\
&\quad - s'\log(s') + x'\log(x') + y'\log(y') + z\log(z)
\end{aligned}$$

Mining Good Models

AlignFar To find a good initial alignment between p_1 and p_2, we use ALIGNFAR (Cüppers, Kalofolias, and Vreeken 2022). However, we can not use ALIGNFAR directly, we first have to transform the data. We first explain the functionality ALIGNFAR and then how we transform our data.

AlignFar takes a set of sets I as input, each set $U \in I$ is a set of positive integers i.e. $U \in \mathbb{N}^+$. AlignFar finds that $\mu^* \in \mathbb{R}$ that minimizes the squared difference to the closest $d \in U$ over all sets $U \in I$, formally that is

$$\mu^* = \operatorname*{argmin}_{\mu \in \mathbb{R}} \sum_{U \in I} \min_{d \in U} (\mu - d)^2 \quad .$$

Given a set of windows of p_1 and a second set of windows of p_2, we are after that alignment, between the last event of the windows of p_1 and the first of p_2, that minimizes delay variance. For each window of pattern p_1 we build a set of delays U to all following p_2 windows, that is all who are in the same sequence S. If $|U| = 0$ i.e. there does not exist a occurrence of p_2 we can align the respective occurrence of p_1 to, we omit this U from I.

This gives us a set of delay sets I. AlignFar then finds that delay $\mu^* \in \mathbb{R}$ that minimizes the squared difference to the closest delay in each set. We then pick for each $U \in I$ that d with minimal distance to μ^* i.e. $\operatorname{argmin}_{d \in D} |\mu^* - d|$. With that, we have an initial alignment from p_1 to p_2.

Algorithm 3: AlignCandidate

Input : p_1, p_2
Output: estimated gain, pattern candidate $p' = p_1 \oplus p_2$

$A_N \leftarrow$ AlignNext$(C_{p_1 \bullet}, C_{\bullet p_2})$
$A_F \leftarrow$ AlignFar$(C_{p_1 \bullet}, C_{\bullet p_2})$
$gain_N, A_N \leftarrow$ OptimizeAlignment$(p_1 \oplus p_2, A_N)$
$gain_F, A_F \leftarrow$ OptimizeAlignment$(p_1 \oplus p_2, A_F)$
return $gain_N, p_1 \oplus p_2$ with A_N **if** $gain_N > gain_F$ **else** $gain_F, p_1 \oplus p_2$ with A_F

Align Candidate In Algorithm 3 we show pseudocode of the alignment procedure. With $C_{p_1 \bullet}$, we refer to, the set of positions of the last event of pattern p_1 under the current cover C, analog $C_{\bullet p_2}$ for the first events of p_2. In AlignNext we map each occurrence of p_1 to the next one of p_2, provided they are in the same sequence S.

We optimize both alignments, as described in the main body of the paper, and return the one for which we estimate a higher gain.

Algorithm 4: FillGaps

Input : pattern p, index i
Output: pattern p'

foreach e between $p'[i]$ and $p'[i+1]$ in cover C **do** in order of frequency
  $gain_1, A_1 \leftarrow$ AlignCandidate(e,$p'[i+1:]$)
  $gain_2, A_2 \leftarrow$ AlignCandidate($p'[i:]$,e,)
  $p' \leftarrow p'[:i] \oplus e \oplus p'[i+1:]$
  if $gain_1 > 0 \land gain_2 > 0 \land L(D, M) > L(D, M \oplus p')$ **:**
    $p_1^* \leftarrow$ FillGaps(p', i)
    $p' \leftarrow$ FillGaps$(p_1^*, i + 1 + |p_1^*| - |p'|)$
    return p'
return p

Fill Gaps Before adding a pattern p to our model M, we refine the gap introduced by combining p_1 with p_2. We show the pseudocode in Algorithm 4. To use FillGaps(p, i) we need a cover that includes p, we use this cover to see which patterns, incl. singletons, are frequently used within gap i of p. We test these for addition, in order of frequency. If we choose to extend pattern p we call FillGaps on the two newly introduced gaps. This is a recursive algorithm that fills all gaps until we no longer get a gain by adding events.

Toy Example — Search Iteration In this paragraph we go through one iteration of the main search loop, Algorithm 2. We consider a toy input sequence shown in Figure 3 (1), over the alphabet $\Omega = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{f}\}$. We begin by generating a set of candidates, we do so by taking the cross product between all existing patterns (incl. singletons), hence $Cand = \{\texttt{aa}, \texttt{ab}, \texttt{ac}, \texttt{ba}, \texttt{bb}, \texttt{bc}, \texttt{ca}, \texttt{cb}, \texttt{cc}, \dots\}$ (line 1 in Alg. 2).

Let $p_1 p_2$ be the candidate from patterns p_1 and p_2. Next, we sort all candidates by $|p_1| usg(p_1) + |p_2| usg(p_2)$, where $usg(p_1)$ is usage of p_1 in the current cover, so the usage multiplied by the pattern length, favoring long and frequent patterns. For this toy example we assume the order of the candidates to be `[ab, ba, ac, bc, ...]`.

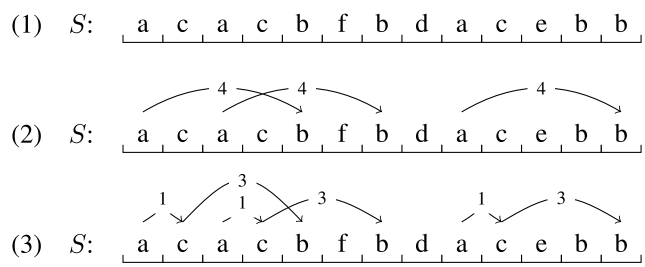


Figure 3: Toy example of main search algorithm, (1) input sequence, (2) alignment between symbols `a` and `b`, and (3) pattern after 'filling in' symbol `c` between `a` and `b`.

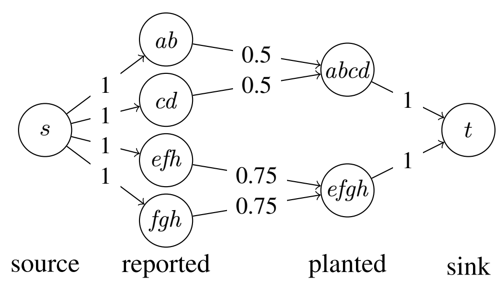


Figure 4: Toy example of reported and planted patterns. We maximize flow from source to sink. We consider the max flow to be numnber of correclty found patterns when computing *recall, precision* and F1.

Next, we test the candidates in order, we begin with `ab`. First, we estimate the gain $\Delta\overline{L}(p_1 \oplus p_2)$ (line 3), i.e. we estimate if including this pattern in our model will decrease the total encoding cost. If we estimate a gain we *align* p_1 with p_2 (line 4). When aligning p_1 with p_2 we find an mapping such that the variance in delay between p_1 and p_2 is minimal. For candidate `ab` we find the alignment shown in Figure 3 (2). As a "bonus" we get a better estimation of the gain, if this one is still positive we test (not estimate) if adding the pattern actually improves the total encoding cost (line 5). Once we have added a pattern we further improve it by filling in events that frequently occur between p_1 and p_2, naturally we only add events if it improves our total encoding cost. In our example we extend our pattern with event `c`, this results in pattern `abc`, Figure 3 (3). Finally, we extend our candidates with the crossproduct between all existing pattern in the model and the newly added pattern, hence $Cand = \{\texttt{acba}, \texttt{acbb}, ...\}$. We keep iterating until we no longer find any patterns that improve our score or our early stopping criteria is met (considering up to $|\Omega|^2/100$, but at least 1 000, unsuccessful candidates in a row).

Experiments

In this section we provide additional details on the evaluation, the experimental setup and additional experimental results.

Evaluation

To compute the number of correctly reported patterns we consider all pairwise combinations of reported and planted patterns and weight each pair by

$$w(p_r, p_p) = \max(1 - lev(p_r, p_p)/|p_p|, 0) \quad ,$$

where p_r is the reported pattern, p_p the planted, and lev is the levenshtein edit distance between two words. We want to avoid that one planted pattern can be "discovered" more than once, hence we limit the total reward to one per planted pattern, analog a reported pattern should at the most count for one as well. In practice, we achieve this by modeling it as a flow network, where we have a flow from reported patterns to planted patterns. In detail, we connect a source to all reported patterns with capacity one, this limits the reward to one per reported pattern, we then connect each reported to each planted and set the capacity to $w(p_r, p_p)$, finally, we connect all planted pattern to the sink, again with a capacity of one, this limits the reward per planted and discovered pattern. We then compute the max flow with the Edmonds-Karp Algorithm (Edmonds and Karp 1972), we consider the max flow as the number of correctly found patterns. In Figure 4 we show a toy example of four reported patterns and two planted ones. With max flow as true positives, we then use the usual definitions of *recall*, *precision*, and F1,

$$recall = \frac{true\ positives}{\#planted\ patterns} \quad , \quad precision = \frac{true\ positives}{\#reported\ patterns} \quad , \quad F1 = \frac{2 * precision * recall}{precision + recall} \quad .$$

Experimental Setup

All experiments were executed single-threaded on an Intel Xeon Gold 6244 @ 3.6 GHz, with 256GB of RAM (shared between multiple simultaneously running processes). We report wall clock runtime.

Hopper We run all experiments, synthetic and real-world data, with the default parameter settings, that is a max delay of 200 and precision $p = 1$. We did not optimize max delay or precision. We set the max delay to 200 as we did not expect longer delays for any of the synthetic and real-world datasets. We set the precision to one as it gives the necessary precision for data sequences where the minimal delay between events is one.

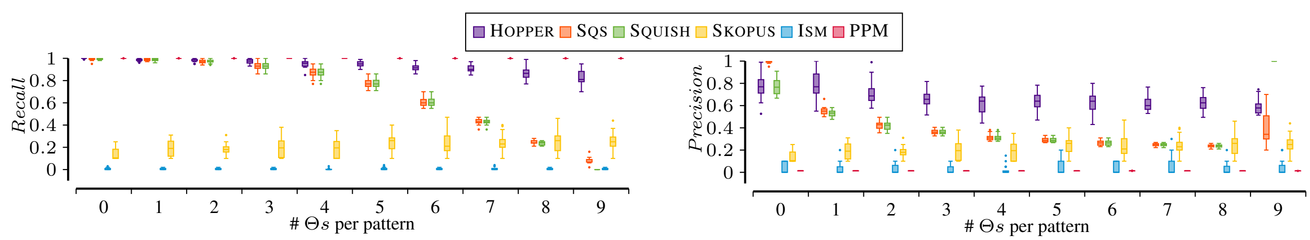
Sqs and Squish are parameter-free.

Ism The number of iterations has been set to 500 and the number of structure steps to 1000, the max runtime was set to 24h for synthetic experiments and 48h for real-world data sets except for the text datasets where we set the 72h. Parameters were chosen to keep the runtimes within workable durations.

Skopus We use the default interesting measure (leverage), and the default parameter for smoothing and support (Lapace). For all experiments we set $k = 10$ i.e. we get the top 10 patterns, and the maximum length to $l = 10$. We set the parameters to match the ground truth of the synthetic experiments, in the sense that we planted 10 patterns with length 10. We kept these values throughout.

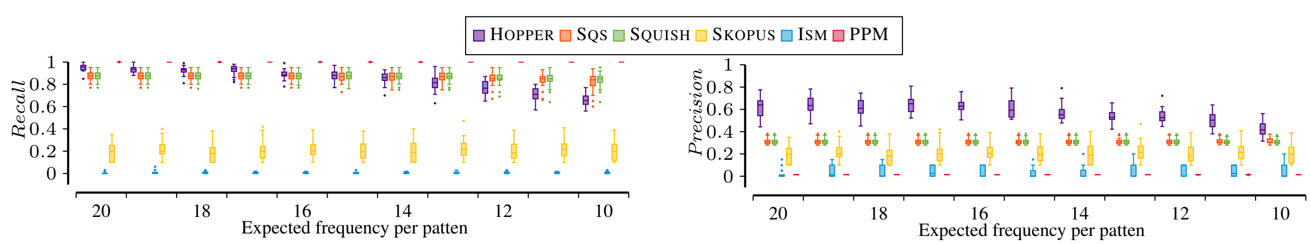
Ppm is parameter free.

Recall and Precision Results



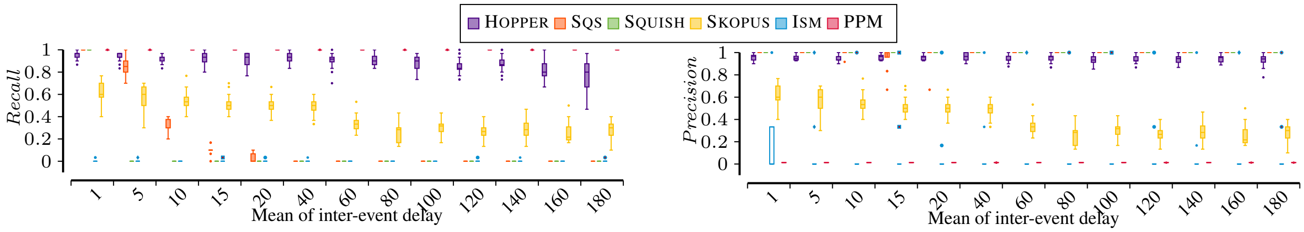
(a) (b)

Figure 5: Recall (a) and Precision (b) results for recovered patterns on synthetic data. We vary the number of delay distributions per pattern, from 0 to 9.



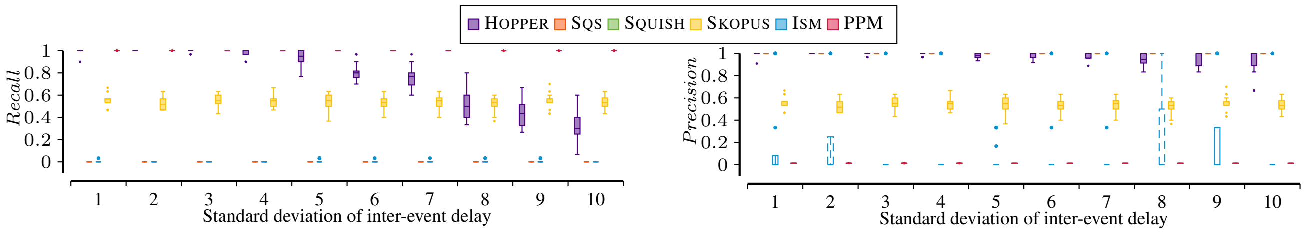
(a) (b)

Figure 6: Recall (a) and Precision (b) results for recovered patterns on synthetic data. We decrease the number of planted instances from 200 to 100. We sample for each planted instance, uniform at random, which of the 10 unique pattern to plant.



(a) (b)

Figure 7: Recall (a) and Precision (b) results for recovered patterns on synthetic data. We vary the mean of the inter-evnet delay.



(a) (b)

Figure 8: Recall (a) and Precision (b) results for recovered patterns on synthetic data. We vary the the variance of the inter-event delay.

All Holidays

Here we show all pattern reported on the *Holidays* dataset by the respective methods.

Hopper

- `[May 1`st `(155 days) National Holiday (83 days) 1`st `Christmas Day (1 day) 2`nd `Christmas Day, (6 days) New Year's (80 to 112 days) Good Friday (3 days) Easter Monday (49 days) Whit Monday]`

where all delay distributions are uniform.

Sqs

- `[1`st `Christmas Day, 2`nd `Christmas Day, New Year's]`
- `[May 1`st`, Ascension Thursday, Whit Monday]`
- `[Good Friday, no holiday, Easter Monday]`

Squish

- `[no holiday, 1`st `Christmas Day, 2`nd `Christmas Day]`

ISM

- `[New Year's, May 1`st`]`

PPM

- `[no holiday]`
- `[no holiday]`
- `[no holiday]`
- `[National Holiday, 1`st `Christmas Day, 2`nd `Christmas Day, New Year's]`
- `[May 1`st`]`
- `[National Holiday, 1`st `Christmas Day, 2`nd `Christmas Day, New Year's]`
- `[May 1`st`]`
- `[Good Friday, Easter Monday, Ascension Thursday, Whit Monday]`
- `[Good Friday, Easter Monday, Whit Monday]`

- [no holiday]
- [no holiday]
- [May 1st]
- [Ascension Thursday, Ascension Thursday]
- [Good Friday, Easter Monday, Ascension Thursday, Whit Monday]

SKOPUS

- [New Year's, Good Friday, Easter Monday, May 1st, Ascension Thursday, Whit Monday, National Holiday, 1st Christmas Day, 2nd Christmas Day]
- [New Year's, Easter Monday, May 1st, Ascension Thursday, Whit Monday, National Holiday, 1st Christmas Day, 2nd Christmas Day]
- [New Year's, Good Friday, May 1st, Ascension Thursday, Whit Monday, National Holiday, 1st Christmas Day, 2nd Christmas Day]
- [New Year's, Good Friday, Easter Monday, Ascension Thursday, Whit Monday, National Holiday, 1st Christmas Day, 2nd Christmas Day]
- [New Year's, Good Friday, Easter Monday, May 1st, Ascension Thursday, Whit Monday, National Holiday, 2nd Christmas Day]
- [New Year's, Good Friday, Easter Monday, May 1st, Ascension Thursday, Whit Monday, 1st Christmas Day, 2nd Christmas Day]
- [New Year's, Good Friday, Easter Monday, May 1st, Whit Monday, National Holiday, 1st Christmas Day, 2nd Christmas Day]
- [New Year's, Good Friday, Easter Monday, May 1st, Ascension Thursday, National Holiday, 1st Christmas Day, 2nd Christmas Day]
- [New Year's, Good Friday, Easter Monday, May 1st, Ascension Thursday, Whit Monday, National Holiday, 1st Christmas Day]
- [Good Friday, Easter Monday, May 1st, Ascension Thursday, Whit Monday, National Holiday, 1st Christmas Day, 2nd Christmas Day]

Extended Real World Statistics

Dataset	$\lvert D\rvert$	$\lvert\Omega\rvert$	$\lvert\lvert D\rvert\rvert$	HOPPER					SQS				SQUISH			ISM			PPM				SKOPUS		
				$\lvert P\rvert$	$\mathbb{E}(\lvert p\rvert)$	$\mathbb{E}(w)$	$\#\Theta$	sec	$\lvert P\rvert$	$\mathbb{E}(\lvert p\rvert)$	$\mathbb{E}(w)$	sec	$\lvert P\rvert$	$\mathbb{E}(\lvert p\rvert)$	sec	$\lvert P\rvert$	$\mathbb{E}(\lvert p\rvert)$	sec	$\lvert P\rvert$	$\mathbb{E}(\lvert p\rvert)$	$\mathbb{E}(w)$	sec	$\lvert P\rvert$	$\mathbb{E}(\lvert p\rvert)$	sec
Holidays	1	11	36525	1	8.0	393.0	7	7	3	3.0	19.2	0	1	3.0	8	1	2.0	5615	14	2.1	51.6	12504	10	8.1	7167
Radio	1	494	15597	22	3.4	48.2	43	950	15	3.0	5.8	7	5	3.2	10	1	2.0	173080	587	1.4	71.9	251	-	-	7d
Lifelog	1	77	40520	37	5.2	129.5	68	3457	58	2.8	3.9	24	36	2.3	20	3	2.0	173021	1609	1.2	119.1	35904	-	-	7d
Samba	1	118	28879	40	4.9	110.4	101	5690	221	2.6	2.7	107	115	2.7	114	1	2.0	172974	1430	1.1	17.1	445	-	-	7d
Chorales	100	493	4693	56	2.5	4.7	57	148	114	2.4	2.6	2	96	2.4	14	115	3.0	172801	433	1.2	2.6	18	-	-	7d
Rolling Mill	1000	555	53788	237	4.4	7.4	489	4223	470	4.6	5.0	1080	497	5.0	3942	141	3.0	3290	3663	2.2	181.9	5211	10	9.2	35187
Skating	530	82	25502	86	3.2	9.1	160	479	160	3.4	4.0	65	-	-	-	33	3.3	1491	1466	1.5	55.5	514	10	3.2	18544
Romeo	1	4789	37462	254	2.6	12.9	284	17646	254	2.4	2.8	771	155	2.4	780	0	0.0	101323	2397	1.4	332.7	512	-	-	7d
Room	1	9009	86909	565	2.4	3.1	610	80009	701	2.4	2.5	51441	299	2.4	9253	0	0.0	254654		-	-	-	-	-	7d
Gatsby	1	7463	63974	439	2.5	7.3	488	64714	519	2.5	2.6	9295	303	2.4	7134	0	0.0	248587	4766	1.3	641.9	2820	-	-	7d

Table 2: Statistics of real world datasets, we report number of discovered patterns $|P|$, the average pattern length $\mathbb{E}(|p|)$, for methods that provide information about delays, the average expected distance between the first and last symbol of a pattern $\mathbb{E}(w)$, the number distributions, over all patterns $\#\Theta$, and the runtime in seconds sec. For the missing results: SKOPUS did not terminate within 7 days, and PPM and squish failed due to a bug.