DISCOVERING ACTIONABLE INSIGHTS FROM EVENT SEQUENCES

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University

by

JOSCHA CÜPPERS

Saarbrücken, 2025

DAY OF COLLOQUIUM:
11 September 2025

DEAN OF FACULTY:
Prof. Dr. Roland Speicher

EXAMINATION BOARD:
Chair – Prof. Dr. Raimund Seidel
Advisor, Reviewer – Prof. Dr. Jilles Vreeken
Reviewer – Prof. Dr. Alexandre Termier
Reviewer – Dr. Matthijs van Leeuwen
Academic Assistant – Dr. Lénaïg Cornanguer

ABSTRACT

This thesis explores how to extract actionable insights from event sequences. Event sequences are fundamental across a wide range of domains, from diagnosing chains of failures to analyzing workflow traces in production systems. Instead of assuming that all events stem from a single underlying process, we allow for the possibility of multiple, potentially concurrent mechanisms — resulting in interleaved and complex structures. We aim to identify and represent these structures using sequential patterns that capture the temporal dependencies between events.

We develop methods that yield succinct and easy-to-understand summaries of event sequences. We start by proposing a method to discover predictive patterns that not only predict that a target event is imminent but also when it will occur. For example, which sequence of events predicts an upcoming failure. Next, we explore how to discover patterns characterized by consistent time delays between events; unlike existing methods that penalize gaps uniformly, our approach focuses on identifying and modelling these delays. Beyond co-occurrence, we explore conditional structures in the form of rules. Furthermore, we study summarization of event sequences in terms of patterns that include generalized events — events that can match multiple observed events. To demonstrate the practical relevance of this line of work, we tackle a domain-specific challenge, modeling network flows and generating synthetic data from the learned model. Lastly, we investigate causal relationships between events by introducing a novel causal discovery method that infers a complete causal graph over all event types.

We empirically evaluate all methods and show that they uncover meaningful insights from real-world data. We conclude this thesis by reflecting on the limitations of our approaches and the broader challenges in evaluating pattern discovery methods.

ZUSAMMENFASSUNG

Diese Dissertation erforscht, wie man aussagekräftige Erkenntnisse aus Event Sequenzen extrahieren kann. Event Sequenzen sind in vielen Bereichen von grundlegender Bedeutung, vom Aufdecken von Fehlerketten bis hin zur Analyse von Arbeitsabläufen. Anstatt davon auszugehen, dass alle Ereignisse von einem einzigen Prozess erzeugt werden, lassen wir die Möglichkeit mehrerer, potenziell parallelen, Prozessen zu. Dies kann zu verschachtelten und komplizierten Strukturen führen. Unser Ziel ist es diese Strukturen zu identifizieren und mithilfe von sequenziellen Mustern abzubilden. In dieser Dissertation entwickeln wir Methoden, welche Event Sequenzen leicht verständlich und kurz zusammenfassen. Zunächst schlagen wir eine Methode zur Entdeckung prädiktiver Muster vor, die nicht nur vorhersagen, dass ein Zielereignis bevorsteht, sondern auch, wann es eintreten wird. Zum Beispiel, welche Abfolge von Ereignissen einen bevorstehenden Ausfall vorhersagt. Als Nächstes untersuchen wir wie sich Muster mit gleichmäßigen Zeitverzögerungen zwischen Ereignissen entdecken lassen. Im Gegensatz zu bestehenden Methoden, die Lücken gleichmäßig bestrafen, konzentriert sich unser Ansatz auf die Identifizierung und Modellierung dieser Verzögerungen. Wir untersuchen nicht nur Muster in Event Sequenzen, sondern auch Regeln, um bedingte Strukturen zu finden. Darüber hinaus untersuchen wir wie Event Sequenzen anhand von abstrahiertem Muster zusammengefasst werden können. Abstrahierte Muster beschreiben ein allgemeines Verhalten, das mit unterschiedlichen Ereignissen auftritt. Um die praktische Relevanz dieser Arbeit zu demonstrieren, befassen wir uns mit einem domänenspezifischen Problem: der Modellierung von aufgezeichneten Netzwerkverbindung und der Generierung synthetischer Daten aus dem erlernten Modell. Schließlich untersuchen wir die kausalen Beziehungen zwischen Events, wir schlagen eine Methode vor, welche einen kausalen Graphen über alle Eventtypen lernt.

Wir evaluieren alle Methoden empirisch und zeigen, dass sie aussagekräftige Erkenntnisse in realen Daten finden. Zum Abschluss dieser Arbeit, diskutieren wir die Grenzen unserer Ansätze.

ACKNOWLEDGMENTS

My first thanks goes to my supervisor, Jilles Vreeken. Thank you for both giving me this opportunity and, believing in me throughout this journey. You taught me not just how to do the research, but also the importance of effectively communicating one's findings. Seriously, thank you; I couldn't have pulled this off without you.

I extend my sincere thanks to my co-authors and collaborators: Janis Kalofolias, Paul Krieger, Adrien Schoen, Gregory Blanc, Pierre-François Gimenez, Sascha Xu, Ahmed Musa, Aleena Siji, Osman Mian, Lénaïg Cornanguer, Ghada Nait Said, and Nils Walter. Thank you for your invaluable contributions and the excellent collaboration we shared. It was a true joy working with you all. I am also grateful to the entire EDA team. Thank you for creating a welcoming working environment and for being wonderful colleagues.

A big thank you to Pierre-François Gimenez for hosting me at Inria, Rennes, and for the warm welcome from the research group there. I am proud to include our joint work in this thesis.

My sincere gratitude goes to the external reviewers, Alexandre Termier and Matthijs van Leeuwen, for the time and effort spent reviewing my thesis and for the fruitful discussions and meaningful questions. Thank you to the remaining committee members, Lénaïg Cornanguer and Raimund Seidel, for agreeing to join my committee.

I would also like to thank my friends and family, this thesis would not have been possible without the joyful times away from my academic work. I would especially like to thank Helen Bohleber for being an unwavering friend since the first day of my academic journey, and Miriam Rateike, who became a wonderful friend during my PhD, and with whom I shared many engaging and insightful conversations.

To Tejumade Afonja, meeting you was a gift. Thank you for expanding my horizons, for pushing me in the right way, and for bringing your positive and optimistic outlook into my life. I am grateful for all the little things that I can't possibly list.

My final thank you goes to my parents for their support throughout my studies and for helping me navigate the more challenging periods.

CONTENTS

1 INTRODUCTION

Understanding complex event sequences is critical in many domains, from analyzing user behaviors in online services to monitoring complex networked systems. To illustrate the key focus of this dissertation, let us explore a few such scenarios in which the right analysis tool can provide key insights.

We begin with a familiar example, an online store. Suppose we run an online store that users interact with via a website or an app. To better understand how people use our services, we collect data on how they navigate the site — what products they look at, whether they view details like technical specs, and events such as adding or removing items from the cart, completing a purchase, or abandoning it. Visitors arrive through links on different sites, not always landing on the homepage. Once there, they pursue different goals, some browse available options, others head straight to ordering what they already know they want. Because of this, we can not assume there is a single, general process that explains how users behave.

To really understand how users engage with the site, we need to identify the key behavior patterns. Ideally, we want a compact summary that captures the essential usage structures, we do not want a list of the most frequent patterns with minor variations. A good summary highlights the most informative and actionable trends, without overwhelming decision-makers with unnecessary detail that could bury important insights under a flood of redundant information.

Once we have a interpretable summarization of how the website is actually used, we can use that knowledge to reduce friction and (hopefully) improve retention. It also helps us avoid unintentionally disrupting core usage patterns when we make changes — or, if we do choose to break them, guide users through the new experience in a deliberate way.

User behavior is only one motivating example, structured event sequences also arise in technical and industrial contexts — such as large-scale communication networks or automated production plants — op-

erations are recorded as a sequence of events. Among these, some events indicate normal progress, while others signal potential issues, anomalies, or outright failures. While summarization can be helpful, ideally we would like anticipate such events, or preempt them.

For example, let us consider a setting where we are monitoring a complex networked system, e.g. in a production facility, and are interested in figuring out under what conditions we see specific kinds of failures. Due to interleaved processes and interactions, one failure can trigger another, possibly delayed. As systems grow more complex, so do the dependencies between different failure types. A minor fault in one part, for instance, might cascade into a chain of related failures or alarms. Understanding these chains of events is critical for identifying root causes, mitigating risks, and preventing system-wide disruptions. As an operator of such a network or production plant, one would like to identify delays, faults, and other events that impact operations before they occur. Additionally, we would like to know what caused an undesired event, or at the very least learn patterns that predict such events. Ideally, we would like to intervene in the system to avoid problematic events in future operations. For this, we need to understand the causal connections between events.

Finally, let us consider a specific domain next — network traffic. Network traffic, recorded as individual connection flows over time, often exhibits sequential structures, for example a DNS request, followed by a HTTP request. These sequences hold information about user behavior, and system performance. Understanding and modelling these can help in detecting anomalies, e.g. potential security vulnerabilities. Network traffic is usually recorded at a central point, capturing the flows of many clients in parallel. This results in highly interleaved structures, making it difficult to understand the dependencies between flows. Even a single client, for example a server running multiple services, generates interleaved structures.

What each of these problems share — whether analyzing network flows, cascading alarms and failures, or user behavior — is the fundamental need to understand sequential event data. In each case, the order in which events occur and their temporal relationships provide critical insights into the underlying processes. How to obtain understandable models to gain actionable insight in an easily interpretable manner into event sequences is the topic of this dissertation.

Sequential patterns are an apt construct to capture dependencies and represent them in a human-understandable way. In general, the goal of pattern mining is to discover interesting substructures, the patterns, in a data set. A sequential pattern, in its most basic form, is a list of events. A key limitation of pattern mining, in general, is that the pattern language has to match the mechanism of the data we wish to capture. We generally do not know the generating mechanism we wish to capture. As such, discovering patterns over a more expressive pattern language has the potential to reveal new insights. With this, we can formulate the first research goal of this thesis,

Research Goal 1 (Summarizing Sequential Event Sequences) *Given an event sequence database, discover models that summarize the data so that it provides meaningful, interpretable insight.*

While understanding sequential event data is crucial, it only provides a foundation for analysis and often falls short when it comes to enabling actionable responses. Knowing how events relate temporally is insufficient to make predictions or intervene in a system. Ideally, we seek insights that not only explain the underlying structures but also allow us to act; this is the second research goal we work towards in this thesis,

Research Goal 2 (Discovering Actionable Summaries) *Given an event sequence database, find a summarization that enables action from the gained insight.*

Next, we discuss the individual contributions of this thesis and how they relate to the posed goals.

CONTRIBUTIONS

In this section, we will first outline the contributions made in the different chapters and then discuss how they contribute to Research Goal 1 and Research Goal 2.

CONTRIBUTION 1 Suppose we are given a sequence of discrete events, S_x, for example from a manufacturing plant, and an equally long binary sequence, S_y, that indicates time points of interest, e.g., failures or delays. In Chapter 2, we consider the problem of discovering a set

of patterns from data sequence S_x that reliably predict the indicated time-points in S_y. As a model, we consider a set of tuples, each consisting of a pattern and a delay distribution. The occurrence of a pattern indicates that an event of interest is coming up, and the delay distribution indicates when we can expect the event to occur. To avoid making assumptions about how the delays are distributed, we use a non-parametric delay distribution. To allow for noise, we also model the probability that an interesting event does not occur after we observe a pattern. To avoid overly many patterns and complicated delay distributions, we use the Minimum Description Length (MDL) principle as model selection criteria. As testing all patterns, let alone all possible delay distributions is infeasible, we propose the OMEN algorithm based on an optimistic estimator to overcome local minima in the pattern search.

CONTRIBUTION 2 In Chapter 3, we focus on modeling and discovering patterns with consistent and large gaps. Existing approaches either penalize or limit gaps and thereby introduce a strong bias against patterns with long-range dependencies. We address this shortcoming by explicitly modeling the delays between events and thereby rewarding consistent gaps, no matter the length. We again base our approach on the MDL principle and introduce the HOPPER algorithms to efficiently discover patterns with long-range dependencies.

CONTRIBUTION 3 Sequential patterns can only express co-occurrence between symbols, for example that X and Y frequently happen after one another, but do not capture conditional dependencies, such as if X happens there is a increased probability we will soon see Y too. Discovering conditional relations reveals under which conditions events or sequences of events are likely to occur. In Chapter 4, we study how to capture conditional relationships in terms of rules of the form $X \rightarrow Y$. To discover good rule sets, we propose the SEQRET algorithm.

CONTRIBUTION 4 All existing methods for sequential pattern mining can only discover patterns defined as subsequences of the observed data, such as "dog barks" and "cat meows", but cannot discover generalized patterns of the kind "[pet] [makes sound]". This is the problem we address in Chapter 5. Loosely speaking, a generalized patterns is a pat-

terns that may include surface level events (e.g. a) as well generalized events (e.g. α) that can match a set of observed events ($\alpha = \{b, c\}$). As a model, we consider a set of generalized events and a set of patterns. We propose the FLOCK algorithm that jointly discovers these generalized events and succinctly summarizes the data in terms of generalized patterns over these.

CONTRIBUTION 5 In Chapter 6, we change tack and rather than solving a general problem in sequential pattern mining, we focus on an application. In particular, we show how to use pattern mining to summarize network flows and generate new synthetic network flows from such a summarization. To this end, we propose a pattern language specific to network flow traces and formalize the problem in terms of MDL. To find a good summarization of the network flows, we introduce the *FlowChronicle* algorithm. To generate realistic synthetic data we sample from the learned model. Since our model is fully interpretable, in contrast to the more common deep learning based approaches, undesired patterns could even be removed from the model to avoid including them in the synthetic generated data. We extensively evaluate and compare the generated data to state-of-the-art synthetic data generators and show that we not only preserve the inter flow correlation of the data, but also the structure between flows.

CONTRIBUTION 6 All previous contributions focus on discovering correlations, and while these provide insight (Chapters 3, 4, 5) and are actionable (Chapters 2 and 4), they do not give guarantees that they model actual causal processes. For our final contribution, we study the causal relation between event types in event sequences. Our goal is to learn a directed causal graph over all event types. Understanding the causal relations is critical to intervene in a system, for example, to prevent failures from occurring in the future. While predictive and co-occurring events might be causally connected, they could also just be correlated due to a shared confounder. In Chapter 7, we study under which conditions we can learn the causal relationships between events. We propose CASCADE to discover a causal graph from a given timestamped event sequence.

These contributions address Research Goal 1 and Research Goal 2 in different ways. We make progress towards Research Goal 1 primarily

Publication	Chapter
Joscha Cüppers and Jilles Vreeken. "Just Wait for it... Mining Sequential Patterns with Reliable Prediction Delays." In: *2020 IEEE International Conference on Data Mining*, IEEE, 2020, pp. 82–91. Joscha Cüppers, Janis Kalofolias, and Jilles Vreeken. "Omen: discovering sequential patterns with reliable prediction delays." In: *Knowledge and Information Systems 64*, Springer, 2022, pp. 1013–1045.	Chapter 2
Joscha Cüppers, Paul Krieger, and Jilles Vreeken. "Discovering Sequential Patterns with Predictable Inter-event Delays." In: *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI ,2024, pp. 8346–8353.	Chapter 3
Aleena Siji, Joscha Cüppers, Osman Ali Mian, and Jilles Vreeken. "Seqret: Mining Rule Sets from Event Sequences." In: *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI ,2026.	Chapter 4
Joscha Cüppers and Jilles Vreeken. "Below the Surface: Summarizing Event Sequences with Generalized Sequential Patterns." In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ACM, 2023, pp. 348–357.	Chapter 5
Joscha Cüppers*, Adrien Schoen*, Gregory Blanc and Pierre-Francois Gimenez. "FlowChronicle: Synthetic Network Flow Generation through Pattern Set Mining." In: *Proceedings of the ACM on Networking 2 CoNEXT4*, ACM, 2024, Article No: 26.	Chapter 6
Joscha Cüppers*, Sascha Xu*, Ahmed Musa and Vreeken, Jilles. "Causal Discovery from Event Sequences by Local Cause-Effect Attribution." In: *Advances in Neural Information Processing Systems*, Curran Associates, 2024, pp. 24216–24241.	Chapter 7

Table 1.1: [**Publications**] List of publications with reference to chapters. Equal Contribution marked by *.

in Chapters 3, 4, 5, and 6. Research Goal 2 is addressed by Chapters 2, 4, and 7.

Every chapter of this thesis is based on one or more manuscripts, all published at highly selective peer-reviewed venues. We list all publi-

cations and their corresponding chapters in Table 1.1. I am the first author of all publications associated with Chapters 2, 3, and 5. I contributed to all aspects of these papers, that is: main idea, theory, implementation, experimentation, and write up. Chapters 6 and 7 are based on publications with shared first authorship. For Chapter 6, both first authors, Adrien Schoen and I, contributed to the main ideas and the writing of the manuscript. In addition, I contributed the pattern mining part, and Adrien Schoen contributed the evaluation and comparison to the state of the art. For Chapter 7, both first authors, Sascha Xu and I, contributed to the main idea, theory, and writing of the manuscript. In addition, I contributed the implementation and evaluation, and Sascha Xu contributed the main idea of the algorithm. Finally, Chapter 4 is based on a paper where I am the second author. I contributed to the main ideas, theory, algorithm design, experiment design, and the writing of the manuscript, and in addition provided supervision and guidance to the first authors.

In addition to the papers included in this dissertation, I contributed to shared work with Sascha Xu and Jilles Vreeken [193] in which we explored how to succinctly capture interactions between input variables to explain decisions made by black-box models. We omit this work from this thesis to permit a clear focus on event sequences.

2

MINING SEQUENTIAL PATTERNS WITH RELIABLE PREDICTION DELAYS

In this chapter, we present our first contribution. We propose a method that, given one binary target sequence and one data sequence over an alphabet, discovers patterns in the data sequence that predict when upcoming events in the target sequence will occur. We formally define this problem in terms of the Minimum Description Length principle, by which we identify the best patterns as those that describe the occurrences of target time points most succinctly. Through extensive empirical evaluation we show that our method works well in practice.

2.1 INTRODUCTION

Suppose we are given a discrete valued time series S_x of observed events, and an equally long binary sequence S_y that indicates at which points in time something of interest happened that we would like to predict—earthquakes, for example. We consider the problem of mining a small set of interpretable and actionable patterns from S_x that reliably predict those interesting events. With *reliable* and *actionable* we mean those patterns that not only highly accurately predict *that* an interesting event will follow, but which additionally can tell with high precision *how long* it will be until that event will happen. That is, we are after patterns that have a compact distribution of delays between pattern occurrences and predicted events. As real processes are rarely trivial, it is unlikely that a single patterns will suffice to explain all interesting events, and hence we consider the problem of discovering a small and non-redundant set of patterns that *together* reliably predict the interesting events.

This chapter is based on [34]: Joscha Cüppers and Jilles Vreeken. "Just Wait for it... Mining Sequential Patterns with Reliable Prediction Delays." In: *2020 IEEE International Conference on Data Mining*. 2020, pp. 82–91. and [31]: Joscha Cüppers, Janis Kalofolias, and Jilles Vreeken. "Omen: discovering sequential patterns with reliable prediction delays." In: *Knowledge and Information Systems, 64*. 2022, pp. 1013–1045.

Event prediction is well-studied in time series analysis. Most work considers continuous-valued data, and focuses on tasks such as detecting abrupt distributional changes [80] and identifying events that precede such changes [156]. As we aim to discover patterns that explain the interesting time points, our work is closer to that of sequence classification [205] and similar to the task of learning patterns to reconstruct a labels of a sequence [197]. Existing solutions, however, focus purely on discovering all patterns that sufficiently accurately predict that an interesting event will follow *some* time after their occurrence [206], rather than our goal of discovering a small set of patterns for which we can reliably say *how long* it will take before that event occurs. As such, our work is related to information flow [159] and Granger causality [69], in the sense that patterns are only interesting if their occurrences provide significantly more information about S_y than the history of S_y does by itself.

We formalize the problem of finding a set of patterns predictive for S_y, in terms of the Minimum Description Length (MDL) principle [152], by which we identify the best patterns in S_x as those that describe S_y most succinctly. We model the data such that for every occurrence of a pattern in S_x we encode the delay until the predicted associated interesting event: the more peaked this distribution, the cheaper it will be to encode the delays, and hence, we particularly favor patterns that accurately predict both the occurrence of and time until an interesting event. Discovering the optimal explanation of S_y given a set of patterns, i.e. the alignment between the pattern occurrences and the interesting events, as well as discovering the optimal set of patterns, are both hard problems that do not permit straightforward optimization. We therefore split the problem in two, and propose effective algorithms for each. To find a good explanation of S_y given a single pattern we propose both a general purpose solution, as well as one that is particularly suited for long range predictions. To discover good pattern sets, we present Omen, a greedy heuristic that iteratively optimizes the alignment of pattern occurrences to interesting events, and uses this alignment to discover the best refinements of the patterns. We show a visualization of the Omen algorithm in Figure 2.1, and will explain the details in Section 2.4. We additionally introduce fOmen, a faster alternative that is robust against high time delays. Neither imposes restrictions on the delay distribution, both allow for overlap

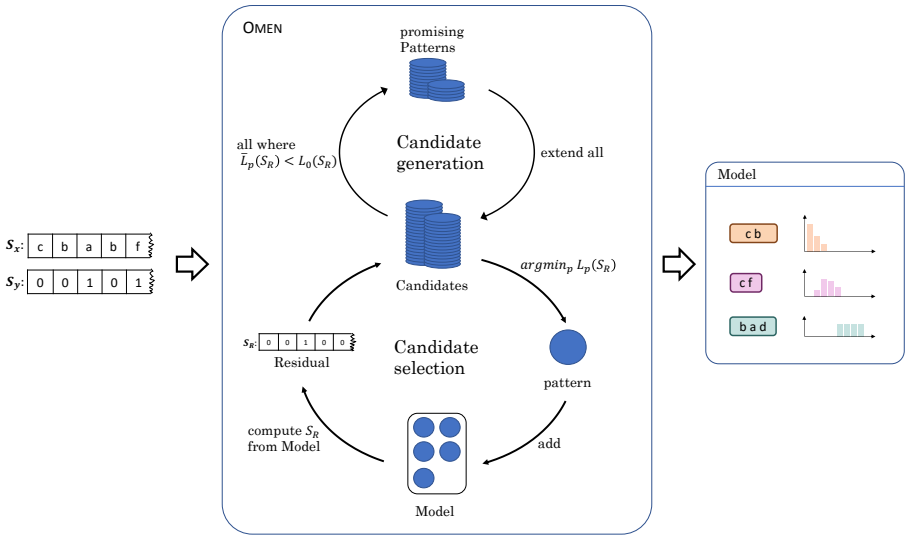


Figure 2.1: Visualization of the OMEN algorithm: As input we are given a event sequence S_x and a target sequence S_y and as output we obtain a set of patterns that together best explain S_y given S_x. The pattern search works by alternating between candidate generation and evaluation. We use the MDL principle to keep the pattern set small and non-redundant.

between predictions, and only have one hyper parameter that allows the user to specify to what extend gaps in patterns are allowed.

We empirically evaluate OMEN on both synthetic and real-world data. We show that our score reliably determines the predictiveness of patterns, and compares favorably to state-of-the-art information flow scores [17, 159]. We show that OMEN highly accurately reconstructs the ground truth, both in terms of discovering predictive patterns, as well as their delay distributions, outperforming four supervised and unsupervised sequential pattern miners [57, 170, 183, 205]. On real-world data we confirm that OMEN discovers meaningful and actionable patterns that give insight in the data generating process.

2.2 PRELIMINARIES

We start by introducing the notation and preliminaries we will use throughout this work.

2.2.1 *Notation*

Given an alphabet Ω of events $e \in \Omega$ we study finite sequences $S_x \in \Omega^n$ of these events, where $|S_x| = n$ is the length of the sequence. We denote $S_x[i]$ to refer to the i^{th} event in S_x, and $S_x[i:j]$ to denote the subsequence $S_x[i], ..., S_x[j]$. We write $||S_x||_a$ for the number of times we see event $a \in \Omega$ in S_x.

Our data consist of two sequences $S_x \in \Omega^n$, and $S_y \in \{0,1\}^n$, both of length n. The former encodes an sequence of observed events and the latter indicates the occurrence of something 'interesting' at every point i for which $S_y[i] = 1$.

We consider sequential patterns $p \in \Omega^m$ of length $m = |p| < n$. We say that a pattern p *occurs in* a window $w = S_x[i:j]$ when all events of p occur in w in the order specified by p. We call such a window minimal iff there does not exist any subwindow $w' \prec w$ in which p occurs. By considering minimal windows we avoid double counting of occurrences [169]. We say a pattern p *matches* sequence S_x at the jth event, $S_x[j]$, iff there exits a minimal window $S_x[j-a:j]$ of maximum window length $a \leqslant m \times g_f + g_a$—where g_f and g_a are user defined parameters that allow to control the number of gaps we permit, respectively in terms relative to the pattern length, and absolute in number of gap events.

Given a pattern p and an event sequence S_x, we can trivially construct a binary sequence $S_z^p \in \{0,1\}^n$ in which $S_Z[j] = 1$ if pattern p matches $S_x[j]$, and 0 otherwise. A *predictive* pattern p is a sequential pattern p with an associated discrete delay distribution that specifies the probability density $\phi_p(\delta)$ that something interesting will happen δ time steps after an occurrence of the pattern.

All logarithms are base 2, and we use the convention that $0 \log 0 = 0$.

2.2.2 *Minimum Description Length*

The Minimum Description Length (MDL) principle [71] is a computable and statistically well-founded model selection criterion based on Kolmogorov Complexity [103]. For a given model class $\mathcal{M}$, it identifies the best model $M \in \mathcal{M}$ as the one that minimizes the number of bits needed to describe both model and data, $L(M) + L(D \mid M)$ where

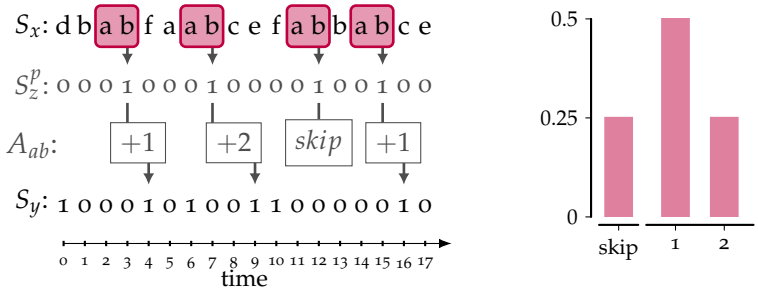


Figure 2.2: [**Toy Example**] On the left, we show an encoding example where pattern ab covers three out of five interesting events. Occurrence vector S_z^p encodes the four matches of ab in S_x. Alignment A_{ab} maps these occurrences to interesting events in S_y. On the right, we show the resulting time delay distribution ϕ_{ab}.

$L(M)$ is the length of model M in bits and $L(D \mid M)$ the length of data D given M.

This is known as two-part, or crude MDL—in contrast to one-part, or refined MDL [71], which although preferred from a theoretical perspective, is not computable for arbitrary models. We use two-part MDL because we are particularly interested in the model: those patterns that, given S_x allow us to describe S_y most succinctly. Note that our goal here is to *select* the best model for the data at hand, and not to actually compress the data; we are hence not concerned with materialized codes, and only care about ideal code lengths.

To use MDL we have to define a model class $\mathcal{M}$, and encodings for data and model. We do so in the next section.

2.3 THEORY

In this section, we introduce the problem at hand. We start with a informal definition of the problem, after which we define a model class, show how to encode a model and data given a model, and formally state the problem at hand.

2.3.1 *The Problem, Informally*

We are interested in discovering that set of predictive sequential patterns p that most reliably predict the interesting events in S_y given

observed events S_x. Our models consist of tuples (p, ϕ_p) of sequential patterns p and their associated delay distributions ϕ_p, i.e. $M = \{(p_1, \phi_{p_1}), (p_2, \phi_{p_2}), \ldots, (p_k, \phi_{p_k})\}$.

Given S_x and a pattern p, we have a binary sequence S_z^p that marks those time points at which p matches S_x. Every occurrence of a pattern predicts (in principle) that an interesting event is about to happen in S_y. We call the mapping of occurrences of a pattern p to interesting events in S_y its alignment A_p. We allow for additive and destructive noise in S_x and S_y, which is to say, we do not require that every occurrence of a true pattern p to be followed by an interesting event in S_y, and neither require that all interesting events in S_y are predictable by true patterns. To permit the former, we allow predictions to be 'skipped'. Formally, an alignment is hence a function a_s that maps each occurrence of pattern p to either an interesting event in S_y or to a 'skip' token.

A delay distribution ϕ_p provides the probabilities of an interesting event occurring δ time steps after an occurrence of pattern p. The higher the probability of δ, the fewer bits we will need to encode that particular value. Overall, the fewer predictions we have to 'skip' and the more peaked the delay distribution is, the more cost effective we can describe A_p in bits per interesting event, and hence the more succinctly we will be able to describe S_y. a

Example 1. *To illustrate, we consider a running example. We show in Fig. 2.2 an event sequence S_x of length 18, over an alphabet $\Omega = a, \ldots, f$. There are four occurrences of pattern ab, which together define occurrence sequence S_z^p. The best possible alignment of these occurrences to interesting events in S_y is given as A_{ab}, which maps the first, second, and fourth occurrence to actual interesting events, resp. with delays of $+1$, $+2$ and $+1$ time steps, and as there is no interesting event corresponding to the third occurrences, it maps that one to a 'skip'. We show the corresponding delay distribution on the right of the figure, in which we see that in 50% of the cases an interesting event happens one time step after the occurrence of the pattern, in 25% two time steps later, as well as that it has a 25% probability of falsely predicting the occurrence of an interesting event ('skip').*

Whereas in the toy example our model exist of only one pattern, $M = \{(ab, \phi_{ab})\}$, in general we allow S_y to be complex, in the sense that multiple patterns may have generated S_y, and are hence needed to reliably predict all interesting events. In other words, we allow a single pattern $p \in M$ to predict only some of the interesting events in

S_y. We denote by $\widehat{S_y^p}$ the binary sequence of interesting events in S_y that are predicted by pattern p. Loosely speaking $\widehat{S_y^p} = A_p(S_z^p)$.

Ideally, together the patterns in M predict each and every interesting event in S_y, i.e. the combination of all predictions $\widehat{S_y} = \bigvee_{p \in M} \widehat{S_y^p}$ equals S_y. Some interesting events, may however not have patterns that can explain their occurrence. To be able to fairly compare between models we need to ensure losslessness, and will therefore additionally need to transmit a residual sequence S_R that encodes all interesting events that are not predicted by any pattern. Formally, we define S_R as the bit-wise XOR between the predicted $\widehat{S_y}$ and the true S_y, $S_R = \widehat{S_y} \oplus S_y$.

2.3.2 *MDL for Predictive Sequential Patterns*

Based on the above intuition we now proceed to define our score. We will first discuss how to encode sequence S_y given a model M and observed events S_x, and then detail how to encode such a model M.

Encoding the Data given a Model

Given event sequence S_x and a pattern $p \in M$, it is straightforward to construct the corresponding pattern occurrence sequences S_z^p. To determine $\widehat{S_y}^p$ from S_z^p, we need alignment A_p which gives us the delays and skips between pattern occurrences and predicted events.

To encode an alignment A_p, we have to transmit each delay δ corresponding to every pattern occurrence of p in S_x. We do so using optimal prefix codes over the time delay distribution ϕ_p [28], which is to say, the lower the probability $\phi_p(\delta)$ of a delay δ, the more bits we need. The length of an alignment A_p for pattern p is defined as

$$L(A_p \mid \phi_p) = -\sum_{i=0}^{|A_p|} \log \phi_p(A_p[i]) \ .$$

Once we know alignment A_p we can reconstruct $\widehat{S_y}^p$, and if we do so for all patterns $p \in M$, we therewith have $\widehat{S_y}$.

To be able to reconstruct S_y from $\widehat{S_y}$ without loss, we additionally need to encode the residual sequence S_R. We have

$$L(S_R) = L_{\mathbb{N}}(||S_R||_1) + \log \binom{|S_x| \ - ||\hat{S}_y||_1}{||S_R||_1} \ ,$$

where we first encode the number of 1s in S_R using $L_{\mathbb{N}}$, the MDL-optimal encoding for integers [153]. It is defined as

$$L_{\mathbb{N}}(z) = \log^* z + \log c_0$$

where $\log^* z$ is defined as $\log z + \log\log z + \ldots$, only including the positive terms in the sum. To obtain a valid encoding, i.e. to satisfy the Kraft inequality, we set $c_0 = 2.865064$ [153].

As now we know the number of 1s in S_R, we can optimally encode the actual sequence S_R via an index over a canonically ordered set of all sequences of length n with $||S_R||_1$ ones. Since we already know the location of predicted interesting events, we know these will be 0 in the residual and we can hence ignore $||\hat{S}_y||_1$ possible locations. As the binomial coefficient greatly increases with every additional interesting event in S_R we favor residuals that cover fewer interesting events.

Example 1. ***(continued)*** *Consider again Figure 2.2. Although data S_x is 18 events long, pattern ab predicts three interesting events, by which there are 15 out of 18 possible time steps at which the 2 unexplained events can occur. As such, we have $L(S_R) = L_{\mathbb{N}}(2) + \log \binom{15}{2}$.*

Putting these two parts together, we have

$$L(S_y \mid M, S_x) = \left(\sum_{(p,\phi_p) \in M} L(A_p \mid \phi_p) \right) + L(S_R)$$

for the number of bits to encode S_y given a model M and observed events S_x.

Encoding a Model

Next, we formalize how we encode a model $M \in \mathcal{M}$ in bits. At a high level we have

$$L(M) = L_{\mathbb{N}}(|M|) + \sum_{(p,\phi_p) \in M} L(p) + L(\phi_p) \ ,$$

where we first encode the number of patterns in the model, and then the patterns and their associated delay distributions.

Patterns are essentially just a sequence of k events from Ω. We use $L_{\mathbb{N}}$ to encode their length, and to avoid any bias towards events $e \in \Omega$,

we encode the actual events in p using an index over Ω. Thus the cost of one pattern is

$$L(p) = L_{\mathbb{N}}(|p|) + |p| \log |\Omega| \,.$$

To encode a time delay distribution, it suffices to encode which time deltas have a probability greater than zero, and then encode how likely each of these deltas is. We write $\Delta_p = \{\delta \mid \phi_p(\delta) > 0\}$ for the set of δ values with non-zero probability, and $\delta^* = \max(\Delta_p)$ for the highest value of delta with non-zero probability. Formally, we then have

$$L(\phi_p \mid S_x) = L_{\mathbb{N}}(\delta^*) + \log(\delta^*) + \log \binom{\delta^* + 1}{k} + \log \binom{||S_z^p||_1 - 1}{k - 1} \,,$$

where we first encode interval of possible deltas, $[0, \delta^*]$, simply by encoding the value of δ^* using $L_{\mathbb{N}}$. As we aim for actionable patterns, we are not interested in 'instantaneous' predictions. This allows us to repurpose $\delta = 0$ to mean 'skip', so avoiding unnecessary padding of the possible values that can be sent. Next we encode the number of δs with non-zero probability mass, $k = |\Delta_s|$, for which we need $\log \delta^*$ bits. We then encode those values $\delta \in \Delta_s$ through a strong number composition. The intuition is that the more deltas are left unused, i.e. $\phi_p(\delta) = 0$, the higher this cost. Finally, we have to specify the probability mass per δ, which reduces to encoding an index over a number composition, i.e. an index over every possible way to distribute the $||S_z^p||_1$ occurrences (balls) over k non-empty bins. Overall, the flatter the distribution, the more deltas we have to consider, the higher the cost will be, and hence we prefer peaked distributions.

Example 1. ***(continued)*** *To illustrate this, we continue with our running example. Consider again Fig. 2.2. The delay distribution has a non-zero entries for 'skip', and values of delta of* 1, 2*. This means we first encode the maximum delta, $\delta^* = 2$, and then how many deltas out of the range $[0, 2]$ have a probability greater zero. We then identify which, in this case three values, out of this range have non-zero probability. As there are only three possible values, this is trivial; the cost is 0 bits. Finally, we have to distribute the total probability mass of the four pattern occurrences of pattern ab over our three chosen delta, requiring 1.58 bits.*

This concludes the description of how we encode a model M in bits.

2.3.3 *The Problem, Formally*

With the above we can now formally state the problem.

The Minimal Event Prediction Problem *Given event sequence S_x over alphabet Ω and binary sequence S_y indicating time points of interest, find that set of predictive sequential patterns and associated time delay distributions $M = \{(p_1, \phi_{p_1}), (p_2, \phi_{p_2}), \ldots, (p_k, \phi_{p_k})\}$ and that alignment A of pattern occurrences to interesting events in S_y, such that the total encoded length*

$$L(S_y, M \mid S_x) = L(M) + L(S_y \mid M, S_x)$$

is minimal.

To solve this problem exactly we would have to consider a rather large, triply exponentially sized search space. As we do not wish to a priori limit the maximum length of any pattern beforehand, patterns can be up $|S_x| - 1$ long, resulting in $\sum_{i=1}^{|S_x|-1} |\Omega|^i$ possible patterns. Per pattern p, there exist $(||S_y||_1 + 1)^{||S_z^p||_1}$ possible alignments. This leaves the final part, in which we have to select a set of patterns-alignment tuples. We can limit the number of tuples in our model to $||S_y||_1$. Combined this gives us

$$\sum_{j=0}^{||S_y||_1} \binom{\sum_{p \in P} (||S_y||_1 + 1)^{||S_z^p||}}{j}$$

possible solutions, where P is the set of all patterns. Although we are not necessarily afraid of large search spaces, unfortunately this particular search space does not exhibit structure such as (weak) monotonicity, convexity, or submodularity, that we could exploit to guide our search.

Hence, we resort to heuristics.

2.4 ALGORITHM

In this section we present the OMEN algorithm for heuristically solving the Minimal Event Prediction Problem. Rather than solving it at once, we split the problem in two parts and propose effective solutions for both. First, given a pattern we aim to find its delay distribution by optimizing the alignment between pattern occurrences and interesting events. Second, we consider the problem of discovering pattern sets.

We first introduce notation that will ease the exposition below. Whenever clear from context, we will write $L_p(S_y)$ rather than $L(S_y, \{(p, \phi_p)\} \mid S_x)$ to denote the encoded length of S_y under a model M that consists of a single pattern p. Analogue, we write $L_0(S_y)$ to denote the length of S_y using the empty, or null model M_0, which encodes all interesting events through the residual S_R. Finally, we overload the notation of an alignment A_p, allowing ourselves to represent an alignment as a set of tuples (i, j) where i is the location of the pattern match in S_x, and j the location of the aligned interesting event (or 'skip') in S_y.

2.4.1 *Discovering Alignments and Delay Distributions*

We start by discussing how to optimize an alignment A_s for a given pattern p. Solving $L_p(S_y)$ exactly would require us to consider all possible alignments, which is computationally unfeasible. We will therefore instead minimize $L_p(S_y)$ heuristically. The overall strategy is as follows. Given an initial alignment A_p, and a corresponding delay distribution ϕ_p, we iteratively optimize $L_p(S_y)$ by taking all pattern occurrences with the lowest $\phi_p(\delta)$ and for each either reassigning it to an interesting event such that we obtain a higher $\phi_p(\delta')$, or if no such interesting event exists, we map this pattern occurrence to 'skip' instead. We repeat this process for all $\delta \neq$ 'skip' and finally, return that alignment A_s with the lowest $L_p(S_y)$. We show a visualization of this process in Figure 2.3. In the worst case, we will have to consider all possible $\delta \in \Delta_p$ for each pattern occurrence that we reassign, by which we have a time complexity of $\mathcal{O}(||S_z^p||_1|\Delta_s|)$.

To optimize our alignment we need a initial alignment that can be optimized. We will introduce two such methods. We give a general purpose approach below and discuss an approach particular adapt at long range prediction in Section 2.4.4.

Our general purpose alignment initialization algorithm, AlignNext, consists of three main steps, each updating the alignment. First, based on the assumption that each pattern occurrence predicts the directly following interesting event, we simply align every occurrence $S_x[i] = p$ of pattern p in S_x to that interesting event in S_y that is closest in time but at least one time step into the future, i.e. $A_p = \{(i, j) \mid S_z^p[i] = 1 \wedge \arg\min_{j>i} S_y[j] = 1\}$, and then determine the corresponding delay distribution ϕ_p from this alignment. As by naively mapping occur-

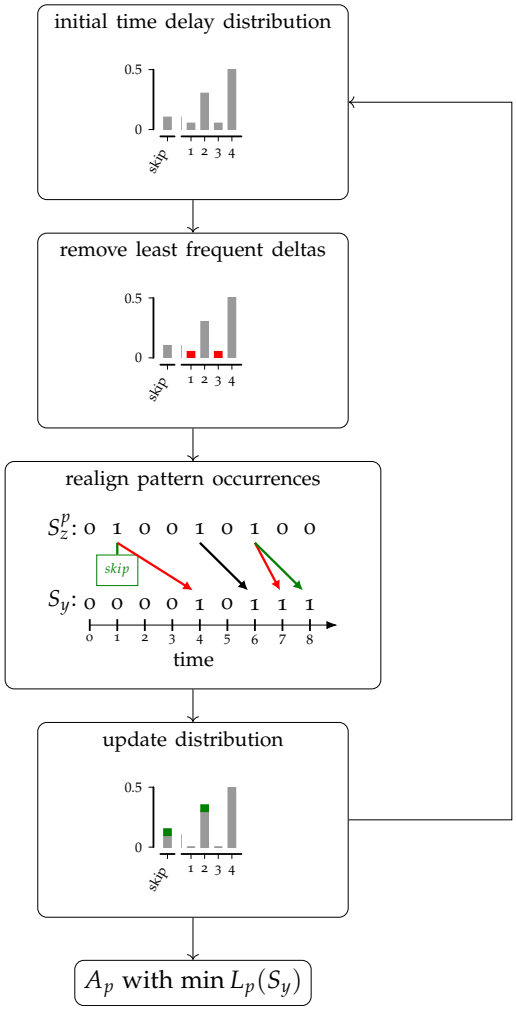


Figure 2.3: **[Alignment optimization process]** Given an initial time delay distribution, our goal is it to simplify the distribution to that version that minimizes the number of bits needed to describe S_y, essentially we trade of number of events explained against the number of deltas in the distribution. To simplify the given distribution we first drop all least frequent deltas from the distribution. Secondly reassign the corresponding pattern occurrences to another event in S_y or, if not possible, to 'skip'. Third we update the delay distribution to match the new assignment. We repeat these steps until no deltas are left in our distribution, while keeping track of how many bits each alignment does need to encode S_y. Finally we return that alignment with min $L_p(S_y)$.

rences to the earliest interesting event, multiple pattern occurrences may map to the same interesting event, while leaving other interesting events unexplained. We hence next re-align all such pattern occurrences to 'skip', except for the one with maximal $\phi_p(j-i)$, and re-infer the delay distribution. Last, we now use the new distribution to align every pattern occurrence mapped to 'skip' to that interesting event $S_y[j] = 1$ that maximizes $\phi_p(j-i)$. If there is no interesting event with non-zero probability under ϕ_p, we map it to 'skip'. Reassigning a 'skip' requires us to consider all possible $\delta \in \phi_p$ in the worst case, resulting in a complexity of $\mathcal{O}((||S_z^p||_1)^2)$. We can now, given a pattern, find an alignment between S_z^p and S_y.

As we will see in the experiments, this approach works well in practice. However, it is easy to see that it has a bias towards alignments where interesting events are close in time to the pattern occurrences. Differently put, if we would increase the mean of the delay distribution, this approach will give a more and more undefined delay distribution. To obtain good initial alignments for such cases, we introduce an alternative initialization algorithm in Section 2.4.4.

With the above, we know how to find an alignment A_p for a given pattern p, and can hence compute our score. Next, we discuss how to find good patterns.

2.4.2 *Discovering a Good Set of Patterns*

There exist exponentially many patterns, and exponentially more *sets* of patterns. Evaluating these exhaustively is not feasible, and as there is also no structure that we can exploit, exact search for the best set of patterns is not an option. Instead, we propose a heuristic to find a good set of patterns. In particular, we take a greedy bottom-up approach, where we iteratively add and refine patterns in the model. Because of the complexity of S_x and sparsity of S_y, it will often be the case however that only (large parts of) a true pattern p will lead to a gain in compression, while all small fragments of p do not improve over the null model. That is, we cannot directly use the score defined above to identify whether a pattern is 'promising', and as only in trivial cases singleton events in S_x will help to compress S_y we cannot start by adding 'good' singletons and then refine them.

Rather than resorting to exhaustively scoring every possible pattern p under some arbitrary constraints, we define an optimistic estimator $\bar{L}_p(S_y)$ by which we bound the length $L_{p'}(S_y)$ of the theoretically best possible extension p' of p. The key idea is that by extending p to p', the number of occurrences will monotonically decrease. We hence aim to estimate the score of the theoretically best possible extension (refinement) p' of p that exactly obtains the subset of occurrences of p that align best with S_y. Interestingly, this is equivalent to only aligning the 'best' occurrences of p to interesting events in S_y, and treating the remaining pattern matches as if they do not exist—i.e. play-pretending that the (unknown) p' simply does not match those occurrences of p. We can straightforwardly achieve this by mapping the non-matching occurrences of p to 'skip', and treating the encoding cost for such skipped pattern occurrences as zero. Hence, $\bar{L}_p(S_y)$ gives the length of S_y where we set $\phi_p(skip) = 1$ for the encoding of A_s and $\phi_p(skip) = 0$ for the encoding of P_s, as if only the pattern occurrences aligned to interesting events exist.

Exactly optimizing the subset of occurrences of p, i.e., finding those that together compress S_y best, is infeasible as it requires us to find the best alignment for every possible subset of occurrences. We therefore instead start by inferring an initial alignment using either the algorithm described above or the one in Sec. 2.4.4, optimize that alignment as described in Section 2.4.1, but setting the cost of 'skip' to zero, and so obtain $\bar{L}_p(S_y)$. With $\bar{L}_p(S_y)$ we know, if a gain is found, whether there exists a subset of occurrences of p that would improve our model and the next task is hence to find whether there indeed exists an extension p' of p that does improve our score.

With the optimistic estimator in place, we can now introduce the Omen algorithm for mining sets of predictive patterns. We give the pseudo code as Algorithm 1. Omen starts with an empty model where all interesting events are unexplained by patterns, i.e. encoded via residual S_R (line 1). The main idea is to iteratively add patterns to the model M that predict interesting events in S_R, by which we focus the search on those interesting events that are currently not yet predicted.

Starting from the singletons as candidate set C, we take a breadth-first search approach where we extend all candidates that are promising with regard to our optimistic estimate (l. 4–6) and identify those that help to better compress S_y (l. 7–8). As extensions of promising pat-

Algorithm 1: OMEN

input : event sequence S_x and binary sequence S_y
output: model M

1 $M \leftarrow \emptyset$; $S_R \leftarrow S_y$
2 $C \leftarrow \Omega$; $C' \leftarrow \emptyset$; $P \leftarrow \emptyset$
3 **while** C is not empty **do**
4 **foreach** $p \in C$ **do**
5 **if** $\bar{L}_p(S_R) < L_0(S_R)$ **then**
6 $C' \leftarrow C' \cup \{p + e, e + p \mid \forall e \in \Omega\}$
7 **if** $L_p(S_R) < L_0(S_R)$ **then**
8 **add** p to P
9 **foreach** $p \in P$ ordered by $L_p(S_R)$ **do**
10 **if** $L_p(S_R) < L_0(S_R)$ **then**
11 $p', \phi_{p'} \leftarrow$ REFINE(p)
12 **add** $(p', \phi_{p'})$ to M
13 $\widehat{S_y} \leftarrow$ compute from M
14 $S_R \leftarrow \widehat{S_y} \oplus S_y$
15 $C \leftarrow C'$; $C' \leftarrow \emptyset$; $S \leftarrow \emptyset$
16 **return** M

terns we consider all patterns p' where we add any single symbol from alphabet Ω at either the end or beginning of p (l. 6). Next, we iteratively consider adding those candidates $p \in S$ that passed the compression-check into the model (l. 9–14). We do so in order of how much they help to compress (l. 9) and to avoid redundancy only consider those that indeed improve the score (l. 10)—for example, if *abcd* is the true pattern and $ab \in S$ and $cd \in S$ then both (probably) explain the same interesting events. Once we added one, the other does not offer any additional information. When adding a pattern to our model we search for the best possible refinement, that is we greedly extend our pattern to that version that gives us the highest gain in bits saved, we refer the reader to Appendix a.1.1 where we provide a more detailed explanation. After adding a pattern we recompute the residual (l. 13–14). We repeat these steps until we have no further candidates (l. 3), and then

return the final model M. In Figure 2.1 we show a visualization of this algorithm.

In the worst case OMEN considers every possible sequential pattern over Ω, which means a complexity of $\mathcal{O}(|\Omega|^{|S_x|-1})$.

2.4.3 *Faster* OMEN

Algorithm 2: FOMEN

input : event sequence S_x and binary sequence S_y
output: model M

$M \leftarrow \varnothing$; $S_R \leftarrow S_y$
$C \leftarrow \{\{a, b\} \mid a, b \in \Omega, \quad a \neq b\}$
do
  $\{a, b\} \leftarrow \arg\max_{\{a,b\} \in C} ||\hat{S}_y^a \wedge \hat{S}_y^b||_1$ // *Ignore* `used` *events for singleton pairs.*
  $C \leftarrow C \backslash \{a, b\}$
  $p \leftarrow \arg\max_{p \in \{ab, ba\}} ||S_z^p||_1$
  if $\bar{L}_p(S_R) < L_0(S_R)$ **then**
    foreach $\tilde{p} \in \{a, b\}$ **where** $|\tilde{p}| = 1$ **do**
      $C \leftarrow C \cup \{\{\tilde{p}, p'\} \mid p' \in \bigcup_C\}$
    $A_p \leftarrow$ from $\bar{L}_p(S_R)$
    foreach $(i, j) \in A_p$ where $j \neq$ '*skip*' **do**
      mark those events belonging to i as used
    if $L_p(S_R) < L_0(S_R)$ **then**
      $p^*, \phi_{p^*} \leftarrow$ FREFINE(p)
      $M \leftarrow M \cup (p^*, \phi_{p^*})$
      $\hat{S}_y \leftarrow$ compute from M
      $S_R \leftarrow \hat{S}_y \oplus S_y$
    else
      $C \leftarrow C \cup \{\{p, p'\} \mid p' \in \bigcup_C\}$
while $|C| > 0$
return M

As we will see in the experiments, OMEN works very well in practice. At the same time, it is a bit naive: it considers as candidates ex-

tended patterns *es* and *se*, without taking into account whether these extensions e predicts any of the same interesting events as p. As this information is readily available, an alternate and more targeted search strategy presents itself. We call this, much faster, procedure FOMEN. We give the pseudo code of FOMEN as Algorithm 2. We first describe the algorithm in general, and then detail the special case of singletons.

The main idea is to greedily combine those patterns in our candidate set that have the largest intersection of predicted events (l. 4). When combining two such patterns, a and b, we choose that ordering ab or ba that results in the most pattern matches in S_x (l. 6). If the optimistic estimator indicates that the new pattern is potentially compressing, we mark the events in S_x for each *used* pattern occurrence as *used*[1] (l. 12), for reasons that will become apparent when we study the singletons. We add all pairs between patterns already in our candidate set and the newly created pattern to our candidate set (l. 19). Unless the new pattern predicts previously unexplained events, i.e. those in the residual, then we greedily refine it to its best version (l. 14) and add the refined version to our model, compute a new partial prediction and update our residual accordingly (l. 15 to 17). We repeat this process until no candidates are left. To prune candidates that are very unlikely to lead to improvement, we permit the user to set a threshold on the minimum overlap in predicted events. Per default we set this threshold to $0.01 \times ||S_y||_1$.

The refinement of an already compressing pattern works similar to the mining approach. We simply keep combing those patterns with the highest intersection of predicted interesting events, up until the optimistic estimator indicates no better extension exists. We then return the refined pattern with lowest $L_p(S_R)$. We give the pseudo code in Appendix a.1.1, Algorithm 15.

We now consider the special case of candidates (a, b) where a and b are singletons. If we were to treat such candidates the same as we do above, and for example find that ba is the most promising instantiation of (a, b), we would remove (a, b) and add any combination (ba, p) where p is a pattern in C to the candidate set. For non-singleton patterns this is fine, as long as we can re-build them from scratch if needs be, which is even desirable from a run-time perspective. However, to be able to re-build patterns we do need to ensure that the singletons,

1 We say a event is *used* if it is part of pattern that is aligned to an interesting event.

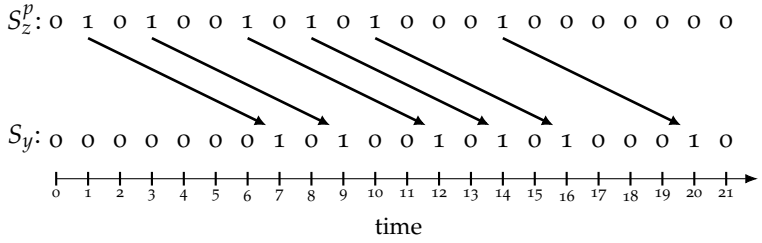


Figure 2.4: Toy example for time-overlapping predictions. S_z^p shows the pattern occurrences of pattern p. The arrows show the ground truth alignment to the interesting events in S_y. The second (and third) pattern match occur before the interesting event predicted by the first pattern. To find good alignments in settings like this, we give a specialized alignment approach in Section 2.4.4.

the most elementary building blocks, are always present as candidates. To make sure they are, we hence should not simply remove singleton pairs (a, b) after exploring (and possibly accepting into the model) their refinement. At the same time, we should not just put (a, b) back just like that, as the just-extended pattern will already explain some or even many of the interesting events. If we do not account for this we would hence be too optimistic in our estimate $\bar{L}_p(\cdot)$ of the potential gain of any new pattern that is based on the re-added (a, b). To this end, we mark those events that the just-extended candidate pattern *uses*, and update the intersection counts of the singleton pair with unused interesting events. If the optimistic estimate of (a, b), $\bar{L}_{(a,b)}(S_R)$, indicates a gain, we add (a, b) back into the candidate set, and otherwise discard it.

2.4.4 *Alignment with Overlapping Predictions*

As the final technical contribution, we propose a fast alignment strategy that is particularly suited for cases where there are long delay between patterns and interesting events, and hence a high chance of overlap of occurrence-prediction intervals. As an example of why such an algorithm is necessary in addition to the general purpose one described above, consider Fig. 2.4. It is easy to see that the vanilla strategy, where we initially map a pattern occurrence to the earliest interesting event, would fail to discover a sensible (let alone, the ground truth)

initial alignment; because the third, fourth, and fifth occurrence would all be mapped to an interesting event at the next time step, they would amass as much probability density as the true delay of 6 time steps, and so create a local minimum out of which the optimization cannot escape.

We therefore present an alternative alignment algorithm, AlignFar, which is unaffected by such overlapping predictions. At a glance, it first computes the mean of our delay distribution and then assigns to each pattern the interesting event that is closest to this mean. More specifically, we seek this delay μ for which each pattern occurrence delayed by μ lies as close as possible to an interesting event occurrence following the pattern occurrence. We propose an algorithm that computes this optimal delay within $O(\|S_z^p\|_1 \|S_y\|_1 \log \|S_z^p\|_1)$ time.

For convenience we define the set of occurrence indices for the patterns, $I_p = \{i \mid S_z^p[i] = 1\}$, and the set of occurrences for the interesting events, $I_{S_y} = \{i \mid S_y[i] = 1\}$. Given a pattern occurrence $i_p \in I_p$ we can now express its candidate interesting event occurrences as the set $I_{S_y}^{i_p} = \{i_Y | i_Y \in I_{S_y} \wedge i_Y > i_p\}$; this set consists of all event occurrences that follow the pattern occurrence. Intuitively, we seek to find the delay μ that minimizes the total distance between every pattern i_p—delayed by μ, and the event occurrence within $I_{S_y}^{i_p}$ that lies nearest to i_p. As a distance between these two occurrence indices we adopt the quadratic one, after which we can formally express our problem as:

$$\mu^* = \arg\min_{\mu \in \mathbb{R}} \sum_{i_p \in I_p} \min_{i_Y \in I_{S_y}^{i_p}} (i_p + \mu - i_Y)^2 \,. \tag{2.1}$$

To demonstrate the intuition of this we consider the toy setting of Figure 2.5, in which the pattern matches $I_p = \{3, 6, 8\}$ and the interesting events $I_{S_y} = \{4, 5, 11, 12\}$. Out of the latter, the candidate interesting event occurrences for the each pattern match are the sets $I_{S_y}^3 = \{4, 5, 11, 12\}$, $I_{S_y}^6 = \{11, 12\}$, and $I_{S_y}^8 = \{11, 12\}$, respectively.

We can optimize Equation 2.1 by the following procedure. We first create the union of all event candidate offsets $I_{S_y}^{\cup} = \bigcup_{i_p \in I_p} I_{S_y}^{i_p}$, sorted in ascending order.

Lemma 1. *Let $I_{S_y}^{\cup}$ be the sorted (multi-set) union of all possible interesting event candidates and denote U the midpoints between each 2 consecutive elements of $I_{S_y}^{\cup}$. Then, the candidate within $I_{S_y}^{\cup}$ that lies closest to a pattern*

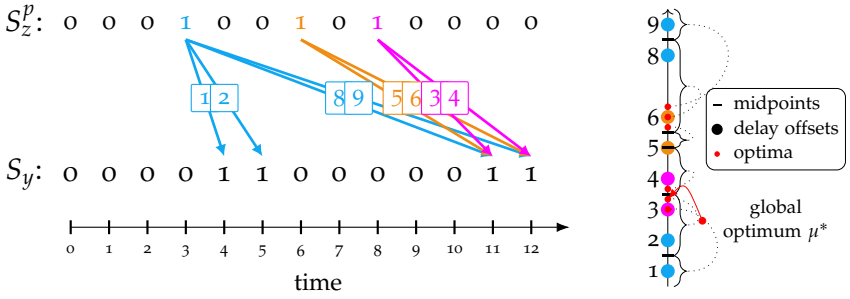


Figure 2.5: Toy example showing three pattern matches with all possible alignments (left) and the resulting midpoints (right) that define the intervals with local optimum to be tested in the AlignFar alignment algorithm. The global optimum lies between 3 and 4 as it has the smallest distance to all candidate pattern matches.

occurrence changes only when μ crosses a midpoint in U. This change affects exactly those pattern occurrences that contributes an elements in $I^{\cup}_{S_y}$ adjacent to the crossed midpoint in U.

Using Lemma 1 we can partition the real line in exactly $|I^{\cup}_{S_y}| + 1$ segments,[2] within which the configuration of closest events remains constant, and therefore also the objective value of Eq. (2.1). For each segment we can compute this value, as well as its optimiser, by carefully keeping track of the updates induced to the optimiser and optimal of the preceding segment, and in fact in constant time. By using appropriate data structures and a mutatis mutandis `MergeSort` algorithm we can perform the search for the next midpoint in $O\big(\log \|S_z^p\|_1\big)$ time. Overall, this gives our algorithm a computational complexity of $O(|I^{\cup}_{S_y}| \log \|S_z^p\|_1)$.

Returning to the example of Figure 2.5, the delay offsets $i_p - i_Y$ in Eq. (2.1) for each pattern match lie in the sequences $(1, 2, 8, 9)$, $(5, 6)$, and $(3, 4)$, respectively, as shown in Figure 2.5 (right). Importantly, the union of all midpoints between every two consecutive elements in each of the above sequences gives the set $I^{\cup}_{S_y} = \{1.5, 3.5, 5, 5.5, 8.5\}$. Due to Lemma 1, for each interval between consecutive midpoints the best offset $i_p - i_Y$ for each pattern remains constant, and yields one single minimiser of μ for the specific interval (which need not necessarily lie

2 Equal elements in the set are gracefully treated, but they still contribute to the complexity of the algorithm.

within the interval itself). Thus, the global optimizer can be retrieved as the minimum over all these per-interval minimisers, which here is $\mu^* = \min\{3, \frac{10}{3}, \frac{11}{3}, \frac{17}{3}, 6, \frac{19}{3}\} = \frac{10}{3}$.

Once we have computed our optimizer μ^* of Eq. (2.1) we can simply select for each pattern occurrence i_p the interesting event closest to $i_p + \mu$ from it's candidate set $I_{S_y}^{i_p}$. If the candidate set is empty we align that pattern occurrence to a 'skip'. This gives us an alignment for all pattern occurrences without a bias towards closer interesting events.

2.5 RELATED WORK

Our work is related both to prediction and information flow in time series, as well as to pattern mining.

At its core, the OMEN score aims to measure how the occurrences of a pattern p in S_x help to reliably predict the occurrences of interesting events in S_y. As such, it is strongly related to Granger causality [69]. Granger causality is based on the idea that if the past of a time series A does help to predict S_y, given the past of S_y, it "Granger-causes" S_y. Linear [69] and nonlinear [23] scores have been proposed, whereas others studied the effects of events on time series [156]. Transfer Entropy (TENT) [159] and CUTE [17], are both information-theoretic instantiations of Granger causality for discrete data, where the one measures information flow in terms of entropy, and the other in terms of MDL. In the experiments we will compare our score to both.

Prediction and forecasting in time series [27, 100, 155, 188, 189, 204, 206] is a classic research topic to which OMEN is related. A prototypical example is failure prediction, where the goal is to predict upcoming failures with sufficient time to act on the prediction. Existing work focuses on the case where S_x is continuous valued and the goal is to discover time points where the distribution of S_x changes [80, 186]. Another popular task is the prediction of upcoming events based on social media activities and text [62, 148]. Related to prediction is the task studied by Yeh et al. [197] they considered the problem of reconstructing a boolean annotation sequence given a real valued time series.

Mining sequential patterns from event sequences has a rich history. Traditional sequential pattern miners focus on finding all frequent patterns [4, 99, 112]. We can differentiate between two settings, based on

the notion of support of a pattern. The first, where we have a database of sequences and support is defined by the number of instances containing a pattern [183], and the second where we have one or multiple sequences where support is defined as the number of occurrences within a sequence, measured using either a sliding window [112] or counting the number of minimal windows [99]. Both settings suffer from exponentially many patterns, making interpretation hard to impossible. Closed episodes [182, 196] 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 [86, 108, 144, 172]. 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 [57] takes a probabilistic Bayesian approach. SQS [170] 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 [11] builds upon SQS and additionally allows interleaved and nested patterns. Bertens et al. [10] propose a pattern set miner for multivariate event sequences. All use greedy search algorithms, iteratively adding patterns to a model until convergence. Although all related to OMEN in the sense that they also discover small sets of patterns, these methods are all strictly unsupervised. In the experiments we will compare to appropriately modified versions of both SQS [170] and ISM [57].

Identifying patterns that predict the occurrence of events can be approached as a supervised sequence classification problem, where given a labeled sequence database the goal is to find those sequential patterns that allow a classification [8, 48, 205]. SCIS [205], a recently proposed rule-based sequence classifier, is trained by mining rules above a set interesting threshold, where interestingness is the product of cohesion (average proximity of items that make up that rule) and frequency. A unseen sequence is classified by taking the top-k best fitting rules. We consider a different setting, where instead of a sequence database, we only have two sequences S_x and S_y, where S_y can be interpreted as our labels. We include a comparison to SCIS in the experiments.

2.6 EXPERIMENTS

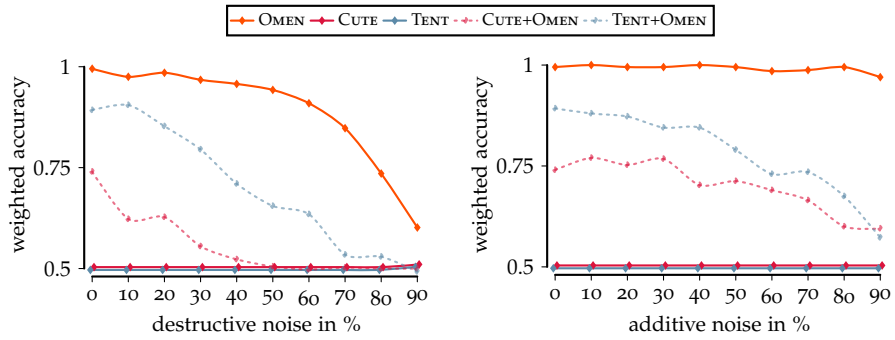
In this section, we will empirically evaluate on synthetic and real-world data. To determine how well our score and methods can tell predictive from non-predictive patterns, we compare to TENT [159] and CUTE [17], and to determine how well they discover predictive patterns from data we compare to SQS [170], BIDE+ [183], ISM [57], and SCIS [205].

We implemented OMEN in C++, and provide the source code for research purposes, along with all datasets, experiment specifications, and generators.[3] All experiments were executed single-threaded on machines with two Intel Xeon CPU E5-2643 processors and 256 GB of memory, running Linux. We report wall-clock running times.

2.6.1 *Synthetic Data*

To assess performance against known ground truth we consider synthetic data, which we generate as follows. We first generate event sequence S_x of length $n = 300\,000$ by uniformly at random drawing n events from an alphabet Ω of length 50, i.e. $S_x \in \Omega^n$, and initialize $S_y = \{0\}^n$. We add structure by planting 20 predictive and 10 non-predictive patterns. Every pattern is generated independently, where we first draw its length l from $[3, 6]$, its events $p = \{e_1, \ldots, e_l\}$ from Ω, and its frequency f from $[5, 50]$, again all uniformly at random. We plant patterns into S_x by sampling u.a.r. f insertion positions $i \in [0, n]$, where we simply overwrite the existing values in S_x, i.e. $S_x[i : i + l] = p$. To ensure the ground truth holds, we do not overwrite previously planted patterns. For the predictive patterns p we additionally generate interesting events in S_y by, per insertion position i, sampling a delay δ u.a.r. from $[8, 12]$ and setting $S_y[i + l + \delta] = 1$. Finally, we add noise to the data by flipping values in S_y. We consider both destructive noise, where we flip 1s to 0, as well as additive noise, where we flip 0s to 1. Unless specified otherwise, all results on synthetic data are averaged over 10 independently generated datasets.

3 https://eda.rg.cispa.io/prj/omen/



(a) Accuracy under destructive noise.

(b) Accuracy under additive noise.

Figure 2.6: [Higher is Better] OMEN reliably determines predictiveness of patterns, both for (a) destructive and (b) additive noise, meaning that in S_y we flip 1s to 0s, resp. 0s to 1s. Stand-alone, neither CUTE nor TENT differentiate at all between predictive and non-predictive patterns. When applied on the $\widehat{S_y^p}$ discovered by OMEN and ALIGNNEXT, they do achieve reasonable performance (CUTE+OMEN, resp. TENT+OMEN, dashed lines). OMEN beats all methods by a large margin.

2.6.2 *Evaluating the Score*

We first evaluate how well OMEN can tell predictive from non-predictive patterns. For OMEN, we consider a pattern to be predictive if it helps to compress S_y. We compare OMEN to TENT [159] and CUTE [17], two state of the art methods based on Granger causality that measure how much information S_z^p provides towards S_y. For both we say S_z^p predicts S_y if they conclude that S_z^p Granger-causes S_y. We optimize the lag-parameter of TENT over $[1, 15]$ per experiment.

We generate data with varying amounts and type of noise as described above, and for every planted pattern in every dataset test whether the score considers it predictive or not. We report average weighted accuracy, defined as

$$\frac{1}{2}\left(\frac{tn}{tn+fp}+\frac{tp}{tp+fn}\right)$$

where tp = true positives, fp = false positives, tn = true negatives, fn = false negatives. We report the results for increasing additive and destructive noise in Figure 2.6. We see that OMEN is able to identify

predictive patterns with high accuracies even for large amounts of noise, whereas TENT and CUTE applied on S_z^p and S_y reduce to a coin flip as they expect all (most) interesting events to be explained by S_z^p. When we apply TENT and CUTE not on the raw data S_y but rather on that $\widehat{S_y^p}$ that OMEN discovers as the best explanation of S_y given p (CUTE+OMEN and TENT+OMEN), we see that their accuracies increase up to 90%, yet remain much worse than those of OMEN.

2.6.3 *Evaluating the Discovered Patterns*

Next, we evaluate how well OMEN discovers predictive patterns from synthetic data.

EVALUATION METRIC We first explain how we evaluate a discovered pattern set, as metric to evaluate success, we compute recall and precision,

$$\text{recall} = \frac{tp}{|\{\text{planted patterns}\}|} \quad , \quad \text{precision} = \frac{tp}{|\{\text{recived patterns}\}|} \quad ,$$

and the harmonic mean between the two, the $F1$ score,

$$F1 = \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}}.$$

We compute the scores over a pair-wise mapping between planted and discovered patterns. We map each reported pattern to at most one planted pattern and to each planted pattern at most one reported pattern. We allow a mapping between two patterns if the found pattern is a subsequence of the planted one or vice versa. We choose that mapping that results in the maximum number of pairs. This also allows us to rewards partial discoveries. As the number of true positives, i.e. correctly received pattern, we consider the number of pairs in the mapping, consequently the false positives are $|\{\text{recived patterns}\}| -$ true positives. Next, we explain how we compute this mapping.

To find the maximum number of pairs we reformulate the problem as a flow network optimization problem. We connect each planed pattern to our source and each found pattern to the sink. We connect a planted to a found pattern if the found is a subsequence of the planted one or vice versa. The capacity of all links is set to one. We then compute the maximum flow of the created flow network which maximizes

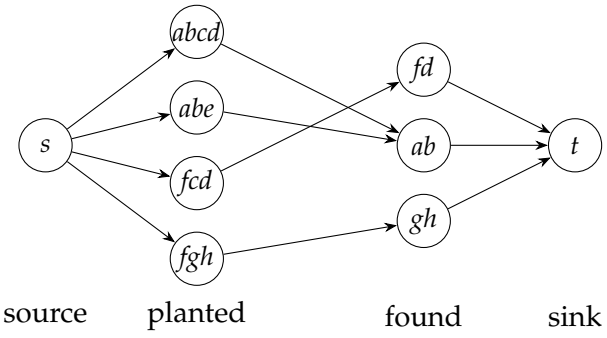


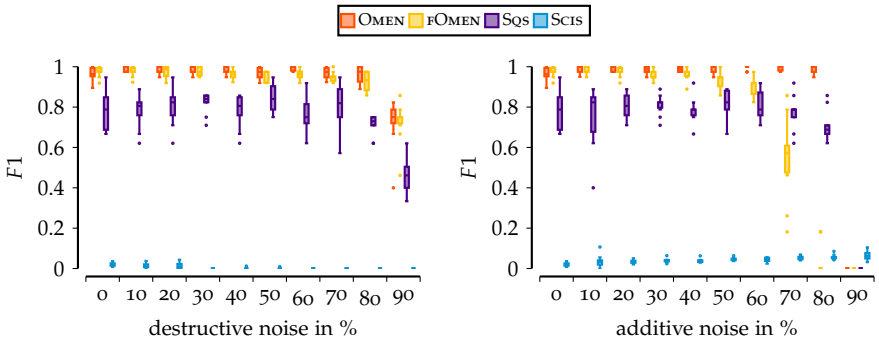
Figure 2.7: Toy example of how we match discovered to planted patterns. A discovered pattern *ab* will be matched to either *abcd* or *abe*, but not both; maximizing the flow gives us the number of recovered patterns.

the number of planted, found pairs. This setup ensures that we match at most one found pattern to a planted pattern and vice versa. We maximize the flow using the Edmonds-Karp Algorithm [47]. In Figure 2.7 we give a toy example of such a flow network.

While there might exist multiple equivalent pairing solutions, this is not a problem for us, as we are not interested in the actual pairing. In the Example shown in Figure 2.7 we find pattern *ab* with just the pattern we do not know if it is a partial pattern of the planted pattern *abcd* or *abe* or both and hence count it as one correctly found pattern, which means we did not find, according to our evaluation metric, either pattern *abcd* or *abe*.

SETUP We compare OMEN to BIDE+ [183], SCIS [205], SQS [170], and ISM [57]. SCIS discovers predictive patterns (rules) given a labeled sequence database as input. As positive samples we consider the window of w events in S_x that lead up to each interesting event in S_y. As negative samples we then split the remaining data into non-overlapping windows of length w, which avoids skew (in positive and negative samples) as well as bias (non-intersecting with positive samples). We ensure SCIS can discover all planted patterns by setting $w = 20$. From all reported rules we consider those who predict the positive class.

BIDE+, SQS, and ISM are all unsupervised by nature, but, we can use them to discover predictive patterns as follows. First, we split S_x at

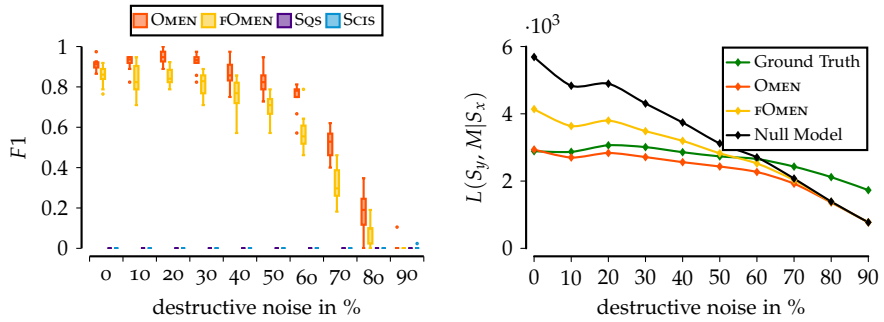


(a) $F1$ scores under destructive noise (b) $F1$ scores under additive noise

Figure 2.8: [Higher is better] $F1$ score results for OMEN, FOMEN, SQS, and SCIS, on synthetic data with "frequently" occurring patterns and (a) destructive resp. (b) additive noise in S_y. SCIS fails to report any meaningful patterns. SQS, with our score to filter out non-predictive patterns, recovers a large fraction of the ground truth patterns. FOMEN outperform both by a large margin, except for high levels of additive noise. OMEN is the best method overall.

the location of the interesting events in S_y—so creating a database of sequences leading up to the next interesting event. We then run the respective method on the created sequence database—and subsequently use the OMEN alignment and score to identify, out of all reported patterns, those who are predictive. As in Section 2.6.2 we consider a pattern to be predictive if it does compress S_y. BIDE+ tends to discover far too many patterns, i.e. millions more than planted, and hence we only consider the top-200 of its results. SQS and ISM discover reasonable number of results, and those we consider all.

We first consider a setting which favors unsupervised pattern mining methods like SQS, ISM and BIDE+ by planting predictive patterns that are frequent in S_x, sampling the number of pattern occurrences from $[25 : 500]$. As we find that ISM reports only singletons, and BIDE+ does not return any predictive patterns, we omit them from further analysis. For the remaining methods we report the F1 scores in Figure 2.8. We see that OMEN and FOMEN both outperform SCIS and SQS by a wide margin. Quantitatively, SQS is a good second, but SCIS returns very many patterns that do not match the planted ones, and hence obtains very low precision and recall scores. FOMEN, and especially OMEN, have close to perfect F1 scores for nearly all noise levels.



(a) $F1$ scores under destructive noise (b) Encoded size under destructive noise

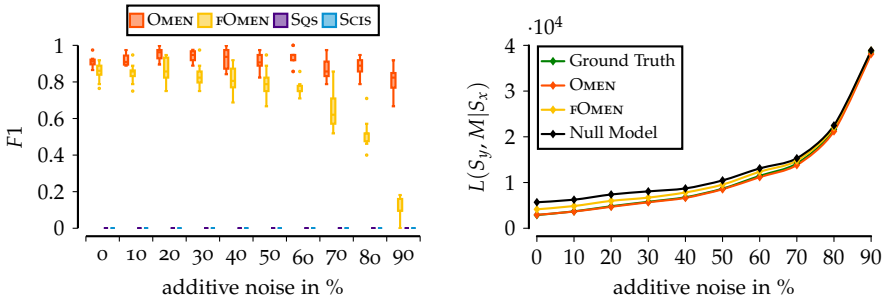
Figure 2.9: $F1$ score results (a, higher is better), and number of bits needed to encode S_y given different models (b, lower is better) on data with destructive noise. As models we consider the empty null model, the ground truth model, and the models discovered by OMEN and FOMEN respectively. OMEN and FOMEN report most of the planted patterns up to 60% of noise. SQS and SCIS do not report any predictive patterns. We observe that the reported models match the ground truth closely.

For the next analysis, we hence consider a more challenging setting where we sample the frequency of the planted predictive patterns instead from the range $[5 : 50]$. We report the F1 scores on the left-hand side of Figures 2.9, 2.10 and 2.11, and provide the precision and recall plots in Appendix a.2.1.

We see that in this more sparse setting, SQS and SCIS do no longer discover any predictive patterns; OMEN and FOMEN, on the other hand, still discover most up to all. We additionally see that OMEN is especially robust against additive noise, while for destructive resp. combined noise it performs well up to 60% noise. FOMEN performs slightly worse, but obtains these results in only a fraction of the time that OMEN takes, on average over these experiments it needs only 8 instead of 81 seconds.

2.6.4 *Evaluating the Discovered Models*

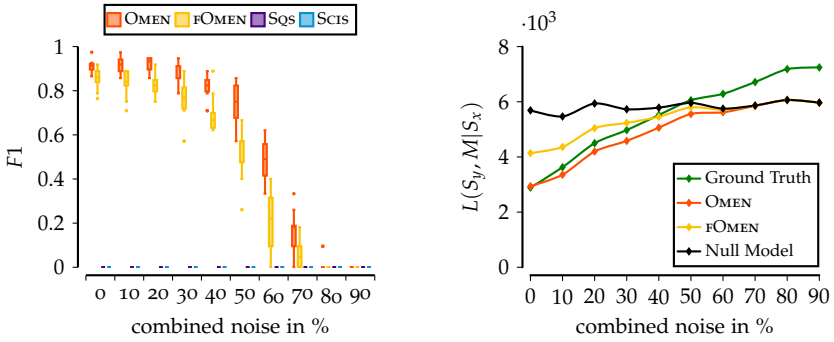
In addition to measuring performance in recall and precision, we evaluate how close the models that we discover get to the ground truth. We start with a sanity check, considering data where S_y is indepen-



(a) $F1$ scores under additive noise

(b) Encoded size under additive noise

Figure 2.10: $F1$ score results ((a), higher is better) and number of bits needed to encode S_y given the null model, the ground truth model and discovered model by OMEN and FOMEN respectively ((b), lower is better) on data with "unfrequent" patterns and additive noise. OMEN is especially robust against additive noise. We observe that the reported models match the ground truth near exactly, especially OMEN.



(a) $F1$ scores under additive and destructive noise.

(b) Encoded size under additive and destructive noise.

Figure 2.11: $F1$ score results ((a), higher is better) and number of bits needed to encode S_y given the null model, the ground truth model and discovered model by OMEN and FOMEN respectively ((b), lower is better) on data with "unfrequent" patterns, destructive and additive noise. OMEN and FOMEN report most of the planted patterns up to 50% of noise. SQS and SCIS do not report any predictive patterns. We observe that the reported models match the ground truth closely.

dent from S_x. We do so by generating data without noise and then destroying any dependence between S_x and S_y by randomly permuting S_y. When we run either OMEN or FOMEN on this data, both correctly report no patterns.

Using the same data generating scheme as above, we now compare the number of bits that OMEN resp. FOMEN require to describe the data, to the encoded length using either the ground truth model, and the null model where S_y is described without any patterns. We describe in Section 2.3 how the number of bits needed to encode S_y and M, under the respective models, is computed. We report the average over all experiment per noise level. We give the results on the right hand side of Figures 2.9, 2.10 and 2.11. In addition, we give the worst-case performance per noise level in Appendix a.2.1.

Overall, we see that both OMEN and FOMEN return high quality models, and are both highly robust against noise. We see that the results of OMEN in particular are always very close to the ground truth, and far from the null model—except for when the null model is the most succinct description of the data, after which it correctly returns the null model as the best (simplest) description of the data.

In some cases we discover a shorter description then the ground truth, which can be explained by the fact that due to noise, the ground truth pattern set and alignment does not necessarily have to be the shortest description of the data at hand. If we consider, for example, the case of destructive noise where, with increasing noise, pattern will predict fewer and fewer interesting events. At some point it requires less bits to describe these events using the residual than using patterns. This also explains the drop in F1 that we observed above for when there is destructive noise; above 80%, the data simply does no longer exhibit any significant structure.

Last, we consider the robustness of OMEN and FOMEN against patterns with noisy delay distributions. To this end we generate synthetic data as above, but, to keep the results interpretable, we now plant only a single pattern of length 6 that predicts 2 000 events. As time delay distribution we consider a Normal distribution with mean 50, and vary the standard deviation from 2 to 60 in steps of 2. We record the number of bits $L(S_y, M \mid S_x)$ needed by resp. the null model, the ground truth model, and the model discovered by our methods. We give the average results per standard deviation, out of 10 experiments, as Fig. 2.12.

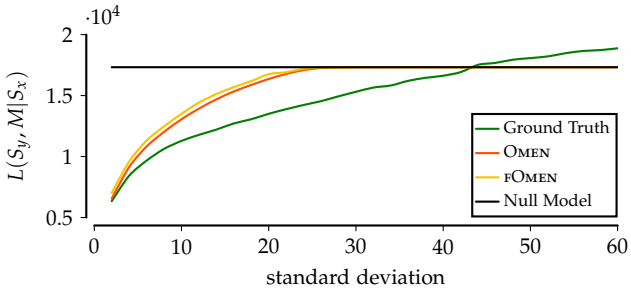


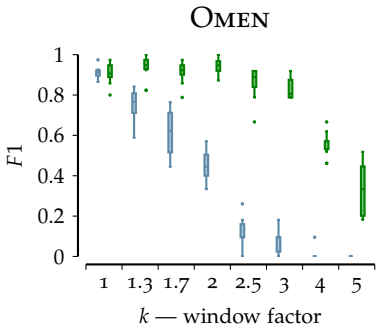
Figure 2.12: [Lower is better] For data with Gaussian-distributed time delays of mean 50, OMEN and FOMEN recover high quality models up to a standard deviation of 20.

We see that both OMEN and FOMEN are robust to patterns with wide delay distributions, discovering models that compress better than the null—and hence, discovering the true pattern—consistently up to a delay distribution with a standard deviation of 24. From a standard deviation of 44 onward the null model beats the planted model.

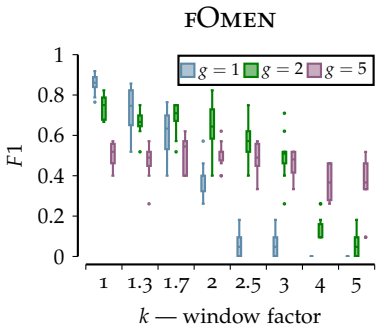
2.6.5 *Evaluating on Data with Gaps*

Until now, we have only looked at synthetic data with strict subsequences as patterns. Next, we evaluate on synthetic data where the planted patterns do have gaps. We keep the data generating process the same except we plant a pattern by sampling u.a.r. f insertion points $i \in [0, n]$ and for each sampled i we select a window $S_x[i : i + lk]$ where l is the length of the pattern to be planted and k is a factor, specifying how large the window is. To plant the pattern we sample l insertions points from the window and plant the pattern accordingly. For predictive patterns we set $S_y[\max \text{ sample} + \delta] = 1$

To evaluate how well our methods recover patterns with gaps, we consider a setting where we increase k in the experiment generation from 1 (no gaps) to 5 (window 5 times longer than the actual pattern). We present the results in Figure 2.13. Our base setup is to either not allow any gaps ($g = 1$) and allowing for a minimal window lengths of up to twice the length of the discovered pattern ($g = 2$). By its



(a) $F1$ scores for OMEN under increasing window sizes, i.e. more gaps.



(b) $F1$ scores for FOMEN under increasing window sizes, i.e. more gaps.

Figure 2.13: $F1$ score results on synthetic data where predictive patterns include gaps (higher is better). OMEN (a) and FOMEN (b) without allowing for gaps (i.e. by setting a maximum minimum-window length factor $g = 1$) does unsurprisingly poorly on data with gaps (i.e. data generated with a window factor $k > 1$). OMEN performs surprisingly well until we plant patterns with significantly larger windows than we allow it to model. FOMEN is much faster, allowing us to consider mine at a much larger gap factor ($g = 5$) but overall performs worse than OMEN as it is more susceptible to randomly generated (not planted) pattern occurrences.

efficiency, we also consider fOMEN with windows of up to 5 times the pattern length.[4]

We observe that if we do not allow for gaps, yet plant patterns with increasing window length, performance quickly converges to 0. When we do allow for gaps up to factor of twice the pattern length, we see that OMEN recovers (close to) all patterns, and does so up to a planted window length of $k = 3$, after which performance deteriorates. fOMEN shows the same performance for $g = 1$, and somewhat worse results for $g = 2$. As it is much more efficient than OMEN, we can also run it with a much higher gap factor of 5, which coincides with a much larger search space. We see that for this large gap factor the results are reasonable at best—by allowing for such large windows, it gets confused by random 'patterns' that are only due to chance.

2.6.6 *Evaluating on Data with Large Time Delays*

In our final experiment in synthetic data, we evaluate how well our methods deals with overlapping predictions, as shown in Figure 2.4. In particular, we aim to compare our two alignment approaches. To this extend we generate data where we slowly increase the expected overlap. We start with our usual experimental setup of 20 predictive and 10 frequent patterns, except that we now set the median of our planted delay distribution to 350, and slowly increase the number of pattern occurrences. We start by sampling from the range $[10, 50]$ and shift it up to $[100, 500]$. This slowly increases the number of pattern and therefore the likelihood that pattern predictions will overlap. We report the result in Figure 2.14.

For low overlap we observe that ALIGNFAR does significantly worse than ALIGNNEXT, which is explained by the small number of pattern occurrences; given the small pattern occurrences relative to the sequence length it is very likely that there exists a better alignment for pattern generated by chance. The bias of ALIGNNEXT towards interesting events that happen closer to the pattern occurrences here helps it to discover the correct patterns. Once we increase the overlap, however, we see that ALIGNFAR based methods quickly start to perform on par with ALIGNNEXT, and for an expected frequency of 120 and up it over-

4 Technically we allow for window of length $g \times l + 1$ where g is the gap factor and l the pattern length, except for no gaps case ($g = 1$).

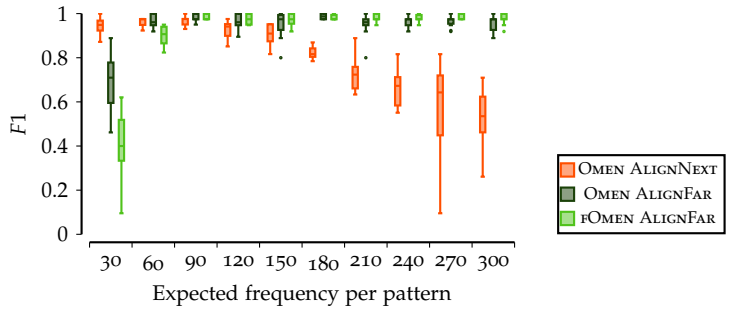


Figure 2.14: [Higher is better] $F1$ scores for synthetic data with increasing overlap. We observe that OMEN with ALIGNNEXT does well for lower frequencies, but performance decreases when frequency (and therewith the overlap) increase. With ALIGNFAR, OMEN is unaffected by the increasingly overlapping predictions.

takes it. We further see that for particularly high expected frequencies, FOMEN with ALIGNFAR consistently performs best.

2.6.7 *Evaluating on Real Data*

Last, we evaluate our methods on real world data. We consider three datasets, electrocardiograms (*ECG*) of a exercise stress test, a daily activities log (*Lifelog*) and water levels combined with precipitation records (*Saar*). We give the basic statistics in Table 2.1. Since FOMEN and ALIGNFAR essential reported the same patterns, analog performance relative to OMEN as for the synthetic data, we present the results reported by OMEN with the ALIGNNEXT, we compare to SQS and SCIS. As SQS allows for gaps, and real world patterns might show these, we interpret its results as minimal windows for which we again use the OMEN score and alignment to determine which ones are predictive.

On the *ECG* dataset the goal is to find patterns that predict the occurrence of a heartbeat. Our dataset is based on the first record (id 300.1) of the MIT-BIH ST Change Database.[5] We subsampled the record, replacing each 5 subsequent values with their average, transformed the result into a relative sequence by replacing each value with the difference to the previous value. Finally, using SAX [105] we discretize the

5 https://physionet.org/content/stdb/1.0.0/

				SCIS	SQS	OMEN		
Dataset	$\lvert S_x \rvert$	$\lvert \Omega \rvert$	$\lvert\lvert S_y \rvert\rvert_1$	$\lvert P \rvert$	$\lvert P \rvert$	$\lvert P \rvert$	$L\%$	t
ECG	107395	3	2558	41318	1	2	64.6	1.9
Saar-Rise	4018	17	278	419	0	7	71.3	0.08
Saar-Fall	4018	17	278	442	0	3	97.3	0.08
Lifelog	5970	40	153	695	0.07	0.7	93.7	1.0

Table 2.1: Results on real data. We give data sequence length, alphabet size, number of interesting events in S_y, and the number of reported patterns for SQS + OMEN and SCIS. For OMEN we additional report compression rate relative to the null model in percent, $\%L$ and runtime in seconds. For *Lifelog* we report the average over 41 independent runs, with different target events.

sequence to 3 symbols. The heartbeats are annotated in the data. As we do not permit instantaneous predictions, we shift the annotation slightly forward such that they are strictly before the heartbeat.

When we run it on this data, SQS discovers 12 patterns out of which only one is predictive: it corresponds to the previous heartbeat. SCIS requires a window length, which we set to the approximate length of one cardiac cycle, excluding the heartbeat ($w = 40$), for which it then returns 41 318 patterns. OMEN needs 1.9 seconds to discover two predictive patterns that together compress S_y to only 65% of the number of bits needed by the null model. The first pattern corresponds to the previous heartbeat, the discovered time delay distribution exhibits structure of a bimodal normal distribution. We visualize this pattern in Figure 2.15. Closer inspection of the data shows that the data is indeed composed of two different modes, high and low intensity exercises.

Next we consider *Lifelog*, which is based on the life of *Sacha Chua* who logs and publishes all her daily activities.[6] We considered the data over 2017, removing any activities with have the same start and stop timestamp. As this dataset provides many events that are potentially interesting, we consider every $e \in \Omega$ as target, and have 40 target sequences with $S_y[i] = 1$ iff $S_x[i] = e$. In addition, we consider a S_y where we marked all business related activities as interesting.

6 http://quantifiedawesome.com/records

Over all these targets, SCIS discovers on average 695 patterns, many of which are redundant and not all make intuitive sense. While SQS only discovers 3 predictive patterns, these do make sense: *Cook, Dinner→Clean the Kitchen* and *Subway, Social→Subway*. OMEN takes between 0.005 and 5.9 seconds per dataset, and overall discovers 32 patterns. Many of these, such as *Sleep→Childcare*, *Cook→Dinner*, *Dinner→Clean the Kitchen*, predict the next action, i.e. a time delay distribution with a peak at 1. A more interesting pattern is *Subway→Subway* which has its peak at $\delta = 2$, and for which a natural interpretation is that *Sacha* takes the subway, logs on average one activity, and then takes the subway back.

Finally, we consider the *Saar* dataset [17], where the goal is to use daily precipitation records[7] to explain the rise (*Saar-Rise*) or fall (*Saar-Fall*) of the Saar river[8] by 10cm or more over one day. We considered the timespan from 2007 to 2018. We discretize the values to 17 symbols using $\lceil \log_{1.25} 1 + x \rceil$ where we accumulate all values $\geqslant 15$ into one symbol.

With SCIS ($w = 10$) we discover more than 400 patterns from either dataset. Most make little to no sense, such as that two successive days without rain predict a *rise* in water level. SQS does not discover any descriptive nor predictive patterns. For both datasets OMEN terminates within 0.08 seconds, and only reports singleton patterns. It finds, for example, that the more it rains, the more likely it is for the Saar to have risen by 10cm or more, either by the next day, or even two days later. For *Saar-Fall* we find an interesting pattern that expresses that approximately three days after heavy rain the water levels quickly drop—which indeed is likely as the water levels first rose due to rain.

2.7 DISCUSSION

The experiments showed that both OMEN and FOMEN work well in practice. We saw that the OMEN score is very good at telling predictive from spurious patterns, and that the mining algorithms are able to reconstructs the ground truth without picking up spurious or redundant patterns. In experiments on synthetic data we observed that

7 https://www.dwd.de/DE/leistungen/klimadatendeutschland/klarchivtagmonat.html (Ensheim weather station)

8 Measured at Sankt Arnual by the German Federal Institute of Hydrology (BfG)

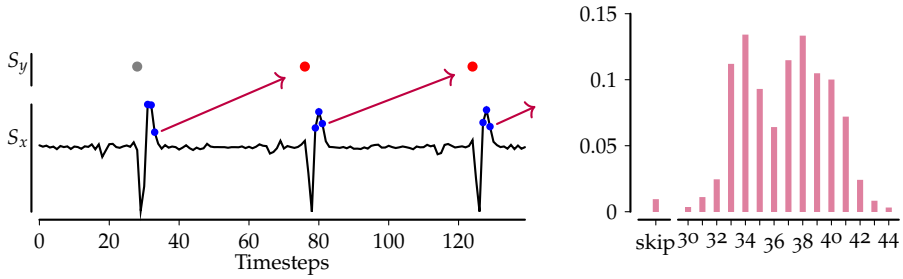


Figure 2.15: Window of ECG record with pattern *ccc* overlayed (left) and the reported time delay distribution (right).

it is highly robust against noise, can handle patterns with large gaps and high time delays, whereas the state of the art fails to report meaningful patterns. On real world data we showed that it discovers easily interpretable patterns that reliably explain our target events. The results of the *ECG* experiment demonstrate that OMEN finds interesting patterns and time delay distribution that represent the real world data well. In summary, on all considered settings it discovers small, easily interpretable and non-redundant sets of reliable patterns that together predict the interesting events well.

We do observe that when we planting particularly long patterns with high occurrences, OMEN tends to report partial rather than complete patterns. That is, if we plant pattern *abcdef* OMEN is likely to report *abcd* whenever that is already sufficient to accurately predict all interesting events that *abcdef* matches. It is easy to see why this is the case, as our score is only concerned with describing S_y as succinctly as possible, and hence does not provide any incentive to extend patterns further than strictly necessary. As in certain cases it is important to retrieve the entire predictive pattern, we plan to extend OMEN to report longer patterns when possible. One possible approach, to this end, is to additional encode S_x along with S_y creating an incentive for longer patterns.

Although OMEN is effective at discovering meaningful patterns, it currently considers a relatively simple pattern language. At the expense of additional computation, it is straightforward to extend our score and search to sequences with complex multivariate patterns [10]. In a similar line of thought, we currently model time delay distribu-

tions non-parametrically. While this comes with the advantage of not being restricted to any specific distribution, it does increase the sample complexity of our method—we may miss certain patterns or fail to predict certain events, simply because we have too little data to model the delay distribution well. It hence makes sense to contemplate an extension of OMEN where we model the delay distributions parametrically, e.g. using domain knowledge to choose it as a Gaussian or Poisson distribution, as this will permit discovering more subtle patterns. In the next chapter we explore modeling delays between events of patterns, where we model the delay distributions parametrically.

We consider mining predictive rather than causal patterns. We do note, however, the close kinship between the two, in the sense that the discovered pattern could cause our interesting event. We share at least one common assumption: a cause needs to precede the effect in time [69, 138]. In Chapter 7 we explore the causal relationships between events in event sequences.

2.8 CONCLUSION

We considered the problem of discovering small sets of sequential patterns that not only predict that something interesting will happen, but for which it is additionally easy to tell how long it will be until the predicted event. We formulated the problem in information-theoretic terms using the Minimum Description Length principle. As the resulting problem does not lend itself to efficient exact optimization, we propose the OMEN and FOMEN to heuristically discover good pattern sets. Both rely on a method to infer a initial alignment between pattern occurrences and interesting events. For which we propose two algorithms one general purpose one and an alternative approach which is particular adapt at long range predictions. Extensive evaluation on synthetic and real world data showed that OMEN and FOMEN compares favorably to the state of the art. In particular, our score performs very well in telling predictive from associative patterns, even under large quantities of noise. OMEN efficiently discovers high quality sets of predictive patterns give clear insight into the data generating process.

3

DISCOVERING SEQUENTIAL PATTERNS WITH PREDICTABLE INTER-EVENT DELAYS

In the previous chapter, we explored how to discover predictive patterns for target events, focusing on whether and when an event will occur. To this end we explicitly modeled the delay between predictive patterns and the target events.

In this chapter we consider a setting without a target sequence, and instead of discovering patterns that predict when target events will occur, we consider the task of discovering patterns with predictable inter-event delays. Specifically, we will discuss how to summarize event sequences using serial episodes. Unlike existing approaches, our method explicitly models the delay between events, prioritizing patterns with consistent delays; this allows us to identify patterns with long delays, as long as the delays are consistent.

3.1 INTRODUCTION

Most existing methods, summarize event sequences, in terms of serial episodes and allow for gaps [170] and interleaving [11] 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 [57] 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

This chapter is based on [32]: Joscha Cüppers, Paul Krieger, and Jilles Vreeken. "Discovering Sequential Patterns with Predictable Inter-event Delays." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. 2024, pp. 8346–8353.

hence 'move' around. Existing methods have no trouble finding holidays that occur right after each another, e.g. 1^{st} *Christmas Day* right before 2^{nd} *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 chapter, 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) [71], 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.[1]

3.2 PRELIMINARIES

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

3.2.1 *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 $\phi_{p,i}(\cdot \mid \Theta_{p,i})$. Whenever clear from context we simply write $\phi_{p,i}(\cdot)$.

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 $\forall_{i \in [1,|p|-1]} \phi_{p,i}(w^S[i+1] - w^S[i]) > 0$, 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 $0 \log(0) = 0$.

3.3 MDL FOR PATTERNS WITH PREDICTABLE DELAYS

In this section we formally define the problem using the Minimum Description Length principle. We first give the intuition by explaining how to decode a sequence from a given encoding and then formally define the encoding.

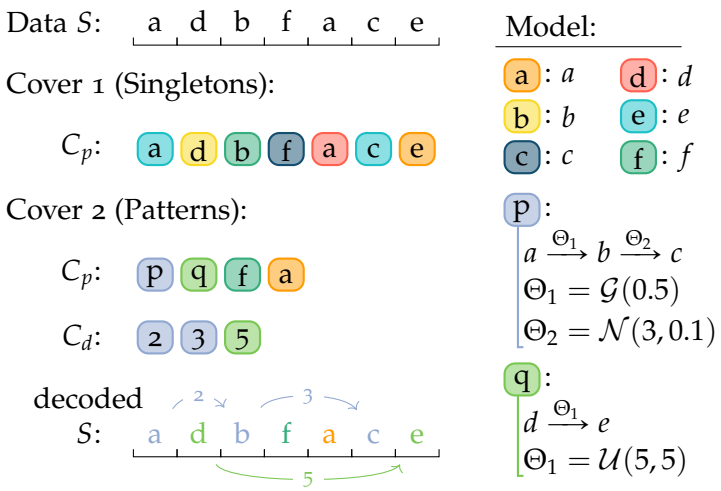


Figure 3.1: Toy example showing two possible encodings of the same data. Cover 1 uses only singletons, Cover 2 additionally uses two patterns, (p) and (q). A cover consist of the pattern stream C_p encoding the patterns, and the delay stream C_d encoding the inter-event delays. The first gap of pattern (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.

3.3.1 *Decoding the Database*

We start 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 3.1 we show a toy example. We show a sequence S, a model M, and two covers of S using M.

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

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

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.

3.3.2 *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 cardinalities using the MDL-optimal encoding for integers [153].

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 (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 $\phi_{p,i}(d_j)$. We hence have

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

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 [176]. 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 $\phi(\cdot \mid \Theta)$ 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 num-

ber of bits needed to encode a real number up to user-set precision p [114]. It does so by shifting θ by d digits, such that $\theta \cdot 10^d \geqslant 10^p$.

3.3.3 *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 [11]. 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.

3.4 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.

3.4.1 *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

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

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

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

3.4.2 *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,

$$\Delta\overline{L}(M \oplus p') = -\hat{L}(p') + L(\arg\min_{p \in \{p_1, p_2\}} usg(p))$$

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 3: OPTIMIZEALIGNMENT

Input : 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 [Chapter 2, Section 2.4.4] 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 us 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 $\phi \in \Psi$ and choose that $\phi^*_{p'}(\cdot \mid \Theta^*)$ 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

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

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 3.

Algorithm 4: HOPPER

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

1 $CT \leftarrow \Omega$; $Cand \leftarrow CT \times CT$;
2 **forall** $p_1, p_2 \in Cand$ **do** ordered descending on $|p_1|usg(p_1) + |p_2|usg(p_2)$
3 **if** $\Delta\overline{L}(p_1 \oplus p_2) > 0$ **then**
4 $gain, p' \leftarrow$ ALIGNCANDIDATE(p_1, p_2)
5 **if** $gain > 0 \wedge L(D, M) > L(D, M \oplus p')$ **then**
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

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 4. 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 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 Appendix b.1. In a nutshell, it returns the best optimized result out of ALIGNNEXT and 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 [12, 71] in compression. We provide further details on the pattern mining procedure in Appendix b.1.

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.

3.5 RELATED WORK

In Chapter 2 Section 2.5 we already discussed the related work on pattern mining on event sequences. Here we discuss how they handle gaps. Ism [57] allow for gaps, but do not model them, that is neither penalize nor prefer pattern with consistent gaps. Sqs [170] and Squish [11] 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 [29, 41, 63, 200], and methods that discovers delay information from the data [126, 198]. 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. [58] 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 [Chapter 2] 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.

3.6 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.[2] We compare HOPPER to SKOPUS [144] as a representative statistically significant sequential pattern miner, SQS [170], SQUISH [11] and ISM [57] as representatives of the general class of pattern set miners, and to PPM [58] 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. 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 give the full setup description in Appendix b.2.

2 eda.rg.cispa.io/prj/hopper

3.6.1 *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. To compute the number of true positives we follow the same flow network based approach as in Chapter 2 Section 2.6.2, where we set the flow capacity between reported pattern p_r and planted pattern p_P based on the Levenshtein edit distance, that is,

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

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.

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. 3.2. 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 Appendix b.2.

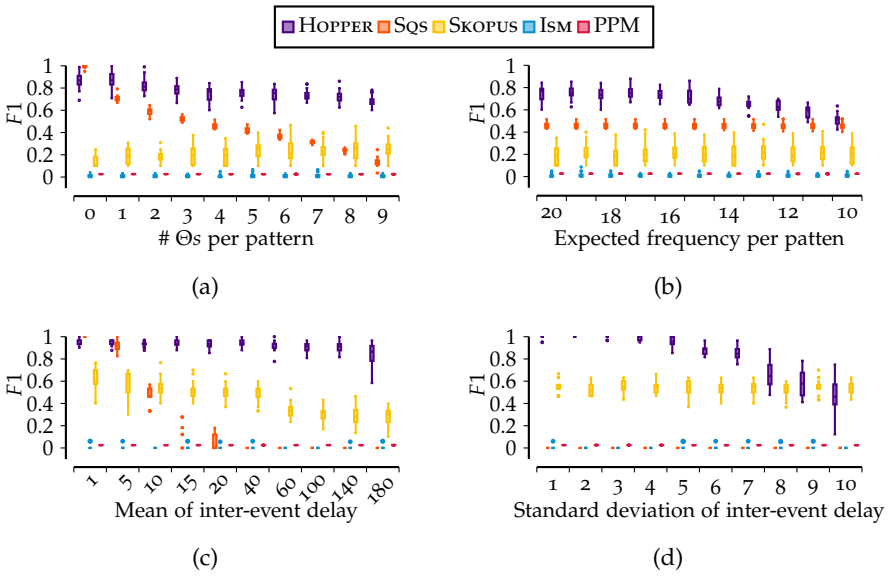


Figure 3.2: [Higher is better] F1 scores for recovering patterns from synthetic data. From (a) to (d), we evaluate for varying numbers of inter-event distributions, expected frequency of a pattern, mean inter-event delay, resp. different standard deviations for normally distributed delays. We see that 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.

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 Panel (b) of Fig. 3.2. 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 Panel (c) of Fig. 3.2. 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 Panel (d) of Fig. 3.2.

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.

3.6.2 *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*[3] of all activities of one person recorded in over seven years, the MIDI data of hundred Bach *Chorales* [92], all commits to the *Samba* project for over ten years [58], the *Rolling Mill* production log of a steel manufacturing plant [187], the discretized muscle activations of professional ice *Skating* riders [123], and finally, three text datasets from 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 3.1 and further statistics in Appendix b.2.

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

3 https://quantifiedawesome.com/

			HOPPER			SQS		PPM	
Dataset	$\lvert\Omega\rvert$	$\lvert\lvert D\rvert\rvert$	$\lvert P\rvert$	$\#\Theta$	$\mathbb{E}(w)$	$\lvert P\rvert$	$\mathbb{E}(w)$	$\lvert P\rvert$	$\mathbb{E}(w)$
Holidays	37k	11	1	7	393	3	19.2	14	51.6
Radio	16k	494	22	43	48	15	5.8	587	71.9
Lifelog	40k	77	37	68	129	58	3.9	1.6k	119.1
Samba	29k	118	40	101	110	221	2.7	1.4k	17.1
Chorales	7k	493	56	57	4.7	114	2.6	433	2.6
Rolling	54k	555	237	489	7.4	470	5.0	3.6k	181.9
Skating	26k	82	86	160	9.1	160	4.0	1.4k	55.5
Romeo	37k	4789	254	284	12.9	254	2.8	2.3k	332.7
Room	87k	9009	565	610	3.1	701	2.5	–	–
Gatsby	64k	7463	439	488	7.3	519	2.6	4.7k	641.9

Table 3.1: Results on real-world data. For HOPPER, SQS, and PPM we report the number of discovered patterns ($\lvert P\rvert$) 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 ($\#\Theta$).

number of discovered delay distributions ($\#\Theta$). We postpone the results of ISM and SKOPUS, along with the metrics runtime and average events per pattern to Appendix b.2. 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* $1^{st} \xrightarrow{155}$ *National Holiday* $\xrightarrow{83}$ 1^{st} *Christmas Day* $\xrightarrow{1}$ 2^{nd} *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 1^{st} *Christmas Day*, 2^{nd} *Christmas Day*. We show the results for all methods in Appendix b.2.

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 $\textit{Jingle} \xrightarrow{0} \textit{Ads} \xrightarrow{0} \textit{News} \xrightarrow{0} \textit{Jingle} \xrightarrow{\mathcal{U}(3,5)} \textit{Jingle}$ 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 $\textit{Solo Para} \xrightarrow{\mathcal{G}(0.02)} \textit{As It Was} \xrightarrow{\mathcal{N}(48,25)} \textit{I Believe} \xrightarrow{\mathcal{P}(24)} \textit{Anyone for You}$ 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.

3.7 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. 3.2, 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. Next, we will study how to summarize event sequences using conditional dependencies — in the form or rules.

4

MINING RULE-SETS FROM EVENT SEQUENCES

In the previous chapter we summarized event sequences in terms of serial episodes. In this chapter we will study how to summarize event sequences in terms of conditional dependencies. We do so by discovering rules of the form $X \rightarrow Y$, where X and Y are sequential patterns, expressing that Y is more likely to occur after we have observed X. Rules like these are simple to understand and provide a clear description of the relation between the antecedent and the consequent.

4.1 INTRODUCTION

In many applications data naturally takes the form of events happening over time. Examples include industrial production logs, the financial market, device failures in a network, etc. Existing methods for analyzing event sequences primarily focus on mining unconditional, frequent sequential patterns [4, 111, 168]. Loosely speaking, these are subsequences that appear more often in the data than we would expect. Real world processes are often more complex than this, as they often include conditional dependencies. The formation of tropical cyclones (C) in the Bay of Bengal, for example, is often but not always followed by heavy rainfall (R) on the coast. Knowing such a relationship is helpful both in predicting events and in understanding the underlying data generating mechanisms.

In this chapter, we are interested in discovering rules of the form $X \rightarrow Y$ from long event sequences, where X and Y are sequential patterns. Existing methods for mining such rules either suffer from the pattern explosion, i.e. are prone to returning orders of magnitude more results than we can possibly analyze [22, 52], or are strongly

This chapter is based on [162]: Aleena Siji, Joscha Cüppers, Osman Ali Mian, and Jilles Vreeken. "Seqret: Mining Rule Sets from Event Sequences." In: *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI ,2026.

limited in the expressivity, e.g. require the constituent events to occur in a contiguous order [14].

We aim to discover succinct sets of rules that generalize the data well. We explicitly allow for gaps between the head and the tail of the rule, as well as in the occurrences of X and Y themselves. To ensure we obtain compact and non-redundant results, we formalize the problem using the Minimum Description Length (MDL) principle [71]. Loosely speaking, we are after that set of sequential rules that together compresses the data best.

However, the problem we so arrive at is computationally challenging. For starters, there exist exponentially many rules, exponentially many rule sets, and then again exponentially many ways to describe the data given a set of rules. Moreover, the search space does not exhibit structure we can use to efficiently obtain the optimal result. To mine good rule sets from data we therefore propose the greedy SEQRET algorithm. We introduce two variants. SEQRET-CANDIDATES constructs a good rule set from a set of candidate patterns by splitting them into high-quality rules. SEQRET-MINE, on the other hand, only requires the data and mines a good rule set from scratch. Starting from a model of singleton rules, it iteratively extends them into more refined rules. To avoid testing all possible extensions, we consider only those extensions that occur significantly more often than expected.

Through extensive evaluation, we show that both variants of SEQRET work well in practice. On synthetic data we show that they are robust to noise and recover the ground truth well. On real-world data, we show that SEQRET returns succinct sets of rules that give clear insight into the data generating process. This in stark contrast to existing methods which either return many thousands of rules [52] or are restricted to rules where events occur contiguously [14]. We make all code and data available online.[1]

4.2 PRELIMINARIES

In this section we introduce basic notation.

1 https://eda.rg.cispa.io/prj/seqret/

4.2.1 *Notation*

As data we consider a sequence database D of $|D|$ event sequences. A sequence $S \in D$ consists of $|S|$ events drawn from a finite alphabet Ω of discrete events $e \in \Omega$. We denote the total number of events in the data as $||D||$. We write S_t for the t^{th} sequence in D. To avoid clutter, we omit the subscript whenever clear from context. We write $S[i]$ to refer to the i^{th} event in sequence S, and $S[i,j]$ for the subsequence from the i^{th} up to and including the j^{th} event of S. We denote an empty sequence by ϵ.

A serial episode X is a sequence of $|X|$ events drawn from Ω. A sequential rule r captures the conditional dependence between a serial episode X and a serial episode Y. Intuitively, it expresses that whenever we see X in the data it is more likely that Y will follow soon. We refer to X as the *head* or antecedent of r, denoted $head(r)$, and to Y as the *tail* or consequent of r, denoted $tail(r)$. If X is an empty pattern, $X = \epsilon$, we call $X \rightarrow Y$ an *empty head rule*. We refer to empty head rules where $|Y| = 1$ as a *singleton* rule.

A subsequence $S[i,j]$ is a window of pattern X iff X is a subsequence of $S[i,j]$, and subsequently we say $S[i,j]$ *matches* X and vice-versa we say that X occurs in $S[i,j]$. A pattern window $S[i,j]$ is *minimal* for X iff no proper sub-window of $S[i,j]$ matches X. A window of a rule r is a tuple of two pattern windows $S[i,j]$ and $S[k,l]$ when $S[i,j]$ matches $head(r)$, $j < k$, and $S[k,l]$ matches $tail(r)$. We denote a rule window by $S[i,j;k,l]$.

We say a window $S[i,j]$ *triggers* rule r when it is a minimal window of $head(r)$. A rule window $S[i,j;k,l]$ *supports* a rule r if $S[i,j]$ triggers $head(r)$ and $S[k,l]$ matches $tail(r)$. We call the number of events that occur in a rule window between the rule head and the rule tail, $k - j - 1$, the *delay* of the rule instance. We give an example in Fig. 4.1. We denote the number of windows over all sequences $S \in D$ that trigger a rule r as the trigger count $trigs(r)$. We define the *support* of a rule r as the number of rule windows $S[i,j;k,l]$ in D where $S[k,l]$ is a minimal window of $tail(r)$ and follows the head with minimum delay. Finally, we define the confidence of a rule r as its support relative to its trigger count, formally $conf(r) = supp(r)/trigs(r)$.

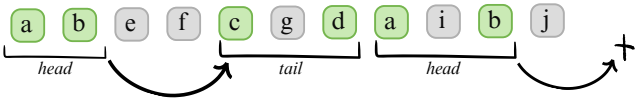


Figure 4.1: Toy example of a rule $ab \rightarrow cd$ in an event sequence. Each occurrence of head ab triggers the rule. The first is followed by tail cd and hence a 'hit' whereas the second is not and hence a 'miss'. $supp(ab \rightarrow cd) = 1$ and $conf(ab \rightarrow cd) = 0.5$.

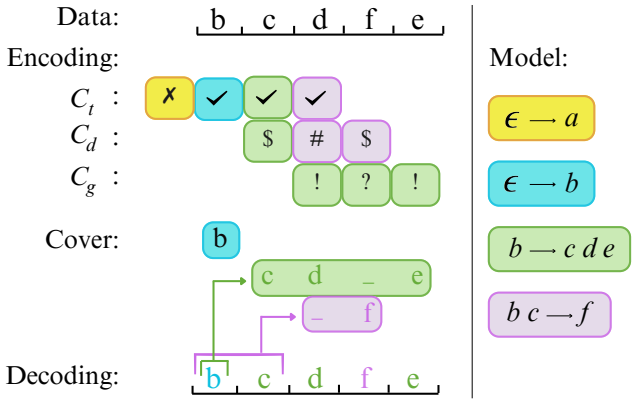


Figure 4.2: Toy example showing an encoding of sequence S using rule set R. The encoding consists of three code streams. C_t encodes if a triggered rule *hits* or *misses*. C_d encodes the delay between the trigger and the rule tail. C_g encodes the gaps in the tails. Together, they form a cover C of D given rule set R.

4.3 MDL FOR SEQUENTIAL RULES

We now formally define the problem we aim to solve. We consider sets R of sequential rules as our model class $\mathcal{R}$. By MDL, we are interested in that set of rules $R \in \mathcal{R}$ that best describes data D.

4.3.1 *Decoding an Event Sequence*

Before we formally define how we *encode* models and data, we give the main intuition of our score by *decoding* an already encoded sequence. We give an example in Fig. 4.2.

To decode a symbol, we consider the rules from R that are currently triggered. For those, we read codes from the trigger stream C_t. Initially, the context is empty and hence only empty-head rules trigger. The first trigger code is a *miss* for singleton rule $\epsilon \to a$. The second trigger code is a *hit* for rule $\epsilon \to b$. As empty-head rules do not incur delays, we can write b as the first symbol of the sequence.

This triggers rule $b \to cde$. We hence read a code from the trigger stream, and find that it is a hit. As this rule does not have an empty head, there may be a delay between the head and its tail and we read a code from the delay steam C_d to determine if this is the case. It is a start code, so we write the first symbol of the tail (c).

This creates a minimal window of bc and hence rule $bc \to f$ triggers. We read from C_t to find that it hits, and from C_d to find that its tail is delayed. To determine if we may write the next symbol from tail cde, we read from the gap stream C_g. This is a fill code, meaning there is no gap, and hence we write d.

This time, no new rule triggers. Tail cde is not yet completely decoded and f is delayed. For each delayed tail we read a code from C_d, and for each incomplete tail we read a code from C_g. Here, we read a start code for tail f and a gap code for tail cde, we hence write f. Again, no new rule is triggered. Now only tail cde is not yet fully decoded. We read from C_g and as it is a fill code we write e as the last symbol of the sequence.

To summarize, sequences are encoded from left to right, and rules automatically trigger whenever we observe a minimal window of the head. For each trigger, we encode whether the tail follows using a *hit* or *miss* code. When a rule hits, we encode whether its tail follows immediately or later, using a *start* resp. *delay* code. Finally, we encode whether *gaps* occur in the rule tail using *fill* and *gap* codes. Empty-head rules never incur a delay. To avoid unnecessary triggers, we only encode those of empty-head rules if no other rule encodes the current symbol (e.g. all active tails say 'gap').

4.3.2 *Computing the Description Lengths*

Now that we have the intuition, we can formally describe how to encode a model, respectively the data given a model.

ENCODING A MODEL A model $R \in \mathcal{R}$ is a set of rules. To reward structure between rules, e.g. chains where the tail of one rule is the head of another (e.g. $r_1 = \epsilon \rightarrow AB$, and $r_2 = AB \rightarrow CD$), we first encode the set P of all non-empty and non-singleton heads and all non-singleton tails. Formally,

$$P = \{head(r) \mid \forall r \in R\} \cup \{tail(r) \mid \forall r \in R\} \setminus (\Omega \cup \epsilon) \ .$$

The encoded length $L(P)$, is defined as

$$L(P) = L_{\mathbb{N}}(|P| + 1) + \sum_{p \in P} L_{\mathbb{N}}(|p|) + |p| \log_2(|\Omega|) \ ,$$

where we first encode the number of these patterns using $L_{\mathbb{N}}$ the MDL-optimal encoding for integers [153]. Since P can be empty and $L_{\mathbb{N}}$ is only defined for numbers $\geqslant 1$ we offset it by one. Next, we encode each pattern $p \in P$ where we use $L_{\mathbb{N}}$ to encode its length and then choose each subsequent symbol $e \in p$ out of alphabet Ω.

Now that we have the set of all heads and tails, we have

$$L(R \mid P) = L_{\mathbb{N}}(|R| + 1) + |R|(\log_2(|P| + |\Omega| + 1) + \log_2(|P| + |\Omega|)) \ ,$$

as the encoded length in bits of a set of rules. We first encode the number of rules, and as R can be empty, we again offset by one. Next, for each rule $r \in R$, we choose its head from $P \cup \Omega$, and then its tail from $P \cup \Omega$.

Putting this together, the number of bits to describe a rule set $R \in \mathcal{R}$ without loss is,

$$L(R) = L(P) + L(R \mid P) \ .$$

ENCODING DATA GIVEN A MODEL As we saw in the example, to reconstruct the data we need the three code streams C_t, C_d, and C_g. For an arbitrary database we additionally need to know how many sequences it includes, and how long these are. Formally, the description length of data D given a model R hence is

$$L(D \mid R) = L_{\mathbb{N}}(|D|) + \left(\sum_{S \in D} L_{\mathbb{N}}(|S|) \right) + L(C_t) + L(C_d) + L(C_g) \ . \quad (4.1)$$

To encode the code streams C_t, C_d, C_g we use prequential codes [71]. Prequential codes work by assuming an initial usage of $c = 0.5$ for

all possible codes, and updating these counts with every transmitted (resp. received) code. This way we not only ensure that we always have a valid coding distribution, but also achieve asymptotic optimality *without* having to transmit the counts beforehand [71]. Formally, we have

$$L(C_j) = \sum_{i=1}^{|C_j|} \log_2 \frac{usg_i(C_j[i] \mid C_j) + c}{i + unique(C_j) \cdot c} \quad ,$$

where $usg_i(C_j[i] \mid C_j)$ denotes the number of times $C_j[i]$ has been used in C_j up to the i^{th} position, $unique(C_j)$ denotes the number of unique symbols in C_j.

4.3.3 *The Problem, Formally*

We can now formalize the problem we aim to solve.

The Sequential Rule Set Mining Problem *Given a sequence database D over alphabet Ω, find the smallest rule set $R \in \mathcal{R}$ and cover C such that the total encoded size*

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

is minimal.

The search space of this problem is enormous. To begin with, there exist super-exponentially many covers of D given R. The optimal cover depends on the code lengths, which in turn depend on the code usages. Even if the optimal cover is given, the problem of finding the optimal rule set is super-exponential: there exist exponentially many patterns p in the size of the alphabet Ω, exponentially many rules r in the number of patterns, and exponentially many sets of rules. None of these sub-problems exhibit substructure, e.g. monotonicity or submodularity, that we can exploit to efficiently find the optimal solution. Hence, we resort to heuristics.

4.4 THE SEQRET ALGORITHM

In this section we introduce our method, SEQRET, for discovering high-quality **seq**uential **r**ule-s**et**s from data. We break the problem down

into two parts: optimizing the description of the data given a rule set, and mining good rule sets. For the latter we propose SEQRET-CANDIDATES for doing so given a set of candidate patterns, and SEQRET-MINE for mining rule sets directly from data.

4.4.1 *Selecting a Good Cover*

A lossless description of D using rules R correspond to a set of rule windows such that each event e in D is covered by exactly one window. We are after that *cover* that minimizes $L(D \mid R)$. Finding the optimal cover is infeasible, and hence we instead settle for a good cover and show how to find one greedily.

The main idea is to define an order over the rule windows and greedily select the next best window until the data is completely covered. To minimize the encoded length, we prefer to cover as many events as possible with a single rule with few gaps. Therefore, we prefer using rules with long tails, high confidence, and high support. Similarly, among windows of otherwise equally good rules, we prefer those with lower delays and fewer gaps in the rule tail. As a final tie breaker, we consider the starting position of the rule tail. Combining this, we define the WINDOW ORDER as descending on $|tail(r)|$, $conf(r)$, and $supp(r)$, and finally ascending on $l - j - |tail(r)|$, and k, where r is a rule, $S[i, j; k, l]$ is a rule window. To avoid searching for all possible rule windows, we start with the best window per rule trigger and look for the next best only if we do not select the former due to conflicts, i.e. its constituent events are already covered by a previously selected window. We define the best rule window per trigger as the one with the fewest gaps in its rule tail window, and among those with same gap count, the one with the lowest delay.

We give the pseudocode of COVER as Algorithm 5. We start by initializing cover C with the empty set and window set W with for each rule the best rule windows per trigger (lines 1-2). We then greedily add rule windows to C in order of WINDOW ORDER. If a window conflicts with an already selected window (line 4), we skip it and search for the next best rule window for the corresponding trigger and add it to W (line 7). We continue this process until all events in D are covered. To avoid evaluating hopeless windows, we limit ourselves to those within a user-set *max delay* ratio and *max gap* ratio. We provide further details

and pseudo code for BestRuleWin and NextBestWin procedures in Appendix. c.1.1.

The worst case time complexity of Cover depends on the number of rules in R, total number of events in D, and the lengths of the heads and tails per rule. In Appendix. c.2.2 we show the complexity of Cover is $\mathcal{O}(|R| * ||D||(h + t^3 + t \log_2(|R| \cdot ||D||t))$, where h is the max head length and t the max tail length.

Next, we consider the problem of discovering good rule sets.

Algorithm 5: Cover

Input: Sequence database D, rule set R
Output: Cover C

while $\exists S_t \in D$ where, $\exists e \in S_t$ not covered by C **do**
  $w \leftarrow$ next $w \in W$ in window order;
  $W \leftarrow W \backslash \{w\}$;
  if $\nexists z \in C$ that **conflicts** with w **then**
    $C \leftarrow C \cup \{w\}$;
  else
    $W \leftarrow W \cup \{\text{NextBestWin}(w, C, D)\}$;
return C

4.4.2 *Selecting Good Rule Sets*

We first propose an approach that does so given a set of sequential patterns as input. We start from the intuition that, if the ground truth

Algorithm 6: Seqret-Candidates

Input: Sequence database D, set of patterns F
Output: Rule set R

$R \leftarrow \{\epsilon \rightarrow e \mid \forall e \in \Omega\}$
for $p \in F$ ordered descending by $L(D, F \backslash \{p\}) - L(D, F)$ **do**
  $r \leftarrow \arg\max_{r' \in \text{Split}(p)} L(D, R) - L(D, R \cup \{r'\})$
  if $L(D, R \cup \{r\}) < L(D, R)$ **then**
    $R \leftarrow M \cup \{r\}$;
return R;

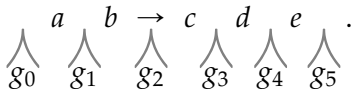
includes a sequential rule $a \rightarrow bc$, a good sequential pattern miner will return abc. This means we can reconstruct ground truth rules by considering splits of candidate patterns XY into candidate rules $X \rightarrow Y$ and using our score to select the best split.

To this end, we propose SEQRET-CANDIDATES, for which we give the pseudocode as Algorithm 6. We initialize the rule set with all singleton rules (line 1) to ensure we can encode the data without loss. We then iterate over each candidate pattern (line 2) in descending order of contribution to compression [170]. We split each pattern into candidate rules – for example, pattern abc generates candidate rules $\epsilon \rightarrow abc$,$a \rightarrow bc$, and $ab \rightarrow c$ – and choose the candidate rule that minimizes our score (line 3). We add it to our model if it improves the score (line 5) and iterate until all patterns are considered.

The run time is dominated by the number of cover computations, i.e. how many times we have to compute $L(D|R)$. We have to compute a new cover for each rule we test, and each pattern p can be split into $|p|$ rules. We test each $p \in F$ as such the complexity of SEQRET-CANDIDATES is $\mathcal{O}(|F|(\max_{p \in F} |p|))$.

4.4.3 *Generating Good Rules*

Next, we move our attention to generating good rule sets directly from data. The first step is to generate good candidate rules. Given a rule r from the current model, we consider extending it with events $e \in \Omega$ that occur significantly more often within or directly adjacent to the rule windows of r. A rule has $|r| + 1$ such gap positions, i.e. before its first event, between its constituent events, and after its last event. For example, rule $ab \rightarrow cde$ has 6 gap positions,



For each rule r we test for every gap position g_i if $e \in \Omega$ is more frequent than expected in its rule windows. Our null hypothesis is

$$H_0 : \sum_{w \in B} \mathbb{1}(e \in g(i, w)) \leqslant \sum_{w \in B} \Pr(e \in g(i, w))$$

where B is the set of best rule windows of r, $B = \text{BESTRULEWIN}(r, D)$, $w \in B$ is a window of rule r, and $g(i, w)$ a function that returns gap i from window w.

When computing the probability of an event e in gap g_i, we have to account for differences in lengths of gaps between different windows. The probability of e occurring in gap g_i of a rule window w_p is given by

$$\Pr(e \in g(i, w_p)) = 1 - \left(1 - \frac{supp(\epsilon \rightarrow e)}{|D|}\right)^{|g(i,w_p)|} .$$

To test for statistical significance, we can model the expected neighborhood as a Poisson binomial distribution [184]. That is, the trials are the rule windows, and the success probability per trial is decided by the length of the gap at the position of interest. Computing the CDF of the Poisson binomial distribution is expensive [20, 79, 178]. As a fast approximation, we use the normal approximation with continuity correction [79] for cases where the number of trials, i.e $supp(r)$, is greater than 10. If less than or equal to 10, we simply check if the actual count of occurrences is greater than the expected count by more than one.

If event e is measured to be significantly more frequent in g_i than expected, we generate a new rule by inserting e at the position of g_i in the rule. We give the pseudo-code in Appendix. c.1.3.

4.4.4 *Mining Good Rule Sets*

Finally, we describe SEQRET-MINE for mining good rule sets directly from data. We provide the pseudocode as Algorithm 7. We initialize rule set R with all the singleton rules (line 1). Next, we build candidates from rule set R (line 3). As we want to generate the most promising candidate rules first, we start with rules with high support and high confidence. We define a greedy EXTEND ORDER as 1) $\uparrow supp(r)$, 2) $\uparrow conf(r)$, 3) $\uparrow |tail(r)|$ and 4) $\uparrow |head(r)|$, where $\uparrow$ indicates that higher values are preferred. For each we generate a set of candidate rules as described above, we test them for addition in the order of their p-values (line 4).

We add those rules into the model whose inclusion results in a significant reduction in the total encoded size (line 5). We use the no-hypercompression inequality [12, 71] to test for significance at level α,

writing $\ll_\alpha$ for "significantly less".[2] In case adding a candidate rule r' to the model does not improve compression, we test if replacing rule r with r' leads to compression (line 7 and 8). To ensure we can always describe the data without loss, we never remove singleton rules.

After adding a new rule, SEQRET-MINE performs a pruning step to remove existing rules that may have become redundant or obsolete (line 10). The PRUNE method iterates over the non-singleton rules in the model and removes those whose exclusion reduces the total encoded size. We do so in PRUNE ORDER where we consider rules in order of lowest usage, highest encoded size, and lowest tail length.

We repeat generating candidate rules, adding them, and pruning redundant rules until convergence. Convergence is guaranteed as our score is lower bounded by 0. The worst case time complexity of one iteration of SEQRET-MINE is, $\mathcal{O}(|R||\Omega|(h+t))$. We provide the full derivation in Appendix. C.2.2.

Algorithm 7: SEQRET-MINE

Input: Sequence database D over Ω, significance level α
Output: Rule set R

1 $R \leftarrow \{\epsilon \rightarrow e \mid \forall e \in \Omega\}$;
2 **do**
3 **for** $r \in R$ in EXTEND ORDER **do**
4 **for** $r' \in$ CANDRULES(D, r) in order of p-value **do**
5 **if** $L(D, R \cup \{r'\}) \ll_\alpha L(D, R)$ **then**
6 $R \leftarrow R \cup \{r'\}$;
7 **else if** $r \notin \{\epsilon \rightarrow e \mid \forall e \in \Omega\}$ **and** $L(D, R \cup \{r'\} \backslash \{r\}) \ll_\alpha L(D, R)$ **then**
8 $R \leftarrow (R \backslash \{r\}) \cup \{r'\}$;
9 **if** R updated **then**
10 $R \leftarrow$ PRUNE(D, R);
11 **continue** with next r
12 **while** R updated;
13 **return** R

2 In our experiments we set α to 0.05, which by the no-hypercompression inequality corresponds to a minimum gain of 5 bits.

4.5 RELATED WORK

Classical rule miners for event sequences operate similar to frequent pattern mining, but in addition to the frequency requirement also impose a minimum confidence threshold. Various approaches have been proposed to address different data modalities, such as rules over itemsets ordered by time [53, 54, 56], or rules over events in sequences [30, 39, 201]. The former generally count the number of sequences containing a rule as its support, whereas the latter use sliding or minimal windows to capture multiple rule occurrences within a sequence.

Rules can be further categorized into partially ordered rules [22, 52] where the rule tail follows the rule head but the constituent events of the rule head and the rule tail may appear in any order, and sequential rules where the order both between the rule head and the rule tail as well as within the rule head and the rule tail has to match [201]. Each of the above approaches consider the quality of individual patterns and hence suffer from the pattern explosion. To address this, Fournier-Viger and Tseng propose TNS [55], a method that reports the top-k non redundant rules, it uses a strict notion of redundancy and is not able to avoid semantically redundant rules.

Most closely related to our approach are existing methods that use MDL to select or mine rules. OMEN [Chapter 2] is a supervised method for mining 'predictive patterns'. It is not applicable in our setting as it requires a target. Existing rule set miners for event sequences either filter down an existing set of rules [21], or do not allow for gaps [14]. As such, none of the existing methods directly addresses the problem we consider.

4.6 EXPERIMENTS

In this section we empirically evaluate SEQRET-CANDIDATES and SEQRET-MINE. We implement both in Python and provide the source code, synthetic data generator, and real-world data online.[3]

We compare SEQRET to POERMA [52] and POERMH [22] as representatives of frequent rule mining, to TNS [55] as a top k non-redundant rule set miner, to COSSU [14] as an MDL-based rule set miner, and to SQS [170] and SQUISH [11] as MDL-based sequential pattern miners.

3 https://eda.rg.cispa.io/prj/seqret/

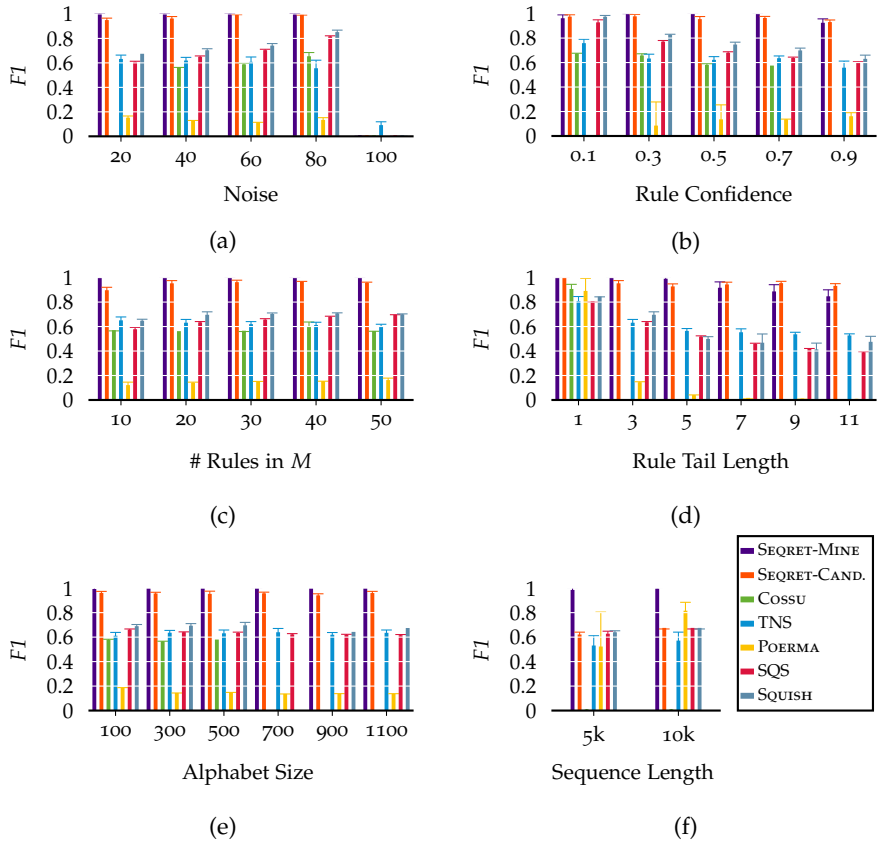


Figure 4.3: [Higher is better] $F1$ scores for synthetic data. We observe that SEQRET is robust against (a) high noise, (b) rule confidence, (c) number of true rules, (d) rule tail length, and (e) large alphabets. In (f) we evaluate rule recovery where heads and tails are only as frequent as by chance, SEQRET-MINE still picks up the ground truth.

As candidate patterns for SEQRET-CANDIDATES we use the output of SQS [170] because SQUISH crashes regularly. For TNS we set k to the number of rules, and for POERMA and POERMH, the minimum support and minimum confidence values according to the ground truth when known. In the case of real datasets where the ground truth is unknown, we set k for TNS as the number of rules returned by SEQRET-MINE. For POERMA and POERMH we use a minimum support threshold of 10 where feasible, and 20 otherwise. We allow all methods a maximum runtime of 24 hours. With the exception of COSSU, which we allow a maximum runtime of 48 hours, as it generally took longer to complete. We ran all experiments on an Intel Xeon Gold 6244 @ 3.6 GHz, with 256GB of RAM. We provide additional details in Appendix. c.3.1.

4.6.1 *Synthetic Data*

We first consider data with known ground truth. To this end, we generate synthetic data from a randomly generated rule set R. For a given alphabet size, number of rules, sizes of the heads and tails, and confidence, we generate rule heads and the rule tails by selecting events from alphabet Ω uniformly at random with replacement.

GENERATING DATA We generate event sequences S as follows. We first generate background noise by sampling uniformly at random from the alphabet. Next, we plant patterns, i.e. the empty-head rules from R. We sample uniformly from all empty-head rules in the model and write the tails to S at random positions while making sure we do not overwrite existing rules. Finally, we go over the generated sequence and wherever a non-empty-head rule is triggered, we sample according to the desired confidence of the rule whether the trigger is a hit or a miss. If it is a hit, we sample the delay and then insert the corresponding rule tail. For all tails we plant, we sample gap events. The probability of delay and gaps are set as input parameters. We provide additional details of synthetic data generation in Appendix. c.3.2.

Unless stated otherwise, we generate sequences of length 10 000 over alphabets of size 500, rule sets of size 20 with rule confidence 0.75. We generate 20 datasets per configuration in each experiment.

EVALUATION METRIC As evaluation metric, we consider the $F1$ score, we again follow the flow network based approach introduced in Chapter 2. We base the flow capacity on the similarity between the rules, and we compute the similarity using the Levenshtein edit distance without substitution, i.e the longest common subsequence distance [127]. To keep the similarities comparable between rules we normalize by the combined lengths. Formally, we have

$$sim\,(X, U) = 1 - lcsd(X, U)\,/\,(|X| + |U|)\,,$$

where $lcsd(X, U) = |X| + |U| - 2|lcs|$. We want to evaluate similarity of the whole rules, the head, and the tails, as such the similarity between two rules is then a weighted average,

$$sim\,(X \to Y, U \to V) = sim\,(XY, UV)\,/2 + sim\,(X, U)\,/4 + sim\,(Y, V)\,/4\,.$$

SANITY CHECK We first evaluate if our score indeed prefers rules over other patterns. To this end, we generate synthetic data using a ground truth rule set consisting of 6 pairs of the form $\{\epsilon \to X, X \to Y\}$. We then compare the encoded sizes of the ground truth model against alternative models of the form $\{\epsilon \to X, \epsilon \to Y\}, \{\epsilon \to XY\}$ resp. $\{\epsilon \to X, \epsilon \to XY\}$. We find that our score always prefers the ground truth. Next, we evaluate on data without structure. We find that for 60 trials on sequences varying in size from 5 000 to 15 000, SEQRET-MINE, in 51 instances correctly reports no rules. In 9 instances, it returns a single, rule with a true confidence between 10% and 40%.

DESTRUCTIVE NOISE Next, we evaluate robustness against destructive noise. To this end, we generate data as above and then add noise by flipping individual events $e \in S$ with probability ranging from 20% to 100%. We show the results in Figure 4.3a. We observe that SEQRET is robust against noise and still recovers the ground truth well even at 80% noise. At 100% noise, there is no structure in the data and all methods except TNS correctly discovers no rules. Throughout all synthetic experiments POERMA performs better than POERMH, we hence omit POERMH from the synthetic experiments results. Further, as SQUISH regularly crashes, we report averages over finished runs only.

RULE CONFIDENCE Next, we evaluate recovery under different rule confidence levels. We vary the rule confidence from 0.1 to 0.9. We show

the results in Figure 4.3b. We observe that SEQRET is robust against low confidence rules and outperforms the state-of-the-art methods by a clear margin.

VARYING SIZE Next, we evaluate how SEQRET performs for ground truth models, alphabets, resp. rule tails of different sizes. First, we consider data with 10 to 50 ground truth rules. We show the results in Fig. 4.3c. SEQRET works consistently well across all model sizes. Next, we vary the alphabet from 100 to 1000 unique events. We give the results in Fig. 4.3e and observe that SEQRET recovers the ground truth consistently well. Finally, we vary the length of the rule tails from 1 to 11. We show the results in Figure 4.3d. We observe that, except for the case where the rule tail size is 1, SEQRET recovers the rules well and outperforms the competition by a large margin.

RANDOM RULE TRIGGERS Finally, we evaluate if SEQRET can recover conditional dependencies even when the rule heads and rule tails are infrequent in the data. For this, we consider the case where the rule heads occur only by chance. To this end, we generate synthetic data where no rule heads are planted. We insert the rule tails wherever the corresponding rules have triggered. To ensure that the rule heads do occur in the data, we limit its size to 1. We also limit the size of the tail to 1 to ensure they do not stand out as patterns by themselves. We show the results in Figure 4.3f. SEQRET-MINE is able to consistently recover the rules. Here, SEQRET-CANDIDATES performs significantly worse because SQS is not able to find good patterns in this challenging setting.

4.6.2 *Experiments on Real Datasets*

In this section, we examine if SEQRET mines insightful rules from real world data. We first discuss the datasets and then the results.

DATASETS We use eight datasets drawn from five different domains. We consider two text datasets, *JMLR*, which contains abstracts from the JMLR journal, and *Presidential*, which contains addresses delivered by American presidents [170]. *POS* contains sequences of parts-of-speech tags obtained by using the Stanford NLP tagger on the book "History

of Julius Caesar" by Jacob Abbott [145]. *Ordonez* [133] and *Lifelog*[4] contains the daily activities logged by a person over several days. *Rolling Mill* contains the process logs from a steel manufacturing plant [187]. *Ecommerce* contains the purchase history from an online store for several users over 7 months.[5] Finally, *Lichess* contains sequences of moves from chess games played online.[6] In Table 4.1 we provide statistics on all datasets, as well as on the results of the different methods.

GENERAL OBSERVATIONS Overall, we observe that frequency-based methods like POERMA and POERMH discover a high number of rules, making interpretation difficult up to impossible. TNS produces largely redundant rules. COSSU is limited by its restrictive rule language and discovers only very few rules. SQS is a sequential pattern miner that identifies meaningful patterns that permit examination by hand, but does not capture conditional dependencies that SEQRET does successfully model. This difference is evident when comparing the compression achieved by different methods: SQS compresses the data less effectively than SEQRET, likely because SEQRET is more expressive. SEQRET-CANDIDATES improves upon SQS but still compresses worse than SEQRET-MINE.

CASE STUDIES Next, we present illustrative examples to highlight how the results from SEQRET differ from those of state-of-the-art methods. To better understand the results, we examine a phrase from the *JMLR* dataset, '*support vector machine*', that all methods identify in some form. SEQRET-MINE discovers the rule $< \epsilon \rightarrow$ *support, vector*> as well as the rule <*support, vector* $\rightarrow$ *machine*> which expresses that <*support*, *vector*> is a pattern, and that whenever it occurs it increases the probability of but is *not necessarily* followed by *machine*. In contrast, SEQRET-CANDIDATES and SQS both treat the entire phrase as a single pattern, <*support, vector, machine*>, failing to capture the independent existence of <*support, vector*> and the conditional dependency involved. On the other end of the spectrum, POERMA and TNS discover 12 resp. 14 rules involving either *support* or *vector*, many of which are semantically redundant.

4 https://quantifiedawesome.com/

5 https://www.kaggle.com/datasets/mkechinov/ecommerce-behavior-data-from-multi-category-store

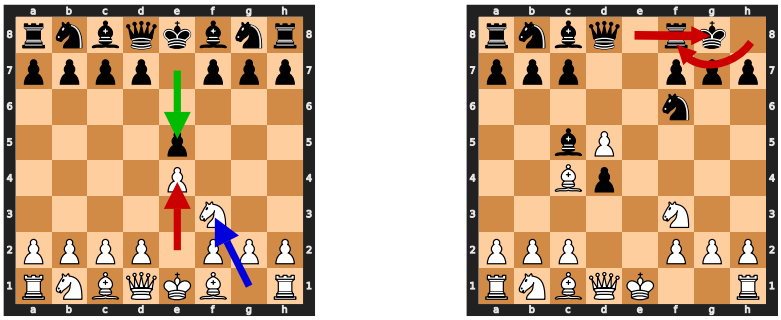
6 https://www.kaggle.com/datasets/datasnaek/chess

In the *POS* dataset, SEQRET discovers common sentence structures, such as the pattern <*to*, *verb-base-form*>, and the pattern <*determiner*, *cardinal number*> capturing phrases such as "the first". SEQRET also captures rules, e.g. <*to*, *verb-base-form* → *personal-pronoun*> and <*to*, *verb-base-form* → *possessive-pronoun*>. These rules correctly identify how either personal pronouns or possessive pronouns tend to follow phrases like "to tell" or "to give". COSSU, the method closest to our approach, fails to find any of the rules discussed. Meanwhile, SQS finds these structures as several independent patterns disregarding the conditional dependency. The frequency-based methods again return overly many and highly redundant rules.

On the *Rolling Mill* data, SEQRET discovers rules that clearly represent different parts of the production process. For example, <*stab*, *gies*, *sort* → *brwa*, *lmbr*, *tmbr*> captures the transition from steel mill (where hot iron is casted and sorted to slabs) to the rolling mill (where slabs are rolled to plates). This demonstrates the power of rules and patterns in the same set, as there are instances where <*stab*, *gies*, *sort*> is not followed by <*brwa*, *lmbr*, *tmbr*> but such instances are rare. SQS again finds several patterns involving parts of <*stab*, *gies*, *sort*, *brwa*, *lmbr*, *tmbr*> but does not explicitly model the conditional dependency. COSSU fares better than in other datasets but nevertheless misses many important dependencies resulting in poor compression.

For the *Lichess* dataset, SEQRET finds the well known "King's Pawn Game" opening move, as the pattern <*white:e4*, *black:e5*>. In addition, it discovers 12 rules with rule head <*white:e4*, *black:e5*>, capturing the different variations that often follow. For example the rule <*white:e4*, *black:e5* → *white:Nf3*>, we show this rule in Figure 4.4a, the red arrows corresponds to move <*white:e4*>, green to <*black:e5*>, and blue to <*white:Nf3*>. SQS, on the other hand, needs several partly redundant patterns, i.e. repeating moves <*white:e4, black:e5*> and <*white:Nf3*>, to explain the same dependencies.

Diving deeper into results on *Lichess*, SEQRET discovers rules involving "King's side castling" (denoted by O-O and shown in Figure 4.4b) which despite being insightful conditional dependencies are missed by SQS. Examples are <*black:O-O* → *black:Re8*> and <*black:O-O* → *white:Qe2*>. The former captures black moving its rook to e8, a position originally occupied by the King and made available only after castling. The latter captures white moving its queen to e2 following



(a) (b)

Figure 4.4: Rules discoverd on the *Lichess* dataset: In (a) we show the rule <*white:e4 (red arrow), black:e5 (green arrow)* → *white:Nf3 (blue arrow)*>. In (b) we show black castling <*black:O-O*>. Castling is a special chess move where the king moves two squares toward a rook, and the rook jumps over the king to the adjacent square.

black castling, as a deterrence to black rook (as e8 is in the line of attack of the queen). Another example is the pattern <*black:Nf6*, *black:O-O*> which makes sense as moving the knight away is a prerequisite for castling. For frequency-based methods, we find anywhere between 128 and 1370 rules involving castling, most of which are redundant.

4.7 CONCLUSION

We considered the problem of mining a succinct set of rules from event sequences. We formalized the problem in terms of the MDL principle and presented the SEQRET-CANDIDATES and SEQRET-MINE algorithms. We evaluated both on synthetic and real-world data. On synthetic data we saw that SEQRET recovers the ground truth well and is robust against noise, low rule confidence, different alphabet sizes, and rule set sizes. On real-world data SEQRET found meaningful rules and provides insights that existing methods cannot provide.

As future work, we consider it highly interesting to study the causal aspects of sequential rules. Our approach lends itself to a causal framework by mapping the rule heads and tails to temporal variables and re-modeling the rules as structural equations involving these variables.

In Chapter 7 we move towards causal rules by studying causal relationships between events.

				SEQRET-CANDS			SEQRET-MINE			SQS		POERMA	POERMH	TNS	COSSU	
Dataset	$\|\|D\|\|$	$\|D\|$	$\|\Omega\|$	$\|P\|$	$\|R\|$	$\%L$	$\|P\|$	$\|R\|$	$\%L$	$\|P\|$	$\%L$	$\|R\|$	$\|R\|$	$\|R\|$	$\|R\|$	$\%L$
Ordonez	739	2	10	2	0	11.45	1	5	16.85	2	11.45	95923	113337	6	2	-2.42
JMLR	14501	155	1920	62	9	1.90	2	203	3.33	116	1.61	127	1282442	205	–	–
Rollingmill	18416	350	446	158	50	52.22	9	313	56.33	247	50.46	–	–	247	46	20.56
Lichess	20012	350	2273	81	18	2.42	11	326	4.57	113	2.13	4326	2068	348	–	–
Ecommerce	30875	4001	127	77	10	13.72	89	130	27.87	95	13.59	43513	231824	219	6	-0.09
Lifelog	40520	1	78	36	6	8.85	16	79	9.97	59	6.51	2001521	1932296	95	5	-1.66
POS	45531	1761	36	65	6	18.36	38	33	18.12	160	12.64	–	–	71	5	-0.52
Presidential	62010	30	3973	30	4	0.46	2	129	0.96	58	0.38	57	90552	131	–	–

Table 4.1: Results on real-world data. We report the number of discovered patterns (non-empty-head rules) P and rules R. For SEQRET, SQS and COSSU, we report the percentage of bits saved against the SEQRET null model as $\%L$. Failed runs, e.g. because of excessive runtime (COSSU) or out-of-memory errors (POERMA and POERMH), are marked by '–'.

5

SUMMARIZING EVENT SEQUENCES WITH GENERALIZED SEQUENTIAL PATTERNS

In the previous chapters, we studied how to summarize event sequences in terms of serial episodes and sequential rules. The proposed methods provide summaries in terms of what we call *surface* level patterns. That is, patterns over *observed* events in the data. *Surface* level patterns can not capture patters that, at specific location, emit one symbol out of a restricted set of symbols, for example *ab* followed by *c* or *d*. To address this, we study the problem of succinctly summarizing a database of event sequences in terms of *generalized* sequential patterns. That is, we are interested in patterns that are not exclusively defined over observed surface-level events, as is usual, but rather may additionally include *generalized* events that can match a set of events. We are not only interested in discovering *generalized* patterns but also the *generalized* events — both directly from the data.

5.1 INTRODUCTION

Succinctly summarizing a database in easily understandable terms is one of the key problems in data mining. Pattern set mining, where we mine a small sets of patterns that together model the data well, has proven to be particularly successful [57, 170, 180]. Existing methods, however, only consider what we call surface-level patterns. These are patterns that are exclusively defined over observed events, and therewith also only match exact instances in the data.

To illustrate the limitations of surface-level patterns, let us consider a toy example. The two sentences *'the cat meows'* and *'the dog barks'* share only the event *'the'*. Any method that only considers surface-

This chapter is based on [35]: Joscha Cüppers and Jilles Vreeken. "Below the Surface: Summarizing Event Sequences with Generalized Sequential Patterns." In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2023, pp. 348–357.

level events would either just report 'the' as a common pattern, or, if they occur frequently often enough in the data report both sentences as patterns, neither of which is particularly useful. In contrast, any human would immediately see that these sentences are both instances of the general statement *'the [pet] [makes noise]'*, and would be annoyed to get a summary that would both explicitly report all variants of this general pattern (e.g. mice squeaking, horses whinnying) as well as fail to report rare instances (e.g. fishes saying blub). For natural language, there exist high-quality word ontologies that we can use to analyse text through a more general lens [9, 70]. However, for event sequence data in general, this is not the case. This raises the question, how can we automatically discover a set of patterns that succinctly describes the data in terms of more general patterns?

In this chapter we consider the problem of discovering generalized events and generalized patterns from event sequence data. A generalized event is a symbol that can match different observed events e.g. $\alpha = \{a, b\}$ matches a and b. A generalized pattern is a sequential pattern that is defined over observed and generalized events, e.g. pattern c, α, d matches c, a, d and c, b, d. This more expressive pattern language enables us not only to summarize event sequence data more effectively, but also to provide deeper insights, as it is less prone to under- or over-fitting compared to a pattern language of surface-level patterns. In this context underfitting means that patterns are either not reported or only partially, overfitting means semantically identical patterns are reported multiple times.

We define the problem of discovering the best set of generalizations and patterns in terms of the Minimum Description Length principle [71]. Loosely speaking, we are after those that together provide the best lossless compression. The search space for this problem is vast, triply-exponential, and is not favourably structured, which is why we propose the FLOCK algorithm to heuristically mine good models from data. FLOCK finds high-quality generalizations by considering those events that frequently appear in the same (pattern) context, and finds high-quality generalized patterns by iteratively merging patterns and extending them with discovered generalizations.

FLOCK aside, very few methods consider sequential patterns beyond surface level patterns, and either require a beforehand known structure [9, 70, 163] or can only model 'generalizations' that are limited to a

single location in a pattern [11]. As we will see, methods that only consider surface-level patterns are prone to highly redundant results – after all, they cannot generalize and are hence bound to report every sufficiently frequent variation of a true generating pattern – but also to underfitting, because they only report sufficiently frequent instances rather than the more rare but important variants.

Through an extensive set of experiments and comparisons to a wide array of competitors, we show that our method works well in practice. On synthetic data, we show that FLOCK recovers surface-level patterns as well as the state of the art, but that it outperforms these competitors by a large margin in recovering generalized patterns and generalizations. On real-world data, we show that the small sets of highly expressive patterns that FLOCK discovers provide clear insight into the data-generating process that goes far beyond what surface-level patterns can provide.

5.2 PRELIMINARIES

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

5.2.1 *Notation*

We consider a database D of $|D|$ event sequences. An event sequence $S \in D$ consists of $|S|$ events drawn from an alphabet Ω_o of *observed* events $e \in \Omega_o$. We write $S[j]$ to refer to the j^{th} event in S and $S[j:k]$ to mean a subsequence $S[j] \dots S[k]$. Note, we do not allow multiple events to occur at time point j.

In addition to observed events $e \in \Omega_o$, we also consider *generalized* events $\alpha \in \Omega_g$. Generalized events are special in that they match multiple observed events $e \in \Omega_o$, e.g. $\alpha = \{a, b\}$ will match either a or b. We allow generalizations to be nested, e.g. $\beta = \{\alpha, c\}$ will match any out of a, b, or c. We can flatten a generalized event, $fl(\alpha)$, to obtain all observed events that α can match.

As patterns we consider serial episodes. A serial episode $p \in \Omega^{|p|}$ is a sequence of $|p|$ events over an alphabet $\Omega = \Omega_o \cup \Omega_g$. We say that a sequence S *contains* an instance of a pattern p if there exists a window $S[j:k]$ that matches p. We explicitly allow gaps between the events in

p. To avoid spurious matches we consider windows up to a length of $|p| + |p|n$, where n is a user-chosen parameter. The *support* of a pattern in D is the number of unique matches of p, note that one pattern can match multiple times per sequence.

During search we iteratively refine the generalized alphabet Ω_g by adding and removing (events from) generalizations $\alpha \in \Omega_g$. We write $(\alpha, R, \oplus)$ to denote that refinement of Ω_g where we add event set $R \subset \Omega$ to existing generalization $\alpha \in \Omega_g$, or adding a new generalization $\alpha = R$ to Ω_g if $\alpha \notin \Omega_g$. Analogously, we write $(\alpha, R, \ominus)$ whenever we want to remove (events from a) generalization α. Wherever clear from context we do not write the $\oplus$ and $\ominus$. To denote a set of additive refinements for Ω_g we write $\Omega_g^{\oplus}$, and analog $\Omega_g^{\ominus}$ to denote a set of removal refinements.

5.3 MDL FOR GENERALIZED SEQUENTIAL PATTERNS

We will now define the problem we aim to solve. As model class $\mathcal{M}$ for a dataset D over observed alphabet Ω_o, we consider tuples that define a generalized alphabet Ω_g, and a set of patterns P over $\Omega = \Omega_o \cup \Omega_g$. To ensure that every model $M \in \mathcal{M}$ can validly encode D we require P to always include all singleton patterns, i.e. $P \supseteq \Omega_o$. By MDL, we are interested in that $M \in \mathcal{M}$ that most succinctly describes D without loss.

5.3.1 *Decoding a Sequence*

Before we define how we *encode* a dataset given a model M, we first give the intuition on its main components by explaining how to *decode* an already encoded sequence S. We give an example in Fig. 5.1. In Cover 1, S has been encoded using singleton patterns only. To decode it, we simply iteratively read pattern codes from the pattern stream C_p, and use model M to decode these to the correct events.

Cover 2 utilizes model M better. We again iteratively read codes from C_p. The first code is for pattern p, and we can immediately append its first event (a '*d*') to the decoded sequence. To determine whether there is a gap or not, we read a code from the meta stream C_m. This happens to be a fill-code (!), meaning we can write the next event of p. This is the generalized event α that can match either e or

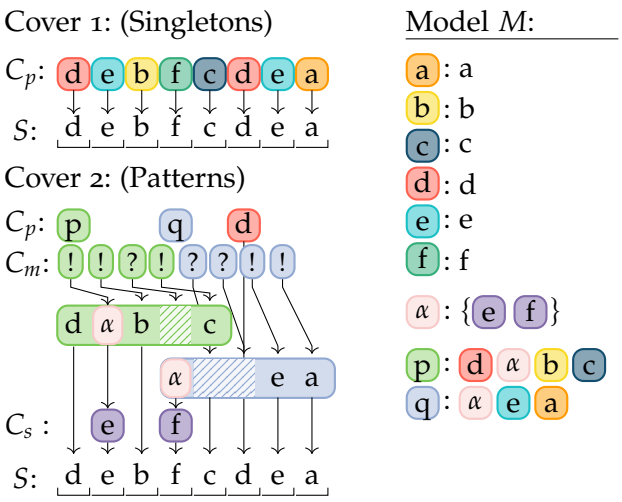


Figure 5.1: Toy example showing two ways to encode the same sequence S. Cover 1 uses only singletons, while Cover 2 uses the entire model M. A cover C consists of (up to) three different code streams: C_p contains codes for patterns, C_m defines how these interleave, and C_s specifies which observed events $e \in \Omega_o$ the generalized events $\alpha \in \Omega_g$ in the cover map to.

f. To determine which of these two events we have to emit, we read a code from the specification stream C_s, and proceed accordingly. We then continue as before, reading another fill code, and writing a 'b'. Next, we read a gap-code (?) from the meta-stream, which informs us that there is a gap in pattern p. To fill this gap, we have to read the next code from the pattern stream. We read the code for pattern q, and hence write its first event to the sequence. We now have two patterns that could emit the next event. We therefore read as many meta codes as there are active patterns. If all of these are gap codes, we read from the pattern stream, and otherwise we emit the next event for that pattern for which we read a fill code. Here, the latter is the case for p, we write the corresponding 'c', and are finished decoding p. To wrap things up, we read the next meta-code for q, which is a gap that we fill according to the next pattern code ('d') and finally read two fill codes for q and hence emit 'e' and 'a', after which we have decoded S without loss.

5.3.2 *Calculating the Encoded Length*

Now that we know what we need to encode, we define how many bits these codes should cost.

ENCODING THE DATA We start by defining how to compute the encoded cost of a database D given a model M. Formally, we have

$$L(D|M) = L_{\mathbb{N}}(|D|) + \left(\sum_{S_i \in D} L_{\mathbb{N}}(|S_i|) \right) + L(C_p) + L(C_m) + L(C_s) \quad . \tag{5.1}$$

We first encode the number of sequences, and then the length of each sequence in the database. We then encode the pattern stream C_p, meta stream C_m, and specification stream C_s. We encode the number and length of the sequences using $L_{\mathbb{N}}$, the MDL-optimal encoding for integers $z \geqslant 1$ [153].

We next discuss the three code streams. We start with the pattern stream C_p, Eq. (5.2). Because the occurrences of pattern codes in the

pattern stream are independent, we encode these using optimal prefix codes. Formally, we have

$$L(C_p) = -\sum_{p \in M} usg(p) \log \left(\frac{usg(p)}{\sum_{p' \in M} usg(p')} \right) \quad , \tag{5.2}$$

where $usg(p)$ is the number of times the code for p appears in pattern stream C_p. To use optimal prefix codes we will have to explicitly encode the usages in the model.

In contrast, the occurrences of codes in the meta stream C_m, Eq. (5.3), are dependent on which patterns we are currently decoding, meaning we need to know (many) conditional probabilities. To avoid having to make arbitrary choices on how to explicitly encode these in the model, we propose to use prequential codes [71] (see Chapter 4 Section 4.3). Formally, we have

$$L(C_m) = \sum_{p \in P} \left(- \sum_{i=1}^{fills(p)} \log \left(\frac{\epsilon + i}{2\epsilon + i} \right) - \sum_{i=1}^{gaps(p)} \log \left(\frac{\epsilon + i}{2\epsilon + fills(p) + i} \right) \right), \tag{5.3}$$

where $fills(p)$ and $gaps(p)$ refers to the number of fills resp. gaps of pattern p in meta-stream C_m, and ϵ is a small constant. As is common in prequential coding, we set ϵ to 0.5.

This leaves the encoding of the specification stream C_s, Eq. (5.4). Because specification codes depend on the context of the generalization at hand, we will again use prequential codes. Generalizations *within* a pattern p, however, can additionally be dependent on *each other*. for example, cats meow, dogs bark. To exploit and reveal such structure, we allow for dependencies between generalizations within a pattern. We provide the details in the model encoding below. For now, what matters is that we encode the specification code for an event $e \in fl(\alpha)$ for the current generalization α of pattern p conditioned on an earlier emitted event d of p. Formally, the length in bits of the entire stream is

$$L(C_s) = \sum_{p \in P} \sum_{\alpha \in p} \sum_{i=1}^{usg(p)} -\log \left(\frac{\epsilon + usg_i(e|d)}{|fl(\alpha)|\epsilon + \sum_{c \in fl(\alpha)} usg_i(c|d)} \right) \quad , \tag{5.4}$$

where for each pattern $p \in P$ (first sum), and each generalization $\alpha \in p$ (second sum), we encode the surface-event e conditioned on the value

of event d using prequential codes (third sum). Note that if a generalization $\alpha \in p$ is *not* dependent on an earlier emitted generalization $\beta \in p$, d will be a fixed constant by which the above becomes a standard unconditional prequential code.

ENCODING THE MODEL Next, we define how to compute the encoded cost of a model. We start by defining the encoded cost for the generalized alphabet Ω_g. We have

$$L(\Omega_g) = L_{\mathbb{N}}(|\Omega_o|) + L_{\mathbb{N}}(|\Omega_g| + 1) + \sum_{k=1}^{|\Omega_g|} \left(\log k + \log \binom{k-1}{l} + \log(|\Omega'_o|) + \log \binom{|\Omega'_o|}{m} \right) \quad , \tag{5.5}$$

where we first encode the sizes of the observed[1] resp. generalized alphabets using $L_{\mathbb{N}}$. We then encode the generalizations $\alpha \in \Omega_g$ in turn. For each generalization $\alpha_k \in \Omega_g$, we first transmit how many nested generalizations it includes, denoted as l, and then identify which these are using a data-to-model code over the $k-1$ generalizations transmitted so far. We then transmit the number of observed events, denoted as m, in Ω_o that α includes, which are not already defined by its nested generalizations. Once we know this number, encode which events out of Ω'_o these are, where Ω'_o is the set of observed events excluding events already defined by its generalizations, formally $\Omega'_o = \Omega_o \setminus \bigcup_{\beta \in \alpha_k} fl(\beta)$.

Given the generalizations, we can next encode the pattern set P and their respective usages, i.e. the code table, Eq. (5.6). We first transmit the number $|P'|$ of non-singleton patterns $P' \subset P$, and then the *combined* usage of all patterns. We finally encode each pattern $p \in P'$. We have

$$L(CT) = L_{\mathbb{N}}(|P'|) + L_{\mathbb{N}}(usg(P)) + \log \binom{usg(P) + |\Omega_o| - 1}{|P'| + |\Omega_o| - 1} + \sum_{p \in P'} L(p) \; . \tag{5.6}$$

To encode a pattern $p \in P'$, Eq. (5.7), we first transmit its length using $L_{\mathbb{N}}$. We then encode which events and generalizations it includes, and finally for each $\alpha \in p$ we encode whether and if so on which ear-

1 Note that as the size of Ω_o is constant for any model of the same data, it is unnecessary to include the first term for optimization, but we include it to have a lossless code.

lier generalization it depends (the plus one corresponds to a dummy symbol that represents independence). Formally,

$$L(p) = L_{\mathbb{N}}(|p|) + |p| \log(|\Omega|) + \sum_{i=1}^{|p|} \log(k+1) \quad , \tag{5.7}$$

with k is $|\{j \mid p[j] \in \Omega_g, i < j\}|$ if $p[i] \in \Omega_g$ else $k = 0$. By which we have a lossless encoding for model M, $L(M) = L(\Omega_g) + L(CT)$, and data D, by which we can now formally state the problem.

The Minimal Generalized Pattern Set Problem *Given a sequence database D over an event alphabet Ω_o, find the smallest pattern set P and generalization set Ω_g such that the total encoded size*

$$L(D, M) = L(M) + L(D|M)$$

is minimal.

For a given database D over observed alphabet Ω_o there exist exponentially many patterns sets P, exponentially many generalization sets Ω_g, and exponentially many possible covers C. Worst of all, the search space of neither the overall nor of the subproblems exhibits any structure such as (weak) monotonicity or submodularity that we can exploit for our search. Hence, we resort to heuristics.

5.4 ALGORITHM

To find good models in practice we propose to break the problem into two parts: 1) given a model M find a good cover C, and 2) given a cover C find a good model M. We discuss these in turn.

5.4.1 *Covering the Data*

We start by determining a good cover C given a model M. A valid cover is a set of windows that covers each event in database D only once. To find a C that minimizes $L(D|M)$ we first need for each pattern $p \in P$ all windows in D that match p. To find these efficiently, we use an inverted index.

Next, we describe how we find a cover C given a set of windows. Given that there are exponentially many possible covers [11], determining the optimal cover is computationally not feasible, therefore we

Algorithm 8: REFINE

input : pattern p and current cover C
output: set of candidates Q

$F \leftarrow \text{FREQUENTFOLLOWERS}(p, C)$
$Q \leftarrow p \times \{q \in F \mid |q| > 1\}$
$Q \leftarrow \{q \in Q \mid \Delta\bar{L}(q) > 0\}$
$Q \leftarrow Q \cup \text{EXTENDPATTERN}(p, \{q \in F \mid |q| = 1\})$
return Q

approach this problem greedily. To this end, we define an order over all windows where we consider window $w_1 > w_2$ if, in order of priority, $|p_1| > |p_2|$, $gaps(w_1) < gaps(w_2)$, $support(p_1) > support(p_2)$, and finally lexicographically, where w_1 is a window of pattern p_1, analogously for w_2. Intuitively, we prefer patterns that cover many events with as few gaps as possible, as these will likely result in the shortest description of D. To find a cover given a window set, we consider each window w. If there does not exist a higher ranked window that conflicts, i.e. overlaps, with w, all windows that conflict with w are discarded. We repeat this process until all conflicts have been resolved, resulting in a valid cover of D.

5.4.2 *Finding Good Models*

Given a sequence database D, our overall goal here is to discover a set of generalizations Ω_g and a set of patterns P that together describe D well. The general idea of our proposed algorithm is to start with an 'empty' model M_0 that only includes the observed events Ω_o and to iteratively and greedily refine this model by adding patterns and generalized events that improve the total encoded length. In each iteration, we generate a set of candidate patterns based on the current model, evaluate these, and if a candidate improves the model, we add it to the model. To avoid getting stuck with stale patterns and generalizations, we clean up at the end of each iteration by flattening generalizations and merging similar patterns. We now explain these steps in detail.

GENERATING CANDIDATES A key part of our proposed algorithm is to find improved or *refined* versions of a given pattern p. We provide

the pseudocode as Algorithm 8. The general idea for refining a pattern p is to check whether there is any structure in the events and patterns that often occur soon after p in cover C. We then generate candidate patterns by concatenating pattern p with patterns q (line 2) that occur within the maximum number of allowed gaps $n|p|$ (line 1). We discard all candidates for which we estimate that they will not lead to any gain (line 3). With $\Delta L(q)$ we denote how many bits we actually save (or lose) by adding q to our model, but as computing this exactly is computationally costly we instead use an efficiently computable *optimistic estimate* $\Delta\overline{L}$ that we define below.

It is relatively straightforward to see how to instantiate the above strategy for refining an existing pattern with a singleton or pattern $q \in P$, as it essentially amounts to counting how often in C every possible q occurs within the maximum window length around p. It is much less clear how to discover good candidate generalizations α, however. The first idea that comes to mind is to 'simply' first use the above strategy to find a model M that only includes patterns over Ω_o, and then to merge those patterns in M that are most similar, replacing the events where they differ with a new generalized event α. While this strategy works to a certain extent, it can only discover the most frequent generalized events and patterns, and will not truly solve the problem at hand.

We therefore propose an improved strategy for discovering generalizations, where for a given pattern p we consider the distribution of when which events happen close to p in C. We provide pseudocode in Algorithm 9. The main idea is that if two or more events a and b often occur within a similar number of time steps after pattern p, they are good candidates to be included in a new generalized event as there is evidence they have a similar contextual (possibly, semantic) relation to p. Specifically, we propose to generate candidate generalized events based on the similarity of the distributions of delays between pattern p and occurrences of events $e \in \Omega$. A delay distribution of event $e \in \Omega$ relative to pattern p captures how often and how many times steps after p event e occurs; technically we implement this non-parametrically using a histogram with one bin per time step. We construct these delay distributions in line 1 of Alg. 8 for all $q \in \Omega$ that occur within the $n|p|$ time steps after p in C.

Algorithm 9: ExtendPattern

input : pattern p and delay distributions F
output: Candidate pattern $p*$

1 $F \leftarrow \text{ExtendWithGeneralizations}(F)$
2 $e' \leftarrow \arg\max_{e \in F} \frac{counts(e)}{\tilde{E}}$
3 $p' \leftarrow p \times e'$, $p* \leftarrow p$
4 $\Omega_g^{\oplus} \leftarrow \varnothing$
5 $Q \leftarrow [(e, (|p|, e')) \quad | e \in F]$
6 **while** $\Delta\overline{L}(p') > \Delta\overline{L}(p*)$ increasing **do**
7 $\quad p* \leftarrow p'$
8 $\quad e, (i, e') \leftarrow \text{top}(Q)$
9 $\quad p_1 \leftarrow cp_i(p, i{-}1, e)$ **if** $\tilde{E}{<}\tilde{E}'$ **else** $cp_i(p, i, e)$
10 $\quad$ **if** $e' \in \Omega_g$ **then**
11 $\quad\quad \alpha \leftarrow e'$
12 $\quad\quad \Omega_g^{\oplus\prime} \leftarrow \Omega_g^{\oplus} \cup \{(\alpha_{id}, \{e\})\}$
13 $\quad$ **else**
14 $\quad\quad \alpha \leftarrow \varnothing$
15 $\quad\quad \Omega_g^{\oplus\prime} \leftarrow \Omega_g^{\oplus} \cup \{(\alpha_{id}, \{e', e\})\}$
16 $\quad p_2 \leftarrow cp_r(p, i, \alpha)$
17 $\quad p' \leftarrow \arg\max_{p \in \{p1, (p2, \Omega_g^{\oplus\prime})\}} \Delta\overline{L}(p)$
18 $\quad \Omega_g^{\oplus} \leftarrow \Omega_g^{\oplus\prime}$ **if** $p' = p2$
19 $\quad$ update Q
20 **return** $(p*, \Omega_g^{\oplus})$

To maximize the chance that the resulting generalization will improve the overall cost, we start extending a pattern p with the event e' that has the highest delay probability mass, i.e. occurs frequently with the same delay after p, formally $e' = \arg\max_{e \in F} counts(e)/\tilde{E}$, where $\tilde{E}$ is the median delay between p and e (line 2-3). Next, we seek if there are other events that together with the just added e' would instead form a promising generalization α. We do this by testing events $e \in F$ in order of the most similar delay distribution to e' normalized by frequency. That is, $W1(E', E)/counts(e)$ where E' and E refer to the

respective distributions, and $W1$ is the Wasserstein distance [177] between two delay distributions, defined as

$$W1(E1, E2) = \min_{N} \sum_{i \in E1} \sum_{j \in E2} n_{i,j} d_{i,j} \quad , \tag{5.8}$$

where $n_{i,j}$ refers to the probability mass that has to be moved between i and j and $d_{i,j}$ to the distance. By $counts(e)$ we refer to the frequency within $n|p|$ time steps after p.

This may once again seem like a sound strategy, but comes with the problem that events that are next to each other will have similar delay distributions. That means we have to additionally test whether event e is truly part of a generalization together with e' or is simply a regular neighbor to that event or generalization. We therefore, evaluate the gain in compression by either extending the generalization with a new event (line 16) or by extending the pattern directly with the event left or right (line 9). Note with $cp_i(p, i, e)$ we create a new pattern where event e is inserted behind the i^{th} event of pattern p, and analog with $cp_r(p, i, \alpha)$ we create a new pattern where the i^{th} event is replaced by α. Given both extensions, we take the one for which our estimated gain is higher (line 17). After each added event we update the priority queue. Here we described the procedure of extending a pattern by adding events or patterns to the back; we do the same with preceding events and patterns. This concludes the description of how we create a set of pattern candidates and generalization candidates given a pattern and the current cover.

FINE-TUNING CANDIDATES The overall algorithm takes the candidates generated above, and tests, in order of estimated gain, for addition to the model. Testing a candidate for addition involves computing $L(D, M)$ and therewith a new cover C'. That is, we now know where and which instances of p are used, and can take advantage of that information and further refine the pattern to minimize the total encoded length. We do so by pruning the generalized events to only include those instances that are used in a way that aids compression. To this end, we test for each event in $\Omega_g^{\oplus}$ whether removing it will improve the gain in bits (PRUNEGEN). As we allow for gaps in the occurrences, we can further take advantage of the cover C' by extending pattern p with events that frequently occur in these gaps (REFINEINTERLEAVING). This includes generalized events that are built from multiple observed

Algorithm 10: MERGE

input : Pattern set P and generalization Ω_g, cover C
output: Pattern set P and generalization Ω_g, cover C

1 $Q \leftarrow []$
2 **forall** $p \in P$ **do**
3 $\quad q \leftarrow \arg\max_{\{q \in P \mid |q|=|p|\}}$ overlap between p and q
4 $\quad$ **if** overlap between p and $q > 1$ **then** $Q.add(p, q)$;
5 **forall** $p, q \in Q$ **do** in order of 1. overlap 2.combined usage
6 $\quad p', \Omega_g^{\oplus} \leftarrow$ merge p and q
7 $\quad$ **if** $\Delta L(p') > 0$ **then**
8 $\quad\quad P \leftarrow (P \cup \{p'\}) \backslash \{p, q\}$
9 $\quad\quad$ apply $\Omega_g^{\oplus}$ to Ω_g
10 $\quad\quad$ replace all p and q with p' in C
11 **return** P, Ω_g, C

events that occur in the gaps of p. We provide the pseudocode for both procedures, as well as a more detailed description, in Appendix d.1.

SIMPLIFYING THE MODEL By iteratively adding more specialized patterns and generalizations to the model, previously discovered patterns and generalizations may no longer positively contribute to the MDL score. We therefore prune the model after each iteration by merging similar patterns and by flattening generalizations. We discuss these in turn.

We provide pseudocode of the merge procedure as Algorithm 10. We consider merging two patterns p and q if they have the same length (line 3) and have an overlap of at least two events (line 4) where we say an event overlaps if $p[i] = q[i]$. To prioritize pattern mergers likely to improve compression and meaningful generalizations, we merge patterns in order of overlap and combined pattern usage, both decreasing (line 5). When merging two patterns we create for all events where $p[i] \neq q[i]$ a new generalization $\alpha = \{p[i], q[i]\}$. Through this process it is possible to create the same generalization twice, if that happens we replace all instances with the same generalization and delete the other one. Since the new pattern will match all windows that the source patterns matched, we do not have to recompute a new cover and can

Algorithm 11: FLATTEN

input : Pattern set P and generalization Ω_g
output: Pattern set P and generalization Ω_g

1 **forall** $\alpha \in \Omega_g$ **do**
2 **if** α used in just one other $\beta \in \Omega_g$ **then**
3 extend β with $fl(\alpha)$
4 **forall** $p \in P$ **do**
5 replace α with β in p
6 $\Omega_g \leftarrow \Omega_g \backslash \{\alpha\}$
7 **if** $L(D, M)$ did not decrease **then**
8 revert all changes
9 **return** P, Ω_g

directly compute by how many bits our encoding will change (line 7). If we have a positive gain we keep the new pattern, and the corresponding generalization, and discard the two source patterns (line 8).

Next, we discuss how we simplify the generalizations. We provide the pseudocode as Algorithm 11. If a generalization α is used in only one other generalization β, we consider merging it with its parent(s) (line 3), meaning we add all events $e \in fl(\alpha)$ to β and replace α with β in all patterns $p \in P$ (line 5). Similar to above, as the updated patterns match the same positions as before we can compute the total encoded size without recomputing the cover. If we obtain a gain, we keep the change, otherwise we revert (line 8).

As both types of simplification steps can create new candidates for the other, we call MERGE and FLATTEN alternating until convergence. The SIMPLIFY algorithm can also be applied to post-process the results of traditional sequential pattern miners in order to reveal generalizations from surface-level patterns. We will use it as such in the experiments to permit a comparison to the state of the art.

ESTIMATING GAINS Exact computation of our MDL score requires computing the cover, which is a computationally costly operation. Rather than always relying on the exact score, we use an *optimistic estimator* $\Delta\overline{L}(p, \Omega_g^{\oplus})$ of the gain in compression where possible. We can estimate

the gain $\Delta\overline{L}$ by breaking it down into two parts: the cost of pattern p in the model and the change to the encoding of the data by the updated model. The bits needed to describe a new pattern or generalization can be computed efficiently as shown in Section 5.3, and no estimation is necessary; when extending an existing generalization we simply consider the difference in encoding cost between the old generalization and extended generalization. We propose to optimistically estimate the encoded cost of the data given the updated model, by using the usage statistics of the previous cover. To give the intuition, suppose we create a new pattern p by concatenating q and r, we then estimate the usage of p by the minimum usage of q and r, and estimate the new usages of these patterns by subtracting exactly that amount. We then simply compute $L(D|M)$ with these estimated usages. We give further details on how we estimate the new usages and how to compute $\Delta\overline{L}(p', \Omega_g^{\oplus})$ in Appendix d.1.

5.4.3 *The* Flock *Algorithm*

Now that we have seen all the individual parts, we can explain the Flock algorithm in detail.[2] The general idea is to start with an empty pattern set P and an empty generalization set Ω_g and iteratively add patterns and generalizations until adding new patterns no longer improves our model M. We give the pseudocode in Algorithm 12. To keep track of patterns we want to extend to a more refined version we maintain a set O, and initialize O with all singletons (line 1). At each iteration, we refine all $p \in O$ to pattern candidates (line 5). A candidate base on pattern p consists of two parts, a more specific pattern p' and a generalization extension $\Omega_g^{\oplus}$. All pattern candidates are added to a priority queue that we order by the estimated gain.

Next, we test each candidate. If the candidate gives us an actual gain, we fine-tune the candidate (PruneGen, line 11, and RefineInterleaving, line 12) and add it to the model (line 15). To allow for different refinements of source p and further refinements of pattern p' we add both to the open set O. Iteratively adding candidates to the model does not necessarily result in the most succinct representation, therefore we simplify the model after each iteration (line 17). As patterns

2 The name Flock comes from the expression 'birds of a feather flock together', which was the inspiration for how we search for generalizations.

Algorithm 12: FLOCK

input : sequence database D over alphabet Ω_o
output: pattern set P, generalization set Ω_g

1 $O,\ P \leftarrow \Omega_o,\ \Omega_g \leftarrow \varnothing,\ C \leftarrow D,\ gain \leftarrow +\infty$
2 **while** $gain > 0$ **do**
3 $PQ \leftarrow [],\quad gain \leftarrow 0$
4 **foreach** $p \in O$ **do**
5 $PQ.addAll(\textsc{Refine}(p, C))$
6 $O \leftarrow \varnothing$
7 **while not** $PQ.empty()$ **do** in order of $\Delta\overline{L}(p', \Omega_g^{\oplus})$
8 $p', \Omega_g^{\oplus}, p \leftarrow \text{top}(PQ)$
9 $gain', C' \leftarrow \Delta L(p, \Omega_g^{\oplus})$
10 **if** $gain' > 0$ **then**
11 $\Omega_g^{\oplus}, C \leftarrow \textsc{PruneGen}(p', \Omega_g^{\oplus}, C')$
12 $p', \Omega_g^{\oplus}, C \leftarrow \textsc{RefineInterleaving}(p', \Omega_g^{\oplus}, C)$
13 $O \leftarrow O \cup \{p', p\}$
14 apply $\Omega_g^{\oplus}$ to Ω_g
15 $P \leftarrow P \cup \{p'\}$
16 $gain \leftarrow \max(gain, gain')$
17 $P, \Omega_g, C \leftarrow \textsc{Simplify}(P, \Omega_g, C)$
18 **return** $\textsc{Prune}(P, \Omega_g, C)$

can become superfluous as more specific patterns are added, we test for each pattern whether removing it will decrease the total number of bits, before returning the model (line 18).

COMPLEXITY Finally, we consider the time complexity of FLOCK. In the worst case, we have to find all windows for all possible sequential patterns over alphabet Ω to cover the data. We can find all windows of a pattern p in $\mathcal{O}(||D||^2)$ [11], where $||D||$ denotes the total number of events in D. To cover the data, in the worst case, we have to sort all windows, $\mathcal{O}(||D|| \log ||D||)$ and check each window against all already selected $\mathcal{O}(||D||\ |C|)$. Combined, this gives us an overall complexity of $\mathcal{O}(||D||\ |\mathcal{F}|(||D|| + \log(||D||) + |C|))$.

5.5 RELATED WORK

While not technically pattern mining, research in Natural Language Processing (NLP) on finding synonyms, computing similarities, and constructing ontologies over words are related to our work. Extensive ontologies have been constructed that capture relationships between words [122] but only little work exists on automatically discovering high-quality ontologies directly from data. Neural networks have been shown to produce embeddings that place semantically similar words close to each other [44, 121] yet unlike our patterns these embeddings do not allow for a straightforward interpretation. In the experiments we will compare FLOCK to WORD2VEC with resp. to the bag-of-words and skip-gram architecture [121].

Process mining is more closely related to sequential pattern mining as it also considers event sequences. Instead of mining insightful patterns, it however focuses on discovering process models with explicit temporal semantics for reconstructing sequences from start to finish [187]. As these processes can get very complex, methods have been proposed to abstract sub-processes into high-level activities [74, 166, 173], whereas we aim to find generalizations over individual events. The key difference to process mining is that we focus on event sequence data in general, and are interested in patterns that characterize these sequences without requiring that every sequence has been produced by the same process.

Frequent pattern miners can be adapted to only report patterns that match predefined constraints, including `or`-structures through regular expressions [60, 139, 140], in contrast to our method the specific `or`-structures have to be provided beforehand and are not discovered.

Mining *generalized* sequential patterns has been studied in the seminal work of Srikant and Agrawal [163, 164] they however require a taxonomy and suffer from the well-known pattern explosion of frequent pattern mining. Closer to our method are the proposals of Grosse and Vreeken [70] and Beedkar and Gemulla [9] who both study the problem of summarizing data *given* an ontology. We, on the other hand, aim to discover both the generalizations and patterns without prior knowledge.

SQUISH [11] comes closest to our approach, as it is able to discover patterns that include `or`-structures and follows a similar MDL based

approach. SQUISH discovers `or`-structures in a post-processing step where it combines discovered pattern instances that are exactly the same except for one event. In contrast, we jointly search for generalizations and generalized patterns, by which we can identify much richer (more subtle) generalizations. We allow generalizations to be re-used between patterns, as well as explicitly model dependencies between generalizations within a pattern in order to obtain highly informative models. We compare to SQUISH [11] in the experiments.

5.6 EXPERIMENTS

In this section, we empirically evaluate FLOCK on synthetic and real-world data. We implemented FLOCK in `C++` and provide the source code for research purposes, along with the used datasets in the supplementary material.[3] We compare to SQS [170], SQUISH [11], ISM [57], SKOPUS [144], and WORD2VEC [121].

As SQS, ISM, and SKOPUS only consider surface-level events, we post-process their results using the SIMPLIFY Algorithm, to extract generalized patterns and generalized events from their results. We consider both the bag-of-words and skip-gram defined versions of WORD2VEC [121], clustering the embedding using DBSCAN [50], merging events part of the same cluster into a new generalized event, and finally apply SQS on the resulting data to find generalized sequential patterns. As per default, we set the gap parameter of FLOCK to $n = 10$, in Appendix d.2 we provide a sensitivity analysis that shows that FLOCK is robust against the parameter choice. For data with very low structure we see that a low n produces better results. We give a more detailed description of the experimental setup, and an ablation study on parts of the algorithm, in Appendix d.2.

5.6.1 *Synthetic Data*

To evaluate how well FLOCK recovers the ground truth we consider synthetic data. We sample, uniformly at random, databases D consisting of 100 sequences S, each of length 200, over an alphabet Ω_o of size 500. We additionally consider between 0 to 6 generalized events $\alpha \in \Omega_g$,

3 `https://eda.rg.cispa.io/prj/flock/`

each of which, unless stated otherwise, consists of five observed events $e \in \Omega_o$ that are sampled uniformly at random. We plant patterns of length 10. We ensure 10% of all planted instances are interleaved in the data, meaning the next pattern starts before the last one ends. We ensure that each planted instance does not collide (overwrites) with an earlier planted instance.

All results on synthetic data are averaged over ten independently generated datasets. In terms of runtime FLOCK is comparable to SQS and SQUISH; for all reported experiments all three finish within seconds to minutes. The competing methods take much longer: for SKOPUS, which is a top-k method, we had to set k to 50 and limit the pattern length to 10 to keep the run-time under 24h.

We evaluate the reported pattern sets with standard $F1$ score, we again follow the flow network based approach introduces in Chapter 2. We do the same for generalizations, where we allow a mapping if the planted is a subset of reported generalization, or vice versa.

First, as a sanity check, we run FLOCK on data without any structure, i.e. no planted patterns. FLOCK correctly reports no patterns. Next, we consider the setting where we start with 30 independent patterns without any generalization, and in each subsequent experiment we replace five of these with one generalized pattern p, where one event in p is a generalized event. Colloquially speaking, we answer the question: "How well does FLOCK pickup generalized patterns compared to surface level patterns?". We show the results in Figure 5.2a. We observe in the initial setting without any generalization we are on par with SQS and SQUISH, the *Word2Vec* based approaches are next best, while ISM and SKOPUS perform worst. However, as we increase the number of generalized patterns, FLOCK maintains a high $F1$ score throughout while the score of all other methods decreases significantly.

Next, we consider a more difficult setting. We sample five generalized events and plant 5 patterns each containing 2 generalizations. Note that this means different patterns share the same generalization. To investigate performance under decreasing support, we decrease the total number of planted pattern instances from 400 to 50 in steps of 50. This setup aims to answer the question: "How frequent have patterns to be to be discovered?". We show the results in Figure 5.2b. We observe that FLOCK beats the other methods by a wide margin, with SQS and SQUISH in second place. FLOCK performs very well up to 100

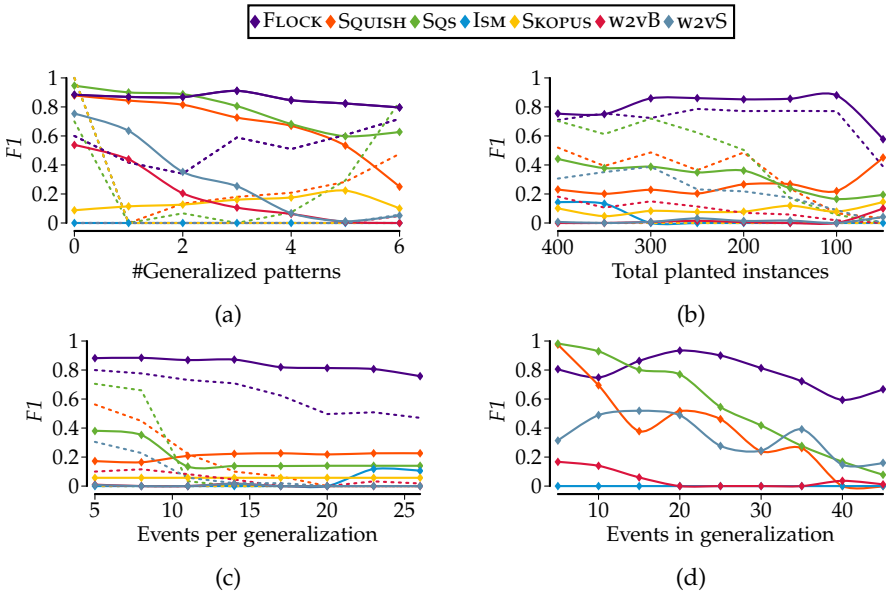


Figure 5.2: F1 score for recovery of planted patterns (solid line, Fig. a-c) on synthetic data over (a) number of patterns containing one generalized event, (b) total number of planted pattern instances, and (c & d) number of observed events per generalized event. F1 score for recovery of generalized events (dotted line, Fig. a-c). In Plot (d) we evaluate (F1 score) the recovery of events per generalization. Overall we see that FLOCK beats the other competitors by a wide margin.

planted instances, at 50 the score drops significantly; as an individual instance of a pattern on expectation then only occurs 0.4 times this is unsurprising.

To test how FLOCK behaves when the pattern frequency stays the same, but the individual instances get less frequent, we consider the case where we increase the number of events per generalization (cf. Figure 5.2c). We observe a very wide margin to all other methods. An increase in generalization size only has a very small effect on FLOCK's ability to recover the planted patterns.

Finally, we evaluate the quality of the reported generalizations (cf. Figure 5.2d). To do so we generate data containing two patterns, sharing one generalization. To see how well we recover the generalization if some events are much less frequent, we decrease the usage of events in the generalization linearly to zero, and in each subsequent experiment we increase the number of events per generalization. With that, we aim to answer the question: "How accurately does a generalized event get recovered?". We again report the $F1$ score, this time computed over how well the individual events within the generalization are recovered. We omit SKOPUS from this experiment as a single run did not terminate within 24h. We see that FLOCK recovers the generalization well, while SQUISH and SQS do well in a simpler setting, they are however not robust against larger generalizations, unlike FLOCK.

On the synthetic data experiments we have seen that FLOCK outperforms all other methods clearly. In some simple settings SQS and SQUISH are on par with FLOCK. The WORD2VEC approaches only do reasonably well on data with no or very few generalizations, inspecting the results this is likely mostly due to SQS. ISM and SKOPUS do worst throughout the experiments.

5.6.2 *Real-World Data*

To evaluate if FLOCK finds meaningful structure in real-world data, we test FLOCK on five distinct datasets, electrocardiograms (*ECG*),[4] a business event log (*BPI-2015*),[5] a rolling mill production log (*Rolling Mill* [187]), and two text datasets (*JMLR* and *Moby* [170]). We compare FLOCK to the two best performing competitors, SQS and SQUISH. Since

4 https://physionet.org/content/stdb/1.0.0/

5 https://data.4tu.nl/collections/BPI_Challenge_2015/5065424/1

		FLOCK		SQS		SQUISH	
Dataset	$\lvert\Omega_o\rvert$	$\lvert P\rvert$	$\lvert\Omega_g\rvert$	$\lvert P\rvert$	$\lvert\Omega_g\rvert$	$\lvert P\rvert$	$\lvert\Omega_g\rvert$
ECG	200	4	1	128	12	88	6
Short-ECG	200	3	2	3	0	4	1
BPI-2015	192	189	53	400	6	525	35
Rolling Mill	836	195	56	430	35	554	73
Moby	10276	239	1	231	0	202	26
JMLR	3845	466	5	580	0	480	87

Table 5.1: Results on real datasets. We given alphabet size $|\Omega_o|$, and the number of reported patterns $|P|$, and number of generalization $|\Omega_g|$, for each method. Overall we observe that FLOCK reports fewer patters and more generalizations, making the model easier to interpret.

the *ECG* and text datasets have a very low amount of structure we set $n = 2$. We run all three methods on each dataset and report the number of discovered patterns and generalizations in Table 5.1.

First, we consider the *ECG* dataset, we note that compression rates are similar but there is a big difference in the number and quality of patterns. FLOCK can capture the key structure in just four patterns, while SQS reports 128 patterns and SQUISH reports 88 patterns. As this dataset contains enough events such that each individual instance is still strongly represented we reduced the number of events drastically, from 100k to 3k (*Short-ECG*). We find that SQS and SQUISH report shorter patterns than FLOCK, FLOCK is able to discover generalized events enabling it to find longer patterns over this extended alphabet.

The next two datasets we consider are event logs (*BPI-2015* and *Rolling Mill*), characterized by strongly repetitive and structured behavior. FLOCK finds patterns that describe a more general behavior which we do not observe for the other methods. To demonstrate that FLOCK discovers generalizations with strong dependencies between each other we show a pattern discovered on the *Rolling Mill* dataset in Figure 5.3. The instance of α has a strong influence on β while β determines the value of γ and finally the value of γ has a strong influence on the value of δ, see zoom box. This pattern covers 14 possible

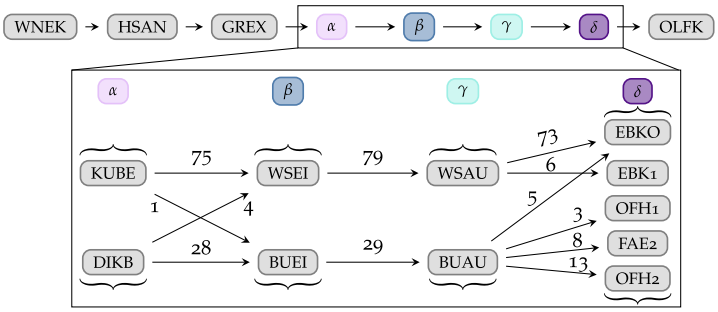


Figure 5.3: Example pattern discovered by FLOCK on the *Rolling Mill* dataset, of length 8 out of which 4 are generalized events ($\alpha, \beta, \gamma, \delta$). We see the value of the previous generalization strongly influences the next generalization (see zoom box): e.g. if α has value "KUBE", β has value "WSEI" in 75 of 76 cases.

instances within one pattern, including rare instances where the usual procedure is not followed.

Finally, to see how well FLOCK handles settings with large alphabets we consider text data. We consider a set of abstracts from the *JMLR* journal, and the novel *Moby* Dick by Herman Melville. For the *JMLR* dataset, we see that FLOCK reports fewer patterns than SQS and SQUISH while capturing the same amount of structure, allowing for a more interpretable representation (see Table 5.1). We show a selection of the patterns discovered by FLOCK on *JMLR*, *Moby* and *BPI-2015* in Appendix d.2.

5.7 DISCUSSION

The experiments on real-world data show that FLOCK performs well in practice. It recovers surface-level patterns as well as the state of the art, but additionally is also able to recover ground-truth generalizations, generalized patterns, as well as the dependencies between generalizations within patterns. The models that FLOCK discovers are smaller, less redundant, and the more expressive patterns it discovers provide clear insight into the data-generating process.

It is worth commenting on interpretability. A single surface-level pattern is arguably easier to interpret than a generalized pattern, and if matching the ground truth, so is a set of surface-level patterns com-

pared to a set of generalized patterns. One of the key strengths of FLOCK is that its MDL objective will automatically determine if it is better to model the data at hand with surface-level or generalized patterns; for the former, the experiments show that it is as able as SQS [170], ISM [57], and SQUISH [11] in discovering true surface-level patterns, while it is unique in its capability to discover generalizations that allow it to show the forest for the trees.

As good as its results are, we consider it very interesting direction to explore how to incorporate background knowledge in the form of a given ontology, a set of generalizations, i.e. sets of surface-level events that we know or expect should behave similarly, or a similarity matrix over events. To a certain extent, our current problem formulation already allows for this: we can initialize FLOCK with a model M that includes the corresponding generalizations. It is, however, not immediately clear how to best continue from there; should FLOCK consider these given generalizations as immutable parts that cannot be pruned, or as a suggestion that can be refined?

5.8 CONCLUSION

We considered the problem of summarizing an event sequence database with generalized sequential patterns. To that end, we introduced the concepts of generalized events and generalized sequential patterns. To find succinct and non-redundant models, we formalized the problem using the Minimum Description Length principle, and presented the efficient FLOCK algorithm to find good pattern sets in practice.

Experiments on synthetic and real world data showed that FLOCK works well in practice and provides insight beyond what is possible with existing surface-level pattern mining methods, even with post-processing. To further improve FLOCK we plan, as future work, to study how to best incorporate background knowledge as well as how to scale up through continuous optimization based search.

6

SYNTHETIC NETWORK FLOW GENERATION THROUGH PATTERN SET MINING

In the past chapters we studied methods for event sequences in general. In this chapter, we study a specific data type of event sequence, namely network flows recorded over time. A network flow captures one connection e.g. one TCP session will be captured as one flow. We propose a domain specific pattern language and learn a summarization of the network flow data. We then use this learned summarization to generate new synthetic network flows. We compare our approach to state-of-the-art methods and compare generated data on several criteria, namely realism, diversity, compliance, and novelty.

6.1 INTRODUCTION

Evaluating network security tools, such as intrusion detection systems (IDS) [171] or firewalls [67], and conducting network measurement campaigns, such as applications testing [75] or device identification [130], requires the systematic collection and sharing of network traffic datasets.

However, multiple studies have highlighted recurring issues with network traffic datasets, such as quality [97], density [93], and labeling accuracy [72]. In fact, instead of using actual network captures, which may not be shared due to confidentiality and privacy risks [93], researchers have proposed to generate traffic in a controlled environment. This synthetic traffic does not result from human network activities but from network automatons, such as web crawlers [151], email generators [75], or bots that operate specific applications following predefined user profiles [161]. Simulating traffic effectively sidesteps the issues associated with human-generated traffic [151]: e.g., the risk of

This chapter is based on [33]: Joscha Cüppers, Adrien Schoen, Gregory Blanc and Pierre-Francois Gimenez. "FlowChronicle: Synthetic Network Flow Generation through Pattern Set Mining." In: *Proceedings of the ACM on Networking 2 CoNEXT4*, ACM, 2024, Article No: 26.

leaking confidential information is minimized since the *users* are simulated and not real. Furthermore, the controllable behavior of these simulated users allows easier labeling of the resulting traffic compared to traffic generated from human interactions [161].

One major drawback of traffic simulation is scalability: once the simulation is launched and the traffic is being recorded, the behavior cannot be adapted to a new constraint that was not implemented at the starting time. The resulting traffic cannot be adapted to produce unplanned behavior and, therefore, hardly corresponds to another configuration in terms of hosts or activities [1]. Such adaptation will often necessitate re-running the entire simulation, which is time-consuming and costly. To address the issues of both real and simulated traffic, the research community has resorted to synthetic data generation, which relies on a modeling algorithm that learns existing traffic characteristics to reproduce them [1, 5, 129]. It is expected that such algorithms enable the generation of new traffic to assess the generalization of network security measures to new environments [1] while preserving consistency/compliance [125].

This article focuses on the generation of benign traffic in the format of *network flows*. For this specific task, several classes of synthetic data generation models have been proposed so far, including *Generative Adversarial Networks* (GAN) [5, 6, 13, 136, 150], *Variational AutoEncoders* (VAE) [64, 119], autoregressive model [195], and Bayesian networks [157]. A problem these models have with network flow generation is their lack of modeling the temporal dependencies among flows [5]: for example, several works [6, 150, 157] are sampling network flows independently. We argue that this type of generation is insufficient for real-life applications.

We therefore propose *FlowChronicle*, a novel synthetic network flow generation method based on pattern mining [4, 180]. Our first contribution is a powerful pattern language especially designed to match network flows, that not only captures relevant value combinations within flows but also between different network flows. We formalize this problem as mining a set of non-redundant patterns that best summarize the training network flow dataset.

Our second contribution is a data generation mechanism based on the learned representation. As we will show, this approach generates highly realistic network traffic and respects the protocol specifications.

Moreover, since *FlowChronicle* generates synthetic data directly from the temporal patterns mined from the training dataset, this results in synthetic traffic that preserves temporal relations among network flows. In addition, the patterns mined by our generating model are interpretable and auditable.

Finally, we compare our model to other state-of-the-art generators to show that our model, on top of providing realistic synthetic traffic, also preserves time dependencies between flows.

The remainder of this chapter is structured as follows: we introduce preliminaries in Section 6.3. We detail our contributions, namely the pattern language and the network flow generation method in Sections 6.4, 6.5 respectively. The evaluation against other generators and their results are discussed in Section 6.6, before concluding in Section 6.8.

6.2 RELATED WORKS

Although many statistical models can be used for synthetic data generation, in recent years deep-learning-based methods have been increasingly favored due to their ability to generate high-dimensional data such as images. Especially GAN [68], VAE [94], and Transformers [147, 175] have been shown to produce highly realistic synthetic data. These methods have also been applied to generate multiple types of network traffic data, including raw packet contents [24, 46, 76, 116], sequences of headers [160, 199], flow features [6, 107, 113, 150, 157, 199] or even temporal series of features [82, 106]. In the following, we will focus on methods for synthetic network flow generation.

The first use of these generative methods for creating synthetic legitimate network flow generation has been proposed by Ring et al. [150]. It was quickly followed by Manocchio et al. [113] who have shown that, when applied to network traffic, WGAN-GP [73] is prone to a phenomenon called *mode collapse*, which is when the generated data only cover a part of the training data distribution. Anande et al. [6] and Bourou et al. [13] have shown that synthetic data generation methods for tabular data [192] also work well for network flow datasets.

The major limit of these solutions is that while the generation preserves the dependencies across network features within a flow, it does not consider the dependencies among the flows. For example, before

establishing an HTTP connection, a client might have to reach a DNS server to resolve the domain name of the requested website. That is, a single action of the client will, hence, lead to two flows, one to the DNS server and the other to the website host. All the previous solutions, due to sampling new network flows independently, do not model those inter-flow dependencies [5]. To solve that issue, Xu et al. [195] implement a solution that not only model dependencies within one network flow, but also dependencies across network flows by using an autoregressive model. Lin et al. [106] propose a different approach where they model the problem of network traffic generation as a temporal series generation, where a multidimensional time series represents the activity. This method was then adapted by Yin et al. [199] to generate complete network flows. Recently, Schoen et al. [157] have shown that this solution does not generate realistic network flows and tend to produce flows that do not comply with basic network-specific checks. Therefore, generating realistic network flows that also include temporal dependencies remains an open challenge.

We propose a different approach based on pattern set mining. Pattern set mining has been shown to be able to learn representations suitable for synthetic data generation [179]. Existing pattern set miners, including our own, do not directly address our needs. We hence introduce our own novel pattern language that can model transactional relations, i.e., structure in a flow, as well as sequential relations, i.e., structure between flows. Researchers have successfully used pattern mining for various tasks related to network data, such as near-live network monitoring [104], efficient computation of heavy hitters [134], and to provide succinct visualization of network flow traces [65]. Most works use pattern mining in the context of anomaly detection: Jakhale and Patil [83] build on the work of Li and Deng [104] to detect anomalous flows. In contrast, Brauckhoff et al. [15] use frequent pattern mining to summarize flows that cause anomalies. Paredes-Oliva et al. [135] propose to classify extracted patterns as either anomalous or not, in contrast to individual flows or time intervals. Unlike us, none consider patterns over multiple flows, and all use frequent pattern mining. To the best of our knowledge, we are the first to use pattern mining to generate synthetic network flows.

6.3 BACKGROUND

In this section, we present some background knowledge we rely upon for *FlowChronicle*, in particular, related to pattern mining and machine learning.

6.3.1 *Network flows*

Network traffic encompasses all the packets that are exchanged among hosts within a specific network over a designated time frame. A *network flow* is an abstraction that describes a sequence of packets that share five common key attributes: *source IP address*, *destination IP address*, *source port*, *destination port*, and *transport protocol*. Two scopes of network flows exist: unidirectional and bidirectional. Unidirectional flows only contain packets sent from the designated source to the designated destination, while bidirectional flows group packets in both directions. A network flow record embeds the statistical data related to the communication identified by the 5-tuple, such as the *Duration* of the communication or the *Number of Bytes* transmitted. Such extra features depend on the network flow format, and several competing formats exist. In the following, for brevity sake, "network flow record" will be abbreviated to "flow".

6.3.2 *Bayesian networks*

Bayesian networks are a class of generative statistical models that represent probability distributions [138] and are widely used in statistics. Syntactically, they are described as a directed acyclic graph, where each node is a random variable, and a set of conditional probability tables. These conditional probability tables, one per node, describe the probability of their associated node depending on the values of its parents. Due to their structure, Bayesian networks are considered to be explainable models: the edges between nodes are indicative of their statistical correlations.

6.3.3 *Notation*

In the following, we denote F each feature of the network flow (such as source IP, protocol, etc.) and $\underline{F}$ its domain, i.e., the set of possible values for F. Let us denote n the number of features. A flow f is simply a tuple of the n features: the domain of flows is therefore $\underline{F}_1 \times \underline{F}_2 \times \cdots \times \underline{F}_n$. We will typically use the letter t to denote timestamps. Finally, a dataset D is a sequence of timestamped network flows (t, f).

6.4 PATTERN LANGUAGE OF FLOWCHRONICLE

In this section, we formalize the language of patterns that are identified by *FlowChronicle*.

6.4.1 *Intuition*

Given a dataset D of network flows, we aim to identify patterns that can describe which combinations of flows occur frequently in D. Some patterns can concern values inside a flow. For example, destination port `53` is frequently associated with protocol `UDP`, intuitively, we could use a pattern to automatically complete the protocol given the destination port. Some patterns can also concern several flows. For example, HTTP(S) requests are typically preceded by a DNS request. Similarly, an IMAP request (to read emails) can be followed by HTTP(S) requests if URLs of images are present in an email. For this reason, our patterns can span over multiple flows.

Classical pattern mining searches for deterministic relations, e.g., destination port `53` and the UDP protocol. We consider they are not sufficient to properly encompass network flows dependencies. For this reason, we propose to also include in our pattern statistical relations. For example, if a machine has both an SSH and an HTTPS server, then when this machine is contacted, the destination port will probably be `22` or `443` but not any other port.

6.4.2 *Pattern language*

A pattern is composed of two parts: the partial flows, to mine discrete temporal dependencies, and the dependency structure, to mine statis-

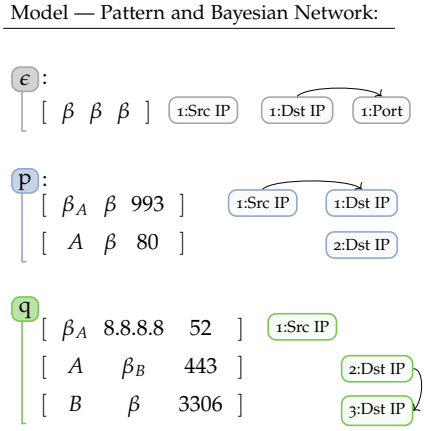
tical temporal dependencies. Since a pattern can span across several flows and can specify the values of features for these flows, we need to store this information. For that purpose, patterns contain a table, called the partial flows, where columns are the network features and each row corresponds to a flow. With *cell* we refer to a cell on this table.

Each cell of the partial flow can be one of three types. Firstly, there are *fixed* cells: these are cells which values are directly defined in the partial flows. For example, the first partial flow could have destination port 53 and the second, destination port 443. Secondly, there are *free* cells: these are the cells not defined by the partial flows, and their values are determined by the dependency structure (described below). Lastly, there are *reuse* cells: the values of these cells are equal to the values of other cells in preceding partial flows of the pattern. A common illustration of this type of cell could be the Source IP address of a first flow that is reused as the Destination IP of a subsequent flow.

Because some cells can be free, each pattern also contains a *dependency structure*. The dependency structure is a Bayesian network that represents the joint probability distribution of the free cells.

More precisely, a pattern p is a tuple (Z, BN), where Z is a sequence of partial flows, and BN is the Bayesian network representing the dependency structure between free cells. We write $Z[j]$ to denote the j^{th} partial flow. Each cell of a partial flow can be either fixed (i.e., associated with a $f_i \in \underline{F_i}$), free (denoted β, for "Bayesian") or reuse (denoted with an uppercase identifier). Besides, free cells can be marked for reuse. This is denoted by adding an uppercase identifier in subscript to β. For example, the value of β_A will be used for the reuse cell denoted A. If an identifier is defined in $Z[j]$, then it can only be used in later partial flows, i.e., in $Z[k]$ such that $k > j$. BN is a Bayesian network defined over all the free cells in partial flows, i.e. all $\beta \in p$.

Examples of patterns are shown in Figure 6.1. Three patterns are defined: ϵ has one partial flow with only free cells. Pattern p has two partial flows. The identifier A in the reuse cell is used to ensure that the source IP is the same in both flows. The ports are fixed while the IP addresses are free. Finally, pattern q has three partial flows. The identifiers A and B and the reuse cell ensure that the source IP of the second flow is equal to the source IP of the first flow and that the source IP of the third flow is equal to the destination IP of the second flow.



Data and Pattern Windows:

Time	Src IP	Dst IP	Port
12	134.96.235.78	142.251.36.5	993
56	134.96.235.129	8.8.8.8	52
89	134.96.235.78	212.21.165.114	80
113	134.96.235.129	198.95.26.96	443
145	198.95.26.96	198.95.28.30	3306
156	134.96.235.78	134.96.234.5	21
178	134.96.235.36	185.15.59.224	993
206	134.96.235.36	128.93.162.83	80

Figure 6.1: Toy example: On the left side we show a model with 3 patterns, on the right side we show the dataset and how the patterns of the model cover the dataset. The three patterns are: ϵ has one partial flow with only free cells (β). Pattern p has two partial flows. The identifier A in the reuse cell is used to ensure that the source IP is the same in both flows. The ports are fixed while the IP addresses are free. Finally, pattern q has three partial flows. The identifiers A and B and the reuse cell ensure that the source IP of the second flow is equal to the source IP of the first flow and that the source IP of the third flow is equal to the destination IP of the second flow.

6.4.3 *Dataset cover*

To select the best model able to capture the patterns in a dataset, we need to compute the term $L(D|M)$. For this, we have to use the patterns to compress the data, i.e., *cover* the datasets with patterns and encode the locations of the patterns and the values of their non-fixed values. To properly define this cover, we first define a window of a pattern p, which indicates for each partial flow of p an index in the data, such that the partial flow matches the flow at that index in the data.

For example, in Figure 6.1, pattern p is associated to two windows, (12,89) and (178,206), and pattern q is associated to the window (56,113,145). Remark that the fixed values of the pattern always match the data. Be-

sides, the reuse cells also match: for example, in the window (12,89) of pattern p, both source IP are indeed identical.

A *dataset cover* is a set of windows such that all flows of the dataset are associated to exactly one window. To ensure that this is always possible, we define a "catch-all pattern", denoted ϵ (see Fig. 6.1), that has only one partial flow and whose cells are all free. ϵ is called the *empty pattern*. Remark that multiple covers can explain the data for a given set of patterns, and that finding the optimal cover is a NP-hard problem [91]. For this reason, we use a greedy algorithm to find a cover.

6.4.4 *Model encoding*

Now that we have given the intuition, we formally describe our MDL encoding, which has two parts: model encoding and data encoding given a model. We start with the model encoding.

As model M we consider a set of patterns, we need to encode the number of patterns, and each pattern. Hence,

$$L(M) = L_{\mathbb{N}}(|M|) + \sum_{p \in M} L(p) \quad ,$$

where $|M|$ refers to the number of patterns in M. We encode $|M|$ using the MDL-optimal encoding for integers $z \geqslant 1$ [153].

To encode a pattern p, we first encode the number of partial flows a pattern contains, then all the partial flows, and finally the Bayesian network,

$$L(p) = L_{\mathbb{N}}(|p|) + \left(\sum_{j=1}^{|p|} L(Z[j]|p) \right) + L(BN_p) \quad .$$

To encode the number of partial flows we again use the MDL-optimal encoding for integers.

We split the partial flow encoding into three parts: we encode 1) the fixed cells, 2) which free cells that are marked for reuse, and 3) if there are values marked for reuse in earlier flows, where and if we want to use them. We will explain the encoding in turn.

We start by encoding for how many of the n flow features we want to encode a fixed value, with $\log n$ bits. To select which k cells, we require

$\log \binom{n}{k}$ bits. Finally, we encode the respective values by choosing one value from the respective domain. To encode which cells we mark for reuse, we first encode how many, out of the $n-k$ remaining, and then, choose the l cells we want to mark. Finally, we encode for which cells we want to reuse values: we again encode how many of the remaining $n-k-l$ and choose which m cells. For each of the m selected cells, we select which of the previously marked cells we reuse. Formally this is,

$$L(Z[j]|p) = \log(n) + \log\binom{n}{k} + \left(\sum_{i \in R_j} \log|\underline{F}_i|\right) +$$
$$\log(n-k) + \log\binom{n-k}{l} +$$
$$\mathbb{1}(|\pi(j,p)| > 0)\Big(\log(n-k-l) + \log\binom{n-k-l}{m} + m\log(|\pi(j,p)|)\Big),$$

where R_j is the set of all features with a fixed cell, and $\pi(j|p)$ is a set of all cells marked for reuse before the j^{th} flow.

We encode the Bayesian network by encoding for each node its number of parents c, in $\log K$ bits, where K is the maximum number of parents passed as a parameter to the learning algorithm, and then select the parents out of all $|B|-1$ possible parents, where B is the set of free cells described by the Bayesian network (formally $B = \{(j,i) \mid Z[j]_i = \beta \vee Z[j]_i = \beta_A\}$). So:

$$L(BN_p) = \sum_{(j,i) \in B} \log K + \log\binom{|B|-1}{c_{j,i}} \quad .$$

We do not encode the conditional probability tables, as explained in the next subsection.

6.4.5 *Data encoding given a model*

Now that we know how to encode a model M, we describe how to encode D using a model M. We define the encoding of D by M as the *cover* of D. Its length in bits is $L(D|M)$. Because M is a set of patterns, the cover is a set of pattern windows. In Figure 6.1, we show a toy example of a dataset, a model and its cover. Before we define how we encode a dataset D, using a set of patterns, let us give the intuition by

Code Sequence — encoding of data with model:

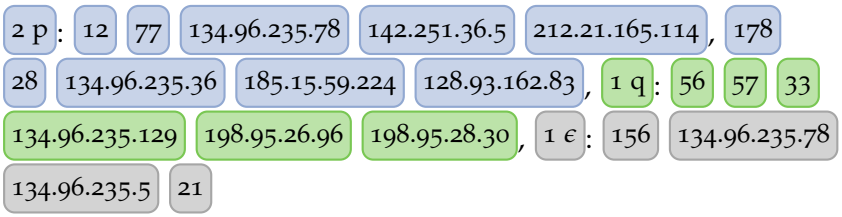


Figure 6.2: Encoding of the data shown in Fig. 6.1, i.e. how the data is described using the model.

describing how we decode a dataset from a given cover. More precisely, a cover is a sequence of codes, encoding how often each pattern occurs, where these occurrences are, and the values of the free cells.

In Figure 6.2, we show the sequence of codes corresponding to the toy example in Figure 6.1. To decode the cover, we start by reading the first code from the cover, in our toy example (2 p), indicating that we use pattern 'p' twice in the cover. Next, we read for each partial flow in the pattern the codes corresponding to the timestamps (12) and (77), the first one being the start time, and the second one the delay to the first partial flow. Next, we read for each free cell one code—decoding the values. We repeat these steps for the second occurrence of pattern 'p'. We do the same for pattern 'q'. Finally, code (1 ϵ), tells us there is one flow is covered by the empty pattern. We read again the timestamp code as well as the encoded values. With that, we have fully decoded the cover.

Now that we have seen how data encoding works, we will formally describe how many bits we need to encode it. For each p, we encode how often we want to use it, i.e., the number of windows in the cover. We then encode each window. Formally this is:

$$L(D \mid M) = \sum_{p \in M} \big(L_{\mathbb{N}}(|W_p|) + L(W_p)\big) \quad ,$$

where W_p denotes the set of windows of pattern p used to describe D.

The length of W_p is the timestamps plus the free cells, encoded based on the probabilities given by the Bayesian network,

$$L(W_p) = \sum_{i=1}^{|W_p|} \left(L(t_1 \text{ of } w_i) + \sum_{k=2}^{|p|} L(t_k \text{of } w_i \mid t_{i-1}) \right) - \log(Pr(w_i | BN_p, \{w_j | j < i\}))$$

where $L(t) = \log(t_{\max} - t_{\min})$ and $t_{\max}$ (resp., $t_{\min}$) refers to the maximum (resp., lowest) timestamp in the data D, $L(t_i \mid t_j) = L_{\mathbb{N}}(t_i - t_j)$. As we expect lower delays between flows, we chose $L_{\mathbb{N}}$ to encode the difference between time points. It closely follows a geometric distribution, hence giving higher probability mass to lower delays, and thereby requiring fewer bits for those.

To avoid having to encode the contingency table of the BN and make arbitrary encoding choices in the process, we use prequential codes [71](see Chapter 4 Section 4.3).

6.5 ALGORITHM

In this section, we present the whole generation pipeline: data preprocessing, pattern identification—the previous section describes the MDL loss used to choose a model but does not explain how to find the candidate model—and data sampling from the selected model.

6.5.1 *Preprocessing a network flow dataset*

Network flow data are tabular data, where each network flow is a line in the table, and each feature is a column. The features are either categorical (e.g. *Transport Protocol*), or continuous (e.g. *Duration of the flow*). To mine patterns in that tabular dataset, we first need to discretize the numerical features in our network flow description. Similar to Schoen et al. [157], we discretize the numerical features into 40 categories, such that each category contains the same number of samples.

6.5.2 *Pattern Miner*

In this section, we explain how to discover a good model and a description of the data under a given model M. We begin with the latter.

6.5.2.1 *Finding a cover*

Given a model M, we want the cover C that minimizes the encoding cost $L(D \mid M)$. Finding the optimal cover is a NP-hard problem, so we propose a greedy method. For this, we have to find out where we can use a pattern p, i.e., we have to find the windows of p. We only consider minimal windows, i.e., windows for which there does not exist a window $\overline{w}(p)$ whose interval $I(\overline{w}(p))$ is a proper sub-interval of $I(w(p))$. We sort all windows by 1) the number of covered flows (decreasing), and 2) inter-flow delays (increasing). Note the empty pattern ϵ is defined to cover 0 flows, i.e., it should only be used to cover flows that are not covered by any other pattern. Finally, we greedily add windows to the cover until all flows are covered. If one window overlaps with precedent windows, we skip it. With that, we have a description of D in terms of pattern.

6.5.2.2 *Iterative Pattern Search*

The general approach is an iterative search procedure: at each iteration, we generate pattern candidates and test if these candidates help in reducing the description length. If so, we add them to our model. At each step of the search, we ensure no source or destination IP has a fixed value. Indeed, we do not want to learn the behavior of specific IP addresses. Besides, we restrict reuse cells and cells marked for reuse to only be source IP or destination IP.

CANDIDATE GENERATION The initial set of candidates is created as follows: for each pair of features, and for each combination of values of these two features (except the source IP and destination IP), we create a pattern with a single flow with two fixed cells. The rest of the cells are free and described by a Bayesian network.

During the search, we build new candidates by extending existing patterns. Given a pattern, we have three different ways to generate new candidates: 1) by directly creating a fixed cell, either from a previously free cell or by adding a new row of free cells and transforming one into a fixed cell—once again, this fixed cell cannot be a source IP or destination IP; 2) by merging existing patterns. If both patterns have only one partial flow and have no conflicting fixed cells, we merge them into a new single-flow pattern. For patterns over multiple flows,

Algorithm 13: *FlowChronicle*

Input : set of flow D,
continous misses threshold t
Output: model M and Cover of D

$M \leftarrow \{\epsilon\}$
$C \leftarrow$ all pairwise combinations
mode $\leftarrow$ *single-flow*, misses$\leftarrow 0$
while misses$< t$ or mode $=$ *single-flow* **do**
  $C \leftarrow C \cup \textsc{BuildCandidates}(M, \text{mode})$
  $c \leftarrow \arg\max_{c \in C} cs(c)$
  if $L(D, M) > L(D, M \cup c)$ **then**
    $M \leftarrow \textsc{Prune}(M \cup c)$
    misses$\leftarrow 0$
  else
    misses$\leftarrow$ misses$+1$
  if misses $> t$ and mode $=$ *single-flow* **then**
    mode $\leftarrow$ *multi-flow*
    misses $\leftarrow 0$
return M, C

we create candidates by appending them; 3) by transforming a free cell into a reuse cell. Such cells can only appear in multi-flow patterns because a reuse cell can only reference a marked cell from previous partial flows. This reuse cell can reference previously marked cells or mark new previous cells to reference them.

CANDIDATE SCORE As testing all candidates is not feasible in a reasonable time, we want to test the most promising candidates first. To this end, we derive a candidate score. The candidate scores capture how many values a pattern can cover: the number of non-overlapping windows multiplied by the number of fixed or reuse cells in the pattern.

MINING A MODEL We show the pseudocode of *FlowChronicle* in Algorithm 13. We initialize our model with the empty pattern. We start our search with the initial set of patterns. The basic idea is to take the best candidate c according to the candidate score $cs(c)$, and if it reduces $L(D, M)$, we add it to the model. If a pattern fails to reduce $L(D, M)$, we will not test it again in future iterations[1].

As testing all candidates is not feasible, we propose an early stopping criteria. We propose to stop when we exceed a *consecutive misses* threshold t. This threshold is defined by the user.

To avoid building uninformative patterns spanning many rows, we begin by searching for patterns within single flows. In practice, we do that by only generating single-flow candidates. We continue this until we surpass the *consecutive misses* threshold; at this point, we reset the *consecutive misses* threshold and also allow the construction of candidates that cover multiple flows.

Adding a new pattern to M can make existing patterns redundant. We hence prune redundant patterns by testing for all patterns p where the usage in the cover has been reduced: if $L(D, M \backslash \{p\}) < L(D, M)$, we remove it from the model M.

6.5.3 *Parallelization*

To improve run-time on larger data sets, we propose to split the preprocessed data into n chunks. We learn independently a model for

1 For better readability, we omitted this part from the pseudocode.

each chunk, so each chunk can be processed in parallel. The learned models then capture local characteristics and provide a cover of the respective chunk. Since each model is a set of patterns and the corresponding BN, we can simply take the union of all models resulting in a new model for the entire dataset. The cover of the entire dataset can be constructed by appending all individual covers. Finally, we relearn the BN of the empty pattern (the pattern used to cover all flows not covered by any other pattern).

6.5.4 *Synthetic data generation*

Once we have a set of patterns and the cover, we can use them to generate a synthetic dataset. From the cover, we can learn the probability distribution over the usage of each pattern (including the empty pattern). To generate data we sample patterns form this distributions. Given a sampled pattern, we sample the initial timestamps for each pattern. As some patterns might occur more frequently during some periods (e.g., fewer emails during lunch, more OS updates in the morning, etc.), and to not make any assumption about the shape of the distribution, we estimate the frequency over time via a *Kernel Density Estimation* (KDE). We then sample the timestamps from this distribution. For patterns with multiple partial flows, we estimate a distribution of the delay between consecutive partial flows, again with KDE, and sample from this distribution. Finally, we have to fill the cells of all sampled pattern occurrences. Cells with a *fixed* value are already set. For the free cells, we sample from the Bayesian network BN associated with the pattern. Finally, we set the values of *reuse* cells. This completes the generation of a synthetic flow dataset.

6.6 EVALUATION METHOD

Our goal in this part is to provide an evaluation framework to compare the generated data of different models. This evaluation will be twofold: we will first evaluate the different generated flows independently, without any consideration for temporal dependencies, and second, we will study how well the generation preserves temporal dependencies present in real data.

6.6.1 *Independent evaluation*

Independent evaluation involves analyzing the network flow distribution generated by a model and comparing it with the training data to determine if the model has captured the essential characteristics needed to create new data. For each evaluated model, we compare its synthetic network flow distribution with the real network flows from the *week-3* dataset (the training dataset). As described by Schoen et al. [157, 158], this comparison should elucidate four key attributes of the generated data: **Realism**, **Diversity**, **Compliance**, and **Novelty**. Realism ensures that a generated network flow should be sampled from the same distribution as the real network flows. Diversity ensures that the generated network flows should cover the entire real network flow distributions. Compliance refers to the criterion that checks if the generated network flow adheres to specific network rules. Lastly, Novelty ensures that the generated network flows are not mere replicas of the real network flows.

REALISM Similarly to Schoen et al. [157], we evaluate the realism of the joint distribution with the Density metric [124], the realism of the conditional probability distributions with CMD (Contingency Matrix Difference) and PCD (Pairwise Correlation Difference). CMD is the difference between the generated data's correlation matrix and the training data's correlation matrix (for numerical features). PCD is the difference between the generated data's contingency matrix and the training data's contingency matrix (for categorical features).

DIVERSITY Similarly to Schoen et al. [157], we evaluate the diversity of the joint distribution with the Coverage metric [124]. A low score indicates that the generated distribution does not cover the entire training distribution. We also evaluate the marginal distributions of each features. with the JSD (Jensen Shannon Divergence) for categorical features and the EMD (Earth Mover's Distance) for numerical features.

COMPLIANCE Compliance is the property that the generated network flow should respect network protocol specifications. For example, a generated UDP flow should not contain any TCP flags. We eval-

uate this property with the DKC (Domain Knowledge Check) [150], a succession of boolean tests for the generated network flow, each test representing one property that we want to enforce in the generation. We use the implementation proposed by Schoen et al. [157]. A lower DKC means fewer tests have failed and a more compliant generation.

NOVELTY We evaluate the novelty similarly to Schoen et al. [157] with he Membership Disclosure (MD) metric. This metric evaluates the privacy risk of synthetic datasets generated by models trained on real datasets. It involves comparing the synthetic samples to the training and testing sets from the original data by computing the Hamming distances between each pair of generated and real samples. When a synthetic sample is sufficiently similar to a real sample (i.e., the Hamming distance is below a certain threshold), the real sample is considered a potential leak from the training set. By varying the threshold, a detection method for training samples is established, and the effectiveness of this detector is measured using the F1-score. The overall privacy risk is quantified by integrating the F1-scores over all possible threshold values. In a network context, very similar flows like DNS or NTP requests are not uncommon, so the level of novelty in synthetic data should mirror that of a reference set of real data.

6.6.2 *Preservation of temporal correlation*

The above metrics only account for individual network flows, but we seek to generate data that preserves temporal dependencies, so we also need to evaluate this aspect. We propose to use feature-wise metrics to assert whether a generated dataset preserves the temporal dependencies present in the training set. We will consider the numerical features on one hand and the categorical features on the other.

NUMERICAL FEATURES Inspired by several papers [109, 128, 131, 165], we evaluate the temporal dependencies between a generated dataset and a training dataset by comparing the Autocorrelation Functions (ACF) of every numerical features. The ACF of a numerical feature is the (linear) autocorrelation between the value of the feature at a timestamp t and its value at a later timestamp $t+l$, where l is the lag.

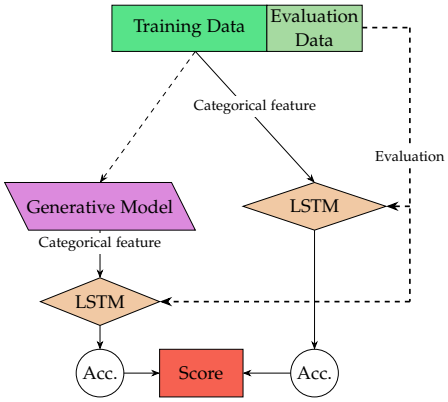


Figure 6.3: TSTR methodology: two LSTMs are trained, one on the training data, and the other on the generated data. Their accuracy are then compared on evaluation data. This is repeated for each categorical feature.

However, since not all lags exhibit strong autocorrelation, calculating the difference between ACFs across all lags may smooth out the differences for those lags that do reveal significant temporal dependencies. To address this, we discard any lags whose autocorrelation in the training data does not exceed the Bartlett confidence interval [7]. This ensures that we compute the ACF difference only for the lags that demonstrate strong temporal dependencies. This approach allows us to verify whether a temporal dependency present at a certain lag in a training feature is accurately reproduced at the same lag in the generated feature, and we apply this procedure across all numerical features of the training dataset.

CATEGORICAL FEATURES For categorical features, we decided to implement a TSTR (Train on Synthetic, Test on Real) method [208]. It is commonly used when it comes to evaluate the preservation of temporal dependencies [128, 165, 167]. This method compares the performance of a model in a machine learning task when it is trained on the training dataset and when it is trained on the generated dataset. To apply this methodology and highlight how a model preserves temporal dependencies, we compare the performance of an LSTM (a temporal deep learning model) when it is trained on training data versus when it is trained on generated data for a feature-wise autoregressive task.

For one categorical feature, we first encode its values in a one-hot encoded vector. Then, we train an LSTM to predict the next one-hot encoded value of that feature given the previous values in a context window. This first training is done on the training dataset. Afterward, we train another LSTM with the same hyperparameters (same number of hidden dimensions, same context size, etc.) on the generated dataset. We compute the accuracy of the two LSTMs on the evaluation set, and the final TSTR score is the difference between the two values. We do this process (illustrated in Figure 6.3) for every categorical feature. In practice, because we do not want our score to rely too heavily on one configuration of the LSTM, we repeat the operation multiple times while varying its hyperparameters. The score for each categorical feature will be the average of the differences in accuracy for every LSTM.

6.7 EXPERIMENTS

In this section, we would like to verify whether *FlowChronicle* is able to generate network flows with a higher quality than other model-based generation methods. The implementation of *FlowChronicle*, as well as the experimental setup, are available online as open source software[2].

6.7.1 *Competing methods*

We compare *FlowChronicle* with **CTGAN** [192], **TVAE** [192], **E-WGAN-GP** [150] and **NetShare** [199]. **NetShare** does model temporal dependencies, so we can compare our model against another method with temporal dependencies. We were also interested in adding STAN to our benchmark, but despite our best efforts, we were unable to reproduce the work of Xu et al. [195] from the author's repository. We therefore omit it from comparison. We also compare our method to the Bayesian Network proposed by Schoen et al. [157]: we call it **IndependentBN**. We also propose a variation that generates a sequence of five network flows instead of generating every network flow independently: we call it **SequenceBN**. Recently **Tranformer**-based methods have been shown to be great synthetic data generators. As a represen-

2 https://github.com/joschac/FlowChronicleCoNEXT

Feature	Description of the feature
Date first seen	Timestamp of the first packet of the flow
Proto	Transport protocol
Src IP Addr	Source IP Address (Client)
Dst IP Addr	Destination IP Address (Server)
Dst Pt	Destination Port
In Byte	Number of Bytes coming to the client
In Packet	Number of Packets coming to the client
Out Byte	Number of Bytes sending from the client
Out Packet	Number of Packets sending from the client
Flags	Type of flags contained in the flow
Duration	Duration of the flow

Table 6.1: Set of Features in our dataset

tative, we compare against the GPT2 architecture [147]. We tokenize the data as we do for our method, and use a context window of 60 tokens.

6.7.2 *Experimental protocol*

We evaluate the methods on the CIDDS-001 dataset [151]. It is a simulated dataset of 4 weeks of traffic from 30 terminals (5 servers, 3 printers, 4 Windows clients, and 15 Linux clients). Because we focus on generating benign traffic, we only kept the data recorded in the OpenStack environment.

We use *week-3* as a training set, and *week-4* as a held-out **Reference** set. The creation of *week-4* followed the same process as *week-3*, and thus, we consider this Reference set as the best possible synthetic generation. Because some parts of our evaluation methodology also require another evaluation subset (see Section 6.6.2, Categorical features), we consider the benign traffic of *week-2* as an evaluation set. We process the original unidirectional flows into bidirectional flows. We consider the 11 flow features shown in Table 6.1. We did not include the *Source Port* because it is generally randomly sampled in a particular range (cf. RFC6056 [98]). In the original dataset, the external public IP ad-

dresses were anonymized [151]. The value generated by our method will, therefore, be anonymized too.

6.7.3 *Time-independent Evaluation*

We begin by evaluating the flows independently, without considering temporal dependencies between flows. The different metrics were computed 20 times on 20 different subsamples of both the generated and training data. Each subsample includes 10000 flows. The average value of each metric as well as the ranking of all our models according to each of them are reported in Table 6.2. With this global benchmark, we can see that *FlowChronicle* is on average above the other model-based methods, with CTGAN being a close second.

	Density	**CMD**	**PCD**	**EMD**	**JSD**	**Coverage**	**DKC**	**MD**	**Rank**
	Real. ↑	*Real.* ↓	*Real.* ↓	*Real./Div.* ↓	*Real./Div.* ↓	*Div.* ↑	*Comp.* ↓	*Nov.* =	*Average Ranking*
Reference	(0.69)	(0.06)	(1.38)	(0.00)	(0.15)	(0.59)	(0.00)	(6.71)	-
IndependentBN	7 (0.24)	5 (0.22)	6 (2.74)	8 (0.11)	4 (0.27)	4 (0.38)	4 (0.05)	4 (5.47)	5.25
SequenceBN	6 (0.30)	2 (0.13)	5 (2.18)	7 (0.08)	3 (0.21)	3 (0.44)	2 (0.02)	3 (5.51)	3.875
TVAE	3 (0.49)	4 (0.18)	3 (1.84)	2 (0.01)	5 (0.30)	5 (0.33)	6 (0.07)	5 (5.17)	4.125
CTGAN	2 (0.56)	3 (0.15)	2 (1.60)	3 (0.01)	2 (0.15)	2 (0.51)	8 (0.11)	2 (5.70)	3.0
E-WGAN-GP	8 (0.02)	7 (0.34)	8 (3.63)	5 (0.02)	7 (0.38)	8 (0.02)	7 (0.07)	6 (4.66)	7.0
NetShare	5 (0.32)	6 (0.28)	**1 (1.47)**	6 (0.03)	6 (0.36)	6 (0.22)	5 (0.05)	7 (3.82)	5.25
Transformer	**1 (0.62)**	8 (0.78)	7 (3.62)	**1 (0.00)**	8 (0.55)	7 (0.03)	3 (0.05)	8 (3.75)	5.375
FlowChronicle	4 (0.41)	**1 (0.03)**	4 (2.06)	4 (0.02)	**1 (0.10)**	**1 (0.59)**	**1 (0.02)**	**1 (5.87)**	**2.125**

Table 6.2: Ranking of our different models without considering the preservation of temporal dependencies. For each metric, the average of the score is given between parentheses. Real.: Realism, Div.: Diversity, Comp.: Compliance, Nov.: Novelty, ↓: Lower is better, ↑: Higher is better. =: closer to Reference is better.

REALISM We see that Transformer and CTGAN achieve a pretty high density (0.62 and 0.56 respectively). However, Transformer seems unable to represent cross-feature correlation as illustrated by its CMD and PCD (0.78 and 3.62, respectively). Moreover, E-WGAN-GP seems unable to create realistic data (0.02 of Density, 0.34 of CMD and 3.63 of PCD). This might be because our dataset is bidirectional, whereas the encoding IP2Vec was originally intended for unidirectional datasets. *FlowChronicle* creates above-average data regarding the Realism of its synthetic network flows.

DIVERSITY While the Transformer produce a rather realistic result, it fails to produce diverse results: it has a low Coverage (0.03) and a high JSD (0.55). This is because the Transformer model fell into a well-known behavior of autoregressive generative models during training called *degeneration* [59, 203]. This phenomenon consists of the model learning to generate one specific sequence of network flow and keep repeating it during the generation process. We also see the difficulty for Bayesian Networks to work with numerical variables (EMD of 0.08 and 0.011 for SequenceBN and IndependentBNs, respectively) – a phenomenon already highlighted by Schoen et al. [157]. *FlowChronicle* and CTGAN are among the best models for covering the entire training distribution.

COMPLIANCE Apart from CTGAN (DKC of 0.11), the models are able to generate traffic that is compliant with our set of rules. *FlowChronicle* produces data with the least compliance issues.

NOVELTY With its MD of 5.87, *FlowChronicle* is closest to the reference data set (6.71), denoting its ability to generate fresh data. On the other hand, Transformer and NetShare introduce too little novelty in the synthetic data.

6.7.4 *Preservation of temporal correlation*

The previous evaluation did not consider preserving temporal dependencies between the flows during the generation. This is the goal of this subsection. Overall, *FlowChronicle* preserves temporal dependencies in both categorical and numerical features, making it closest to the reference.

NUMERICAL FEATURES: DIFFERENCE OF AUTOCORRELATION In Figure 6.4, we represent the differences of the autocorrelation function (ACF) across all numerical features between the real training dataset and the generated dataset. CTGAN and TVAE are the worst models for preserving temporal dependencies in numerical features. Both models come from the same library [137] and do not take into account temporal dependencies. Both Transformer and NetShare preserve the temporal dependencies well since these models are designed to preserve

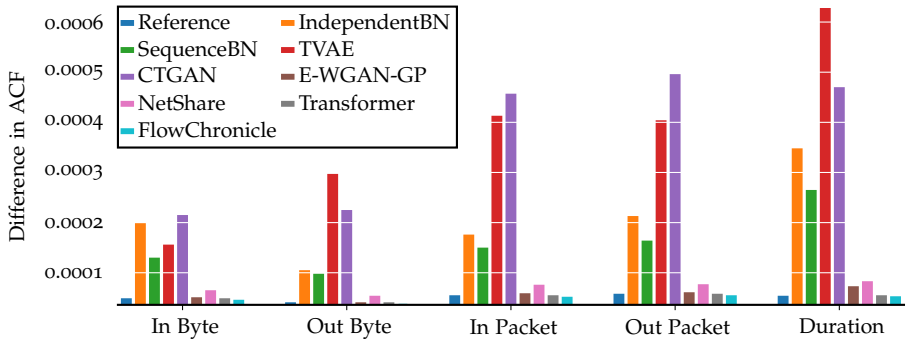


Figure 6.4: Difference of ACF between the generated data and the train data across all the numerical features for our different generative methods. Lower is better.

such dependencies. More surprisingly, E-WGAN-GP, which samples network flow independently has also a low score. *FlowChronicle* is better than those methods and reproduces well the different autocorrelation across the different numerical features. The differences in ACFs between the set generated by *FlowChronicle* and the Reference set are almost equivalent.

CATEGORICAL FEATURES: IMPACT OF GENERATED SEQUENCES ON THE ACCURACY OF AN LSTM In Figure 6.5, we see the difference in accuracy between two LSTMs trained on the training data and on synthetic data generated by every model, and this, for every categorical feature in the dataset. *FlowChronicle* is the best among the other generative models for preserving temporal dependencies across categorical features, with a score once again close to the Reference set.

6.7.5 *Computational Cost*

Comparing computing costs can be useful for choosing the right generation method. In Table 6.3, we report the time taken for training each method and generating synthetic data from it. All our experiments have been carried out on a server with 500 GB of RAM, 2 AMD EPYC 7413 CPUs, and 3 A40 Nvidia GPUs.

One drawback of *FlowChronicle* is the time required to train and generate new data. Even if *FlowChronicle* obtained on average the best per-

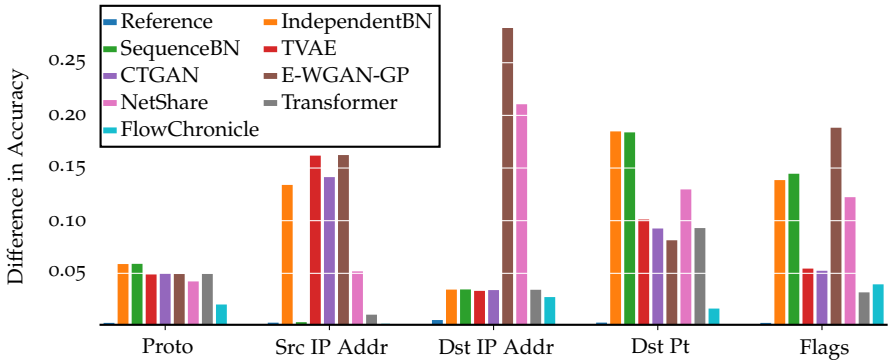


Figure 6.5: Average difference of accuracy of various LSTMs trained on the train data and our generated data from our different generative models. Each subgroup is one feature, and each bar is one generation method. Lower is better.

formances on independent and temporal metrics, it is also the longest to train and produce new data. While a long training time is a known issue of MDL-based methods, we are confident the generation time could be largely lowered due to the simplicity of the process.

6.7.6 *Explainable patterns*

Besides a good-quality generation, *FlowChronicle* has the advantage of learning explainable patterns. In this section, we present a few interesting multi-flow patterns.

The first one contains three partial flows, from the same source IP and the same destination IP. All three partial flows are HTTPS requests (TCP protocol, destination port 443). When a browser requests a webpage, there can be many resources that it needs to download from the same server (images, scripts, styles, etc.) that can be fetched with a different HTTPS connection.

A second pattern is a DNS request (UDP protocol, destination port 53) followed by an HTTP flow (TCP protocol, destination port 80). The source IP is the same. This is a classical network pattern: before a device can access a domain for an HTTP request, it must know its IP. It could be cached locally but sometimes requires a DNS request to obtain it.

	Duration (hh:mm)	
Model	*Training*	*Generating*
IndependentBN	00:12	<0:01
SequenceBN	00:31	<0:01
CTGAN*	29:12	00:02
TVAE*	02:01	00:03
E-WGAN-GP*	00:36	01:59
NetShare*	59:39	05:00
Transformer*	84:02	34:41
FlowChronicle	106:54	85:16

Table 6.3: Training and generating runtimes. Methods annotated with * rely on GPU.

A third pattern contains two partial flows, from the same source IP but to different destination IPs. The first partial flow is an HTTPS request (TCP protocol, destination port 443) and the following one is a DNS request (UDP protocol, destination port 53). It can be explained by the fact that there can be some resources on a web page that are stored on other web servers (scripts or images). In that case, the browser needs to ask for the IP address of the server that stores those resources. This pattern is probably missing some later HTTPS connections. We did not find multi-flow patterns related to non-Web protocols though. We consider that such patterns explanation strongly indicates that *FlowChronicle* is indeed capable of learning relevant patterns that can be verified and explained by experts.

6.8 CONCLUSION

In this chapter, we consider the issue of generating synthetic network traffic, with a focus on preserving the temporal dependencies within the generated data. To achieve this goal, we introduced an innovative data-generating approach, dubbed *FlowChronicle*, which is based on pattern set mining. Initially, *FlowChronicle* learns a set of patterns that effectively encapsulate the data distribution and describe the data within the model. We formalize the problem with the Minimum De-

scription Length (MDL) principle, by which our method is naturally robust against overfitting.

During the evaluation phase, we observed that *FlowChronicle* not only upholds the diversity and realism of the data but also maintains the temporal dependencies among flows. Even without taking into account any temporal dependencies, the generation through pattern mining allows us to reproduce network flows that are really close to the training data. In the non-temporal evaluation, the second-best method was CTGAN, but it struggled to capture temporal dependencies. Conversely, the Transformer model preserved temporal dependencies well but failed to generate high-quality individual flows. *FlowChronicle*, however, consistently ranked highest in both evaluations, excelling in both flow quality and temporal dependency preservation.

Finally, contrary to other methods, *FlowChronicle* outputs patterns that can be manually analyzed. This way, the generation method can be audited, and possibly manually verified and corrected. It is also easy to manually include new pattern to modify the generation without a relearning procedure.

Looking ahead, we see potential for improvement in a more powerful pattern language. Although our language is already robust, there are certain concepts, such as repeating flows as observed in video streaming, that we cannot represent effectively yet. A more expressive pattern language will come with additional challenges, such as the increased search space.

7

CAUSAL DISCOVERY FROM EVENT SEQUENCES BY LOCAL CAUSE-EFFECT ATTRIBUTION

In the previous chapters we have summarized event sequences using different pattern languages, that is we discovered correlations between events in the event sequences. In this chapter we study causality between events in event sequences.

The gold standard for inferring causal relationships is the randomized controlled trial, commonly a population sample is randomly divided into two groups: one group receives an intervention on the suspected causal variable, while the other serves as a control. The impact of the intervention is then measured and compared across groups. However, in many real-world scenarios, conducting such controlled experiments is either prohibitively expensive or practically infeasible. In this chapter, we study the problem of discovering the underlying causal structure from observed event sequences. To this end, we introduce a new causal model, where individual events of the *cause* trigger events of the *effect* with dynamic delays.

7.1 INTRODUCTION

Suppose we are considering a multivariate event sequence. What caused a specific event to happen? Which variables are causes of each other? Data-driven methods can infer causal relationships from observed data. Existing methods for discovering causal networks from event sequence data [19, 84, 191] are based on *Granger causality* [69]. This purely predictive notion defines a variable X to be a cause of another variable Y if the past of X helps to predict the future Y. It is a relatively weak notion of causality that excludes instantaneous effects and is often un-

This chapter is based on [36]: Joscha Cüppers, Sascha Xu, Musa Ahmed, and Jilles Vreeken. "Causal Discovery from Event Sequences by Local Cause-Effect Attribution." In: *Advances in Neural Information Processing Systems*. 2024, pp. 24216–24241.

able to discover true causal dependencies; in Granger causality, baking a cake is causal to a birthday.

In this chapter, we instead build upon Pearl's model of causality, which assumes the existence of an an underlying causal structure in the form of a directed acyclic graph (DAG) [138]. In our context, such a graph describes the causal relationships between types of events, such as alarms in a network. We propose a new causal model for event sequences based on a one-to-one *matching* of individual events, where we model the process of one individual event of a certain type possibly causing an individual event of another type. In our model, we take into account the uncertainty of whether an event is actually caused or independently generated, the uncertainty of an event actually causing an effect or failing to do so, and the uncertainty of the delay between cause and effect. As we will show, our model has several advantages, such as a clear notion of what event caused another and the identifiability for both instant and non-instant effects.

We base our theory on the Algorithmic Markov Condition (AMC) [85], which postulates that the true causal model achieves the lowest Kolmogorov complexity. As Kolmogorov complexity is not computable, we instantiate it via the Minimum Description Length (MDL) principle [71]. We show that our score is consistent, identifies the true causal direction for both instantaneous and delayed effects, and formally connect it to Hawkes processes. To discover causal networks in practice, we introduce the CASCADE algorithm, which adds edges in topological order. Through extensive empirical evaluation, we show that CASCADE performs well in practice and outperforms the state of the art by a wide margin. On synthetic data, CASCADE recovers the ground truth without reporting spurious edges, and on real-world data, it returns graphs that correspond to existing knowledge.

7.2 PRELIMINARIES

We write $i \to j$ when S_i is a cause of S_j and $pa(j)$ for the set of parents of node j. We assume faithfulness, sufficiency, and the causal Markov condition [96].

INFORMATION-THEORETIC CAUSAL DISCOVERY The Algorithmic Markov Condition (AMC) postulates that the factorization of the joint

distribution according to the true causal network achieves the lowest Kolmogorov complexity [85]. The Kolmogorov complexity $K(x)$ of a binary string x is the length of the shortest program p for a universal Turing machine $\mathcal{U}$ that computes x and halts [103]. For a distribution P, it is the length of the shortest program that uniformly approximates P arbitrarily well,

$$K(P) = \min_{p \in 0,1^*} \{|p| : \forall_y |\mathcal{U}(p, y, q) - P(y)| \leqslant \frac{1}{q}\} .$$

The AMC states that the Kolmogorov complexity of the joint distribution $P(X)$ is the sum of the complexities of the conditional distributions $P(X_i|pa(i))$ of the true DAG G^*, i.e.

$$K(P(X)) = \sum_{i=1}^{p} K\left(P(X_i|pa(i))\right) , \tag{7.1}$$

up to a constant independent of the input. Due to, among others, the halting problem, Kolmogorov complexity is not computable, but we can approximate it from above. A statistically well-founded way to do so is by Minimum Description Length (MDL) [71, 115]. For a fixed class of models $\mathcal{H}$, MDL identifies a description length L of encoding data X together with its optimal model,

$$L(X \mid \mathcal{H}) = \min_{h \in \mathcal{H}} \left(L(X \mid h) + L(h)\right) .$$

Next, we introduce the assumed data generating process, its corresponding model class $\mathcal{H}$ and encoding length function L, and show under which conditions it can be identified.

7.3 THEORY

To be able to infer causal relationships from observational data, we need to make assumptions about the underlying data-generating process [138]. The key assumption we make here is that an individual event of type i at time t with probability $\alpha_{i,j}$ causes an individual event of type j at time $t' \geqslant t$. To illustrate, we give a toy example in Fig. 7.1 in which event sequence S_i causes event sequence S_j. The individual events in S_i occur uniformly at random. The first and third events in S_i cause events in S_j, resp. with a delay of 0.2 and 0.3. The other two

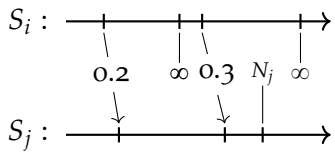


Figure 7.1: Cause-effect matching, where S_i causes S_j.

events in S_i do not cause events in S_j, denoted by a delay of ∞. The final event in S_j is due to noise, marked by N_j.

Next, we formally describe the causal mechanism. We differentiate between source and effect nodes.

Source nodes are nodes i in G^* with an empty parent set $pa(i)$. For source nodes i we assume that the events in S_i occur uniformly at random with a rate of λ_i events per time unit. This mechanism, commonly known as a homogeneous *Poisson process*, is used, for example, as a model for accident rates requiring hospital admission [185]. In this work, we focus on the delay times between individual events, denoted as d_k for the delay $t_k - t_{k-1}$ and as $\Delta_{i\to i} = \{d_k\}_{k=1}^{n_i}$ for the sequence. For a Poisson process, the delay times are independently and exponentially distributed. Thus, we model a source event sequence S_i as

$$S_i = \{t_k\}_{k=1}^{n_i}, \text{ where } t_k = \sum_{l=1}^{k} d_l, \quad \Delta_{i\to i} = \{d_k \sim \text{Exp}(\lambda_i)\ iid\}_{k=1}^{n_i}. \tag{7.2}$$

Effect nodes are nodes j in G^* with at least one parent $pa(j)$. For each effect node j, the individual events in S_j are either caused by an individual event in an S_i with $i \in pa(j)$ or due to noise. That is, reasoning from the causing node i, every event $t_k \in S_i$ may trigger an event of the effect S_j with a probability of $\alpha_{i,j}$. If triggered, an individual event in S_j will occur after a random delay d_k, drawn from a cause-effect specific delay distribution $\Phi_{i,j}$ parameterized by $\theta_{i,j}$, e.g. the rate λ of an exponential distribution, and $\alpha_{i,j}$. If no event is triggered, then we model the delay as infinite, i.e. $d_k = \infty$. The sequence of delays $\Delta_{i\to j}$ from $S_i \to S_j$ is modeled as

$$\begin{aligned} \Delta_{i\to j} = \{d_k\}_{k=1}^{n_i}, \qquad & d_k \sim \Phi_{i,j}(\alpha_{i,j}, \theta_{i,j})\ iid, \\ \phi_{i,j}(d) = & \begin{cases} 1 - \alpha_{i,j} & \text{if } d = \infty \\ p(d; \theta_{i,j}) \cdot \alpha_{i,j} & \text{else} \end{cases} \end{aligned} \tag{7.3}$$

where $\phi_{i,j}(d)$ denotes the density of the delay distribution. Thus, given the event sequence S_i of the cause and delays $\Delta_{i\to j}$, the individual events in S_j caused by S_i are obtained by adding the delays d_k to the time stamps t_k of the individual cause events, with the reconstruction function f as

$$f_{i,j}(S_i, \Delta_{i\to j}) = \{t_k + d_k \mid d_k \neq \infty\}, \text{ for } k = 1, \ldots, n_i \,. \tag{7.4}$$

In addition, individual events in S_j can also be due to noise. Like for source nodes, we assume these a Poisson process as per Eq. (7.2) with rate λ_j, i.e. $N_j \sim \text{Poisson}(\lambda_j)$. Putting this together, given causal structure G^*, an effect event sequence S_j is generated by taking the union of the individual delays from the causal parents $pa(j)$ and the time stamps due to noise N_j,

$$S_j = \left(\bigcup_{i \in pa(j)} f_{i,j}\left(S_i, \Delta_{i\to j}\right) \right) \cup N_j \,. \tag{7.5}$$

Next, we instantiate an MDL score for this causal model and consider its identifiability.

7.3.1 *Minimum Description Length Instantiation*

We now develop a score for our causal model using MDL [71]. It consists of the cost of the data given the model, $L(S \mid \Theta)$, i.e. the negative log-likelihood of the data, and the cost of the model, i.e. that of the parameters, $L(\Theta)$, and that of the graph, $L(G)$, all measured in bits.

DATA COST The cost of data in bits directly corresponds to its negative log-likelihood, i.e. the likelihood of each delay as per Eq. (7.3) over all the event sequences corresponding to the parents of node j and that of the noise events. Formally, we have

$$L\left(S_j \mid S_{pa(j)}, \Theta\right) = \sum_{i \in pa(j)} \sum_{d_k \in \Delta_{i\to j}} -\log(\phi_{i,j}(d_k)) + \sum_{d_l \in \Delta_{j\to j}} -\log\left(\phi_{j,j}(d_l)\right) \,. \tag{7.6}$$

The first term encodes those events that were caused by the parent S_i through the delays $\Delta_{i\to j}$. Here, we use a Shannon-optimal coding that

requires $-\log(\phi_{i,j}(d_k))$ bits per sample [71]. In the second term, we encode all remaining events as noise using the delay distribution of a Poisson process. For source events, i.e. variables without any parents, only the noise term is present. The cost of all sequences is then simply

$$L(S \mid G, \Theta) = \sum_{j \in [p]} L(S_j \mid S_{pa(j)}, \Theta) \quad .$$

PARAMETER COST Next, we define the costs of the DAG, $L(G)$, and that of the parameters, $L(\Theta)$. We encode the DAG in topological order. Per node we encode its number of parents $|pa(i)|$ and identify which those are, i.e. $L(G) = \sum_{k=0}^{d-1} \left(\log(k) + \log \binom{k}{|pa(i)|} \right)$. Depending on their type, we encode the parameters $\theta \in \Theta$. For parameters $\theta \in \mathbb{N}$ we use $L_{\mathbb{N}}$, the MDL-optimal encoding for integers [153]. For parameters $\theta \in \mathbb{R}$ we use $L_{\mathbb{R}}(\theta) = L_{\mathbb{N}}(d) + L_{\mathbb{N}}(\lceil \theta \cdot 10^d \rceil) + 1$ as the number of bits needed to encode a real number up to a user-specified precision [114]. For an edge $i \to j$, the parameters are the trigger probability $\alpha_{i,j}$ and those of the delay distribution ϕ. For the cost of an edge we hence have $L(i \to j) = L_{\mathbb{R}}(\alpha_{i,j}) + \sum_{\theta \in \phi_{i,j}} L(\theta)$. For Θ as a whole, we have $L(\Theta) = \sum_{i \to j \in G} L(i \to j)$. The overall MDL score is then

$$L(S \mid G, \Theta) + L(G) + L(\Theta) \, . \tag{7.7}$$

7.3.2 *Identifiability*

We now study the identifiability guarantees of our model and score, i.e. under what conditions we can identify from a given pair which is the cause and which the effect. Consider a pair of event sequences S_i and S_j, where $S_i \to S_j$ and the cause S_i is a source event while S_j is an effect event.

INSTANT EFFECTS We begin with the case of instant effects only. Instant effects are observed when the sampling frequency of the data, e.g. a daily time scale, is insufficient to pick up a difference in time, such as a financial crash that can spread across the globe within hours. It is well-known that Granger causality cannot identify the causal direction for instant effects [142]. In Pearl's causal framework, on the other hand, the causal direction between two binary variables is identifiable [16, 95, 143]. We can build upon these results and show that our

causal model and MDL-based score can identify the causal direction for non-deterministic instant effects.

Theorem 1. *Let S_i be an event sequence generated by a Poisson process as per Eq. (7.2) and S_j be an effect of S_i as per Eq. (7.5), with, low noise $\lambda_j < (1 - \alpha_{i,j})\lambda_i$, and a trigger probability $\alpha_{i,j} < 1$.*

In the case of exclusively instant effects, i.e. $\phi_{i,j}(d) = \delta(d)$, where $\delta(d)$ is the Dirac delta function, the MDL score in the true causal direction is lower than in the anti-causal direction, i.e.

$$\lim_{n_i \to \infty} L(S_j \mid S_i, \Theta_1) + L(S_i \mid \Theta_1) < L(S_i \mid S_j, \Theta_2) + L(S_j \mid \Theta_2) \ .$$

We provide the full proof in the Appendix. e.1.1, the general idea is under a non-deterministic trigger mechanism, i.e. $\alpha_{i,j} < 1$. Then, in the causal direction, we can fully explain S_j with S_i, but not vice-versa, as the cause is generated by a Poisson process. If $\alpha_{i,j} = 1$, i.e. the process is deterministic, we always observe cause and effect together, making them indistinguishable.

DELAYED EFFECTS Next, we consider the case of exclusively delayed effects. Here, there is an inherent asymmetry in the benefit of knowing the cause versus the effect. As shown by Didelez [45] for marked point processes, and later used by Eichler, Dahlhaus, and Dueck [49] and Xu, Farajtabar, and Zha [191] for Granger causality in Hawkes processes, the intensity of observing the cause after an event of the effect is unchanged. That is, the future of the cause is independent of the past of the effect, while if a cause triggers an effect, the intensity of the effect is increased by the cause. We have the following identifiability guarantee.

Theorem 2. *Let S_i be an event sequence generated by a Poisson process as per Eq. (7.2) and S_j be an effect of S_i as per Eq. (7.5), such that $H\left(\phi_{j,j}\right) > H(p(;\theta_{i,j})) + \alpha_{i,j}^{-1} H\left(\mathcal{B}(\alpha_{i,j})\right) + \alpha_{j,j}^{-1} H\left(\mathcal{B}(\alpha_{j,j})\right)$, where H denotes the entropy and $\mathcal{B}$ the Bernoulli distribution.*

Then the matching in the anti-causal direction $\Delta_{j \to i}$ of the effect S_j to the cause S_i has a worse MDL score than the true matching $\Delta_{i \to j}$, i.e.

$$L(S_j \mid S_i, \Theta_{i \to j}) + L(S_i \mid \Theta_i) < L(S_i \mid S_j, \Theta_{j \to i}) + L(S_j \mid \Theta_j) \ .$$

We provide the full proof in the Appendix. e.1.2. The idea being that in the anti-causal direction $S_j \to S_i$, the delay times follow the same

exponential distribution of $\mathrm{Exp}(\lambda_i)$, leading to no gain in score compared to the self-delay encoding. On the other hand, in the true causal direction, knowing the times of the cause leads to a better knowledge of the delay and hence a lower cost, so long as the delay distribution $\phi_{i,j}$ provides a better description than treating it as noise. This requirement is closely related to the Algorithmic Markov Condition, which postulates that the shortest description of a variable is given through its parents.

7.3.3 *Connection to Hawkes Processes*

Hawkes processes [78] are analytically convenient and well-suited for modeling real-world processes where events trigger further events, e.g. earthquakes triggering aftershocks. Consequently, the majority of methods focusing on Granger causality are based on Hawkes processes [19, 84, 191]. The Hawkes process extends the Poisson process by incorporating the influence of past events on the intensity, i.e. the rate of occurrence of future events. This is done by means of excitation functions $v_{i,j}(t - t_k)$, which increase/inhibit the intensity of future events based on past events. The intensity function of a Hawkes process under a DAG structure is given by

$$\lambda_j(t) = u_j + \sum_{i \in pa(j)} \sum_{t_k < t, t_k \in S_i} v_{i,j}\left(t - t_k\right) \ . \tag{7.8}$$

Each event $t_k \in S_i$ increases the intensity of seeing an effect by $v_{i,j}(t - t_k)$. The main difference between our model and a Hawkes process is our direct trigger model from cause to effect. In a Hawkes process, an event of type i increases the intensity and, therewith, the probability of effect events occurring. That is, contrary to our framework, in a Hawkes process there is no explicit one-to-one relationship between causing and effect events, i.e. no one event can be attributed solely to causing another. Nonetheless, in Appendix e.1.5 we show how to identify S_i as a parent of S_j by constructing a sequence of delays $\Delta_{i \to j}$ with the most-influential past event and therewith $\phi_{i,j}$. If $\phi_{i,j}$ fulfills Theorem 2, we can identify S_i as a parent of S_j. Hence, should the data be generated by a Hawkes process, our method can still pick up the causal relationship between the two event classes, so long as there are sufficiently many events where S_i is the primary cause.

7.4 ALGORITHM

With our model in place, we now turn to the problem of discovering the underlying causal structure from an observed sequence of events. In recent years, several methods that find and proceed on a topological ordering of the true graph have been introduced [18, 26, 154], which outperform other score-based frameworks such as GES [25] in terms of accuracy. We here propose the CASCADE algorithm that instantiates this idea for information-theoretic scores. We prove that in the limit, it recovers not only the correct topological ordering but also the correct parent set of each node. CASCADE derives its guarantees from the *gain* in bits of adding an edge $i \to j$ to the model, i.e.

$$g(i \to j \mid \Theta) = L(S_j \mid S_{pa(j)}, \Theta) - L(S_j \mid S_{pa(j)\cup i}, \Theta \cup \theta_{i,j}) + L(i \to j) \,. \tag{7.9}$$

The edge cost $L(i \to j)$ is constant and independent of the number of samples n_i. In the limit $n_i \to \infty$, the gain inherits the identifiability guarantees from Sec. 7.3.2, such that $g(i \to j \mid \Theta) > g(j \to i \mid \Theta)$ if S_i is a true ancestor of S_j. In other words, the gain of an edge is greater in the causal than in the anti-causal direction.

7.4.1 *High Level Overview*

CASCADE initializes the model with an empty graph G. During the search, we maintain a set of nodes $C = [p]$, from which we remove nodes in a topological order of G^*. We iterate over the following four steps until C is empty.

1. **Source Node Selection**: Select that node $i \in C$ with minimal gain for any edge $j \to i$, $j \in C$, i.e.

$$\arg\min_{i \in C} \max_{j \in C} g(j \to i \mid \Theta) - g(i \to j \mid \Theta) \,. \tag{7.10}$$

2. **Edge Adding**: Add all *outgoing* edges from $i \to j$, $j \in C$, to G that *improve* our score.

3. **Edge Pruning**: Remove all *incoming* edges $j \to i$ from G that harm our score.

4. **Node Set Update** Remove i from C.

Each iteration, CASCADE selects that node i, which has the minimal achievable gain when adding any edge $j \to i$ to the current graph G, expressed in Eq. (7.10); below, we will show that under our causal model this node is guaranteed to be a true source of the graph G^*. We then add all edges from i to nodes $j \in C$ that improve our score; provided that all true causal edges $i \to j$ were added, there is now at least one node $j \in C$ whose parents are all accounted for, that in the next iteration can be identified as a source. We prune edges $j \to i$ from G to remove shortcuts. By repeating this process, CASCADE proceeds in a topological order of the true graph G^*. In total, CASCADE requires p iterations, leading to an overall cubic complexity $O(p^3)$.

SOURCE NODE SELECTION. To identify a source node in the graph, we can use the identifiability guarantees from Sec. 7.3.2. They show that the gain $g(i \to j \mid \Theta)$ correctly orients the edge $i \to j$ in the unconfounded bi-variate case. We additionally require that the edge gain is *pathwise oracle*, i.e. it can identify the direction of the path from i to k.

Theorem 3. *Given an event sequence S generated by a causal structure G^*, let S_i be a source node of G^* and S_v be a descendant of S_i, where there exists a path $i \to j \to \cdots \to v$ in G^*.*

Then, the gain in the causal direction of the path $g(i \to v \mid \Theta) - g(v \to i \mid \Theta)$ is greater.

We provide the proof in Appendix. e.1.3. We can now show that the criterion in Eq. (7.10) selects nodes in a topological ordering of G^*. Initially, CASCADE has to identify a true source of G^*, i.e. a node i without parents. For that node i, all other nodes j are either ancestors or independent of i. If i is an ancestor of j, then $g(j \to i \mid \Theta) - g(i \to j \mid \Theta) < 0$, i.e. the gain in the anti-causal direction is lower. If i is independent of j, then $g(j \to i \mid \Theta) = 0$ and $g(i \to j \mid \Theta) = 0$. Hence, the maximum achievable gain for a node without parents is zero.

Now consider a node v which does have a parent. For this node, there exists an ancestor u which is a true source. Hence, for that pair $g(u \to v \mid \Theta) - g(v \to u \mid \Theta) > 0$. Consequently, the maximum achievable gain is positive, whilst for a source node, we can maximally achieve zero, allowing us to identify true sources with Eq. (7.10).

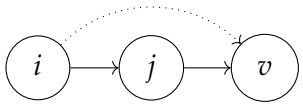


Figure 7.2: Causal chain

In the next step, we add all outgoing edges from the source i to G that improve the score. As G^* is a DAG, we are now guaranteed to have another node j, whose incoming edges are all accounted for in G. Then, as per the causal model from Eq. (7.5), the only events that remain are those of the noise N_j. Hence, j is now a source node for which the guarantees from above apply. By repeating this process, CASCADE thus follows a topological order of G^*.

EDGE ADDITION Given a source i, CASCADE adds all outgoing edges $i \to j$ that improve the score. We restrict the set to nodes $j \in C$ from the candidate set only, i.e. to nodes further down the topological order. By the Algorithm Markov Condition, the description length of the true set of parents of a node j is smaller than the description length of any other set of parents, and hence the gain of the true edge is positive in the limit of $n_i \to \infty$.

When adding an edge $i \to j$, where there is already an edge $v \to j$, we use an Expectation Maximization approach to attribute all events to their respective cause. That is, we first find the bi-variate alignment $\Delta_{i \to j}$ using all events in S_j. Now, it is very likely that there are conflicts between $\Delta_{i \to j}$ and $\Delta_{v \to j}$, as the same event can be attributed to both i and v. In those cases, we choose that event where the density ($\phi_{i,j}$ or $\phi_{v,j}$) is higher and set the delay to infinity in the other matching. After re-assigning all events, we refit the delay distribution function $\phi_{i,j}$ using the new matching.

EDGE PRUNING Lastly, we deal with removing any shortcuts that have been added in the previous iteration. With the previous two steps, we are guaranteed to have a superset of all true causal edges incoming to i. Fortunately, we can prune such edges directly with MDL by removing any incoming edge $i \to j$ that does not improve the MDL score. In the chain graph $i \to j \to v$, shown in Figure 7.2, we would remove shortcut $i \to v$ as the edge $j \to v$ is sufficient to explain the

data. In practice, given the current set of parents of i in G, we search for the true set of parents by starting with the empty set and greedily adding only those edges that improve the score. As we show in Appendix. e.1.3, a shortcut always has a lower gain than the true edge and hence will not be re-added. In this manner, we are asymptotically left with only the true causal parents. We can now finally show the consistency of CASCADE.

Theorem 4. *Given an event sequence S, where each individual subsequence S_i was generated as per Eq.* (7.5) *by an underlying causal graph G^*. Assuming all $\Delta_{i\to j}$ are the true causal matchings. Under the Algorithmic Markov Condition,* CASCADE *recovers the true graph G^* for $n \to \infty$.*

We postpone the proof to Appendix. e.1.7. In the experiment section, we show that CASCADE recovers the true DAG even in challenging settings and works well on real-world data.

7.5 RELATED WORK

Causal discovery on observational data is an active research topic. Two main research directions exist: constraint-based [138] and score-based [25, 149] methods. Our approach belongs to the latter and is based on the Algorithmic Markov Condition [85]. While Kolmogorov complexity is uncomputable, Marx and Vreeken [115] formally showed that if we instantiate the AMC with two-part MDL [71], we, in expectation, achieve the same results. MDL has been successfully used for bivariate causal inference [16, 114, 194], causal discovery [120], identifying hidden confounding [90], identifying mechanisms shifts [110], and identifying selection bias [89].

In this chapter, we consider point processes. Particularly close to our method are Hawkes processes [78], as a way to model the influence of past events onto future events. As such, our work is also related to the concept of transfer entropy [159], which measures the influence in terms of Shannon entropy. Budhathoki and Vreeken [17] proposed an MDL-based method for bivariate causal inference on event sequences, which is unsuitable for learning a global causal structure.

Existing methods for discovering causal graphs from event sequence data focus on different instantiations of Granger causality and can mostly be categorized by different intensity functions. Most common

are parametric approaches with different regularizing [19, 191, 207]. ADM4 [207] uses the nuclear matrix norm in combination with lasso, THP [19] uses BIC for regularization. The method MDLH by Jalaldoust, Hlaváčková-Schindler, and Plant [84] is most closely related, as they also use MDL for regularization. NPHC [2] takes a non-parametric approach by using a moment matching method to fit second and third-order integrated cumulants. A recent development is neural point processes. Mei and Eisner [118] propose a deep neural network that learns the dependencies [118], which Xiao et al. [190] extended to include attention mechanisms. Zhang et al. [202] first learn a neural point process and then use a feature importance attribution method to obtain a weight matrix of pairwise variable influence.

7.6 EXPERIMENTS

We evaluate CASCADE on both synthetic and real-world data. CASCADE is implemented in Python. We provide the source code, along with the synthetic data generator and the used real-world datasets online.[1] We compare our method to four of state of the art methods: THP [19] as representative for the regularized parametric approaches, CAUSE [202] as representative for the neural point processes and NPHC [2] as a representative non-parametric approach, and MDLH [84] who also rely on MDL, as our most closely related competitor. CAUSE and NPHC do not return a graph but rather a weight matrix where the weight indicates the strength of the causal relation. On synthetic data, we can obtain a graph by thresholding such that we optimize the $F1$ score.

7.6.1 *Evaluation*

We evaluate the estimated graphs in terms of structural similarity by the Structural Hamming Distance (SHD) [87], in terms of causal similarity by the Structural Intervention Distance (SID) [141], and predictive performance by F1 score. To compare graphs of different sizes, we report the scores normalized by the maximally achievable SHD/SID and show the unnormalized scores in Appendix e.2.

1 https://eda.rg.cispa.io/prj/cascade/

NPHC, CAUSE, and MDLH can and often do return cyclic graphs. As SID is strictly only defined for acyclic graphs, we omit these methods from the SID evaluation.

7.6.2 *Synthetic Data*

We begin by comparing all methods on data with known ground truth. To this end, we generate synthetic data. We generate both data within and outside our causal model and vary aspects such as noise intensity, number of event types and the number of parents of a variable. We describe the full data-generating process in Appendix e.2.

SANITY CHECK We start with a sanity check on data without any structure over 20 variables, CASCADE correctly does not report any causal edge. THP reports in 45% of the cases at least one spurious edge. We omit the results of CAUSE and NPHC as it is unclear how to choose a meaningful, non-trivial threshold, in this setting. MDLH did not terminate within 96 hours.

SCALABILITY We evaluate how well each method scales under an increasing number of variables. We vary the number of nodes, which correspond to the number of unique event types, from 5 to 50 and report the results in Fig. 7.3a. As MDLH did not terminate within 96 hours for 15 variables, we omit it from here on out. For a lower number of nodes, both CASCADE and THP obtain far better results than NPHC and CAUSE. With increasing event types, all methods SID and F1 scores worsen. Amongst all methods, CASCADE scales best with an increasing number of nodes, whereas Granger causality based methods such as THP and NPHC find many spurious edges of connected but not causal variables. On the other hand, CASCADE is the most accurate method for a higher number of nodes, showing the efficacy of its causal model and MDL-based approach.

NOISE Next, we assess the impact of noise, which are events that is not caused by any parent. To this end, we vary the 'cause' probability and the fraction of events due to additive noise. We do so by varying a noise parameter a, adding an additional $n_i \cdot a$ events to S_i (additive noise), and by setting the 'cause' probability $\alpha = 1 - a$, i.e. we decrease

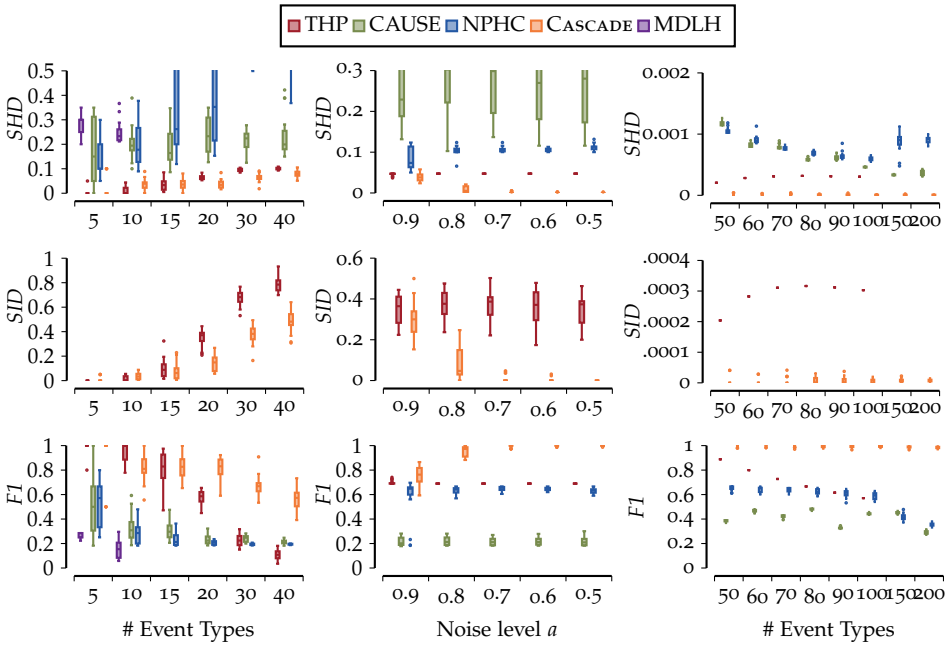


Figure 7.3: DAG recovery in different settings. We show normalized SHD, normalized SID, and $F1$ score, the Y-axis are truncated for better visualization. In (a) we vary the number of event types, on the SID score we observe that the graph reported by Cascade is casually, the most similar to the true DAG. In (b) we decrease the noise, Cascade does recover a close causal graph, even under high noise. Finally, in (c) we increase the number of parents of a collider, we observe that a high number of parents does not pose a problem for Cascade.

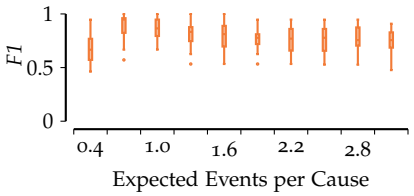


Figure 7.4: DAG recovery on data generated by a Hawkes process.

additive noise and increase trigger probability. We show the results in Fig. 7.3b. We observe that CASCADE does quite well for high noise and that for noise levels of $a = 0.7$ and lower, it (mostly) recovers the true DAG. All other methods perform considerably worse.

COLLIDERS Matching an effect event to the correct parent, resp. modeling the correct excitation, becomes increasingly challenging for a larger number of parents. We test this through a setting where half ($\lceil \frac{p-1}{2} \rceil$) the variables converge into a collider, and the other half ($\lfloor \frac{p-1}{2} \rfloor$) are independent. We vary the total number of variables, p and we show the results in Fig. 7.3c. We observe that CASCADE achieves almost perfect results. THP is robust, but with an increasing number of nodes, it starts to miss edges. Beyond 100 variables, it does not terminate within 24 hours. To validate that our method can recover structures with multiple colliders, we repeat the same experiment where 10% of nodes are colliders. That is, for 50 event types, 5 are colliders and 23 direct causes of all 5 colliders. The remaining 22 are independent. Resulting in an $F1$ score of 0.97 for 50 event types, slightly decreasing to 0.82 for 200 event types; as such CASCADE can deal well with multiple colliders.

INSTANTANEOUS EFFECTS Next, we evaluate performance under instantaneous effects. First, we consider data with exclusively instant effects. CASCADE achieves an average unnormalized SHD of 32.8. The second best-performing method, NPHC, achieves 46.85. Next, we generate a setting where 90% of the effects are instantaneous and the others occur with a small delay. CASCADE improves to an SHD of 19.45, while NPHC achieves the second lowest average with 47.5. We provide all results in the Appendix e.2.

HAWKES PROCESSES Finally, we evaluate how effectively CASCADE recovers the true DAG on data generated by a Hawkes process. We vary the intensity of the excitation function, i.e., the expected number of events generated per cause. We show the results in Figure 7.4. We observe that CASCADE performs best when generation is close to our assumptions, i.e. when there is, on expectation, one effect per cause or fewer, but still demonstrates strong performance across all settings.

7.6.3 *Real-World Data*

We evaluate CASCADE on three distinct datasets of real-world event sequences. We begin by evaluating CASCADE on a dataset of network alarms, where the causal structure is known.

NETWORK ALARMS This data was provided by Huawei for the NeurIPS 2023 CSL-competition[2] and consists of data from a simulated network of devices in which alarms can cause other alarms. We run all methods and get an (unnormalized) SHD score of 42 for CASCADE, 127 for THP, 214 for NPHC, and 1564 for CAUSE. As the network connectivity structure is known, we can take it into account during the search. THP supports this natively, CASCADE can be trivially constrained to only consider the given edges. CASCADE correctly identifies 142 out of 147 causal edges, THP 20. Neither method reports spurious edges. We show the full recovered graph in Appendix e.2.5.

GLOBAL BANKS Second, we run CASCADE on a daily return volatility dataset [43], we follow the preprocessing of Jalaldoust, Hlaváčková-Schindler, and Plant [84], specifically we turn the time series into an event sequence by rolling a one year window over the data and register an event if the last value is among the top 10%. The dataset includes the 96 world's largest publicly traded banks. We show the largest discovered subgraph in Fig. 7.5. In addition, three unconnected subgraphs are discovered, one covering banks in Australia and two others connecting banks in Japan, which we provide in Appendix e.2.5.

2 https://github.com/huawei-noah/trustworthyAI/tree/master/competition/NeurIPS2023/sample

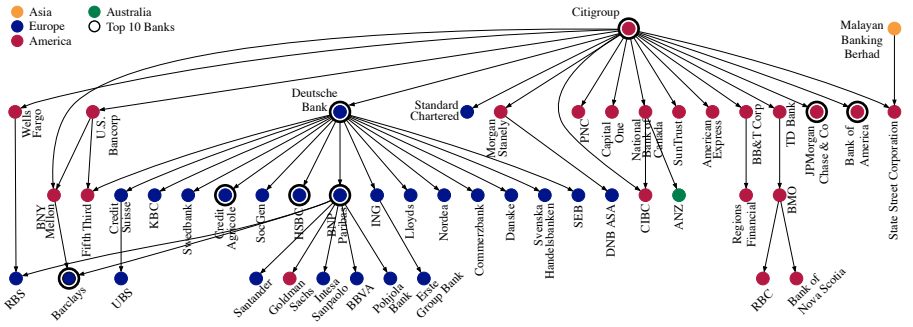


Figure 7.5: Result of CASCADE on the *Global Banks* dataset, we show the largest subgraph, we highlight the 10 largest, by assets, banks. We clearly see CASCADE recovers locality and that larger banks have a strong influence on the market, both information not provided in the input.

DAILY ACTIVITIES We run CASCADE on a dataset of recorded daily activities [133]. Our method reports plausible causal connections such as *Sleeping End* → *Showering Start* → *Showering End* → *Breakfast Start* → *Breakfast End*, etc. We show the complete graph in the Appendix e.2.5. This result reinforces the suitability of our causal model and CASCADE for real-world data, and illustrates the potential of our method to discover causal structures in a wide range of applications.

7.7 CONCLUSION

We studied the problem of causal discovery from event sequences, we proposed a cause-effect matching approach to learn a fully directed acyclic graph (DAG). To this end, we introduced a new causal model and an MDL based score. We proposed the CASCADE algorithm to discover causal graphs through a topological search from observational data. Finally, we evaluated CASCADE on synthetic and realistic data. On synthetic data, we find that CASCADE is either the best or close to the best-performing method across all settings, both within and outside our causal model. In particular, whenever conditions get challenging, e.g. due to noise or with multiple colliders, CASCADE outperforms all other methods by a significant margin. We examined how CASCADE performs on real world event sequences, where the true data-generating process may lie outside our causal model. We found that

CASCADE recovers meaningful graphs that match with a common understanding of the world.

Limitations As is necessary, we have to make causal assumptions. The most prominent in our work is the direct matching between a cause event and an effect event – which precludes modeling of a single event causing multiple other events, as well as multiple events jointly causing a single effect event – and that we only consider excitatory effects – which precludes modeling the absence of events due to a cause. Our proof of identifiability for instantaneous effects depends on the strengths of the trigger resp. noise probabilities.

Future Work Currently, our structural equations are 'or' relations over the parent's variables. An interesting future direction would be to explore 'and' relations, e.g., A and B together cause C. This raises several questions, like how close to each other A and B have to occur or if the order matters. Another interesting future direction is to allow matching of multiple causing events to one event, where each parent *could* have caused the event. This would allow us to answer counterfactual questions, such as if a causing event had not occurred, would we nevertheless observe its effect? This strongly relates to the firing squad example by Pearl [138], where multiple guards shoot a prisoner at the same time; if one guard did not shoot, the prisoner would still have died.

8

CONCLUSION

In this thesis, we explored various methods for the exploratory analysis of event sequences. In this chapter, we summarize our contributions, discuss the challenges of evaluating pattern set miners, reflect on the limitations of our work, and conclude with an outlook on future directions.

8.1 SUMMARY OF CONTRIBUTIONS

We studied summarization of event sequences by identifying meaningful patterns and predictive models. Our proposed methods use the Minimum Description Length (MDL) principle as a model selection criteria, ensuring that discovered models are both informative and compact. We made progress towards two main goals in this thesis, summarization of event sequences, and the discovery of actionable insights from event sequences. We summarize our contributions towards these goals in turn. The first problem was

Research Goal 1 (Summarizing Sequential Event Sequences) *Given an event sequence database, discover models that summarize the data so that it provides meaningful, interpretable insight.*

We proposed a set of methods that provide a richer summarization of event sequences, that go beyond the traditional definition of serial episodes. In Chapter 3, we studied the gaps between events. Existing methods, if they allow for gaps, treat them as undesirable noise, something to be minimized, or ignore them completely. We took a opposing view point towards gaps, rather than treating gaps as noise, we modeled delays explicitly, thereby rewarding consistent behavior. This allows us to capture patterns with long consistent delays between their consecutive events. We formalized the problem in terms of the Minimum Description Length principle. To discover good patterns sets in practice we introduced a greedy search algorithm, HOPPER, that iteratively builds up patterns by combining patterns form the model. To

discover patterns with long delays we used the AlignFar Algorithm introduced in Chapter 2. Experiments showed that on synthetic data Hopper 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.

Next, in Chapter 4, we studied summarization in terms of rules. While patterns express co-occurrence, rules express conditional dependencies between patterns and events. We again formalized the problem in terms of MDL. To discover rule sets, we explored two approaches Seqret-Candidates and Seqret-Mine. Seqret-Candidates constructs a rule set from a given set of candidate patterns whereas Seqret-Mine discovers a set of rules directly form the data. In Seqret, to avoid testing unpromising candidates, we only test extensions that occur statistical significant more often than expected. In the experiments we showed that Seqret discovers a meaningful summarization of the data in terms of rules, e.g. on the chess data, capturing the different variation after the opening moves.

Existing methods summarize event sequences in terms of surface level patterns — serial episodes defined over observed events. In Chapter 5 we explored *generalized* patterns, that is, patterns not only over observed *surface* level patterns but also over *generalized* events. *Generalized* events are events that can match a set of observed events. We considered the problem of discovering not only *generalized* patterns, but also the *generalized* events directly from the data. We proposed the greedy Flock algorithm that jointly discovers *generalized patterns* and *generalized events* directly from the data. Through experiments on synthetic and real-world data we have shown that Flock recovers patterns that are inclusive of rare instances, which would not have been classified as patterns when considered in isolation.

For our final contribution towards Research Goal 1, we explored summarization of sequential data for a specific domain, i.e. network flows. A network flow is a summary of network packages, e.g. a TCP session. Each flow has a timestamp and a set of attributes, e.g. ports, source and destination IPs etc. We developed a specific pattern language that can not only model patterns over attributes and different network flows, but also model some of the key characteristics of network flows, e.g. that patterns occur frequently but with different Src

Ip addresses. We again formalized the problem in terms of MDL and proposed a greedy search algorithm. We used the learned summarization to generate synthetic network flows, we did so by sampling from the model. Evaluation showed that we not only model in inter-flow dependencies but also preserve the temporal relation between flows.

Beyond summarization, we studied predictive models, where the goal is not only to describe event sequences but also use the model to predict upcoming events, bringing us to our second research goal we focused on,

Research Goal 2 (Discovering Actionable Summaries) *Given an event sequence database, find a summarization that enables action from the gained insight.*

Naturally, SEQRET (Chapter 4) provides a predictive summarization, as rules not only explain but also forecast upcoming patterns and events.

In Chapter 2 we explicitly tackled predictive summarization. We considered the problem of discovering a set of actionable patterns that jointly predict if and when target events will occur. We introduced the OMEN algorithm to discover a small set of non redundant predictive patterns. To overcome local minima we introduced an optimistic estimator. Our model is a set of tuples consisting of patterns and their associated delay distribution. The delay distribution not only states how likely we are to observe the target event but also when we can expect it. To avoid bias we model the delay distribution non-parametrically. Experiments showed that the OMEN score performs very well, at discriminating between predictive and non-predictive patterns, and that the OMEN algorithm recovers predictive patterns well.

Finally in Chapter 7, we investigated causal relationships in event sequences. We considered the problem of discovering a fully oriented causal graph over the different event types. Asking the question which events cause which other events. Knowing all the causal parents and the causal relation between cause and effect, naturally, also functions as a predictor of the effect. Unlike a purely predictive modelling, knowing the causal relations allows to make informed decision about interventions in systems. We based our approach on the Algorithmic Markov Condition [85], by which we identify the causal network as the one which minimizes the Kolmogorov Complexity. As the Kolmogorov Complexity is not computable we instantiated our model

with MDL, and showed under which assumptions we can identify the causal direction. Experiments showed that our method finds graphs that inherently make sense and match with existing knowledge.

In summary, this thesis presents novel methods for pattern discovery, rule-based summarization, predictive modeling, and causal discovery on event sequences. By formalizing problem in terms of the MDL principle our approaches result in small interpretable models that give insight into the data at hand.

8.2 EVALUATING UNSUPERVISED METHODS

In this section we discuss the challenges in evaluating pattern set miners. All methods proposed in Chapters 2 to 7 are unsupervised.[1] Whereas supervised methods can reasonably straightforwardly be evaluated in an objective manner (e.g. by comparing prediction accuracies), meaningfully and fairly evaluating unsupervised methods poses many challenges. In this section we will discuss these challenges and how we addressed them.

WITH KNOWN GROUND TRUTH To establish how well the proposed pattern set miners, presented in Chapters 2 to 5, recover the ground truth, we evaluated them on data with known ground truth — synthetic data. Even with known ground truth evaluation poses a challenge, let us consider the recall and precision metrics. Recall is defined as the fraction of correctly discoverd patterns over all true patterns in the data, formally,

$$recall = \frac{true\ positives}{true\ positives + false\ negatives} \quad .$$

Precision is the fraction of correctly discoverd patterns out of all reported patterns, formally,

$$precision = \frac{true\ positives}{true\ positives + false\ positives} \quad .$$

While the denominators are straight forward to define, $true\ positives + false\ negatives = number\ of\ planted\ patterns$ and $true\ positives + false$

1 We consider OMEN [Chapter 2] as an unsupervised method, as we are interested in discovering predictive patterns and hence evaluate the discovered patterns and not the predictive performance.

positives = *number of returned patterns*, it is not inherently clear how to define the number of true positives. Consider the case where *abcd* is the true pattern, and a method recovers *abc*, or *ab* and *bc*, or *acd* and *bc*. None of them are the true pattern, but arguably they are not wrong and capture some structure of the true pattern. The question now becomes how *correct* these returned patterns are. As throughout all chapters our objective is to discover a set of patterns, we also want to evaluate how well the true set of patterns, as a whole, was recovered. Suppose the ground truth set contains the patterns *abcx* and *abcy*. Now a method recovers only *abc*. Should we say that both *abcx* and *abcy* are 75% recovered? Or should we say that we recovered just one of them — either *abcx* or *abcy* — to 75%?

The naive way to count true positives is to tally up the exact recovered patterns, treating everything else as wrong. This is a very strict approach, as it not only fails to reward partial discoveries but actually penalizes them, putting them on the same footing as entirely spurious patterns. That is, a partial hit and a completely wrong pattern would be considered equally good.

To avoid that and reward partial discovery one possibility is to define a distance metric between patterns, e.g. Levenshtein distance. To compute the true positives for the recall metric we compute the most similar pattern for each planted pattern, e.g. *abc* matches *abcx* to 75%, and take the sum over all. To compute the number of true positives for the precision metric we do the analog for all reported patterns. This approach has some drawbacks. For example, it can lead to different counts of true positives when calculating recall and precision. In some cases, the number of true positives, when computed over all reported patterns, might even be higher than the actual number of planted patterns. It can also cause double counting, such as when a single discovered pattern matches multiple planted patterns best and is therefore credited more than once.

To alleviate these problems we propose to compute the number of true positives by constructing a flow network between discovered and planted patterns, the maximum flow is then considered the number of true positives. We provide the technical details on how we construct the flow network in Chapter 2, Section 2.6. This ensures that each planted respectively discoverd pattern contributes at most as one full discovery.

Throughout this thesis we considered different pattern languages that are more expressive, which adds another layer of complexity to the evaluation. Consider FLOCK, we not only want to evaluate how well we recover the patterns but also how well we recover the generalizations, how we evaluate this we explained in the respective chapters.

WITH UNKNOWN GROUND TRUTH Precision, recall, and its harmonic mean — the F1 score — are not the only way to measure the quality of discovered patterns. Another way to compare discovered models is to compare the likelihood of the data under the discovered model, or the MDL equivalence — the encoding cost. This does not require known ground truth and is therefore suitable to compare output of real-world results. This, however, raises other questions, like choosing the model class under which to evaluate the model. Unless two methods optimize for the same model class their encoding costs, respectively likelihoods, are only of limited comparability. Unlike F1, a compression or likelihood score does not allow direct interpretation of the quality of the results; a compression of 3% can be a good result if most of the data is noise, but on a different dataset, with lots of structure, it would be considered a poor result. In contrast F1, recall and precision each give an interpretable number of how good the results are, for example a recall score of 0.95 tells us that 95% of all relevant patterns were recovered.

To further evaluate the results we conducted a qualitative evaluation for all methods. We did this by manually analyzing the results, showing examples that demonstrated that a method discovers meaningful patterns that existing methods are not capable of discovering. Such kind of evaluation is limited to fairly well understood datasets or well understood patterns. A truly novel discovery in the data would require further validation or interpretation from a domain expert, we hence restricted our evaluation to datasets we can interpret. The goal of this kind of evaluation is not to measure the quality of the results but to show that the proposed methods is capable of discovering new insights that existing methods can not discover.

EVALUATING SYNTHETIC GENERATED NETWORK FLOWS So far we have discussed evaluation of discovered pattern sets, in Chapter 6 we use a learned pattern set for a domain specific task — generat-

ing synthetic network flows. As such we are not directly interested in the quality of the pattern set but the generated data based on the pattern set. To evaluate the quality of synthetic network flows is no trivial task, but has been studied [157], and the field of synthetic tabular data generation as a whole has established a set of metrics we can use [3, 40].

EVALUATING CAUSAL GRAPHS Similarly, the causal discovery method we proposed in Chapters 7, while it is an unsupervised task, there is no ambiguity about whether an edge was recovered or not. As such, when the ground truth is known, we can use well-established metrics for evaluation, like the Structural Hamming Distance (SHD) [87], and the Structural Intervention Distance (SID) [141].

Evaluating performance is challenging, and no single metric captures all aspects. However, in combination, that is testing on both synthetic and real-world data, along with manual inspection of real-world results, gives a good insight on the capabilities and shortcomings of a method.

8.3 LIMITATIONS

In this section we discuss the limitations of our methods.

For all proposed methods we assume a static process over time, that is, we assume the generating process is the same throughout the sequence, respectively for all sequences in the database. If this does not hold our approaches will likely still capture the relevant patterns, but they will do so in a suboptimal way. If we can detect different segments, or model each time point as a mixture of different generating processes, a more efficient and insightful model could be discovered. Dalleiger and Vreeken, explored this for tabular data [37], and Van Leeuwen and Siebes for incoming transactions over time [174]. Gautrais et al. [61] study segmentation from a different angle, identifying segments, over timestamped transactions, with shared item sets.

Our approaches require a given sequences or a set of sequences, however, event sequences are mostly recorded over time. An interesting future direction is to study how a live summary can be provided that summarizes the current generating process live.

To discover a set of patterns we design a pattern language and its encoding, this inherently introduces a bias about what kind of patterns we find; patterns that do not match this pattern language can, at best, be described in a suboptimal way. This means that we prefer certain patters over others, however we can also use this to our advantage, if we want to find patterns of certain shape we can design a pattern language to fit these kind of patterns very well.

So far we have discussed the impact of our modelling choices and how they impact the results. One of the key limitations in scaling to large alphabets is the combinatorial search. While we introduce optimistic estimators [Chapter 2], gain estimates [Chapter 5], and significance tests [Chapters 4] to speed up the search, they reach their limit on very large alphabets. In recent years differential based approaches have been introduced to address this problem [38, 51, 181]. Instead of discovering patterns through combinatorial search, pattern representations are learned through continuous optimization. These approaches, in essence, work by regularizing and constraining a network such that the learned weights are interpretable. To the best of our knowledge, no such approach has been proposed for event sequences. The Transformer [175] architecture has been very successful used to model event sequences, how to extract interpretable patterns from it is still an open problem. How to combine the noise robustness of MDL based approaches with combinatorial search with the speed of differential approaches is another promising future direction.

Finally, real-world data is often complex and comes with additional attributes, e.g. on the *Rolling Mill* dataset, events associated with measuring the width, the value of this measurement will be relevant e.g. when the measurement does not match the specifications future events will be different than when it does. We already do that to model the network flows in Chapter 6, but do not do it in general.

8.4 OUTLOOK

In this thesis we proposed richer pattern languages, going beyond the traditional definition of frequent serial episodes, used pattern mining for a domain specific problem, and studied the causality between events. In this section we discuss promising future directions that ad-

dress some of the current limitations of exploratory data analysis, with a specific focus on pattern mining.

In Chapter 7, we studied the causal relationships between event sequences. The ability to identify a system's causal relations is a powerful tool for gaining insight, as it enables both intervention and reasoning about the outcomes of those interventions. It would be interesting to further study causality in event sequences and connect it to methods presented in this thesis. Consider OMEN, presented in Chapter 2, once we understand which patterns cause which events, e.g. errors, we can make the changes to prevent the causing events from occurring and thereby the errors. Causal discovery relies on assumptions [138] one assumption is the one of causal sufficiency, i.e. no unobserved confounders. If we wrongly assume no hidden confounders, the causal graph we discover might be wrong, and implementing interventions might not results in the desired effects. Kaltenpoth and Vreeken [88] propose a way to detect confounding on continuous-valued variables in a tabular setting. How to detect confounding on event sequences is still an open research question.

In the Limitations Section, we discussed the scaling issue of combinatorial search based approaches and how differential approaches can address this. This line of research has the potential to complement existing combinatorial methods. While combinatorial methods do well on smaller datasets and have been shown to be very robust to high amounts of noise, they fail to scale to larger alphabets. This is where differential methods excel. To bring such approaches to different data modalities, like sequences and graphs, or increasing the expressivity of the discovered patterns, like generalized patterns, has the potential to open up new discoveries from datasets so far out of reach for combinatorial methods.

The ultimate goal of pattern mining is not to discover patterns, but to discover new insight into the data, that are useful in some manner. Most approaches, including our own, do not differentiate between knowledge already known and truly novel knowledge. While there exist approaches that aim to discover insights that are most surprising given some background knowledge [42], they require formalization of all knowledge, and do not provide a way to link discovered patterns to existing knowledge. Ideally, we are interested in patterns that are truly novel, or novel to us. The vast majority of discovered knowledge

is, unfortunately, not captured in queryable databases but in the form of text.

The recent emergence of foundation models, in the form of Large Language Models (LLMs), offers a potential solution to this. LLMs can essentially be used as a universal knowledge base, in combination with domain-specific publication databases and Retrieval-augmented generation (RAG) [102]. Future work could focus on which structures are already known and which ones are novel. Combined approaches that hide well-understood patterns that every domain expert knows are in the data, provide sources to the less understood ones, and highlight the ones that can not be linked to any existing knowledge, have the potential to increase acceptance and trust in the discovered patterns. To further increase trust, domain experts also need to be able to effectively explore the data and the patterns within, e.g. through interactive tools that show where in the event sequences a pattern occurs. While visualizations and interactive tools have been proposed [66, 101, 117, 146], less attention has been paid to understanding which types of visualizations and interactions actually help build trust in the discovered insights. How to do that is an open question and not so much a data exploration problem, but a human-computer interaction problem.

In this thesis, we explored richer pattern languages and, while combining existing approaches and introducing more expressive concepts, for example loops or sub-patterns will enable discovery of new insights. It's unlikely to exactly address the needs of domain experts; every domain will have its own specific requirements. In short, domain experts know best what kind of patterns they look for, however they often lack the expertise to design specialized pattern languages and the needed algorithms to discover them. While there exist methods that can discover patterns under a given set of constraints [81, 132] i.e. excluding specific cases from the pattern set. Future work could be working on approaches that allow for data exploration under custom pattern languages, without the need for extensive formalization and designing of optimization algorithms. Such an approach has the potential to speed up exploration by increasing iteration time.

To conclude this thesis, we studied exploratory methods for event sequences, focusing on summarization and predictive models that enable action. The models presented are easily interpretable and provide actionable insights that can support decision-making processes. Looking

ahead, we hope future research will build on and extend the ideas discussed in this section — particularly in the direction of scalability, integration of causal discovery, and the development of domain-adaptive pattern languages. Addressing these challenges will not only advance the theoretical foundations of pattern mining but also enhance its practical applicability, making it more accessible and impactful for domain experts across diverse fields.

Appendix

a

MINING SEQUENTIAL PATTERNS WITH RELIABLE PREDICTION DELAYS

In this appendix to Chapter 2, we provide additional technical details as well as additional metrics on the experiments presented.

A.1 REFINEMENT ALGORITHMS

In this section we give additional details ans pseudo-code on the refinement algorithm.

A.1.1 OMEN *Refinement*

When adding a pattern to our model we first refine, i.e. extend it, to the best version we can find, we do so in a greedy fashion. We give the pseudo-code for refining a pattern as Algorithm 14. We start with a pattern s (l. 1) and consider the extensions es and se, where $e \in \Omega$ and choose the one with maximal frequency. To do so efficiently, we only consider events e that are adjacent to pattern occurrences that are currently aligned to an interesting event (l. 3–4). The key idea is that extending a pattern makes it more specific, and hence reduce recall—while, by maintaining the current predictions, we maximize precision. We repeat this process until our optimistic estimator no longer gives a better estimation than the best seen pattern up to this point. We then return that pattern with lowest $L_p(S_R)$.

A.1.2 FOMEN *Refinement*

When we find a compressing pattern we greedily refine it to its most compressing form. We do this by combining our compressing pattern with the candidate pattern that has the highest intersection of predicted interesting events. We repeat this process until our optimistic estimator no longer estimates a better refinement. From the chain of

created patterns we return the most compressing version. As Algorithm 15 we give the pseudo code of the refinement procedure.

Algorithm 14: REFINE Pattern

input : predictive pattern p,
output: greedy refinement of pattern p with associated delay distribution ϕ_p

$p^* \leftarrow p \quad p' \leftarrow p$
while $\bar{L}_{p'}(S_R) < L_{p'}(S_R)$ **do**
  $H \leftarrow \{(i,j) \in A_{s'} \mid j \neq skip, X[i+1] = s'e\}$
  $T \leftarrow \{(i,j) \in A_{s'} \mid j \neq skip, X[i] = es'\}$
  if $\max_{e\in\Omega} |T| > \max_{e\in\Omega} |H|$ **then**
    $p' \leftarrow p' + \arg\max_{e\in\Omega} |T|$
  else
    $p' \leftarrow \arg\max_{e\in\Omega} |H| + p'$
  if $L_{p^*}(S_R) > L_{p'}(S_R)$ **then**
    $p^* \leftarrow p'$
return (p^*, ϕ_{p^*})

A.2 EXPERIMENTS

In this section we provide additional metrics on experiments presented in Chapter 2, Section 2.6.

A.2.1 *Additional Experiment Results*

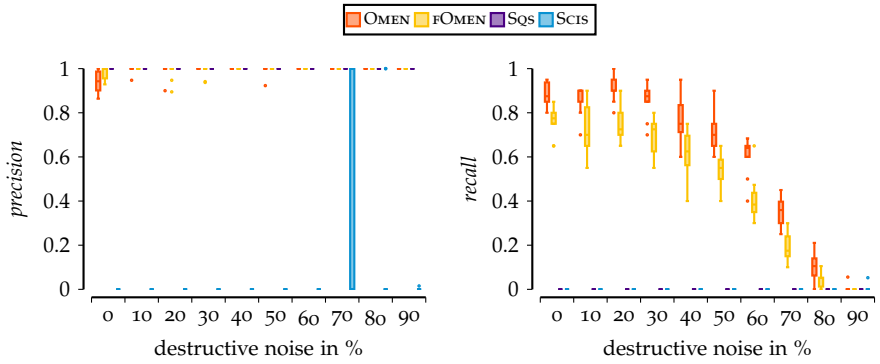
In Section 2.6 we show the F1 results for synthetic data. Here we provide the recall and precision results for destructive noise in Figure. a.1, additive noise in Figure. a.2, and for both additive and destructive (combined) noise in Figure. a.3. For all three settings we see that OMEN and FOMEN have near perfect precision whereas the recall scores reflect the F1 scores. From this we can conclude, as the signal to noise ration increases it becomes more difficult to find the planted patterns but our scores correctly filters non predictive patterns even under high noise.

Algorithm 15: fRefine

input : predictive pattern p,
output: refinement of pattern p and delay distribution ϕ_p

$p* \leftarrow p; \quad p' \leftarrow p$
do
  $\tilde{p} \leftarrow \arg\max ||\hat{S}_y^{\tilde{p}} \wedge \hat{S}_y^{p'}||_1$ // *Ignore* `used` *events for singleton patterns.*
  $C \leftarrow (C \backslash \{\tilde{p}, p'\}) \cup \Omega$
  $p' \leftarrow \arg\max_{p \in \{\tilde{p}p', p'\tilde{p}\}} ||S_z^p||_1$
  if $L_{p*}(S_R) < L_{p'}(S_R)$ **then**
    $p* \leftarrow p'$
while $\bar{L}_{p'}(S_R) < L_{p'}(S_R)$
return $(p*, \phi_{p*})$

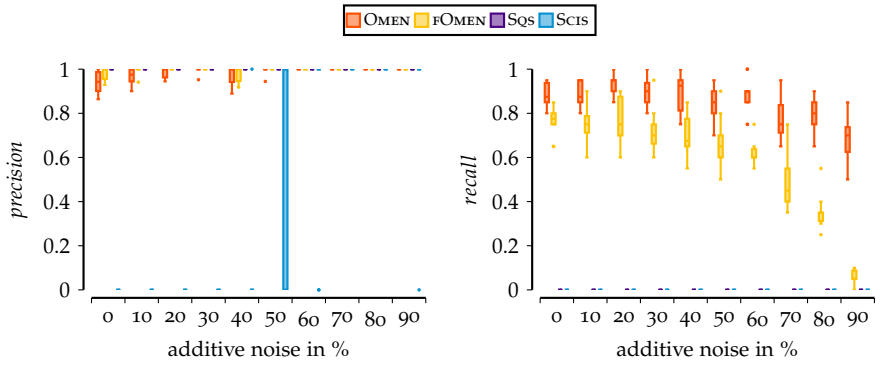
In Figures a.4 and a.5 we report the worst model discovered by Omen and fOmen respectively. The worst model is the one that has the largest difference between the number of bits needed to encode S_y compared to the ground truth model. We see that there are no big differences between the average and the worst case showing that Omen and fOmen return consistently good models.



(a) Precision under destructive noise.

(b) Recall under destructive noise.

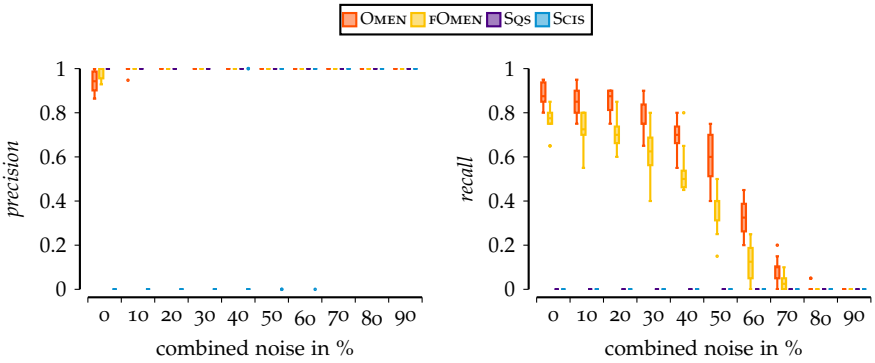
Figure a.1: [Higher is better] *Precision* and *recall* score result on synthetic data for destructive noise. We see that OMEN and FOMEN only return true positives and recall most patterns even under high noise.



(a) Precision under additive noise.

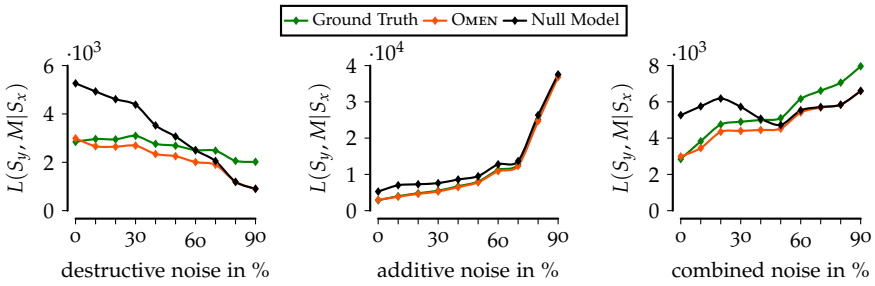
(b) Recall under additive noise.

Figure a.2: [Higher is better] *Precision* and *recall* score result on synthetic data for additive noise. We see that OMEN and FOMEN only return true positives and OMEN recalls most patterns even when 90% of interesting events are noise.



(a) (b)

Figure a.3: [Higher is better] *Precision* and *recall* score result on synthetic data for combined destructive and additive noise. We see that OMEN and FOMEN only return true positives and recall most patterns even under high noise



(a) (b) (c)

Figure a.4: [Lower is better] FOMEN discovers models close to the ground truth for destructive (a), additive (b) and combined (c) noise. Plots show bits needed to encode S_y given the null, planted and discovered model. We show for each noise level the experiment with the worst performance, i.e. where the difference, in bits, between discovered and planted model is greatest.

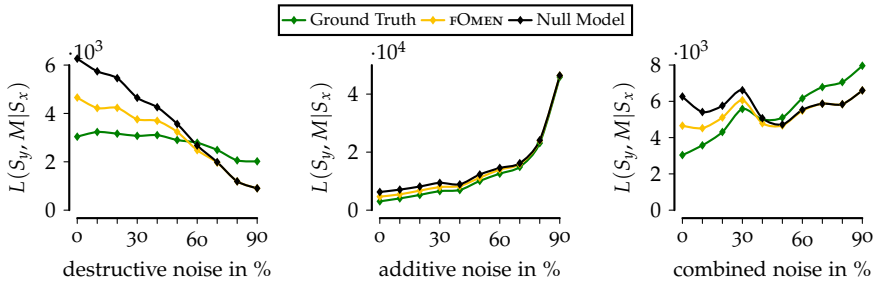


Figure a.5: [Lower is better] Omen discovers models close to the ground truth for destructive (a), additive (b) and combined (c) noise. Plots show bits needed to encode S_y given the null, planted and discovered model. We show for each noise level the experiment with the worst performance, i.e. where the difference, in bits, between discovered and planted model is greatest.

DISCOVERING SEQUENTIAL PATTERNS WITH PREDICTABLE INTER-EVENT DELAYS

In this appendix to Chapter 3, we provide additional details on method and experiments presented.

B.1 ALGORITHM

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

B.1.1 *Estimating Candidate Gain*

In this subsection, we provide the derivation of

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

where 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 $s' = s - z$ [10].

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,

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

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

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

B.1.2 *Mining Good Models*

ALIGNFAR To find a good initial alignment between p_1 and p_2, we use AlignFar [Chapter 2]. However, we can not use AlignFar directly, we first have to transform the data. We first summarize the functionality AlignFar, we give a full explanation in Chapter 2 Section 2.4.4.

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

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

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. $\arg\min_{d \in D} |\mu^* - d|$. With that, we have an initial alignment from p_1 to p_2.

Algorithm 16: ALIGNCANDIDATE

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

1 $A_N \leftarrow$ ALIGNNEXT$(C_{p_1\bullet}, C_{\bullet p_2})$
2 $A_F \leftarrow$ ALIGNFAR$(C_{p_1\bullet}, C_{\bullet p_2})$
3 $gain_N, A_N \leftarrow$ OPTIMIZEALIGNMENT$(p_1 \oplus p_2, A_N)$
4 $gain_F, A_F \leftarrow$ OPTIMIZEALIGNMENT$(p_1 \oplus p_2, A_F)$
5 **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 16 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 Chapter 3, and return the one for which we estimate a higher gain.

Algorithm 17: FILLGAPS

Input : p, index i
Output: pattern p'

1 **foreach** e between $p'[i]$ and $p'[i+1]$ in cover C **do** in order of frequency
2 $gain_1, A_1 \leftarrow$ ALIGNCANDIDATE(e,$p'[i+1:]$)
3 $gain_2, A_2 \leftarrow$ ALIGNCANDIDATE($p'[i:]$,e,)
4 $p' \leftarrow p'[:i] \oplus e \oplus p'[i+1:]$
5 **if** $gain_1 > 0 \wedge gain_2 > 0 \wedge L(D, M) > L(D, M \oplus p')$ **then**
6 $p_1^* \leftarrow$ FILLGAPS(p', i)
7 $p' \leftarrow$ FILLGAPS$(p_1^*, i + 1 + |p_1^*| - |p'|)$
8 **return** p'
9 **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 17. 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 4. We consider a toy input sequence shown in Figure b.1 (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. 4).

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, ...]`.

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

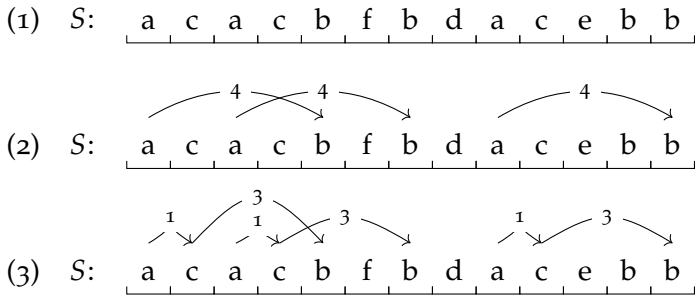


Figure B.1: Toy example of main search algorithm, (1) input sequence, (2) alignment between symbols `a` and `b`, and (3) pattern after 'filling in' symbol `c` between `a` and `b`.

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 B.1 (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 B.1 (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).

B.2 EXPERIMENTS

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

B.2.1 *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.

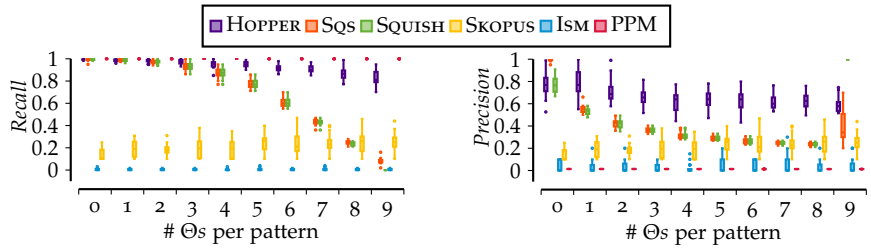
HOPPER We run all experiments, synthetic and real-world data, with the default parameter settings, that is a max delay of 200 and precision $p = 1$. We did not optimize max delay or precision. We set the max delay to 200 as we did not expect longer delays for any of the synthetic and real-world datasets. 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.

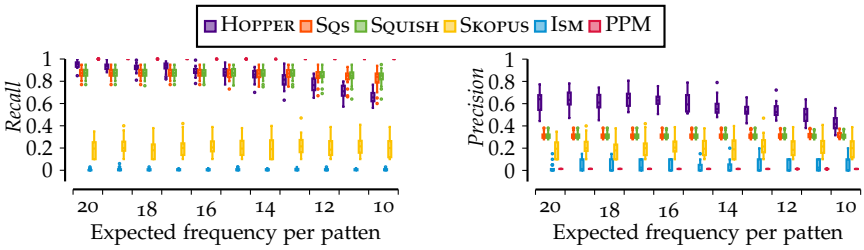
SKOPUS We use the default interesting measure (leverage), and the default parameter for smoothing and support (Laplace). For all experiments we set $k = 10$ i.e. we get the top 10 patterns, and the maximum length to $l = 10$. We set the parameters to match the ground truth of the synthetic experiments,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

PPM is parameter free.



(a) (b)

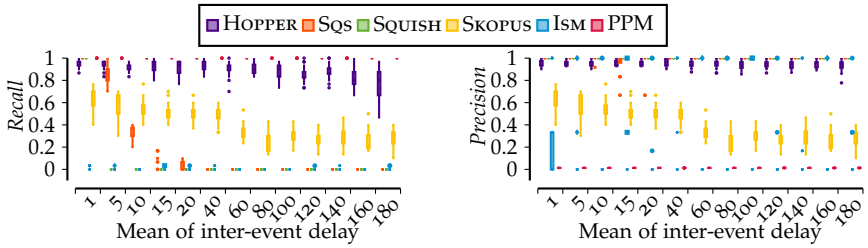
Figure b.2: Recall (a) and Precision (b) results for recovered patterns on synthetic data. We vary the number of delay distributions per pattern, from 0 to 9.



(a) (b)

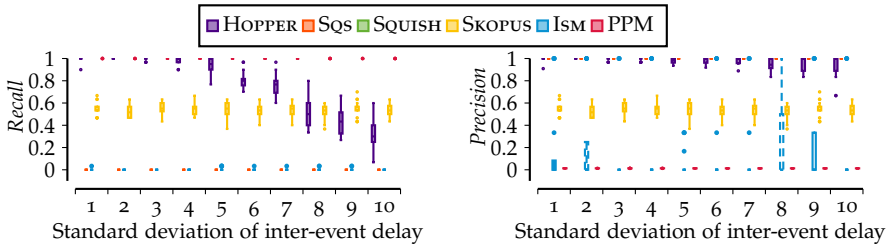
Figure b.3: Recall (a) and Precision (b) results for recovered patterns on synthetic data. We decrease the number of planted instances from 200 to 100. We sample for each planted instance, uniform at random, which of the 10 unique pattern to plant.

B.2.2 *Recall and Precision Results*



(a) (b)

Figure b.4: Recall (a) and Precision (b) results for recovered patterns on synthetic data. We vary the mean of the inter-evnet delay.



(a) (b)

Figure b.5: Recall (a) and Precision (b) results for recovered patterns on synthetic data. We vary the the variance of the inter-event delay.

B.2.3 *All Holidays*

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

HOPPER

- `[May 1st (155 days) National Holiday (83 days) 1st Christmas Day (1 day) 2nd 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

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

SQUISH

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

ISM

- [New Year's, May 1st]

PPM

- [no holiday]
- [no holiday]
- [no holiday]
- [National Holiday, 1st Christmas Day, 2nd Christmas Day, New Year's]
- [May 1st]
- [National Holiday, 1st Christmas Day, 2nd Christmas Day, New Year's]
- [May 1st]
- [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]

B.2.4 *Extended Real World Statistics*

				Hopper					Sqs				Squish			Ism			PPM				Skopus		
Dataset	$\|D\|$	$\|\Omega\|$	$\|\|D\|\|$	$\|P\|$	$\mathbb{E}(\|p\|)$	$\mathbb{E}(w)$	#Θ	*sec*	$\|P\|$	$\mathbb{E}(\|p\|)$	$\mathbb{E}(w)$	*sec*	$\|P\|$	$\mathbb{E}(\|p\|)$	*sec*	$\|P\|$	$\mathbb{E}(\|p\|)$	*sec*	$\|P\|$	$\mathbb{E}(\|p\|)$	$\mathbb{E}(w)$	*sec*	$\|P\|$	$\mathbb{E}(\|p\|)$	*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 b.1: 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 #Θ, 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.

C

MINING RULE-SETS FROM EVENT SEQUENCES

In this appendix to Chapter 4, we provide additional details on the SEQRET methods as well as a detailed complexity analysis. We detail the experiment setup and the synthetic data generation.

C.1 ALGORITHMS

In this section we give additional details on the SEQRET method.

C.1.1 *Rule Windows*

A rule window $S[i, j; k, l]$ captures the positions at which a rule occurs in a sequence S. Here, $S[i, j]$ is the window within which the rule head occurs and $S[k, l]$ is the window within which the rule tail occurs such that $j \leqslant k$. To avoid double counting and minimize gaps, we use minimal windows to identify the rule head patterns that trigger the rules. But what about the rule windows? For each rule head, we prefer the nearest minimal window of the rule tail pattern to complete the rule window. We prefer minimal windows as they minimize the gaps and treat the rule tail as a cohesive unit. If multiple minimal windows of the rule tail exist following the trigger, then we pick the nearest one so as to minimize the delay. Further, we restrict the search to windows that follow a user-set *max delay* ratio such that $k - j - 1/|tail(r)| \leqslant max\ delay$, and a *max gap* ratio such that $l - k + 1/|tail(r)| \leqslant max\ gap$ and $j - i - 1/|head(r)| \leqslant max\ gap$.

Algorithm 18 outlines the pseudo-code to find the best rule windows for a given rule.

The BESTRULEWIN method, however, assumes that none of the events forming the preferred windows for different rules have already been covered. As we start covering the sequence with these windows, however, it may happen that, at some point, events that are common to multiple rule tails have already been covered. In such cases, we look for the next best rule window. The next best rule window is the near-

Figure c.1: An illustration of potential rule windows for the rule $ab \rightarrow cd$. We pick the nearest minimal window of the rule tail as our preferred window.

est minimal window of the rule tail following the trigger such that the events forming the rule tail are not already covered. Algorithm 19 outlines the pseudo-code for the NextBestWin method.

As an example, consider the sequence $\langle a, a, b, b, c, c, d, d \rangle$ in Figure c.1 and rule $ab \rightarrow cd$. The minimal window $S[1, 2]$ captures the rule head that triggers the rule. Assuming none of the events in the sequence have already been covered, we pick $S[5, 6]$ as the rule tail window. The alternate windows for the rule tail are considered if and only if positions 5 or 6 have already been covered by other rules.

Algorithm 18: BestRuleWin

Input: *rule*, D
Output: *windows*

```
1 windows ← {};
2 triggers ←
    {S[i,j] | S ∈ D, (j − i + 1) − |head(rule)| ⩽ max gap ∗ |head(rule)|} ,
3   where S[i,j] is a minimal window of head(rule);
4 for S[i,j] ∈ triggers do
5 |   k,l ← indices of first minimal window S[k,l] of tail(rule)
  |     such that (k − j − 1) ⩽ max delay ∗ |tail(rule)| and
  |     (l − k + 1) − |tail(rule)| ⩽ max gap ∗ |tail(rule)|;
6 |   if S[k,l] exists then
7 |   |   windows ← windows ∪ {(rule, S[i,j;k,l])};
8 return windows
```

Algorithm 19: NEXTBESTWIN

Input: $win, cover, D$
Output: win'
/* win contains a pointer to the rule ($win.rule$) and the window in the format $S[i, j; k, l]$ */
$k', l' \leftarrow$ indices of first minimal window $S[k', l']$ of $tail(win.rule)$ such that $\forall e \in \{e \mid S[k', l'] \text{ covers } e\}$, $\nexists w \in cover$ where w covers e and $(k' - j - 1) \leqslant max\ delay * |tail(win.rule)|$ and $(l' - k' + 1) - |tail(win.rule)| \leqslant max\ gap * |tail(win.rule)|$;
$win' \leftarrow (win.rule, S[i, j; k', l'])$;
return win'

C.1.2 *Prune Algorithm*

In Algorithm 20 we show the pseudo-code for the PRUNE procedure. The general idea is iterative over all rules in PRUNE ORDER, that is we consider rules in order of lowest usage, highest encoded size, and lowest tail length, and remove all rules that harm compression.

Algorithm 20: PRUNE

Input: D, R
Output: pruned R
for $r \in R$ ordered by PRUNE ORDER **do**
  if r is not a singleton rule **then**
    if $L(D, R \setminus \{r\}) < L(D, R)$ **then**
      $R \leftarrow R \setminus \{r\}$;
return R

C.1.3 *CandRules Algorithm*

In Algorithm 21 we give the pseudo-code for the CANDRULES procedure. Given a rule, it returns a set of candidates, it does so by extending the rules with events that occur significant more frequent than expected, We give a detailed explanation in Chapter 4 Section 4.4.

Algorithm 21: CANDRULES

Input: $D, \Omega, rule$
Output: *candidates*

```
1  candidates ← {};
2  windows ← BESTRULEWIN(rule, D);
3  for position ∈ {h_0, h_1, ..., h_|head(rule)|} ∪ {t_0, t_1, ..., t_|tail(rule)|} do
       /* h represents rule head and t represents rule tail */
4      for e ∈ Ω do
5          count ← |{w ∈ windows | w contains e in the gap at position}|;
6          p_{e^c} ← 1 − supp(ϵ→e)/|D|; /* e^c refers to the complement of e */
7          expected ← |windows| − Σ_{w∈windows} (p_{e^c})^{|g_w|};
           /* g_w refers to the gap in w at position */
8          if count significantly greater than expected then
9              candidates ← candidates ∪ {INSERT(rule, e, position)}
                   /* INSERT inserts e to rule at position */
10 return candidates
```

Algorithm 22: SPLIT

Input: Pattern p
Output: Set of rules R

```
1  R ← ∅
2  i ← 0
3  while i < |p| do
4      R ← R ∪ (p[0, i], p[i + 1, |p|])
5      i ← i + 1
6  return R
```

C.1.4 *Split Algorithm*

In Algorithm 22 we show the pseudo-code for the SPLIT procedure. Given a pattern it splits it ino a set of rules, e.g. *abc* into $\epsilon \rightarrow abc$, $a \rightarrow bc$, and $ab \rightarrow c$.

C.2 TIME COMPLEXITY ANALYSIS

In this chapter we provide a complexity analysis.

C.2.1 *Time Complexity of the Rule-Set Mining Problem*

To evaluate the time complexity of the problem, let us split the problem into two parts - one, to find the optimal cover given a rule set, and two, to find the optimal rule set. For simplicity, let us assume a single long sequence S in the database, drawn from the alphabet Ω.

Given a rule set R, we know that a cover is a many-to-one mapping between the events in S to rules in R. In other words, it is a permutation with replacement of the rules in R over $|S|$ events. Therefore, we can compute the number of possible covers as $|R|^{|S|}$. The worst-case time complexity of the first part of our problem is

$$\mathcal{O}(|R|^{|S|}) \ .$$

Now let us compute the time complexity of the second part of our problem. The longest rule that can occur in S would be of length $|S|$. The total number of rules possible would be the sum of the number of rules possible per size, with size ranging from 1 to $|S|$. Considering that rules can be built from sequential patterns, let us first compute the number of sequential patterns possible for size k. A sequential pattern is a permutation of the alphabet with replacement. Therefore, for size k we get $|\Omega|^k$ possible sequential patterns. Now, including the possibility of an empty-head, we can choose k positions to split the pattern into a rule head and a rule tail. Thus, for size k, we can compute the number of rules possible as $k * |\Omega|^k$. The total number of rules possible is then given by

$$\sum_{k=1}^{|S|} k * |\Omega|^k \ .$$

A rule set being a subset of all possible rules, we can compute the number of possible rule sets as the size of the power-set. Since we retain the singleton rules in every possible rule set, we can compute

the number of valid rule sets as $2^{\sum_{k=1}^{|S|} k*|\Omega|^k - |\Omega|}$. The worst-case time complexity of the second part of our problem is then given by

$$\mathcal{O}(2^{\sum_{k=1}^{|S|} k*|\Omega|^k - |\Omega|}) \approx \mathcal{O}(2^{|\Omega|^{|S|}}) .$$

C.2.2 *Time Complexity of* SEQRET-MINE

Let us now analyze the time complexity of our solution. We will analyze each part of the problem separately. We consider the worst-case where all the events in the database occur as a single long sequence S. The set of distinct events form the alphabet Ω.

TIME COMPLEXITY OF COVER Given a rule set R, we first compute the complexity of finding the rule windows. To do so, the method looks for all rule triggers and for each rule trigger, finds the nearest minimal window of the rule tail. Iterating over S and looking for triggers of each rule $r \in R$ results in a worst-case time complexity of $\mathcal{O}(|S| * \sum_{r \in R} |head(r)| * max\ gap)$. Ignoring the *max gap* parameter that stays constant irrespective of the problem size and upper bounding the size of any rule head by $\max_{r \in R} |head(r)|$, denoted by *max_head_size*, we can rewrite the same as

$$\mathcal{O}(|S| * |R| * max\_head\_size) .$$

To complete the rule window for each trigger, BESTRULEWIN next looks for the nearest minimal window of the rule tail until the maximum allowed delay. Given a trigger, looking for a rule tail, for any rule r, requires computational time in the order of $\mathcal{O}(|tail(r)| * max\ delay * |tail(r)| * max\ gap)$. Once again ignoring the constant parameters and using *max_tail_size* to upper bound the size of a rule tail, we can rewrite this as $\mathcal{O}(max\_tail\_size^2)$. Thus, we can compute the worst-case time complexity of BESTRULEWIN as

$$\mathcal{O}\left(|S| * |R| * (max\_head\_size + max\_tail\_size^2)\right) . \tag{C.1}$$

As triggers are bounded by minimal windows, and only one minimal window can exist per starting or ending position, the number of triggers per rule is upper bounded by $|S|$. Since BESTRULEWIN finds only the one nearest minimal window of the rule tail for each trigger, we

can upper bound the total number of rule windows returned to $|R| * |S|$. The next step in COVER is to sort the rule windows in WINDOW ORDER. This incurs a time complexity of

$$\mathcal{O}(|R| * |S| * \log_2(|R| * |S|)) \ . \tag{C.2}$$

The final step in COVER is to consider each rule window in the sorted order and cover the sequence. However, there could arise cases where the considered rule windows are in conflict with previous rule windows which already covered the same events. This in turn leads to the execution of NEXTBESTWIN. Each time NEXTBESTWIN is called for a rule trigger, it looks for the next nearest minimal window of the rule tail until the maximum allowed delay. If such a rule window is found, then it is added to the sorted list of rule windows maintaining the order. A single call to NEXTBESTWIN for a trigger of rule r, in the worst-case, incurs computational time in the order of $|tail(r)|^2$ to find the next nearest minimal window of the rule tail (ignoring the parameters *max gap* and *max delay*). Suppose W denotes the sorted list of rule windows at any point in time. Once (if) the next best rule window is found, the method incurs additional computational time in the order of $\log_2(|W|)$ to find the position of insertion using a binary search.

The question is how many such calls to NEXTBESTWIN could happen in the worst case. We could also upper bound the size of W by the same value. To answer this question, let us consider when NEXTBESTWIN is called. It is called whenever an event that participates in a rule window is already covered by a previous rule window. From the point of view of a rule trigger, each event following it until a limit determined by *max delay* and *max gap* times the rule tail, can potentially participate in a rule window. The SEQRET starts with one such rule window and looks for the next best rule window if and only if any of the participating events is already covered. Further, the next best rule window omits the previously covered events. Therefore, the maximum number of times NEXTBESTWIN gets called is limited by the number of events following the rule trigger, given by $|tail(r)| * (max\ delay + max\ gap + 1)$. Ignoring the constants for the purpose of complexity analysis, we can rewrite it as $|tail(r)|$. Over all triggers for all rules, we can then compute

the worst-case time complexity of finding the next best windows and adding them to the sorted list of rule windows as

$$\mathcal{O}\left(\sum_{r\in R}|S|*|tail(r)|*\left(|tail(r)|^2+\log_2(|W|)\right)\right) ,$$

where W is the list of rule windows. Once again, as worst-case, we use *max_tail_size* to rewrite the same. Further, we can limit the size up to which W can grow by the number of times NextBestWin gets called, i.e in the order of $|R|*|S|*max\_tail\_size$. Putting it all together, we find the total computational time for all calls to NextBestWin to be in the order of

$$\begin{aligned}&\mathcal{O}(|R|*|S|*max\_tail\_size*\log_2(|R|*|S|*max\_tail\_size)\\&\quad+|R|*|S|*max\_tail\_size^3)\end{aligned} \tag{c.3}$$

Finally, using the ordered list of rule windows W, we cover the sequence S. As singleton rules are also included in R, it is guaranteed to cover the entire sequence in one iteration over all the rule windows in W (in practice, it will be much lesser as many events get covered by a single rule window). Therefore, we can compute the worst-case time complexity to loop over the list of rule windows and cover the sequence S as

$$\mathcal{O}(|R|*|S|*max\_tail\_size) . \tag{c.4}$$

Thus, from equations c.1, c.2, c.3 and c.4, we can compute the total worst-case time complexity of Cover as

$$\begin{aligned}&\mathcal{O}(|R|*|S|*(max\_head\_size+max\_tail\_size^2)\\&\quad+|R|*|S|*\log_2(|R|*|S|)\\&\quad+|R|*|S|*max\_tail\_size\\&\quad+|R|*|S|*max\_tail\_size*\log_2(|R|*|S|*max\_tail\_size)\\&\quad+|R|*|S|*max\_tail\_size^3) .\end{aligned}$$

Considering only the dominating terms, we get

$$\begin{aligned}&\mathcal{O}(|R|*|S|*(max\_head\_size\\&\qquad+max\_tail\_size^3\\&\qquad+max\_tail\_size*\log_2(|R|*|S|*max\_tail\_size)) .\end{aligned} \tag{c.5}$$

TIME COMPLEXITY OF SEQRET-MINE Next, we analyze the time complexity of the greedy miner, SEQRET-MINE. Let us consider a single iteration of the miner. Let R' be the candidate rule set at that time point. Then, SEQRET-MINE grows the rule set by searching for a new rule that improves the encoding cost by extending each rule, at each position, with their significant neighbors. As worst-case, let us assume that the miner had to search over all rules, at all positions. Further, let us assume that all events in the alphabet Ω are significant (although this is impossible). To simplify the computations, we use $\max_{r' \in R'} |head(r')|$ as the *max_head_size* and $\max_{r' \in R'} |tail(r')|$ as the *max_tail_size* to upper bound the lengths of $head(r')$ and $tail(r')$ for any $r' \in R'$. Then, the computational time of the search will be in the order of $\mathcal{O}(|R'| * (max\_head\_size + max\_tail\_size) * |\Omega|)$. Once a new rule is added to the rule set, SEQRET-MINE tries to prune the rule set by removing each non-singleton rule. The computational time required by PRUNE will be in the order of $|R'| - |\Omega|$. Considering only the dominating term, we can thus conclude the worst-case time complexity of each iteration as

$$\mathcal{O}(|R'| * (max\_head\_size + max\_tail\_size) * |\Omega|) \,, \tag{c.6}$$

where R' denotes the candidate rule set at that time point. Next, we try to analyze the number of iterations possible before the algorithm converges.

We know that in each iteration, SEQRET-MINE adds a new rule to the current rule set only if the addition improves the encoding cost. Similarly, a rule is pruned from the current rule set only if the removal improves the encoding cost. If the encoding cost cannot be improved anymore, then the algorithm halts. In other words, SEQRET-MINE will never revert back to a rule set from which it grew in the past. Therefore, the maximum number of iterations is upper bounded by the number of unique rule sets possible. In Section C.2.1, we saw that the number of possible rule sets is in the order of $\mathcal{O}(2^{|\Omega|^{|S|}})$.

CACHING FOR FASTER RUNTIME In practice, however, we observe the number of rule sets considered by the greedy approach to be much smaller. Further, we cache the rule windows found for each rule as and when they are first encountered. Therefore, if the same rule is present in a future rule set, we do not recompute the rule windows. In

other words, BestRuleWin is invoked only once per rule. The same is true for CandRules. We cache the neighbors found for each rule as and when they are first encountered. As a result, the time complexity in practice would be much lower, even if Seqret-Mine attempted the worst-case possibility of all unique rule sets before converging. Further, we do not reconsider rules once pruned in the future iterations.

C.3 EXPERIMENTS

In this section we describe the experiment setup as well as the synthetic data generation.

C.3.1 *Setup*

We ran all experiments on an Intel Xeon Gold 6244 @ 3.6 GHz, with 256GB of RAM. For the methods Poerma and Poermh, the JVM max heap size was increased upto 64 GB. Both these methods discover partially ordered rules from long event sequences, albeit with different definitions of rule support. To ensure fair comparison, we constraint our synthetic data generation to rules where constituent events appear in lexicographical order, and re-arrange the partially-ordered rules found to this order. We set a time limit of 24 hours for all methods except cossu. As Cossu took a very long time to complete across all experiments, we increased the time limit for Cossu alone to 48 hours. Across all synthetic experiments except where the rule tail size was varied, Seqret-Mine completed within a few seconds to 1 hour max. In experiment varying rule tail size, Seqret-Mine took upto 3 hours in a few instances indicating that data with highly interleaved rules take up more time in rule search. We report the runtimes on real datasets in Table C.1.

C.3.2 *Synthetic Data Generation*

Given an alphabet as input, we first generate a random rule set. We take in as parameters the rule set size, the rule-head size, the rule-tail size and the rule confidences. We also parameterize whether or not the rule heads occur as independent patterns, i.e for a rule $X \rightarrow Y$, whether $\epsilon \rightarrow X$ exists or not. If $\epsilon \rightarrow X$ doesn't exist, then X is only as

frequent as expected by chance. Given these parameters, we randomly select events from the alphabet to form the rule heads and the rule tails, and add them to the rule set.

Next, using the rule set as ground truth, we generate the sequence database. We first generate an initial sequence using all the empty-head rules, and then insert the rule tails wherever the non-empty-head rules have triggered. We take in as parameters an initial sequence size and noise percentage. By noise, we mean the events in the sequence that can be covered only using one of the singleton rules. Therefore, given a noise percentage, we uniformly sample from the singleton rules, i.e the alphabet, to generate the stipulated percentage of the initial sequence size. Following this, we uniformly sample from the empty-head non-singleton rules and fill them into random positions to generate the remaining sequence. Finally, we go over the generated sequence, identify rules that have been triggered and insert the corresponding rule tails as per the specified rule confidences.

As for the delays and gaps, we take in as parameters a delay probability, i.e probability with which the data generation algorithm skips positions following a rule trigger, and a gap probability, i.e probability with which the data generation algorithm skips positions within rule tails. Note that the insertion of rule tails will alter the sequence size and the noise percentage. We keep the gap and delay probabilities low at 0.1 and 0.2 respectively. To run SEQRET, we set the *max delay* and the *max gap* both to 2.

Dataset	Runtime
JMLR	4 h
Presidential	8 h
POS	3 h
Lifelog	2 h
Ordonez	3 s
Ecommerce	11 h
Rollingmill	22 h
Lichess	17 h

Table C.1: Runtime of SEQRET-MINE for different datasets.

SUMMARIZING EVENT SEQUENCES WITH GENERALIZED SEQUENTIAL PATTERNS

In this appendix to Chapter 5, we provide additional details about the FLOCK method and the experiments, presented in Chapter 5.

D.1 ALGORITHM

In this Section we provide additional details on the FLOCK Algorithm.

D.1.1 *Cover*

Given a Model M, we want to find the shortest description of sequence database D given model M, i.e. that cover C that minimizes $L(D|M)$. To this end we need for each pattern $p \in CT$, all subsequences $S[j_1, j_2, \ldots, j_{|p|}]$ where p matches, that is window set W.

In Chapter 5, we explained what a valid cover is and what it means for two windows to be in conflict. Here we describe how we actually find a valid cover C.

We show pseudocode in Algorithm 23. Given a set of windows W we first sort W by j_1. The general idea is now to move a pointer i from left to right through W where all windows to the left of i are conflict free whereas to the right we still have to resolve all conflicts. We iterate over the window list W while maintaining three pointers, the first pointer i points to the lowest non conflict free window. The second one, greatest window gw, is used to keep track of the current greatest window after i, that is not in conflict with a greater window. The third one, greater window searcher gws, is used to find conflicting greater window then the current greatest window gw.

We initialize i with 0 such that it points to the first window in W, at this point this is, trivially, also the greatest window we have seen so far. We now search for a greater window that overlaps with gw, line 6. If we find one we update our gw pointer. Once we can't find a greater window we remove all windows that overlap with gw. To this end,

Algorithm 23: GREEDYCOVER

```
   input  : set of windows W and D
   output : list of ordered windows covering D
1  i ← 1
2  sort W by j_1
3  while i < |W| do
4      gw ← i
5      gws ← gw + 1
6      while W[gws].first ⩽ W[gw].last do
7          if W[gws] > W[gw]
             and W[gws] in conflict with W[gw] then
8              gw ← gws
9          gws ← gws + 1
10     if gw = i then i ← i + 1 ;
11     wr ← i
12     while W[wr].first ⩽ W[gw].last do
13         if W[wr] in conflict with W[gw] then
14             remove wr from W
15         wr ← wr + 1
16 return W
```

we start again at position i and increase a window remover pointer wr until the first position of $W[wr]$ is larger than the last position of $W[gw]$, at this point we can be sure to have considered all windows that might overlap. If i points to the greatest window we increase i by one. Note i might overlap with gw in this case we can still simply remove it, i then just points to the next window, which is exactly what we want. We repeat this process until i points to the last window in W, at this point each event in the database D is covered by exactly one window.

WINDOW SEARCH To efficiently find all windows given a pattern p we use an inverted index. We add triples (i, j, k) to a set O, where i refers to the sequence j to the next event $S_i[j]$ to be tested against $p[k]$. We initialize O with all $(i, j+1, 2)$ where $S_i[j] = p[1]$. We increase j until $S_i[j] = p[k]$, this means we found the next event in S_i that

matches the next unmatched event in p. Hence we increment the j and k pointer by one, $(i, j+1, k+1)$, and additionally add $(i, j+1, k)$ to O. To illustrate why we add the second case, where we do not increase k, consider the case with sequence *abbc* and pattern *abc*, to capture both windows, i.e. the one with the first and the second b we need one instance to move over b without matching it.

We continue this process until $k = |p| + 1$, at this point we found a window w that matches p and can remove the triple form O. We also remove a triple from O if the respective windows grows larger than the maximum window length of $|p| + n|p|$. We continue this process until O is empty, at this point we have found all windows of p.

D.1.2 *Generalization refinement example*

Example of applying a set of generalized refinements $\Omega_g^{\oplus}$ to a set of generalized events Ω_g. Let us consider the following case we have $\Omega_g^{\oplus} = \{(\alpha_{id}, \{c, d\}), (\gamma_{id}, \{\alpha, e, f\})\}$ and $\Omega_g = \{\alpha, \beta\}$ where $\alpha = \{a, b\}$ and $\beta = \{g, h\}$. We now apply $\Omega_g^{\oplus}$ to Ω_g. Since Ω_g already contains a generalization α we extend it with the specified events, hence $\alpha = \{a, b, c, d\}$. The generalization γ on the other hand not does not yet exist, we hence add a new generalization γ to Ω_g, hence $\Omega_g = \{\alpha, \beta, \gamma\}$ where $\gamma = \{\alpha, e, f\}$. As $\Omega_g^{\oplus}$ does not specific any additions for β it is not affected.

D.1.3 *Optimistic Gain Estimation*

In this section, we will explain how we compute the gain estimation $\Delta\overline{L}(p, \Omega_g^{\oplus})$. That is we want to estimate by how our encoding cost $L(D, M)$ changes by adding pattern p to CT and applying a set of generalization refinements $\Omega_g^{\oplus}$ to Ω_g.

The cost of encoding a new pattern, or generalized event, can be directly computed as shown in Section 5.3. However, when extending a generalization i.e adding new elements to an existing generalization, we have to update the cost of this generalization. To compute the difference in cost we subtract the cost of the old generalization and add the cost of the new one. Since we allow generalization to contain other generalization, and we encode these differently, we also have to do so here. So when extending an existing generalization α with k' new gen-

eralization $\beta_1, \ldots, \beta_{k'}$ and m' new events. With that, we estimate the increased cost as,

$$\begin{aligned}\Delta\overline{L}(\alpha) = &- \log\binom{i-1}{k} + \log\binom{i-1}{k+k'} \\ &- \log(|\Omega_o^*|) + \log(\Omega_o') \\ &- \log\binom{|\Omega_o^*|}{m} + \log\binom{\Omega_o'}{m+m'}\end{aligned} \tag{d.1}$$

where Ω_o^* are those observed events not defined by the old nested generalizations $\beta \in \alpha$, i.e. $\Omega_o^* = \Omega_o \backslash \bigcup_{\beta\in\alpha_i} fl(\beta)$ and Ω_o' those observed events not defined by the extended generalizations i.e. $\Omega_o' = \Omega_o^* \backslash \bigcup_{j=1}^{k'} fl(\beta_j)$, k is the number of generalizations in α and m the number of events in α before the extension. This covers the estimated cost on the model side.

Next, we consider the difference in the cover cost. To estimate the gain of a new model we have to estimate how the cost of the C_p, C_m and C_s changes. To avoid recovering the data for each candidate we estimate the effect a pattern extension, new generalization, or extending a generalization has on the usage of all patterns and singletons. We will later explain how we estimate these usages for all cases. Given the estimated usage changes for all patterns, we can compute the difference in bits needed for the pattern code stream C_p,

$$\begin{aligned}\Delta\overline{L}(C_p) = \sum_{p\in P\cup\{p'\}} &usg_{new}(p) * \log(usg_{new}(p)) \\ &-usg_{old}(p) * \log(usg_{old}(p))\end{aligned} \tag{d.2}$$

For the specification sequence. Given an estimation how often each pattern instance will be used for each pattern, we can create specification sequence $C_{s\ new}$ representing the new specification, and with $C_{s\ old}$ we denote the old specification sequence.

As we can't compute the pattern structure at this point we assume all generalization to be independent, to more accurately estimate the gain we make the same assumption for the existing patterns. The difference in encoding cost is then,

$$\Delta\overline{L}(C_s) = L(C_{s\ new}) - L(C_{s\ old}) \quad .$$

As we do not have a good way to estimate the number of needed gaps and fill codes we do not estimate the difference for the meta stream. All operations we consider create a new pattern based on an existing pattern. Importantly we do not remove the existing one i.e. the old one is still in the P. We infer a frequency estimate on the number of instances we count in Algorithm 8. Next, we consider all different refinement cases and describe how we estimate the new frequencies for the specific cases.

Case 1 (New pattern): This case we already explained in Chapter 5 and included here, in extended form, for completeness. Creating one new pattern p^* with p_1 and p_2 where p_2 is another pattern. From F we get c, how often p_2 follows p_1. The usage of our new pattern $p_1 \times p_2$ is simply c, while we reduce the usage of p_1 and p_2 by c. Hence $usg_{new}(p^*) = c$ and $usg_{new}(p_1) = usg_{old}(p_1) - c$, $usg_{new}(p_2) = usg_{old}(p_1) - c$. If we extend a p_1 by more than one singleton c is simply the minimum count out of all extensions, we then subtract c from all extensions. The frequency of how often events of a generalization are used we adjust proportionally to the pattern frequency. That is for each event $e \in fl(\alpha)$ where generalization α in the patterns p_1, respectively p_2 we adjust the usage to $usg^{p_1}_{new}(e) = usg^{p_1}_{old}(e) - \frac{c \; usg^{p_1}_{old}(e)}{usg_{old}(p_1)}$. Usage of same e in the new pattern p^* is $usg^{p^*}_{new}(e) = \frac{c \; usg^{p_1}_{old}(e)}{usg_{old}(p_1)}$.

Case 2 (New pattern, extended with Generalization): Next, we consider the case where we extend a pattern with an existing or newly formed generalization α. To estimate the frequency of the patterns, we can treat this case analog to Case 1, c is now just the sum of all counts of events $e \in fl(\alpha)$. The usage of $e \in \alpha$ is set to the count of e, whereas existing generalizations are adjusted as in Case 1.

Case 3 (Extending an existing Generalization): Extending an existing generalization with new elements does effect all patterns that use this generalization. The effects are hence not contained to the pattern we refine. First we just consider p^* which contains a generalization α which we extend with event e. We increase the the frequency of pattern p^* by the number of counts of e. Of course we reduce the usage of e by the same amount. The estimated usage of event e in p^* is naturally also the same.

Algorithm 24: PRUNEGEN

input : pattern p, generalization extensions $\Omega_g^{\oplus}$, cover C
output: pruned generalization extensions $\Omega_g^{\oplus}$, updated cover C

```
forall (α_id, R) ∈ Ω_g^⊕ do in order of increasing usg(e)
    if Δ̄L(p', Ω_g^⊕\(α_id, R)) > Δ̄L(p', Ω_g^⊕) then
        Ω_g^⊕ ← Ω_g^⊕\(α_id, R)
return Ω_g^⊕, C
```

To estimate the usage of other patterns p' that contain α. We search for all windows of p' and assume that all *new* windows are used. That is all windows that did not match p^* before the extension of α with e. From the windows we can directly infer the respective frequency, which we use as a usage estimation.

D.1.4 *Pattern search*

In Chapter 5, we describe the main search procedure, in this section we will expand on that explanation, providing more details.

GENERALIZATION PRUNING We show pseudocode in Algorithm 24. When we initially build new generalization and extend existing ones we estimate the usage of the individual elements in the generalization. Once we have actually computed a cover we have actual usage counts. Hence we test, for each added event e, if under these counts we still get a gain when adding e.

REFINE INTERLEAVING Before adding pattern p to our model, we search within the gaps of p for possible generalization. We provide pseudocode in Algorithm 25. At this point, we have a cover C that includes pattern p. We count for all gaps of pattern p how frequent each event is in each respective gap. The frequency of all events between the $i-1$ and the i^{th} event in pattern p is given by $c_i = counts[i]$. We refine one gap at a time, we start with the gap with the most frequent event. Per gap, we test all events in order of frequency, if we add a second event we create a new generalized event that matches the first and second event, analog for the third, forth, etc. event.

Algorithm 25: REFINEINTERLEAVING

input : pattern p, generalization extensions $\Omega_g^{\oplus}$, cover C
output: refined pattern p', extended generalization extensions $\Omega_g^{\oplus}$, updated cover C

for $i = 0; i < |p| - 1; i++$ **do**
  $counts[i] \leftarrow$ get frequencies for all events between $p[i]$ and $p[i+1]$
sort *counts* by $\max counts[i]$
forall $c \in counts$ **do**
  sort c by frequncy in decending order
  forall $e \in c$ **do**
    $p', \Omega_g^{\oplus\prime} \leftarrow$ extend p and $\Omega_g^{\oplus}$ with e
    if $\Delta\overline{L}(p', \Omega_g^{\oplus\prime}) > 0$ **then**
      $p, \Omega_g^{\oplus} \leftarrow p', \Omega_g^{\oplus\prime}$
return $p, \Omega_g^{\oplus}, C$

Algorithm 26: SIMPLIFY

input : Pattern set P and generalization Ω_g, cover C
output: Pattern set P and generalization Ω_g, cover C

while P changes **do**
  $P, \Omega_g, C \leftarrow$ MERGE(P, Ω_g, C)
  $P, \Omega_g \leftarrow$ FLATTEN(P, Ω_g)
return P, Ω_g, C

SIMPLIFY The SIMPLIFY algorithm simply calls MERGE and FLATTEN until convergence. This algorithm is applicable to any kind of pattern set P. The required generalized alphabet Ω_g is simply the empty set, and the cover C is computed using GREEDYCOVER.

PRUNING The pruning step is only done once, as a final step. We test for each pattern if removing it improves the model. If it does we remove it otherwise we keep it.

IMPLEMENTATION DETAILS After we test 100 candidates back-to-back without any actual gain we break the current search loop, i.e.

Algorithm 27: PRUNE

input : Pattern set P and generalization Ω_g, cover C
output: Pattern set P and generalization Ω_g, cover C

```
forall p ∈ P do
    if L(D, P) > L(D, P\{p}) then
        P ← P\{p}
        update C and Ω_g accordingly
return P, Ω_g, C
```


move on to the next iteration. In preliminary experiments, this did not have an effect on the results.

D.1.5 *Dependence structure*

When adding a pattern to our model we compute its dependency structure. Meaning we choose for each generalized event that earlier generalization that minimizes the specification cost for this generalization. This is, essentially, equivalent to picking that generalization with the lowest conditional entropy. More precisely when the penalty of the prequential encoding is ignored the encoding cost is equivalent to the conditional entropy. The specification cost of one generalization without the epsilon. $\sum_{i=1}^{usage(p)} -\log\left(\frac{usg(e|d)}{\sum_{c\in fl(\alpha)} usg(c|d)}\right)$ The inner part is just the probability of $p(e|d)$. Using Bayes' theorem we can transform that into $\frac{p(e,d)}{p(d)}$, summing over all usages is equivalent to summing over all combinations each weighted by its joined probability. Making it equivalent to the conditional entropy, defined as $H(Y \mid X) = -\sum_{x\in X, y\in Y} p(x,y) \log \frac{p(x,y)}{p(x)}$. In practice however we directly compute the bits needed by the prequential encoding, which is asymptotical equivalent to picking that generalization with the lowest entropy. In practice it can be cheaper to encode a specifications of a generalization independently of any previous generalization, hence we also allow that.

D.2 EXPERIMENTS

In this section we give further details about the experiment setup, evaluation and additional examples reported by FLOCK.

D.2.1 *Experiment Setup*

SKOPUS reports the top-k patterns, where k is a hyperparameter. In preliminary experiments we tested setting k to the number of all reported instances of FLOCK. The number instances of a generalized pattern p are the unique instances that match p. In the setting we considered these where between 100 and 200 instances. Since SKOPUS did not terminate with 24h, we set k to 50 and limited the pattern length to 10.

The word2vec architecture [121] has been shown to be good at training an embedding where items that occur in similar contexts are placed close to each other, while originally proposed for text the architecture lends itself to arbitrary sequences over a discrete alphabet. In short, we train an embedding into 4 dimensions using the word2vec architecture [121] with a window size of 10.

To extract generalizations from this embedding, we use DBSCAN with a maximum distance of $\epsilon = 0.2$ and with a minimum number of samples of 3. Next, we replace elements in D contained within a cluster with a new generalized event, representing that cluster. Finally, we apply SQS on the modified dataset. The groups found by DBSCAN we consider our generalizations and the pattern found by SQS are our patterns. We tuned the hyperparameters to produce clusters as close as possible to the true planted generalizations while also trying to avoid spurious clusters. We did this on synthetic generated data where each generalization was only used within one pattern.

The *ECG* dataset is based on the first record (id 300.1) of the MIT-BIH ST Change Database.[1] We subsampled the record, replacing each 5 subsequent values with their average, and transformed the result into a relative sequence by replacing each value with the difference to the previous value. Finally, using SAX [105] we discretize the sequence to 200 events.

1 https://physionet.org/content/stdb/1.0.0/

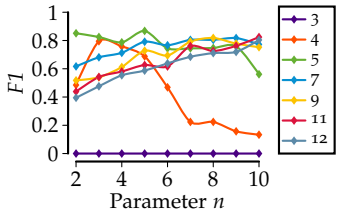


Figure d.1: Parameter sensitivity analysis: On seven datasets with pattern lengths from 3 to 12, each pattern includes two generalization. We see FLOCK is robust against the parameter choice. The dataset with a pattern length of 4 stands out, with half the events being generalization true patterns become hard to distinguish from random correlations.

On the datasets *ECG*, *Short-ECG*, *Moby*, and *JMLR* we run FLOCK with gap parameter $n = 2$, in d.2.2 we show that small values of n works better for datasets with a low amount of structure.

D.2.2 *Parameter Analysis*

FLOCK comes with one user parameter n the factor of how many gaps we allow relative to the pattern length. We consider a setting with patterns of different lengths, 3 to 12, each pattern contains two generalization. We report the $F1$ of FLOCK given the parameter n.

D.2.3 *Real-World Pattern Examples*

In Table d.1 we show a selection of patterns and generalization reported by FLOCK on real-world datasets. Note, the rich generalization β of the *JMLR* dataset was discovered with the default gap parameter of $n = 10$.

D.2.4 *Ablation Study*

FLOCK consists of several subroutines, we evaluate the impact of disabling REFINEINTERLEAVING and SIMPLIFY. We evaluate the performance on synthetic data, we use the same setting as for the experiment shown in Figure 5.2(b), with 400 planted pattern instances. We report the $F1$

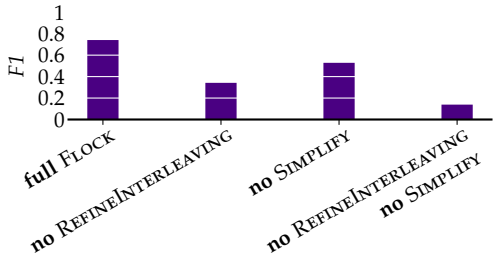


Figure d.2: Ablation study on synthetic with subroutines turned off. We observe that without REFINEINTERLEAVING and SIMPLIFY the $F1$ score decreases by a large margin.

score. The results show that REFINEINTERLEAVING and SIMPLIFY provide key functionality that greatly improves performance.

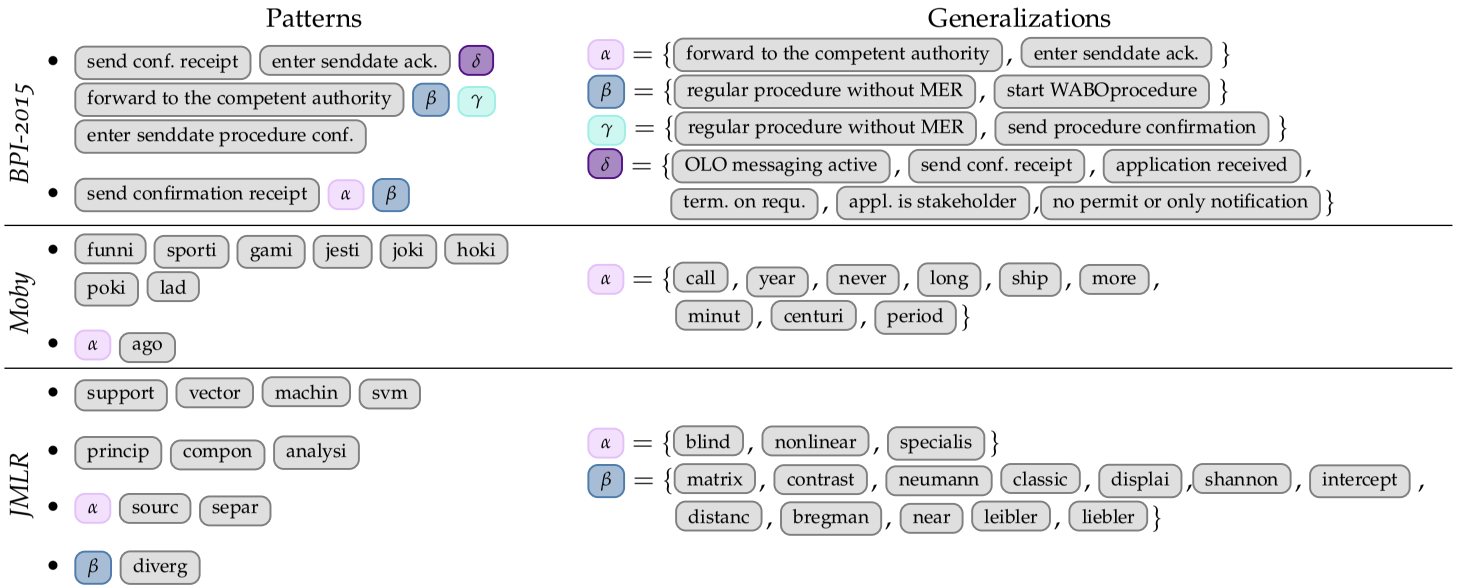


Table d.1: Selection of found patterns on real-world datasets. We see that FLOCK discovers rich generalization and patterns that use multiple generalization. On the *BPI-2015* dataset, we see two patterns using the same generalization.

				Flock		Sqs	Squish
Dataset	$\|D\|$	$\|\|D\|\|$	$\|\bar{S}\|$	$\%L$	t	$\%L$	$\%L$
ECG	1	107395	107395	97.2	17k	96.7	98.9
Short-ECG	1	3384	3384	98.2	1	98.3	98.7
BPI-2015	1199	51867	43.36	79.1	43k	60.7	63.2
Rolling Mill	1000	51390	51.39	63.2	1k	49.9	52.9
Moby	1	105719	105719	99.3	0.6k	99.3	99.4
JMLR	788	75646	96	96.7	0.7k	96.6	97.6

Table d.2: Extended results on real datasets. We given number of sequences, $|D|$, total number of events $||D||$, and average sequence length $|\bar{S}|$. For each method we give the relative length under our model. For Flock we provide the runtime in seconds.

CAUSAL DISCOVERY FROM EVENT SEQUENCES BY LOCAL CAUSE-EFFECT ATTRIBUTION

In this appendix to Chapter 7, we provide proofs and additional details on the experiments.

E.1 THEORY

In this section we provide the proofs of the theorems of Chapter 7.

E.1.1 *Proof - Identifiability on Instant Effects*

Theorem 1. *Let S_i be an event sequence generated by a Poisson process as per Eq. (7.2) and S_j be an effect of S_i as per Eq. (7.5), with, low noise $\lambda_j < (1 - \alpha_{i,j})\lambda_i$, and a trigger probability $\alpha_{i,j} < 1$.*

In the case of exclusively instant effects, i.e. $\phi_{i,j}(d) = \delta(d)$, where $\delta(d)$ is the Dirac delta function, the MDL score in the true causal direction is lower than in the anti-causal direction, i.e.

$$\lim_{n_i \to \infty} L(S_j \mid S_i, \Theta_1) + L(S_i \mid \Theta_1) < L(S_i \mid S_j, \Theta_2) + L(S_j \mid \Theta_2) \ .$$

Proof. Let n_i be the number of events in S_i and n_j the number of events in S_j. As $n_i \to \infty$

$$L(S_i) = n_i H(\phi_{i,i}), \qquad L(S_j|S_i) = n_i H(\mathcal{B}(\alpha_{i,j}))$$

where $\mathcal{B}$ is the Bernoulli distribution. In the reverse direction, we have

$$L(S_j) = n_j H(\phi_{j,j}), \qquad L(S_i|S_j) = n_i H(\mathcal{B}(\alpha_{i,i})) + (n_i - n_j) H(\phi_{i,i})$$

To show

$$L(S_i) + L(S_j|S_i) < L(S_j) + L(S_i|S_j) \tag{e.1}$$

$$n_i H(\phi_{i,i}) + n_i H(\mathcal{B}(\alpha_{i,j})) < n_j H(\phi_{j,j}) + n_i H(\mathcal{B}(\alpha_{i,i})) + (n_i - n_j) H(\phi_{i,i}) \tag{e.2}$$

$\alpha_{i,j} = \alpha_{i,i}$ since every event that does not cause an i, can not be explain by j in the reverse direction, hence

$$n_i H(\phi_{i,i}) + \cancel{n_i H(\mathcal{B}(\alpha_{i,j}))} < n_j H(\phi_{j,j}) + \cancel{n_i H(\mathcal{B}(\alpha_{i,i}))} + (n_i - n_j) H(\phi_{i,i}) \tag{e.3}$$

$$n_i H(\phi_{i,i}) < n_j H(\phi_{j,j}) + n_i H(\phi_{i,i}) - n_j H(\phi_{i,i}) \tag{e.4}$$

$$\cancel{n_i H(\phi_{i,i})} < n_j H(\phi_{j,j}) + \cancel{n_i H(\phi_{i,i})} - n_j H(\phi_{i,i}) \tag{e.5}$$

$$n_j H(\phi_{i,i}) < n_j H(\phi_{j,j}) \tag{e.6}$$

$$H(\phi_{i,i}) < H(\phi_{j,j}) \tag{e.7}$$

$$\tag{e.8}$$

For $H(\phi_{i,i}) < H(\phi_{j,j})$ to hold $n_i > n_j$.

$$n_i \propto \lambda_i \,, \qquad n_j \propto \alpha_{i,j}\lambda_i + \lambda_j \tag{e.9}$$

$$\tag{e.10}$$

$$\lambda_i > \alpha_{i,j}\lambda_i + \lambda_j \tag{e.11}$$

$$\lambda_i - \alpha_{i,j}\lambda_i > \lambda_j \tag{e.12}$$

$$(1 - \alpha_{i,j})\lambda_i > \lambda_j \tag{e.13}$$

$$\tag{e.14}$$

It directly follows that $H(\phi_{i,i}) < H(\phi_{j,j})$, and hence for $n_i \to \infty$

$$L(S_j|S_i, \Theta_1) + L(S_i|\Theta_1) < L(S_i|S_j, \Theta_2) + L(S_j|\Theta_2) \,.$$

□

E.1.2 *Proof - Identifiability on Delayed Effects*

Theorem 2. *Let S_i be an event sequence generated by a Poisson process as per Eq. (7.2) and S_j be an effect of S_i as per Eq. (7.5), such that $H\left(\phi_{j,j}\right) > H(p(;\theta_{i,j})) + \alpha_{i,j}^{-1} H\left(\mathcal{B}(\alpha_{i,j})\right) + \alpha_{j,j}^{-1} H\left(\mathcal{B}(\alpha_{j,j})\right)$, where H denotes the entropy and $\mathcal{B}$ the Bernoulli distribution.*

Then the matching in the anti-causal direction $\Delta_{j\to i}$ of the effect S_j to the cause S_i has a worse MDL score than the true matching $\Delta_{i\to j}$, i.e.

$$L(S_j \mid S_i, \Theta_{i\to j}) + L(S_i \mid \Theta_i) < L(S_i \mid S_j, \Theta_{j\to i}) + L(S_j \mid \Theta_j) \,.$$

We will show that the delays between S_i itself, i.e.$\Delta_{i\to i}$, and the delays between S_j to S_i, i.e. $\Delta_{i\to j}$, are equivalent.

Proof. We consider a source event S_i with exponentially distributed delays, i.e. $\Delta_{i\to i} \sim \exp(\lambda_i)$. Consider any event $t_k \in S_j$, then the intensity of observing an event in S_i at time $t > t_k$ is given by λ_i. The distribution of the delay to the next event in S_i is exponential with $\lambda = \lambda_i$. Thus, the difference between

$$L(S_i|S_j) - L(S_i) = \sum_{d_k \in \Delta_{i\to j}} \log(\lambda_i) - d_k/\lambda_i - \sum_{d_l \in \Delta_{i\to i}} \log(\lambda_i) + d_l/\lambda_i = 0 \,.$$

It remains to show that the likelihood in the causal direction is better when conditioning effect on the cause, i.e. $L(S_j|S_i) < L(S_j)$.

GAIN BY $i \to j$ For i to cause j it has to provide information about j, that is the cost of selecting which i events cause j, and with what dealys. Additional it has to ofset the cost which j events do not have to be encoded as a self delay. Formally this is,

$$n_i H(\mathcal{B}(\alpha_{i,j})) + n_{i,j} H(p(;\theta_{i,j})) + n_j H(\mathcal{B}(\frac{n_{i,j}}{n_j})) < n_{i,j} H(\phi_{j,j}) \quad ,$$

where $n_{i,j}$ are the number of events in j caused by i, and $H(p(;\theta_{i,j}))$ is the entropy of distribution described by the pdf p.

As $n \to \infty$,

$$L(S_j) = n_j H(\phi_{j,j}) \qquad L(S_j|S_i) = n_i H(\phi_{i,j}) + (n_j - n_{i,j}) H(\phi_{j,j}) + n_j H(\mathcal{B}(\alpha_{j,j}))$$

To show,

$$L(S_j) > L(S_j|S_i)$$

$$n_j H(\phi_{j,j}) > n_i H(\phi_{i,j}) + (n_j - n_{i,j}) H(\phi_{j,j}) + n_j H(\mathcal{B}(\alpha_{j,j})) \tag{E.15}$$

$$n_{i,j} H(\phi_{j,j}) + \cancel{(n_j - n_{i,j}) H(\phi_{j,j})} > n_i H(\phi_{i,j}) + \cancel{(n_j - n_{i,j}) H(\phi_{j,j})} + n_j H(\mathcal{B}(\alpha_{j,j})) \tag{E.16}$$

$$n_{i,j} H(\phi_{j,j}) > n_i H(\phi_{i,j}) + n_j H(\mathcal{B}(\alpha_{j,j})) \tag{E.17}$$

we can substitute $n_i H(\phi_{i,j}) = n_{i,j} H(p(;\theta_{i,j})) + n_i H(\mathcal{B}(\alpha_{i,j}))$

$$n_{i,j} H(\phi_{j,j}) > n_{i,j} H(p(;\theta_{i,j})) + n_i H(\mathcal{B}(\alpha_{i,j})) + n_j H(\mathcal{B}(\alpha_{j,j})) \quad \text{(e.18)}$$

Now, note that the number of caused items $\alpha_{i,j} n_i = n_{i,j}$ and $\alpha_{j,j} n_j = n_{i,j}$, then it follows

$$n_{i,j} H(\phi_{j,j}) > n_{i,j} H(p(;\theta_{i,j})) + \frac{n_{i,j}}{\alpha_{i,j}} H(\mathcal{B}(\alpha_{i,j})) + \frac{n_{i,j}}{\alpha_{j,j}} H(\mathcal{B}(\alpha_{j,j})) \quad \text{(e.19)}$$

$$H(\phi_{j,j}) > H(p(;\theta_{i,j})) + \frac{1}{\alpha_{i,j}} H(\mathcal{B}(\alpha_{i,j})) + \frac{1}{\alpha_{j,j}} H(\mathcal{B}(\alpha_{j,j})) \quad \text{(e.20)}$$

Hence, we show that $i \to j$ is identifiable. □

E.1.3 *Proof - Path Identifiability*

Theorem 3. *Given an event sequence S generated by a causal structure G^*, let S_i be a source node of G^* and S_v be a descendant of S_i, where there exists a path $i \to j \to \cdots \to v$ in G^*.*

Then, the gain in the causal direction of the path $g(i \to v \mid \Theta) - g(v \to i \mid \Theta)$ is greater.

Proof. We begin by proving that the path identifiability holds for a triplet of nodes $i \to j \to v$, by constructing a new alignment from $i \to v$.

From $\Delta_{i \to j}$ and $\Delta_{j \to v}$ we can construct $\Delta_{i \to v}$. For each $d_k \in \Delta_{i \to j}$, there is a corresponding $d_l \in \Delta_{j \to v}$, i.e. the trigger time of the triggered event. To construct $\Delta_{i \to v}$, we consider the following cases:

1. If $d_k = \infty$ for $d_k \in \Delta_{i \to j}$, then $d_k \in \Delta_{i \to v}$, is set to $d_k = \infty$.
2. Let $d_l \in \Delta_{j \to v}$ be the delay of event a of type j where a has been caused by delay d_k. If $d_k \neq \infty$ for $d_k \in \Delta_{i \to j}$ and $d_l = \infty$ then then $d_k \in \Delta_{i \to v}$, is set to $d_k = \infty$.
3. Let $d_l \in \Delta_{j \to v}$ be the delay of event a of type j where a has been caused by delay d_k. If $d_k \neq \infty$ for $d_k \in \Delta_{i \to j}$ and $d_l \neq \infty$ then then $d_k \in \Delta_{i \to v}$, is set to $d_k = d_k + d_l$.

As $\Delta_{i \to v}$ is another valid alignment, and i remains a source node, the guarantees of Theorem 1 and 2 hold, i.e. the path is identifiable. This

extends to a path of arbitrary length. Consider an additional edge $v \to w$, then we construct the alignment $\Delta_{i \to w}$ by considering the delays of $\Delta_{i \to v}$ and $\Delta_{v \to w}$.

Furthermore, we can show that the true path $j \to v$ has a better gain than the shortcut $i \to v$, so that we can remove the shortcut in the pruning stage. (1) Since $i \to j$ and $j \to v$ are independent processes, it follows either $\mathrm{Var}(\phi_{i,v}) > \mathrm{Var}(\phi_{j,v})$, or $\alpha_{i,v} > \alpha_{j,v}$ and by that a more costly description of v.

(2) If $j \to v \notin G$, we can construct a new function

$$f_{i,v} = f_{j,v}(f_{i,j}(S_i, \Delta_{i \to j}) \cup N_j, \Delta_{j \to v})$$

by Theorem 2 it follows that edge $i \to v$ improves our score.

(3) Assume $j \to v \in G$ then each individual v event that is matched to by $i \to v$ is already matched to by $j \to v$, and from (1) we know it does so cheaper, hence we get no gain by adding the shortcut $i \to v$.

□

E.1.4 *Consistency*

Proof. Here we will show that $L(S^n, \Theta)$ asymptotically behaves like *BIC*. $L(S, \Theta)$ directly corresponds to the log likelihood, which we rewrite as $\log p(S^n | \Theta, G)$ Our approach can be instantiated with arbitrary delay distribution, to show consistency we have to upper bound the number of parameters by $\mathcal{O}(\log n)$, this trivially holds for the parametric setting we focus on here, because $|pa(i)| \in \mathcal{O}(\log n)$ [120]. The encoding of the graph G is independent of n, i.e. fixed for a given network, hence in $\mathcal{O}(1)$. Finally this results at,

$$\log p(S^n | \Theta, G) + c \log n + \mathcal{O}(1)$$

we set $c = \frac{d}{2}$ where d is the number of free parameters, arriving at the *BIC* score. □

From Haughton [77] and Chickering [25] we know that *BIC* identifies a Markov equivalence class of the true DAG. For the identifiability of undirected edges we refer to Theorem 1 and 2. □

E.1.5 *Connection to Hawkes Processes*

The key difference between a linear Hawkes process and our model is the assumption of direct triggers, that is the mechanism of one event and one event only causing another. In a linear Hawkes process 'cause events' increase the intensity and therewith the probability of events occurring. However, one can generally not label for a specific event another as the 'cause', as each event is the result of a multitude of causes.

In this section, we are going to explore under which conditions we can identify a Hawkes process under our causal model.

Given an event sequence $S_j = \{t_k\}_{t_k=0}^{n_j}$ generated by a linear Hawkes process,

$$\lambda_j(t) = u_j + \sum_{i \in pa(j)} \sum_{t_k < t, t_k \in S_i} v_{i,j}\,(t - t_k)\ . \tag{e.21}$$

We construct a set of primary causes for each event $t_k \in S_j$ as

$$C_j = \left\{ (t, i, t_k) \mid t \in S_j,\ (i, t_k) = \underset{(i,t_k),\ i \in pa(j), t_k < t, t_k \in S_j}{\arg\max} v_{i,j}(t - t_k) \right\}\ , \tag{e.22}$$

where we consider as primary cause of an event that past event with the highest influence at time point t from a causal parent $i \in pa(j)$. Using these delays, we construct an alignment (mapping of delays) as

$$\Delta_{i \to j} = \{\beta(t_k) \mid t_k \in S_i\} \qquad \beta(t_k) = \begin{cases} t - t_k & \text{if } v_{i,j}(t - t_k) > u_j \\ \infty & \text{else} \end{cases}\ , \tag{e.23}$$

where $t = \arg\max_{t \in S_j \wedge \exists (t,i,t_k) \in C_j} v_{i,j}(t - t_k)$ if $\nexists (t, i, t_k) \in C_j$ then $\beta(t_k) = \infty$. We consider $t_k \in S_i$ a cause of an event $t \in S_j$ if it is the primary cause of t and t_k has no stronger influence on any other event of S_j. Finally, the influence has to be stronger than that of the base intensity of S_j.

To be able to identify a causal edge between S_i and S_j the improvement gained by this alignment must outweigh the edge cost $L(i \to j)$.

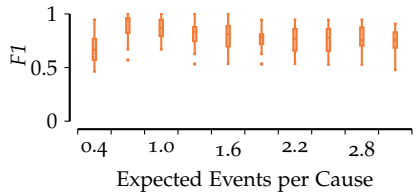


Figure e.1: DAG recovery on data generated by a Hawkes process.

For this, there are two conditions: firstly, the number of primary cause events from i to j, i.e. those instances where an event from S_i has the maximum influence on an event from S_j, must be large enough. This is the case as long as $|\Delta_{i\to j}|$ increases with n_j, i.e. the total number of events of S_j. Then, in the limit $n_j \to \infty$ the number of primary cause events is large enough to offset the constant edge cost.

The achievable score gain is obtained by constructing a delay distribution from $|\Delta_{i\to j}|$ as

$$\phi_{i,j}(d) = \begin{cases} 1 - \alpha_{i,j} & \text{if } d = \infty \\ p_{\Delta_{i\to j}}(d) \cdot \alpha_{i,j} & \text{else} \end{cases} \qquad \text{where } \alpha_{i,j} = 1 - P(\Delta_{i\to j} = \infty)\ . \tag{e.24}$$

If this density $\phi_{i,j}(d)$ fulfills the conditions of Theorem 2, we can identify S_i as a parent of S_j in the limit of $n_j \to \infty$.

In conclusion, CASCADE can identify a causal pair generated under a Hawkes process, if there exist sufficiently many events from $S_i \to S_j$, where $v_{i,j}(t)$ has the strongest influence on $\lambda_j(t)$ for some of the t. By aligning the delays of these events, we can identify the causal edge and recover the underlying causal structure.

E.1.6 *Empirical Evaluation*

[From Chapter 7] To empirically evaluate how effectively CASCADE recovers the true DAG on data generated by a Hawkes process, we generate synthetic data using the *tick* library[1]. We vary the intensity of the excitation function, i.e., the expected number of events generated per cause. We show the results in Figure e.1. We observe that CASCADE

1 https://x-datainitiative.github.io/tick/

performs best when generation is close to our assumptions, i.e. when there is, on expectation, one effect per cause or fewer, but still demonstrates strong performance across all settings.

E.1.7 *Consistency of Algorithm*

Theorem 4. *Given an event sequence S, where each individual subsequence S_i was generated as per Eq. (7.5) by an underlying causal graph G^*. Assuming all $\Delta_{i\to j}$ are the true causal matchings. Under the Algorithmic Markov Condition,* CASCADE *recovers the true graph G^* for $n \to \infty$.*

Proof. We begin by proving that in the first step, CASCADE identifies a true source node of G^*. We denote the parents of i in the true causal graph G^* as $pa(i)$, whilst we write for the parents in the graph maintained by CASCADE as $pa'(i, G)$.

Let $i \in [p]$ be a node of G^* without parents, i.e. a source. By Theorem 3, for a path $i \to \cdots \to v \in G^*$ the gain in the causal direction $g(i \to v \mid \Theta)$ is greater than the gain in the reverse direction $g(v \to i \mid \Theta)$.

For all nodes $v \in [p], v \neq i$, v is either an descendant of i or unrelated.

1. If v is a descendant of i, then the gain of $g(i \to v \mid \Theta)$ is greater than the gain of $g(v \to i \mid \Theta)$.
2. If v is unrelated to i, then the gain in both sides is 0.

Hence it follows, that for a node i where $pa(i) = \varnothing$,

$$\max_{j \in C} g(j \to i \mid \Theta) - g(i \to j \mid \Theta) = 0 \,.$$

On the other hand, consider a node i with parents $pa(i) \neq \varnothing$. Then, as G^* is a DAG, there exists an ancestor v of i, where v is a source node, i.e. $pa(v) = \varnothing$. For that v, it holds that the gain from v to i is greater than the gain from i to v. Hence, for a node i with parents, it holds that

$$\max_{j \in C} g(j \to i \mid \Theta) - g(i \to j \mid \Theta) > 0 \,.$$

Therefore, by taking the argmin over all nodes, CASCADE identifies the true source node of G^*, i.e.

$$\arg\min_{i \in C} \max_{j \in C} g(j \to i \mid \Theta) - g(i \to j \mid \Theta) \implies pa(i) \cap C = \varnothing \,.$$

EDGE ADDITION Now, we show that CASCADE always identifies a true causal edge for $n_i \to \infty$. First, note that given a source node i, there do not exist any incoming causal edges in the graph G^*,

$$pa(i) \cap C = \varnothing \implies \nexists j \in C : j \to i \in G^* .$$

Hence, by fitting outgoing edges only, we test all possible edges for i and never add a false oriented edge,

$$\forall j \in C : i \to j \in G^* \implies j \to i \notin G .$$

Finally, we recall that the true causal graph G^* is the graph that minimizes the description length of the data as per the Algorithm Markov Condition. Hence, adding a true causal edge to the graph will result in a lower description length, i.e.

$$\forall j \in C, i \to j \in G^* : L(S_j | S_{pa'(j,G)}, \Theta') > L(S_j | S_{pa'(j,G) \cup i}, \Theta' \cup \theta_{i,j}) .$$

EDGE REMOVAL Consider the node i, where $pa(i) \cap C = \varnothing$, and given a graph G where all true causal edges have been added, i.e.

$$\forall j \in \bar{C} : \forall j \to v \in G^* : j \to v \in G .$$

Then, for i it holds that

$$\forall j \in pa(i) : i \to j \in G .$$

It follows, that $pa'(i, G) \supseteq pa(i)$. By the Algorithm Markov Condition, the shortest description length of the data is achieved by the true causal graph G^*. Hence, a superset of the true parents of i will result in a higher description length, and it holds that

$$L(S_i | S_{pa'(i)}, \Theta') > L(S_i | S_{pa(i)}, \Theta) .$$

Therefore, by testing that subset of the parents of i results in a lower description length, CASCADE identifies the true parents of i.

OVERALL CONSISTENCY For $n_i \to \infty$, we note that in each step for the node i it holds that

1. i has no parents in the candidate set $pa(i) \cap C = \varnothing$.

2. We add no false oriented edges to the graph, as $pa(i) \cap C = \varnothing \implies \nexists j \in C : j \to i \in G^*$.
3. We add all true edges $i \to j$ to the graph G, i.e. $\forall j \in C, i \to j \in G^* : L(S_j|S_{pa'(j,G)}, \Theta') > L(S_j|S_{pa'(j,G)\cup i}, \Theta' \cup \theta_{i,j})$.
4. For i, the current graph G contains a superset of all true parents, i.e.$pa'(i, G) \supseteq pa(i)$, while the description length of the data is minimized by the true graph $L(S_i|S_{pa'(i)}, \Theta') > L(S_i|S_{pa(i)}, \Theta)$.

Hence, by repeating the edge addition and pruning in a topological order, in the limit of $n_i \to \infty$ under our causal model and by the Algorithm Markov Condition, CASCADE identifies the true causal graph G^*. □

E.2 EXPERIMENTS

In this section we provide additional detail on the synthetic data generation and the experiment setup. Additional we provide further metrics on the synthetic experiments. For the real-world data we provide additional results.

E.2.1 *Synthetic Experiments*

We generate synthetic data according to our causal model. We discretize the timestamps to 1 million unique timestamps. Throughout the experiments we vary the following parameters:

- Variables: Number of unique events types p.
- Edges: The total number edges in the generating causal graph $G*$.
- Delay Distribution: For all synthetic experiments we generate delays according to geometric distribution (as a discretize instantiation of the exponential).
- Delay Distribution Parameter: For each causal edge we sample the rate λ uniformly from a specified range.
- Cause probability: For each causal edge we sample α uniformly from a specified range.

- # Source Events: Number of events sampled for source nodes (variables without any parents in the DAG).
- Additive noise parameter: percentage of additional added events to the caused events, also applies to source nodes, where # Source Events are considered as 'caused'.
- Instant effect: Except for the 'Instant Effect' experiments no instant effects are created.

For all experiments, unless otherwise stated a random DAG is generated. And for each parameterization 20 independent samples are generated.

SANITY CHECK We set the number of types to 20 and generate 100 root events per source node (in this every node is a source node).

INCREASE OF EVENT TYPES In this experiment we increase the number of event types p from 5 to 40, We set the number of edges to $(d^2 - d)/(2 * 5)$, that is 20% of all possible edges. To avoid overly many events in the colliders we set the number of root events to 20, for 40 variables this results in up to $\approx$ 30.000 events. We do not include any additive noise and set $\alpha = 1$. For the delay distribution, we sample λ from a range between of $[0.3, 1]$.

DECREASE OF NOISE In this experiment we increase the probability of α, and decrease the fraction of additive noise. We set the number of variables to 20 and set the number of root events to 100. We sample λ from a range of $[0.1, 0.4]$.

DISTRIBUTION MISSPECIFICATION To further evaluate robustness of CASCADE we test recovery on generated data where the actual distribution does not match the assumed distribution. To this end, we change the assumed distribution of CASCADE and use the same setup as the previous experiment (Increase of Event Types) with 20 unique events. For the, true, exponential we observe an average F1 score of 0.82, for the Poisson 0.81, with a Normal distribution 0.76, and uniform 0.75. While recovery is best when assumed and generating distribution match CASCADE still performs well under misspecification.

Number of Colliders	F1
5	0.97 ±0.01
6	0.96 ±0.01
7	0.95 ±0.02
8	0.93 ±0.01
9	0.92 ±0.02
10	0.91 ±0.01
15	0.88 ±0.02
20	0.82 ±0.01

Table e.1: Average *F*1 score on *Multiple Parents* experiment with multiple colliders.

MULTIPLE PARENTS For this experiment we specify a DAG, where $\lceil \frac{n-1}{2} \rceil$ are direct parents and $\lfloor \frac{n-1}{2} \rfloor$ are independent, the n^{th} node is the collider. We plant 30 events per root cause and increase the number of variables from 50 to 200. We add 30 % of additive noise and set the cause probability randomly between 0.9 and 0.6. We repeat the same experiment where 10% of nodes are colliders. That is, for 50 event types, 5 are colliders and $\lceil \frac{p-5}{2} \rceil$ direct causes of all 5 colliders. The remaining $\lfloor \frac{p-5}{2} \rfloor$ are independent. We show the results in Table e.1.

INSTANT EFFECTS For the instant effects experiments we again use 20 variables with 100 root events, we shift the geometric delay distribution and set $\lambda = 0.9$, such that 90% of the events are generated at the same timestamp. We randomly sample the trigger probability between 0.7 and 0.5. For the exclusively instant effects we set $\lambda = 1$. We show the full results in Table e.2 and

E.2.2 *Method Parameterization*

CASCADE We set the precision parameter for all experiments to 2. For all synthetic experiments we consider events as potential causes of

Method	F1	SHD	SID	SHD-Norm	SID-Norm
CASCADE	0.74	19.45	104.10	0.05	0.27
CAUSE	0.19	264.50	NaN	0.69	NaN
NPHC	0.57	47.50	NaN	0.12	NaN
THP	0.23	64.20	180.70	0.16	0.47

Table E.2: Average results on 90% instant data

Method	F1	SHD	SID	SHD-Norm	SID-Norm
CASCADE	0.55	32.80	216.05	0.08	0.56
CAUSE	0.19	249.60	NaN	0.65	NaN
NPHC	0.57	46.85	NaN	0.12	NaN

Table E.3: Results of instant effects, we omit the results of THP as it only reports empty DAGs

# Event Types	CASCADE	CAUSE	NPHC	THP	MDLH
5	2.05	6.50	4.30	2.35	3.00
10	2.80	9.50	4.25	3.55	16489.00
15	7.10	16.75	4.40	13.80	NaN
20	35.20	42.95	4.15	40.90	NaN
30	362.65	225.70	4.75	305.50	NaN
40	1957.65	890.45	4.85	1984.60	NaN

Table e.4: Mean runtime, in seconds, of Increase Event Types Experiment

at most 100 timestamps. For all experiments we consider a geometric distribution, which we shift back to cover instant effects.

MDLH For the results of the *Increase event types experiment* we use the sparse version, where we set the maximum degree to the true maximal degree and set T =1000. In an effort to reduce runtime with higher number of types (i.e. nodes), we tested it with T=100, where it also did not terminate within 96 hours.

OTHER For all other competing methods we used the default parameters.

E.2.3 *Compute Recourses*

All experiments where executed on a internal cluster on compute nodes equipped with a AMD EPYC 7773X 64-Core Processor (2.2 GHz; Turboboost: 3.5 GHz), with 2 TB of RAM, while in practice a fraction of that was necessary. We provide the average runtimes below.

E.2.4 *Network Alarms*

In the provided dataset each event happens on a specific device. In addition to the event sequences a topology $\mathcal{T}$ over the devices is provided. An event can cause an event on each neighboring device, in addition to the device where the event occurred. To support this we

Noise	Cascade	CAUSE	NPHC	THP
0.10	147.85	231.50	4.55	97.20
0.50	66.85	176.85	4.40	71.45
0.60	42.75	142.30	4.50	61.55
0.70	25.20	113.30	4.40	51.35
0.80	14.05	76.75	4.25	41.65
0.90	7.55	48.60	4.45	32.95

Table E.5: Mean runtime, in seconds, under increasing Noise.

# Event Types	Cascade	CAUSE	NPHC	THP
50	45.80	323.65	6.30	587.45
60	79.60	536.75	6.10	1596.55
70	131.30	904.50	5.90	3175.30
80	196.80	1129.00	6.45	5393.70
90	283.90	1513.60	6.95	8295.25
100	392.55	1941.80	7.20	12923.50
150	1479.60	4694.75	10.05	NaN
200	3736.70	9736.05	16.95	NaN

Table E.6: Mean runtime, in seconds, of Increase Event Types (Collider Experiment)

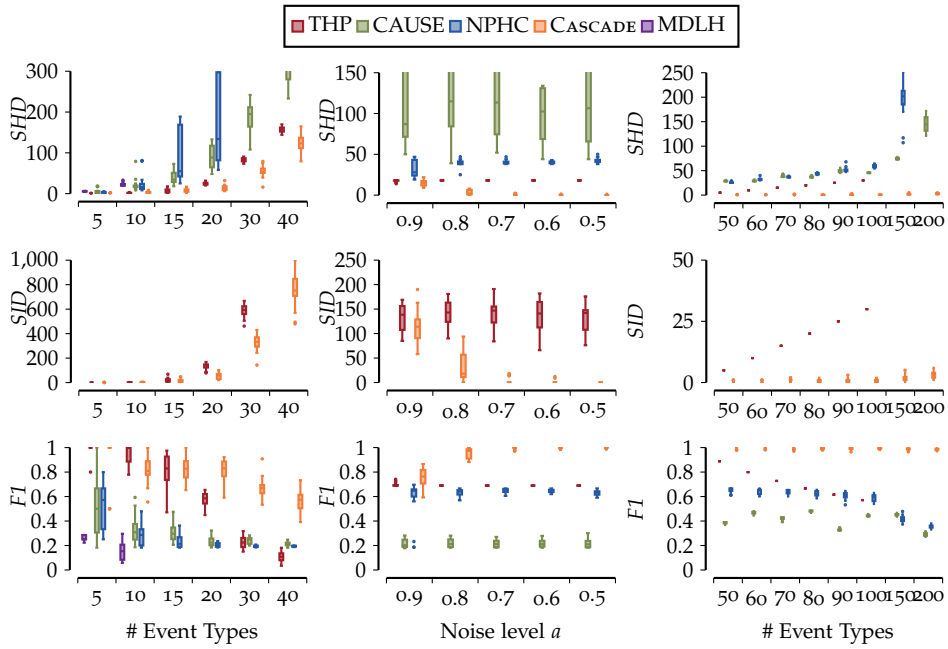
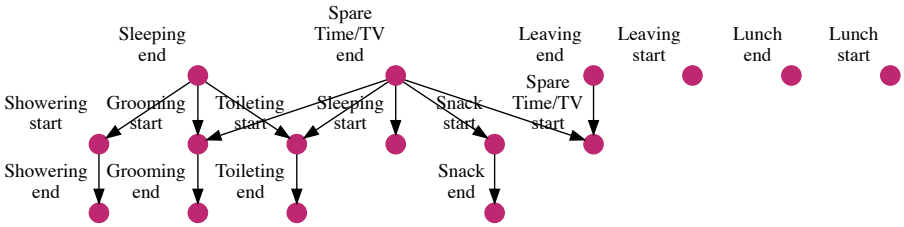


Figure e.2: SHD, SID, and F1 score for the synthetic experiments

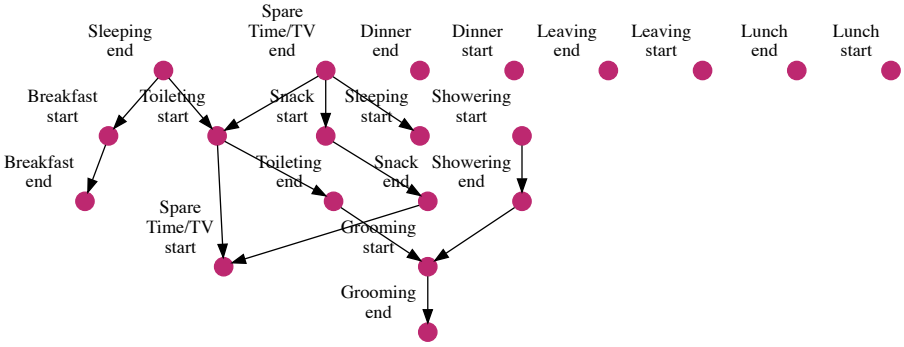
can include a matching for connected devices. That is if $\{a, b\} \in \mathcal{T}$ we include $\Delta_{i\to j}^{(a,b)}$ and $\Delta_{i\to j}^{(b,a)}$, we include both directions since events on a can cause events on b and events on b can cause events on a. For all devices we include the self loop $\Delta_{i\to j}^{(a,a)}$.

E.2.5 *Real World Experiment*

In this section we provide the Causal Graphs reported by CASCADE on the real world data. For the *Global Banks* dataset additionally provide the graph reported by THP.



(a)



(b)

Figure e.3: Recovered Causal Graphs on the two *Daily Activities* datasets.

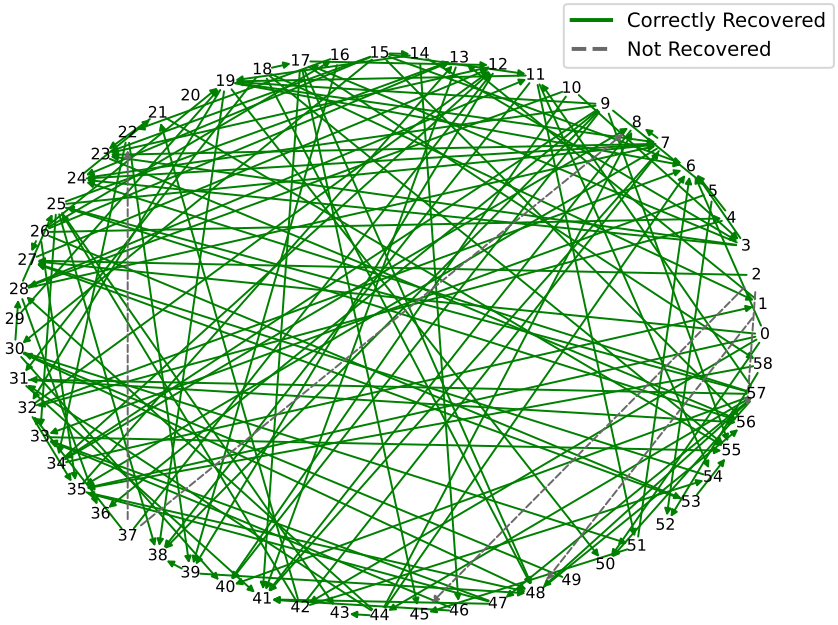


Figure e.4: Recovered Causal Graph on Network Alarms dataset.

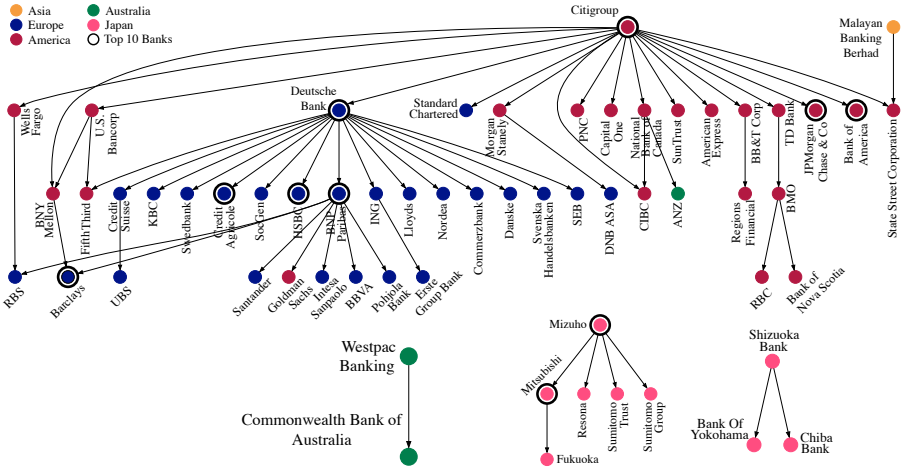


Figure e.5: DAG reported by CASCADE on the *Global Banks* dataset [43]. We omit unconnected nodes for clarity.

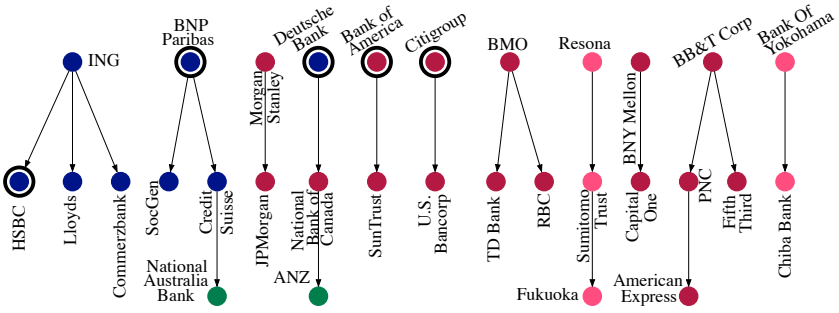


Figure E.6: DAG reported by THP on the *Global Banks* dataset [43]. We omit unconnected nodes for clarity.

BIBLIOGRAPHY

[1] Sebastian Abt and Harald Baier. "A plea for utilising synthetic data when performing machine learning based cyber-security experiments." In: *Proceedings of the 2014 workshop on artificial intelligent and security workshop*. 2014, pp. 37–45.

[2] Massil Achab, Emmanuel Bacry, Stéphane Gaïffas, Iacopo Mastromatteo, and Jean-François Muzy. "Uncovering causality from multivariate Hawkes integrated cumulants." In: *Journal of Machine Learning Research* 18.192 (2018), pp. 1–28.

[3] Tejumade Afonja, Dingfan Chen, and Mario Fritz. "MargCTGAN: A "Marginally" Better CTGAN for the Low Sample Regime." In: *DAGM German Conference on Pattern Recognition*. Springer. 2023, pp. 524–537.

[4] Rakesh Agrawal and Ramakrishnan Srikant. "Mining sequential patterns." In: *Proceedings of the 11th International Conference on Data Engineering (ICDE), Taipei, Taiwan*. Los Alamitos, CA, USA: IEEE Computer Society, 1995, pp. 3–14.

[5] Tertsegha J Anande and Mark S Leeson. "Generative adversarial networks (gans): a survey of network traffic generation." In: *International Journal of Machine Learning and Computing* 12.6 (2022), pp. 333–343.

[6] Tertsegha J Anande, Sami Al-Saadi, and Mark S Leeson. "Generative adversarial networks for network traffic feature generation." In: *International Journal of Computers and Applications* 45.4 (2023), pp. 297–305.

[7] M. S. Bartlett. "On the Theoretical Specification and Sampling Properties of Autocorrelated Time-Series." In: *Supplement to the Journal of the Royal Statistical Society* 8.1 (1946), pp. 27–41.

[8] Iyad Batal, Gregory F Cooper, Dmitriy Fradkin, James Harrison, Fabian Moerchen, and Milos Hauskrecht. "An efficient pattern mining approach for event detection in multivariate temporal data." In: *Knowledge and Information Systems* 46.1 (2016), pp. 115–150.

[9] Kaustubh Beedkar and Rainer Gemulla. "LASH: Large-Scale Sequence Mining with Hierarchies." In: *PODS*. May 2015, pp. 491–503.

[10] Roel Bertens, Jilles Vreeken, and Arno Siebes. "Keeping It Short and Simple: Summarising Complex Event Sequences with Multivariate Patterns." In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. San Francisco California USA: ACM, Aug. 2016, pp. 735–744.

[11] Apratim Bhattacharyya and Jilles Vreeken. "Efficiently summarising event sequences with rich interleaving patterns." In: *Proceedings of the SIAM International Conference on Data Mining, Houston, TX*. SIAM. 2017, pp. 795–803.

[12] Peter Bloem and Steven de Rooij. "Large-scale network motif analysis using compression." In: *Data Mining and Knowledge Discovery* 34 (2020), pp. 1421–1453.

[13] Stavroula Bourou, Andreas El Saer, Terpsichori-Helen Velivassaki, Artemis Voulkidis, and Theodore Zahariadis. "A review of tabular data synthesis using GANs on an IDS dataset." In: *Information* 12.09 (2021), p. 375.

[14] Erwan Bourrand, Luis Galárraga, Esther Galbrun, Elisa Fromont, and Alexandre Termier. "Discovering useful compact sets of sequential rules in a long sequence." In: *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2021, pp. 1295–1299.

[15] Daniela Brauckhoff, Xenofontas Dimitropoulos, Arno Wagner, and Kavé Salamatian. "Anomaly Extraction in Backbone Networks Using Association Rules." In: *IEEE/ACM Transactions on Networking* 20.6 (Dec. 2012), pp. 1788–1799.

[16] Kailash Budhathoki and Jilles Vreeken. "Accurate Causal Inference on Discrete Data." In: *Proceedings of the IEEE International Conference on Data Mining*. IEEE, 2018.

[17] Kailash Budhathoki and Jilles Vreeken. "Causal inference on event sequences." In: *Proceedings of the 2018 SIAM International Conference on Data Mining*. SIAM. 2018, pp. 55–63.

[18] Peter Bühlmann, Jonas Peters, and Jan Ernest. "CAM: Causal additive models, high-dimensional order search and penalized regression." In: *The Annals of Statistics* 42.6 (2014), pp. 2526–2556.

[19] Ruichu Cai, Siyu Wu, Jie Qiao, Zhifeng Hao, Keli Zhang, and Xi Zhang. "THPs: Topological Hawkes processes for learning causal structure on event sequences." In: *IEEE Transactions on Neural Networks and Learning Systems* (2022).

[20] Lucien Le Cam. "An approximation theorem for the Poisson binomial distribution." In: *Pacific Journal of Mathematics* 10.4 (1960), pp. 1181–1197.

[21] Xinhong Chen, Wensheng Gan, Shicheng Wan, and Tianlong Gu. "MDL-based Compressing Sequential Rules." In: *arXiv preprint arXiv:2212.10252* (2022).

[22] Yangming Chen, Philippe Fournier-Viger, Farid Nouioua, and Youxi Wu. "Mining partially-ordered episode rules with the head support." In: *Big Data Analytics and Knowledge Discovery: 23rd International Conference, DaWaK 2021, Virtual Event, September 27–30, 2021, Proceedings 23*. Springer. 2021, pp. 266–271.

[23] Yonghong Chen, Govindan Rangarajan, Jianfeng Feng, and Mingzhou Ding. "Analyzing multiple nonlinear time series with extended Granger causality." In: *Physics letters A* 324.1 (2004), pp. 26–35.

[24] A. Cheng. "PAC-GAN: Packet Generation of Network Traffic using Generative Adversarial Networks." In: *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference*. 2019, pp. 0728–0734.

[25] David Maxwell Chickering. "Optimal structure identification with greedy search." In: *Journal of machine learning research* 3.Nov (2002), pp. 507–554.

[26] Spencer Compton, Kristjan Greenewald, Dmitriy A Katz, and Murat Kocaoglu. "Entropic causal inference: Graph identifiability." In: *International Conference on Machine Learning*. PMLR. 2022, pp. 4311–4343.

[27] Charles Corbière, Nicolas Thome, Avner Bar-Hen, Matthieu Cord, and Patrick Pérez. "Addressing failure prediction by learning model confidence." In: *Advances in neural information processing systems* 32 (2019).

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

[29] Damien Cram, Benoît Mathern, and Alain Mille. "A Complete Chronicle Discovery Approach: Application to Activity Analysis." In: *Expert Systems* 29.4 (2012), pp. 321–346.

[30] Boris Cule and Bart Goethals. "Mining association rules in long sequences." In: *Advances in Knowledge Discovery and Data Mining: 14th Pacific-Asia Conference, PAKDD 2010, Hyderabad, India, June 21-24, 2010. Proceedings. Part I 14*. Springer. 2010, pp. 300–309.

[31] Joscha Cüppers, Janis Kalofolias, and Jilles Vreeken. "Omen: discovering sequential patterns with reliable prediction delays." In: *Knowledge and Information Systems* 64.4 (2022), pp. 1013–1045.

[32] Joscha Cüppers, Paul Krieger, and Jilles Vreeken. "Discovering Sequential Patterns with Predictable Inter-event Delays." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. 8. AAAI, 2024, pp. 8346–8353.

[33] Joscha Cüppers, Adrien Schoen, Gregory Blanc, and Pierre-Francois Gimenez. "FlowChronicle: Synthetic Network Flow Generation through Pattern Set Mining." In: *Proceedings of the ACM on Networking*. Vol. 2. CoNEXT4. ACM, 2024, p. 26.

[34] Joscha Cüppers and Jilles Vreeken. "Just Wait for it... Mining Sequential Patterns with Reliable Prediction Delays." In: *2020 IEEE International Conference on Data Mining*. IEEE, 2020, pp. 82–91.

[35] Joscha Cüppers and Jilles Vreeken. "Below the Surface: Summarizing Event Sequences with Generalized Sequential Patterns." In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2023, pp. 348–357.

[36] Joscha Cüppers, Sascha Xu, Musa Ahmed, and Jilles Vreeken. "Causal Discovery from Event Sequences by Local Cause-Effect Attribution." In: *Advances in Neural Information Processing Systems*. Vol. 37. 2024, pp. 24216–24241.

[37] Sebastian Dalleiger and Jilles Vreeken. "Explainable data decompositions." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04. 2020, pp. 3709–3716.

[38] Sebastian Dalleiger and Jilles Vreeken. "Efficiently factorizing boolean matrices using proximal gradient descent." In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 4736–4748.

[39] Benjamin Dalmas, Philippe Fournier-Viger, and Sylvie Norre. "TWINCLE: A constrained sequential rule mining algorithm for event logs." In: *Procedia computer science* 112 (2017), pp. 205–214.

[40] Fida K Dankar, Mahmoud K Ibrahim, and Leila Ismail. "A multi-dimensional evaluation of synthetic data generators." In: *IEEE Access* 10 (2022), pp. 11147–11158.

[41] Yann Dauxais, Thomas Guyet, David Gross-Amblard, and André Happe. "Discriminant chronicles mining: Application to care pathways analytics." In: *Artificial Intelligence in Medicine: 16th Conference on Artificial Intelligence in Medicine, AIME 2017, Vienna, Austria, June 21-24, 2017, Proceedings 16*. Springer. 2017, pp. 234–244.

[42] Tijl De Bie. "Subjective interestingness in exploratory data mining." In: *International Symposium on Intelligent Data Analysis*. Springer. 2013, pp. 19–31.

[43] Mert Demirer, Francis X Diebold, Laura Liu, and Kamil Yilmaz. "Estimating global bank network connectedness." In: *Journal of Applied Econometrics* 33.1 (2018), pp. 1–15.

[44] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "Bert: Pre-training of deep bidirectional transformers for language understanding." In: *arXiv preprint arXiv:1810.04805* (2018).

[45] Vanessa Didelez. "Graphical models for marked point processes based on local independence." In: *Journal of the Royal Statistical Society Series B: Statistical Methodology* 70.1 (2008), pp. 245–264.

[46] Baik Dowoo, Yujin Jung, and Changhee Choi. "PcapGAN: Packet capture file generator by style-based generative adversarial networks." In: *2019 18th IEEE International Conference On Machine Learning And Applications*. IEEE. 2019, pp. 1149–1154.

[47] Jack Edmonds and Richard M Karp. "Theoretical improvements in algorithmic efficiency for network flow problems." In: *Journal of the ACM* 19.2 (1972), pp. 248–264.

[48] Elias Egho, Dominique Gay, Marc Boullé, Nicolas Voisine, and Fabrice Clérot. "A user parameter-free approach for mining robust sequential classification rules." In: *Knowledge and Information Systems* 52.1 (2017), pp. 53–81.

[49] Michael Eichler, Rainer Dahlhaus, and Johannes Dueck. "Graphical modeling for multivariate Hawkes processes with nonparametric link functions." In: *Journal of Time Series Analysis* 38.2 (2017), pp. 225–242.

[50] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise." In: *Proceedings of the 2nd ACM International Conference on Knowledge Discovery and Data Mining, Portland, OR*. Vol. 96. 1996, pp. 226–231.

[51] Jonas Fischer and Jilles Vreeken. "Differentiable pattern set mining." In: *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*. 2021, pp. 383–392.

[52] Philippe Fournier-Viger, Yangming Chen, Farid Nouioua, and Jerry Chun-Wei Lin. "Mining partially-ordered episode rules in an event sequence." In: *Intelligent Information and Database Systems: 13th Asian Conference, ACIIDS 2021, Phuket, Thailand, April 7–10, 2021, Proceedings 13*. Springer. 2021, pp. 3–15.

[53] Philippe Fournier-Viger, Usef Faghihi, Roger Nkambou, and Engelbert Mephu Nguifo. "CMRules: Mining sequential rules common to several sequences." In: *Knowledge-Based Systems* 25.1 (2012), pp. 63–76.

[54] Philippe Fournier-Viger, Ted Gueniche, Souleymane Zida, and Vincent S Tseng. "ERMiner: sequential rule mining using equivalence classes." In: *Advances in Intelligent Data Analysis XIII: 13th International Symposium, IDA 2014, Leuven, Belgium, October 30–November 1, 2014. Proceedings 13*. Springer. 2014, pp. 108–119.

[55] Philippe Fournier-Viger and Vincent S Tseng. "TNS: mining top-k non-redundant sequential rules." In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. 2013, pp. 164–166.

[56] Philippe Fournier-Viger, Cheng-Wei Wu, Vincent S. Tseng, Longbing Cao, and Roger Nkambou. "Mining Partially-Ordered Sequential Rules Common to Multiple Sequences." In: *IEEE Transactions on Knowledge and Data Engineering* 27.8 (2015), pp. 2203–2216.

[57] Jaroslav Fowkes and Charles Sutton. "A subsequence interleaving model for sequential pattern mining." In: *Proceedings of the 22nd ACM International Conference on Knowledge Discovery and Data Mining, San Francisco, CA*. 2016, pp. 835–844.

[58] Esther Galbrun, Peggy Cellier, Nikolaj Tatti, Alexandre Termier, and Bruno Crémilleux. "Mining periodic patterns with a MDL criterion." In: *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases, Dublin, Ireland*. Springer. 2018, pp. 535–551.

[59] Jun Gao, Di He, Xu Tan, Tao Qin, Liwei Wang, and Tie-Yan Liu. "Representation Degeneration Problem in Training Natural Language Generation Models." In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[60] Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. "Mining sequential patterns with regular expression constraints." In: *IEEE Trans Knowl Data Eng* 14.3 (2002), pp. 530–552.

[61] Clément Gautrais, Peggy Cellier, Matthijs van Leeuwen, and Alexandre Termier. "Widening for MDL-based retail signature discovery." In: *Advances in Intelligent Data Analysis XVIII: 18th International Symposium on Intelligent Data Analysis, IDA 2020, Konstanz, Germany, April 27–29, 2020, Proceedings 18*. Springer. 2020, pp. 197–209.

[62] Matthew S Gerber. "Predicting crime using Twitter and kernel density estimation." In: *Decision Support Systems* 61 (2014), pp. 115–125.

[63] Fosca Giannotti, Mirco Nanni, Dino Pedreschi, and Fabio Pinelli. "Mining Sequences with Temporal Annotations." In: *Proceedings of the 2006 ACM Symposium on Applied Computing*. Dijon France: ACM, Apr. 2006, pp. 593–597.

[64] Aceto Giuseppe, Fabio Giampaolo, Ciro Guida, Stefano Izzo, Antonio Pescape, Francesco Piccialli, and Edoardo Prezioso. "Synthetic and Privacy-Preserving Traffic Trace Generation using Generative AI Models for Training Network Intrusion Detection Systems." In: *Available at SSRN 4643250* (2023).

[65] Eduard Glatz, Stelios Mavromatidis, Bernhard Ager, and Xenofontas Dimitropoulos. "Visualizing Big Network Traffic Data Using Frequent Pattern Mining and Hypergraphs." In: *Computing* 96.1 (Jan. 2014), pp. 27–38.

[66] Bart Goethals, Sandy Moens, and Jilles Vreeken. "MIME: A Framework for Interactive Visual Pattern Mining." In: *Proceedings of the 17th ACM International Conference on Knowledge Discovery and Data Mining, San Diego, CA*. ACM, 2011, pp. 757–760.

[67] Korosh Golnabi, Richard K Min, Latifur Khan, and Ehab Al-Shaer. "Analysis of firewall policy rules using data mining techniques." In: *2006 IEEE/IFIP Network Operations and Management Symposium NOMS 2006*. IEEE. 2006, pp. 305–315.

[68] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets." In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[69] Clive WJ Granger. "Investigating causal relations by econometric models and cross-spectral methods." In: *Econometrica* (1969), pp. 424–438.

[70] Kathrin Grosse and Jilles Vreeken. "Summarising event sequences using serial episodes and an ontology." In: *Proceedings of the Workshop on Interactions between Data Mining and Natural Language Processing@ ECML/PKDD*. Vol. 17. 2017.

[71] Peter Grünwald. *The Minimum Description Length Principle*. MIT Press, 2007.

[72] Jorge Luis Guerra, Carlos Catania, and Eduardo Veas. "Datasets are not enough: Challenges in labeling network traffic." In: *Computers & Security* 120 (2022), p. 102810.

[73] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. "Improved training of wasserstein gans." In: *Advances in neural information processing systems* 30 (2017).

[74] Christian W Günther and Wil MP Van Der Aalst. "Fuzzy mining–adaptive process simplification based on multi-perspective metrics." In: *International conference on business process management*. Springer. 2007, pp. 328–343.

[75] Arash Habibi Lashkari., Gerard Draper Gil., Mohammad Saiful Islam Mamun., and Ali A. Ghorbani. "Characterization of Tor Traffic using Time based Features." In: *Proceedings of the 3rd International Conference on Information Systems Security and Privacy - ICISSP*. INSTICC. SciTePress, 2017, pp. 253–262.

[76] L. Han, Y. Sheng, and X. Zeng. "A Packet-Length-Adjustable Attention Model Based on Bytes Embedding Using Flow-WGAN for Smart Cybersecurity." In: *IEEE Access* 7 (2019), pp. 82913–82926.

[77] Dominique MA Haughton. "On the choice of a model to fit data from an exponential family." In: *The annals of statistics* (1988), pp. 342–355.

[78] Alan G Hawkes. "Spectra of some self-exciting and mutually exciting point processes." In: *Biometrika* 58.1 (1971), pp. 83–90.

[79] Yili Hong. "On computing the distribution function for the Poisson binomial distribution." In: *Computational Statistics & Data Analysis* 59 (2013), pp. 41–51.

[80] Bryan Hooi and Christos Faloutsos. "Branch and border: Partition-based change detection in multivariate time series." In: *Proceedings of the 2019 SIAM International Conference on Data Mining*. SIAM. 2019, pp. 504–512.

[81] Amin Hosseininasab, Willem-Jan van Hoeve, and Andre A Cire. "Constraint-based sequential pattern mining with decision diagrams." In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 33. 01. 2019, pp. 1495–1502.

[82] S. Hui, H. Wang, Z. Wang, X. Yang, Z. Liu, D. Jin, and Y. Li. "Knowledge Enhanced GAN for IoT Traffic Generation." In: *Proceedings of the ACM Web Conference 2022*. WWW '22. Virtual Event, Lyon, France: Association for Computing Machinery, 2022, pp. 3336–3346.

[83] A R Jakhale and G A Patil. "Anomaly Detection System by Mining Frequent Pattern Using Data Mining Algorithm from Network Flow." In: *International Journal of Engineering Research* 3.1 (2014).

[84] Amirkasra Jalaldoust, Kateřina Hlaváčková-Schindler, and Claudia Plant. "Causal discovery in Hawkes processes by minimum description length." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 6. 2022, pp. 6978–6987.

[85] Dominik Janzing and Bernhard Schölkopf. "Causal inference using the algorithmic Markov condition." In: *IEEE Transactions on Information Theory* 56.10 (2010), pp. 5168–5194.

[86] Steedman Jenkins, Stefan Walzer-Goldfeld, and Matteo Riondato. "SPEck: Mining Statistically-Significant Sequential Patterns Efficiently with Exact Sampling." In: *Data Min Knowl Disc* 36.4 (July 2022), pp. 1575–1599.

[87] Markus Kalisch and Peter Bühlman. "Estimating high-dimensional directed acyclic graphs with the PC-algorithm." In: *Journal of Machine Learning Research* 8.3 (2007).

[88] David Kaltenpoth and Jilles Vreeken. "We are not your real parents: Telling causal from confounded using MDL." In: *Proceedings of the 2019 SIAM International Conference on Data Mining*. SIAM. 2019, pp. 199–207.

[89] David Kaltenpoth and Jilles Vreeken. "Identifying Selection Bias from Observational Data." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI, 2023.

[90] David Kaltenpoth and Jilles Vreeken. "Nonlinear Causal Discovery with Latent Confounders." In: *Proceedings of the International Conference on Machine Learning*. PMLR, 2023.

[91] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 2010.

[92] Markelle Kelly, Rachel Longjohn, and Kolby Nottingham. *UCI Machine Learning Repository*.

[93] A. Kenyon, L. Deka, and D. Elizondo. "Are public intrusion datasets fit for purpose characterising the state of the art in intrusion event datasets." In: *Computers & Security* 99 (2020), p. 102022.

[94] Diederik P Kingma and Max Welling. "Auto-encoding variational bayes." In: *arXiv preprint arXiv:1312.6114* (2013).

[95] Murat Kocaoglu, Alexandros Dimakis, Sriram Vishwanath, and Babak Hassibi. "Entropic causal inference." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. 1. 2017.

[96] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[97] Maxime Lanvin, Pierre-François Gimenez, Yufei Han, Frédéric Majorczyk, Ludovic Mé, and Eric Totel. "Errors in the CICIDS2017 dataset and the significant differences in detection performances it makes." In: *International Conference on Risks and Security of Internet and Systems*. Springer. 2022, pp. 18–33.

[98] Michael Larsen and Fernando Gont. *Recommendations for transport-protocol port randomization*. Tech. rep. 2011.

[99] Srivatsan Laxman, PS Sastry, and KP Unnikrishnan. "A fast algorithm for finding frequent episodes in event streams." In: *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2007, pp. 410–419.

[100] Srivatsan Laxman, Vikram Tankasali, and Ryen W White. "Stream prediction using a generative model based on frequent episodes in event sequences." In: *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2008, pp. 453–461.

[101] Matthijs van Leeuwen and Lara Cardinaels. "VIPER–visual pattern explorer." In: *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part III 15*. Springer. 2015, pp. 333–336.

[102] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. "Retrieval-augmented generation for knowledge-intensive nlp tasks." In: *Advances in neural information processing systems* 33 (2020), pp. 9459–9474.

[103] Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications*. Vol. 3. Springer, 2008.

[104] Xin Li and Zhi-Hong Deng. "Mining Frequent Patterns from Network Flows for Monitoring Network." In: *Expert Systems with Applications* 37.12 (Dec. 2010), pp. 8850–8860.

[105] Jessica Lin, Eamonn Keogh, Li Wei, and Stefano Lonardi. "Experiencing SAX: a novel symbolic representation of time series." In: *Data Mining and Knowledge Discovery* 15.2 (2007), pp. 107–144.

[106] Z. Lin, A. Jain, C. Wang, G. Fanti, and V. Sekar. "Using GANs for Sharing Networked Time Series Data: Challenges, Initial Promise, and Open Questions." In: *Proceedings of the ACM Internet Measurement Conference*. IMC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 464–483.

[107] Zilong Lin, Yong Shi, and Zhi Xue. "Idsgan: Generative adversarial networks for attack generation against intrusion detection." In: *Pacific-asia conference on knowledge discovery and data mining*. Springer. 2022, pp. 79–91.

[108] Cecile Low-Kam, Chedy Raissi, Mehdi Kaytoue, and Jian Pei. "Mining Statistically Significant Sequential Patterns." In: *ICDM*. Dallas, TX, USA: IEEE, Dec. 2013, pp. 488–497.

[109] Ralf Korn Magnus Wiese Robert Knobloch and Peter Kretschmer. "Quant GANs: deep generation of financial time series." In: *Quantitative Finance* 20.9 (2020), pp. 1419–1440.

[110] Sarah Mameche, David Kaltenpoth, and Jilles Vreeken. "Learning Causal Models under Independent Changes." In: *Proceedings of Neural Information Processing Systems*. PMLR, 2023.

[111] Heikki Mannila and Chris Meek. "Global Partial Orders From Sequential Data." In: *Proceedings of the 6th ACM International Conference on Knowledge Discovery and Data Mining, Boston, MA*. 2000, pp. 161–168.

[112] Heikki Mannila, Hannu Toivonen, and A Inkeri Verkamo. "Discovery of frequent episodes in event sequences." In: *Data mining and knowledge discovery* 1 (1997), pp. 259–289.

[113] L. D. Manocchio, S. Layeghy, and M. Portmann. "Flowgan-synthetic network flow generation using generative adversarial networks." In: *2021 IEEE 24th International Conference on Computational Science and Engineering*. IEEE. 2021, pp. 168–176.

[114] Alexander Marx and Jilles Vreeken. "Telling cause from effect by local and global regression." In: *Knowledge and Information Systems* 60 (2019), pp. 1277–1305.

[115] Alexander Marx and Jilles Vreeken. "Formally Justifying MDL-based Inference of Cause and Effect." In: *AAAI Workshop on Information-Theoretic Causal Inference and Discovery (ITCI'22)*. 2022.

[116] A. Meddahi, H. Drira, and A. Meddahi. "SIP-GAN: Generative Adversarial Networks for SIP traffic generation." In: *2021 International Symposium on Networks, Computers and Communications*. 2021, pp. 1–6.

[117] Marvin Meeng and Arno Knobbe. "Flexible enrichment with cortana–software demo." In: *Proceedings of BeneLearn*. 2011, pp. 117–119.

[118] Hongyuan Mei and Jason M Eisner. "The neural hawkes process: A neurally self-modulating multivariate point process." In: *Advances in neural information processing systems* 30 (2017).

[119] Fabien Meslet-Millet, Sandrine Mouysset, and Emmanuel Chaput. "NeCSTGen: An approach for realistic network traffic generation using Deep Learning." In: *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*. 2022, pp. 3108–3113.

[120] Osman A Mian, Alexander Marx, and Jilles Vreeken. "Discovering fully oriented causal networks." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 10. 2021, pp. 8975–8982.

[121] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient estimation of word representations in vector space." In: *arXiv preprint arXiv:1301.3781* (2013).

[122] George A Miller. "WordNet: A Lexical Database for English." In: *Communications of the ACM* 38.11 (1995), pp. 39–41.

[123] Fabian Moerchen and Dmitriy Fradkin. "Robust Mining of Time Intervals with Semi-Interval Partial Order Patterns." In: *SDM*. 2010, pp. 315–326.

[124] M. F. Naeem, S. J. Oh, Y. Uh, Y. Choi, and J. Yoo. "Reliable fidelity and diversity metrics for generative models." In: *International Conference on Machine Learning*. PMLR. 2020, pp. 7176–7185.

[125] S. Nakayama and D. Watling. "Consistent formulation of network equilibrium with stochastic flows." In: *Transportation Research Part B-methodological* 66 (2014), pp. 50–69.

[126] Mirco Nanni and Christophe Rigotti. "Extracting Trees of Quantitative Serial Episodes." In: *Knowledge Discovery in Inductive Databases*. Ed. by Sašo Džeroski and Jan Struyf. Vol. 4747. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 170–188.

[127] Gonzalo Navarro. "A guided tour to approximate string matching." In: *ACM computing surveys* 33.1 (2001), pp. 31–88.

[128] Muhammad Haris Naveed, Umair Sajid Hashmi, Nayab Tajved, Neha Sultan, and Ali Imran. "Assessing deep generative models on time series network data." In: *IEEE Access* 10 (2022), pp. 64601–64617.

[129] Hojjat Navidan, Parisa Fard Moshiri, Mohammad Nabati, Reza Shahbazian, Seyed Ali Ghorashi, Vahid Shah-Mansouri, and David Windridge. "Generative Adversarial Networks (GANs) in networking: A comprehensive survey & evaluation." In: *Computer Networks* 194 (May 2021), p. 108149.

[130] Euclides Carlos Pinto Neto, Sajjad Dadkhah, Raphael Ferreira, Alireza Zohourian, Rongxing Lu, and Ali A. Ghorbani. "CICIoT2023: A Real-Time Dataset and Benchmark for Large-Scale Attacks in IoT Environment." In: *Sensors* 23.13 (2023).

[131] Hao Ni, Lukasz Szpruch, Marc Sabate-Vidales, Baoren Xiao, Magnus Wiese, and Shujian Liao. "Sig-Wasserstein GANs for time series generation." In: *Proceedings of the Second ACM International Conference on AI in Finance*. 2021, pp. 1–8.

[132] Siegfried Nijssen and Albrecht Zimmermann. "Constraint-based pattern mining." In: *Frequent pattern mining* (2014), pp. 147–163.

[133] Fco Javier Ordónez, Paula De Toledo, and Araceli Sanchis. "Activity recognition using hybrid generative/discriminative models on home environments using binary sensors." In: *Sensors* 13.5 (2013), pp. 5460–5477.

[134] Ignasi Paredes-Oliva, Pere Barlet-Ros, and Xenofontas Dimitropoulos. "FaRNet: Fast Recognition of High-Dimensional Patterns from Big Network Traffic Data." In: *Computer Networks* 57.18 (Dec. 2013), pp. 3897–3913.

[135] Ignasi Paredes-Oliva, Ismael Castell-Uroz, Pere Barlet-Ros, Xenofontas Dimitropoulos, and Josep Sole-Pareta. "Practical Anomaly Detection Based on Classifying Frequent Traffic Patterns." In: *2012 Proceedings IEEE INFOCOM Workshops*. Orlando, FL, USA: IEEE, Mar. 2012, pp. 49–54.

[136] Noseong Park, Mahmoud Mohammadi, Kshitij Gorde, Sushil Jajodia, Hongkyu Park, and Youngmin Kim. "Data synthesis based on generative adversarial networks." In: *Proc. VLDB Endow.* 11.10 (2018), pp. 1071–1083.

[137] Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. "The Synthetic data vault." In: *IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. 2016, pp. 399–410.

[138] Judea Pearl. *Causality*. Cambridge university press, 2009.

[139] Jian Pei, Jiawei Han, and Wei Wang. "Mining sequential patterns with constraints in large databases." In: *CIKM*. 2002, pp. 18–25.

[140] Jian Pei, Jiawei Han, and Wei Wang. "Constraint-based sequential pattern mining: the pattern-growth methods." In: *JIIS* 28.2 (2007), pp. 133–160.

[141] Jonas Peters and Peter Bühlmann. "Structural intervention distance for evaluating causal graphs." In: *Neural computation* 27.3 (2015), pp. 771–799.

[142] Jonas Peters, Dominik Janzing, and Bernhard Schlkopf. *Elements of Causal Inference: Foundations and Learning Algorithms*. The MIT Press, 2017.

[143] Jonas Peters, Dominik Janzing, and Bernhard Scholkopf. "Causal inference on discrete data using additive noise models." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.12 (2011), pp. 2436–2450.

[144] François Petitjean, Tao Li, Nikolaj Tatti, and Geoffrey I. Webb. "Skopus: Mining Top-k Sequential Patterns under Leverage." In: *DAMI* 30.5 (Sept. 2016), pp. 1086–1111.

[145] Yao Jean Marc Pokou, Philippe Fournier-Viger, and Chadia Moghrabi. "Authorship attribution using small sets of frequent part-of-speech skip-grams." In: *The Twenty-Ninth International Flairs Conference*. 2016.

[146] Kai Puolamäki, Emilia Oikarinen, Bo Kang, Jefrey Lijffijt, and Tijl De Bie. "Interactive visual data exploration with subjective feedback: an information-theoretic approach." In: *Data Mining and Knowledge Discovery* 34.1 (2020), pp. 21–49.

[147] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. "Language Models are Unsupervised Multitask Learners." In: (2019).

[148] Kira Radinsky and Eric Horvitz. "Mining the web to predict future events." In: *WSDM*. 2013, pp. 255–264.

[149] Joseph Ramsey, Madelyn Glymour, Ruben Sanchez-Romero, and Clark Glymour. "A million variables and more: the fast greedy equivalence search algorithm for learning high-dimensional graphical causal models, with an application to functional magnetic resonance images." In: *International journal of data science and analytics* 3 (2017), pp. 121–129.

[150] Markus Ring, Daniel Schlör, Dieter Landes, and Andreas Hotho. "Flow-based network traffic generation using generative adversarial networks." In: *Computers & Security* 82 (2019), pp. 156–172.

[151] Markus Ring, Sarah Wunderlich, Dominik Grüdl, Dieter Landes, and Andreas Hotho. "Flow-based benchmark data sets for intrusion detection." In: *Proceedings of the 16th European Conference on Cyber Warfare and Security (ECCWS)*. ACPI, 2017, pp. 361–369.

[152] Jorma Rissanen. "Modeling by shortest data description." In: *Automatica* 14.5 (1978), pp. 465–471.

[153] Jorma Rissanen. "A Universal Prior for Integers and Estimation by Minimum Description Length." In: *The Annals of Statistics* 11.2 (1983), pp. 416–431.

[154] Paul Rolland, Volkan Cevher, Matthäus Kleindessner, Chris Russell, Dominik Janzing, Bernhard Schölkopf, and Francesco Locatello. "Score matching enables causal discovery of nonlinear additive noise models." In: *International Conference on Machine Learning*. PMLR. 2022, pp. 18741–18753.

[155] Amal Saadallah, Matthias Jakobs, and Katharina Morik. "Explainable online deep neural network selection using adaptive saliency maps for time series forecasting." In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2021, pp. 404–420.

[156] Erik Scharwächter and Emmanuel Müller. "Two-Sample Testing for Event Impacts in Time Series." In: *Proceedings of the SIAM International Conference on Data Mining*. SIAM. 2020, pp. 10–18.

[157] Adrien Schoen, Gregory Blanc, Pierre-François Gimenez, Yufei Han, Frédéric Majorczyk, and Ludovic Me. "A Tale of Two Methods: Unveiling the limitations of GAN and the Rise of Bayesian Networks for Synthetic Network Traffic Generation." In: *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE. 2024, pp. 273–286.

[158] Adrien Schoen, Gregory Blanc, Pierre-François Gimenez, Yufei Han, Frédéric Majorczyk, and Ludovic Mé. "Towards generic quality assessment of synthetic traffic for evaluating intrusion detection systems." In: *RESSI 2022-Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information*. 2022, pp. 1–3.

[159] Thomas Schreiber. "Measuring information transfer." In: *Phys. Rev. Lett.* 85.2 (2000), p. 461.

[160] M. R. Shahid, G. Blanc, H. Jmila, Z. Zhang, and H. Debar. "Generative Deep Learning for Internet of Things Network Traffic Generation." In: *2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC)*. 2020, pp. 70–79.

[161] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization." In: *International Conference on Information Systems Security and Privacy*. 2018.

[162] Aleena Siji, Joscha Cüppers, Osman Ali Mian, and Jilles Vreeken. "Seqret: Mining Rule Sets from Event Sequences." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI, 2026.

[163] Ramakrishnan Srikant and Rakesh Agrawal. "Mining Sequential Patterns: Generalizations and Performance Improvements." In: *International Conference on Extending Database Technology*. Springer. 1996, pp. 1–17.

[164] Ramakrishnan Srikant and Rakesh Agrawal. "Mining Generalized Association Rules." In: *Future Generation Computer Systems* 13.2-3 (Nov. 1997), pp. 161–180.

[165] Michael Stenger, Robert Leppich, Ian Foster, Samuel Kounev, and André Bauer. "Evaluation is key: a survey on evaluation measures for synthetic time series." In: *Journal of Big Data* 11.1 (2024), p. 66.

[166] Yaguang Sun and Bernhard Bauer. "A Graph and Trace Clustering-based Approach for Abstracting Mined Business Process Models:" in: *ICEIS*. Rome, Italy, 2016.

[167] *Synthetic Data Metrics*. Version 0.12.0. DataCebo, Inc. Oct. 2023.

[168] Nikolaj Tatti. "Significance of Episodes Based on Minimal Windows." In: *Proceedings of the 9th IEEE International Conference on Data Mining, Miami, FL*. 2009, pp. 513–522.

[169] Nikolaj Tatti. "Discovering episodes with compact minimal windows." In: *Data Mining and Knowledge Discovery* 28.4 (2014), pp. 1046–1077.

[170] Nikolaj Tatti and Jilles Vreeken. "The long and the short of it: summarising event sequences with serial episodes." In: *Proceedings of the 18th ACM International Conference on Knowledge Discovery and Data Mining, Beijing, China*. 2012, pp. 462–470.

[171] Ankit Thakkar and Ritika Lohiya. "A review of the advancement in intrusion detection datasets." In: *Procedia Computer Science* 167 (2020), pp. 636–645.

[172] Andrea Tonon and Fabio Vandin. "Permutation strategies for mining significant sequential patterns." In: *2019 IEEE International Conference on Data Mining*. IEEE. 2019, pp. 1330–1335.

[173] Boudewijn F van Dongen and Arya Adriansyah. "Process Mining: Fuzzy Clustering and Performance Visualization." In: *BPM*. Springer. 2009, pp. 158–169.

[174] Matthijs Van Leeuwen and Arno Siebes. "Streamkrimp: Detecting change in data streams." In: *Machine Learning and Knowledge Discovery in Databases: European Conference, Antwerp, Belgium, September 15-19, 2008, Proceedings, Part I 19*. Springer. 2008, pp. 672–687.

[175] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." In: *Advances in neural information processing systems* 30 (2017).

[176] Nikolai K. Vereshchagin and Paul M. B. Vitányi. "Kolmogorov's Structure Functions and Model Selection." In: *IEEE Transactions on Information Theory* 50.12 (2004), pp. 3265–3290.

[177] Cédric Villani. *Optimal Transport: Old and New*. Grundlehren Der Mathematischen Wissenschaften 338. Berlin: Springer, 2009.

[178] A Yu Volkova. "A refinement of the central limit theorem for sums of independent random indicators." In: *Theory of Probability & Its Applications* 40.4 (1996), pp. 791–794.

[179] Jilles Vreeken, Matthijs Van Leeuwen, and Arno Siebes. "Preserving privacy through data generation." In: *Seventh IEEE International Conference on Data Mining*. IEEE. 2007, pp. 685–690.

[180] Jilles Vreeken, Matthijs van Leeuwen, and Arno Siebes. "KRIMP: Mining Itemsets that Compress." In: *Data Mining and Knowledge Discovery* 23.1 (2011), pp. 169–214.

[181] Nils Philipp Walter, Jonas Fischer, and Jilles Vreeken. "Finding interpretable class-specific patterns through efficient neural search." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. 8. 2024, pp. 9062–9070.

[182] Jianyong Wang and Jiawei Han. "BIDE: Efficient mining of frequent closed sequences." In: *Proceedings. 20th international conference on data engineering*. IEEE. 2004, pp. 79–90.

[183] Jianyong Wang, Jiawei Han, and Chun Li. "Frequent closed sequence mining without candidate maintenance." In: *IEEE Transactions on Knowledge and Data Engineering* 19.8 (2007), pp. 1042–1056.

[184] Yuan H Wang. "On the number of successes in independent trials." In: *Statistica Sinica* (1993), pp. 295–312.

[185] Donald C Weber. "Accident rate potential: An application of multiple regression analysis of a Poisson process." In: *Journal of the American Statistical Association* 66.334 (1971), pp. 285–288.

[186] Gary M Weiss and Haym Hirsh. "Learning to Predict Rare Events in Event Sequences." In: *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*. Vol. 98. 1998, pp. 359–363.

[187] Boris Wiegand, Dietrich Klakow, and Jilles Vreeken. "Mining easily understandable models from complex event logs." In: *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*. SIAM. 2021, pp. 244–252.

[188] Qitian Wu, Yirui Gao, Xiaofeng Gao, Paul Weng, and Guihai Chen. "Dual Sequential Prediction Models Linking Sequential Recommendation and Information Dissemination." In: *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*. 2019, pp. 447–457.

[189] Zonghan Wu, Shirui Pan, Guodong Long, Jing Jiang, Xiaojun Chang, and Chengqi Zhang. "Connecting the dots: Multivariate time series forecasting with graph neural networks." In: *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 2020, pp. 753–763.

[190] Shuai Xiao, Junchi Yan, Mehrdad Farajtabar, Le Song, Xiaokang Yang, and Hongyuan Zha. "Learning time series associated event sequences with recurrent point process networks." In: *IEEE transactions on neural networks and learning systems* 30.10 (2019), pp. 3124–3136.

[191] Hongteng Xu, Mehrdad Farajtabar, and Hongyuan Zha. "Learning granger causality for hawkes processes." In: *International conference on machine learning*. PMLR. 2016, pp. 1717–1726.

[192] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. "Modeling Tabular data using Conditional GAN." In: *Advances in Neural Information Processing Systems*. 2019.

[193] Sascha Xu, Joscha Cüppers, and Jilles Vreeken. "Succinct Interaction-Aware Explanations." In: *Proceedings of the 31th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2025.

[194] Sascha Xu, Osman Mian, Alexander Marx, and Jilles Vreeken. "Inferring Cause and Effect in the Presence of Heteroscedastic Noise." In: *Proceedings of the International Conference on Machine Learning*. PMLR, 2022.

[195] Shengzhe Xu, Manish Marwah, Martin Arlitt, and Naren Ramakrishnan. "STAN: Synthetic Network Traffic Generation with Generative Neural Models." In: Sept. 2021, pp. 3–29.

[196] Xifeng Yan, Jiawei Han, and Ramin Afshar. "CloSpan: Mining: Closed Sequential Patterns in Large Datasets." In: *SDM*. SIAM. 2003, pp. 166–177.

[197] Chin-Chia Michael Yeh, Nickolas Kavantzas, and Eamonn Keogh. "Matrix profile IV: using weakly labeled time series to predict outcomes." en. In: *Proceedings of the VLDB Endowment* 10.12 (Aug. 2017), pp. 1802–1812.

[198] Show-Jane Yen and Yue-Shi Lee. "Mining Non-Redundant Time-Gap Sequential Patterns." In: *Applied Intelligence* 39.4 (Dec. 2013), pp. 727–738.

[199] Y. Yin, Z. Lin, M. Jin, G. Fanti, and V. Sekar. "Practical gan-based synthetic ip header trace generation using netshare." In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022, pp. 458–472.

[200] Mariko Yoshida, Tetsuya Iizuka, Hisako Shiohara, and Masanori Ishiguro. "Mining Sequential Patterns Including Time Intervals." In: *AeroSense 2000*. Ed. by Belur V. Dasarathy. Orlando, FL, Apr. 2000, pp. 213–220.

[201] Mohammed J Zaki. "SPADE: An efficient algorithm for mining frequent sequences." In: *Machine learning* 42 (2001), pp. 31–60.

[202] Wei Zhang, Thomas Panum, Somesh Jha, Prasad Chalasani, and David Page. "Cause: Learning granger causality from event sequences using attribution methods." In: *International Conference on Machine Learning*. PMLR. 2020, pp. 11235–11245.

[203] Zhong Zhang, Chongming Gao, Cong Xu, Rui Miao, Qinli Yang, and Junming Shao. "Revisiting Representation Degeneration Problem in Language Modeling." In: *Findings*. 2020.

[204] Liang Zhao, Jieping Ye, Feng Chen, Chang-Tien Lu, and Naren Ramakrishnan. "Hierarchical incomplete multi-source feature learning for spatiotemporal event forecasting." In: *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 2085–2094.

[205] C. Zhou, B. Cule, and B. Goethals. "Pattern Based Sequence Classification." In: *IEEE Transactions on Knowledge and Data Engineering* 28.5 (2016), pp. 1285–1298.

[206] Cheng Zhou, Boris Cule, and Bart Goethals. "A pattern based predictor for event streams." In: *Expert Syst. Appl.* 42.23 (2015), pp. 9294–9306.

[207] Ke Zhou, Hongyuan Zha, and Le Song. "Learning social infectivity in sparse low-rank networks using multi-dimensional hawkes processes." In: *Artificial Intelligence and Statistics*. PMLR. 2013, pp. 641–649.

[208] Pasquale Zingo and Andrew Novocin. "Introducing the TSTR Metric to Improve Network Traffic GANs." In: *Advances in Information and Communication*. Ed. by Kohei Arai. Cham: Springer International Publishing, 2021, pp. 643–650.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

`https://bitbucket.org/amiede/classicthesis/`

Final Version as of 2025-12-09 (`classicthesis` version 0.0).