Eindhoven University of Technology
Department of Mathematics and Computer Science
Process Analytics

Probabilistic models in process discovery

Internship Report

Daniël Barenholz

Supervisor:
Dirk Fahland

1.0

Eindhoven, August 2021

Contents

Contents ii

1 Introduction 1
1.1 Background 1
1.2 Research Questions 2
1.3 Method 2
1.4 Findings 3

2 Preliminaries 3
2.1 Process Mining 3
2.2 Probability 6
2.3 Graph Theory 9
2.4 Neural Networks 10

3 Infrequent Behaviour in Process Discovery 10
3.1 Categorisation of Process Discovery Methods 10
3.2 Overview 16

4 Probabilistic Models - An Overview 17
4.1 Graphical Models 17
4.2 Non-Graphical Models 21
4.3 Probabilistic Circuits 23
4.4 Overview 28

5 Combining Probabilistic Models and Process Mining 28
5.1 From Event Data to Random Variables 28
5.2 Method 29

6 Discussion 34
6.1 Threats to validity 34
6.2 Infrequent behaviour 34
6.3 Future work 34

7 Conclusion 35

References 35

Appendix 44
A - Acronyms 44
B - Reflection 46

Acknowledgements

I would like to express my deep gratitude to professor Dirk Fahland, my internship supervisor and overall mentor during my masters, for his patient guidance, enthusiastic encouragement, and useful feedback on this research project, as well as allowing me this great opportunity. My thanks are also extended to my friends, in particular Stijn Beukers and Gerben Beintema, for proofreading and being interested in my work. Finally, I wish to thank my mom for her support and encouragement throughout the project.

1 Introduction

1.1 Background

Process discovery (PD) in process mining is the task of transforming an event log into an interpretable model, usually a Petri net (PN). Traditionally this transformation is achieved in an algorithmic sense, for instance using the α-algorithm [1]. The event log, which is the input data to the algorithm that solves PD, stems from some unknown probability distribution p^*. Empirically any event log contains what would be considered "noise" by any discovery method. This noise, however, is usually not measurement noise — what the field of machine learning (ML) refers to as noise: unwanted data items — but rather various deviations in and variations of the data. We also call this infrequent behaviour, and its presence makes the PD problem more complex: PD algorithms need to somehow deal with (many of) these deviations. Oftentimes the purpose of process discovery is to detect these various deviations and to represent them in an explainable model subject to four main criteria: *fitness*, *precision*, *generalization*, and *simplicity* [2].

Current algorithmic approaches typically aim for identifying main behaviour, and then use conformance checking and model repair methods to study the deviations and differences. This approach of aiming for identifying main behaviour makes a fundamentally incorrect assumption about how said behaviour manifests: it assumes that behaviour that occurs frequently is also considered the main behaviour, and that it can be explained by a simple model. This assumption may initially seem correct, but it does not need to be the case: consider some process with n parallel events (events are explained in Section 2.1.1). To see all behaviour in such process, we need to see $n!$ traces (traces are explained in Section 2.1.1). If some of these $n!$ traces occur more frequently than others, and traces with infrequent directly-follows pairs (see Section 3.1.1) are filtered out by the chosen PD method, then we end up with a more complex and worse model than desired.

Using an ML-based approach attempts to generalise away from such assumptions, as done in the thesis by D. Sommers [3] where a graph convolutional network (GCN) is used to solve the PD problem. GCNs, however, do not consider learning the true underlying distribution p^*, even though this true distribution (p^*) can trivially be used to optimally solve classification, prediction, and data imputation tasks. As such, in this internship the goal is to investigate if using probabilistic models, in particular generative models which learn the true distribution of the data, are useful in solving the PD problem. In other words, we are interested in how probabilistic models are linked to process models, if at all. Probabilistic models are used to perform *inference*: answering particular questions (queries) that we have, based on the underlying true distribution of some dataset. If we can compute an answer to such query in polynomial time, then inference is *tractable* (when saying that inference is tractable, we usually mean that it is tractable in terms of model complexity; see also Section 4). An explainable model is desired, but a trade-off in explainability for superior performance is accepted.

1.2 Research Questions

To systematically answer this question, that is, whether or not we can use probabilistic models and in particular generative models for solving the PD problem, we developed the following specific research questions.

RQ 1 How do process discovery algorithms deal with infrequent behaviour?

RQ 2 Does it make sense to use probabilistic models *directly on event data* for process discovery?

(a) Do we care about tractable inference?

(b) Do we need the expressiveness of intractable models to properly model event logs?

(c) Can probabilistic models learn dependencies within a trace? Specifically:

i. Long-term dependency on a single previous item: $\langle a, \ldots, b \rangle$ where b occurs because earlier a occurred, where there are many events in-between a and b.

ii. Short-term dependency on a single previous item: $\langle a, b \rangle$ where b occurs because earlier a occurred, where there are no events in-between a and b.

iii. Long-term dependency on multiple previous items: $\langle a, b, c \ldots, d \rangle$ where d occurs because earlier a, b, c occurred in precisely those indices in the trace, where there are many events in-between c and d.

iv. Short-term dependency on multiple previous items: $\langle a, b, c \rangle$ where c occurs because earlier a, b occurred in precisely those indices in the trace, where there are many events in-between b and c.

(d) Can we turn a probabilistic model into a petri net? If yes, how?

RQ 3 How do we use use the results of **RQ 2** in process discovery?

The questions under **RQ 2** are theoretically oriented: they ask about particular properties of probabilistic models when applied on event data. **RQ 3** is a follow-up question that is practically oriented: it asks how we can actually *use* the information from **RQ 2**. By conducting this research, we hope to identify which existing probabilistic models can be used to address open problems in process discovery or whether there are larger gaps between both fields.

1.3 Method

The intended methodology to answer the research questions is to first conduct a structured literature study on how infrequent behaviour ("noise") is handled in PD, as well as a literature study on probabilistic models. For PD it is important to understand the various quality criteria (*fitness*, *precision*, *generalization*, and *simplicity*) used to evaluate the various types of models. For probabilistic models the literature study is extensive: a general overview on different types of probabilistic models will be given, as well as a section on the data transformation task, that is, how to transform data into a probabilistic framework. Naturally, the proposed method utilising GCNs is to be included in the literature study on PD methods. By conducting the literature studies, one can compare the state of the art of both PD and probabilistic models to identify similarities and differences between the fields. Some experiments within the two fields may be conducted if necessary to gain a better understanding of relevant papers and models, either by re-running existing experiments or applying existing implementations. It is evidently also important to experiment using sequential data (e.g. event logs) on some suitable probabilistic model to gain experience with how probabilistic models can handle sequential data. This suitability follows from the performed literature study.

1. Conduct a literature study on how infrequent behaviour is handled in the process discovery phase.
2. Create a systematic literature overview in a new field: probabilistic models.

   (a) Show a general overview of probabilistic models.
   (b) Show specific literature pertaining to data transformation tasks.

3. Compare techniques and research problems of different fields: process discovery and probabilistic models.

   (a) Apply existing implementations / Re-run existing experiments.

4. Identify research gaps and assess how they could be closed.

   (a) Run experiments that uses both fields (e.g. experiment using event log data on some suitable probabilistic model).
   (b) Investigate feasibility of using probabilistic models on sequential data (e.g. event logs).

1.4 Findings

We propose a method that allows any probabilistic model to be used as a preprocessing step for any PD method. It includes a conversion from input data (logs) to random variable (RV)s, as well as multiple ideas on converting from a learned distribution ($\hat{p}(x)$) to an event log, which can then be used by some PD method. Using this preprocessing step, which is effectively a way to filter data, one can select precisely which traces are deemed interesting, based on probability. The methodology is explained in Section 5. We also discuss the similarities between learning process models and learning probabilistic models.

The remainder of this report is structured as follows. Section 2 contains preliminaries for understanding the fields of process mining and probabilistic models. In Section 3 we elaborate on how PD methods handle infrequent behaviour, which answers **RQ 1**. Section 4 attempts to give a categorisation of various existing probabilistic models, reporting on what each model is commonly used for, and in Section 5.1 we demonstrate that interpreting event data as a random variable (RV) is non-trivial. In particular, we show an interesting way to interpret events as RVs under the assumption that we use a *simple* event log. The report is concluded by a discussion on our results and on future directions for research in Section 6, and a conclusion in Section 7.

2 Preliminaries

This section contains all required background knowledge from the process mining field (Section 2.1), from probability (Section 2.2), from graph theory (Section 2.3) and from ML (Section 2.4) to understand the conducted research.

2.1 Process Mining

In process mining we assume there is some real-life (unknown) system S. This system, commonly called the underlying process, can be anything. Examples range from ordering something from a web-shop to commuting to work and even how a caterpillar turns into a butterfly. Since the system S is unknown, we cannot do analysis on it. In stead, we have to use concrete datasets: event logs L, containing data on various executions of S. Since L may be (very) complex, we (usually) cannot use it directly. As such, we want to use L to create some model M that describes S, subject to some criteria of "goodness", usually *fitness*, *precision*, *generalization*, and *simplicity.*

2.1.1 Event Log

An **event**, usually denoted with lowercase letters $a, b, \ldots$, are atomic elements of activity as observed during a particular execution of a process. Since events are atomic elements of activity, we sometimes refer to an event as an activity, even if this is formally not the correct name. A **trace** σ, denoted as a sequence of events surrounded by $\langle\cdot\rangle$, describes the execution of one specific

instance, commonly called a **case**, of the logged process. Traces formally consist of an arbitrary number of events, including zero. A collection of traces is an **event log** L, sometimes simply called a *log*. A log commonly is the input data to any process discovery algorithm. It contains all event information that is related to a particular process. Formally, an event log is a multi-set of any number of traces. It may be an empty set, but in such case there is no meaningful process mining to be done. We use $L(\sigma)$ to denote the frequency of particular trace σ in some log L. Note that there are various standards that allow to represent event logs in digital format such as MXML [4] and XES [5], the latter being backed by IEEE.

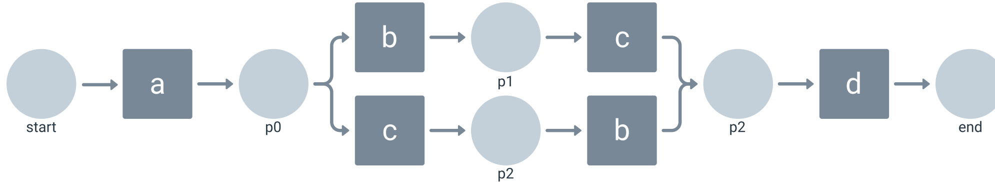

Figure 1: A labelled Petri net corresponding to log $L = \{\langle a, b, c, d\rangle^2, \langle a, c, b, d\rangle^1\}$

To further illustrate the elements of an event log, consider some log $L = \{\langle a, b, c, d\rangle^2, \langle a, c, b, d\rangle^1\}$, which contains a total of 2 cases (number of unique traces). For the first trace, one sees $L(\langle a, b, c, d\rangle) = 2$: this trace occurs 2 times in the log L. Similarly, for the second trace we have $L(\langle a, c, b, d\rangle) = 1$: it occurs a singular time only. There are a total of 4 observed events (a, b, c, d), and each case starts by executing activity a, and ends with executing activity d. In between the starting and ending activity there must be an execution of activities b and c in any order. Looking at an event log can provide us with interesting insights into how a process works as illustrated, but usually an event log contains *many* items, which makes manually looking at the event data an infeasible task.

2.1.2 (Labelled) Petri Nets & Workflow Nets

A **Petri net (PN)** is an example of such model M. Formally, it is a triple (P, T, F), where P is a finite set of *places* (denoted by circles), T a finite set of *transitions* (denoted by squares or rectangles) such that $P \cap T = \emptyset$ and $F \subseteq (P \times T) \cup (T \times P)$ a set of directed arcs called *flow relation*. A **labelled Petri net** is a tuple (P, T, F, A, ℓ) with P, T and F identical to the definitions in a PN, A a set of activity labels and $\ell : T \to A$ a labelling function. An example of a labelled PN is shown in Figure 1, corresponding to the example log L as given in Section 2.1.1. A **workflow net (WF-net)** is a specific type of labelled PNs where it is required that there are unique source and sink places, and each node is on some path from the source to the sink.

A key point of modelling with PNs is *replay*, informally referred to as "playing the token game". **Tokens**, denoted by black dots within a place, have no real meaning — except when working with a Coloured Petri net (CPN) where tokens are given data properties — and form the base for any simulation performed with PNs. For a transition t to *fire*, it needs a minimum of 1 token in each of its' incoming places. When t fires, it *consumes* 1 token from each of its' incoming places, and subsequently *produces* 1 token in each of its' outgoing places. The incoming and outgoing places are formally called **preset** and **postset**, and they are denoted by $^\bullet t$ and $t^\bullet$ respectively. A PN combined with tokens is referred to as a **marked** Petri net. The initial state of these tokens is thus called an initial marking, where markings are formally functions $M : T \to \mathbb{N}$ that assign a natural number to transitions. Knowing that markings exist, and realising that they may be used to model particular constructs, suffices.

As hinted in previous paragraph, there are different modelling constructs supported by (labelled) PNs (or WF-nets). A few are sketched below. The easiest one is **sequential** execution, as illustrated in Figure 2: first transition a fires, after which transition b fires.

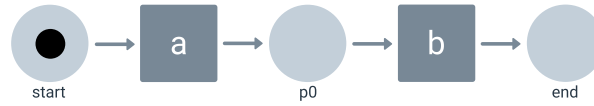

Figure 2: A sequence.

Since transitions can only fire if all input places have (at least one) token in them it means that we can model decisions, or **choices**, as illustrated in Figure 3: after firing a, only b or c can fire, but not both. As such, this models a decision between b (and anything modelled after the transition b has fired) and c (and anything modelled after the transition c has fired). Usually we call this structure an XOR-split. There are also XOR-joins, which, as the name implies, allows to join a decision back into the 'main' flow, as illustrated in Figure 4: regardless of a c or b that fires, the next transition that gets enabled is always d. Note that Figure 1 is an example that combines both an XOR-split and XOR-join, effectively modelling a choice between the 2 sequences $\langle b, c \rangle$ and $\langle c, b \rangle$.

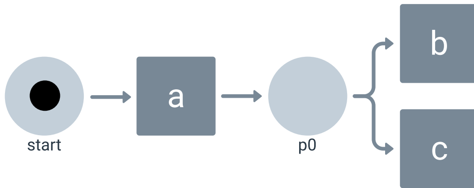

Figure 3: An XOR-split.

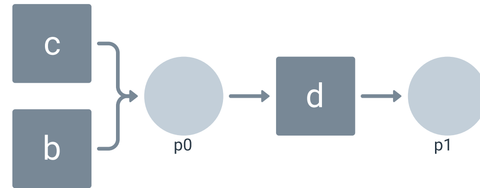

Figure 4: An XOR-join.

Similar to choices PNs can also be used to model **concurrency**. We call this AND-splits and AND-joins, as illustrated in Figures 5 and 6 respectively: after a fires the sequences starting with b and c can fire independently, allowing for concurrent execution.

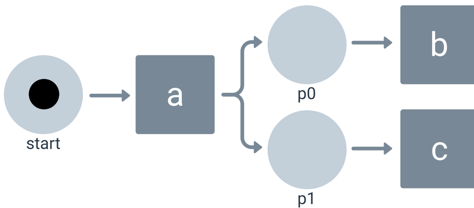

Figure 5: An AND-split.

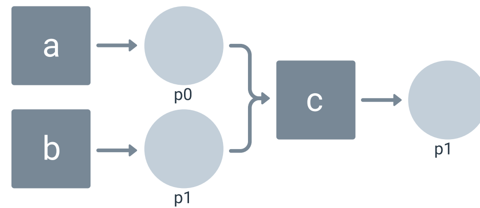

Figure 6: An AND-join.

2.1.3 Conformance Checking

Conformance checking in process mining effectively computes measures of "goodness" of a model. These measures are important for all types of models, but for the sake of our discussion here we constrain ourselves to the Petri net family. A conformance checking method (specific to the PN family) generally computes 4 measures by "playing the token game" as mentioned earlier: *fitness* (the discovered model should allow for behaviour as seen in log L), *precision* (the discovered model should not allow for behaviour completely unrelated to what was seen in log L), *generalisation*,(the discovered model should generalise the example behaviour as seen in log L) and *simplicity* (the discovered model should be as simple as possible) [6].

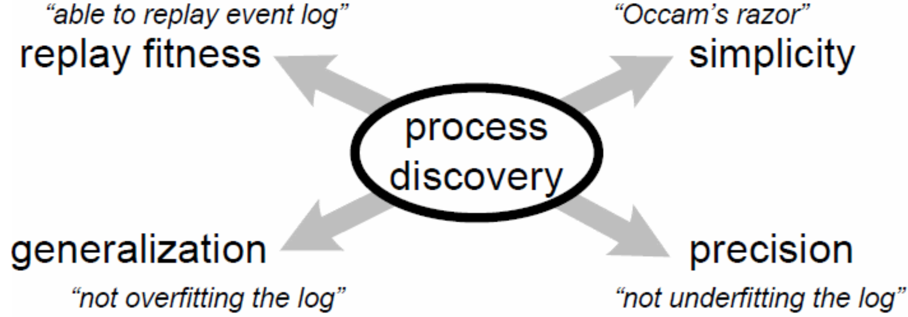

Figure 7: The 4 measures of "goodness" in PD [6].

2.2 Probability

2.2.1 Probability Space

All *hard* or *complex* computations related to probability can be brought back to the basic notion of a **probability space**. As such, we deem it important that the reader is knowledgeable on the basic definitions of probability. We always use the same running example where applicable: rolling a fair 6-sided die, where fair means that all possible outcomes have equal probability. The set of all possible outcomes is the **sample space**, denoted by Ω. Formally, Ω can be any non-empty set, and it describes the universe in which randomness takes place. Elements $\omega \in \Omega$ are called **atomic events**. When we roll the die, we assume that there exists some *random selection mechanism* that picks such atomic event ω from Ω. The main idea of probability is to capture this randomness. One way of capturing this randomness is the frequentist definition (or interpretation) of probability: suppose that we use the random selection mechanism infinitely many times, then the probability that the outcome is a particular atomic event ω is equal to the counting how many times it occurs, and dividing it by the number of trials n approaching infinity as shown in Equation 1.

$$p(\omega) := \lim_{n \to \infty} \frac{\sum_{i=1}^{n} \mathbb{1}\left[\omega_i = \omega\right]}{n} \tag{1}$$

We can extend atomic events to **events**: an **event** is any subset $A \subseteq \Omega$, where (by definition) the elements of A are ω. In the running example, we could define an event for "rolling an even number" i.e. $A = \{2, 4, 6\}$. By extending the frequentist definition of probability to sets we define a **probability measure**, as shown in Equation 2. Note the relation between two definitions: $\mathbb{P}(\{\omega\}) = p(\omega)$.

$$\mathbb{P}(A) := \lim_{n \to \infty} \frac{\sum_{i=1}^{n} \mathbb{1}\left[\omega_i \in A\right]}{n} \tag{2}$$

As can be seen, a probability measure is defined on subsets of the sample space. Rosenthal [7] demonstrated, however, that it is impossible to assign probability to *all* subsets of some Ω. Instead, we must use a σ-algebra over the sample space. A **σ-algebra** Σ defined on some sample space Ω is a set of subsets $A \subseteq \Omega$ — one commonly also sees the technically incorrect notation $A \in \Omega$ — such that it contains the empty set ($\emptyset \in \Sigma$), it is closed under complement ($A \in \Sigma \implies \overline{A} \in \Sigma$), and it is closed under union ($\forall A_1, A_2, \ldots \in \Sigma : (\bigcup_i A_i) \in \Sigma$). By definition of the σ-algebra it is possible to assign probability to all $A \in \Sigma$.

Finally, a **probability space** is a triple $(\Omega, \Sigma, \mathbb{P})$ consisting of some sample space Ω, a σ-algebra Σ defined on all events $A \subseteq \Omega$, and a probability measure $\mathbb{P} : \Sigma \to \mathbb{R}$ mapping events A to the real numbers $\mathbb{R}$ such that $\mathbb{P}(\emptyset) = 0$ and $\mathbb{P}(\Omega) = 1$.

2.2.2 Random Variables

Let $(\Omega, \Sigma, \mathbb{P})$ be some probability space as defined above, and let $\mathcal{X}$ be some space, for instance the real numbers. Any *measurable* function $X : \Omega \to \mathcal{X}$ is a RV. Note how RVs are functions (not variables), and not random. If $\mathcal{X}$ is countable, then we say that X is a **discrete** RV, and in this case we have a **probability mass function (PMF)** $p_X : \mathcal{X} \to \mathbb{R}$ that specifies the probability measure $\mathbb{P}_X$ as $p_X(x) := \mathbb{P}_X(\{x\})$. Suppose that $\mathcal{Y}$ is an uncountable set, and that $Y : \Omega \to \mathcal{Y}$. We call Y a **continuous** RV, on which we define a probability density function (PDF) $p_Y : \mathcal{Y} \to [0, \infty)$ such that $\forall A \in \Omega : \int_A p_Y(y)dy = \mathbb{P}_Y(A)$. Notice how p_X is overloaded: it can be both a PDF or PMF. To make notation less cumbersome, p is simply called a *density*. When in context it is clear about which RV we are talking about, the subscript is dropped.

Suppose we have RVs $X_1, \dots, X_n$ mapping Ω to $\mathcal{X}_1, \dots, \mathcal{X}_n$ respectively. A **random vector**, or multivariate random variable, is then defined as $\mathbf{X} = (X_1, \dots, X_n)$. We call $\mathcal{X} = \times_{i=1}^{n} \mathcal{X}_i$ a **joint state space**. If each X_i is discrete, then $\mathbf{X}$ is discrete, and has a **joint PMF**. If any X_i is continuous, then $\mathbf{X}$ is continuous, and has a **joint PDF**.

2.2.3 Marginalisation

Marginalisation provides one routine that is used during inference — the act of manipulating (joint) distributions to answer a particular query (question). Let p be the distribution of some random vector $\mathbf{X}$, and let $\mathbf{Y}$ and $\mathbf{Z} = \{Z_1, \dots, Z_k\}$ be any partition of $\mathbf{X}$, that is, $\mathbf{Y} \cap \mathbf{Z} = \emptyset$ and $\mathbf{Y} \cup \mathbf{Z} = \mathbf{X}$. Then we define the **marginal** distribution of $\mathbf{Y}$ as shown in equations 3 (continuous case) and 4 (discrete case). Note that we abuse notation, as commonly done in the field of probability: we say that some vector $\mathbf{z}$ can be written as a list of its components $z_1, \dots, z_k$ without caring about order, even though order usually matters.

$$p(\mathbf{Y}) = \int_{\mathcal{Z}} p(\mathbf{y}, \mathbf{z}) d\mathbf{z} = \int_{\mathcal{Z}_1} \dots \int_{\mathcal{Z}_k} p(\mathbf{y}, z_1, \dots, z_k) d\mathbf{z}_1 \dots dz_k \tag{3}$$

$$p(\mathbf{Y}) = \sum_{\mathcal{Z}} p(\mathbf{y}, \mathbf{z}) = \sum_{\mathcal{Z}_1} \dots \sum_{\mathcal{Z}_k} p(\mathbf{y}, z_1, \dots, z_k) \tag{4}$$

Marginalization can be seen as "summing out" a (subset of) particular RVs. In Bayesian reasoning, the marginal distribution is called the **prior**.

2.2.4 Conditioning

Conditioning provides another basic routine that is used during inference. Let $p(\mathbf{Y}, \mathbf{Z})$ be the joint distribution of some $\mathbf{Y}$ and some $\mathbf{Z}$. The **conditional distribution** is defined as shown in Equation 5 for the continuous case. For the discrete case, the integral is replaced with a summation. Note how marginalization is used in the step denoted by $\stackrel{*}{=}$.

$$p(\mathbf{y}|\mathbf{z}) = \frac{p(\mathbf{y}, \mathbf{z})}{p(\mathbf{z})} \stackrel{*}{=} \frac{p(\mathbf{y}, \mathbf{z})}{\int_{\mathcal{Y}} p(\mathbf{y}, \mathbf{z}) d\mathbf{y}} \tag{5}$$

Conditioning can be seen as introducing evidence. An alternative interpretation is fixing random variables to a particular value that is already known. In Bayesian reasoning, the conditional distribution is a **likelihood**. The joint distribution gives rise to the **posterior** $p(\mathbf{z}|\mathbf{y})$.

2.2.5 (Conditional) Independence

Independence is computationally a highly desirable property. For two random vectors $\mathbf{X}$ and $\mathbf{Y}$, we say that $\mathbf{X}$ and $\mathbf{Y}$ are independent, denoted by $\mathbf{X} \perp\!\!\!\perp \mathbf{Y}$, if *any* of the following conditions hold:

- $p(\mathbf{X}, \mathbf{Y}) = p(\mathbf{X}) \cdot p(\mathbf{Y})$
- $p(\mathbf{X} \mid \mathbf{Y}) = p(\mathbf{X})$

- $p(\mathbf{Y} \mid \mathbf{X}) = p(\mathbf{Y})$

We can also say that two random variables **X** and **Y** are conditionally independent given **Z**, denoted by $\mathbf{X} \perp\!\!\!\perp \mathbf{Y} \mid \mathbf{Z}$, if any of the following conditions hold:

- $p(\mathbf{X}, \mathbf{Y} \mid \mathbf{Z}) = p(\mathbf{X} \mid \mathbf{Z})p(\mathbf{Y} \mid \mathbf{Z})$
- $p(\mathbf{X} \mid \mathbf{Y}, \mathbf{Z}) = p(\mathbf{X} \mid \mathbf{Z})$
- $p(\mathbf{Y} \mid \mathbf{X}, \mathbf{Z}) = p(\mathbf{Y} \mid \mathbf{Z})$

Note that unconditional independence is a special case with $\mathbf{Z} = \emptyset$.

2.2.6 Probabilistic queries

Queries (q) are particular questions that we want an answer for. A probabilistic model (m) gives us an answer to these queries by using inference routines such as marginalisation (Section 2.2.3) and conditioning (Section 2.2.4). For the purpose of this work we consider the different query types as explained in [8]. Some queries are 'easier' than others, which is formalised by the computation of answering them being *tractable*. If some query is tractable, it just means that it is realistically feasible to compute it within acceptable time-frame. Formally, if an algorithm solves a problem of size n in some number of time steps that can be expressed as a polynomial function of n, then we say that the problem is solved tractably in terms of its size.

1. **EVI – Complete Evidence** These type of queries ask a question with complete evidence, or complete knowledge, of the things that the query depends on. In this meaningless example, $\mathbf{X} = \{A, B, C\}$ is a random vector, consisting of 3 random variables.

$$q(m) = p_m(\mathbf{X} = \{a, b, c\}) = p_m(A = a, B = b, C = c)$$

2. **MAR – Marginal Queries** These types of queries are similar to EVI, but do not provide complete evidence, only partial evidence. If MAR is tractable, conditional queries are tractable too. Note that MAR in its general form is an integral. Here, B is not provided, hence the query is a marginal one.

$$q(m) = p_m(A = a, C = c)$$

3. **CON – Conditional Queries** These queries are simply conditional probabilities. An example: "What is the probability that Daniël is going to Veghel, given that today is a Thursday?", then **q** denotes the event that Daniël goes to Veghel, and **e** denotes that today is a Thursday.

$$q(m) = p_m(\mathbf{q} \mid \mathbf{e}) = \frac{p_m(\mathbf{q}, \mathbf{e})}{p_m(\mathbf{e})}$$

4. **MAP – Maximum A Posteriori** Also commonly referred to as Most Probably Explanation (MPE): These queries are particularly interesting. An example: "Which type of cheese is most likely to be bought by Daniël on Mondays?", then the evidence (**e**) then is the day of the week (Monday) and the fact that the buyer is Daniël, while the query (**q**) is about cheese.

$$q(m) = \arg\max_{\mathbf{q}} p_m(\mathbf{q} \mid \mathbf{e})$$

5. **MMAP – Marginal MAP** A marginalised variant of MAP where we do not introduce full, but only partial evidence. This is also referred to as *Bayesian Network MAP*. MMAP is to MAP what MAR is to EVI. So, an example would then be "Which type of cheese is most likely to be bought by Daniël?", where the day of the week is not given. Note that the summation over **h** is a summation over *hidden* or *latent* variables: these are the ones not present in the query. Furthermore, $\mathbf{Q} \cup \mathbf{H} \cup \mathbf{E} = \mathbf{X}$, with **X** being the original random vector.

$$q(m) = \arg\max_{\mathbf{q}} \sum_{\mathbf{h}} p_m(\mathbf{q}, \mathbf{h} | \mathbf{e})$$

6. ADV – Advanced Queries These queries are even more complicated than previous ones. You can see them as any combination of MAP, logical events, counts, group comparisons, ... The example from [8] asks "Which day is most likely to have a traffic jam on my route to campus?", which is represented as shown below.

$$q(m) = \arg\max_d p_m \left(\text{Day} = d \wedge \bigvee_{i \in \text{route}} \text{Jam}_{\text{street}_i} \right)$$

2.2.7 Curse of Dimensionality: why models exist

A final item to note is that the curse of dimensionality plays a large role when using probabilistic models. In Section 2.2.6 we show that there are various queries q that can be answered using a probabilistic model m by means of marginalisation (Section 2.2.3) and conditioning (Section 2.2.4). Let us consider an example, where we want to model a system of ten physical particles with 'spin up' and 'spin down' states, modelled as binary RVs. The complete joint distribution in this scenario has a total of $2^{10} = 1024$ combinations that it can take. A thousand states is fine to represent on a computer as-is. Now consider that we have twenty particles in stead. Then, the complete joint distribution has $2^{20} = 1048576$ combinations, which while still doable, already might run into some problems on older systems to represent. In the case that there are 40 particles, we can no longer represent the joint distribution directly, as $2^{40} = 1099511627776$ is simply too large. By using a model m that represents probability in a different way from just raw data, we (sometimes) can still deal with these extremely high numbers.

2.3 Graph Theory

2.3.1 Basic definitions

An **undirected graph** $G = (V, E)$ is a pair of sets V and E, with elements $v \in V$ being **nodes** (also called vertices), and elements $e \in E$ being **undirected edges** (also called lines or links). Undirected edges are defined as a set of exactly two vertices $e = \{u, v\}$ with $u \neq v$ and $u, v \in V$. For some $e = \{u, v\}$ we say that u and v are **neighbours**. We denote the set of neighbours as $\mathbf{nb}(v)$.

Similarly, a **directed graph** $G' = (V, E')$ is a pair of sets V and E', with V being identical to the undirected case, but elements $e \in E'$ are now **directed edges**, defined as ordered pairs (u, v) where $u \neq v$ and $u, v \in V$. If $(u, v) \in E$ then there is a *directed* edge from node $u \in V$ to node $v \in V$: u is the **parent** of v and v is the **child** of u. We denote the set of parents as $\mathbf{pa}(v)$ and the set of children as $\mathbf{ch}(v)$.

Given a graph (directed or undirected) G, we can define a **walk** as a sequence of nodes $(v_1, \ldots, v_n)$ such that for each two consecutive nodes v_i, v_{i+1} it holds that they are neighbours. A **directed walk** is a walk where in stead of neighbours we have that for consecutive nodes v_i and v_{i+1} the latter is a child of the first. If we constrict a walk to only unique nodes, then we call it a **path**. Note that there may be an undirected walk in a directed graph (that may go against the direction of the arrows). If between every pair of nodes u and v there is a walk, then we call the graph **connected**. **Cycles** are walks $(v_1, \ldots, v_n)$ with $v_1 = v_n$, and they can be directed if the walk itself is directed. An **acyclic** graph is a graph without cycles. Finally, a **clique** is a subset of nodes $v_1, v_2, \ldots$ of an undirected graph such that every two nodes v_i and v_j are neighbours, where $v_i \neq v_j$. A **maximal clique** is a clique that cannot be extended by including one more neighbour.

2.3.2 Types of Graphs

By requiring some graph G to be acyclic or connected we can distinguish between different types of graphs. As "base" graph G we consider undirected graphs. In particular, we distinguish between

forests, **trees**, and their directed equivalents. Note that the distinction as made in this work is not universal: there exist multiple naming conventions of which we simply chose one.

Forest If the only restriction that we impose on G is *acyclicity*, then we have a forest. Nodes v in a polyforest may have any number of neighbours, and there may be any number of components.

Polyforest If we require G to be both *acyclic* and *directed*, then we call it a polyforest. A polyforest is also called a *directed forest* or a directed acyclic graph (DAG). Nodes v may have any number of parents or children, and there may be any number of components.

Tree If we not only require *acyclicity*, but also *connectedness*, then we have a polytree. Similar to a polyforest, nodes v in a polytree may have any number of neighbours, but there is only one component.

Polytree If we require G to be *acyclic*, *connected*,and *directed*, then we call it a polytree. A polytree is also called a *directed tree*. Nodes v may have any number of parents or children, but there is only one component.

2.4 Neural Networks

Deep generative models are a subclass of probabilistic models utilising various types of neural networks. As such, we assume the reader is familiar with following concepts: 1. dense and convolutional layers, 2. activation functions such as *sigmoid*, 3. Jacobian matrices. For a proper introduction to neural networks see [9].

3 Infrequent Behaviour in Process Discovery

We categorise different process discovery methods and explain their main ideas in Section 3.1. In Section 3.2 we summarise how each of the investigated methods handle infrequent behaviour.

3.1 Categorisation of Process Discovery Methods

Previous studies have categorised [10, 11] or otherwise summarised [12] various process discovery (PD) methods. Based on these studies and further investigation of the plethora of methods for PD we consider following approaches: algorithmic, genetic, clustering-based, integer linear programming (ILP), heuristic, and ML. For each category we describe the main idea and illustrate this, where necessary, using an example discovery method. If applicable, we sketch commonalities between how PD methods in this category handle behaviour.

3.1.1 Algorithmic[1]

Since PD methods categorised as algorithmic do not particularly overlap, we sketch for each algorithm separately how infrequent behaviour is handled. To ensure mutual understanding of what a PD method actually does, we show how the α-algorithm [1] works.

[1]One can argue that *all* methods are algorithmic in some sense. Look at the algorithmic category as containing PD methods that do not fit in the other categories.

Algorithm 1: The α-algorithm [1].

input : An event log L
output: A PN $\alpha(L) = (P_L, T_L, F_L)$

1 $T_L = \{t \in T \mid \exists \sigma \in L : t \in \sigma\}$
2 $T_I = \{t \in T \mid \exists \sigma \in L : t = \mathbf{first}(\sigma)\}$
3 $T_O = \{t \in T \mid \exists \sigma \in L : t = \mathbf{last}(\sigma)\}$
4 $X_L = \{(A, B) \mid (A, B \subseteq T_L) \wedge (A \neq \emptyset) \wedge (B \neq \emptyset) \wedge (\forall a \in A \forall b \in B : a \rightarrow b) \wedge (\forall a_0, a_1 \in A : a_0 \# a_1) \wedge (\forall b_0, b_1 \in B : b_0 \# b_1)\}$
5 $Y_L = \{(A, B) \in X_L \mid \forall (A', B') \in X_L : (A \subseteq A') \wedge (B \subseteq B') \implies (A, B) = (A', B')\}$
6 $P_L = \{p_{(A,B)} \mid (A, B) \in Y_L\} \cup \{i_L\} \cup \{o_L\}$
7 $F_L = \{(a, p_{A,B)}) \mid (A, B) \in Y_L \wedge a \in A\} \cup \{p_{(A,B)}, b \mid (A, B) \in Y_L \wedge b \in B\} \cup \{(i_L, t) \mid t \in T_I\} \cup \{(t, o_L) \mid t \in T_O\}$
8 **return** $\alpha(L) = (P_L, T_L, F_L)$

The α-algorithm first creates a set of all transitions (line 1), and it fixes the set of start (line 2) and end (line 3) activities by looking at traces. Then, it internally constructs directly follows relations, namely *direct succession* ($a > b$ if for some trace we have $\langle \dots, a, b, \dots \rangle$), *causality* ($a \rightarrow b$ if $a > b$ but not $b > a$), *choice* ($a \# b$ if neither $a > b$ or $b > a$) and *parallel* ($a || b$ if both $a > b$ and $b > a$). Using these relations it computes pairs of sets of transitions (line 4), deletes non-maximal ones (line 5), and determines places from them (line 6). A source (i_L) and sink (o_L) place is added to the collection of places. The flow relation is created by connecting computed places and transitions (line 7), which allows algorithm to return a Petri net (P_L, T_L, F_L).

Shortcomings of the α method include self-loops (L1L, for instance $L = \{\langle a, c \rangle, \langle a, b, c \rangle, \langle a, b, b, c \rangle, \langle a, b, b, b, c \rangle\}$, where there is a loop of length 1 for activity b), short loops (L2L, for instance $L = \{\langle a, b, d \rangle, \langle a, b, c, b, d \rangle, \langle a, b, c, b, c, b, d \rangle\}$, where there is a loop of length 2 for subtrace b, c), long term dependencies ($L = \{\langle a, c, d \rangle, \langle b, c, e \rangle\}$, where if a happens then later d happens, or if b happens then later e happens), invisible/optional tasks ($L = \{\langle a, b, c \rangle, \langle a, c \rangle\}$, where b is optional) and non-free choice constructs. To alleviate these shortcomings Wen et al. developed α^{++} to discover invisible tasks [13], $\alpha^{\#}$ to discover non-free choice constructs [14], and later $\alpha^{\$}$ that takes the good points of $\alpha^{\#}$ and α^{++} and adds a way to deal with L1L and L2L constructs [15].

The investigated algorithmic PD methods:

1. $\alpha^{\$}$ [15] and its predecessors ($\alpha^{\#}$ [14], α^{++} [13], and α [1]), which all take input log L as-is without filtering behaviour. There is no distinction between infrequent and frequent behaviour.
2. *Declare miner* [16], which attempts to give temporal constraints based on a log. Declarative models (such as the one this paper produces) do not assume block-structure, and hence are more suitable for processes that have less structure. It works by generating a set of candidate constraints, and then prunes these constraints based on measures taken from association rule mining (support and confidence). There are thresholds for these measures, meaning that the model is filtered based on the frequency of particular model constructs. There is, however, no distinction between (nor model of) frequent and infrequent behaviour.
3. *Data-aware Declare miner* [17], which is similar to the *Declare miner* [16] but adds in a data aspect. It effectively extends the Declare constraints from [16] with data conditions in the form of first-order temporal logic, and then uses these extended constraints to discover a Declare model. The model is filtered based on the frequency of particular model constructs. There is, however, no distinction between (nor model of) frequent and infrequent behaviour.
4. *MINERful*$^{++}$ [18], which works in two steps. It first builds a knowledge base, and then uses said knowledge base to discover a set of constraints weighted by their support (calculated as normalised fraction of cases in which the constraint is verified over some set of input traces). There is a threshold on minimum required support for a particular constraint, so similar to

[16] and [17], *MINERful*$^{++}$ filters the model on the frequency of particular model constructs. There is, however, no distinction between (nor model of) frequent and infrequent behaviour.

5. *Process Skeletonization* [19], which creates process maps of some user-settable abstraction. There is no difference between infrequent and frequent behaviour, and the desired abstraction may be set such that no behaviour is filtered at all (though, arguably, this does not produce a usable model due to its complexity). Note that the abstraction level is applied *after* discovery; it only influences the shown model by hiding certain parts, but the discovery method stays identical.
6. *WoMaN* [20], which is based on first-order logic and uses inductive logic programming to discover both a set of facts expressing a workflow schema, and a set of rules expressing conditions on them. It has user-settable parameter N for "noise handling" which allows particular (infrequent) transitions to be filtered if they don't meet the threshold. There is no explicit model of infrequent and frequent behaviour, but infrequent behaviour can be consciously filtered on activity-level.
7. *Hybrid miner* [21], which automatically divides the log into declarative and procedural parts. How it handles behaviour is subject to the underlying PD methods used for the respective parts.
8. *DAG Extraction* [22], which creates a DAG from a log L. Since dummy nodes are used to explain loops, all behaviour from L is also present in the extracted graph. The method does not mention infrequent or frequent behaviour; it only cares that the extracted graph is representative of L.
9. *CNMining* [23], which converts the task of PD into a Constraint Satisfaction Problem (CSP) [24] and solves said CSP to discover causal networks. It takes parameters δ and τ, the latter being a threshold for filtering edges on the created precedence graph. For $\tau = 0$ the graph remains unchanged, and there is no filtering. There is no distinction between (nor model of) frequent and infrequent behaviour, but behaviour may be filtered due to the workings of the algorithm.
10. *Maximal Pattern Matching* [25], which, as the name implies, looks at the PD problem as a maximal pattern matching one — given some graph $G = (V, E)$, a matching M in G is a set of pairwise non-adjacent edges such that no two edges share common vertices; a maximal matching is a matching that is not a subset of any other matching — to discover a process map. The authors reflect on different types of noise (infrequent behaviour and ML noise) and they allow the method to filter behaviour using a trace-frequency based threshold *thresh*.
11. *RegPFA* [26], which discovers a regularised probabilistic finite automata as model. It is a probabilistic technique that models probability distributions over the set of all conceivable event sequences. The paper contains described two software artifacts, one for learning the distributions, and one for visualising them. Within the visualiser (which converts the learned distributions to an interpretable model), the user can set a desired abstraction level ε such that the visualised model is filtered, or unfiltered. Note that the abstraction level is applied *after* discovery; it only influences the shown model by hiding certain parts, but the discovery method stays identical.
12. *CSMMiner* [27], which discovers composite state machines (CSM) with a focus on different process perspectives. The visualisation of the model utilises thresholds for support, confidence and lift (measures from association rule learning) to decide which parts are visible, or not visible. Note that the abstraction based on the thresholds is applied *after* discovery; it only influences the shown model by hiding certain parts, but the discovery method stays identical.
13. τ [28], which constructs a token log and mines a PN from said token log. All original traces are considered for the token log: there is no distinction between frequent and infrequent behaviour, and there is no filtering.

14. *PGminer* [29], which constructs Conditional Partial Order Graphs (CPOG) representations of logs. The paper proposes exact CPOG and concurrency-aware CPOG approaches, both discovering models that cover all traces of the log. There is no distinction between frequent and infrequent behaviour, and there is no filtering.
15. *SQL Miner* [30], which, as the name implies, utilises SQL to mine declarative constraints from event log data stored in RXES format [31]. The method uses *minSupp* and *minConf* thresholds for support and confidence respectively, meaning that the resulting model is filtered based on the frequency of particular model constructs. There is, however, no distinction between (nor model of) frequent and infrequent behaviour.

3.1.2 Genetic

A genetic algorithm is one that attempts to mimic Darwinian evolution [32], and is based on the ideas of *crossover*, *mutation* and *selection*. In general terms, a genetic algorithm consists of five phases, of which the last three are looped until some condition has been met, usually convergence of the set of solutions.

1. **Initial population** Create (randomly or according to some strategy) n valid *solutions* to the problem that needs to be solved. For instance, n neural networks that solve a particular classification task.
2. **Fitness function** Define what it means for a solution to be *good*. The algorithm needs this function during the selection stage.
3. **Selection** Select a subset of individuals (solutions) that are fittest (best), to be used during crossover.
4. **Crossover** This is the most significant phase. For each pair of individuals (solutions) gained during the selection phase, generate a child solution that possesses properties from both parents. These are added to the population.
5. **Mutation** Mutate a solution, based on some probability. Mutation is necessary to maintain diversity and prevent premature convergence.

Since a genetic algorithm defines a fitness function that describes how good a solution is, the behaviour with respect to infrequent behaviour rests on this fitness function. If the method assumes that frequent behaviour is main behaviour when creating such fitness function, then by design of the algorithm infrequent behaviour will be filtered out.

The investigated genetic PD methods:

16. *Evolutionary Declare Miner (EDM)* [33], which takes its input log L as-is and specifically states to not have a guarantee on fitness (in the process sense, not genetic sense).The genetic fitness function is *not* based on the usual four quality criteria (*fitness*, *precision*, *generalisation*, *simplicity*) but is an harmonic mean between "fitness" and "precision" measures defined in the paper. These measure do *not* explicitly take frequency of behaviour into account. As such, there is no difference between infrequent and frequent behaviour, but behaviour (both infrequent and frequent) may be unintentionally filtered out due to the workings of the method.
17. *Evolutionary Tree Miner (ETM)* [34], which *does* take frequency of behaviour into account in its fitness function that is a combination of the usual four quality criteria (*fitness*, *precision*, *generalisation*, *simplicity*), where each criteria has its own user-settable parameter to control how important it is. It is thus possible to ensure that ETM only cares about fitness (in the process sense, not genetic sense), ensuring that no behaviour is filtered. Based on this fitness function (genetic sense, not process sense), omitting infrequent behaviour is penalised less than omitting frequent behaviour.

18. *Diversity Guided Evolutionary Mining (DGEM)* [35], which is a method that discovers hierarchical process models. Its evolution strategy tries to maximise the coherence of log and model behaviour, i.e. models that better represent the event log are considered 'fitter'. No explicit difference is made between frequent and infrequent behaviour, but due to the inner workings of the method (namely, the genetic fitness function), similar to [34], omitting infrequent behaviour is penalised less than omitting frequent behaviour. Interestingly, the reported *trace fitness* measure is always equal to 1, though this cannot be generalised.

19. *ProDiGen* [36], which, contrasting to previously discussed genetic PD methods, **explicitly** filters infrequent behaviour: using user-settable frequency parameter ε, traces with frequency less than $\mu - \varepsilon$ are filtered out, where μ comes from the normal distribution of traces. By setting ε to a number such that $\mu - \varepsilon \leq 0$ it is possible to never filter behaviour. Another main difference between this method and the earlier discussed genetic ones, is that *ProDiGen* uses a fitness function that is hierarchical in nature: the algorithm focuses the search on *complete* (measure of completeness of a model) individuals, before caring about the next 'level' in the hierarchy of the fitness function.

3.1.3 Clustering

Often confused with classification, the clustering task in machine learning is one where given (unlabelled) data, one has to decide on how many classes (clusters) there are. Some well-known clustering algorithms are DBSCAN [37], OPTICS [38] and BIRCH [39]. In process discovery methods clustering can be used to cluster traces or events.

The investigated clustering PD methods:

20. *BPMN Miner* [40], which, given some log L, computes event type clusters and their hierarchy, projects the log over said hierarchy and then discovers models from the projected logs to obtain a hierarchical process model. The paper gives great thought to process-subprocess relations in the hierarchy. Even though it does not differentiate between frequent and infrequent behaviour (it only mentions ML noise), it filters traces after projection, that is, within a discovered subprocess, but only if there is "noise" in the data.

21. *Stage Miner* [41], which, after building a flow graph from the log, recursively decomposes (clusters) it into sets of nodes using the notion of min-cut as calculated by the Ford-Fulkerson algorithm [42]. The paper isn't clear on whether or not behaviour is filtered (either explicitly or implicitly).

22. *Decomposed Process Miner* [43], which works by discovering clusters and selecting best ones, then filtering the log to get sublogs, on which it discovers models (using some *other* PD method) that are then merged to return the overall process model. The way that this method handles behaviour is entirely reliant on how the chosen discovery method handles behaviour.

3.1.4 Integer Linear Programming

Note: Not to be confused with inductive logic programming, which has also been used for PD methods [20]. Linear programming, also referred to as linear optimisation, is a mathematical modelling technique in which a (set of) linear function(s) is optimised subject to a set of constraints. Solving a linear programming task is not trivial, but there are existing algorithms that can do it such as the Simplex method [44]. ILP requires all variables used in the (set of) linear function(s) to optimise to be integer. Whereas a solution to a linear programming problem can be computed in tractable fashion, ILP is NP-complete (this can be proven in many ways, examples include reductions from 3SAT to ILP or from ILP to vertex cover).

The investigated ILP PD methods:

23. *Aim* [45], which uses a Parikh representation [46] of the log to discover a guaranteed fitting PN. As such, no behaviour gets filtered, nor is there a distinction between frequent and infrequent behaviour.

24. *Proximity miner* [47], which transforms the PD problem into an ILP one by looking at a proximity measure based on the distance between events (both the proximity and distance measures are defined in the paper). The paper discusses the difference between frequent behaviour, ML noise, and infrequent behaviour (which it calls "positive noise"). There is a way (changing the ILP constraints) for a domain expert to indicate what infrequent behaviour is desired, and what should be filtered out.

25. *ILP Miner* [48], which is based on the theory of regions [46] (Parikh representation) to discover a PN. Note that it internally uses the Simplex method [44] to solve the constructed ILP formulation. It does not filter, nor differentiate between, behaviour.

26. *Hybrid ILP miner* [49], which too uses theory of regions (Parikh representation) to discover a PN. In contrast with the ILP Miner, the Hybrid ILP Miner has a filter threshold that can be set such that fitness is always 1. Internally it uses heuristics to enforce solutions describing frequent behaviour. As such, it does differentiates between infrequent and frequent behaviour.

3.1.5 Heuristics

Heuristic algorithms are those that, as the name implies, utilise heuristics to solve a problem *well* with considerable computational speedup as opposed to optimally and slow. A prominent field where heuristics are used is game AI: in chess, developing the knights to the centre is generally considered better than developing them to the sides of the board. By taking many of such "rules of thumb" and encoding them into a single scoring function to be optimised, one utilises heuristics. Interestingly, most of the investigated heuristic PD methods are based upon [50].

The investigated heuristic PD methods:

27. *HK miner* [51], which takes a target parameter x setting a desired target "accuracy" (metric defined in the paper). The method uses a best-first tree search approach, where "best" is heuristically defined. Infrequent behaviour is considered different from frequent behaviour, and gets filtered out during execution by design.

28. *Inductive Miner - Infrequent* [52], which constructs the directly follows graph (based on directly follows - see the explanation of the α-algorithm in Section 3.1.1) and attempts to find structures in it. Then, it splits the log based on those structures, and recurses on each sublog. By default it discovers an "80/20" model — 80% of the observed behaviour can be explained by a model that is only 20% of the model required to describe all behaviour" — but this can be changed as desired. There is *local* filtering of subtraces (or even individual events) that occurs only when no model constructs can be found. As such, filtering behaviour (be it frequent or infrequent) is dependent on whether or not model constructs can be found in the log, and the set parameters that control how fitting the model should be.

29. *Heuristic Miner* [53] (building on the ideas of [50] that uses heuristics based on directly follows relations, and thus differentiates between frequent or infrequent behaviour), which uses structuring, soundness repair, and clone removal techniques to structure the output of the method it is based on. Since the structuring step of [53] rests on refined process structure trees (RPST) which cannot handle cycles (regardless of length), it is clear that behaviour (regardless of frequency) that includes cycles is filtered. There is, however, no distinction between frequent and infrequent behaviour in either method; only between ML noise and data.

30. *Fusion miner* [54], which combines a heuristic approach [50] and the earlier mentioned Declare miner, and heavily utilises dependency nets [16]. The method does not particularly care about types of behaviour. It has input parameter e for pruning *entropic* behaviour — behaviour that contains high entropy: a measure of the amount of information present — which can be set such that no behaviour is filtered.

31. *Fodina* [55], which similar to [53] too builds upon [50], with only different heuristics to avoid certain types of deadlocks [2]. It does not differentiate between types of behaviour, and filtering occurs due to the working of the algorithm ([50] cannot handle L1L or L2L constructs), that is, the used heuristics.

32. *Split Miner* [56], which is similar to both *Fodina* [55] and [50] in that it utilises directly follows graphs. In contrast to previously mentioned methods, the *Split Miner* prunes the directly follows graph in an attempt to deal with L1L and L2L constructs and concurrency. Since the method uses the directly follows relations, it differentiates between frequent and infrequent behaviour. It does not guarantee a perfect fitting model: filtering of behaviour may occur due to the used heuristics.

3.1.6 Machine learning

Usually associated with neural networks, ML is a computational paradigm where one gives the computer a problem to solve, without code to solve it. The idea is that the machine learns how to solve it, hence the name machine learning. The most common example when thinking about ML is classification of the MNIST [57] images: given an image of size 28×28 of a handwritten digit, output what digit it is. Using ML in process mining is relatively new. As such, we only look at the very elaborate proof of concept by Sommers [3].

The investigated ML PD method:

33. The GCN based approach [3], which models the log L as a graph, effectively turning the PD problem into a node classification one. While the thesis has given considerable thought on the difference between frequent and infrequent behaviour, the approach itself — or rather proof of concept — does not differentiate between them. Behaviour is filtered due to the approach, but this is not a design decision.

3.2 Overview

There are a plethora of different process discovery methods, of which we investigated 33. Some are declarative in nature [16, 17, 18, 20, 30, 33], some are procedural [3, 19, 22, 23, 25, 26, 27, 28, 29, 34, 35, 36, 40, 41, 45, 47, 49, 51, 52, 53, 55, 56] — a PN is an example of a procedural model — and some pick the best of both worlds and mix them [21, 43, 54]. Only *one* [26] of the discussed methods uses the underlying (empirical) probability distribution. When it comes to differentiating between infrequent and frequent behaviour and filtering of said behaviour, we distinguish following categories.

No distinction between frequent and infrequent behaviour

These are the methods that do not differentiate between frequent and infrequent behaviour. We categorise these as follows.

1. No filtering [1, 22, 28, 29, 45, 48]
2. Possibly optional *conscious* filtering, either by means of user input (usually a parameter)
    (a) Filtering occurs only on *activity level* [20]
    (b) Filtering occurs only on *model constructs* [16, 17, 18, 30]
3. Possible *non-conscious* filtering, due to the inner workings of the algorithm [23, 33, 40, 41]

Distinction between frequent and infrequent behaviour

These are the methods that *do* differentiate between frequent and infrequent behaviour. As expected, when methods differentiate there is always *some form* of filtering. We categorise these as follows.

1. Optional *conscious* filtering, either by means of user input (usually a parameter)
    (a) Filtering occurs only on *trace level* [25, 36, 47, 49]
2. *Conscious* filtering, due to the inner workings of the algorithm [34, 35, 51, 52, 53, 55]

Other / Not applicable

1. Depends on underlying PD methods [21, 43]
2. Filtering only affects the visual representation of the model, not the PD method [19, 26, 27]

4 Probabilistic Models - An Overview

This section gives an overview of various probabilistic models. Since there is no canonical categorisation of probabilistic models we use our own categorisation based on the representation and related interpretation of the models. In particular, we focus on whether or not it is represented as a probabilistic graphical model (PGM), or as a non-graphical model, while discussing expressiveness and tractability for exact inference (when we write 'inference' in this section, we always imply exact inference). For an overview of some probabilistic models coloured based on tractability see Figure 8. Note that not all of them will be covered in this literature study. Furthermore, note that *all* discussed models are a specific way to represent some probability distribution: there is no "extraction" of a distribution; the models are used directly to perform inference.

4.1 Graphical Models

In a PGM, each node represents an RV and each edge represents a conditional dependency between its connected nodes. Since an RV is not dependent on itself (or rather, explicit self-dependence makes no sense), self-loops are disallowed. As such, the graph types we consider as basis in this section are Polyforests (DAGs) for the directed case (Section 4.1.1) and undirected graphs without self-loops for the undirected case (Section 4.1.2). *Tractability discussions all come from [8].*

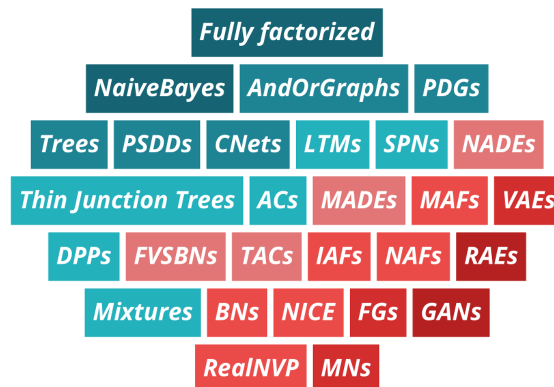

Figure 8: Some probabilistic models, coloured on tractability [8]. Models with blue shades are more tractable and models with red shades are less tractable.

4.1.1 Directed: Bayesian Networks

Based on DAGs, a Bayesian network (BN) graphically represents a joint probability as shown in Equation 6. Formally, it is a pair $(G, \mathcal{P})$, where G is a DAG with its nodes representing $X \in \mathbf{X}$ and $\mathcal{P}$ a collection of conditional probability distributions (CPD)s $p(X \mid \mathbf{pa}(X))$, where $\mathbf{pa}(X)$ is the set of all parents of X. For a more thorough introduction to BNs we refer to [58]. Exact inference in BNs is commonly done by applying the variable elimination algorithm [59]. There are also various algorithms for approximate inference such as stochastic Markov chain Monte Carlo simulation [60] and message passing [61], also known as (generalised) belief propagation. Note that a particular type of a BN is a **causal network**, where edges $u \to v$ can be interpreted as u having caused v. Causal networks may be interesting for modelling processes due to their causality.

$$p(\mathbf{X}) = \prod_{X \in \mathbf{X}} p(X|\mathbf{pa}(X)) \tag{6}$$

There are *many* applications of BNs, as they effectively are just a representation of some joint distribution combined with conditional dependencies coming from assumptions or prior knowledge. To illustrate, BNs can be used for diagnostics, for instance diagnosing types of diseases from medical data [62]. They can also be used for predictive tasks such as prediction of possible forest fire causes [63].

By adding restrictions to the structure of G for a particular BN we get different classes of models. In particular, if we restrict G to be a (rooted) directed tree such that for each node v there is at most one parent then we have a *tree-structured BN* [64] which is commonly called a Chow-Liu tree (CLT) [65], after the Chow-Liu learning algorithm (CLT algorithm) for tree-structured BNs. A CLT is effectively a second-order product approximation of the joint distribution, where second-order means that one RV is dependent on at most one other RV, hence the restriction that nodes are only allowed to have at most one parent.

We can relax the CLT definition by allowing each node v to have up to $k-1$ parents, giving rise to the kth-order *t-cherry Junction tree* [66], based on Junction trees [67]. A Junction tree is also known as a tree decomposition, or a clique tree. Intuitively, a Junction tree represents vertices of a particular graph G as subtrees of a tree, as illustrated in Figure 9. For the t-cherry Junction trees, if $k = 2$, that is, the second-order t-cherry Junction tree, we precisely have a CLT.

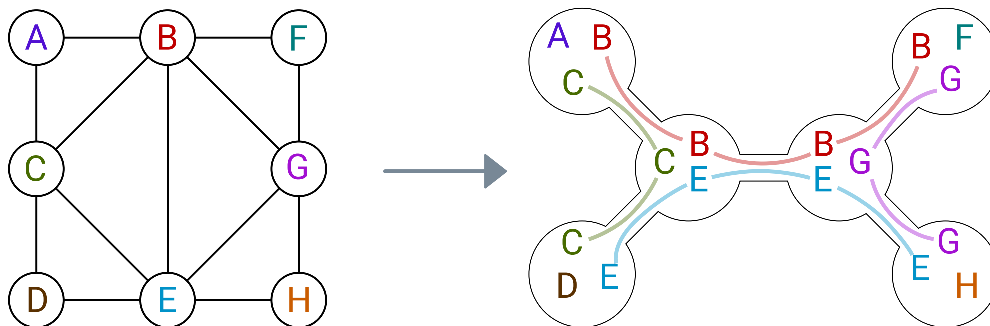

Figure 9: Junction tree structure from a graph.

A second way to relax the definition of CLTs is by allowing for multiple roots, giving rise to a non-tree, multiple parent structured BN. Note that this is identical to saying that we restrict G to be a Polytree as opposed to a Polyforest. Polytree-structured BNs are more expressive than CLTs, but pay for their expressiveness with the removal of tractable inference for some queries [68].

Another type of BNs is the thin Junction tree (TJT) [69], also based on Junction trees. A TJT is a Junction tree of limited *treewidth*: the *width* of some decomposition is the size of its largest set $\mathbf{X}_i$ minus 1. Informally it is an integer number specifying how far a particular graph is from being a tree (a tree has treewidth equal to 1). For the example TJT shown in Figure 10, the size of its largest set is 3 ($\{X_1, X_3, X_4\}$), hence its treewidth is 2. Notice how the treewidth of a CLT is precisely 1, as it is a tree. One can intuitively see that, since treewidth is no longer limited to 1, a TJT is more expressive than a CLT.

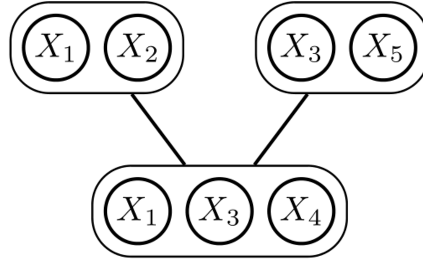

Figure 10: Example Thin Junction tree [8].

Finally, we can consider so-called *fully factorised* models, where there are only nodes and no edges. An example of a fully factorised is a Product of Bernoulli's (PoBs). Naturally, these can do tractable inference (specifically inference for **EVI**, **MAR**, **MAP** and **MMAP** queries is linear in model size), but they are rather inexpressive as they cannot model any form of dependency [8].

4.1.2 Undirected: Markov Networks

The counterpart of a BN is a Markov network (MN), as these are built upon undirected graphs as opposed to directed ones. An important difference, as clear from the introductory paragraph of Section 4.1, is that MNs *do* allow cycles in their underlying graph structure. Formally, a MN is a pair of an undirected graph G (without self-loops) and a set of *positive factors* $\mathcal{F}$: $(G, \mathcal{F})$. The set of factors is constructed as $\mathcal{F} := \{f(\mathbf{C})\}_{\mathbf{C} \in \mathcal{C}}$, where $\mathcal{C}$ is the set of maximal cliques (see Section 2.3) in G (and thus, consequently, $\mathbf{C}$ is any maximal clique in G). For a proper introduction to factors (including a rigorous definition on what $f(\mathbf{C})$ means) see [59]. In short: a positive factor f is a positive real number defined on a graph-structure (such as a clique) that *may*, but does not need to, be interpretable as a probability. Factors f_1 and f_2 can be multiplied (*multiplying out*), as well as summed (*summing out*). These particular operations on *factor graphs* are necessary for inference. The joint distribution of a MN is defined as shown in Equation 7, where $\mathcal{Z}$ is a normalisation constant. Computing it is the hardest marginalisation task there is.

$$p(\mathbf{X}) = \frac{1}{\mathcal{Z}} \prod_{\mathbf{C} \in \mathcal{C}} f(\mathbf{C}) \tag{7}$$

MNs, just like BNs, can be used for *many* things. Examples range from using MNs as texture models [70] — a texture model is a mathematical procedure capable of producing and describing a textured image — to computer vision [71] and computational biology [72]. Arguably the most well-known MN is the Ising model, a mathematical model of ferromagnetism in statistical mechanics, which can be used for various applications, including denoising images [73].

Similar to BNs one can impose structural constraints on MNs, but these do not make a MN support more types of queries in tractable fashion. They are also (considerably) less widely used than their directed counterparts. Note that the *fully factorised* models from Section 4.1.1 can also

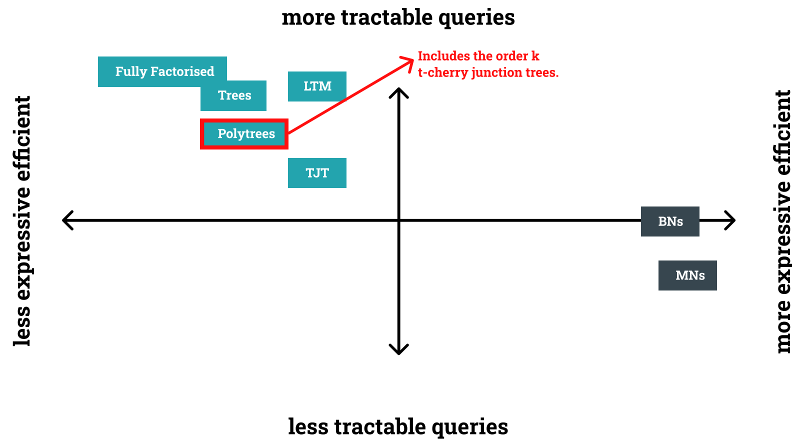

Figure 11: Tractability spectrum for PGMs, adapted from [8].

be interpreted as being MNs due to the lack of (directed or undirected) edges. Indeed, values from $p \in \mathcal{P}$ in fully factorised models are positive integers, hence interpretable as factors.

4.1.3 Intermezzo: Graphical Latent Variable Models

A *latent variable* is an RV that is *not directly observed*, but rather inferred from other *observed* variables. A simple example from psychology is the latent variable named *IQ* that is estimated using an IQ-test with various questions (that are all observed variables). Usually, latent variables are denoted with Z. A particularly interesting class of (graphical) latent variable models is a latent tree model (LTM). These LTMs can be directed (BNs) or undirected (MNs) and are defined as a Polyforest and Forest respectively (see Section 2.3). In either case, an LTM models observed variables as leaf nodes and latent variables as internal nodes (all nodes that are not root or leaf).

A latent tree model is mathematically, in terms of how it defines its probabilities, not different from the definitions as given in Section 4.1.1 for the directed case and Section 4.1.2 for the undirected case. The *interpretation* is what differs, which may allow for better modelling constructs. For applications of LTMs we refer you to the survey done by Mourad et al. [74], which investigates their usage in detail.

4.1.4 Tractability of graphical models

To conclude tractability for PGMs, Figure 11 shows a summarisation of their tractability and expressiveness (expressiveness is a shorthand for how many types of probability distributions can be modelled using a particular model). Clearly, fully factorised models are least expressive, as they do not allow for *any* dependency between RVs. This does mean, however, that they allow for tractable inference of more query types. In fact, each of the 6 types of queries listed in Section 2.2.6 can be inferred tractably using fully factorised models. CLTs and their undirected counterparts can tractably infer **EVI**, **MAR**, **CON**, **MAP** and **MMAP** queries. Polytrees (and the kth order t-cherry Junction trees), and their undirected counterparts, can no longer tractably do **MMAP** queries (but all "easier" queries are still possible in tractable fashion). For TJTs **MAP** is no longer possible to compute tractably. Moreover, if the treewidth bound is "too high" (≈ 20) then **MAR** and **CON** queries are no longer feasible in practise [8]. Finally, one can see that BNs in general can compute more queries in tractable fashion than MNs. We hypothesise this is likely due to the acyclic nature of BNs, but further research is necessary to confirm (or deny)this hypothesis.

4.2 Non-Graphical Models

Whereas graphical models have a clear graphical representation (that has underlying semantics), non-graphical models lack such representation. These models are generally only defined in terms of their distribution $p(\mathbf{x})$. Any "graphical" representation that is shown in this section is simply an illustration of how the models works. Most of these non-graphical models are based on neural networks.

4.2.1 Distribution Estimation (NADE)

The distribution estimation task attempts to produce an estimator of $p(\mathbf{x})$ based on samples from it. A solution to solving the distribution estimation task is to give all samples to a particular neural network, and then (in some way) tell it learn the estimator. This is precisely what ML-based distribution estimators are. We explain the neural autoregressive distribution estimation (NADE) model for binary observations [75].

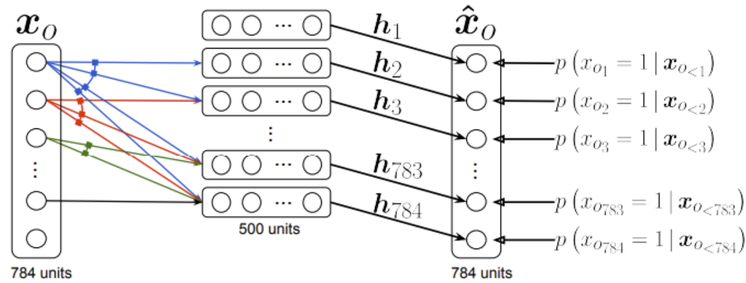

Figure 12: Binary NADE, from [75].

The NADE model for binary observations is a neural network with precisely one hidden layer. It is based on the fact that any D-dimensional distribution can be factored into a product of one-dimensional distributions of some order o, which is a permutation of integers $1, \dots, D$, illustrated in Equation 8. Here, o_d represents a particular dimension d according to ordering o, and x_{o_d} is a the value of said dimension of $\mathbf{x}$. Similarly, $o_{<d}$ represents a vector of length $d-1$, ordered according to o, and $\mathbf{x}_{o_{<d}}$ is the corresponding subvector from $\mathbf{x}$ for said dimensions.

$$p(\mathbf{x}) = \prod_{d=1}^{D} p\left(x_{o_d} \mid \mathbf{x}_{o_{<d}}\right) \tag{8}$$

NADE parametrises each conditional $p(x_{o_d} \mid \mathbf{x}_{o_{<d}})$ according to Equations 9 and 10. The parameters of NADE are — with D the size of the input and H the number of hidden units — input matrix $\mathbf{V} \in \mathbb{R}^{D \times H}$ with biases $\mathbf{b} \in \mathbb{R}^{D}$ and hidden layer matrix $\mathbf{W} \in \mathbb{R}^{H \times D}$ with shared biases $\mathbf{c} \in \mathbb{R}^{H}$. An illustration of NADE is shown in Figure 12.

$$p(x_{o_d} = 1 \mid \mathbf{x}_{o_{<d}}) = \operatorname{sigm}\left(\mathbf{V}_{o_d,\cdot}\mathbf{h}_d + b_{o_d}\right) \tag{9}$$

$$\mathbf{h}_d = \operatorname{sigm}\left(\mathbf{W}_{\cdot,o_{<d}}\mathbf{x}_{o_{<d}} + \mathbf{c}\right) \tag{10}$$

Similar models exists for real-valued observations (real-valued neural autoregressive distribution estimator (RNADE) [76]), multinomial observations (DocNADE [77]) and even a deep variant that uses more than 1 hidden layer (DeepNADE [78]). The family of NADE models are instances that solve the problem of unsupervised distribution and density estimation. NADE has been used for topic modelling [79], sequential music modelling [80], and has even been applied to a (pre-existing) image classifier integrating an attention mechanism [81].

4.2.2 Variational Auto Encoder (VAE)

As can be inferred from the name, a variational auto encoder (VAE) is an autoencoder (a neural network trained to copy its input to its output [82]), where the latent space is interpreted as a probability distribution. Figure 13 shows the VAE architecture: similar to an autoencoder there are decoder and encoder parts. The encoder encodes the input (samples) to latent space. The decoder decodes latent space to some distribution, which can then be sampled to get a new sample. As such, VAEs also estimate the original distribution $p(\mathbf{x})$ based on samples [83]. Since the latent space is interpreted as a probability distribution, one can consider many VAE variants [84], including a Categorical VAE and Beta VAE.

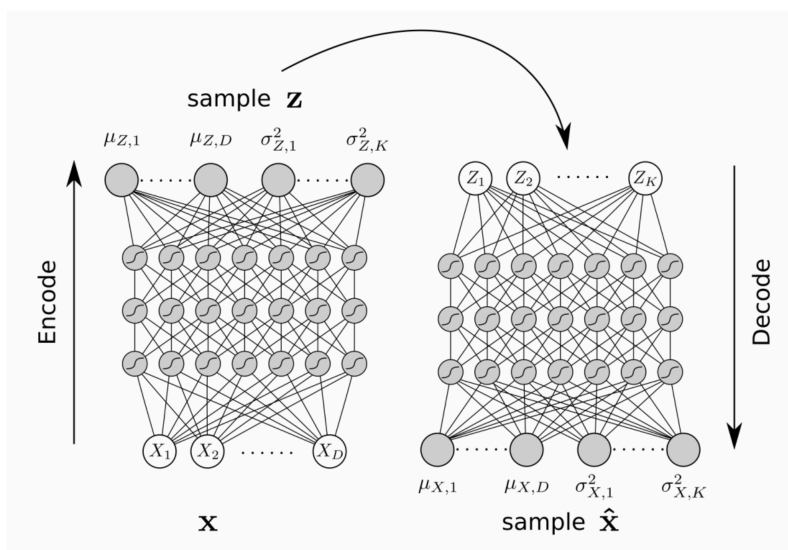

Figure 13: Architecture of a VAE with Gaussian latent space, taken from [85].

Example uses of VAEs range from oneshot learning [86], to regression [87] and classification and anomaly detection of videos [88].

4.2.3 Generative Adversarial Network (GAN)

A generative adversarial model (GAN) is a model based on *adversarial training*; see [89] for an introduction. In short GANs work by using two neural networks, one being a generator (the probabilistic part) and one being the discriminator. The task of the generator G is to generate a sample from random noise, and the discriminator D needs to classify samples as either being generated by G, or coming from the 'real' dataset. The architecture is illustrated in Figure 14.

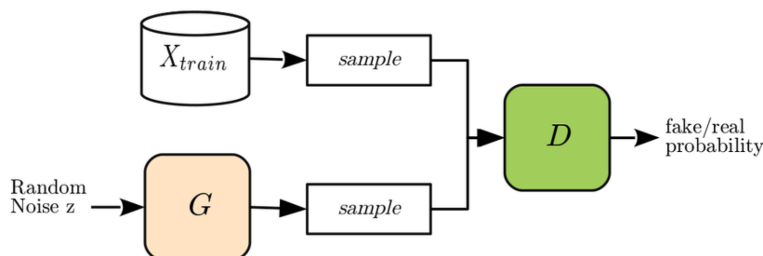

Figure 14: Architecture of a GAN, taken from [90].

GANs are commonly used to generate new things. A very well-known example is StyleGAN [91] that generates high-quality images of people that do not exist. By visiting this page, one can see examples of generated images (refresh for new images). For a structured review on applications of GANs see [92].

4.2.4 Normalising Flow

Normalising flows are commonly called *flows*. They are a probabilistic model that rests upon the change of variable theorem and invertible functions. Let $p_{\mathbf{U}}$ be some base distribution (the choice does not matter as flows are universal approximators [93]). Let $f : \mathbb{R}^D \to \mathbb{R}^D$ be some invertible function, and assume that both f and its inverse f^{-1} are differentiable. Let $\mathbf{X}$ and $\mathbf{Y}$ be two random vectors with $\mathbf{Y} = f(\mathbf{X})$ (consequently also $\mathbf{X} = f^{-1}(\mathbf{Y})$). The change of variable theorem can be found in Equations 11 and 12, where J is a Jacobian matrix.

$$p_{\mathbf{Y}}(\mathbf{y}) = p_{\mathbf{X}}(\mathbf{x}) \left|\det J_f(\mathbf{x})\right|^{-1} = p_{\mathbf{X}}(\mathbf{x}) \left|\det J_{f^{-1}}(\mathbf{y})\right|, \qquad \mathbf{x} = f^{-1}(\mathbf{y}) \tag{11}$$

$$p_{\mathbf{X}}(\mathbf{x}) = p_{\mathbf{Y}}(\mathbf{y}) \left|\det J_{f^{-1}}(\mathbf{y})\right|^{-1} = p_{\mathbf{Y}}(\mathbf{y}) \left|\det J_f(\mathbf{x})\right|, \qquad \mathbf{y} = f^{-1}(\mathbf{x}) \tag{12}$$

The modelled density ($p_{\mathbf{X}}(\mathbf{x})$) is entirely specified by the inverse function f^{-1}, as shown in Equation 13. For sampling it is required that $(f^{-1})^{-1} \equiv f$, then: $f(\mathbf{u}) = \mathbf{x}$ with $\mathbf{u} \sim p_{\mathbf{U}}$.

$$p_{\mathbf{X}}(\mathbf{x}) = p_{\mathbf{U}}\left(f_\Theta^{-1}(\mathbf{x})\right) \times \left|\det J_{f_\Theta^{-1}}(\mathbf{x})\right| \tag{13}$$

Examples of neural flows (flows based on neural networks) are RealNVP [94], masked autoregressive flow (MAF) [95], masked autoencoder for distribution estimation (MADE) [96] and PixelRNN [97]. Example usecases range from anomaly detection on audio data [98] to image completion [97].

4.2.5 Mixture Models

A Mixture model, or simply *mixture*, is a convex combination of k *simpler* models. Mixture models are used the same way any model that models the underlying probability distribution is used: numerical flow simulation [99] and discovering motifs in bipolymers [100] are two examples.

4.2.6 Tractability of non-graphical models

From worst to best in terms of supporting queries tractably, GANs do not have an explicit model of likelihood and thus all queries are intractable. Second, VAEs have an explicit likelihood model, but computing the probability distribution (of the latent space) is still intractable: it is an infinite and uncountable mixture, resulting in intractable **EVI** queries. Thirdly, flows which also have an explicit likelihood model (and structured Jacobians), giving rise to tractable **EVI** queries. If f is a 'simple' function such as a bijection then **MAR** queries are also tractable. In the general case, f is complicated, and thus **MAR** queries are intractable due to integration. Fourth, the NADE models are similar to flows, but do not have a restriction on some function f and have tractalbe **EVI** and **MAR** queries. Finally, mixtures can be very expressive [101]. The **EVI**, **MAR** and **CON** queries scale linearly in k, and are tractable if all of the k models can do the queries tractably. Since **MAP** is intractable for any latent variable model — LTMs and VAEs are examples of latent variable models — it is also intractable for mixtures as they marginalise a categorical latent variable Z with k values.

4.3 Probabilistic Circuits

An overview of various graphical and non-graphical models can be found in Sections 4.1 and 4.2 respectively, where we illustrated the expressiveness-tractability issue. In this Section we introduce the probabilistic circuit (PC) family of both expressive, and tractable models (given some structural constraints, different for each "flavour" of PC). It is important to note that PCs

are *not* PGMs, even though we represent them graphically. In PGMs nodes represent RVs, edges represent dependencies, and inference is done using conditioning, elimination or message passing algorithms. In PCs nodes represent units of computation, edges represent order of execution, and inference is done using feedforward and backward passes. PCs are computational graphs, similar to neural networks: a PC $\mathcal{C}$ over variables $\mathbf{X}$ is a computational graph encoding a (possibly unnormalized) probability distribution $p(\mathbf{X})$. *All PCs are representations of some underlying probability distribution, and can thus be used for any generative task.*

In this Section we will mostly focus on a particular type of PC, namely the Sum-Product Network (SPN) [102], to explain particular notions and ideas that are identical (or very similar) for all “flavours“ in the PC family. There are two types of basic nodes: a distribution node and a logical node. The first, that is, a distribution node, is a node encoding (for instance) a Gaussian PDF for some continuous RV. Distribution nodes are denoted by bell curves inside nodes. The second, that is, a logical node, is one that encodes some sort of logical constraint such as $X_1 > \theta$ (for a real-valued X_1) or $\neg X_2$ (for a boolean X_2). These basic nodes can be combined in two ways: by multiplying them, or summing them. For multiplication there is the product node, which establishes factorisation. This is depicted in Figure 15, showing how three RVs are the factors of some multiplication. Summation is provided by the sum nodes which allow PCs to easily model mixtures of various distributions, as depicted in Figure 16.

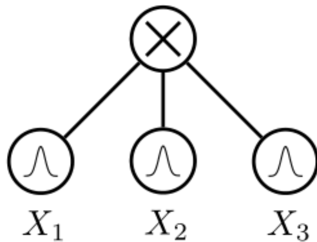

Figure 15: A product node, from [8].

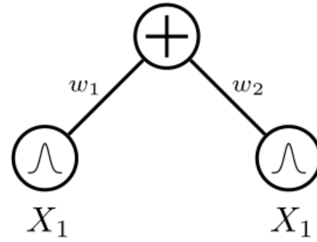

Figure 16: A sum node, from [8].

4.3.1 Structural Constraints

To allow tractable inference, structural constraints on the model are required. For SPNs these structural constraints follow from properties that product nodes and sum nodes can have, as enumerated below. For other members of the PC family, similar and sometimes identical definitions exist.

Decomposability A product node is decomposable if its children depend on disjoint sets of variables. When all product nodes in a PC are decomposable, then we call the circuit decomposable.

Smoothness Also known as completeness; a sum node is smooth if its children depend on the same variable sets. Smoothness can be easily enforced [103]. When all sum nodes in a PC are smooth, then we call the circuit smooth.

Determinism Also known as selectivity; a sum node is deterministic if the output of only one of its children nodes is non-zero for any input. Determinism is illustrated in Figure 17. When all sum nodes in a PC are deterministic, then we call the circuit deterministic.

Structured decomposability A product node is structured decomposable if it decomposes into a *vtree*: a full binary tree on variables in S whose leaves have a one-to-one correspondence with the variables in S. When all product nodes in a PC are structured decomposable, then we call the circuit structured decomposable.

Consistency A product node is consistent if any variable shared between its children appears in a single leaf node. If a product node is decomposable, it is consistent. When all product nodes in a PC are consistent, then we call the circuit consistent.

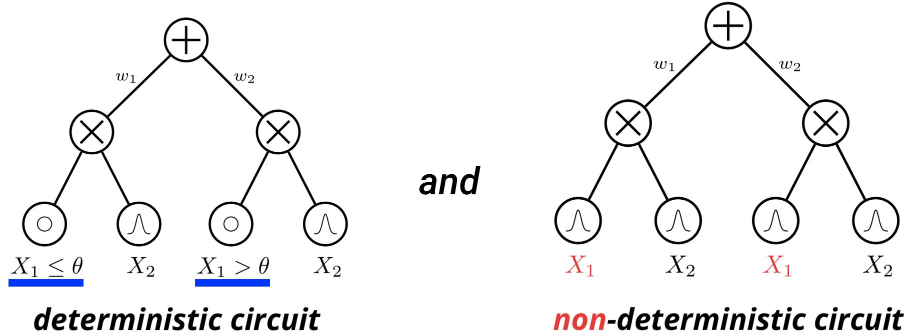

Figure 17: Deterministic and non-deterministic circuits, after images from [8]. The blue line indicates what makes the left circuit deterministic.

If a circuit is decomposable and smooth, then it tractably supports answering **MAR** and **CON** queries linear in circuit size [104]. If a circuit is decomposable and deterministic (an equivalent condition is for the circuit to be deterministic and consistent), then **MAP** queries are tractable linear in circuit size. Finally, if a circuit is structured decomposable, then all queries are tractable linear in circuit size.

4.3.2 Tractability of various PCs

The SPN [102], which is decomposable and smooth and, therefore, tractably supports **EVI**, **MAR** and **CON** queries. Even though SPNs are generally *not* deterministic, it is possible to learn deterministic SPNs [105], which we then call *selective*. Selective SPNs are deterministic, and thus tractably support **MAP** queries. The building blocks of an SPN have already been introduced, and are shown in Figures 15 and 16. SPNs can be used for image generation or image completion [106].

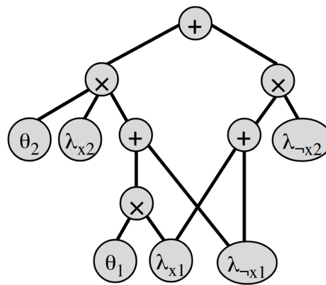

Figure 18: Example arithmetic circuit (AC), from [8].

Another PC is the arithmetic circuit (AC) [107], which is decomposable, smooth and deterministic. As such, ACs tractably support **EVI**, **MAR**, **CON**, and **MAP** queries. ACs also utilise sum (Figure 16) and product (Figure 15) nodes as building blocks. The leaves of an AC contain model parameters: in Figure 18 one can see lambdas $\lambda_{x1}, \lambda_{\neg x1}, \lambda_{x2}$, and $\lambda_{\neg x2}$ which are *evidence indicators* — for some boolean RV X_1 the evidence indicator λ_{x1} specifies that $X_1 = \text{true}$ — and the thetas θ_1, and θ_2, which are probabilities normally found in a probability table. ACs can be used for any generative task such as data imputation.

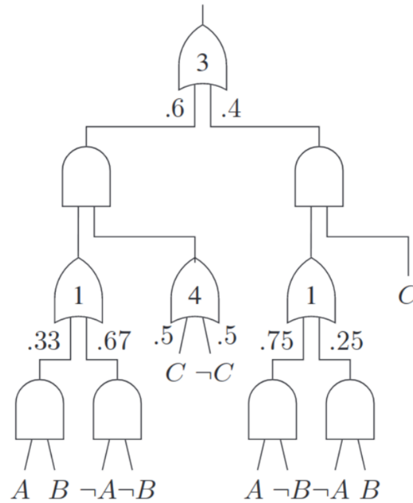

Figure 20: An example PSDD, from [8].

A cutset network (CNet) [108] is the third PC we discuss. Similar to ACs, CNets are decomposable, smooth and deterministic. Therefore, CNets tractably support **EVI**, **MAR**, **CON**, and **MAP** queries. Structurally they are rooted OR trees with CLTs as leaves, of which an example is shown in Figure 19. Each OR node (these are sum nodes) represents conditioning over a particular RV. The reasoning behind CNets is that we desire a simple, tractable and scalable approach to improving the accuracy of CLTs, which are relatively bad approximations. The usecases of CNets are identical to those of CLTs or any other model that learns the underlying probability distribution.

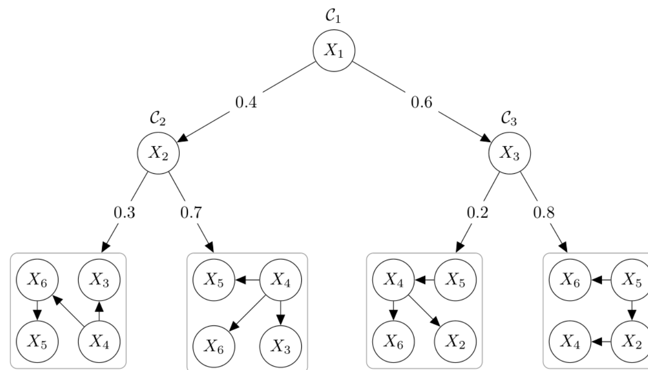

Figure 19: An example CNet, from [8].

A probabilistic sentential decision diagram (PSDD) [109] is the first PC we discuss that supports *all* queries in tractable fashion: PSDDs are structured decomposable, smooth and deterministic. An example PSDD is shown in Figure 20 where one can see that "AND" (product) nodes and "OR" (sum) nodes are shown with their logical circuit notation. The numbers illustrate how logical decision nodes are transformed into probabilistic ones by supplying a distribution over the branches. Note that PSDDs are defined over boolean RVs, but by means of one-hot encoding or binning *any* RV can be converted so that PSDDs can be used. The usecase of a PSDD is any generative task.

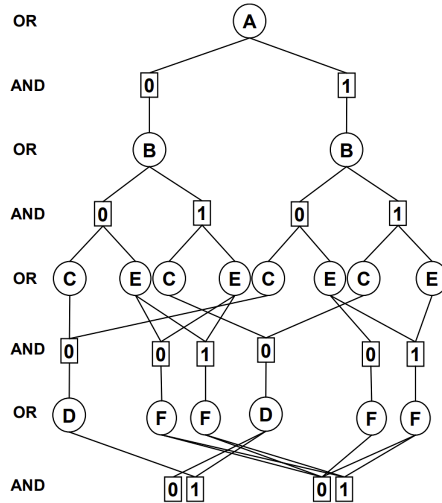

Figure 22: An example AoG, from [113].

$$p(X) = \prod_{i=1}^{n} p(X_i \mid \mathscr{A}_i) \tag{14}$$

Another PC is the probabilistic decision graph (PDG) [110]. These are structured decomposable, smooth and deterministic, and as such support all queries in tractable fashion. Figure 21 shows an example PDG. Lowercase v enumerates all nodes in the model. Uppercase V_i denotes the set of all nodes v labelled with RV X_i. A PDG specifies a joint by the factorisation shown in Equation 14, where $\mathscr{A}_i$ is a partition of the product set of all possible solutions of each X_i [111]. While this may look similar to BNs, the independence structures captured by each respective model is different. Furthermore, PDGs can do inference in tractable fashion whereas this is generally not the case for BNs.

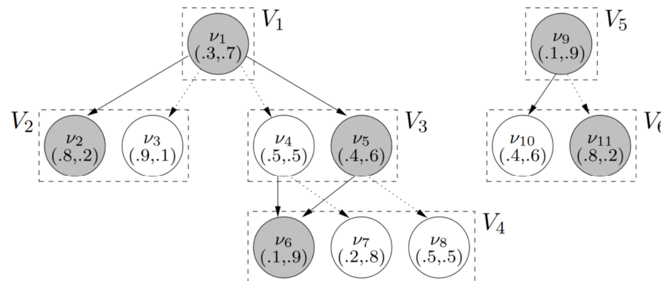

Figure 21: An example PDG, from [8].

To finish up the discussion on different PCs we show the AndOrGraph (AoG) [112]. Just like PSDDs and PDGs these too are structured decomposable, smooth and deterministic, and thus also support all queries in tractable fashion (linear in model size). An example AoG is shown in Figure 22, where "AND" nodes (product nodes) are square and "OR" nodes (sum nodes) are circles. This particular example is an and-or-tree for solving 0/1 ILP [113].

4.4 Overview

As illustrated, there are *many* different types of probabilistic models, some more tractable than others. Models can be based on a graphical representation rooting in graph theory, where they can be directed or undirected, or they can be based on deep learning. Figure 23 shows an overview of all covered probabilistic models in terms of tractability and expressiveness. Turquoise models support more tractable queries, black models are more expressive, and purple models are types of PCs.

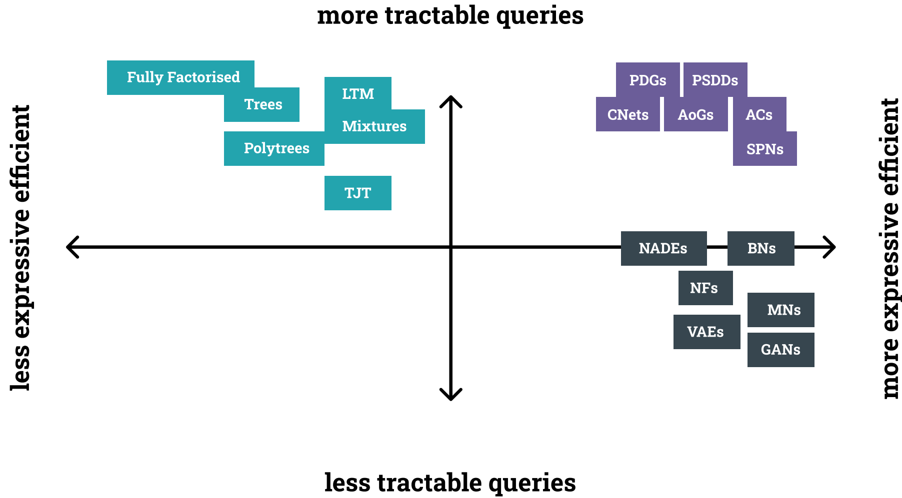

Figure 23: Tractability spectrum. Turquoise models support more tractable queries. Black models are more expressive. Purple models are types of PCs.

5 Combining Probabilistic Models and Process Mining

Sections 3 and 4 present the respective literature studies on infrequent behaviour and probabilistic models. This Section is an attempt to bridge the gap between both fields, and identify where possibly interesting research lays. First, we present a method to transform a log L into a probabilistic domain, and illustrate how such probability distribution may then be used, and how it should be evaluated. Second, we show the similarities between discovering a process model from data, and learning a probabilistic model from data.

5.1 From Event Data to Random Variables

Event data is stored in logs (Section 2.1.1) denoted by L. Table 1 shows three cases from a real-life event log of sepsis — a life threatening condition typically caused by an infection — cases from a hospital [114]. Each row contains a case identifier, an activity name, a timestamp, and finally an integer representing the activity. Existing data attributes and XES extensions have been removed to keep the example simple.

An initial idea is to use a categorical random vector $\mathbf{X} = (X_1, \ldots, X_n)$ with categorical RVs for each index in the trace and n being the length of the longest recorded trace. RV X_i then denotes a categorical probability distribution over $\mathcal{A} \cup \perp$, where $\mathcal{A}$ denotes the set of all activities and $\perp$ denotes "no activity". This idea assumes that the log is complete, and that the set of possible activities is known beforehand. Under the false assumption for the purpose of explanation that the log as shown in Table 1 is indeed complete, there are a total of three cases, identified by E, M and R respectively. The longest trace is E with a total of 8 activities. Let $\mathbf{X} = (X_1, \ldots, X_8)$ be a random vector of 8 categorical RVs, one per index of the trace. The sample space Ω of each X_i is

Case	Activity	Timestamp	Converted Activities
E	ER Registration	2015-01-15 20:14:58	1
E	ER Triage	2015-01-15 20:16:03	2
E	ER Sepsis Triage	2015-01-15 20:17:15	3
E	IV Liquid	2015-01-15 20:30:32	4
E	CRP	2015-01-15 20:31:00	5
E	Leucocytes	2015-01-15 20:31:00	6
E	LacticAcid	2015-01-15 20:31:00	7
E	IV Antibiotics	2015-01-15 20:35:59	8
M	ER Registration	2014-10-10 03:08:37	1
M	ER Triage	2014-10-10 03:10:38	2
M	ER Sepsis Triage	2014-10-10 03:10:54	3
R	ER Registration	2014-11-30 12:38:16	1
R	ER Triage	2014-11-30 12:40:09	2
R	ER Sepsis Triage	2014-11-30 12:40:29	3
R	CRP	2014-11-30 12:45:00	5
R	Leucocytes	2014-11-30 12:45:00	6

Table 1: Part of a real-life event log on sepsis patients [114].

$\{\texttt{ER Registration}, \texttt{ER Triage}, \texttt{ER Sepsis Triage}, \texttt{IV Liquid}, \texttt{CRP}, \texttt{Leucocytes}, \texttt{LacticAcid}, \texttt{IV Antibiotics}, \bot\}$. Giving probabilities to each item $\omega \in \Omega$ is trivially done according to the definition of a probability measure (Equation 2) by means of counting. For instance, $p(X_1)$ is shown in Equation 15. Not all probability distributions are as trivial. Equation 16 shows that $p(X_4)$ is considerably more involved (within context of the example log, of course).

$$p(X_1) = \begin{cases} 1 & \text{if } X_1 = \texttt{ER Registration} \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

$$p(X_4) = \begin{cases} \frac{1}{3} & \text{if } X_4 = \texttt{IV Liquid} \\ \frac{1}{3} & \text{if } X_4 = \bot \\ \frac{1}{3} & \text{if } X_4 = \texttt{CRP} \end{cases} \tag{16}$$

If desired, one can apply Laplace smoothing [115], alternatively called additive smoothing, which effectively replaces true counts with pseudo-counts. Laplace smoothing is useful to avoid dividing by zero issues when working with the empirical distribution.

Naturally, using complete activity names makes notation cumbersome. It makes sense to use shorter identifiers in stead. For instance, if we map each event to a unique integer identifier, starting at 1 and counting upwards, we get the fourth column of Table 1. Case E reduces to the trace $\langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle$, case M to $\langle 1, 2, 3 \rangle$ and case R to $\langle 1, 2, 3, 5, 6 \rangle$. These representations are much easier to implement in code. Furthermore, traces can now be interpreted as simple integer sequences. Similarly, any integer sequence may be interpreted as a converted log.

5.2 Method

An overview of the proposed method is presented in Figure 24. Following sections explain each step in sequential order.

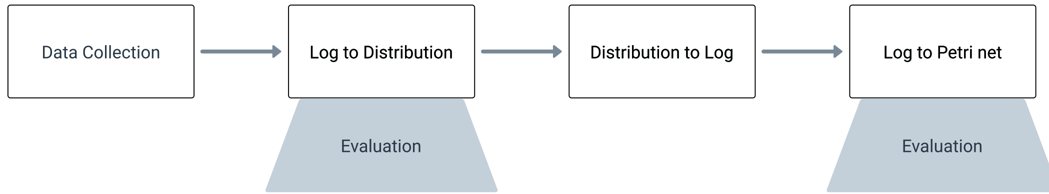

Figure 24: Proposed methodology.

5.2.1 Data Collection

The first step is data collection, that is, retrieving event logs. The data collection step is depicted in Figure 25, in which we distinguish between 3 different possibilities. The first is a 'real-life' log, as recorded from some process instance. Various real-life logs are readily available on `data.4tu.nl`; the earlier used event log on sepsis cases has been uploaded to said location. A second way to get logs is to simulate a Generalised Stochastic Petri Net (GSPN) [116]. While the precise definition is out of scope, the only important thing to realise is that with GSPNs there are no simulation parameters that the simulation is subject to, as all information is encoded in the model. Finally, as argued in Section 5.1, arbitrary integer sequences may be interpreted as event logs, though.

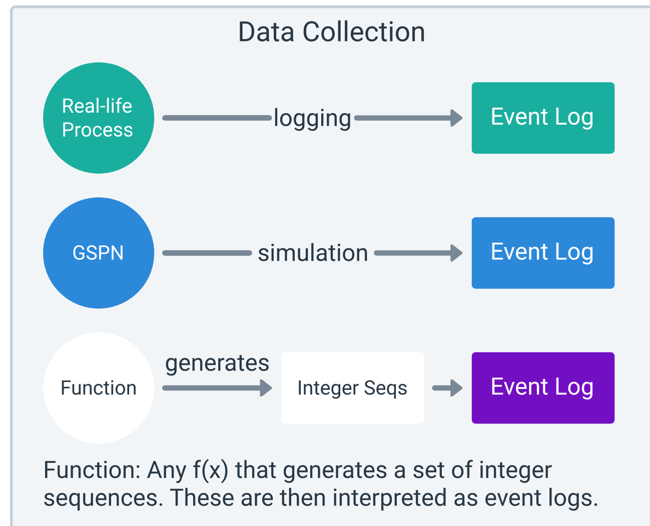

Figure 25: Data Collection

Integer sequences are readily available on OEIS [117], but these are individual sequences that correspond to individual traces (e.g. a sequence of prime numbers, or the Fibonacci sequence). Whilst there may be *some* use in considering *all* sequences on OEIS as coming from a single log, we assume that this will not give interesting results. This assumption is made since the sequences have different formulas and ideas behind them (prime numbers and numbers in the Fibonacci sequence are thus assumed to be from a different "log"). As such, we have written an integer sequence generator that can generate sequences of integers, available on GitHub [2] with documentation both in source code and available online [3]. There are two main files, one being the integer sequence generator, and the other being a XES converter that takes an XES file and returns an integer sequence.

[2]See `https://github.com/dbarenholz/integer_sequence`
[3]See `https://dbarenholz.github.io/integer_sequence/generator/index.html`

5.2.2 Log to Distribution

To learn probability distributions from logs, one can use the conversion to probability spaces as explained in Section 5.1, which effectively computes the empirical distribution. Recall that using (or even representing) a full joint distribution may not be feasible due to the curse of dimensionality, as illustrated in Section 2.2.7. By converting from log-space to probability-space, logs can be seen as datasets consisting of samples from some unknown but interesting probability distribution that we want to learn, which is precisely the setting of many of the probabilistic models. Learning these probabilistic models, however, can be non-trivial: for models based on neural networks (Sections 4.2 and 4.3) it is simply the act of training that learns (how to represent) the distribution, but for graphical models one needs to learn *both* the model parameters, as well as the structure of the underlying graph.

If a graphical model (Section 4.1) has an explicit likelihood formula — a likelihood ($\mathcal{L}$) describes the joint probability of the observed data as a function of the parameters of the chosen statistical model — one can use the *maximum likelihood framework* to learn parameters [118]. For instance, in BNs log-likelihood decomposes as shown in Equation 17. For MNs however, log-likelihood does not decompose as illustrated in Equation 18. Whether or not (log) likelihood decomposes is important for speed of computation: when likelihoods decompose to local ones, its easier to computer than if this is not the case.

$$\begin{aligned}
\mathcal{L} &= \sum_{i=1}^{n} \log p(\mathbf{x}^i) \\
&= \sum_{i=1}^{n} \log \prod_{X \in \mathbf{X}} p(\mathbf{x}^i_X \mid \mathbf{x}^i_{\mathbf{pa}(X)}) \\
&= \sum_{i=1}^{n} \sum_{X \in \mathbf{X}} \log p(\mathbf{x}^i_X \mid \mathbf{x}^i_{\mathbf{pa}(X)}) \\
&= \sum_{X \in \mathbf{X}} \overbrace{\sum_{i=1}^{n} \log p(\mathbf{x}^i_X \mid \mathbf{x}^i_{\mathbf{pa}(X)})}^{\mathcal{L}_X\text{:local log-likelihood for } X}
\end{aligned} \tag{17}$$

$$\begin{aligned}
\mathcal{L} &= \sum_{i=1}^{n} \log \frac{1}{\mathcal{Z}} \prod_{\mathbf{C} \in \mathcal{C}} f\left(\mathbf{x}^i_{\mathbf{C}}\right) \\
&= \sum_{i=1}^{n} \sum_{\mathbf{C} \in \mathcal{C}} \log f\left(\mathbf{x}^i_{\mathbf{C}}\right) - \log \mathcal{Z} \\
&= \sum_{i=1}^{n} \sum_{\mathbf{C} \in \mathcal{C}} \log f\left(\mathbf{x}^i_{\mathbf{C}}\right) - \overbrace{\log \sum_{\mathbf{x}} \prod_{\mathbf{C} \in \mathcal{C}} f\left(\mathbf{x}^i_{\mathbf{C}}\right)}^{\text{couples all factors}}
\end{aligned} \tag{18}$$

When it comes to learning structure, we differentiate between learners based on conditional independence tests (CI-based learners) and those based on optimising a score (score-based learners). The CI-based structure learning requires *many* **c**onditional **i**ndependence tests, which are not very reliable on finite datasets. An example of a CI-based learner is the (R)FCI algorithm [119], which avoids computing redundant (if some RV A is independent from some RV B, then we no longer need to test if B is independent from A) conditional independence tests. Still, score-based learners are much more prominent. A score-based learner views structure learning as discrete optimisation, where the goal is to find $G^* = \arg\max_{G \in [G]} \mathcal{S}(G, D)$, where $[G]$ is a family of graphs over the RVs in $\mathbf{X}$ and $S(G, D)$ is a suitable score defined on a graph and input data $D = \left\{\mathbf{x}^i\right\}_{i=1}^n$ consisting of n samples. An example of a good score is the *minimum description length* as shown in Equation 19, which trades off log-likelihood ($\mathcal{L}(G, \Theta, D)$) with the size of parameter space (Θ), weighted by the logarithm of the number of samples [120].

$$\mathcal{S}_{MDL} = \max_{\Theta} \mathcal{L}(G, \Theta, D) + \frac{\log n}{2} \cdot |\Theta| \tag{19}$$

When working with event data, D represents an entire log, and a single sample $\mathbf{x}^i$ represents a trace. Such a trace, as explained in Section 5.1, is a categorical random vector $\mathbf{x}^i = (X_1, \ldots, X_m)$ with categorical RVs X_j for each index $1 \leq j \leq m$ in the trace, and m being the length of the longest recorded trace.

It is interesting to note here that structure learning using score-based learners is a similar problem to PD. To reiterate the problem of a score-based structure learner: find $G^* = \arg\max_{G \in [G]} \mathcal{S}(G, D)$, where $[G]$ is a family of graphs over the RVs in $\mathbf{X}$ and $S(G, D)$ is a suitable score defined on a graph and input data $D = \{\mathbf{x}^i\}_{i=1}^n$ consisting of n samples. In simpler terms: "Find some graph from a family of graphs that is best according to some scoring function based on data". By replacing the term *graph* with *process model*, we effectively end up at the process discovery problem. The main difference between the two problems, is that PD algorithms in most cases fail to leverage the true underlying probability distribution, whereas representing said distribution is done with a probabilistic model for which a structure is learned.

5.2.3 Distribution to Log

Converting a distribution back to a log can be done by means of incremental inference, where one can distinguish between which behaviour is desired to convert: for frequent behaviour one takes activities with high probability, whereas for infrequent behaviour one can set thresholds such that only activities with low probability are used. The main idea of incremental inference is to, for each activity at index i in a (new) trace, compute the probability distribution of X_i given its predecessors. For instance, if we have already decided that a starting activity is a and a second activity is b, and we are interested in the third activity, then we compute $p(X_3 \mid X_1 = a, X_2 = b)$.

By converting the learned distribution back to an event log, one can consciously decide on which behaviour (frequent vs infrequent) is interesting, and consequently which behaviour is not desired. Comparing the converted event log to the original one may give meaningful insights into precisely which behaviour was filtered and whether or not this was desired, though further investigation is necessary. It may also be interesting to compare process models mined (using any PD method, preferably one that does not filter behaviour) on the original event log and the converted one, but again further investigation is needed.

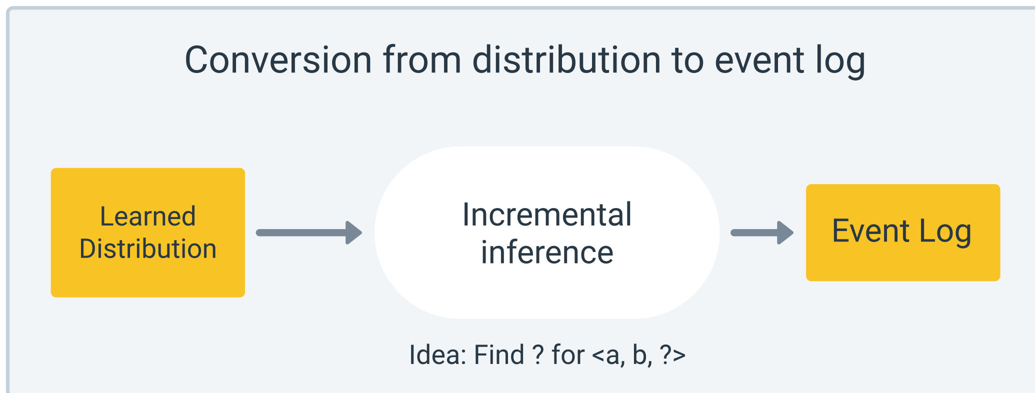

Figure 26: Converting a distribution to a log: incremental inference.

It is important to see that there are fundamental differences between evaluating log-model, log-log, model-model, and distribution-distribution. The first corresponds to the classical conformance checking idea (Section 2.1.3): measure how well a particular model represents the log it was learned on. In [121] they propose a method to evaluate log-model in a stochastic manner: by annotating the log with empirical probabilities (converting the log into a stochastic language), they construct a *reallocation matrix* describing the probability mass required to move between log and model, and then use said matrix to derive the *Earth Movers' Stochastic Conformance* measure. The measure defined in the paper is based on the *Earth Movers' Distance*, which is a method to evaluate similarity between two distributions [122]. Evaluating on a distribution level (described in Section

5.2.4) is done to find whether or not the used probabilistic model properly learned the underlying true distribution.

The methodology in [121] is one of the few conformance checking methods (that we are aware of) that can also be applied on a log-log (and model-model level), by tweaking the underlying definition of a reallocation matrix. Comparing log-log in our method can be powerful to investigate what happens *precisely* to an input log. It should be able to show which traces have effectively been filtered out.

5.2.4 Evaluation on Distributions

Given some learned distribution $\hat{p}$, it is paramount to evaluate how well this distribution describes the true distribution. Since the true distribution is unknown, the best we can do is compare it to the empirical distribution p that is created by counting (or pseudo-counting with Laplace smoothing) in the original event log. Comparing two distributions is done by computing their divergences. The most well-known divergence is the *Kullback-Leibler* divergence, also called relative entropy. Equation 20 shows how it is computed in a discrete case. It measures how one distribution is different from a reference distribution. The KL-divergence is *asymmetric*, meaning that $D_{KL}(p||\hat{p}) \neq D_{KL}(\hat{p}||p)$.

$$D_{KL}(\hat{p}||p) = \sum_{x \in \mathcal{X}} \hat{p}(x) \log \frac{\hat{p}(x)}{p(x)} \tag{20}$$

Another option that has interesting interpretations is the Jensen–Shannon divergence, shown in Equation 21, which is essentially a smoothed version of the KL-divergence. In particular, it is the mutual information between X associated to a mixture between p and q, and binary indicator Z used to switch between p and q to produce the mixture. As such, it is bounded by 0 and 1, assuming $\log_2$ (which makes interpretation nice, as opposed to 'unbounded' KL-divergence). An alternative interpretation of the Jensen-Shannon divergence is the average relative entropy of p and q to the entropy of the average distribution $m = \frac{p+q}{2}$.

$$D_{JSD}(p||\hat{p}) = \frac{1}{2} D_{KL}(p(x)||m(x)) + \frac{1}{2} D_{KL}(\hat{p}(x)||m(x) \ , \quad m = \frac{1}{2}(p(x) + \hat{p}(x)) \tag{21}$$

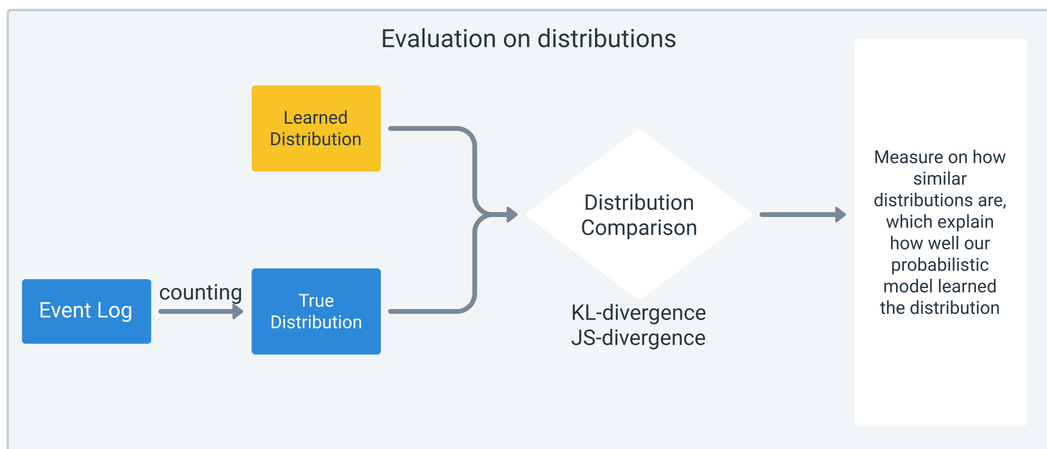

Figure 27: Evaluation on distributions: divergences.

6 Discussion

This section discusses the work that has been done. In particular, we look at possible issues with the methodology presented in Section 5.2 and argue in which cases it can and cannot be used. This argumentation can be found in Section 6.1. In Section 6.2 we quickly touch on the relation between infrequent behaviour, and the proposed methodology. We finish the discussion by looking at tasks that can be investigated another time in Section 6.3.

6.1 Threats to validity

The first problem is one fundamental to any learning task that attempts to learn from (real-life) data: there may not be enough data to learn from, and the data may not be accurate. To illustrate, suppose that M depicted in Figure 28 is a known model, say from domain knowledge, for some process. Assume that this process was something from the past, and that there are no new events to be logged e.g. the process is finished, and let $L = \left\{\langle a,b,c,d\rangle^5, \langle a,c,b,d\rangle^5\right\}$ be the corresponding log. Notice how it exists of *only* 10 traces. Looking at M it is clear that $\langle a,b,b,d\rangle$ $\langle a,c,c,d\rangle$ are perfectly valid traces, but they are not present in L. Since they are not present in L, which is the empirical data, there is no possibility for *any* learning algorithm to learn the correct M.

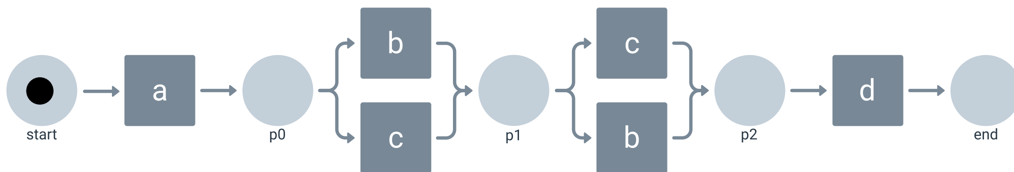

Figure 28: Example true model M.

Not only does this data problem manifest on the process discovery level, it manifests in probability as the way we (naively) compute empirical distributions is by (pseudo-) counting (recall the frequentist definition of probability shown in Equation 1, and see Section 2.2.2 on how this is applied to log data). More importantly, when learning a probability distribution *only* on the event log, we ignore domain knowledge that we may have: First, it can be the case that there is a partial process model for which we already know it is correct, and we wish to extend it. Second, by converting event data to RVs as proposed in 5.1 we ignore the fact that L may have known block structure. The current methodology as cannot work with domain knowledge in either case.

6.2 Infrequent behaviour

In Section 5.2.3 we propose a method that can convert a (representation of a) learned probability distribution to an event log by means of 'incremental inference'. This method allows choice for what behaviour (based on frequency) is deemed interesting, and then filters the behaviour that is unwanted by the process analytics expert. The filtered log can then be used by any PD method, without further filtering as all information present is already interesting, to learn a process model on *only* behaviour that is interesting. This filtering method may be particularly interesting to integrate with any PD method that guarantees a fitting model, such as the *ILP Miner* [49]: by doing so, one can create a model fitting to the behaviour that is interesting.

6.3 Future work

There are quite some interesting areas of research that one may dive into when it comes to combining the fields of probability and process discovery. While out of scope for this work, it is highly interesting to look at how infrequent behaviour is handled in process streaming algorithms such

as Competition Miner [123]. It is most likely non-trivial to decide which behaviour is infrequent, when not all behaviour has yet been seen.

A second avenue of research that may be looked at is using probabilistic models (in particular causal networks, where there is a clear interpretation besides only 'dependency') as a *direct* output of a discovery algorithm. It is interesting to see how such models would be evaluated, and whether or not they can model more or less types of behaviour (some process models are better used for block-structured data – how this would translate to a probabilistic model seems an interesting research question).

In stead of considering probabilities *directly* on the event log, one might consider probability on the (process) model structure instead. When using one of the discussed discovery methods that discover a petri net, it would be interesting to enhance such method with the underlying empirical distribution to see what differences it makes on the resulting model structure, and whether or not it can then directly mine models where frequencies are encoded, such as a generalised stochastic petri net.

As final idea for future work: in BNs there is the notion of independence. If there are there certain independence assumptions that are known to hold beforehand, perhaps these can then guide some discovery algorithm. This connects with the threat to validity of ignoring domain knowledge: if we have certain dependencies known, one could simply design a BN as model, and only learn its parameters using any applicable learner. Speaking of dependency: within a process view, there is only a dependency when there is some choice to be made within a process. Perhaps these dependencies are directly related to a probabilistic dependency in a model.

7 Conclusion

In this work we showed how infrequent behaviour is handled in process discovery (Section 3) which is concluded in Section 3.2 (**RQ 1**). State of the art algorithms may filter behaviour for various reasons, on various levels, or not at all. Another point of contribution is a classification of different types of probabilistic models (Section 4), all of which can be used to model the underlying distribution of a dataset. We show a methodology that leverages probability distributions and inference to generate a new event log (Section 5.2), on which any PD algorithm can be run to find a process model. Finally, in Section 6.3 we illustrate various interesting items of research that as far as we are aware have not yet been extensively looked at. Due to time constraints and other reasons we did not get to **RQ 2** or **RQ 3**; please see Appendix *B* for a brief reflection on why this the case.

References

[1] W. van der Aalst, A. Weijters, and L. Măruşter, "Workflow mining: Discovering process models from event logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 9, pp. 1128–1142, 10 2004. [Online]. Available: https://doi.org/10.1109/TKDE.2004.47 1, 10, 11, 16

[2] W. M. P. van der Aalst, *Process Mining*, 2nd ed. Springer Berlin Heidelberg, 2016. [Online]. Available: https://doi.org/10.1007/978-3-662-49851-4 1, 16

[3] D. Sommers, "Process discovery using machine learning," Master's thesis, Eindhoven University of Technology, 5612 AZ Eindhoven, 11 2020. 1, 16

[4] B. F. van Dongen and W. M. P. van der Aalst, "A meta model for process mining data," in *EMOI - INTEROP'05, Enterprise Modelling and Ontologies for Interoperability, Proceedings of the Open Interop Workshop on Enterprise Modelling and Ontologies for*

Interoperability, Co-located with CAiSE'05 Conference, Porto (Portugal), 13th-14th June 2005, ser. CEUR Workshop Proceedings, M. Missikoff and A. D. Nicola, Eds., vol. 160. CEUR-WS.org, 2005. [Online]. Available: http://ceur-ws.org/Vol-160/paper11.pdf 4

[5] "IEEE standard for eXtensible event stream (XES) for achieving interoperability in event logs and event streams," *IEEE Std 1849-2016*, pp. 1–50, 2016. [Online]. Available: https://doi.org/10.1109/IEEESTD.2016.7740858 4

[6] W. M. P. van der Aalst, *Process Mining*, 1st ed. Springer Berlin Heidelberg, 2011. [Online]. Available: https://doi.org/10.1007/978-3-642-19345-3 5, 6

[7] J. S. Rosenthal, *A First Look at Rigorous Probability Theory*, 2nd ed. WORLD SCIENTIFIC, 2006. [Online]. Available: https://doi.org/10.1142/6300 6

[8] A. Vergari, R. Peharz, Y. Choi, and G. V. den Broeck, "Probabilistic circuits: Representation and inference," 2020, european Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases ECML-PKDD. [Online]. Available: http://starai.cs.ucla.edu/slides/ECML20.pdf 8, 9, 17, 19, 20, 24, 25, 26, 27

[9] M. A. Nielsen, "Neural networks and deep learning," 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/ 10

[10] J. D. Weerdt, M. D. Backer, J. Vanthienen, and B. Baesens, "A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs," *Information Systems*, vol. 37, no. 7, pp. 654–676, 3 2012. [Online]. Available: https://doi.org/10.1016/j.is.2012.02.004 10

[11] D. Brons, "Algorithmic principles in process discovery: Strengths, weaknesses and addressment of specific problems - a structured literature review," 11 2019, capita Selecta. 10

[12] A. Augusto, R. Conforti, M. Dumas, M. L. Rosa, F. M. Maggi, A. Marrella, M. Mecella, and A. Soo, "Automated discovery of process models from event logs: Review and benchmark," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 4, pp. 686–705, 2019. [Online]. Available: https://doi.org/10.1109/TKDE.2018.2841877 10

[13] L. Wen, W. M. P. van der Aalst, J. Wang, and J. Sun, "Mining process models with non-free-choice constructs," *Data Mining and Knowledge Discovery*, vol. 15, no. 2, pp. 145–180, 2007. [Online]. Available: https://doi.org/10.1007/s10618-007-0065-y 11

[14] L. Wen, J. Wang, W. M. van der Aalst, B. Huang, and J. Sun, "Mining process models with prime invisible tasks," *Data & Knowledge Engineering*, vol. 69, no. 10, pp. 999–1021, 2010. [Online]. Available: https://doi.org/10.1016/j.datak.2010.06.001 11

[15] Q. Guo, L. Wen, J. Wang, Z. Yan, and P. S. Yu, "Mining invisible tasks in non-free-choice constructs," in *Business Process Management*, 9 2015, pp. 109–125. [Online]. Available: https://doi.org/10.1007/978-3-319-23063-4_7 11

[16] F. M. Maggi, R. P. J. C. Bose, and W. M. P. van der Aalst, "Efficient discovery of understandable declarative process models from event logs," in *CAiSE: International Conference on Advanced Information Systems Engineering*, 6 2012, pp. 270–285. [Online]. Available: https://doi.org/10.1007/978-3-642-31095-9_18 11, 12, 16

[17] F. M. Maggi, M. Dumas, L. García-Bañuelos, and M. Montali, "Discovering data-aware declarative process models from event logs," in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 81–96. [Online]. Available: https://doi.org/10.1007/978-3-642-40176-3_8 11, 12, 16

[18] C. D. Ciccio and M. Mecella, "A two-step fast algorithm for the automated discovery of declarative workflows," in *2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, 2013, pp. 135–142. [Online]. Available: https://doi.org/10.1109/CIDM.2013.6597228 11, 16

[19] M. Abe and M. Kudo, "Business monitoring framework for process discovery with real-life logs," in *BPM 2014: Business Process Management*, 9 2014, pp. 416–423. [Online]. Available: https://doi.org/10.1007/978-3-319-10172-9_30 12, 16, 17

[20] S. Ferilli, "Woman: Logic-based workflow learning and management," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, no. 6, pp. 744–756, 2014. [Online]. Available: https://doi.org/10.1109/TSMC.2013.2273310 12, 14, 16

[21] F. M. Maggi, T. Slaats, and H. A. Reijers, "The automated discovery of hybrid processes," in *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 392–399. [Online]. Available: https://doi.org/10.1007/978-3-319-10172-9_27 12, 16, 17

[22] O. Vasilecas, T. Savickas, and E. Lebedys, "Directed acyclic graph extraction from event logs," in *Communications in Computer and Information Science*. Springer International Publishing, 2014, pp. 172–181. [Online]. Available: https://doi.org/10.1007/978-3-319-11958-8_14 12, 16

[23] G. Greco, A. Guzzo, F. Lupia, and L. Pontieri, "Process discovery under precedence constraints," *ACM Transactions on Knowledge Discovery from Data*, vol. 9, no. 4, pp. 1–39, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2710020 12, 16

[24] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. USA: Prentice Hall Press, 2009, ch. 6, pp. 202–233. 12

[25] V. Liesaputra, S. Yongchareon, and S. Chaisiri, "Efficient process model discovery using maximal pattern mining," in *Lecture Notes in Computer Science*. Springer International Publishing, 2015, pp. 441–456. [Online]. Available: https://doi.org/10.1007/978-3-319-23063-4_29 12, 16, 17

[26] D. Breuker, , M. Matzner, P. Delfmann, J. Becker, , and and, "Comprehensible predictive models for business processes," *MIS Quarterly*, vol. 40, no. 4, pp. 1009–1034, Apr. 2016. [Online]. Available: https://doi.org/10.25300/misq/2016/40.4.10 12, 16, 17

[27] M. L. van Eck, N. Sidorova, and W. M. P. van der Aalst, "Discovering and exploring state-based models for multi-perspective processes," in *Lecture Notes in Computer Science*. Springer International Publishing, 2016, pp. 142–157. [Online]. Available: https://doi.org/10.1007/978-3-319-45348-4_9 12, 16, 17

[28] C. Li, J. Ge, L. Huang, H. Hu, B. Wu, H. Yang, H. Hu, and B. Luo, "Process mining with token carried data," *Information Sciences*, vol. 328, pp. 558–576, Jan. 2016. [Online]. Available: https://doi.org/10.1016/j.ins.2015.08.050 12, 16

[29] A. Mokhov, J. Carmona, and J. Beaumont, "Mining conditional partial order graphs from event logs," in *Transactions on Petri Nets and Other Models of Concurrency XI*. Springer Berlin Heidelberg, 2016, pp. 114–136. [Online]. Available: https://doi.org/10.1007/978-3-662-53401-4_6 13, 16

[30] S. Schönig, A. Rogge-Solti, C. Cabanillas, S. Jablonski, and J. Mendling, "Efficient and customisable declarative process mining with SQL," in *Advanced Information Systems Engineering*. Springer International Publishing, 2016, pp. 290–305. [Online]. Available: https://doi.org/10.1007/978-3-319-39696-5_18 13, 16

[31] B. V. Dongen and S. Shabani, "Relational xes: Data management for process mining," in *CAiSE Forum*. CEUR-WS.org, 2015. [Online]. Available: http://ceur-ws.org/Vol-1367/paper-22.pdf 13

[32] J. R. Koza, "Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems," Stanford University, Stanford, CA, USA, Tech. Rep., 1990. [Online]. Available: https://doi.org/10.5555/892491 13

[33] S. K. L. M. vanden Broucke, J. Vanthienen, and B. Baesens, "Declarative process discovery with evolutionary computing," in *2014 IEEE Congress on Evolutionary Computation (CEC)*, 2014, pp. 2412–2419. [Online]. Available: https://doi.org/10.1109/CEC.2014.6900293 13, 16

[34] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst, "Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity," *International Journal of Cooperative Information Systems*, vol. 23, no. 01, Mar. 2014. [Online]. Available: https://doi.org/10.1142/s0218843014400012 13, 14, 16, 17

[35] T. Molka, D. Redlich, M. Drobek, X.-J. Zeng, and W. Gilani, "Diversity guided evolutionary mining of hierarchical process models," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, Jul. 2015. [Online]. Available: https://doi.org/10.1145/2739480.2754765 14, 16, 17

[36] B. Vázquez-Barreiros, M. Mucientes, and M. Lama, "A genetic algorithm for process discovery guided by completeness, precision and simplicity," in *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 118–133. [Online]. Available: https://doi.org/10.1007/978-3-319-10172-9_8 14, 16, 17

[37] M. Ester, H. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *KDD*, 1996. [Online]. Available: http://www.cs.ecu.edu/~dingq/CSCI6905/readings/DBSCAN.pdf 14

[38] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "Optics: Ordering points to identify the clustering structure," in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 49–60. [Online]. Available: https://doi.org/10.1145/304182.304187 14

[39] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: An efficient data clustering method for very large databases," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 103–114. [Online]. Available: https://doi.org/10.1145/233269.233324 14

[40] R. Conforti, M. Dumas, L. García-Bañuelos, and M. L. Rosa, "BPMN miner: Automated discovery of BPMN process models with hierarchical structure," *Information Systems*, vol. 56, pp. 284–303, Mar. 2016. [Online]. Available: https://doi.org/10.1016/j.is.2015.07.004 14, 16

[41] H. Nguyen, M. Dumas, A. H. M. ter Hofstede, M. L. Rosa, and F. M. Maggi, "Mining business process stages from event logs," in *Advanced Information Systems Engineering*. Springer International Publishing, 2017, pp. 577–594. [Online]. Available: https://doi.org/10.1007/978-3-319-59536-8_36 14, 16

[42] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. New York, NY: The MIT Press, 2001, ch. 26.2, p. 651–664. 14

[43] H. M. W. Verbeek, W. M. P. van der Aalst, and J. Munoz-Gama, "Divide and conquer: A tool framework for supporting decomposed discovery in process mining," *The Computer Journal*, vol. 60, no. 11, pp. 1649–1674, May 2017. [Online]. Available: https://doi.org/10.1093/comjnl/bxx040 14, 16, 17

[44] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. New York, NY: The MIT Press, 2001, ch. 29.3, pp. 790–804. 14, 15

[45] J. Carmona and J. Cortadella, "Process discovery algorithms using numerical abstract domains," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 12, pp. 3064–3076, 2014. [Online]. Available: https://doi.org/10.1109/TKDE.2013.156 15, 16

[46] R. Lorenz, S. Mauser, and G. Juhás, "How to synthesize nets from languages: A survey," in *Proceedings of the 39th Conference on Winter Simulation: 40 Years! The Best is yet to Come*, ser. WSC '07. IEEE Press, 2007, p. 637–647. 15

[47] B. N. Yahya, M. Song, H. Bae, S. ook Sul, and J.-Z. Wu, "Domain-driven actionable process model discovery," *Computers & Industrial Engineering*, vol. 99, pp. 382–400, Sep. 2016. [Online]. Available: https://doi.org/10.1016/j.cie.2016.05.010 15, 16, 17

[48] J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, and A. Serebrenik, "Process discovery using integer linear programming," in *Applications and Theory of Petri Nets*. Springer Berlin Heidelberg, 2008, pp. 368–387. [Online]. Available: https://doi.org/10.1007/978-3-540-68746-7_24 15, 16

[49] S. J. van Zelst, B. F. van Dongen, W. M. P. van der Aalst, and H. M. W. Verbeek, "Discovering workflow nets using integer linear programming," *Computing*, vol. 100, no. 5, pp. 529–556, Nov. 2017. [Online]. Available: https://doi.org/10.1007/s00607-017-0582-5 15, 16, 17, 34

[50] A. Weijters, W. Aalst, van der, and A. Alves De Medeiros, *Process mining with the HeuristicsMiner algorithm*, ser. BETA publicatie : working papers. Technische Universiteit Eindhoven, 2006. [Online]. Available: https://research.tue.nl/files/2388011/615595.pdf 15, 16

[51] Z. Huang and A. Kumar, "A study of quality and accuracy tradeoffs in process mining," *INFORMS Journal on Computing*, vol. 24, no. 2, pp. 187–341, 3 2011. [Online]. Available: https://doi.org/10.1287/ijoc.1100.0444 15, 16, 17

[52] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Discovering block-structured process models from incomplete event logs," in *PETRI NETS 2014: Application and Theory of Petri Nets and Concurrency*, 6 2014, pp. 91–110. [Online]. Available: https://doi.org/10.1007/978-3-319-07734-5_6 15, 16, 17

[53] A. Augusto, R. Conforti, M. Dumas, M. L. Rosa, and G. Bruno, "Automated discovery of structured process models from event logs: The discover-and-structure approach," *Data & Knowledge Engineering*, vol. 117, pp. 373–392, Sep. 2018. [Online]. Available: https://doi.org/10.1016/j.datak.2018.04.007 15, 16, 17

[54] J. D. Smedt, J. D. Weerdt, and J. Vanthienen, "Fusion miner: Process discovery for mixed-paradigm models," *Decision Support Systems*, vol. 77, pp. 123–136, Sep. 2015. [Online]. Available: https://doi.org/10.1016/j.dss.2015.06.002 16

[55] S. K. vanden Broucke and J. D. Weerdt, "Fodina: A robust and flexible heuristic process discovery technique," *Decision Support Systems*, vol. 100, pp. 109–118, Aug. 2017. [Online]. Available: https://doi.org/10.1016/j.dss.2017.04.005 16, 17

[56] A. Augusto, R. Conforti, M. Dumas, and M. L. Rosa, "Split miner: Discovering accurate and simple business process models from event logs," in *2017 IEEE International Conference on Data Mining (ICDM)*. IEEE, Nov. 2017. [Online]. Available: https://doi.org/10.1109/icdm.2017.9 16

[57] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012. [Online]. Available: https://doi.org/10.1109/MSP.2012.2211477 16

[58] D. Heckerman, "A tutorial on learning with bayesian networks," in *Innovations in Bayesian Networks*. Springer Berlin Heidelberg, 2008, pp. 33–82. [Online]. Available: https://doi.org/10.1007/978-3-540-85066-3_3 18

[59] A. Darwiche, *Inference by Variable Elimination*. Cambridge University Press, 2009, ch. 6, p. 126–151. [Online]. Available: https://doi.org/10.1017/CBO9780511811357.007 18, 19

[60] P. Diaconis, "The markov chain monte carlo revolution," *Bulletin of the American Mathematical Society*, vol. 46, no. 2, pp. 179–205, Nov. 2008. [Online]. Available: https://doi.org/10.1090/s0273-0979-08-01238-x 18

[61] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988. 18

[62] P. Suchánek, F. Marecki, and R. Bucki, "Self-learning bayesian networks in diagnosis," *Procedia Computer Science*, vol. 35, pp. 1426–1435, 2014. [Online]. Available: https://doi.org/10.1016/j.procs.2014.08.200 18

[63] V. Sevinc, O. Kucuk, and M. Goltas, "A bayesian network model for prediction and analysis of possible forest fire causes," *Forest Ecology and Management*, vol. 457, p. 117723, Feb. 2020. [Online]. Available: https://doi.org/10.1016/j.foreco.2019.117723 18

[64] M. Meila and M. I. Jordan, "Learning with mixtures of trees," *J. Mach. Learn. Res.*, vol. 1, p. 1–48, Sep. 2001. [Online]. Available: https://dl.acm.org/doi/10.1162/153244301753344605 18

[65] C. Chow and C. Liu, "Approximating discrete probability distributions with dependence trees," *IEEE Transactions on Information Theory*, vol. 14, no. 3, pp. 462–467, May 1968. [Online]. Available: https://doi.org/10.1109/tit.1968.1054142 18

[66] T. Szántai and E. Kovács, "Hypergraphs as a mean of discovering the dependence structure of a discrete multivariate probability distribution," *Annals of Operations Research*, vol. 193, no. 1, pp. 71–90, Nov. 2010. [Online]. Available: https://doi.org/10.1007/s10479-010-0814-y 18

[67] R. G. Cowell, P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter, *Probabilistic Networks and Expert Systems*. Springer-Verlag, 1999, ch. 4, pp. 43–61. [Online]. Available: https://doi.org/10.1007/b97670 18

[68] S. Dasgupta, "Learning polytrees," in *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, ser. UAI'99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, p. 134–141. 18

[69] F. R. Bach and M. I. Jordan, "Thin junction trees," in *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, ser. NIPS'01. Cambridge, MA, USA: MIT Press, 2001, p. 569–576. 19

[70] G. R. Cross and A. K. Jain, "Markov random field texture models," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-5, no. 1, pp. 25–39, 1983. 19

[71] S. Z. Li, "Markov random field models in computer vision," in *Computer Vision — ECCV '94*. Springer Berlin Heidelberg, 1994, pp. 361–370. [Online]. Available: https://doi.org/10.1007/bfb0028368 19

[72] M. Banf and S. Y. Rhee, "Enhancing gene regulatory network inference through data integration with markov random fields," *Scientific Reports*, vol. 7, no. 1, Feb. 2017. [Online]. Available: https://doi.org/10.1038/srep41174 19

[73] C. Yue, "Markov random fields and gibbs sampling for image denoising," 2018. 19

[74] R. Mourad, C. Sinoquet, N. L. Zhang, T. Liu, and P. Leray, "A survey on latent tree models and applications," *Journal of Artificial Intelligence Research*, vol. 47, pp. 157–203, May 2013. [Online]. Available: https://doi.org/10.1613/jair.3879 20

[75] B. Uria, M. Côté, K. Gregor, I. Murray, and H. Larochelle, "Neural autoregressive distribution estimation," *CoRR*, vol. abs/1605.02226, 2016. [Online]. Available: http://arxiv.org/abs/1605.02226 21

[76] B. Uria, I. Murray, and H. Larochelle, "Rnade: The real-valued neural autoregressive density-estimator," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'13. Red Hook, NY, USA: Curran Associates Inc., 2013, p. 2175–2183. 21

[77] S. Lauly, Y. Zheng, A. Allauzen, and H. Larochelle, "Document neural autoregressive distribution estimation," *CoRR*, vol. abs/1603.05962, 2016. [Online]. Available: http://arxiv.org/abs/1603.05962 21

[78] B. Uria, I. Murray, and H. Larochelle, "A deep and tractable density estimator," in *Proceedings of the 31st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, E. P. Xing and T. Jebara, Eds., vol. 32, no. 1. Bejing, China: PMLR, 22–24 Jun 2014, pp. 467–475. [Online]. Available: http://proceedings.mlr.press/v32/uria14.html 21

[79] Y. Zheng, Y.-J. Zhang, and H. Larochelle, "A deep and autoregressive approach for topic modeling of multimodal data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 6, pp. 1056–1069, 2016. 21

[80] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent, "Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription," *arXiv preprint arXiv:1206.6392*, 2012. 21

[81] Y. Zheng, R. S. Zemel, Y.-J. Zhang, and H. Larochelle, "A neural autoregressive approach to attention-based recognition," *International Journal of Computer Vision*, vol. 113, no. 1, pp. 67–79, Sep. 2014. [Online]. Available: https://doi.org/10.1007/s11263-014-0765-x 21

[82] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org. 22

[83] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2014. [Online]. Available: http://arxiv.org/abs/1312.6114 22

[84] A. Subramanian, "Pytorch-vae," https://github.com/AntixK/PyTorch-VAE, 2020. 22

[85] R. Peharz, "Lecture slides: Variational autoencoders 2," May 2021. 22

[86] O. Cífka, A. Ozerov, U. Simsekli, and G. Richard, "Self-supervised VQ-VAE for one-shot music style transfer," *CoRR*, vol. abs/2102.05749, 2021. [Online]. Available: https://arxiv.org/abs/2102.05749 22

[87] Q. Zhao, E. Adeli, N. Honnorat, T. Leng, and K. M. Pohl, "Variational autoencoder for regression: Application to brain aging analysis," *CoRR*, vol. abs/1904.05948, 2019. [Online]. Available: http://arxiv.org/abs/1904.05948 22

[88] S. K. Kumaran, D. P. Dogra, P. P. Roy, and A. Mitra, "Video trajectory classification and anomaly detection using hybrid cnn-vae," *arXiv preprint arXiv:1812.07203*, 2018. 22

[89] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014. 22

[90] J. Hayes, L. Melis, G. Danezis, and E. De Cristofaro, "Logan: Evaluating privacy leakage of generative models using generative adversarial networks," 05 2017. 22

[91] T. Karras, S. Laine, M. Aittala, J. Hellsten, J. Lehtinen, and T. Aila, "Analyzing and improving the image quality of stylegan," *CoRR*, vol. abs/1912.04958, 2019. [Online]. Available: http://arxiv.org/abs/1912.04958 23

[92] J. Gui, Z. Sun, Y. Wen, D. Tao, and J. Ye, "A review on generative adversarial networks: Algorithms, theory, and applications," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2021. 23

[93] C. Huang, D. Krueger, A. Lacoste, and A. C. Courville, "Neural autoregressive flows," *CoRR*, vol. abs/1804.00779, 2018. [Online]. Available: http://arxiv.org/abs/1804.00779 23

[94] L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using real NVP," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=HkpbnH9lx 23

[95] G. Papamakarios, T. Pavlakou, and I. Murray, "Masked autoregressive flow for density estimation," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 2335–2344. 23

[96] M. Germain, K. Gregor, I. Murray, and H. Larochelle, "Made: Masked autoencoder for distribution estimation," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, F. Bach and D. Blei, Eds., vol. 37. Lille, France: PMLR, 07–09 Jul 2015, pp. 881–889. [Online]. Available: http://proceedings.mlr.press/v37/germain15.html 23

[97] A. Van Den Oord, N. Kalchbrenner, and K. Kavukcuoglu, "Pixel recurrent neural networks," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16. JMLR.org, 2016, p. 1747–1756. 23

[98] R. Giri, S. V. Tenneti, F. Cheng, K. Helwani, U. Isik, and A. Krishnaswamy, "Unsupervised anomalous sound detection using self-supervised classification and group masked autoencoder for density estimation," *Challenge on Detection and Classification of Acoustic Scenes and Events (DCASE 2020 Challenge), Tech. Rep*, 2020. 23

[99] M. Manninen, V. Taivassalo, and S. Kallio, "On the mixture model for multiphase flow," 1996. 23

[100] T. L. Bailey, C. Elkan *et al.*, "Fitting a mixture model by expectation maximization to discover motifs in bipolymers," 1994. 23

[101] N. Cohen, O. Sharir, and A. Shashua, "On the expressive power of deep learning: A tensor analysis," *CoRR*, vol. abs/1509.05009, 2015. [Online]. Available: http://arxiv.org/abs/1509.05009 23

[102] H. Poon and P. Domingos, "Sum-product networks: A new deep architecture," in *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*. IEEE, Nov. 2011. [Online]. Available: https://doi.org/10.1109/iccvw.2011.6130310 24, 25

[103] A. Shih, G. V. den Broeck, P. Beame, and A. Amarilli, "Smoothing structured decomposable circuits," *CoRR*, vol. abs/1906.00311, 2019. [Online]. Available: http://arxiv.org/abs/1906.00311 24

[104] R. Peharz, S. Lang, A. Vergari, K. Stelzner, A. Molina, M. Trapp, G. V. den Broeck, K. Kersting, and Z. Ghahramani, "Einsum networks: Fast and scalable learning of tractable probabilistic circuits," *CoRR*, vol. abs/2004.06231, 2020. [Online]. Available: https://arxiv.org/abs/2004.06231 25

[105] R. Peharz, R. Gens, and P. Domingos, "Learning selective sum-product networks," in *ICML Workshop on Learning Tractable Probabilistic Models*, 06 2014. 25

[106] C. J. Butz, J. S. Oliveira, A. E. D. Santos, and A. L. Teixeira, "Deep convolutional sum-product networks," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 3248–3255, Jul. 2019. [Online]. Available: https://doi.org/10.1609/aaai.v33i01.33013248 25

[107] A. Darwiche, "A differential approach to inference in bayesian networks," *Journal of the ACM*, vol. 50, no. 3, pp. 280–305, May 2003. [Online]. Available: https://doi.org/10.1145/765568.765570 25

[108] T. Rahman, P. Kothalkar, and V. Gogate, "Cutset networks: A simple, tractable, and scalable approach for improving the accuracy of chow-liu trees," in *Machine Learning and Knowledge Discovery in Databases*. Springer Berlin Heidelberg, 2014, pp. 630–645. [Online]. Available: https://doi.org/10.1007/978-3-662-44851-9_40 26

[109] D. Kisa, G. Van den Broeck, A. Choi, and A. Darwiche, "Probabilistic sentential decision diagrams," in *Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning*, ser. KR'14. AAAI Press, 2014, p. 558–567. 26

[110] M. JAEGER, "PROBABILISTIC DECISION GRAPHS — COMBINING VERIFICATION AND AI TECHNIQUES FOR PROBABILISTIC INFERENCE," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 12, no. supp01, pp. 19–42, Jan. 2004. [Online]. Available: https://doi.org/10.1142/s0218488504002564 27

[111] P. Billingsley, *Probability and measure*, 3rd ed. New York: Wiley, 1995, ch. 33, pp. 427–444. 27

[112] R. Dechter and R. Mateescu, "AND/OR search spaces for graphical models," *Artificial Intelligence*, vol. 171, no. 2-3, pp. 73–106, Feb. 2007. [Online]. Available: https://doi.org/10.1016/j.artint.2006.11.003 27

[113] R. Marinescu and R. Dechter, "Best-first AND/OR search for 0/1 integer programming," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer Berlin Heidelberg, 2007, pp. 171–185. [Online]. Available: https://doi.org/10.1007/978-3-540-72397-4_13 27

[114] F. Mannhardt, "Sepsis Cases - Event Log," 12 2016. [Online]. Available: https://data.4tu.nl/articles/dataset/Sepsis_Cases_-_Event_Log/12707639 28, 29

[115] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. New York: Cambridge University Press, 2008, oCLC: ocn190786122. 29

[116] C. Eisentraut, H. Hermanns, J.-P. Katoen, and L. Zhang, "A semantics for every gspn," in *International Conference on Applications and Theory of Petri Nets and Concurrency*. Springer, 2013, pp. 90–109. 30

[117] O. F. I. (2021), "The on-line encyclopedia of integer sequences." [Online]. Available: http://oeis.org 30

[118] R. J. Rossi, *Mathematical statistics: an introduction to likelihood based inference*, 1st ed. Hoboken, NJ: John Wiley & Sons, 2018. 31

[119] D. Colombo, M. H. Maathuis, M. Kalisch, and T. S. Richardson, "Learning high-dimensional directed acyclic graphs with latent and selection variables," *The Annals of Statistics*, vol. 40, no. 1, Feb. 2012. [Online]. Available: https://doi.org/10.1214/11-aos940 31

[120] W. Lam and F. Bacchus, "Learning bayesian belief networks: An approach based on the MDL principle," *Computational Intelligence*, vol. 10, no. 4, 1994. 31

[121] S. J. Leemans, W. M. van der Aalst, T. Brockhoff, and A. Polyvyanyy, "Stochastic process mining: Earth movers' stochastic conformance," *Information Systems*, vol. 102, p. 101724, Dec. 2021. [Online]. Available: https://doi.org/10.1016/j.is.2021.101724 32, 33

[122] B. Fisher, "The earth mover's distance." [Online]. Available: https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/RUBNER/emd.htm 32

[123] D. Redlich, T. Molka, W. Gilani, G. Blair, and A. Rashid, "Scalable dynamic business process discovery with the constructs competition miner," in *SIMPDA*. CEUR-WS.org, 2014. [Online]. Available: http://ceur-ws.org/Vol-1293/paper7.pdf 35

Acronyms

AC arithmetic circuit. 25, 26

AoG AndOrGraph. 27

BN Bayesian network. 18–20, 27, 31, 35

CLT Chow-Liu tree. 18–20, 26

CNet cutset network. 26

CPD conditional probability distributions. 18

CPN Coloured Petri net. 4

DAG directed acyclic graph. 10, 12, 17, 18

GAN generative adversarial model. 22, 23

GCN graph convolutional network. 1, 2, 16

ILP integer linear programming. 10, 14, 15, 27

LTM latent tree model. 20, 23

MADE masked autoencoder for distribution estimation. 23

MAF masked autoregressive flow. 23

ML machine learning. 1, 3, 10, 12, 14–16, 21

MN Markov network. 19, 20, 31

NADE neural autoregressive distribution estimation. 21, 23

PC probabilistic circuit. 23–28

PD process discovery. 1–4, 6, 10–17, 32, 34, 35

PDF probability density function. 7, 24

PDG probabilistic decision graph. 27

PGM probabilistic graphical model. 17, 20, 24

PMF probability mass function. 7

PN Petri net. 1, 4, 5, 11, 12, 15, 16

PSDD probabilistic sentential decision diagram. 26, 27

RNADE real-valued neural autoregressive distribution estimator. 21

RV random variable. 3, 7, 9, 17, 18, 20, 24–28, 31, 32, 34

SPN Sum-Product Network. 24, 25

TJT thin Junction tree. 19, 20

VAE variational auto encoder. 22, 23

WF-net workflow net. 4

Reflection

This section has two purposes: first, it contains a reflection on the things learned during the internship, and second it contains an explanation as to why research questions **RQ 2** and **RQ 3** remain unanswered. As it is mostly a personal reflection, it is written from a first-person perspective.

Feedback to myself

I will use a simple enumeration for the points of feedback that I would like to take with me for future future research projects. The first two points can be classified as **time management**, whereas the other points are **scientific method**.

1. **Structuring a report takes time**. In fact, more time that I had initially thought. By not being aware of this fact, it felt that at times I had very little progress, which annoyed me and was detrimental to my mental health. In future research, I should be aware that things may take more time than expected, and should be resilient to such changes from expectation.

2. **Writing $\neq$ editing**. Starting to write is hard. It is easier (to me) to first create an outline (which takes time), and then keep adding bullet points until a proper structure presents itself. Only then should I start writing sections, and *not edit*. Editing should be done *afterwards*, where I go over the entire text, and mark where I dislike sentences for whichever reason. Then, I can process these remarks and edit the text to my liking. Both writing and editing take more time than expected, especially when being a perfectionist to a certain degree.

3. **There exists a scientific method**, which I only found out about during the final stages of implementing feedback. I should *definitely* read up on said methodology, since it will prevent some of the mistakes I made during this project (such as solving the wrong problem, and doing the difficult thing before doing the easy thing).

4. **Understand the problem, before trying to make a solution**. This is something I did plainly wrong during the internship. I should not hurry doing research, but in stead ensure that I have a good understanding of the topics and the problem there is at hand. Explicitly writing down said problem, and evaluating whether or not it is a "research problem" or specific "research question" will help me tremendously. This links back to using the scientific method that exists, but did not know about.

5. **Do easy things before doing hard things**. I **SHOULD NOT** make a problem more complex than it is. There will most likely be things that I can do that are relatively easy, and will help me gain a better understanding.

Why research questions are unanswered

This internship was my first real research project, and I started with it not knowing *anything* about research methodologies and best practises. As such, I have made many 'mistakes' that one might encounter when not following any guideline, as present in above section of 'feedback to myself'. The reason why I did not manage to answer all questions is due to said mistakes.