MASTER

Process discovery using machine learning

Sommers, Dominique

Award date:
2020

Link to publication
Process discovery using Machine Learning

Master Thesis Report

Dominique Sommers

Supervisors:
Dirk Fahland
Vlado Menkovski

November 30, 2020
Abstract

One of the core activities of business process management is process discovery. Process discovery is the task of producing a process model, which is a graphical representation of a business process by looking at the different process execution instances that are recorded. Such a process model gives an abstract interpretation of the process exposing its logic.

Process discovery has solely been tackled algorithmically, where in this thesis machine learning is introduced to the problem to investigate whether this could overcome the drawbacks of current approaches. Algorithmic approaches have both implicit and explicit assumptions on structures within the data as well as what data is relevant and irrelevant. Our approach exploits machine learning to learn patterns from data to generalize beyond such assumptions and can make decisions on relevance of data based on experience and context.

To arrive at a potential method, the state-of-the-art algorithms are analyzed to reveal their drawbacks. Then, other research fields were analyzed, which problems are translated to that of process discovery already using machine learning techniques. A framework incorporating these techniques is developed that can continuously learn from automatically generated synthetic data and produce multiple process models that the model believes fit best to the given process data.

This thesis offers a machine learning based approach to the task of process discovery without the need for manually annotated data. Moreover, it opens up the field for other researchers to further explore the usage of machine learning methods in the field of process discovery.
Acknowledgments

First of all, I would like to thank Dirk Fahland and Vlado Menkovski for their valuable guidance, feedback and support during this master project. Special thanks go out to Dirk for supporting me in the idea to go out on a limb by introducing machine learning to process discovery. Furthermore, I especially appreciate the fact that both supervisors took on the challenge of going outside their regular fields, being machine learning for Dirk and process discovery for Vlado. Finally, I thank my friends and family for their crucial support and distraction during the last nine months.
Contents

I Preamble

1 Introduction 7
1.1 Automated process discovery 7
1.2 State of the art 7
1.3 Research questions 8
1.4 Approach 10
1.5 Findings 10

II Background

2 Preliminaries 12
2.1 Event logs 12
2.2 Petri nets 13
2.3 Alpha relations 15
2.4 Process trees 16

3 Process discovery 17
3.1 Objectives 17
3.2 Quality criteria 18
3.2.1 Fitness 18
3.2.2 Precision 18
3.2.3 Fitness vs precision 18
3.2.4 Generalization 19
3.2.5 Simplicity 21
3.2.6 Soundness 21
3.3 State of the art 22
3.3.1 Heuristics Miner 22
3.3.2 Inductive Miner 23
3.3.3 Split Miner 23
3.3.4 ILP Miner 24

4 Related work 26
4.1 Other fields 26
4.1.1 Medicine discovery 26
4.1.2 Natural language processing 26
4.1.3 Network analysis 27
III Approach

5 General approach
5.1 Roadmap
5.2 Gathering training data
5.3 Handling the input data
5.4 Training the machine learning model
5.5 Evaluating the model

6 Synthetic data generation
6.1 Process trees
6.2 Simulation

7 Data preparation
7.1 Graph construction
7.2 Constructing the trace graph
7.2.1 Node features
7.2.2 Edges
7.3 Constructing the Petri net graph
7.3.1 Transitions
7.3.2 Candidate places
7.3.3 Candidate silent transitions

8 Process model generation
8.1 Sequential model overview
8.2 Neural networks
8.2.1 Propagation network
8.2.2 ‘Add candidate’ network
8.2.3 ‘Choose candidate’ network
8.3 Training
8.3.1 Teacher forcing
8.4 Inference
8.4.1 Beam search
IV Evaluation

9 Synthetic data

9.1 Objective .......................................................... 63
9.2 Experimental setup ............................................. 64
9.3 Execution .......................................................... 64
  9.3.1 Synthetic dataset generation ......................... 64
  9.3.2 Trace variant frequency analysis .................... 65
  9.3.3 Deviating behavior analysis ......................... 66
9.4 Results .......................................................... 68
9.5 Discussion ....................................................... 71

10 Training/testing phase

10.1 Objective ....................................................... 73
10.2 Experimental setup ........................................... 74
10.3 Execution ....................................................... 74
10.4 Results .......................................................... 76
10.5 Discussion ....................................................... 79

11 Evaluation phase

11.1 Objective ....................................................... 83
11.2 Experimental setup ........................................... 84
11.3 Execution ....................................................... 85
11.4 Results .......................................................... 85
11.5 Discussion ....................................................... 89
  11.5.1 Quantitative analysis ................................. 92
  11.5.2 Synthetic data vs real-life data ..................... 93
  11.5.3 Qualitative analysis .................................. 93
  11.5.4 Running times ......................................... 95

V Conclusion

12 Conclusion

12.1 Contributions ................................................. 98
12.2 Limitations of the proposed method ..................... 99
12.3 Future work ................................................... 99

VI Appendices

A Alternate approaches

A.1 RNN input handling ........................................... 109
A.2 Variational autoencoder for process model generation 110
B  Algorithms for constructing candidate places 112
   B.1  Constructing the one-to-one places .......................... 112
   B.2  Constructing the one-to-many and many-to-one places ...... 113
   B.3  Constructing the many-to-many places ........................ 114

C  Discriminative version of our approach 115

D  Information propagation GCN 117

E  Conformance metrics results training phase 119
Part I

Preamble
Chapter 1

Introduction

1.1 Automated process discovery

The process mining life cycle, as illustrated in Figure 1.1, is the core of business process management and consists of process discovery, process analysis, process redesign, process implementation and process monitoring and controlling [9]. The aim of this life cycle is to improve a business process by analyzing, simulating and visualizing the actual process according to the gathered data. Process discovery is the first step in the life cycle and its output quality has impact on all subsequent steps, hence it is a very important aspect in process mining. The task of automated process discovery is a subclass of the process discovery task considering IT-supported processes which aims at revealing how the process actually behaves.

A business process generates event logs containing the activities in a specific process instance. In such an instance, different variants of activities could be present. For example, the process of a customer ordering a product which is in stock is different from where it is not in stock. There are overlapping activities, but some parts differ. A process model captures what was found in these event logs, giving an abstract overview of the different process instances present in the process and thus exposing the logic of the process. This process model is generated during the process discovery phase by taking the event logs into consideration, finding specific patterns and detecting possible deviations.

Important here is that the discovered process model has to be understandable by users and captures the real process in a suitable way, meaning that it does not allow for too much deviation from the recorded process instances, but also that it is not too strict in what process instance variants could be present. A difficulty in process discovery is that the optimal process model is often not known, since various aspects are to be maximized and a process model can not be checked for optimality.

1.2 State of the art

Several methods have been proposed to do so already, like the Heuristics Miner [43], Inductive Miner [19], ILP Miner [39] and the Split Miner [2], all solving the
problem algorithmically. Although they do offer robustness and guarantees on the outcomes, they do have drawbacks. Algorithmic methods contain a representational bias created by their design decisions, since they aim at finding structures in the data assuming that the data actually contains these structures, reducing their scalability. Common assumptions are:

- One task is to be modeled per activity;
- The process model is block-structured;
- Infrequent behavior is deemed irrelevant for the process.

These assumptions are not always well-grounded. Sometimes, activities could be better modeled using multiple tasks, e.g. cancellation events. Real-life processes are seldom as well structured to be block-structured. Infrequent behavior could well hold relevant information for the process although it could also contain information that blurs the process logic. Lastly, the process models are constructed by looking at structures locally, rather than globally.

In this thesis we explore the possibilities of performing process discovery with less a priori assumptions. Our approach for doing so is introducing machine learning to the problem, since machine learning techniques, like neural networks, are able to find both explicit and implicit features in data and is not based on assumptions, but on experience and statistics from previously seen data. Furthermore, a machine learning based method can be designed such that global information is used while making choices locally. The task of process discovery has not been tackled using machine learning as of yet, hence the project is an explorative project in the sense that the method’s potential will be researched rather than focusing on implementing or improving existing techniques.

1.3 Research questions

The research questions (RQ) that are answered throughout this thesis are:

RQ1 To what degree is it possible to have an assumption-free method while achieving comparable results as assumption-based methods?
This is the main question that we want to answer in this project. Since the assumptions in the existing methods are not always well-grounded, an assumption-free method is called after to increase scalability. This is only useful when such methods can compete with the existing methods, raising two sub-questions:

RQ1.1 *By how much can the number of assumptions be reduced*?

Machine learning is known for finding patterns in data and making choices based on what it has seen without having assumptions on the data. Our method aims at incorporating this technique to reduce the number of assumptions. How many assumptions are still required is the question, which is difficult to measure, but it can be reflected on.

RQ1.2 *How does the method perform compared to assumption-based approaches*?

Process discovery is a unsupervised learning problem. However, with conformance metrics we have a notion of quality for the produced process models. Using these metrics, the method can be evaluated and compared to the state-of-the-art methods. The questions here are whether the proposed method is able to produce good process models and when is it superior or inferior to the algorithmic approaches. Lastly, the running times are to be compared between the different methods.

RQ2 *How can the task of processing discovery be tackled accurately using machine learning techniques*?

A challenge is to translate the process discovery problem into a learning task appropriate for machine learning techniques, which is the second research question. This consists of two parts:

RQ2.1 *How can a graph-structured process model be produced from a given sequence-structured event log using machine learning techniques*?

Machine learning techniques exist both for sequence-structured data as well as for graph-structured data, both of which are not the common data structure for machine learning. How to incorporate both into a method that can learn to solve the task of process discovery is the question here.

RQ2.2 *How can it accurately solve the task and generalize beyond the training data*?

With a method that can produce process models from event logs it is not yet necessarily able to learn to solve the task accurately for unseen data. Therefore, the next question that is raised is how it can do so accurately and how well it is able to find patterns in the data such that it generalizes beyond the training data. Generalization is required to also accurately perform on unseen data.

RQ3 *How to acquire a valid synthetic dataset that can be used for training a machine learning model*?

Machine learning needs lots of data, since this is where the knowledge is extracted from. Not a lot of real-life data is readily available, meaning that synthetic data has to be generated. The question here is whether this is possible to do in such a way that this synthetic data has similarities with real-life datasets in terms of deviations.
1.4 Approach

The approach taken starts by researching the problem of process discovery itself and the existing methods, finding their advantages and disadvantages, elaborated on in Chapter 3. Then, other fields are researched that are already using machine learning to solve tasks that can be translated to that of process discovery, focused on in Chapter 4. Techniques are taken from these fields in order to develop a new approach. This development started by creating and analyzing synthetic data that can be used to train the machine learning model, which can be found in Chapters 6 and 9 respectively.

Our proposed technique first prepares the input in the form of graphs to match the data structure of output data being process models, where the transition corresponding to the activities are fixed nodes in the graph. Then, similarly to how the ILP Miner operates, candidates in the form of places with arcs to and from these transitions, are constructed from which the machine learning model ought to choose a subset from. It learns to identify when which candidates are appropriate from event log structures. The model is trained on automatically generated synthetic data, after which it is applied to and evaluated on real-life data.

The data preparation process is described in Chapter 7. Then the actual machine learning model is designed, of which the main component is the graph convolutional neural network of which the workings are elaborated on in Chapter 4. Our method solves the problem by sequentially choosing candidates to end up with a process model that the machine learning model is most certain of. Exact details altogether with how the model is trained is explained in Chapter 8.

1.5 Findings

Our approach is evaluated both in terms of its performance with regard to the synthetic data as well as on real-life data comparing it to state-of-the-art methods. These evaluations can be found in Chapters 10 and 11 respectively and are summarized here.

By training a machine learning model on synthetic data which is clean and without noise, we showed that it is well able to learn patterns and generalize this for unseen datapoints. By evaluating on real-life data which is possibly unclean and noisy, we show that our method can even generalize to such data with fewer assumptions required. On this real-life data, it competes with the current state-of-the-art algorithmic methods in terms of conformance metrics, which are used to measure how well a process model fits a given event log, further explained in Chapter 3.

A qualitative analysis shows that our method has potential to overcome problems in algorithmic approaches caused by assumptions on structure and relevance of infrequent data. On the other hand, our approach still has limitations in patterns that it is able to discover.

Altogether, this thesis opens up the field for other researchers to see if machine learning methods can be further exploited in the field of process discovery and serves as inspiration.
Part II

Background
Chapter 2

Preliminaries

This chapter introduces notations that are used throughout this thesis. These preliminaries are key concepts of process mining and illuminate how the process discovery task is defined. Section 2.1 introduces what the data available entails: event logs. Then Section 2.2 introduces Petri nets as the chosen notation for process models. Section 2.3 touches upon Alpha relations relating elements from the event logs and Petri nets. The chapter ends with another notation for process models called process trees in Section 2.4.

2.1 Event logs

Data about process executions is recorded in business information management systems [11]. Consider a single process within an company, these process executions consist of multiple activities that occurred in a specific order and could differ across cases of the process. The executions of the activities in the process are recorded as events, and a single process instance consisting of various, possibly duplicate activities, is called a trace. Important is that a trace has an order and could be different or equal across process executions. Generally, these events are recorded in the systems together with an identifier of the process instance and a timestamp such that they can be correctly extracted. In the simplest form, this is all the data covers and could be sufficient to discover the corresponding process model. Having a table with all executed events with their timestamps and corresponding identifier, each process instance can be converted to the traces. \( \langle a, b, c \rangle \) is a simple example trace where the events \( a \), \( b \) and \( c \) occurred in order. Note that here, the identifier and timestamps can be abstracted from and only the order is important. An event log contains multiple such traces denoting all process executions and could contain duplicates because of the abstractions, hence it is a multiset of traces. An example event log is \( \{(a, b, c)^2, \langle a, c \rangle \} \) where the traces \( \langle a, b, c \rangle \) and \( \langle a, c \rangle \) occurred twice and once respectively.

Furthermore, the set of distinct activities in the event log is important throughout this thesis and is called the alphabet of log \( L \), denoted as \( \Sigma(L) \). The set of distinct traces is both denoted as the trace variants and the log’s behavior \( \mathcal{B}(L) \).

Note that in information management systems, generally much more information about the activities in the process executions are recorded. Examples are start time,
end time, resources used or other properties such as costs. All this information could be utilized for the process discovery task but it not consistent throughout all datasets. The actual firing of events are always present and therefore only used in our approach.

Table 2.1 shows an example event log of a simple product ordering process, where a product is ordered by a customer, after which it is prepared and sent. During the preparation, the customer can complete the payment. The process ends with the customer receiving the product. Possible trace variants for this process are: \[\langle O, P, PP, S, R \rangle^{15}, \langle O, PP, P, S, R \rangle^{10}, \langle O, PP, S, P, R \rangle^{1}\], where the letters correspond to the first letter(s) of the activities.

Note that event logs are finite and could possibly not be complete, i.e. contain all possible traces from the process. Furthermore, it could contain deviations or noise, which is recorded behavior that is faulty either due to the recording system or a process execution itself. Whether these deviations should be contained in the process model is a question and case specific. A possible example of such deviation is in the trace variants shown above, where in almost all cases the payment is done before sending the product, however, one trace instance shows that payment is done after sending. Whether this behavior is actually part of the process but just does not happen often or it was a one time mistake by an employee is not known.

2.2 Petri nets

The task of process discovery entails the construction of a process model corresponding to the event log as described in Section 2.1 modeling the structural concepts as described in Section 2.3. In order to express such a process model, various modeling languages exist of which Petri nets is one [25].

A Petri net, of which an example is shown in Figure 2.1 is a bipartite, directed graph over places and labeled or unlabeled transitions, respectively the circular
and square nodes. The labeled transitions correspond to the events found in the event log, where the label equals the event’s activity name. The places connect all transitions together. In a bipartite graph, nodes of the same type can never be connected.

A marking is the notion of having tokens at certain places, where the initial and final markings show at which place(s) each process execution should start and end respectively. The initial marking is denoted as the green place in a Petri net whereas the final marking is denoted as a double lined orange place.

The tokens in places determine which transitions are enabled. Namely, a transition is enabled when all its incoming places contain a token. An enabled transition is ready for firing, where it consumes the tokens on all its incoming places and produces a token on each of its outgoing places, possibly enabling other transitions. The firing of a transition is denoted as an event in the trace, which is a complete run through the Petri net.

Workflow nets are a subclass of Petri nets only considering runs from the initial marking to the final marking. Each possible run adhering to this constraint is a trace variant allowed by the Workflow net. The set of all these trace variants is denoted as the Petri net’s behavior \( B(M) \). A trace can therefore be acquired from a firing sequence of the labeled transitions in the Petri net by taking the activity names and putting them in the corresponding order.

The Petri net from Figure 2.1 shows the same process as our previous example that can produce the event log \( L = \{\langle a, b, c, d \rangle, \langle a, c, b, d \rangle, \langle a, b, c, e \rangle, \langle a, c, b, e \rangle \} \).

Let us introduce four key structural concepts in processes: sequence, choice, parallelism, cycle. These are the four basic workflow patterns which capture aspects of process control [38], and are further explained with corresponding minimal example Petri nets below.

• A sequence of events is what was shown in the example trace in Section 2.1, namely the sequence of events \( a, b \) and then \( c \), denoting their subsequent nature in the event log. This is illustrated in Figure 2.2a, where a token starting at the initial marking has only one way of reaching the final marking;

• The choice construct is also shown in Figure 2.2b, which has a split at the first place, denoting a choice between either \( a \) or \( b \). A place with multiple outgoing arrows denotes a choice. When a token would reside in this place, it can be absorbed by one of the connected outgoing transitions. The event log corresponding to this Petri net is \( L_{\text{choice}} = \{\langle a \rangle, \langle b \rangle\} \);  

• Parallelism is where multiple activities happen concurrently meaning that the events could occur in any order. Consider the event log \( L_{\text{parallel}} = \{\langle a, b, c, d \rangle, \langle a, c, b, d \rangle\} \) and the corresponding Petri net in Figure 2.2c. Here the split is at a transition, namely \( a \), meaning that when \( a \) is fired, it produces a token in each of its outgoing places, enabling both the \( b \) and \( c \) transition. Event \( d \) can then only fire when its two incoming places contain a token;

• The cycle construct is when an event or a sequence of events occur multiple times. This construct is shown in Figure 2.2d and is similar to the choice but here transition \( b \) loops back to a place that was already visited. An event log that could be produced by this Petri net is \( L_{\text{cycle}} = \{\langle a, b, c \rangle, \langle a, c \rangle, \langle a, b, a, c \rangle\} \).
A last structural concept of Petri nets, regarding unlabeled transitions is shown in Figure 2.2e. This black transition is called a silent transitions, sometimes also denoted as a $\tau$ (tau) transition. This is a transition that can be fired in the process model, but is not found in the event log. The event log $L_{\text{silent transition}} = \{\langle a, b, c \rangle, \langle a, c \rangle\}$, could not be modeled by a Petri net without using a silent transition.

### 2.3 Alpha relations

Much information about the behavioral relations between events is already explicitly present in the event logs which is exploited by many process discovery algorithms including ours.

The so called Alpha relations address the structural concepts introduced in Section 2.2 and enables to find them explicitly in the event log. The main component for this is the directly follows relations, denoting whether an event is somewhere in the event log followed (directly) by another event, i.e. $\langle\ldots, a, b, \ldots\rangle$. In our example, the directly relations are $a > b$, $b > c$ and $a > c$. These directly follows relations can then be used to derive other relations relating to the structural concepts described above. For example, when both relations $a > b$, $b > a$ are present, the log suggest there exists either a parallel or a cycle relation between events $a$ and $b$. Furthermore, when only having $a > b$ suggests a simple sequence relation between $a$ and $b$. Lastly, having both $a > b$ and $a > c$, without both $b > c$ and $c > b$, suggest a choice after event $a$ between $b$ and $c$. The precise definition of the Alpha relations, taken from \[36\], is as follows:

**Definition 2.3.1.** (Alpha relations) Let $L$ be a log and $\Sigma(L)$ the distinct events therein. Let $a, b \in \Sigma(L)$:

- directly follows relation: $a >_L b \iff \exists T = \langle t_1, t_2, \ldots, t_n \rangle \land 1 \leq i \leq n - 1 : T \in L, t_i = a, t_{i+1} = b$;
eventually follows relation: $a >^f_L b \iff \exists T = \langle t_1 t_2 t_3 \ldots t_n \rangle \land 1 \leq i \leq n - 1 : T \in L, t_i = a, t_{i+f} = b$;
causal relation: $a \rightarrow_L b \iff a >_L b \land b \not\in_L a$;
no relation: $a#_L b \iff a \not\in_L b \land b \not\in_L a$;
parallel relation: $a\parallel_L b \iff a >_L b \land b >_L a$;
cycle relation: $\bigcirc_L a \iff \exists T = \langle t_1 t_2 t_3 \ldots t_n \rangle \land 1 \leq i,j \leq n - 1, i < j : T \in L, t_i = a, t_j = a$.

These relations can be easily expressed in a matrix. An example, with the event log $L = \{\langle a, b, c, d \rangle, \langle a, c, b, d \rangle, \langle a, b, c, e \rangle, \langle a, c, b, e \rangle\}$ is shown in Table 2.2.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>#</td>
<td>→</td>
<td>#</td>
<td>#</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>#</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>→</td>
<td>→</td>
</tr>
<tr>
<td>d</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>e</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
</tbody>
</table>

Table 2.2: Example Alpha relations matrix.

Important to note is that the Alpha relations may not actually hold for the process, since the event log can not be assumed to be complete, hence other relations could exist that break structural concept of the relations.

2.4 Process trees

A different notation for process models is the process tree, which models a process as a tree with the transitions as leaf nodes and the internal nodes denote what substructure, from the ones shown in Figure 2.2 is applicable. Process trees can be converted to Petri nets by inserting these substructures as they are denoted in Figure 2.2 and placing the leaf nodes within them. An example process tree is shown in Figure 2.3 and has the same behavior as the Petri net in Figure 2.1. The substructures sequence, choice, parallelism and cycle are denoted respectively as the symbols $\rightarrow, \times, \land, \bigcirc$ in the internal nodes of the process tree.

The expressiveness of process tree is a subset of that of Petri nets, since the resulting Petri net is always block-structured, meaning that a substructure can be replaced with a single transition having a single incoming and outgoing arc, without making it structurally invalid.
Chapter 3

Process discovery

In this chapter, we dive into the process discovery task to elaborate on its objectives and difficulties. The objectives are introduced in Section 3.1 and the corresponding quality criteria are elaborated on in Section 3.2. Here, also the difficulties come to light. The chapter ends with Section 3.3 by going into detail on current state-of-the-art methods for process discovery focusing on their algorithmic approaches and what their drawbacks are to serve as inspiration for introducing machine learning to the problem that could potentially overcome these drawbacks.

The event log $L_r = \{\langle a, b, c, d \rangle, \langle a, c, b, d \rangle, \langle a, b, c, e \rangle, \langle a, c, b, e \rangle\}$ will be used corresponding to the process model shown in Figure 2.1 throughout this chapter.

3.1 Objectives

Process discovery aims to find a fitting process model for the event data recorded. The primary property of the discovered model is that it should describe the possible behavior for all process executions providing insights in how the process actually behaves. This could be valuable in various ways:

- It offers an interpretable model to see how the process behaves in real life opposed to how it should behave according to the managers, employees or customers;
- It can be used as a reference or documentation for implementing the process in for example an information system for further automation;
- The combination of the event log and the process model enables analysis for optimization of the process either in quality or efficiency.

Since process discovery is the first step in the loop of the business process management lifecycle and directly influences the quality of the subsequent steps as shown in Figure 1.1, its effectiveness is crucial [9].

Overall, the task of process discovery is that with a given event log $L$ containing executions from a process in a system $S$, return a process model $M$ such that $M$ describes $S$ as good as possible based on $L$. The quality criteria determining what as good as possible encompasses are elaborated on in the next section.
3.2 Quality criteria

The five main quality criteria regard fitness, precision, generalization, simplicity and soundness and are elaborated on below. Of these five, the first three consider both the process model and the corresponding log whereas the latter two only consider properties of the process model itself. We will not go into full detail on how these quality measures are exactly computed.

3.2.1 Fitness

The notion of fitness is whether the process model fits the corresponding event log. In other words, is the process model able to replay the traces in the event log? This is the most obvious quality criterion, since the objective of process discovery is to find a process model that represents the event log.

When considering an event log $L$ and its behavior $B(L)$, ideally we want the discovered process model $M$ to fully contain this behavior, i.e. $B(L) \subseteq B(M)$. To get a value for this measurement, fitness is in general the fraction of the log’s behavior that is also allowed by the model \cite{7}:

$$\text{fitness}(L, M) = \frac{|B(L) \cap B(M)|}{|B(L)|} \quad (3.1)$$

Then, when all the log’s behavior is allowed by the model, i.e. $B(L) \subseteq B(M)$, fitness equals 1. On the other hand, when none of the log’s behavior is allowed by the model, i.e. $B(L) \cap B(M) = \emptyset$, fitness equals 0. The objective of process discovery is to achieve a fitness score as close to 1 as possible.

3.2.2 Precision

Precision is the opposite of fitness in the sense that it looks at what behavior is allowed by the process model but not seen in the event log. So now, when considering a process model $M$ and its behavior $B(M)$, ideally we want the event log to fully contain this behavior, i.e. $B(M) \subseteq B(L)$. Similar to the computation of fitness, in the general sense, precision is computed as follows \cite{7}:

$$\text{precision}(L, M) = \frac{|B(L) \cap B(M)|}{|B(M)|} \quad (3.2)$$

Having low precision, a process model $M$ is underfitting on the event log and does not therefore necessarily describe the behavior of the specific process \cite{37}. This makes precision an important quality criterion for process discovery.

3.2.3 Fitness vs precision

The so called flower model as shown in Figure 3.1 is a clear example of a process model with high fitness that is lacking precision. This model always ensures perfect fitness on any event log with the included transitions, since any combination of the
events from $a$ to $e$ are allowed by this model. However, since any event log fits the control-flow of a specific process is not described by this process model.

The notion of fitness and precision is illustrated in Figure 3.2, where the behaviors of the event log $L$, the actual system $S$ and the discovered model $M$ are denoted as the circles in the Venn diagram. The overlap (purple area) is the fitted behavior whereas the red area is log’s behavior not captured by the model and the blue area is behavior from the model not found in the event log.

An ideal model would have both high fitness and high precision, since then the process model perfectly describes the event log: $B(L) = B(M)$. This is where the event log’s and the model’s areas from the Venn diagram in Figure 3.2 perfectly overlap completely.

This suggests that the, also very simple to construct, process model denoted as the trace model as shown in Figure 3.3 would be ideal. Note that here, we do allow for transitions to have duplicates, which is not uncommon. Here each distinct trace variant is modeled as a sequence of transitions. It is obvious to see that this results in perfect fitness and precision. However, this model is overfitting on the event log, meaning that it shows exactly the behavior found in the event log without generalizing beyond it to show the actual control-flow logic of the process [37]. Both from the flower model and this model, one can not immediately see the parallel relation between $b$ and $c$ and the exclusive choice between $d$ and $e$.

### 3.2.4 Generalization

Generalization regards the notion of overfitting on event logs. It is more difficult to measure than fitness and precision, however, since it requires knowledge on the behavior of the actual process itself, also called the system. As already touched upon
in Section 2.1 event logs are finite and could therefore possibly exclude behavior from the actual system $B(S)$. Consider a process with ten activities that occur in parallel, then $10! = 3628800$ different orderings are present. It can not be assumed that all these possibilities are present in the event log and ideally we still want the discovered process to capture this information.

Generalization therefore measures to what degree the process model generalizes beyond the event log to show what actually happens in the process. A process model $M$ generalizes a log $L$ with respect to the system $S$ by capturing behavior from the system that is not captured by the event log:

\[
generalization(S, L, M) = \frac{|(B(S) \setminus B(L)) \cap B(M)|}{|B(S) \setminus B(L)|} \tag{3.3}\]

This is illustrated in the Venn diagram when we add the system’s behavior to it in Figure 3.4, where we see that the behavior of the log and the system are not necessarily exactly the same. Ideally, we would want the model’s (blue) area to exactly overlap with the system’s (yellow) area.

The trace model from Figure 3.3 achieves perfect fitness and precision, however, because it is overfitting on the event log, it is lacking generalization.

With the example Petri net from Figure 2.1 we show that two other Petri nets could replay the example event log, besides the flower model and the trace model. These are shown in Figure 3.5. The next question is then which one of these should be chosen, which is discussed below. Note that the left model is not precise since every combination of firings is again possible, albeit without repetition like in the flower model.
3.2.5 Simplicity

Another important quality measure is simplicity, since the process models are intended to be interpreted by humans; the ones that take on the subsequent steps in the BPM lifecycle after process discovery, from Figure 1.1. We saw that the process model should describe the process’ control-flow logic to match reality to analyze and optimize the process. Moreover, it is used to show the stakeholders, possibly without any technical background, in a clear and understandable way how the process is running right now. Therefore, the process model should be as simple as possible and when multiple solutions are present, according to the Occam’s Razor principle, the simpler solution is preferred over the complex one.

Simplicity is a difficult property to measure, however, since it is generally a subjective matter. Nevertheless, metrics have been proposed to measure the simplicity of a process model. A few are taken from [7] and are listed below:

- size: number of nodes and arcs;
- diameter: length of the longest path from a start to and end node;
- density: relation between the number of arcs and the maximum number of arcs between all nodes;
- connectivity: ratio between the number of arcs and the number of nodes;
- node degree: average or maximum number of nodes each node is connected to;
- separability: ratio between the number of cut-vertices (i.e., the minimal number of vertices needed to disconnect the graph) to the number of nodes;
- sequentiality: number of arcs to/from places and transitions;
- structuredness: the proportion or well-structured parts with respect to the non-structured parts in the process model;
- depth: average or maximum depth for split/join constructs;
- gateway mismatch: absolute sum of all input arcs minus output arcs over all split/join constructs.

3.2.6 Soundness

A single process execution simulated by a workflow net, as described in Section 2.2, starts with a single token in the initial place and ends with a single token in the final place after firing a particular set of transitions. After doing so, no tokens are left in any of the places other than the final marking. This should be true for all
possible paths in the process model from the initial marking to the final marking. When this is not the case, the model is not sound. An unsound model implies that there are deadlocks or improper split and join constructs. The former causes the execution to not reach the final marking and the latter could cause tokens to be left in non-final places after the final marking is reached or cause transitions to never be enabled. A minimal example of the last case is shown in Figure 3.6, where the model start by a choice and finishes with a parallelism. One could see that c is never enabled since only one of a and b can be fired.

Soundness is an important aspect in process discovery since unsound models might introduce ambiguity even though it could be useful for human interpretation. Recall from Section 2.2 that a firing sequence in a workflow net, which we are considering, starts and ends at the initial and final marking of the model. In an unsound model, this may not be true, i.e. tokens could be left on the non-final places, causing the language of the model to be not well-defined. This makes it impossible to reason about the behavioral model qualities like fitness, precision and generalization.

All in all, the problem of process discovery entails that we want to construct a process model $M$ in the form of a Petri net from a given, possibly incomplete, event log $L$ recorded from a process $S$. Process model $M$ should balances between fitness, precision with regard to event log $L$ to properly describe how the process $S$ behaves in reality, while keeping the process model as simple and easy to understand as possible. Ideally, the produced process model is also enforced to be sound.

### 3.3 State of the art

This section lists the existing state of the art process discovery algorithms and discusses their approaches to these quality measures together with their shortcomings. The section ends with a discussion on why the algorithmic approaches fail solve the process discovery problem optimally.

#### 3.3.1 Heuristics Miner

The Heuristics Miner, as proposed in [43], starts by looking at the directly follows abstraction called the dependency graph. This dependency graph is based on the Alpha relations as discussed in Section 2.3, where a graph is constructed from the first relation in Definition 2.3.1. So if the directly follows relations $a \succ_L b$ holds, a directed edge from $a$ to $b$ is added. These dependencies are then filtered using a frequency based metric. A dependency between two events is taken into account
when \( \frac{|a >_L b| - |b >_L a|}{|a >_L b| + |b >_L a| + 1} > \theta \), i.e. when \( a >_L b \) occurs more frequently than \( b >_L a \) with threshold parameter \( \theta \).

Instead of finding Petri nets as process models, the heuristics miner discovers causal nets \([35]\), which can then be converted to Petri nets. The resulting Petri net is not guaranteed to be sound, since the split and join constructs are considered only locally. Furthermore, it often has spaghetti-like structure for larger processes making it difficult for interpretation. An advantage is that it is robust against smaller variations in the data, because only frequent dependencies are considered.

### 3.3.2 Inductive Miner

The dependency graph is also the starting point for the Inductive Miner \([19]\). Rather than a causal net, the Inductive Miner aims at extracting a process tree from the event log. This assumes that the resulting process model is block-structured. It operates in a recursive manner starting at the top-level block, trying to find the suiting footprint of the operator (sequence, choice, parallelism, cycle) by analyzing the directly-follows graph. Then it splits the log according to this operator into one log per sub-block of the footprint, which are the children nodes in the process tree. This recursive process ends when logs of single transitions are found, which are the leafs of the process tree. When no footprint of the operator is found, the Inductive Miner inserts the flower model, from Figure 3.1 on the (sub)log to ensure fitness. Constructing process trees which are block-structured offers high interpretability and ensures soundness.

However, in reality, processes are often not inherently block-structured, and therefore, leading to either decreased fitness or precision. High fitness can often be assured at the expense of using silent transitions causing a decrease in simplicity.

Generalization is similar to the Heuristics Miner again based on the frequencies of dependencies.

### 3.3.3 Split Miner

The Split Miner, as proposed in \([2]\), is based on the methods used in the Heuristics Miner and tries to mitigate its weaknesses by introducing intermediate steps. It again starts with the directly follows graphs (DFGs), but instead of immediately filtering it, it first discovers loops, which are known to cause problems otherwise when detecting concurrency. Using the DFG, first parallel relations between activities are discovered after which the DFG is filtered removing infrequent edges such that the fitness will be maximized. The Split Miner then converts this filtered DFG into a BPMN model (Business Process Model and Notation), which is another graphical notation for process models of which the details can be found in \([42]\). Initially, the BPMN model has no gateways representing constructs such as choices and parallelisms. The discovery of the gateways is the next step ensuring that each activity in the BPMN model has exactly one incoming and one outgoing edge. This is actually done in three steps, again by looking at the Alpha relations, first discovering the splits and then the joins ending with minimizing the number of OR-joins which increases simplicity.
Figure 3.7: Complex control-flow Petri net from log \( L = \{ \langle a, c, d \rangle, \langle b, c, e \rangle \} \).

Note that the filtering is again based on relative frequencies and therefore making, possibly false, assumptions on what data is relevant.

The Split Miner succeeds in achieving high fitness and precision, reaching a sweet spot between them, however, it often produces complex models making them difficult to interpret.

### 3.3.4 ILP Miner

The ILP Miner uses Integer Linear Programming to find the places that make up the Petri net and uses language-based region theory to do so [39]. Here a Petri net is derived from a language such that the resulting net has the smallest behavior possible while still containing the language. It starts with a Petri net containing all transitions, but no places. Then places are chosen that do not prohibit any trace from the language while maximally limiting behavior. This is solved by translating the problem to a Integer Linear Programming problem for each new place. ILP solves a set of linear equations by either maximizing or minimizing a variable. The set of linear equations are given by the abstracted state space and the variable is the place. Then the optimal place is found while still adhering to the equations maximally limiting behavior. The process is sequential, where each step a new place is chosen. This place is then added as an additional constraint on the possible behavior of the Petri net.

The ILP Miner can discover more complex control-flow structures, of which an example is shown in Figure 3.7 (the place labelings can be ignored for now). It can guarantee perfect fitness and, for its representational bias, best precision with the consequence of a decreased generalization. Furthermore, the discovered Petri net is not guaranteed to be sound, since the places are found individually and locally, not looking at the global structure resulting in improper split and join constructs. Lastly, the ILP algorithm is in the worst case exponential in the number of events in the log, although in reality it is often much faster.

A method similar to this is proposed in [24], where the places are chosen that define the minimal language containing the input language, while using a minimal number of places to do so. Here, the abstraction of the state space as in the ILP Miner’s approach is skipped, meaning that more places can be discovered increasing the expressiveness. Rather than choosing the places based on an ILP problem, they are chosen by exploiting token-based replay to determine which place has the highest fitness. Many places are pruned in the process to increase efficiency, although the approach is still computationally expensive.
Our approach is also based on the idea of discovering places that glue the transitions together, where we have all possible places as candidates and choose a subset of them for the final Petri net. Using machine learning, our approach can look at global structure to make decision locally overcoming the disadvantage of the ILP Miner looking only at local structures.

To end this chapter, the assumptions within the algorithmic approaches are summarized and are shown in Table 3.1. As discussed before, methods exist that optimize for certain quality criteria. The difficulty, however, lies in the balancing between these criteria, since we do not know the optimal balancing point. In other words, we do not know when we arrive at the right process model. The ILP Miner has a smart idea to add places as constraints to the model, but does so locally by adding individual constraints causing the model to not be meaningful globally. The Heuristics, Inductive and Split Miner do filter and have some form of global view, but the filtering heuristics in the algorithms are based on assumptions about the data. When the data violates these assumptions, relevant information is possibly filtered out. Machine learning could do better by looking globally without making assumptions on the data, but rather on context and experience.

<table>
<thead>
<tr>
<th></th>
<th>representational bias</th>
<th>noise handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>HM</td>
<td>one transition per activity</td>
<td>infrequent</td>
</tr>
<tr>
<td>IM</td>
<td>one transition per activity</td>
<td>infrequent</td>
</tr>
<tr>
<td></td>
<td>block-structured models</td>
<td></td>
</tr>
<tr>
<td>SM</td>
<td>one transition per activity</td>
<td>infrequent</td>
</tr>
<tr>
<td>ILP</td>
<td>one transition per activity</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>local decisions</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Summary of assumptions in state-of-the-art methods.
Chapter 4

Related work

In this chapter we take a side step from the process discovery problem which is needed because we are using machine learning to tackle the problem. Since machine learning has not been used for process discovery as of yet, we will look at other research fields which have tackled problems using machine learning that could be translated to our problem. The main overlap between the fields and process discovery is that the input or output data is graph structured. Other similarities make it possible to translate the problems to process discovery and potentially use the same techniques to solve it.

4.1 Other fields

4.1.1 Medicine discovery

Medicines in chemistry are molecules with certain properties. A molecule is a group of two or more atoms held together by chemical bonds as a graph where the atoms are the nodes and the bonds are the edges. There are certain limitations to how these atoms can connect and the properties of the molecule can be computed when it’s complete. This way, neural networks are used to generate molecules with hopefully the desired medical properties. This method has proved to be very promising to speed up the discovery process of new medicines \cite{8, 14, 23, 30, 46}. The translation to the problem of process discovery is rather straightforward, since in both cases graphs have to be generated with certain properties. During the generation, certain rules must be obeyed to, e.g. the places where edges are allowed. When the full graph is generated, the final score can be computed, here the molecules’ properties and in our case the conformance metrics.

4.1.2 Natural language processing

Making sense of language is a long time popular field and has recently introduced graph neural networks as well. Since sentences can be transformed into graphs where the words are nodes and their relations are represented by the edges. Text generation is a challenging task since there are many grammatical rules to obey
and long-range dependencies between words. Many approaches use graphs to encode text to eventually generate new text by decoding it [4, 18, 21, 32, 34, 45]. This is then used to generate for example comments on articles or reviews on products. The core concepts of the approaches include recurrent neural networks with gating/attention mechanism since sentences are sequential and related words could be far apart. Here, also graph convolutional networks are used for predicting syntactic dependency trees [4]. Furthermore, the graph representation of text is often called the knowledge graph.

The problem of text generation translates to process discovery since the event log traces relate to sentences and the process model relates to the encoding graph. Sentences and event log traces have similar properties as which events and words may be followed by which events and words. Furthermore, events in traces may have long-range dependencies just as words in sentences. The idea of the process model to be discovered is to be able to generate event log traces similar to the recorded ones by for example simulation. This relates to how the graph encodings from the mentioned NLP research aim at generating sentences similar to the training sentences. The difference is that in process discovery the graph’s main purpose is to better understand the process rather than generating new event logs, while in NLP the interpretation of the knowledge graph is not the priority.

4.1.3 Network analysis

The first thing that comes to mind when thinking about graph structured data is presumably social networks. These are graphs where people are the nodes and they are connected by some kind of property, e.g. being friends on Facebook. Finding clusters, node classification and predicting relations are widely studied problems in this domain [20, 47]. The latter two are related to process discovery since they could respectively be used to predict edges between transitions and places in the petrinet and predict to what type of structure a transition or subtrace is part of. Both papers use graph neural networks to encode the graph structured data to either classify nodes or edges for link prediction.

4.1.4 Electric circuit design

Designing electrical circuits is defined by connecting certain circuit components in a way that the electrical flow is valid and the lines connecting the components have minimal overlap. Existing methods based on domain knowledge and heuristics are successful only for certain types of circuits, hence machine learning in the form of generative adversarial networks are applied [12]. Similarities to process discovery is that a electric circuit can be viewed as a graph and in both situations, this graph has to be generated having certain specified properties and optimizing certain properties. The paper [12] does not use graph neural networks but instead works solely on the adjacency matrix. This can potentially be improved by graph neural networks since they capture the graph data much better than classical neural networks do on the adjacency matrix.
4.1.5 Code abstraction

Source code is similar to natural language in the sense that it has text with words which are connected in some way. However, source code is much more formal and the syntax is more restricted. The connectedness of variables in the source code is determined by the rules of the programming language and can be used to generate a representing graph. This is beneficial over natural language processing techniques since it captures the long-range dependencies better [1, 6]. A gated graph neural network is then used to perform tasks like variable naming and checking misuse of variables. Similar to the translation from NLP to process discovery, again a graph is constructed by connecting related events in process discovery and variables in code abstraction. The way they are connected in the methods used for code abstraction could be leveraged to create useful graph embeddings for process discovery.

4.2 Techniques used

This section describes the different techniques that are used in the approaches of the mentioned papers. The individual techniques are elaborated on rather than the complete approach, since these will be most useful for our approach. The complete approaches are mainly combinations of these individual techniques and are specifically designed for the particular tasks.

4.2.1 Graph convolutional neural network

An important concept mentioned in nearly every paper and used as a basis for most methods is the graph convolutional neural network and it is used to get a node, edge and/or full graph embedding of the input data. This technique is actually used in our approach and is therefore more elaborately explained here.

Graph convolutional neural networks (GCNs) are developed such that they take graph structured data as input and then make decisions on either the full graph or on the individual nodes [17]. It is similar to the 2-dimensional convolutional neural network (CNN) in how it re-uses its parameters to process clusters of vectors to compute an information package on. The clusters of vectors being spatial clusters of pixels for the CNN and clusters of nodes consisting of the nodes’ neighborhoods for the GCN.

It is best explained by the a package delivery system where each node holds a package and for each layer, all nodes broadcast their package over their outgoing edges and receive all packages over their incoming edges. A two layer GCN is illustrated in Figure 4.1 where in the first drawing, the red node in the center broadcasts its information to its neighborhood up to two hops away. The middle drawing shows how the messages from the neighborhood are aggregated to the central red node. Note that the messages are the current hidden states of the nodes denoted by the vectors at each node. The messages flow through two neural networks before arriving at the destination, where one neural network processes the messages from the blue nodes one hop away and another processes the messages from the green nodes, two hops away. To pooling method like summing or averaging
Figure 4.1: Illustration of the workings of a GCN, taken from [13].

is performed to handle multiple packages arriving at a node. The drawing on the right illustrates that a final label is predicted for the red node. During a single forward pass of the network, this process is done for each node simultaneously. In the last layer, the network can decide on for example a classification vector for each node if this is the task. Another task could be graph classification where each node’s hidden state is pooled in some way to get a graph vector which can be processed by another neural network.

No particular restriction holds on the neural networks processing the hidden states. For example, fully connected dense layers or recurrent neural networks could be used here. In the following, we assume that the type of neural network used consists of fully connected dense layers, since this is the most straightforward.

Mathematically the graph convolutional neural network operates as follows: Each node has a feature vector initially, denoted as $h^{(0)}_i$ for node $v_i$, which is also called the node’s hidden state after the first layer. The GCN consists of a number of dense layers which is equal to how many hops are performed through the graph. In our example from Figure 4.1 there are two such layers, the blue one and the green one. A dense layer consists of a learnable weight matrix $W$ of size $R \times C$ equal to the input and output size of the nodes’ hidden states. After being processed by a neural network, this weight matrix $W$ is used manipulate the hidden state.

For the dense layer at depth $l$, the updated hidden state of node $v_i$ is:

$$h^{(l+1)}_i = \sigma \left( \sum_{j \in \mathcal{N}(i)} \frac{1}{c_{ij}} h^{(l)}_j W^{(l)} \right)$$  \hspace{1cm} (4.1)

Where $\mathcal{N}(i)$ denotes the incoming neighbor node of $v_i$. So $h^{(l)}_j$ is the package that is delivered to $v_i$ manipulated by $W^{(l)}$. The pooling method here is averaging, done by $\frac{1}{c_{ij}}$ where $c_{ij} = \sqrt{|\mathcal{N}_i|} \sqrt{|\mathcal{N}_j|}$. Note that when the hidden state in layer $l$ has size $R$ and $W$ has the shape $R \times C$, the next hidden state (in layer $l + 1$) has size $C$.

The $\sigma$ is an activation function, ReLU in [17], which is used to help the network learn complex patterns in the data.

Recall that the computation is done for each node simultaneously in each layer, where Equation 4.1 only computes the next hidden state of node $v_i$. In order to do
this computation efficiently for all nodes at once, a clever matrix multiplication is designed:

$$H^{(l+1)} = \sigma \left( \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)} \right)$$  \hspace{1cm} (4.2)$$

$H^{(l)}$ is the matrix where the rows correspond to the nodes’ hidden state, so its size is $n \times C$ with $n$ nodes and the hidden states’ size $C$. $\tilde{A}$ is the binary $n \times n$ adjacency matrix of the graph where $A_{ij} = 1$ means there exists an edge from node $v_i$ to $v_j$ and $A_{ij} = 0$ means the contrary. This essentially replaces the $\mathcal{N}(i)$ in Equation 4.1. $\tilde{D}$ the $n \times n$ normalized degree matrix with all zeroes except its diagonal. In the diagonal $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$, which equals the number of incoming edges, i.e. the degree, for node $v_i$. $\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$ can be computed beforehand and does not change over time. It is used to replace the $\frac{1}{c_{ij}}$ in Equation 4.1. Note that $H^{(l)} W^{(l)}$ operates exactly the same as $h^{(l)}_j W^{(l)}$ but does so for all nodes simultaneously. Each row in $H^{(l+1)}$ corresponds to the nodes updated hidden states and has size $n \times F$ with $W^{(l)}$ being of size $C \times F$.

### 4.2.2 Graph attention network

The way graph convolutional networks aggregate is structure-dependent hurting the generalizability. In [13], this problem is solved by simply averaging over all neighbor node features. Another type of aggregation, as described in [41], is to use a graph attention network which uses weighted neighbor features with feature dependent and structure-free normalization with attention. This technique is also extensively elaborated on here, since it is used in our approach.

When dealing with sequence-based tasks, it has become very common to add an attention mechanism to the machine learning model. An attention mechanism is used to put weight on certain parts of the input data. The best example that illustrates this is in the task of text translation, where a sequence of words is not the same in different languages. For example ‘The European economic area’ translates to ‘La zone économique européenne’ in French. It is obvious that the words in the translation are reversed when you translate each word individually. So when choosing ‘zone’, the attention is actually at ‘area’ in the input data, skipping ‘European’ and ‘economic’. A sequence to sequence (seq2seq) model with attention learns to create a context vector to do so by creating a mix of the input representations at each time step. The weights of this mix are learned and referred to as the attentions \[33\].

In the graph convolutional neural network as described in Section 4.2.1, the weights of each neighbor’s feature information is equal. In graph data, this attention mechanism can be exploited to put more weight on the information packages of certain neighbors, where these weights are automatically tuned during the learning process. The difference compared to the standard GCN lies in the $\frac{1}{c_{ij}}$ term in Equation 4.1 which is replaced by a learnable $a_{ij}$ for each edge, as proposed in [11]. It works by computing some intermediate values.

$$e^{(l)}_{ij} = \text{LeakyReLU} \left( a^{(l)} \left( \left( h^{(l)}_i W^{(l)} \right) \left( h^{(l)}_j W^{(l)} \right) \right) \right)$$  \hspace{1cm} (4.3)$$
First, a pair-wise unnormalized attention score is computed between each two
neighbors. The nodes’ processed hidden states \((h^{(l)}_i W^{(l)})\) of the two neighbors are
concatenated, denoted as \(\|\) in Equation 4.3. \(\vec{a}^{(l)}\) is a learnable weight vector similar
to \(W^{(l)}\) with shape \(2F \times 1\) when corresponding \(W^{(l)}\) has shape \(C \times F\). Taking the
dot product of \(\vec{a}^{(l)}\) and the concatenated hidden states processes these latter similar
to how \(h^{(l)}_i W^{(l)}\) processes a single node’s hidden state. LEAKYRELU is the chosen
activation function, rather than \text{RELU} in the standard GCN from Section 4.2.1.

The result \(e^{(l)}_{ij}\) is the attention score for the edge from node \(v_i\) to \(v_j\).

\[
\alpha^{(l)}_{ij} = \text{SOFTMAX}_j(e_{ij}) = \frac{\exp(e^{(l)}_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e^{(l)}_{ik})} \tag{4.4}
\]

Before being able to replace \(\frac{1}{c_{ij}}\) with \(e_{ij}\) it has to be normalized first. This is done
using the \text{SOFTMAX} function as shown in Equation 4.4. Here \(\exp(x) = e^x\) is the
exponential function\(^1\). This ensures that for node \(v_i\) the attention weights on its
incoming edges sum to 1: \(\sum_{j \in \mathcal{N}(i)} \alpha^{(l)}_{ij} = 1\).

The new update function with attentions \(\alpha_{ij}\) replacing Equation 4.1 from the
standard GCN becomes:

\[
h^{(l+1)}_i = \sigma \left( \sum_{j \in \mathcal{N}(i)} \alpha^{(l)}_{ji} h^{(l)}_j W^{(l)} \right) \tag{4.5}
\]

Note how \(\alpha^{(l)}_{ji}\) now replaces \(\frac{1}{c_{ij}}\).

This form of attention is called self-attention and one such \(\alpha^{(l)}_{ij}\) is also denoted as
a single attention head of the \(l^{th}\) layer. To enrich the model capacity and stabilize
the learning process of the attention mechanism, \cite{41} extends the mechanism to
employ multi-head attention. Here \(K\) independent attention mechanisms execute
the computations from Equation 4.4 and their features are concatenated:

\[
h^{(l+1)}_i = \left\| \sum_{k=1}^{K} \sigma \left( \sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)k} h^{(l)k}_j W^{(l)k} \right) \right\|_1 \tag{4.6}
\]

where again the \(\|\) again denotes the concatenation of vectors. This simply
repeats the process from a single headed attention network layer \(K\) times and
concatenates the results.

### 4.2.3 Relational graph convolutional network

Graph convolutional networks, which focus mostly on local graph neighborhoods,
can be extended to large-scale relational data. The forward-pass update function of
the regular GCN is changed to incorporate relation-specific transformations depend-
ing on the type and direction of an edge \cite{27}. This can be used for node or graph
classification as well as for link prediction. It could potentially show improvements
for tasks where the graph structured data has various relational properties.

---

\(^1\) \(e\) denotes Euler’s number here.
4.2.4 Recurrent neural network

When dealing with sequential data, recurrent neural networks (RNNs) are rarely not used. They operate by propagating the current input and the processed previously seen input through units in the network. This way, the embedding of each input has the information of previously seen input implicitly in it. Different types of recurrent neural networks are long short-term memory (LSTM), gated recurrent unit (GRU) and bi-directional. The first two differ in the gating mechanism in the units, where GRU is the simpler one and LSTM also has forget gates making it potential of learning memories of many time steps earlier. Bi-directional RNNs are essentially two normal RNNs with either LSTM or GRU units where one has its data reversed. This way, instead of only gathering information about previous time steps, it also incorporates information from future events.

4.2.5 Reinforcement learning

Reinforcement learning in the general sense is that an agent takes actions in an environment interpreted into a reward and a representation of the state. These are fed back to the agent. Using this reward, the agent learns which actions to choose at which state. There are different reinforcement learning algorithms like the classic Q-learning one. However, the one to address here is proximal policy optimization (PPO) [29] in which a policy a learned by the network. A policy can be seen as the plan for the network to follow stating which actions to take at which specific states which can be observed. In reinforcement learning it is important that there is a balance between exploration and exploitation of current knowledge. This is achieved in a relatively easy (to implement) way in PPO. Each iteration, the advantage of the new policy with regard to the old policy is computed by the reward function and the network is updated accordingly.

4.2.6 Variational autoencoder

A variational autoencoder (VAE) is also a technique to generate data. They are all about finding the underlying (possibly complex) distribution from the data by the use of neural networks. Using the neural network to encode the input data, it reduces/transforms the underlying distribution to for example a standard normal distribution. This projected distribution is called the latent space and the aim for the neural network is to map the points from the complex distribution to the latent space accurately such that the process can be reversed. So to generate new data, we can sample from the simple standard normal distribution in the latent space and use again a neural network to decode this sample, mapping it onto the original distribution resulting in a newly generated datapoint. In [31], the authors propose a graph generation method using a variational autoencoder for molecule generation.
Part III

Approach
Chapter 5
General approach

This chapter introduces our approach by illustrating it as a roadmap in Section 5.1 after which each phase is elaborated on in the subsequent sections, also discussing what decisions have been taken to arrive at the final approach. Section 5.3 discusses the importance of (labeled) training data in machine learning and the lack thereof in the process discovery domain. Then, Section 5.4 discusses our approach to handling this data, preparing it for a machine learning model. In Section 5.4, the design of the actual machine learning model is explained after which its evaluation is described in Section 5.5.

5.1 Roadmap

As mentioned before, machine learning is used to tackle the process discovery problem, however this is not known to be done before. So a new approach is designed. Inspiration do to so is taken from other domains where problems using graph data are tackled using machine learning. These problems in other domains were discussed in Chapter 4 together with how they can be translated to the problem of process discovery.

Figure 5.1 illustrates a general roadmap of the taken approach. Each black rectangle depicts an action that is taken and each grey rectangle with rounded corners depicts the result of the respective preceding action. The roadmap consists of four phases:

- The first phase is that of synthetic data generation, needed for training the machine learning model;
- The second phase regards the preparation of the data. Since event logs and process models have complex structures which are not standard for machine learning, these will be encoded as single graph-structured datapoints;
- In the third phase, a machine learning based approach is designed and trained using the prepared synthetic data from the previous two phases;
- In the fourth and last phase, our approach with the trained machine learning model is evaluated on both the synthetic data as well as on real-life data to analyze its performance and to compare it to state-of-the-art methods.
5.2 Gathering training data

It has been touched upon before that labeled data is necessary for any of the chosen approaches to succeed. In general, a machine learning model needs to be fed with lots of data for it to acquire knowledge to achieve a certain goal. The basic workings of a neural network is that it receives a datapoint, does some black box computation and spits out an output. This output is then checked with the desired output and by comparing these, a loss value is decided on. The value is then used to update the inner parameters of the neural network in the black box part such that the next time it receives this particular datapoint as input, the output is a bit closer to the desired output. Only after it has received a large enough amount of datapoints, such a model can generalize to other, never seen before, datapoints as well.

Creating a dataset of realistic event logs and the ideal process model is a tedious, if not impossible task, since the ‘right’ model is not always known. On the other hand, we do know that when having a process model, a corresponding event can be generated by simulation. We also know that block-structured Petri nets are generally easier to interpret than non-block-structured ones, thus having higher simplicity. By exploiting these two facts, a dataset can be created in the form of process trees and then simulating those to generate event logs. This way, a fully automatic dataset generation method is created without the need to annotate anything by hand. The process of doing so is further elaborated on in Chapter [6]

An analysis on this synthetic data is done in Chapter [7] to evaluate how realistic it is by comparing patterns and structure in the synthetic event logs to those in event logs from real-life processes.

5.3 Handling the input data

The first step is actually deciding on how to handle the input data. The input data for process discovery consists of event logs, which in turn consist of traces, possibly
with additional features like frequency. In these traces, the distinct events have strong relations across all traces. Furthermore, the sequential information in the events’ order is of great importance for determining the process model. The first thing that comes to mind is to view the traces as sentences, like in natural language processing. Sentences are generally processed by recurrent neural networks (RNNs) since they tend to perform well on sequential data. However, in the case of event logs, the relation between traces is different from the relation between sentences in a document, since one trace does not necessarily follow one of the others. Therefore, regarding the full event log as a document of multiple sentences is not applicable.

As mentioned before, our approach uses graph convolutional neural networks, which are designed to process graph structured data. It makes sense to model the event log as a graph and process it by propagating information through nodes in this graph. Each event is modeled as a node in this graph. Separate nodes acting as transitions in a Petri net corresponding to the distinct activities in the event log are added to this graph. Information about the preceding and subsequent activities for every occurrence of an activity can now be gathered in the corresponding transition, without having relations between the traces. Inspired from the ILP Miner where places are discovered, we encode candidate places in this graph between the transitions which turns the task of process discovery problem into a node classification task. Chapter 7 goes into detail on how exactly the data is prepared to process the information from an event log to eventually generate a process model from.

Another method that was investigated used an RNN and concatenates all trace variants treating them as one long sentence, where the states of the events are outputted at each event rather than only at the end. The problem of traces having different relations than sentences in documents is mitigated by doing it this way. However, since the trace variants are concatenated, information from earlier traces is regarded when processing the later traces, which is undesired. More information on the workings of this method can be found in Appendix A.1.

5.4 Training the machine learning model

At the core of our approach is a graph convolutional neural network with an attention mechanism, explained in detail in Sections 4.2.1 and 4.2.2. This has to be trained in order to learn how to solve the the process discovery task. Our approach has two variants:

- A discriminative model where in one go, all places are classified to determine whether they should reside in the final Petri net;
- A sequential model where the existence of places is decided on one by one, where each decision is influenced by previous decisions.

As discussed before, the training data consists of event logs with corresponding Petri nets, which are encoded in graphs. The nodes in these graphs have initial features and by processing such a graph $G$ in a GCN, these are updated after some rounds of packet propagation. The nodes’ features at the end denote the determined classification. During training, we know how the candidate places should be classified, so the GCN’s inner parameters can be updated such that next time it sees graph $G$,
the determined classification is closer to the desired one. The complete design and workings of our approach are elaborated on in detail in Chapter 8. Exact details on the training process are also explained here.

Since our approach learns to make decisions based on data, we can safely say that it has fewer assumptions than the state-of-the-art algorithmic approaches. There is no assumption on the structure of the process models and no heuristic filtering is needed to decide on the relevance of infrequent behavior.

Besides using GCNs, other techniques from Section 4 have been investigated as well. Firstly, using reinforcement learning seemed suitable, since it can utilize the existing conformance metrics with the goal to maximize them. This is not the case for other techniques, since the score should be differentiable in order to update the neural network’s inner parameters which is not the case for the conformance metrics. An advantage of reinforcement learning is that it does not require labeled training data, since its goal is to maximize a certain score rather than learning to fit the training data. It is not used, however, since reinforcement learning being a black box provides little insight in how it learns to solve its task.

Another approach, the variational autoencoder, could be a fitting technique offering multiple solutions potentially focusing on different goals, like fitness, precision, simplicity, where the preference could be specified beforehand. A start was made using this technique, but we did not manage to successfully generalize over the data. Workings and theory of this partial approach using VAE can be found in Appendix A.2.

5.5 Evaluating the model

With a trained machine learning model, we evaluated it in two ways:

- Firstly on the training data to see how it performs on the task we actually gave it to learn. We analyze how the machine learning model learns over time during training and how our score metric matches the conformance metrics. Furthermore, we investigate the mistakes the method still makes by acquiring statistics on whether or not all information available is used or an abundance of information caused the mistake. All this is worked out in Chapter 10.

- Secondly, we evaluate the approach by feeding real-life data to the trained machine learning model. This is the actual task of process discovery, since we are generally not interested in the process models of synthetic processes. Furthermore, this shows whether the machine learning model generalizes beyond the block-structured process models with clean event logs. The real-life datasets consists of solely the event log which is not clean. Here, the resulting process models are compared to those produced by the state-of-the-art methods described in Section 3.3 in terms of conformance metrics like fitness, precision, generalization and simplicity. This evaluation is described in Chapter 11 and shows to what degree our approach can keep up with state-of-the-art algorithmic approaches in terms of quality performance, having fewer assumptions.
Chapter 6

Synthetic data generation

The importance of creating a synthetic dataset for process discovery is discussed in the previous section. How this should be done is part of the third research question of this thesis (RQ3). Here, we elaborate on our process of doing so. First, by creating the process models, discussed in Section 6.1, which are then simulated to generate the corresponding event logs as elaborated on in Section 6.2.

6.1 Process trees

As described in the background section, process trees are block-structured process models which means that the structures are not as intertwined as a more complex process model could be. However, it does give a nice starting point for a machine learning model to learn how the event log relates to the different substructures like parallelisms, choices and cycles. A plugin called PTAndLogGenerator, developed in [15], is included in the ProM framework [40] which can be used to generate such process trees. A large number of process trees is generated using this plugin with probabilities for whether a sequence, parallelism, choice, cycle or silent transition is chosen in each internal node of the tree. The specific probabilities used in our experiments is described in Chapter 9. OR gates are left out, since they are more difficult to transform to a Petri net. These generated process trees are then transformed to Petri nets and their silent transitions are reduced, preserving all behavior in the process model, to increase the simplicity of the process model.

6.2 Simulation

The result is a large dataset of block-structured Petri nets. However, it is still incomplete for the learning process since it only consists of the process models, without their respective event logs. So we essentially only have the labels of the data without the datapoints themselves. This is where simulation comes into play. Petri nets are process models ready for simulation to a certain extent. A sound Workflow net is designed in such a way that each process instance starts with a token at the initial marking and fire a certain set of transitions such that at the end of the process instance, a single token is left in the final marking. Using this fact, a
process model can be simulated by setting a single token in the initial marking and firing the enabled transitions making random choices when multiple transitions are enabled at the same time. To make the simulation more realistic, the completion of a transition takes a certain amount of time which again is a randomly generated value. This way, a simulated process execution instance is created of the process model. Repeating this process multiple times results in an event log ready for training. In order to be able to successfully discover a process model from an event log, this event log ought to be complete in the sense that every directly follows relations should be present in the log. To ensure this, a sufficient number of runs should be simulated where we maximize the chance by sampling from uniform distribution in choice constructs, where multiple transitions are enabled.

The execution times of transitions are a randomly generated number between 0 and 1 for each activity, while in real-life situations, different activities generally take different amounts of time. Furthermore, in choice constructs we sample from a uniform distribution, while in real-life situations, this choice is seldom uniform and could even be dependent on activities earlier in the process.

We obviously want as realistic data as possible to feed the machine learning model with. How realistic this synthetic data actually is, is analyzed in Section 9 by comparing it with real-life event logs that are also used for evaluation, looking at statistics about trace variant frequencies and deviations from main behavior.
Chapter 7

Data preparation

This chapter focuses on the research question RQ2.1, which addresses the challenge of producing graph-structured process models from sequence-structured event log. In short: the complete process discovery problem is modeled as a graph and it is turned into a node classification task by forming the traces of the event log into a graph and attaching a Petri net with candidate places which existence is to be determined by the machine learning model. Both the event log $L$ and its corresponding Petri net $M$ are encoded in a graph providing the data for training. The general idea of how the graph is constructed is explained in Section 7.1 after which we go into detail about the construction of the two halves, namely the trace graph in Section 7.2 and the Petri net graph in Section 7.3.

Our running example is $L_r = \{\langle a, b, c, d \rangle^{10}, \langle a, c, b, d \rangle^{12}, \langle a, b, c, e \rangle^{8}, \langle a, c, b, e \rangle^{10}\}$ which is the same as in Chapter 3.

7.1 Graph construction

![Graph construction diagram]

Figure 7.1: Simplified overview of the constructed graph for event log $\{\langle a, b \rangle^{1}, \langle a, b \rangle^{1}\}$.

The process discovery problem will be modeled as a graph turning it to a node
classification task. A simplified version of the graph that is constructed is depicted in Figure 7.1. The graph consists of two parts: the so-called trace graph and the so-called Petri net graph.

- The trace graph encodes the input event log by modeling the traces into a graph where each node corresponds to an event in the log. Artificial start and end events are added to connect the traces.
- The Petri net graph encodes the output process model by first modeling the distinct activities from the event log as transitions. Candidate places are added between the transition nodes, which are motivated by the event log’s Alpha relations, as described in Section 2.3.1.

The two graphs are connected via the events in the trace graph to the corresponding transitions in the Petri net graph. Each node holds an initial feature, which is illustrated in Figure 7.1 by the letters corresponding to the activity names in the nodes.

The task of process discovery can now be done by classifying the candidate places determining their existence. Then, after removing the trace graph and the candidate places that are not chosen, we end up with a Petri net forming the process model for the event log. The initial and final marking are denoted by the places that are attached to the artificial start and end transition respectively.

This design is motivated by the fact that this allows the structural sequential information from the traces to be propagated via the transitions in the Petri net to the possible places. Having this information, the model can learn which set of places should exist in which situation. During training, we know how the candidate places should be classified by looking at the event log’s corresponding Petri net.

The construction of the two graphs is elaborated on in detail in the following sections.

### 7.2 Constructing the trace graph

![Trace Graph for Event Log $L_r$](image)

Figure 7.2: Trace graph for event log $L_r$.

The trace graph encodes the information for the event log. The trace graph for our example event log $L_r$ is illustrated in Figure 7.2. Its construction is straightforward.
ward by processing the event log’s trace variants with their corresponding frequencies.

Algorithm 7.1 describes the general procedure to construct the trace graph. The nodes $V_{\text{event}}$ residing in the trace graph will be referred to as event nodes. It consists of first constructing two event nodes $v_{>}$ and $v_{|}$ acting as the artificial start and end event. Then, for each trace $T = \langle t_1, t_2, \cdots, t_{|T|} \rangle$ and its frequency $f$, the events are constructed as a chain graph which are connected to $v_{>}$ and $v_{|}$. This is clearly illustrated in Figure 7.2 and formally defined in lines 5 to 16 in Algorithm 7.1.

Algorithm 7.1: Algorithm for constructing event nodes $V_{\text{event}}$.

**input:** Event log $L$

**Result:** Event nodes $V_{\text{event}}$ for the trace graph

1. $V_{\text{event}} = \{ v_{>, v_{|}} \}$;
2. $\text{initializeFeature}(v_{>, >}, L, 1)$;
3. $\text{initializeFeature}(v_{|}, |, L, 1)$;
4. $F \leftarrow \sum_{T \in L} f$;
5. for trace $T = \langle t_1, t_2, \cdots, t_{|T|} \rangle \in L$ do
6.  for event $t_i \in T$ do
7.    Add event node $v_{t_i}$ to $V_{\text{event}}$;
8.    $\text{initializeFeature}(v_{t_i}, t_i, L, f / F)$;
9.    if $i = 1$ then
10.       $\text{addEdge}(v_{\text{start}}, v_{t_i}, \text{true})$;
11.    else
12.       $\text{addEdge}(v_{t_{i-1}}, v_{t_i}, \text{true})$;
13.    end
14.  end
15.  Add edge from $v_{t_{|T|}}$ to $v_{\text{end}}$;
16. end

7.2.1 Node features

Recall from Section 4.2.1 that the nodes have a feature vector initially. The event nodes essentially consist of two features: the activity name which is a categorical value and the normalized frequency which is a single number between 0 and 1.

A common feature initialization technique that is used when dealing with such categorical values is using one-hot encodings. Each distinct activity name will be transformed to a one-hot vector of length equal to the number of distinct activities plus two for the artificial start and end transition. A one-hot vector has all zeroes with a single 1 at a certain index indicating the category which in our case is the activity name. One-hot encodings are common when dealing with categorical values, since otherwise when using for example just a label, a neural network will interpret them as ordered values. So when events $a$, $b$ and $c$ map to 1, 2 and 3 respectively, it implies that the average of $a$ and $c$ is $b$, since $\frac{1+3}{2} = 2$. With one-hot encodings, this is not a problem.

The second feature, the normalized frequency, is a single number which is ap-
pended to the one-hot feature vector. A event node’s frequency is defined by the frequency $f$ of the trace variant it is constructed from. It is normalized by dividing it with the total number of traces $F$ found in the log. Line 4 in Algorithm 7.1 computes $F$.

Figure 7.2 shows the initial features for a selection event nodes. With seven distinct activities (>, a, b, c, d, e, |) and the frequency value, the size of the initial feature is eight. Algorithm 7.2 describes the feature initialization formally and is used in Algorithm 7.1. The arguments for this procedure are the node $v$ to add the feature to, the activity name $a$, event log $L$ and the frequency $f$. Lines 1 to 10 are used to get the distinct activity names $A$ in sorted order, using $L$. Line 11 creates a feature vector with $|A|$ zeroes appended with the frequency $f$. Line 13 then changes the zero to a one for the correct activity name.

**Algorithm 7.2: Initialize feature vector.**

<table>
<thead>
<tr>
<th>input: Node $v$, activity name $a$, event log $L$, frequency $f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $A \leftarrow \emptyset$;</td>
</tr>
<tr>
<td>2. for trace $T_i = (t_{i1}, t_{i2}, \ldots, t_{i</td>
</tr>
<tr>
<td>3. \hspace{1em} for event $t_{ij} \in T$ do</td>
</tr>
<tr>
<td>4. \hspace{2em} if $t_j \notin A$ then</td>
</tr>
<tr>
<td>5. \hspace{3em} Add $t_j$ to $A$; // Get distinct activities</td>
</tr>
<tr>
<td>6. \hspace{2em} end</td>
</tr>
<tr>
<td>7. end</td>
</tr>
<tr>
<td>8. end</td>
</tr>
<tr>
<td>9. $A.sort();$ // Sort alphabetically</td>
</tr>
<tr>
<td>10. $A \leftarrow [&gt;, a_1, a_2, \ldots, a_{</td>
</tr>
<tr>
<td>11. /* Initialize feature vector with $</td>
</tr>
<tr>
<td>12. $v.feature = [0, 0, \ldots, 0, f]$;</td>
</tr>
<tr>
<td>13. if $l \neq \text{null}$ then</td>
</tr>
<tr>
<td>14. \hspace{1em} $v.feature[A.index(a)] \leftarrow 1;$ // Set one zero to one</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

### 7.2.2 Edges

Note, in Figure 7.2, the different arrow types on either side of each edge. This denotes that they actually consist of two separate edges, going both directions. Algorithm 7.3 is used by Algorithm 7.1 and describes how these edges are created and how their directions are specified, where the boolean argument ‘reverse’ determines whether to add edges in both directions (lines 3 to 6). The direction is specified as a feature on the edge in lines 2 and 5 by denoting a forward edge by $[1,0]$ and a backward edge by $[0,1]$. Edges for both directions are added to allow the GCN to propagate information also of subsequent events through the graph.
Algorithm 7.3: Add edge.

\begin{algorithmic}
\STATE \textbf{input:} From node $v$, to node $u$, boolean $reverse$
\STATE 1 Add edge $e_{v,u}$ from $v$ to $u$;
\STATE 2 $e_{v,u}.direction = [1, 0]$;
\STATE 3 \textbf{if} $reverse$ \textbf{then}
\STATE 4 \hspace{1em} Add edge $e_{u,v}$ from $u$ to $v$;
\STATE 5 \hspace{1em} $e_{u,v}.direction = [0, 1]$;
\STATE 6 \textbf{end}
\end{algorithmic}

7.3 Constructing the Petri net graph

The Petri net graph, illustrated by the bottom part in Figure 7.1, encodes the process model consisting of transitions and candidate places. By selecting a subset of these candidate places, a Petri net can be produced. The construction of the Petri net graph consists of three phases of which the first two are done in preparation and the third happens during the sequential generation:

- phase 1: constructing the labeled transitions $V_{transition}$ corresponding to the distinct activities from the event log which are connected to the corresponding event nodes $V_{event}$ from the trace graph;
- phase 2: constructing the candidate places $V_{place}$ connecting the transition nodes $V_{transition}$;
- phase 3: constructing the candidate silent transitions $V_{silent}$, connecting the already chosen candidate places $V_{place}^{(chosen)}$.

Each phase is elaborated on in a separate section below.

7.3.1 Transitions

The transitions in a Petri net, as described in Section 2.2, denote the different activities that can be taken in a process. Their construction is straightforward by just creating a separate node for each distinct activity from the event log. For our running example, the transition nodes are added to trace graph as depicted in Figure 7.3.
Algorithm 7.4 describes the general case of constructing the transition nodes. First, in line 1, the two transition nodes, $v_{\text{transition}} >$ and $v_{\text{transition}} |$, for the artificial start and end activity are created. Their features are initialized similarly as those of event nodes. Notice that the frequency feature is 1 here. These nodes are attached to the event nodes in the trace graph by adding edges from $v_{\text{event}} >$ to $v_{\text{transition}} >$ and from $v_{\text{event}} |$ to $v_{\text{transition}} |$. Note that the reverse edges are not added this time, since we only make decisions on candidate places which reside in the Petri net graph, we only want information to flow from the trace graph to the Petri net graph and not vice versa.

Then, by loop over all events from all traces, in lines 5 to 14, the transitions for the distinct activities are added to $V_{\text{transitions}}$ and their features are initialized. During this looping, the corresponding edges are added as well, from the event nodes to the respective transition nodes, denoted by the colored edges in Figure 7.3. The edges are colored merely to distinguish them.

Algorithm 7.4: Algorithm for constructing transition nodes $V_{\text{transition}}$.

```
\textbf{input:} Event log $L$

\textbf{Result:} Transition nodes $V_{\text{transition}}$ for the Petri net graph

1. $V_{\text{transition}} = \{v_{\text{transition}} >, v_{\text{transition}} |\}$;
2. \textsc{initializeFeature}(\textit{v}, >, \textit{L}, 1);
3. \textsc{initializeFeature}(\textit{v}, |, \textit{L}, 1);
4. \textsc{addEdge}(\textit{u}, \textit{v}, false);
5. \textsc{addEdge}(\textit{u}, \textit{v}, false);
6. $A \leftarrow \emptyset$;
7. for trace $T_i = \langle t_{i1}, t_{i2}, \cdots, t_{i|T_i|} \rangle \in L$ do
   8. for event $t_{ij} \in T$ do
      9. if $t_j \notin A$ then
         10. Add $t_j$ to $A$; // Only add nodes for distinct activities
         11. Add $v_{t_j}^{\text{transition}}$ to $V_{\text{transition}}$;
         12. \textsc{initializeFeature}(\textit{v}, $t_j$, \textit{L}, 1);
      end
      14. \textsc{addEdge}($u_{t_{ij}}^{\text{event}}$, $v_{t_j}^{\text{transition}}$, false);
   end
end
```

7.3.2 Candidate places

Places, as described in Section 2.2 connect the transitions to visually expose the process’ logic. Many different places are possible in a Petri net denoting different constructs, shown in Figure 2.2. Which set of places suit best in a particular situation is to be determined by the machine learning model. Here, we construct every candidate place and add it to the Petri net graph. Rather than adding every place connecting every possible set of transitions, we look at the event log’s Alpha relations to only add plausible candidate places $V_{\text{place}}$. 45
Figure 7.4: Petri net graph with candidate places example.

Figure 7.4 shows a selection of candidate places that are constructed for our running example with event log $L_r$. The places can be categorized in three ways, determined by the number of incoming and outgoing transitions: one-to-one, one-to-many/many-to-one and many-to-many. The figure, together with the formal algorithms, are used to describe the process of constructing the candidate places in three phases based on these categories:

**Phase 1:** In the first phase, all one-to-one places $V_{\text{place}}^{(1-1)}$ are constructed, meaning that each place has exactly one incoming and one outgoing transition connected. Algorithm B.1 describes the process of constructing these candidate places. Again, a loop is done in line 2 over all traces in event log $L$ where the artificial start and end events are added to each trace. Then, there is a second argument to the procedure, namely the eventually follows parameter $F$. This parameter denotes which places are added. When $F = 1$, only the places from the directly follows relations are constructed. In the example Petri net from Figure 3.7, places $p_1$ and $p_2$ would not be added, since the directly follows relations $a \rightarrow d$ and $b \rightarrow e$ do not exist, with $F = 1$. With $F > 1$, on the other hand, these would be added. The second loop loops from 1 to $F$ with the look ahead parameter $l$. Then for each event $t_{ij}$ in trace $T_i'$ a place is added with incoming transition $v_{\text{transitions}}^{(transitions)}$ and outgoing transition $v_{\text{transitions}}^{(transitions)}$. Note that this is only done when this place does not exist yet and $j \leq |T_i'| - l$, since otherwise $t_{j+l}$ is undefined.

Mathematically, as Equation 7.1 states, for each one-to-one place $v_{\{a\},{\{b}\}}^{\text{place}}$ there is a value $f$ between 1 and $F$, where the eventually follows relations $a \rightarrow b$ holds. $v_{\text{transitions}}^{(place)}$ is the notation for a place with incoming transitions $T_{in}$ and outgoing transitions $T_{out}$. So with $T_{in} = \{a\}$ and $T_{out} = \{b\}$, $v_{\{a\},{\{b\}}^{\text{place}}$ has an edge from $v_{\text{transitions}}^{(transitions)}$ and an edge to $v_{\text{transitions}}^{(transitions)}$.

\[
\forall v_{\{a\},{\{b\}}^{\text{place}} \in V_{\text{place}}^{(1-1)} : \exists f [1 \leq f \leq F : a \rightarrow b] \]  \hspace{1cm} (7.1)

All candidate places, i.e. one-to-one, one-to-many/many-to-one and many-to-many places, are initialized equally with a null category and 0 frequency, meaning that the one-hot vector contains only zeroes. The edges, together with the reverse edges, are added as described from the incoming transition nodes to the new place and from the new place to the outgoing transition nodes. The one-to-one places in our example in Figure 7.4 are $V_{\text{place},1-1} = \{p_1, p_2, \ldots, p_7\}$.
Phase 2: In the second phase, all one-to-many and many-to-one places \( V_{\text{place}}^{(1-n)} \) and \( V_{\text{place},n-1} \) are constructed, respectively the places with one incoming transition and multiple outgoing transitions and the places with multiple incoming transitions and one outgoing transition. In Figure 7.4, we have \( V_{\text{place}}^{(1-n)} = \{p_{11}, p_{12}\} \) and \( V_{\text{place}}^{(n-1)} = \{p_8, p_9, p_{10}\} \).

Algorithm B.2 describes the construction of these places. First the sets of places with shared incoming transition are taken to get combinations of their outgoing places with multiple incoming and multiple outgoing transitions. These are similarly paralleled to any two transitions in \( T_{\text{out}} \) from \( P \) and if and only if there is no parallel relation in any two transitions from \( T_{\text{out}} \).

Mathematically, as Equation 7.2 states, for each one-to-many place \( v_{(a),T_{\text{out}}} \), \( T_{\text{out}} \) exists in the set of combinations from the outgoing transitions from places with shared incoming transition, i.e. \( v_{(a),T_{\text{out}}} = \left( T_{\text{out}} \right) \) denotes the set of all \( k \)-element subsets of a set \( T_{\text{out}} \), so \( \bigcup_{k=2}^{\left| T_{\text{out}} \right|} \left( T_{\text{out}} \right) \) denotes the set of \( k \)-element subsets with \( 2 \leq k \leq \left| T_{\text{out}} \right| \). For example, with \( T_{\text{out}} = \{a, b, c\} \), this set equals \( \{(a, b), \{a, c\}, \{b, c\}, \{a, b, c\}\} \). \( \exists t_1, t_2 \in T_{\text{out}} : t_1 \parallel t_2 \) denotes the absence of a parallel relation between any two transitions in \( T_{\text{out}} \).

\[
\forall v_{(a),T_{\text{out}}} \left[ v \in V_{\text{place}}^{(1-n)} : T_{\text{out}} = \bigcup_{k=2}^{\left| T_{\text{out}} \right|} \left( T_{\text{out}} \right) \right] \land \neg \exists t_1, t_2 \left[ t_1, t_2 \in T_{\text{out}} : t_1 \parallel t_2 \right] \quad (7.2)
\]

with \( T_{\text{out}} = \{b : v_{(a),\{b\}} \land v \in V_{\text{place}}^{(1-1)}\} \).

The many-to-one places \( v_{T_{\text{in}},\{b\}} \) are constructed in lines 20 to 28. As can be seen from Equation 7.3, these are similar to the one-to-many places. An example of such place can be found in our example in Figure 7.4. \( p_{10} \) is the combination of \( p_6 \) and \( p_7 \).

\[
\forall v_{T_{\text{in}},\{b\}} \left[ v \in V_{\text{place}}^{(n-1)} : T_{\text{in}} = \bigcup_{k=2}^{\left| T_{\text{in}} \right|} \left( T_{\text{in}} \right) \right] \land \neg \exists t_1, t_2 \left[ t_1, t_2 \in T_{\text{in}} : t_1 \parallel t_2 \right] \quad (7.3)
\]

with \( T_{\text{in}} = \{a : v_{(a),\{b\}} \land v \in V_{\text{place}}^{(1-1)}\} \).

Phase 3: In the third phase, all many-to-many places \( V_{\text{place}}^{m-n} \) are constructed, the places with multiple incoming and multiple outgoing transitions. These are similarly constructed as the one-to-many and many-to-one places in the sense that places with shared incoming transitions are taken to get combinations of their outgoing transitions. The difference here is that the places that are combined are one-to-many or many-to-one places, rather than only one-to-one places. Note that no such places are shown in Figure 7.4, \( V_{\text{place}}^{m-n} = \emptyset \).

An example of how a many-to-many place is constructed from two many-to-one places is shown in Figure 7.5, where \( p_3 \) is constructed from \( p_1 \) and \( p_2 \) which share its incoming transitions.
Equation 7.4 together with Equations 7.5 and 7.6 also show that the characteristics of the one-to-many/many-to-one places are similar to those of the many-to-many places.

\[ \forall v^{(\text{place})} \left[ \left| T_{\text{in}} \right| > 1 \land \left| T_{\text{out}} \right| > 1 : P_{\text{in}} \lor P_{\text{out}} \right] \]  (7.4)

with predicates \( P_{\text{in}} \) and \( P_{\text{out}} \) defined in Equations 7.6 and 7.5.

\[ P_{\text{in}} = T_{\text{in}} \in \bigcup_{k=2}^{\left| T_{\text{in}} \right|} \binom{T_{\text{in}}}{k} \land \neg \exists_{t_1, t_2} [t_1, t_2 \in T_{\text{in}} : t_1 \parallel L t_2] \]  (7.5)

with \( T_{\text{in}} = \{ a : v^{(\text{place})}_{T_{\text{in}}, T_{\text{out}}} \land v \in V_{\text{place}}^{(1-n)} \} \).

\[ P_{\text{out}} = T_{\text{out}} \in \bigcup_{k=2}^{\left| T_{\text{out}} \right|} \binom{T_{\text{out}}}{k} \land \neg \exists_{t_1, t_2} [t_1, t_2 \in T_{\text{out}} : t_1 \parallel L t_2] \]  (7.6)

with \( T_{\text{out}} = \{ b : v^{(\text{place})}_{T_{\text{in}}, T_{\text{out}}} \land v \in V_{\text{place}}^{(n-1)} \} \).

Algorithm B.3 shows how these many-to-many places are constructed, which is very similar to Algorithm B.2 so we will not go into detail here.

7.3.3 Candidate silent transitions

As mentioned in Section 2.2, the expressive power of Petri nets is not complete without silent transitions. Silent transitions can be used to skip over subprocesses without breaking the structural soundness. However, in our approach of providing candidates for the machine learning model to decide on, silent transitions so far are not incorporated. Adding candidate silent transitions is also more difficult to do since they can not be found in the log and therefore deciding on what candidate silent transitions should be constructed is a difficult task. A trivial solution is chosen with the assumption that adding the possibility of having silent transitions only between two places, so one incoming place and one outgoing place. This already increases the expressive power of the possible Petri nets already by a lot. This choice is made because adding all possible silent transitions gives too many possibilities for the system to work efficiently. Adding only these simple silent transitions is done by connecting each pair of places twice, one for the forward silent transition and one for the backward one. Note that this increases the number of candidates already quadratically, since for \( p \) candidate places, \( 2p(p-1) \) silent transitions are added.
This still gives too much overhead relative to how many candidate places are constructed, so we added a rule that a silent transition may only be chosen when its connected places have already been chosen. This makes sense, since the silent transitions must always connect to chosen places. Besides, it gives only the small limitation in the generative process regarding the order of choices. Having this rule enables us to limit the number of candidate silent transitions by a significant amount, since only the candidates exist between already chosen places and the number of chosen places is significantly less than the total number of candidate places. This ensures that for every silent transition node $v_{in, out}^{(silent)}$ with incoming and outgoing places $P_{in}$ and $P_{out}$, the following holds: $|P_{in}| = |P_{out}| = 1$ and for each place node $w_{in, out}^{(place)} \in P_{in} \cup P_{out}$ we have $\exists w \in T_{in} \ [w \in V_{transition}]$ and $\exists w \in T_{out} \ [w \in V_{transition}]$. This is illustrated in Figure 7.6, where $P_{in} = \{p_1\}$ and $P_{out} = \{p_2\}$. Note that both $p_1$ and $p_2$ both have labeled incoming and outgoing transitions, even without the silent transition.

The construction of the candidate silent transitions $V_{silent}$ is described by Algorithm 7.5. Note that this is done during the process of choosing candidates, rather than beforehand. After each time a new candidate place is chosen, this procedure is executed. It adds candidate silent transitions between the last chosen place $v_{in, out}^{(place)}$ (line 4) and each place $u$ of the previously chosen places $V_{chosen}^{(place)}$ if a certain condition holds. For a silent transition from $u$ to $v$, there should be a directly follows relation between any of the incoming transitions of $u$ and the outgoing transitions of $v$. For the reverse silent transition, from $v$ to $u$, the opposite should be true. Lines 5 to 11 denote the construction of the silent transitions from $u$ to $v$ with the condition in line 6 and from $v$ to $u$ with the condition in line 12. The initial features of a silent transition node is the average of the connected places’ features, i.e. $w.feature = (u.feature + v.feature)/2$ for silent transition node $w_{u,v}^{(silent)}$. This way, the GCN can encode information about silent transitions already before they are constructed in chosen places.

After each choice, a maximum of $\sum_{i=1}^{\lfloor V_{chosen}^{(place)} \rfloor - 1} 2i$ new candidate silent transitions can be constructed.
Algorithm 7.5: Algorithm for constructing candidate silent transitions $V_{\text{silent}}$.

**input:** Chosen place nodes sequence $V^{(\text{chosen})}_{\text{places}}$, the event log’s Alpha relations $L_{\alpha}$

**Result:** Silent transition nodes $V_{\text{silent}}$ for the Petri net graph

1. if $\not\exists V_{\text{silent}}$
2.  \[ V_{\text{silent}} \leftarrow \emptyset; \quad \text{// Initialize } V_{\text{silent} \text{when it does not yet exist}} \]
3. end

4. \[ v^{(\text{place})}_{\text{in}, \text{out}} \leftarrow V^{(\text{chosen})}_{\text{places}}[|V^{(\text{chosen})}_{\text{places}}|]; \quad \text{// Last chosen place} \]

5. for \[ u^{(\text{place})}_{\text{in}, \text{out}}, v^{(\text{place})}_{\text{in}, \text{out}} \in V^{(\text{chosen})}_{\text{place}}, u \neq v \] do
6.  if $\exists t_i, t_j : t_i \in T_{\text{in}} \land t_j \in T_{\text{out}} \land t_i, t_j \in L_{\alpha}^{(\text{directly follows})}$ then
7.    \[ w, \text{feature} \leftarrow \frac{u, \text{feature} + v, \text{feature}}{2}; \]
8.    ADDEDGE($u, w, \text{true}$);
9.    ADDEDGE($w, v, \text{true}$);
10. end
11. if $\exists t_i, t_j : t_i \in T_{\text{in}}^{(\text{silent})} \land t_j \in T_{\text{out}}^{(\text{silent})} \land t_i, t_j \in L_{\alpha}^{(\text{directly follows})}$ then
12.    \[ w, \text{feature} \leftarrow \frac{u, \text{feature} + v, \text{feature}}{2}; \]
13.    ADDEDGE($u, w, \text{true}$);
14.    ADDEDGE($w, v, \text{true}$);
15. end
18. end

In the end we have four sets of nodes forming the complete graph: $V_{\text{event}}$, $V_{\text{transition}}$, $V_{\text{place}}$, $V_{\text{silent}}$ of which the latter is created during the generation process. $V_{\text{event}}$ encode the input event log. The other nodes encode the Petri net from which the process model can be chosen, consisting of transition $V_{\text{transition}}$ and a selected subset of $V_{\text{place}}$ and $V_{\text{silent}}$.

Note that with this chosen graph construction three assumptions are made on the data:

- each activity is modeled by a single transition, so no activity splitting possible;
- the event log is complete enough in the sense that every eventually follows relation (up to depth $F$) is present in the event log, since otherwise, not all candidate places will be constructed for the final Petri net;
- all silent transition have only one incoming and one outgoing place, where in addition, these places already have at least one incoming and one outgoing labeled transition.
Chapter 8

Process model generation

Using the prepared data as described in Chapter 7, process models of the form of Petri nets are produced by classifying candidate places and silent transitions. This chapter focuses on this process of producing Petri nets from event logs. This aims at both research questions RQ2.1 and RQ2.2, which state the problem of producing the process models from event log data as well as how a machine learning model can learn to do so. The overall method is described in Section 8.1 after which we dive into the different components of the method in Section 8.2. The event log-process model pairs (L, M) from our synthetic dataset, as described in Chapter 6, are used to train the neural networks in our machine learning model. Note that the machine learning model refers to the method of our approach. The training process is described in Section 8.3. As will become clear throughout this chapter, processing an event log L during inference differs slightly from processing a (L, M) pair during training. Inference for our proposed method is elaborated on in Section 8.4.

8.1 Sequential model overview

We have designed two variants of our method to produce process models from event logs: the discriminative model and the sequential model. Both variants incorporate graph convolutional neural networks operating on the graph G\textsubscript{in} encoding the event log and candidate places by classifying these candidates. The discriminative variant does so by classifying them in one go, i.e. after a certain number of propagation rounds in the GCN, the node embeddings of the candidate places contain the probability of their class.

On a sidenote: a node embedding is a mapping of a a node’s initial feature to a vector of continuous numbers, determined by the inner parameters W in the GCN’s layers, as explained in Section 4.2.1. Note that we now have three names for the nodes’ vectors: initial feature, hidden state and embedding, which correspond to the nodes’ information before, during and after being processed by a GCN.

Deciding on each place existence at once gives rise to a couple of problems. It is desirable that choices for places depend on the choices of other places and. In order to make these choices dependent on other places, they have to be chosen in a sequential manner, which is how the sequential variant operates. This variant is the
Figure 8.1: Overview of the sequential model after two rounds.

one in our proposed method and is what the rest of this chapter focuses on. More details on the workings of the discriminative model can be found in Appendix C.

The workings of this sequential model are illustrated in Figure 8.1. At the start is the input graph $G_{in}$, with the trace and Petri net graph. The nodes’ colors depict the feature vectors. Note $G_{in}$ shows a simplified version without all possible candidate places. The squares in the figure denote the different neural networks used in the model. The sequential generation process operates as follows:

- The first step is to update the node embeddings by processing the graph through a propagation network which is a four layer GCN with attention mechanism as explained in Sections 4.2.1 and 4.2.2. Figure 8.1 illustrates this by changing the nodes’ colors. Furthermore, the choice of each candidate places is initialized to false, meaning that they are removed at the end. This is denoted by the gray outline of the places in graph $G'$. Candidates with a green outline depict the chosen candidates that reside in the final Petri net.
- Then the iterative process starts, where each loop consists of the following actions:
  - The first step is to decide whether we should continue the loop of adding new candidates to the Petri net. The ‘Add candidate’ network takes the embeddings of all nodes from the Petri net graph and processes them to compute a graph embedding, i.e. a single vector representing this full graph. From this graph embedding, a single value is computed denoting the probability that the loop should be continued. The choice of continuing or stopping is denoted by the diamond in Figure 8.1.
  - When the choice was to continue the generation process, the next step
is to update the node embeddings again using a propagation network again, this time with a two layer GCN with attention. At least a two layer propagation network is necessary to propagate information about the chosen candidates to the not yet chosen ones. The updated node embedding are visualized again by changing the nodes’ colors in the bottom right graph $G''$. Note that in Figure 8.1 illustrates the third round in this loop. Two rounds have already been done, which can be seen by the two green places in $G''$.

- With the updated node embeddings, the next candidate can be chosen. The ‘Choose candidate’ network takes all node embeddings and computes a probability denoting the classification for each of them. With these probabilities, a candidate can be chosen and labeled as such, e.g. by taking the one with highest probability. The available candidates are $V_{\text{place}} \setminus V_{\text{place}}^{(\text{chosen})}$ and $V_{\text{silent}} \setminus V_{\text{silent}}^{(\text{chosen})}$. The result is a new candidate with a green outline, which is the place from transition $>$ to $a$ in our $G''$ in Figure 8.1.

- An intermediate step, not using neural networks, is to add the candidate silent transitions. As mentioned in Section 7.5, where this process is described, the candidate silent transitions are only added between already chosen places. So after a candidate place has been chosen, silent transitions are added to the graph between the chosen places acting as new candidates. $G'''$ illustrates the new graph in Figure 8.1.

- At the end of the iterative loop, when the ‘Add candidate’ network decides to stop the generation process, the graph contains labeled candidates, depicted by their outline colors. $G_{\text{out}}$ in Figure 8.1 depicts the final output graph. The resulting Petri net can easily be extracted from $G_{\text{out}}$ by first removing the nodes from the trace graph $V_{\text{event}}$ and the unchosen candidates $V_{\text{place}} \setminus V_{\text{place}}^{(\text{chosen})}$ and $V_{\text{silent}} \setminus V_{\text{silent}}^{(\text{chosen})}$. Transitions $>$ and $|$ are used to determine the initial and final marking respectively. Each place with incoming transition $>$ is combined and serves as the initial marking. Similarly, the places with outgoing transition $|$ are combined serving as the final marking.

Each place and silent transition in the produced Petri net now has a probability for its existence. Moreover, the probabilities are joint probabilities, meaning that the choice of the existence of a candidate is based on the choice of previously chosen candidates. A notion of certainty about the produced Petri net can be expressed by this joint probability between all candidates, which by the chain rule equals the product of conditional probabilities:

$$p(v_1, \ldots, v_n) = p(v_1)p(v_2|v_1) \cdots p(v_n|v_1, \ldots, v_{n-1})$$

(8.1)

where $p(v_i|v_1, \ldots, v_{i-1})$ denotes the probability of candidate node $v_i$ having candidate nodes $v_1$ to $v_{i-1}$ already chosen.

### 8.2 Neural networks

Details on the components using neural networks in our machine learning model are elaborated on in this section, starting with the propagation network followed
by the ‘Add candidate’ network and the ‘Choose candidate’ network.

8.2.1 Propagation network

Two propagation networks are present in our model: a four- and a two-layer graph convolutional neural network with multi-head attention mechanism as explained in Sections 4.2.1 and 4.2.2. These networks are responsible for propagating information through the graph and manipulating the nodes’ embeddings such that they contain information to be used by the other networks.

A GCN with $n$ layers denotes that the nodes’ features of the $n$-neighborhood for each node are aggregated. Figure 8.2 illustrates part of a 3-neighborhood of an example place node. This place node $p_2$ is equal to the place node $p_2$ from Figure 7.4. Note that the same event log $L_r = \{\langle a, b, c, d \rangle^{10}, \langle a, c, b, d \rangle^{12}, \langle a, b, c, e \rangle^{8}, \langle a, c, b, e \rangle^{10}\}$ is applied here.

The colored rectangles denote the different layers in the GCN. Recall that reverse edges are added in both the trace and the Petri net graph. In order to distinguish the two, information over these edges is processed by different weight matrices, i.e. a single layer $l$ in our GCN contains two weight matrices $W^{(l)}$ and $W_r^{(l)}$ for the edges with direction [1, 0] and [0, 1] respectively. The first three layers have attention mechanisms with four heads, where the last layer only has one head. The number of neurons in each layer is 32, 64, 32 and 16.

For the first three layers of the network the nodes’ hidden states are updated by:

$$h_i^{(l+1)} = \| \left( \sum_{j \in \mathcal{N}(i)} d_{ij} \begin{bmatrix} \alpha_{ij}^{(l)} h_j^{(l)} W^{(l)} \alpha_{ij}^{(l)} h_j^{(l)} W_r^{(l)} \end{bmatrix} \right) \|$$

with $h_i^{(l)}$ and $h_i^{(l+1)}$ the hidden states at after layer $l$ and $l + 1$ of node $i$. $\|$ denotes concatenation again and $d_{ij}$ denotes the direction of edge from node $i$ to $j$, i.e. [1, 0] or [0, 1]. Note that the hidden states are updated using both weight matrices $W$ and $W_r$. Both updated values are put in a matrix and $d_{ij}$ essentially select the correct
one according to the directions. To illustrate: \([1, 0] \begin{bmatrix} x \\ x_r \end{bmatrix} = x\) and \([0, 1] \begin{bmatrix} x \\ x_r \end{bmatrix} = x_r\).

The update function of the last layer, having a single attention head, is:

\[
h^{(l+1)}_i = \text{ReLU} \left( \sum_{j \in N(i)} d_{ij} \begin{bmatrix} \alpha^{(l)}_{ij} h^{(l)}_j W^{(l)}_j \\ \alpha^{(l)}_{ijr} h^{(l)}_j W^{(l)}_j \end{bmatrix} \right)
\]  

(8.3)

with the \(\text{ReLU}\) activation function.

With \(C\) being the size of the initial features, the shapes of the weight matrices are \(W^{(1)} : C \times 32, W^{(2)} : 32 \times 64, W^{(3)} : 64 \times 32, W^{(4)} : 32 \times 16\). The \(W_r\) weight matrices have respectively the same shapes.

To show what information from the event log is aggregated in a candidate place node \(v^{(\text{place})}_{\text{Fin}, \text{Start}}\), we state which nodes from \(V_{\text{event}}\) are aggregated in \(v\). To do so, let us first define, from graph theory, the notion of a (directed) ball and a (directed) sphere in the graph. A ball \(B_v, r\) with center node \(v\) and radius \(r\) is the set of nodes at most \(r\) hops away from \(v\) only using forward edges:

\[
B_v, r = \{ u \in V : \delta(v, u) \leq r \lor \delta(u, v) \leq r \}
\]  

(8.4)

and directed balls \(B^1_{v,r}\) and \(B^{-1}_{v,r}\):

\[
B^1_{v,r} = \{ u \in V : \delta(u, v) \leq r \}, \quad B^{-1}_{v,r} = \{ u \in V : \delta(v, u) \leq r \}
\]  

(8.5)

where \(\delta(v, u)\) denotes the distance of the shortest path from \(u\) to \(v\), only using forward edges, i.e. where \(e.\text{direction} = [1, 0]\).

\[
\partial B_v, r = \{ u \in V : \delta(u, v) = r \lor \delta(v, u) = r \}
\]  

(8.6)

Directed spheres \(\partial B^1_{v,r}\) and \(\partial B^{-1}_{v,r}\) are equally defined as the directed balls.

For each node, information from the nodes in the two directed balls with radius 4, \(B^1_{v,4}\) and \(B^{-1}_{v,4}\), are aggregated. We dissect these two balls and state which nodes from \(V_{\text{event}}\) reside in the spheres \(\partial B^1_{v,k}\) and \(\partial B^{-1}_{v,k}\) for \(1 \leq k \leq 4\). Figure 8.2 offers clarity in the process:

- \(\partial B^1_{v,\text{place}}, 1\), being one hop away from \(v\), contains the incoming transitions of \(v\) processed by the purple rectangle;
- \(\partial B^{-1}_{v,\text{place}}, 1\) contains the outgoing transitions of \(v\) processed by the green rectangle;
- \(\partial B^1_{v,\text{place}}, 2\), being two hops away from \(v\), contains all event occurrences in the event log corresponding to the activities of the incoming and outgoing transitions of \(v\). These are processed by the orange rectangles;
- \(\partial B^{-1}_{v,\text{place}}, 2\) contains only other candidate places, since there are no edges from \(V_{\text{transitions}}\) to \(V_{\text{event}}\), so it contains no information from the event log and is therefore not visualized in Figure 8.2;
- \(\partial B^1_{v,\text{place}}, 3\) contains the preceding event nodes of the event nodes from \(\partial B^1_{v,\text{place}}, 2\), providing information about one step in the past. These are processed by the yellow rectangles;
- \(\partial B^{-1}_{v,\text{place}}, 3\) contains similar nodes as \(\partial B^1_{v,\text{place}}, 3\), but providing information about one step in the future and are processed by the blue rectangles;
• $\partial B^{1}_{v,4}$ contains again similar nodes as $\partial B^{1}_{v,3}$, but with two steps in the past. The are not visualized in Figure 8.2 because it would be too cluttered;

• $\partial B^{-1}_{v,4}$ contains similar nodes as $\partial B^{1}_{v,4}$, but with two steps in the future, again not visualized in Figure 8.2.

More formal definitions of $\partial B^{1}_{v,k}$ and $\partial B^{-1}_{v,k}$ for $1 \leq k \leq 4$ can be found in Appendix D.

Altogether, for $B^{1}_{v,4}$ and $B^{-1}_{v,4}$, we have that from the activities of all incoming and outgoing transitions of $v$, every occurrence, in the form of an event node, with its two preceding and two subsequent events are aggregated.

The other, two-layer, GCN operates similarly. The first layer, consisting of 32 neurons, again has four attention heads and the last layer, consisting of 32 neurons, only has one attention head. The weight matrices in this network therefore have the shapes: $W^{(1)}: 17 \times 32$, $W^{(2)}: 32 \times 16$. The input dimension is 17 because each nodes’ classifications are added as a feature to the nodes’ embeddings after each choice. $B^{1}_{v,2}$ and $B^{-1}_{v,2}$ here consist solely of the connected transition nodes and their connected nodes, being either an event node or a candidate place node. A network of at least two layers is necessary to propagate the last made decision to the other candidates.

### 8.2.2 ‘Add candidate’ network

The ‘Add candidate’ network is responsible for deciding when to stop adding candidate places or silent transitions. Given the complete Petri net graph, a single boolean value should be outputted. To do so, an embedding of this full graph is created, which aggregates all the nodes hidden states processing it through a fully connected neural network with a single output value. Similar to the network used in [22], a gating function is used denoting how much the overall graph embedding attends on each node. The graph embedding is computed by the following equation:

$$h_G = \sum_{v \in V \setminus V_{\text{event}}} \text{SIGMOID}(h_v W_a)(h_v W_g)$$

with shapes $W_a : 17 \times 1$, $W_g : 17 \times 32$, so the graph embedding $h_G$ has size 32. SIGMOID($h_v W_a$) serves as a gating function to compute the nodes’ weights. The logistic sigmoid function $\frac{1}{1+e^{-x}}$ is used to map a value from $\mathbb{R}$ to a value between 0 and 1.

$$p_{\text{add}} = \text{SIGMOID}(h_G W_d)$$

with shapes $W_d : 32 \times 1$, processing the graph embedding $h_G$ to compute the probability $p_{\text{add}}$ whether to add new candidates. ($h_G W_d$) is a score which is converted to a probability using the logistic sigmoid function. The boolean decision can then be determined by either using a threshold or by sampling from a Bernoulli distribution.
8.2.3 ‘Choose candidate’ network

The ‘Choose candidate’ network is a straightforward network consisting of only a single fully connected layer with input size equal to the nodes hidden state after the rounds of propagation and again a single output value. It is used to decide on the existence of candidate places and silent transitions. This network has one learnable weight matrix $W$ to compute the score $s_v$ for candidate node $v$ denoting how likely it is to be chosen next:

$$s_v = h_v W$$  \hspace{1cm} (8.9)

with learning weight matrix $W$ having shape $16 \times 1$. By normalizing all the candidates’ scores using the Softmax function we get a probability $p_v$ for each node:

$$p_v = \text{SOFTMAX}(s_v) = \frac{\exp(s_v)}{\sum_{u \in V_{\text{candidate}}} \exp(s_u)}$$  \hspace{1cm} (8.10)

with $V_{\text{candidate}} = \left(V_{\text{place}} \setminus V_{\text{place}}^{(\text{chosen})}\right) \cup \left(V_{\text{silent}} \setminus V_{\text{silent}}^{(\text{chosen})}\right)$ and $\exp(x) = e^x$. The choice of the next candidate can then be determined by either taking the node with highest probability or sampling from a multinomial distribution.

8.3 Training

The training data, consisting of the prepared event log - process model pairs encoded in graphs are individually fed to the network which in turn will process it by making a prediction. Since the desired places are known, a loss can be computed which is then used for backpropagation. Backpropagation adjusts the inner parameters in the weight matrices of the neural networks such that next time, the model sees the datapoint, the result will be closer to the desired result. One iteration of processing all datapoints is called an epoch in machine learning. This process repeats until a specified number of epochs is reached. In between each epoch, the dataset is shuffled to prevent the network from using information about the order. The loss is computed by penalizing the probabilities of the choices the network makes. In the discriminative model this is binary cross entropy and for the sequential model it is the product of probabilities of the places that should exist. To ensure stable training, this latter is actually the sum of the negative log probabilities, such that the result will not round to 0. Taking the sum of log probabilities is valid since for any two sets of probabilities $P_1$ and $P_2$, when $\prod_{p_1 \in P_1} p_1 \geq \prod_{p_2 \in P_2} p_2$ holds, also $\sum_{p_1 \in P_1} \log p_1 \geq \sum_{p_2 \in P_2} \log p_2$ holds. The loss function for training is then the negative of this sum which is to be minimized:

$$l = - \sum_{i=1}^{V_{\text{candidate}}^{(\text{chosen})}} \log p(v_i|v_1, \ldots, v_{i-1})$$  \hspace{1cm} (8.11)

with $V_{\text{candidate}}^{(\text{chosen})}$ the set of candidate places and silent transitions that should be chosen and $p(v_i|v_1, \ldots, v_{i-1})$ the probability of choosing candidate node $v_i$ after having chosen candidate nodes $v_1$ to $v_{i-1}$.
Let us define $V^{\text{chosen}}(c)$ (in short) and $L$ to be the supersets of the process model - event log pairs in the training dataset. $V^{(c)}$ consists of the target process models defined by the candidate places and silent transitions that have to be chosen. The model then defines a joint distribution $p(V^{(c)}, \pi|L)$ over chosen candidates $V^{(c)}$, ordering $\pi$ and input event logs data $L$. The ordering $\pi$ is added because of the iterative generation process our model adheres to. When generating samples, both the chosen set of candidates and an ordering are generated by the model. For both training and evaluation, we are interested in the marginal likelihood of the chosen set of candidates and an ordering are generated by the model. For both training and evaluation, we are interested in the marginal likelihood $p(V^{(c)}, \pi|L)$, since it does not matter which ordering is chosen. This marginal likelihood is intractable to compute with moderately large Petri nets, since it involves all permutations of $V^{(c)}$.

There are two ways of evaluating the marginal likelihood without going over all permutations, namely by sampling or approximation. A Monte-Carlo estimate based on sampling gives:

$$p(V^{(c)}|L) = \sum_{\pi} p(V^{(c)}, \pi|L) = \mathbb{E}_{\pi(V^{(c)}, \pi|L)} \left[ \frac{p(V^{(c)}, \pi|L)}{q(\pi|V^{(c)}, L)} \right]$$

(8.12)

with $q(\pi|V^{(c)}, L)$ a proposal distribution over permutations. So when sampling permutations of orderings from $q(\pi|V^{(c)}, L)$ and averaging $p(V^{(c)}, \pi|L)/q(\pi|V^{(c)}, L)$, an estimate of the marginal likelihood can be made. The variance is then minimized when $p(V^{(c)}, \pi|L) = q(\pi|V^{(c)}, L)$.

With a canonical ordering $\pi_c$, we can train and evaluate the model by taking $q(\pi|V^{(c)}, L)$ to put all the probability on this $\pi_c$. However, this choice of $q$ only gives a lower bound on the true marginal likelihood as it does not have full support over the set of all permutations.

With the loss function from Equation 8.11, we are now learning the joint distribution $p(V^{(c)}, \pi|L)$ of the data by minimizing the expected joint log-likelihood:

$$\mathbb{E}_{p_{\text{data}}(V^{(c)}, \pi|L)} [-\log p(V^{(c)}, \pi|L)] = \mathbb{E}_{p_{\text{data}}(V^{(c)}|L)} \mathbb{E}_{p_{\text{data}}(\pi|V^{(c)}, L)} [-\log p(V^{(c)}, \pi|L)]$$

(8.13)

With dataset $L$, we can get samples from $p_{\text{data}}(V^{(c)}|L)$ and we can choose $p_{\text{data}}(\pi|V^{(c)}, L)$ during training. By setting $p_{\text{data}}(\pi|V^{(c)}, L) = q(\pi|V^{(c)}, L)$, we match the training process with the evaluation process. The posterior of the model distribution $p(\pi|V^{(c)}, L)$ approaches the proposal distribution $q(\pi|V^{(c)}, L)$ during training, therefore improving the quality of our estimate of the marginal probability.

For the choice of $q(\pi|V^{(c)}, L)$, an ordering $\pi_c$ is decided on which is used in the training process. Making the model learn a specific ordering is done by teacher forcing.

### 8.3.1 Teacher forcing

Teacher forcing is a common technique in generative machine learning models that outputs a sequence of actions. Instead of computing the loss only at the end, the loss is computed at every step.
Another approach could be to just let the network make a prediction on all the places in a sequence compute a loss on how many it has correct. However, a pitfall in this approach is that when the first predicted place was incorrect, the complete sequence is incorrect. So instead, at every choice the network has probabilities for each place to be the next prediction. The probability of the to be selected place is remembered. This place is then also labeled as selected, rather than letting the network do this itself. This process is called teacher forcing, since a ‘teacher’ keeps track of the candidates the network chooses and corrects this at each step.

As mentioned in the previous section, the ordering of the sequence of places that the model outputs is important during training for it to learn the marginal probability. So besides the graph incorporating the event log and Petri net, also the ordering $\pi_c$ of the correct places has to be specified. Since the flow through a Petri net is always from the initial marking to the final marking, it makes sense to specify the order of the place sequence as such. Options are a depth first search, topological sort or breadth first search. The latter is chosen for our network, which means that the Petri net will be built up from initial marking to final marking by extending each path one step at a time. This ordering $\pi_c$ is canonical in the sense that an algorithm exists that defines the ordering, so $q(\pi|V_{c}^{(e)}, \mathcal{L})$ is chosen such that it puts all probability on $\pi_c$ which is approached by $p(\pi|V_{c}^{(e)}, \mathcal{L})$, made possible by $p_{data}(\pi|V_{c}^{(e)}, \mathcal{L})$ from the data using teacher forcing. Note that a breadth first search does not result in a completely canonical ordering since at each depth in the search the branches are selected in a random order, however, it is as canonical as possible.

A disadvantage of using teacher forcing in the training phase is that the network does not learn from its own predictions which is a problem during inference since the ground truth is not available. This discrepancy between training and inference is called exposure bias and is elaborated on a bit more in the next section.

### 8.4 Inference

Inference is where the machine learning model is given new unseen data and has to find out for itself how to deal with it. Often this is similar if not the same as the training but the final output is not compared to the desired output. However, in our case, since we apply teacher forcing in the training phase, inference is a bit different, though it is still similar. During inference, the machine learning model should select the places and silent transitions on its own and decide when to stop the generation process. Teacher forcing is used in the training phase to adjust the scores after every choice such that with an early wrong choice the later choices are still useful for training. During inference, however, teacher forcing can not be applied since the correct choices are unknown. So after making a choice, this choice is taken for the rest of the generation process. After each choice, the ‘Add candidate’ network has a probability whether to stop generating new places, illustrated by the diamond Figure [8.1]. The choice to stop is made by sampling from a Bernoulli distribution using this probability.

Apart from the decision of the ‘Add candidate’ network, rules can also be added
in this part of the generation process. These rules can be used to ensure certain
conditions on the output Petri net. One such rule that is added is connectedness,
meaning that every transition should have at least one incoming and one outgoing
place connected to it, which is needed for obtaining a valid Petri net.

At each step when a new candidate is to be chosen, the network has a score
for each candidate place and silent transition, which are not already chosen. These
scores are first softmaxed to normalize them such that they can be interpreted as
probabilities. Then one candidate is selected according to either sampling from a
distribution using these probabilities or according to a specified selection method,
e.g. always choose the candidate with highest probability. The reason for this
optional selection method is explained in the next section.

As mentioned before, since the model is trained using teacher forcing, there is a
discrepancy between training and inference causing exposure bias. Techniques have
been proposed to mitigate this problem during training, which are investigated in
[28]. These techniques include mixing in predictions during training and including
multiple candidates causing the network to learn to search for solution. However,
these are not experimented within our approach. A technique to mitigate the
consequences of this exposure bias during inference is beam search.

8.4.1 Beam search

Beam search is the last piece of the puzzle and is necessary for a similar reason
teacher forcing is used, namely to mitigate the consequences of choosing wrongly.
When sampling the choices from a multinomial distribution as described in Section
8.4, the possibility exists that a wrong candidate is chosen. All subsequent choices
are affected by such mistake, causing the joint probability in the end not to be
maximized. A greedy approach like always selecting the candidate with the highest
probability does not resolve this problem, since the probabilities of choices are
affected by previous choices (see Equation 8.1). Choosing the highest probability
candidate in the first step may not always lead to the highest joint probability at
the end of the sequence. Furthermore, exploring all possible combinations of choices
to find the maximum joint probability is intractable, since it grows exponentially. A
greedy approach that aims at exploring the most likely combinations is called beam
search [10]. Beam search is often used in sequence generation problems like sentence
generation. It is obvious that based on the first chosen word, the subsequent words
differ.

Beam search is a heuristic method of exploring the choice sequences to try to find
the solution with the highest joint probability. It operates by going different routes
in its choices, illustrated in Figure 8.3, where \( c_{ij} \) denotes the choice at step \( i \) with \( j^{th} \)
highest probability. Each path from start to a leaf node is a solution. In order to not
explore the whole tree, \( j \) is generally a relatively small number denoted as \( k \) and can
be seen as the branching factor in Figure 8.3. At every step during the beam search
the current joint probability can be computed by multiplying every probability of
the chosen candidates. Note that this joint probability always decreases every step.
By keeping track of the highest probability the search can be pruned when it falls
below this value.
An added value of applying beam search during inference is that not only the process model with the highest joint probability can be outputted, but also for example the process model having the second highest joint probability. Since multiple process models could exist that describe the actual process well, but in different ways, the preference can be determined by the user.

### 8.4.1.1 Length normalization

Maximizing the joint probability also gives rise to a problem though, which is caused by the fact that probabilities are multiplied causes the joint probability to always decrease when new choices are added. Therefore, even when using beam search, shorter solutions are generally preferred over longer ones, although this is not always the best solution. Length normalization is proposed in [44] to account for the fact that solutions of different lengths are compared, which takes the average of the log probabilities of the choices rather than the product. Heuristically, an $\alpha$ parameter is added as an exponent to the solution length. The score function $s$ of a sequence of choices $V_{\text{chosen}}^{(cand)}$ is then:

$$s(V_{\text{chosen}}^{(cand)}) = \frac{1}{n^{\alpha}} \sum_{i=1}^{\left| V_{\text{chosen}}^{(cand)} \right|} \log p(v_i|v_1, \ldots, v_{i-1}) \quad (8.14)$$

with $\alpha = 0.7$, empirically determined.

In the following chapters, this approach is evaluated, starting in Chapter 9 with an evaluation regarding the generated synthetic data to analyze to what degree it shows similar patterns as real-life data. Then the method’s performance with regard to both the synthetic data as well as the real-life data is analyzed in Chapters 10 and 11 where in the latter a comparison is done with the state-of-the-art methods.
Part IV

Evaluation
Chapter 9

Synthetic data

Our machine learning method is trained on synthetically generated data as described in Chapter 6. However, its task is eventually to produce process models for real-life event logs. In order for the machine learning model to properly generalize to real-life scenarios, it is important that this synthetic data closely represents real-life data. This chapter concerns the first experimental evaluation, where we analyze this generated synthetic data by comparing it to real-life datasets. This focuses on the second part of research question RQ3, where the validity of the synthetic data is questioned. These real-life datasets are also later used for the evaluation of our method. This real-life data consists of only the event logs without the corresponding process models. Therefore, also only the event logs from our synthetic data is analyzed. The chapter starts, in Section 9.1, by stating the objective of the experiments and what is to be measured. Then, the experimental setup is elaborated on in Section 9.2 followed by an explanation on how it is executed in Section 9.3. Section 9.4 shows the results which are interpreted and discussed in the last section, Section 9.5.

9.1 Objective

We want to create a synthetic dataset as is described in Chapter 6, using simple distributions for transitions’ execution times and choices when K multiple transitions are enabled, namely the uniform distribution $U(0,1)$ and the categorical distribution $\text{Cat}(K)$ respectively. As is touched upon at the end of that chapter, the question now is whether this results in realistically looking data. The objective of the experiments in this chapter is to see to what degree the event logs are valid to use as training data. Requirements of this are:

- The synthetic event logs should be complete in the sense that every directly follows relation is present in the event log, since otherwise, not all candidate places will be constructed during the data preparation phase;
- The synthetic event logs should show similar patterns in trace variant frequencies as real-life event logs;
- The synthetic event logs should show similar patterns in deviations found in the deviating behavior as real-life event logs.

The first requirement is checked before starting training and ignores the event log
- process model pair when this is not fulfilled. The other two are analyzed in this chapter. Statistics from both the synthetic dataset and various real-life datasets about the trace variant frequencies will be compared as well as how less frequent traces deviate from the main, most frequent, behavior. Note that deviations here are potentially part of the wanted behavior which should be captured by the process model as they show how the infrequent traces deviate from the main behavior.

9.2 Experimental setup

The experiments consist of first creating a synthetic dataset with process models from which event logs are generated. These event log - process model pairs form the training data where the event log is the input and the process model the target output. A tool implemented in ProM called the PTAndLogGenerator is utilized to generate synthetic process models. The event logs from this synthetic dataset as well as some from real-life data are used for the rest of the experiments. Eleven event logs from real-life processes are taken from the Business Process Intelligence challenge held annually during the BPM conference. These logs are publicly available at the 4TU Centre for Research Data.

We want to measure to what degree the synthetic event logs show similar patterns in the trace variant frequencies and in the deviations as the real-life event logs. For the former, statistics about the event logs are compared by doing a linear regression resulting in an $\alpha$ value that tells something about the distributions of the trace frequencies in the data. This $\alpha$ value is compared across the datasets manually. For the latter, the deviations are gathered by comparing the deviating behavior to the main behavior and are analyzed by fitting a geometric distribution for each dataset. The distribution parameter together with the error of the fit are used to get an idea on how patterns in the deviations are across both real-life and synthetic data.

9.3 Execution

The execution of the experiments in this evaluation consists of first creating the synthetic dataset, after which the trace variant frequencies and the deviating behavior are analyzed. These steps are separately elaborated on in the following sections.

9.3.1 Synthetic dataset generation

The synthetic data consists of event log - process model pairs used to train the machine learning model. The process of generating this dataset consists of the following steps:

- Generate process trees: the synthetic process trees are generated using the plugin called PTAndLogGenerator with the following parameters:
  - number of nodes: mode: 8, minimum: 4, maximum: 15;
- structures: sequence: 40%, choice: 32%, parallelism: 20%, loop: 8%, or: 0%.
- number of silent transitions: mean: 4;
- number of trees: 3000.

- **Convert process trees to Petri nets**: ProM is used to convert the process trees to Petri nets.
- **Remove abundant silent transitions**: the conversion from process trees to Petri nets results in an abundance of silent transitions. This is due to the recursive nature of the conversion where each subtree is converted to a Petri net. Silent transitions can offer a robust way to merge them. However, they do not add to the behavior of the Petri net, hence they are abundant and can be removed while preserving all behavior simplifying the Petri net. Again, ProM is used to do so.
- **Label silent transition that can not be modeled**: as discussed in Section 7.3.3, our approach can only model silent transitions with exactly one incoming and one outgoing place. Furthermore, all places should have at least one incoming and one outgoing transition connected to it, even after removing all silent transitions in the Petri net. So it does not make sense to include silent transitions not adhering to these two rules, hence they are labeled with a unique letter.
- **Limit the number of labeled transitions**: from the design of our machine learning model, there is a limit to how many distinct activities can be captured. This is due to the node feature initialization being one-hot vectors. So the number of distinct activities that can be captured is limited by the number of neurons in the input layer. In our case, this is set to 21. With two reserved for the artificial start and end transitions and one for the silent transitions, eighteen are left for activities. The dataset of Petri nets is filtered on this number, every Petri net exceeding eighteen number of transitions, not considering silent transitions, is removed.
- **Generate corresponding event logs**: rather than using the PTANDLOGGENERATOR to generate event logs from the process, our own simulation script is written to do so, since we modified the Petri nets from the process trees. Each Petri net is simulated 500 times, producing 500 traces starting and ending in the initial and final markings for each Petri net. The time it takes for an activity to fire is sampled from a uniform distribution $U(0, 1)$. When $K$ multiple transitions are enabled during a run, one is chosen by sampling from a categorical distribution $\text{Cat}(K)$, with uniform probabilities $p_1 = p_2 = \cdots = p_K = \frac{1}{K}$. Only the distinct trace variants together with their respective frequencies are saved.

### 9.3.2 Trace variant frequency analysis

The trace variant frequencies are assumed to fit a power law distribution and are therefore plotted against three scales: $(x, y), (x, \log y), (\log x, \log y)$. A power law distribution describes data where a small number of trace variants contain a large number of traces, which is what we observed in the event logs. When the data
actually follows this distribution, the \((\log x, \log y)\) plot should show a linear line. Linear regression is then performed on this plot to approximate the \(\alpha\) parameter of the power law distribution together with its error. This is done on both the synthetically generated event logs and the real-life event logs.

Recall that our synthetic process models have a maximum of eighteen distinct activity names, so in order to perform a fair comparison, the real-life event logs are also filtered such that only the eighteen most frequent activities are retained.

Furthermore, the event logs from the BPI challenges of 2012 and 2017 contain multiple different processes, which can be differentiated by the first letter of the activities, which are prefixed by \(A\), \(O\) and \(W\). For both datasets from 2012 and 2017, two separate event logs are created by filtering out only the \(A\) prefixed activities and only the \(O\) prefixed activities.

### 9.3.3 Deviating behavior analysis

For the analysis on deviating behavior, the trace variants from the 80% most frequent traces are considered to be the main behavior of the process \(L_{\text{main}}\). The remaining 20% is considered as deviating behavior \(L_{\text{dev}}\). This separation is used because a power law distribution follows the Pareto principle where 80% of the traces comes from 20% of the trace variants, and vice versa.

In order to analyze how the deviating behavior deviates from the main behavior, each trace variant \(T_d\) from the deviating traces is compared to each of those from the main behavior \(T_m\). This comparison is done using the Damerau Levenshtein distance which is a string metric for measuring the edit distance between two sequences.

For each trace variant \(T_d \in L_{\text{dev}}\), the trace variant \(T_m^{\text{closest}} \in L_{\text{main}}\) closest to \(T_d\) is found to get pairs \((T_d, T_m^{\text{closest}})\) satisfying:

\[
\forall T_m \in L_{\text{main}} \quad \delta(T_d, T_m^{\text{closest}}) \leq \delta(T_d, T_m) \tag{9.1}
\]

where \(\delta\) denotes the Damerau Levenshtein distance.

These pairs \((T_d, T_m^{\text{closest}})\) are taken to see where \(T_d\) deviates exactly from \(T_m^{\text{closest}}\) looking at insertions, substitutions, omissions, repetitions and transpositions, of which simple examples are given:

- insertion: \(\langle a, b, c, d \rangle\) deviates from \(\langle a, d \rangle\) by inserting subtrace \(\langle b, c \rangle\);
- substitution: \(\langle a, d, c \rangle\) deviates from \(\langle a, b, c \rangle\) by substituting \(b\) with \(d\);
- omission: \(\langle a, d \rangle\) deviates from \(\langle a, b, c \rangle\) by omitting subtrace \(\langle b, c \rangle\);
- repetition: \(\langle a, b, b, c \rangle\) deviates from \(\langle a, b, c \rangle\) by repeating \(b\);
- transposition: \(\langle c, b, a, d \rangle\) deviates from \(\langle a, b, c, d \rangle\) by transposing events \(a\) and \(c\).

Patterns in these deviations are then analyzed to compare the synthetic data with the real-life data. Specifically, the occurrences of deviations are assumed to be random variables from a geometric distribution. This assumption is based on the shape of the data, when plotted. An example shown in Figure 9.1 clearly shows an exponential decay. Note that the geometric distribution is the discrete variant of the exponential distribution. The geometric distribution is fitted on the data for each dataset resulting in the shape parameter \(p\) which is 1 divided by the mean
Figure 9.1: Histogram plot showing the occurrences of the distinct deviations.

Figure 9.2: Relative occurrences of deviations at distinct indices and at the main traces.

of the data. A value measuring the quality of this fit is computed by taking the
sum of squared errors (SSE), i.e. \[ \sum_{i=1}^{n} (c(d_i) - G(i, p))^2, \]
with \( c(d_i) \) the number of occurrences of deviation \( d_i \) and \( G(i, p) = (1 - p)^{i-1}p \) the geometric distribution.

Deviations are counted by clustering them in three ways:

- distinct deviations: all different deviations as listed above, e.g. insertion of subtrace \( \langle a, b \rangle \);
- deviations at distinct indices: all distinct places, i.e. indices, in the traces from the main behavior where deviations are found;
- deviations at main traces: all distinct traces from the main behavior where deviations are found.

The clustered deviation data for one arbitrarily selected event log (BPI 2017 O\(_{-}^e\)) is visualized in Figures 9.1-9.2. Figure 9.1 shows the number of occurrences of the distinct deviations that are found in the deviating traces in the form of a histogram. Figure 9.2 shows the main traces where each horizontal line is a trace.
variant $T_m \in L_{\text{main}}$ and each dot is an event in the trace with its activity name right below it. The percentages at the right denote the number of deviations, relative to the number of deviating traces $|L_{\text{dev}}|$, found in the respective main traces. The percentages shown below each event denote the number of deviations, again relative to $|L_{\text{dev}}|$, found at those respective indices.

Only the results of those where the distribution is fitted on at least eight datapoints are taken into account, to ensure that distribution fitting makes sense. Datapoints in the context refer to either the distinct deviations (bars in Figure 9.1), the distinct indices (events with nonzero percentage below in Figure 9.2) or the main traces for the different ways of clustering. The results are plotted on a scatter plot with the shape parameter on the x-axis and the SSE on the y-axis, both for the synthetic and real-life datasets.

### 9.4 Results

The analysis results of the trace variant frequencies for four different event logs are plotted in Figure 9.3. The first two subfigures, Figure 9.3a and 9.3b, are from respectively the Road Traffic Fine and Sepsis real-life datasets, arbitrarily selected. The latter two subfigures, Figure 9.3c and 9.3d, are from two randomly selected synthetic event logs. In each subfigure, the left plot shows the original trace variant frequencies, with on the x-axis the variant and on the y-axis the number of traces for each variant. The middle two show the same data but with scaled axes. The first has only the y-axis as a log scale and the second has both the x- and y-axis as log scale. The right plot in each subfigure shows the result of the linear regression on the log-log scale, with the $\alpha$ and its error in the title. Some background information about the $\alpha$ parameter in power-law distributions is that when it lies between 1 and 2, the mean of the frequencies is in theory infinite while the variance is finite. When $\alpha$ lies between 2 and 3, this is the other way around. This can be seen in the most left subplots in Figure 9.3 where with an $\alpha < 2$, the scale on the y-axis exceeds the scale on the x-axis, and with $\alpha > 2$, vice versa is true.

Figure 9.4 shows the same analysis but for all event logs, where each point in the scatter plot corresponds to an event log. The blue points correspond to real-life event logs and are annotated accordingly. The red points, on the other hand, are synthetically generated event logs. A point in the scatter plot shows the $\alpha$ parameter and error of the fitted power law distribution.

The rest of the results regard the analysis of the deviations in the non-frequent traces.

Figure 9.5 shows a scatter plot where each point again corresponds to an event log, with blue and red points for real-life and synthetic event logs respectively. A point in the scatter plot shows the shape parameter and error of the fitted geometric distribution on the occurrences of specific deviations, where a deviation could be for example the insertion or deletion of a specific subtrace.

The other clusterings of deviations, as introduced in Section 9.3 are the deviations in specific traces from the main behavior and deviations at specific indices in the main traces. The occurrences of deviation in these clusters are also fitted to
Figure 9.3: Trace variant frequencies with alpha estimates (RTF, Sepsis, Synth 1 and 2).
Figure 9.4: Trace variant frequencies scatter plot on $\alpha$’s and main/deviating behavior ratio.

Figure 9.5: Fitting geometric distributions on deviations.
geometric distribution, resulting in a shape parameter and a fitting error for each event log. The points in the scatter plot from Figure 9.6 show these shape parameters and errors for the fitted distributions the occurrences of deviations at specific indices in the main traces. Furthermore, the plot is similar to that shown in Figure 9.5.

Figure 9.7 is again similar and shows a scatter plot where each point corresponds to the shape parameter and error of the fitted geometric distributions on the occurrences of deviations in the traces of the main behavior.

9.5 Discussion

The objective of this analysis is to see whether patterns in the event logs from the synthetic dataset are similar to those in real-life event logs. We have two hypotheses to check this statement:

- The first hypothesis is that the trace variant frequencies of synthetically generated event logs show similar patterns as those from real-life event logs. The fitted distributions, as shown in Figure 9.3, seem to verify this hypothesis. The approximated $\alpha$ values of the assumed power law distributions together with relative number of main behavior trace variants for all datapoints as shown in Figure 9.4 indicate that the distributions of trace variants are very similar to those of real-life event logs. Apart from a few outliers, all synthetic event logs lie within the convex hull of the blue points corresponding to real-life event logs. This convex hull is the convex polygon whose vertices are some of the blue points such that all blue points lie within this polygon.
- The second hypothesis regards the deviations and states that the deviating
behavior deviates from main behavior in similar ways in the synthetic and real-life event logs. Figures 9.5, 9.6 and 9.7 verify this, since again most points from synthetic data lie within the convex hull of the points from real-life data. Furthermore, by looking at the y-axes, the fitted geometric distributions seem reasonable since the sum squared errors are all low. Note that no rigorous analysis is done on the fitted distributions, like acquiring p-values, since this simple analysis is assumed to be sufficient for making conclusions about similarity. A last observation from the figures is that the gross of the points both for synthetic data and real-life data lie in the same area, close to the (0,0)-point.

By looking more into the outliers in the scatter plots, i.e. points not around the majority of points, we saw that these are mostly from very simple process models resulting in a very small number of trace variants and deviations.

In conclusion, although uniform distributions are used both for the execution times of activities and choosing between multiple enabled transitions, the patterns in the data are very similar to those of real-life data. This is especially true for the not very simple processes.
Chapter 10

Training/testing phase

Having the validated synthetic dataset from Chapter 9 consisting of event log-process model pairs, a machine learning model is trained as described in Chapter 8. Recall that the process models in the synthetic dataset are simple block-structured process models and the corresponding event log are cleanly simulated, i.e. without any added noise or deviations. The machine learning model is therefore trained to perform process discovery on a relatively simple dataset. Research question RQ2.2 addresses the accuracy of this task and the generalization to unseen event logs, which are evaluated in this chapter. It has the same structure as Chapter 9 starting with stating the objective, explaining the experimental setup and execution in Sections 10.1, 10.2 and 10.3. Then the results are shown in Section 10.4 which are again interpreted and discussed in Section 10.5 ending the chapter.

10.1 Objective

The objective of this experiment is to analyze how well the proposed method behaves on our synthetic data. This is done by answering four evaluation questions:

EQ1 The evolution of the machine learning model during training in terms of several metrics is observed. The metrics are the loss, as described in Section 8.3 as well as the true positives and false negatives of the classified candidate nodes (places and silent transitions).

EQ2 A deeper analysis is done on the incorrectly classified places by incorporating the Alpha relations about the input data. The objective here is to gain knowledge on why the model makes certain mistakes, e.g. is information in the data missing to make the correct decision or is superfluous information available in the data causing noise and therefore a wrong decision.

EQ3 Furthermore, the quality of the machine learning model after training is analyzed, with regard to the synthetic data, in terms of the process discovery specific conformance metrics, like fitness and precision.

EQ4 Lastly, in order to be able to support the choice of adding the attention mechanism to the graph convolutional neural network used in our model, the attentions are analyzed by looking at the entropy that they hold.
10.2 Experimental setup

The machine learning model as described in Chapter 8 is used for training on the synthetic dataset from Chapter 9, which is split into a train and test dataset, such that the model can not learn from the latter. Recall that the synthetic dataset consists of event log $L$ - process model $M$ pairs. The train and test dataset are non-overlapping subsets of the complete dataset. The train and test dataset consist of 2000 and 663 $(L, M)$ pairs respectively. The quality of the model is measured during training in terms of loss as described in Section 8.3, the relative true positives and false positives (EQ1).

After training, the incorrectly classified places are further analyzed by incorporating the Alpha relations as described in Section 2.3. Note that this analysis is only done for places and not for silent transitions. For each incorrectly classified place, the corresponding Alpha relations are analyzed to see whether enough information was available to deduce the correct decision or whether superfluous information was available causing a wrong classification. Furthermore, for each mistake, other places are checked to see if they contain or are contained in the incorrectly classified place. This notion of containment for two places $p_1$ and $p_2$ is shown in Figure 10.1 and is defined as follows:

$$p_1 \text{ contains } p_2 \iff p_2 \text{ is contained in } p_1 \iff T_{in}^{(p_2)} \subseteq T_{in}^{(p_1)} \land T_{out}^{(p_2)} \subseteq T_{out}^{(p_1)}$$

(10.1)

where $T_{in}^{(p)}$ and $T_{out}^{(p)}$ denote respectively the incoming and outgoing transitions of $p$. Percentages from these analyses are gathered to shed light on the mistakes the machine learning model made (EQ2).

Moreover, the conformance metrics, like fitness, precision, generalization and simplicity, are computed for the trained machine learning model on the synthetic data, showing how well the model behaves on the actual task we want it to solve (EQ3).

Lastly, to analyze the utilization of attention mechanism, the entropies of the attention scores for each node are computed, where the entropy $H$ of a node $i$ is measured by $H(i) = -\sum_{j \in N_{in}(i)} \alpha_{ji} \log \alpha_{ji}$, where $\alpha_{ji}$ is the weight on edge from node $j$ to $i$ (EQ4).

10.3 Execution

The first part of the experiments (EQ1) regards the training of the model itself. The model is trained on the synthetic dataset, described in Section 9.3.1, for 100 epochs and each epoch keeping track of the statistics like loss, true/false positives and false negatives. To define what these are, we first define the set of candidates that are present in the true process model as $V_{(true)}^{(true)}$ and the set of candidates chosen by our method as $V_{(chosen)}^{(chosen)}$ and the set of candidates chosen by our method as $V_{(chosen)}^{(chosen)}$. Then:

- true positives: $V_{(true)}^{(true)} \cap V_{(chosen)}^{(chosen)}$
- false negatives: $V_{(true)}^{(true)} \setminus V_{(chosen)}^{(chosen)}$, i.e. the candidates that should be chosen, but are not;
false positives: $V_{\text{candidate}}^{(\text{chosen})} \setminus V_{\text{candidate}}^{(\text{true})}$, i.e. the candidates that are chosen, but should not have been chosen.

Recall from Section 8.4, there is a mismatch between training and inference because of the use of teacher forcing during training. During training, teacher forcing is used, so the choices are corrected while generating the Petri net. Therefore, after an event log – process model pair has been processed, we do not have the actual prediction but only the loss value. To acquire the actual predicted Petri net and its chosen candidates, inference is performed during the training. Note that this is not necessary for training, but just for gathering the statistics for this evaluation.

A maximum of thirty traces from the event logs in the dataset are used for a less time-consuming training process. Three statistics are computed as described before:

- loss $l = -\sum_{i=1}^{C'} \log p(c_1, \ldots, c_{i-1})$;
- the ratio between the number of true positives and false negatives;
- the ratio between number of false positives and number of places.

After training, the predicted Petri nets are compared to the true Petri nets, also incorporating the Alpha relations from the corresponding event logs. For each incorrectly classified place, the directly follows relations of the corresponding input and output transitions are taken to check for under- and overspecification (EQ2).

For example, for a place $p_1$, as shown in Figure 10.1a, from transition $a$ to transitions $b$ and $c$ and directly follows relations $a >_L b, a >_L d, d >_L c$, the latter two relations are considered overspecification and $a >_L c$ is considered underspecification. Furthermore, all places are checked whether they contain or are contained in incorrectly classified places. For example, the place $p_2$, as shown in Figure 10.1b, from transition $a$ to $b$ is contained in $p_1$, since $T_{\text{in}}^{(p_2)} \subseteq T_{\text{in}}^{(p_1)}$ and $T_{\text{out}}^{(p_2)} \subseteq T_{\text{out}}^{(p_1)}$.

The conformance metrics fitness, precision, generalization and simplicity are computed by using standard functionality from the process mining library for Python called PM4Py [5]. The metrics are all calculated based on token-based replay where an event log and a Petri net are analyzed to compute the metrics from [26]. This is done for each event log and predicted Petri net, as well as the true Petri net. Histograms of the ratios of the scores of the predicted process models and true models are plotted, as well as histograms of the scores themselves for the predicted process models (EQ3).

For the attention mechanism, the attention weights are analyzed in each layer of the graph convolutional neural network during inference for each head of the multi-head attention mechanism (EQ4). The constructed graph, as described in
Section 7 is plotted where the edges are colorized, corresponding to the respective attention weights. Furthermore, for each node $i$, its attention entropy is computed by $H(i) = -\sum_{j\in \mathcal{N}(i)} \alpha_{ji} \log \alpha_{ji}$ as described before. The entropies are plotted in a histogram. Another histogram is plotted for the case where no attention mechanism would be added, i.e. with uniform weights on the edges.

## 10.4 Results

Figure 10.2 and 10.2b show results regarding EQ1. Figure 10.2a shows the loss, relative number of true places classified correctly and the ratio of false positives and true places. The epochs corresponding to the rounds of training are on the x-axis for each plot. The plotted lines show the average of the measured statistics across all datapoints in the dataset. This is extended in Figure 10.2b, where box plots are also added for every fifth epoch, showing other various statistics across the datapoints such as median, outliers, minimum, maximum and the interquartile range.

Moving on to EQ2, the statistics of under- and overspecified of correctly (true positives) and incorrectly (false negatives and false positives), classified places are shown in Table 10.1. Here, the statistics are given about four cases of being exact (not under-, nor overspecified), being only underspecified (under), being only overspecified (over) and both being under- and overspecified (under+over). The numbers themselves correspond to the number of occurrences relative to the total number of places, i.e. true places and false positive ones. What stands out in this table is that the percentages are similar for most classes. Differences can be seen in the large number of false negatives and false positives that are overspecified for the test data. For the training data, the percentage of false negatives being under- and overspecified is higher than that for the test data.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Exact</th>
<th>Under</th>
<th>Over</th>
<th>Under+Over</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TP</td>
<td>FN</td>
<td>FP</td>
<td>TP</td>
</tr>
<tr>
<td>training</td>
<td>74.4%</td>
<td>5.8%</td>
<td>52.1%</td>
<td>1.3%</td>
</tr>
<tr>
<td>test</td>
<td>75.7%</td>
<td>5.9%</td>
<td>34.2%</td>
<td>1.0%</td>
</tr>
</tbody>
</table>

Table 10.1: Under-/overspecifiedness for correctly and incorrectly classified places.

Statistics of the other analysis regarding the incorrectly classified places are shown in Table 10.2. These statistics concern whether these places are contained in or contain other false and true positives.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Incorrect</th>
<th>False positives</th>
<th>True positives</th>
<th>Unknown cause</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Contained in</td>
<td>Contains</td>
<td>Contained in</td>
<td>Contains</td>
</tr>
<tr>
<td>training</td>
<td>4.9%</td>
<td>14.1%</td>
<td>13.9%</td>
<td>2.3%</td>
</tr>
<tr>
<td>test</td>
<td>6.0%</td>
<td>23.6%</td>
<td>15.8%</td>
<td>2.2%</td>
</tr>
</tbody>
</table>

Table 10.2: Analysis of incorrectly classified places.

Moving on to the conformance checking results on the same data with the trained model to answer EQ3. For each event log - process model pair, the conformance
Figure 10.2: Training and testing curves of loss, % true positives and % of false positives.
metrics are computed for both the true Petri net as well as the predicted Petri net. The fitness and precision metrics best reveal how well a process model conforms to an event log. Therefore, the FScore, which is a weighted average between fitness and precision, is taken to visualize the quality of the predicted Petri nets. Rather than just showing the FScore of the predicted Petri nets, the ratio between the FScores of the predicted Petri nets and the FScores of the corresponding true Petri nets is taken, since this better shows how close the machine learning model is to find the correct Petri net. A histogram of these ratios is shown in Figure 10.3 both for the training and the test data. For clarity, the bars show the frequencies of these ratios, with the ratio on the x-axis and the normalized frequency on the y-axis. Histograms showing the frequencies of the FScores themselves for the predicted and ground truth process models of both training and test data can be found in Figure E.1. Figures E.2 and E.3 show similar histogram but for the fitness and precision conformance metrics respectively.

The remainder of this section regards EQ4. As shown in Figure 8.1, two propagation networks are present in the machine learning model. The input layers of both networks are taken to show the edge weights computed by the attention mechanism in Figures 10.4a and 10.4b. Figure 10.4a shows only forward edges, where Figure 10.4b shows backward edges. The darker the color of the edge, the more weight it has. The corresponding scale is shown at the right for each Figure. Furthermore, red nodes are those from the trace graph as described in Section 7.2 and the purple and blue nodes correspond respectively to the transitions and candidate places in the Petri net graph as described in Section 7.3. Note that the positions of the nodes are automatically generated and hold no information.

Lastly, the entropies of the attention weights as described previously are shown in histogram plots in Figures 10.5 and 10.6 again for both networks. Each row of plots correspond to a layer in the network and the different plots in the rows are the different attention heads. The last row contains the entropies of the attentions of the single-headed attention mechanism from the output layer as well as the histogram of the data where no attention mechanism was used, so with a uniform distribution.
10.5 Discussion

To answer the first evaluation question (EQ1), we look at the training curves in Figure 10.2a, where we can see that the model improves steadily over time, whereas the rate of improvement lowers over time. Furthermore, there is a match between the training and test data which suggests that the model is initially not overfitting on the training data. What you often see when training a machine learning model on training and test data, is that there is a point where the loss is still decreasing only for the training data. This point can be seen as the point in time where the model starts to overfit on the training data, not being able to generalize enough to also improve on the test data. In our case, this can also be seen in both the loss and the percentage of true positives.

However, this is not true for the percentage of false positives, which could be due to the mismatch between training and inference. The effect of using teacher forcing during training is overfitting on the loss function and true positives, but not on the false positives. The false positives are due to an early wrong choice or a mistake in the network for deciding to stop adding candidates. Lastly, the loss does not converge to 0, even for the training data. This is due to the fact that the chosen breadth first search ordering in choosing the candidates is not a canonical ordering as explained in Section 8.3.1. Note that this is not necessarily a bad thing but rather an explanation for this observation. The machine learning model will generalize by learning different orderings when present and assigns different probabilities to these orderings, corresponding to how they are represented in the data.

Next, we look at Tables 10.1 and 10.2 to answer the second evaluation question (EQ2). In Table 10.1, some interesting differences between incorrectly classified places from the training and test dataset come up. The false negatives from the training set are mostly under- and overspecified, while those from the test set are
Figure 10.5: Attention entropies network 1.
mostly only overspecified. Mistakes caused by overspecifiedness could mean that
the model has difficulty in knowing what information is relevant for making the
decision. Furthermore, it is interesting to see that a large portion of false positives
are neither under- nor overspecified, showing that more information is needed than
these directly follows relations, from the Alpha relations, to make decisions on the
existence of places. Note that, as specified in Table 10.2, the true positive places
take up respectively 95.1% and 94% in the training and test dataset.

In Table 10.2 we see similar numbers for the training and test dataset. Interest-
ing to see is that almost half of the incorrectly classified places either contain
or are contained in another place, either correct or incorrect. This suggests that
with some post-processing, these could be located in order to potentially reduce the
mistakes by half.

For EQ3, Figure 10.3 shows that the FScores of the predicted Petri nets lie very
close to the intended FScores from the true Petri nets. There is clearly a difference
in training and test data. The predicted Petri nets from the training are more often
exactly the same as the true Petri nets. In the test data, on the other hand, often a
little difference exists between the predicted and true Petri nets, but as can be seen
from the histogram plot, where the highest bars are on the far right, the FScores
of the predicted Petri nets do approach those of the true Petri nets. This shows
that there is a close connection between the conformance metrics and the accuracy
measures, where the results for both the training and test data are good, with those
of the latter slightly worse than those of the former.

To end this chapter, a conclusion is drawn about the usage of the attention
mechanism in the machine learning model (EQ4). First of all, from the graphs of

Figure 10.6: Attention entropies network 2.
the samples with visualized weights on the edges in Figures 10.4a and 10.4b you can see a clear difference in the usage. The first propagation network does not use the attention mechanism very much in the input layer while the second propagation network does, as can be seen by the single dark edge. The nodes’ entropies are further visualized in Figures 10.4a and 10.4b for both networks. The histograms of the entropies from the learned attentions shown in the figures, clearly deviate from those of the entropies from uniform attentions. They deviate in the sense that the entropy scores are much lower on the majority of nodes when using a learnable attention mechanism. Lower entropy scores mean that indeed information is gained from using a non-uniform distribution of weights on the edges. This is especially true for the later layers in the networks.
Chapter 11

Evaluation phase

Having a machine learning model that is trained to solve the task of process discovery properly on synthetic data as discussed in Chapter 10, we evaluate our proposed method for the intended task of unsupervised process discovery on real-life data. In this chapter, the quality of our proposed method is analyzed on real-life data to show that it competes with state-of-the-art methods for process discovery while requiring fewer assumptions, answering research questions RQ1.1 and RQ1.2. The analysis differs from the one in Chapter 10 in the sense that no statistics are gathered on the classified candidates since no ground truth process models are available for the real-life datasets. Recall that process discovery is an unsupervised learning problem in the sense that the optimal process model is unknown. The quality of a process model with regard to the event log is evaluated through the domain-specific conformance metrics being fitness, precision, generalization and simplicity. Therefore, a more rigorous qualitative analysis on the discovered process models themselves is done in this evaluation than that of the evaluation in Chapter 10.

The chapter has again the same structure as Chapter 9 and 10, starting with stating the objective, explaining the experimental setup and execution in Sections 11.1, 11.2 and 11.3. Then the results are shown in Section 11.4 which are again interpreted and discussed in Section 11.5 ending the chapter.

11.1 Objective

This evaluation focuses on analyzing our approach on the intended task of unsupervised process discovery on real-life data. The objective is initially to see whether our approach actually generalizes beyond the training data. Recall that the training data consists solely of block-structured process models with cleanly simulated event logs. Although we showed in Chapter 9 that similar patterns can be found in this synthetic data and the real-life data, the training data is still very limited. The synthetic training data is limited in complexity consisting of only block-structured process models and clean event logs, i.e. without added deviations or noise. Real-life processes are often not as structured, such that they can be modeled perfectly using only block-structured process models. Furthermore, the real-life event logs often contain deviations and/or noise.

Conclusions on the generalization beyond training data can be made by eval-
uating the results on real-life data. These results also enable us to compare our approach to the state-of-the-art methods (RQ1.2), both quantitatively in terms of conformance metrics and qualitatively to find patterns where our approach is superior and inferior as compared to the algorithmic approaches. To analyze whether our approach can compete with the state-of-the-art methods is the second objective. The results on real-life data also enable us to compare the performance of our proposed method regarding synthetic data and real-life data.

Furthermore, to make this analysis complete, a comparison between the quality of the different methods on synthetic data is done to get an idea on how our proposed method compares to the state-of-the-art methods regarding the task it is trained to do.

Having a competing method with our approach, we want to know how much the number of assumptions is reduced (RQ1.1).

Lastly, as mentioned in RQ1.2, the running times are analyzed.

### 11.2 Experimental setup

The data used for the evaluation part is the same as that used for the evaluation of our synthetic data, as described in Section 9.2. To analyze the quality of the results of our approach and compare them to those of the state-of-the-art methods, the conformance metrics fitness, precision, generalization and simplicity are computed, again using built-in functionality of the PM4Py library [5] which calculates these metrics based on token-replay. The Heuristics Miner, Inductive Miner, ILP Miner and Split Miner are taken as state-of-the-art methods. For the first, again PM4Py is used. ProM is used to get results from the ILP Miner and the Inductive Miner. A standalone Java package is used for the Split Miner [1].

The quantitative analysis is done using the conformance metrics of each method, by comparing the computed values manually.

This quantitative analysis enables us to compare the performance of our method on synthetic data and real-life data. Besides the real-life dataset, the synthetic test dataset, as described in Section 10.2, is taken to compute conformance metrics for each method. Note that the test dataset contains 663 event log - process models pairs which the machine learning model has not used for updating its parameters, making the comparison between the various method fair.

Furthermore, a qualitative analysis is done for a subset of the evaluation datasets by manually investigating the produced process models themselves, namely from the 2012 and 2017 BPI challenge, the A_ and O_ prefixed processes respectively. These are chosen since they are the smallest process models that show the strengths and weaknesses of our approach.

Both the quantitative and qualitative analysis are used to discuss to what degree our approach makes assumptions about the data and to what degree it performs adequately without such assumptions.

Lastly, the efficiency is analyzed by measuring the execution time in terms of preparing the data as well as the decision making process.

11.3 Execution

The real-life evaluation event log data that was already preprocessed as described in Section 9.3 is used here to run the different process discovery methods on. As mentioned in the previous section, for the state-of-the-art algorithmic methods, either PM4Py, ProM or a standalone Java package is used. For all methods, the default settings are used. For the usage of our approach, only the traces from the 90% most frequent behavior are taken into account with a minimum of 30 and a maximum of 75 trace variants. These limits are parameters that are empirically determined. The lower limit is set to 30 to enlarge the variety of trace variants when the 90% most frequent behavior consists of only a few trace variants. The upper limit is set to 75 to limit the number of candidate places and therefore the memory usage and running times. Furthermore, for the beam search, a width of three is used and the six process models are produced of which the machine learning model is most certain. These are also empirically chosen parameters to balance between quality and performance in terms of running times.

With all process models discovered, the corresponding event logs are taken to compute the conformance metrics using the PM4Py library again. From the six produced models, the one with best conformance is kept. The results are shown both in tables and in scatter plots. For two results, the actual process models are shown discovered by the best state-of-the-art method and our method for further qualitative analysis.

Process models for all event logs from the synthetic test dataset are produced using all methods. Here, our method is used without beam search to speed up the process. The conformance metrics for all produced process models are computed, also using the PM4Py library.

For a subset of the datasets, the execution times are measured for the data preparation and each decision making step in the iterative generation process of which the mean is computed.

11.4 Results

The outcomes of the conformance metrics for all discovery methods on every evaluation dataset can be found in Tables 11.1 and 11.2. Both for the datasets and discovery methods, abbreviations are used and listed briefly here. The discovery methods are Heuristics Miner (HM), Split Miner (SM), ILP Miner (ILP), Inductive Miner (IM) and the datasets are road traffic fine (RTF), Sepsis (Sepsis), 2012 and 2017 BPI challenge A and O activities (’12A, ’17A, ’12O, ’17O), 2020 BPI challenge domestic declarations, international declarations, permit log, prepaid travel cost and request for payment (’20DD, ’20ID, ’20PL, ’20PTC, ’20RFP).

The FScore is a weighted average of fitness and precision. Note that for each dataset, the methods are sorted according to the order of this FScore. The highest score for each metric is in bold. The rows with results from our approach are emphasized. Table 11.3 shows each method’s average scores with standard deviations for each of the conformance metrics, again sorted by FScore. The last column in
Table 11.3 denotes the average on the conformance metrics, i.e. fitness, precision, generalization and simplicity.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>FScore</th>
<th>Fitness</th>
<th>Precision</th>
<th>Generalization</th>
<th>Simplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTF</td>
<td>HM</td>
<td>0.959</td>
<td>0.929</td>
<td>0.991</td>
<td>0.897</td>
<td>0.548</td>
</tr>
<tr>
<td></td>
<td>SM</td>
<td>0.956</td>
<td>0.916</td>
<td><strong>1.000</strong></td>
<td>0.987</td>
<td>0.615</td>
</tr>
<tr>
<td></td>
<td>ILP</td>
<td>0.936</td>
<td>0.888</td>
<td>0.990</td>
<td>0.997</td>
<td>0.245</td>
</tr>
<tr>
<td></td>
<td><strong>Ours</strong></td>
<td>0.912</td>
<td>0.838</td>
<td>0.999</td>
<td>0.873</td>
<td>0.660</td>
</tr>
<tr>
<td></td>
<td>IM</td>
<td>0.738</td>
<td><strong>0.983</strong></td>
<td>0.590</td>
<td>0.914</td>
<td>0.655</td>
</tr>
<tr>
<td>Sepsis</td>
<td>SM</td>
<td><strong>0.943</strong></td>
<td>0.892</td>
<td><strong>1.000</strong></td>
<td>0.894</td>
<td><strong>0.650</strong></td>
</tr>
<tr>
<td></td>
<td>ILP</td>
<td>0.883</td>
<td>0.807</td>
<td>0.975</td>
<td><strong>0.968</strong></td>
<td>0.139</td>
</tr>
<tr>
<td></td>
<td><strong>Ours</strong></td>
<td>0.812</td>
<td>0.683</td>
<td><strong>1.000</strong></td>
<td>0.896</td>
<td><strong>0.541</strong></td>
</tr>
<tr>
<td></td>
<td>HM</td>
<td>0.791</td>
<td>0.880</td>
<td>0.719</td>
<td>0.783</td>
<td>0.522</td>
</tr>
<tr>
<td></td>
<td>IM</td>
<td>0.700</td>
<td><strong>0.924</strong></td>
<td>0.564</td>
<td>0.935</td>
<td>0.610</td>
</tr>
<tr>
<td>'12A</td>
<td><strong>Ours</strong></td>
<td>0.958</td>
<td>0.919</td>
<td><strong>1.000</strong></td>
<td>0.933</td>
<td><strong>0.600</strong></td>
</tr>
<tr>
<td></td>
<td>HM</td>
<td>0.946</td>
<td>0.898</td>
<td><strong>1.000</strong></td>
<td>0.897</td>
<td>0.654</td>
</tr>
<tr>
<td></td>
<td>ILP</td>
<td>0.907</td>
<td>0.877</td>
<td>0.939</td>
<td><strong>0.988</strong></td>
<td>0.203</td>
</tr>
<tr>
<td></td>
<td>SM</td>
<td>0.901</td>
<td>0.821</td>
<td><strong>1.000</strong></td>
<td>0.985</td>
<td>0.818</td>
</tr>
<tr>
<td></td>
<td>IM</td>
<td>0.759</td>
<td><strong>0.996</strong></td>
<td>0.613</td>
<td>0.983</td>
<td>0.700</td>
</tr>
<tr>
<td>'12O</td>
<td>HM</td>
<td>0.942</td>
<td>0.890</td>
<td><strong>1.000</strong></td>
<td>0.905</td>
<td>0.640</td>
</tr>
<tr>
<td></td>
<td><strong>Ours</strong></td>
<td>0.941</td>
<td>0.888</td>
<td><strong>1.000</strong></td>
<td>0.880</td>
<td><strong>0.680</strong></td>
</tr>
<tr>
<td></td>
<td>SM</td>
<td>0.918</td>
<td>0.848</td>
<td><strong>1.000</strong></td>
<td>0.982</td>
<td><strong>0.867</strong></td>
</tr>
<tr>
<td></td>
<td>ILP</td>
<td>0.917</td>
<td>0.871</td>
<td>0.968</td>
<td><strong>0.984</strong></td>
<td>0.245</td>
</tr>
<tr>
<td></td>
<td>IM</td>
<td>0.673</td>
<td><strong>0.961</strong></td>
<td>0.518</td>
<td>0.982</td>
<td>0.684</td>
</tr>
<tr>
<td>'17A</td>
<td>HM</td>
<td>0.985</td>
<td>0.971</td>
<td><strong>1.000</strong></td>
<td>0.949</td>
<td>0.673</td>
</tr>
<tr>
<td></td>
<td>IM</td>
<td>0.965</td>
<td><strong>0.999</strong></td>
<td>0.933</td>
<td>0.992</td>
<td>0.730</td>
</tr>
<tr>
<td></td>
<td><strong>Ours</strong></td>
<td>0.960</td>
<td>0.925</td>
<td>0.999</td>
<td><strong>0.992</strong></td>
<td><strong>0.786</strong></td>
</tr>
<tr>
<td></td>
<td>SM</td>
<td>0.947</td>
<td>0.899</td>
<td><strong>1.000</strong></td>
<td>0.884</td>
<td>0.756</td>
</tr>
<tr>
<td></td>
<td>ILP</td>
<td>0.939</td>
<td>0.885</td>
<td><strong>1.000</strong></td>
<td><strong>0.994</strong></td>
<td>0.199</td>
</tr>
<tr>
<td>'17O</td>
<td>SM</td>
<td><strong>0.944</strong></td>
<td>0.893</td>
<td><strong>1.000</strong></td>
<td>0.937</td>
<td>0.714</td>
</tr>
<tr>
<td></td>
<td>ILP</td>
<td>0.926</td>
<td>0.868</td>
<td>0.994</td>
<td><strong>0.994</strong></td>
<td>0.247</td>
</tr>
<tr>
<td></td>
<td>HM</td>
<td>0.914</td>
<td>0.846</td>
<td>0.994</td>
<td>0.931</td>
<td>0.530</td>
</tr>
<tr>
<td></td>
<td><strong>Ours</strong></td>
<td>0.889</td>
<td>0.804</td>
<td>0.995</td>
<td><strong>0.982</strong></td>
<td><strong>0.677</strong></td>
</tr>
<tr>
<td></td>
<td>IM</td>
<td>0.838</td>
<td><strong>0.927</strong></td>
<td>0.765</td>
<td>0.992</td>
<td>0.750</td>
</tr>
</tbody>
</table>

Table 11.1: Conformance metrics for real-life datasets part 1/2.

Figures 11.1, 11.2 and 11.3 highlight some of the conformance metrics. Figure 11.1 shows a scatter plot with the FScore and Simplicity metrics on the y- and x-axis respectively where the points correspond to a discovery method and dataset pair. The different datasets are separated by the shape and the methods are shown by the colors as is listed in the plot’s legend.

Figures 11.2 and 11.3 have the same setup and show respectively the conformance metrics of fitness versus precision and fitness versus generalization.

Figures 11.4 and 11.5 show similar plots but for the synthetic test dataset. The small dots refer to values of the conformance metrics of all (L, M) pairs for every method. The colors denote which method is used and is shown in the plots’ legends. Note that here the ground truth results are also shown, since we have the corresponding process models from which the event logs are generated. The
<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>FScore</th>
<th>Fitness</th>
<th>Precision</th>
<th>Generalization</th>
<th>Simplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>20DD</td>
<td>SM</td>
<td>0.962</td>
<td>0.927</td>
<td>1.000</td>
<td>0.787</td>
<td>0.586</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>0.945</td>
<td>0.895</td>
<td>1.000</td>
<td>0.953</td>
<td>0.611</td>
</tr>
<tr>
<td></td>
<td>ILP</td>
<td>0.918</td>
<td>0.895</td>
<td>0.942</td>
<td>0.990</td>
<td>0.245</td>
</tr>
<tr>
<td></td>
<td>HM</td>
<td>0.897</td>
<td>0.930</td>
<td>0.867</td>
<td>0.764</td>
<td>0.620</td>
</tr>
<tr>
<td></td>
<td>IM</td>
<td>0.638</td>
<td>0.971</td>
<td>0.475</td>
<td>0.765</td>
<td>0.651</td>
</tr>
<tr>
<td>20ID</td>
<td>SM</td>
<td>0.949</td>
<td>0.903</td>
<td>1.000</td>
<td>0.973</td>
<td>0.714</td>
</tr>
<tr>
<td></td>
<td>HM</td>
<td>0.922</td>
<td>0.893</td>
<td>0.953</td>
<td>0.879</td>
<td>0.589</td>
</tr>
<tr>
<td></td>
<td>ILP</td>
<td>0.895</td>
<td>0.866</td>
<td>0.925</td>
<td>0.987</td>
<td>0.199</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>0.852</td>
<td>0.773</td>
<td>0.950</td>
<td>0.969</td>
<td>0.662</td>
</tr>
<tr>
<td></td>
<td>IM</td>
<td>0.708</td>
<td>0.903</td>
<td>0.582</td>
<td>0.923</td>
<td>0.765</td>
</tr>
<tr>
<td>20PL</td>
<td>SM</td>
<td>0.956</td>
<td>0.916</td>
<td>1.000</td>
<td>0.947</td>
<td>0.699</td>
</tr>
<tr>
<td></td>
<td>HM</td>
<td>0.906</td>
<td>0.871</td>
<td>0.944</td>
<td>0.892</td>
<td>0.596</td>
</tr>
<tr>
<td></td>
<td>ILP</td>
<td>0.899</td>
<td>0.853</td>
<td>0.950</td>
<td>0.988</td>
<td>0.199</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>0.867</td>
<td>0.781</td>
<td>0.975</td>
<td>0.979</td>
<td>0.614</td>
</tr>
<tr>
<td></td>
<td>IM</td>
<td>0.672</td>
<td>0.935</td>
<td>0.524</td>
<td>0.980</td>
<td>0.686</td>
</tr>
<tr>
<td>20PTC</td>
<td>SM</td>
<td>0.952</td>
<td>0.909</td>
<td>1.000</td>
<td>0.864</td>
<td>0.692</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>0.925</td>
<td>0.860</td>
<td>1.000</td>
<td>0.910</td>
<td>0.543</td>
</tr>
<tr>
<td></td>
<td>HM</td>
<td>0.885</td>
<td>0.901</td>
<td>0.871</td>
<td>0.697</td>
<td>0.594</td>
</tr>
<tr>
<td></td>
<td>ILP</td>
<td>0.867</td>
<td>0.879</td>
<td>0.855</td>
<td>0.974</td>
<td>0.166</td>
</tr>
<tr>
<td></td>
<td>IM</td>
<td>0.637</td>
<td>0.928</td>
<td>0.485</td>
<td>0.927</td>
<td>0.687</td>
</tr>
<tr>
<td>20RFP</td>
<td>SM</td>
<td>0.966</td>
<td>0.934</td>
<td>1.000</td>
<td>0.822</td>
<td>0.600</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>0.929</td>
<td>0.867</td>
<td>1.000</td>
<td>0.812</td>
<td>0.588</td>
</tr>
<tr>
<td></td>
<td>ILP</td>
<td>0.918</td>
<td>0.900</td>
<td>0.937</td>
<td>0.988</td>
<td>0.245</td>
</tr>
<tr>
<td></td>
<td>HM</td>
<td>0.899</td>
<td>0.941</td>
<td>0.860</td>
<td>0.691</td>
<td>0.603</td>
</tr>
<tr>
<td></td>
<td>IM</td>
<td>0.594</td>
<td>0.970</td>
<td>0.428</td>
<td>0.773</td>
<td>0.681</td>
</tr>
</tbody>
</table>

Table 11.2: Conformance metrics for real-life datasets part 2/2.

<table>
<thead>
<tr>
<th>Method</th>
<th>FScore</th>
<th>Fitness</th>
<th>Precision</th>
<th>Generalization</th>
<th>Simplicity</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM</td>
<td>0.945</td>
<td>0.986</td>
<td>1.000</td>
<td>0.915 ± 0.066</td>
<td>0.701 ± 0.084</td>
<td>0.900</td>
</tr>
<tr>
<td>HM</td>
<td>0.913</td>
<td>0.905</td>
<td>0.927 ± 0.085</td>
<td>0.844 ± 0.089</td>
<td>0.597 ± 0.047</td>
<td>0.818</td>
</tr>
<tr>
<td>ILP</td>
<td>0.910</td>
<td>0.892</td>
<td>0.952 ± 0.039</td>
<td>0.807 ± 0.089</td>
<td>0.212 ± 0.035</td>
<td>0.791</td>
</tr>
<tr>
<td>Ours</td>
<td>0.906</td>
<td>0.839</td>
<td>0.993 ± 0.015</td>
<td>0.926 ± 0.054</td>
<td>0.633 ± 0.067</td>
<td>0.859</td>
</tr>
<tr>
<td>IM</td>
<td>0.720</td>
<td>0.954</td>
<td>0.589 ± 0.138</td>
<td>0.924 ± 0.078</td>
<td>0.691 ± 0.043</td>
<td>0.776</td>
</tr>
</tbody>
</table>

Table 11.3: Average conformance metrics with standard deviations.
Figure 11.1: Real-life evaluation datasets, FScore/simplicity.

Figure 11.2: Real-life evaluation datasets, fitness/precision.
large dots show the average scores of all \((L, M)\) pairs for the different methods, to visualize how the different methods perform on average on the synthetic data. Figure 11.4 shows the conformance metrics of FScore versus simplicity and Figure 11.5 shows those of fitness versus precision.

As mentioned in the experimental setup in Section 11.2, we compare our discovered process model to the best performing state-of-the-art methods with regard to the conformance metrics on the real-life evaluation datasets from the 2012 and 2017 BPI challenge, \(A\) and \(O\) prefixed processes respectively. Figures 11.6 and 11.7 show the process models from the BPI 2012 \(A\) event log produced by the Heuristic Miner and our approach respectively.

Figures 11.8 and 11.9 show the process models from the BPI 2017 \(O\) event log produced by the Split Miner and our approach respectively.

The running times for datasets Road Traffic Fine, Sepsis, BPI 2012 and 2017 (\(A\)) and BPI 2020 Permit log are shown in Table 11.4. Here, beam search is not used.

11.5 Discussion

In this section we discuss the results as presented in the previous section to achieve the objectives as stated in Section 11.1. The discussion is presented here in separate subsections, starting with the quantitative analysis on the real-life datasets, after which the conformance metrics regarding the synthetic test datasets are compared.
Figure 11.4: Synthetic datasets, FScore/simplicity.

Figure 11.5: Synthetic datasets, fitness/precision.
Figure 11.6: Model discovered by Heuristics Miner from the 2012 BPI challenge (A).

Figure 11.7: Model discovered by our approach from the 2012 BPI challenge (A).

Figure 11.8: Model discovered by Split Miner from the 2017 BPI challenge (O).

Figure 11.9: Model discovered by our approach from the 2017 BPI challenge (O).

<table>
<thead>
<tr>
<th>Dataset</th>
<th># candidate places</th>
<th>Graph construction (s)</th>
<th>Propagation w/ decision (mean s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTF</td>
<td>296</td>
<td>0.343</td>
<td>0.543</td>
</tr>
<tr>
<td>Sepsis</td>
<td>2450</td>
<td>6.658</td>
<td>3.960</td>
</tr>
<tr>
<td>T2A</td>
<td>297</td>
<td>0.401</td>
<td>0.753</td>
</tr>
<tr>
<td>T7A</td>
<td>58</td>
<td>0.110</td>
<td>0.126</td>
</tr>
<tr>
<td>20PL</td>
<td>700</td>
<td>5.192</td>
<td>1.456</td>
</tr>
</tbody>
</table>

Table 11.4: Running times of our approach for a few real-life event not using beam search.
The remaining parts are the qualitative analysis and the discussion on the running times.

11.5.1 Quantitative analysis

The conformance metrics give a reasonably good overview on how well the produced process models fit the corresponding event logs. So, by first looking at these quantitative results in Tables 11.1 and 11.2 and the scatter plots in Figures 11.1 and 11.3, our approach seems to perform adequately. The results of the different conformance metrics are analyzed separately regarding our approach versus the state-of-the-art methods. From Tables 11.1, 11.2 and 11.3 we can conclude the following:

- **FScore**: for the FScore, it often comes close to the state-of-the-art methods, once even exceeding them all and five times coming up second. On average, it falls 0.039, 0.007, 0.004 short of the better scoring methods;
- **Fitness**: fitness seems to be where our approach is lacking, especially for the Sepsis dataset. For the other datasets, it does not actually fall behind by a lot, namely at most 0.15 to the best scoring method. On average, our method performs the lowest on this regard;
- **Precision**: the precision metric is always very high, with a minimum of 0.950. Always approaching or exceeding the other methods. On average, only the Split Miner outperforms our approach, which always has a perfect precision score;
- **Generalization**: in terms of generalization, our method always scores high. Only the ILP Miner scores higher on average. The Split Miner and Inductive Miner comes close but score a bit lower;
- **Simplicity**: for simplicity, our method always comes close to the best scoring method for every dataset. On average, our method is the third highest scoring method with the Inductive Miner and Split Miner exceeding it.

In comparison, looking at Table 11.3, the Inductive Miner excels at fitness (0.954), with the trade-of of having low precision (0.589). It does, however, produce sound models. Producing only block-structured process models, it also scores high on the simplicity metric (0.691). The ILP Miner excels at generalization (0.987), however, producing very complex, unsound model causing the simplicity to lack a lot (0.212). The Heuristics Miner scores well on the fitness and precision metrics (0.905 and 0.927), although it produces rather complex models, with an average simplicity of 0.597. With regards to generalization, the Heuristics Miner falls behind all other methods (0.844). The Split Miner comes on top on all conformance metrics except on generalization, where it performs second to worst (0.915).

The method of our approach relates with that of the Inductive Miner and the ILP Miner. For the former, this is in the sense that our machine learning model has only seen block-structured process models during the learning process as the Inductive Miner is designed to produce. For the latter, both our method and the ILP Miner construct a process model by discovering places that glue the transitions together. In terms conformance metrics, our method never exceeds all other methods, although it performs above average for every individual metrics. This can be seen in the last column where the conformance metrics are averaged. Our method places
second, only behind the Split Miner, when taking into account all conformance metrics (0.859 and 0.900 respectively). This means that it balances nicely within the metrics, although regarding fitness it performs worst of all methods (0.839).

### 11.5.2 Synthetic data vs real-life data

First, we compare the results of our approach for the synthetic test data and the real-life data, as shown in Figures 11.4 and 11.1. We can see that our method performs a lot better on the synthetic data which it has learned to do. The average FScore is around 0.95 for synthetic data and around 0.85 for real-life data. Furthermore, the average simplicity is around 0.8 for synthetic data and around 0.6 for real-life data. This could mean that either the task it is trained on is easier than the task on real-life data or the machine learning model does not generalize well beyond the training task.

To answer whether the former or the latter is the case, we look at how the other methods perform on the same synthetic data. First of all, we can see that most methods also perform much better on this data than on the real-life data. This suggests that the training task is actually easier than the task on real-life data as is expected, since our synthetic data consists of limited data in terms of process model structure and noise/deviations in the event log. Since the conformance metrics on real-life data are competing with the state-of-the-art methods, we can conclude that the machine learning model generalizes well beyond the synthetic data.

Interesting to see from Figures 11.4 and 11.5 is that our method actually outperforms all other methods on the synthetic test data, since its mean scores are closest to those of the ground truth models.

### 11.5.3 Qualitative analysis

Conformance metrics, however, do not say everything about the quality of the produced models though, since they are computed by approximation methods. Furthermore, the conformance metric are calculated using token-based replay, rather than the more accurate calculation based on alignments. Reason for this is that our method does not guarantee weakly sound models, which is required for using alignment-based conformance metrics.

For a proper evaluation, a qualitative analysis is necessary, which is done by looking at the produced process models of several methods themselves. The analysis is done for two datasets: the A\_ prefixed process of the 2012 BPI challenge and the O\_ prefixed process of the 2017 BPI challenge. These two are chosen since they are the simplest processes that clearly show the differences between the different methods.

**2012 BPI challenge (A\_):** For the event log from the A\_ prefixed process of the 2012 BPI challenge, the process model discovered by our approach is shown in Figure 11.7. Here, some interesting patterns are visible:

- Starting by the complex places following A\_PARTLY\_SUBMITTED. Some kind of combination of parallelism and a choice construct seems to be discovered here to include behavior from activities A\_CANCELLED and A\_DECLINED. Such
cancellation activities turn out to be our method’s Achilles’ heel currently, since such activity can generally occur anywhere in the process and ends it completely. In this process, activity A\_DECLINED is similar. With a parallelism construct, A\_CANCELLED can actually happen anywhere during the process. However, it does not imply that the process ends after this happens. Looking at the silent transitions, this is actually modeled in the process model to allow for stopping anywhere in the process. The Heuristics Miner, as shown in Figure 11.6, on the other hand, implemented the cancellation in another way with silent transitions to stop the process ending with A\_CANCELLED. Similarly, A\_DECLINED is modeled. Note that this construct can actually not be exploited by our approach since we only allow silent transitions between places that already have at least one incoming and outgoing transition, not counting silent transitions.

• The event log suggests that the three activities A\_APPROVED, A\_ACTIVITED and A\_REGISTERED occur in parallel, as is modeled by the Heuristics Miner. Our method did not capture this as such, but modeled it in sequence with a loop to allow A\_ACTIVATED to occur after A\_REGISTERED, although this construct makes the process model unsound. The incorrect modeling of parallelism here could be due to how our method modeled the cancellation using these silent transitions, which is difficult to model in combination with parallelism.

2017 BPI challenge (O\_): Looking at the produced models for the O\_ prefixed process of the 2017 BPI challenge in Figures 11.8 and 11.9 we see that our method captures patterns not captured by the better performing Split Miner:

• The cycle back to the start from various places in the process. Our method modeled this correctly using silent transitions.

• An interesting difference is in the O\_CANCELLED activity, which according to the Split Miner cannot occur after O\_SENT (ONLINE ONLY), although this is allowed by the model from our method and can actually be seen in the event log as well.

• Mistakes in the model from our method can be found on the right side of the model, where firstly, an unsound construct occurs in that the two incoming places from transition O\_REFUSED can not both contain tokens simultaneously. Secondly, the far right place should be part of the final marking, since after O\_REFUSED, no other transition is able to fire. Lastly, after O\_RETURNED there should be a path towards the upper part of the model with activities O\_ACCEPTED and O\_CANCELLED, according to the event log.

Regarding research question RQ1.1, we can see from the produced process models that they are not block-structured showing that our approach does not have this structural assumption even though the seen process models from the training data have this property.

A drawback is that our approach does not guarantee soundness. Often this is due to a improper split and join constructs, where a choice construct is used in combination with a parallelism construct, by having a place with multiple outgoing edges that join at a transition with multiple incoming edges. A minimal example of such improper construct is shown in Figure 11.10a and an example of this happening...
(a) Unsound construct as it is modeled.

(b) Sound construct as it should be modeled.

Figure 11.10: Example of unsound construct and how it should be modeled.

can be found in Figure 11.7 as discussed around transitions \texttt{A\_ACTIVATED} and \texttt{A\_REGISTERED}. This behavior should, however, be modeled using a silent transition as shown in Figure 11.10b. The machine learning model has no knowledge about soundness and is not penalized for using unsound constructs.

11.5.4 Running times

To end the chapter, the running times are touched upon. A side note here is that the implementation of beam search, as described in Section 8.4.1, is not optimized in any way causing the model to recompute the choices for every path in the beam search tree. Though in general, the running time of a single process model prediction is based on two of factors, firstly the size of the graph with the main aspect the number of candidate places that are constructed and secondly the number of places and silent transitions that are to be chosen. Note that the number of candidates increases as more places are chosen, since after each choice several candidate silent transitions are added to the graph. As can be seen from Table 11.4 there is a large difference between different datasets, namely from 0.1 seconds up to 6.7 seconds for the graph construction and 0.1 up to 4.0 seconds for information propagation. For the smaller graphs, with less than 700 candidate places, however, the running times are reasonable. Lastly, note that no GPU is used in our experiments, which could potentially greatly reduce the running times of the propagation networks for the large graphs, since GPUs excel at large matrix computations because of their large number of cores.

All in all, these experiments show that machine learning can definitely be used to tackle the task of process discovery. The results show that our method using machine learning generalizes beyond what it has seen in during training. Within the design of our method, our approach can only model one transition per distinct activity and the only types of silent transitions that can be modeled are those connecting places that already have at least one incoming and one outgoing labeled transition, as defined in 7.3.3 and illustrated in Figure 7.6.

These can be seen as two assumptions about the process model structure in the sense that this provides sufficient expressiveness to model the processes. Note that former assumption is also made by every state-of-the-art method. Apart from these assumptions, we can conclude that no assumptions are made on the structure of the process models as other methods do, since the produced process models are not block-structured although all previously seen data has this structural property.
Furthermore, we saw that our method performs on par with state-of-the-art methods on possibly noisy event log data, even though its previously seen data was clean, showing that no explicit assumptions are required about infrequent data as the other methods do.

This evaluation showed that our approach using machine learning is able to mitigate problems of algorithmic approaches, being data- and statistics-driven rather than based on assumptions, although, there exist important patterns that are not discovered by our approach, leaving room for improvement.
Part V

Conclusion
Chapter 12

Conclusion

The main message of this thesis is that the task of process discovery can be tackled with fewer assumptions than the current approaches do by exploiting machine learning techniques. This chapter concludes the thesis by summing the contributions to the field of process discovery in Section 12.1. Some of the proposed method’s limitations are highlighted in Section 12.2. Potential ways of solving or mitigating these limitations are left for future work and are described in Section 12.3.

12.1 Contributions

In this thesis, machine learning is introduced to the process discovery problem and an approach using machine learning is proposed and developed to solve it. This new approach balances between data-driven and assumption-based problem solving, resulting in close to state-of-the-art results in terms of conformance metrics. The results are not perfect, leaving room for improvements which is elaborated on in the next sections.

All in all, a framework is developed to express a machine learning model that produces multiple process model solutions for a given event log. An expert can then decide on which fits best according to their preferences. In the development of this framework, no manually annotated data is used so the learning process can be completely automated, by generating process trees of different sizes and constructs distributions, simulating them and training the network. This continuous learning process can enable the machine learning model to continuously gain knowledge and experience to solve the task of process discovery. Although the machine learning model only learns about the subclass of process models being process trees, it can generalize beyond them as can be seen in the results. Furthermore, process trees have nice interpretable structures by nature, making them an ideal candidate for training data. The training data solely contains clean event logs while the results show that our machine learning model knows how the handle the noisy data in the real-life event logs. This noise is normally handled by filtering which is based on assumptions. Our method does not need these assumptions to determine what information is relevant and what is not.

This thesis opens up the field for other researchers to see if machine learning methods can be used more in the field of process discovery and hopefully serves as
12.2 Limitations of the proposed method

Although our proposed method performs adequately, it still has some limitations. First of all, within its design, there is a maximum number of distinct activities that can be modeled by the method. This is a drawback, caused by how the nodes are initialized in the data preparation phase, using one-hot vectors, making it unscaleable to larger event logs. This maximum number is limited by the largest process model in the training data, which in our case is 18.

Furthermore, a major drawback is that the method does not ensure to produce sound process models, neither on the real-life data nor on the clean synthetic data, although this synthetic data is all the machine learning model has seen during training and its process models are all sound.

From the qualitative analysis in Section 11.5, a few patterns, like cancellation, are highlighted with which our method has difficulties to model. Since such patterns are not rare in real-life event logs, this is a concerning limitation, though it can possibly be mitigated by including such patterns in the training data.

Lastly, as mentioned in Section 8.3.1, breadth first search for deciding on the order of the candidates to choose does not offer a completely canonical ordering causing the machine learning model not to converge to zero loss.

12.3 Future work

The several limitations, as addressed in the previous section, already offer improvements for future work. Some ideas on solving or mitigating them are presented in here. First of, the implementation of the method has a lot of room for improvement in its efficiency aspect. Especially the beam search algorithm is not yet optimized. Saving states and reusing those in the heuristic search could greatly decrease the computation times.

The other major issue regarding the limitation on the maximum number of distinct activities requires a different approach in the node feature initialization where at the moment one-hot embeddings are used. Embedding the input allowing for a scalable number of activities has to be explored to solve this limitation. Methods exists to do so, but whether they are as good as one-hot encoding is unclear.

With regard to the breadth first search not being completely canonical, an idea is to let the machine learning model choose multiple candidates at each step, referring to all candidates at a specific depth in the breadth first search tree, since this is completely deterministic.

In order to ensure soundness on the produced process models, an extra rule, similar to the one ensuring connectedness, could be added during the generation process. Different from the connectedness rule here, however, is that soundness can not always be satisfied by just adding more candidates. Furthermore, adding such rules potentially increases the search space causing the machine learning model to do a lot of solution exploring, for which it is not performing well enough as of yet.
To end this section, several techniques that could potentially improve the performance are proposed. Recurrent neural networks, which are often used when dealing with sequential data, could be introduced in the graph convolutional neural network, potentially increasing the ability to capture the sequential information from traces. Lastly, instead of producing the Petri net by choosing candidates, the reverse generation process could be investigated, removing candidates at each step. This reduces the graph after each choice and at a certain point in time, with the result that only the final Petri net is left.
Bibliography


## List of Figures

1.1 BPM lifecycle, taken from [9]. ........................................ 8
2.1 A simple Petri net example. ........................................ 13
2.2 Simple Petri net examples showing the different constructs. .... 15
2.3 A simple process tree example. ..................................... 16
3.1 Flower model. .......................................................... 19
3.2 Venn diagram showing the notion of fitness and precision. ...... 19
3.3 Trace model for traces from event log $L_r$. ....................... 20
3.4 Venn diagram showing the notion of generalization. ............ 20
3.5 Example Petri nets. ..................................................... 21
3.6 Minimal example of improper split and join construct. .......... 22
3.7 Complex control-flow Petri net from log $L = \{\langle a, c, d \rangle, \langle b, c, e \rangle\}$. ..... 24
4.1 Illustration of the workings of a GCN, taken from [13]. .......... 29
5.1 Roadmap to our approach. ............................................. 35
7.1 Simplified overview of the constructed graph for event log $\{\langle a, b \rangle^4, \langle a, b \rangle^4\}$. 40
7.2 Trace graph for event log $L_r$. .................................... 41
7.3 Start Petri net graph example. ...................................... 44
7.4 Petri net graph with candidate places example. ................. 46
7.5 Many-to-many place example. ...................................... 48
7.6 Illustration of minimal requirement for silent transition candidate. 49
8.1 Overview of the sequential model after two rounds. ............ 52
8.2 Partial illustration of information propagation through the graph for one node. ......................................................... 54
8.3 Beam search with branching factor $k = 3$. ........................ 61
9.1 Histogram plot showing the occurrences of the distinct deviations. 67
9.2 Relative occurrences of deviations at distinct indices and at the main traces. ............................................................ 67
9.3 Trace variant frequencies with alpha estimates (RTF, Sepsis, Synth 1 and 2). .......................................................... 69
9.4 Trace variant frequencies scatter plot on $\alpha$’s and main/deviating behavior ratio. ......................................................... 70
9.5 Fitting geometric distributions on deviations. ..................... 70
9.6 Fitting geometric distributions on deviations in main trace indices. 71
9.7 Fitting geometric distributions on deviations in main traces.

10.1 Example places \( p_1, p_2 \): \( p_1 \) contains \( p_2 \) \( \iff \) \( p_2 \) is contained in \( p_1 \).

10.2 Training and testing curves of loss, % true positives and % of false positives.

10.3 Training and testing conformance FScore ratio.

10.4 Attention weights on the edges in both propagation networks.

10.5 Attention entropies network 1.

10.6 Attention entropies network 2.

11.1 Real-life evaluation datasets, FScore/simplicity.

11.2 Real-life evaluation datasets, fitness/precision.

11.3 Real-life evaluation datasets, fitness/generalization.

11.4 Synthetic datasets, FScore/simplicity.

11.5 Synthetic datasets, fitness/precision.

11.6 Model discovered by Heuristics Miner from the 2012 BPI challenge (A).

11.7 Model discovered by our approach from the 2012 BPI challenge (A).

11.8 Model discovered by Split Miner from the 2017 BPI challenge (O).

11.9 Model discovered by our approach from the 2017 BPI challenge (O).

11.10 Example of unsound construct and how it should be modeled.

A.1 Processing an event log with an RNN extracting information at event level.

A.2 Processing an event log with an RNN extracting information at event level example.

C.1 Overview of the discriminative model.

E.1 Histogram showing the FScore metric for predictions and ground truths for training and test dataset.

E.2 Histogram showing the fitness metric for predictions and ground truths for training and test dataset.

E.3 Histogram showing the precision metric for predictions and ground truths for training and test dataset.
List of Tables

2.1 Example event log. ............................................. 13
2.2 Example Alpha relations matrix. .............................. 16
3.1 Summary of assumptions in state-of-the-art methods. ......... 25
10.1 Under-/overspecifiedness for correctly and incorrectly classified places. 76
10.2 Analysis of incorrectly classified places. ......................... 76
11.1 Conformance metrics for real-life datasets part 1/2. .......... 86
11.2 Conformance metrics for real-life datasets part 2/2. .......... 87
11.3 Average conformance metrics with standard deviations. ....... 87
11.4 Running times of our approach for a few real-life event not using beam search. .................................. 91
Part VI
Appendices
Appendix A

Alternate approaches

A.1 RNN input handling

An RNN processes the complete event log as one long trace, with separator values between the traces, and then extracting the hidden states of the RNN cells and multiplying them with a matrix such that the hidden states of the distinct events come together to then either take the sum or average from. This way, an RNN can process the event log and output information for each distinct event with a matrix multiplication. This process is illustrated in Figure A.1.

An example with event log \( L = \{\langle a, c, b, d \rangle, \langle a, b, c, d \rangle\} \) that better illustrates this is shown in Figure A.2. The hidden states, corresponding to the events in the RNN are \( h = [h_a, h_c, h_b, h_d, h_{a2}, \ldots] \). While in the end we want information about each distinct event, so by multiply this vector with a matrix \( L_b \), we get \( L \times L_b = \left[ \frac{h_a + h_{a2}}{2}, h_b, h_c, h_d \right] \) with \( L_b \):

\[
L_b = \begin{bmatrix}
.5 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
.5 & 0 & 0 & 0 \\
\vdots
\end{bmatrix}
\]

This approach of input handling focuses on using an RNN to handle the input

![Figure A.1](image1.png)

Figure A.1: Processing an event log with an RNN extracting information at event level.
data to eventually end up with embeddings for each distinct event \( a \) that captures sequential information for every occurrence of that event \( a \) in the event log. This can be information about what events occur before \( a \), but also after \( a \) when using a bi-directional RNN. These event embeddings can then be used in for example a graph convolutional neural network (GCN), elaborated on in Section 4.2.1, to decide on how the process model should be constructed.

### A.2 Variational autoencoder for process model generation

Graph convolutional neural networks are used for link prediction \[47\] which can be exploited for process discovery. In \[16\], typed link prediction is performed with the use of a variational autoencoder, which is the first generative approach that we explored. Here, the distinct events from the event log form the Petri net transitions and are the nodes of the graph for our GCN. A variational autoencoder, as described in 4.2.6, consists of an encoder and a decoder. The aim of the encoder is to represent the complex distribution of the input data to a simple distribution in a latent space. The decoder can then sample from this latent space to generate new, unseen, data points. In our approach, the encoder is a GCN that learns distributions on the input graphs’ edges. The edges are classified corresponding to which place it connects, e.g. with transitions \( t_1 \), \( t_2 \) and \( t_3 \) and edge classifications \( e_{12} = 1 \), \( e_{13} = 1 \), a place \( p_1 \) is predicted going from transition \( t_1 \) to transitions \( t_2 \) and \( t_3 \). Another example with edge classifications \( e_{12} = 1 \), \( e_{13} = 2 \) puts two places \( p_1 \) and \( p_2 \) going from transition \( t_1 \) to \( t_2 \) and from \( t_1 \) to \( t_3 \) respectively.

These distributions on the edges between transitions corresponding to the available places can then be used to sample a Petri net from a given event log. Advantages from retrieving the process models from the latent space are that firstly, all possible process models are captured by the latent space and secondly various likely process models could be sampled by exploring the latent space, where in one dimension, the result could focus on optimizing fitness whereas another could focus on optimizing precision.

The idea of using a variational autoencoder in this way was abandoned after too many obstacles arose. Namely that it does not matter for the resulting process model whether edges \( e_{12} \) and \( e_{13} \) are classified as 1 or 2 as long as they are classified equally. Furthermore, by representing places as classes, the approach is not scalable, since the number of classes are to be specified beforehand. With the first problem
and the large number of classes caused the network not to converge during training. Therefore, different approaches like the discriminative and sequential model have been explored.
## Appendix B

### Algorithms for constructing candidate places

#### B.1 Constructing the one-to-one places

**Algorithm B.1:** Algorithm for constructing one-to-one candidate places

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( V_{\text{place}} = \emptyset );</td>
</tr>
<tr>
<td>2</td>
<td>for trace ( T'_i = \langle &gt;, t_i1, t_i2, \ldots, t_i</td>
</tr>
<tr>
<td>3</td>
<td>for look ahead ( l ) from 1 to ( F ) do</td>
</tr>
<tr>
<td>4</td>
<td>for event ( t_{ij} \in T'_i ) do</td>
</tr>
<tr>
<td>5</td>
<td>if ( v_{t_{ij}, t_{ij+l}}^{(\text{place})} \notin V_{\text{place}} ) and ( j \leq</td>
</tr>
<tr>
<td>6</td>
<td>Add ( v_{t_{ij}, t_{ij+l}}^{(\text{place})} ) to ( V_{\text{place}} );</td>
</tr>
<tr>
<td>7</td>
<td>( \text{INITIALIZEFEATURE}(v, \text{null}, L, 0); )</td>
</tr>
<tr>
<td>8</td>
<td>( \text{ADDEdge}(u_{t_{ij}}^{(\text{transition})}, v, \text{true}); )</td>
</tr>
<tr>
<td>9</td>
<td>( \text{ADDEdge}(v, u_{t_{ij+l}}^{(\text{transition})}, \text{true}); )</td>
</tr>
<tr>
<td>10</td>
<td>end</td>
</tr>
<tr>
<td>11</td>
<td>end</td>
</tr>
<tr>
<td>12</td>
<td>end</td>
</tr>
<tr>
<td>13</td>
<td>end</td>
</tr>
</tbody>
</table>
B.2 Constructing the one-to-many and many-to-one places

Algorithm B.2: Algorithm for constructing one-to-many and many-to-one candidate places $V_{\text{place}}$.

**input:** Place nodes $V_{\text{place}}$

**Result:** Place nodes $V_{\text{place}}$ for the Petri net graph, the event log’s Alpha relations $L_{\alpha}$

1. Initialize $\text{shared}_\text{in}$ and $\text{shared}_\text{out}$ to be a mapping from sets of transitions to initially empty sets;

2. for $v_{T_{\text{in}}, T_{\text{out}}} \in V_{\text{places}}$ do
   3. Add $T_{\text{out}}$ to $\text{shared}_\text{in}[T_{\text{in}}]$;
   4. Add $T_{\text{in}}$ to $\text{shared}_\text{out}[T_{\text{out}}]$;
   5. end

6. for $T_{\text{in}} \in \text{shared}_\text{in}$ do
   7. for $T_{\text{out}} \in \text{combinations}(\text{shared}_\text{in}[T_{\text{in}}], \text{length}_\text{min} = 2)$ do
      8. if $\neg \exists t_i, t_j : t_i \in T_{\text{out}} \wedge t_j \in T_{\text{out}} \wedge t_i, t_j \in L_{\alpha}^{(\text{parallel})}$ then
         9. Add $v_{T_{\text{in}}, T_{\text{out}}}^{(\text{place})}$ to $V_{\text{place}}$;
         10. InitializeFeature($v, \text{null}, L, 0$);
         /* **Following two for-loops are used more often */
         11. for $t_i \in T_{\text{in}}$ do
             12. ADDEDGE($u_{t_i}^{(\text{transition})}, v, \text{true}$);
             13. end
         14. for $t_i \in T_{\text{out}}$ do
             15. ADDEDGE($v, u_{t_i}^{(\text{transition})}, \text{true}$);
             16. end
         17. end
     23. Add edges from $T_{\text{in}}$ to $v$ and from $v$ to $T_{\text{out}}$; // Same as above.**
   24. end
8. end
20. for $T_{\text{out}} \in \text{shared}_\text{out}$ do
   21. for $T_{\text{in}} \in \text{combinations}(\text{shared}_\text{out}[T_{\text{out}}], \text{length}_\text{min} = 2)$ do
      22. if $\neg \exists t_i, t_j : t_i \in T_{\text{in}} \wedge t_j \in T_{\text{in}} \wedge t_i, t_j \in L_{\alpha}^{(\text{parallel})}$ then
         23. Add $v_{T_{\text{in}}, T_{\text{out}}}^{(\text{place})}$ to $V_{\text{place}}$;
         24. InitializeFeature($v, \text{null}, L, 0$);
         25. Add edges from $T_{\text{in}}$ to $v$ and from $v$ to $T_{\text{out}}$; // Same as above.**
   26. end
27. end
28. end
Algorithm B.3: Algorithm for constructing many-to-many candidate places \( V_{\text{place}} \).

**Input:** Place nodes \( V_{\text{place}} \)

**Result:** Place nodes \( V_{\text{place}} \) for the Petri net graph

1. Initialize \( \text{shared}_{\text{in}} \) and \( \text{shared}_{\text{out}} \) to be a mapping from sets of transitions to initially empty sets.;

2. for \( T_{\text{in}} \in V_{\text{places}} \) do
3.   if \( |T_{\text{in}}| > 1 \) then
4.     Add \( T_{\text{out}} \) to \( \text{shared}_{\text{in}}[T_{\text{in}}] \);
5.   end
6.   if \( |T_{\text{out}}| > 1 \) then
7.     Add \( T_{\text{in}} \) to \( \text{shared}_{\text{out}}[T_{\text{out}}] \);
8.   end
9. end

10. for \( T_{\text{in}} \in \text{shared}_{\text{in}} \) do
11.   for \( T_{\text{out}} \in \text{combinations}(\text{shared}_{\text{in}}[T_{\text{in}}], \text{length}_{\text{min}} = 2) \) do
12.     if \( \exists t_i, t_j : t_i \in T_{\text{out}} \land t_j \in T_{\text{out}} \land t_i, t_j \in L_{\text{α}}^{(\text{parallel})} \) then
13.       Add \( v_{T_{\text{in}},T_{\text{out}}}^{(\text{place})} \) to \( V_{\text{place}} \); 
14.       \text{initializeFeature}(v, \text{null}, L, 0);
15.       Add edges from \( T_{\text{in}} \) to \( v \) and from \( v \) to \( T_{\text{out}} \); \quad // Same as above,**
16.     end
17.   end
18. end

19. for \( T_{\text{out}} \in \text{shared}_{\text{out}} \) do
20.   for \( T_{\text{in}} \in \text{combinations}(\text{shared}_{\text{out}}[T_{\text{out}}], \text{length}_{\text{min}} = 2) \) do
21.     if \( \exists t_i, t_j : t_i \in T_{\text{in}} \land t_j \in T_{\text{in}} \land t_i, t_j \in L_{\text{α}}^{(\text{parallel})} \) then
22.       Add \( v_{T_{\text{in}},T_{\text{out}}}^{(\text{place})} \) to \( V_{\text{place}} \); 
23.       \text{initializeFeature}(v, \text{null}, L, 0);
24.       Add edges from \( T_{\text{in}} \) to \( v \) and from \( v \) to \( T_{\text{out}} \); \quad // Same as above,**
25.     end
26. end
27. end
Appendix C

Discriminative version of our approach

The discriminative version of our approach also takes the graph, constructed as described in Chapter 7, encoding the event log and Petri net with candidate places. This approach is also inspired by the ILP Miner [39] and especially [24], where the optimal places are chosen from a set of candidates. By classifying a subset of the candidate places as ones and the rest as zeroes, a Petri net is generated. This discriminative variant, decides on the classification of each candidate place in one go, rather than one by one as the sequential variant operates.

The discriminative model is illustrated in Figure 8.1, which uses a propagation network in the form of a GCN with attention mechanism, similarly to first part of the sequential model. The difference lies in the output size of the network, i.e. the number of neurons in the last layer. So the shapes of the four weight matrices in the network manipulating the nodes’ embeddings are $W^{(1)} : C \times 32$, $W^{(2)} : 32 \times 64$, $W^{(3)} : 64 \times 32$, $W^{(4)} : 32 \times 1$, with $C$ again being the size of the initial features.

After propagation, each node embeddings consists of a single value. For the candidate places, this is the score determining which class it is.

The model is then trained as a straightforward supervised learning problem, with the dataset discussed in Section 6, to put a high score on places that should exist and a low score on places that should not exist. Since it is a binary classification problem, the binary cross-entropy loss is used:

$$l = -\frac{1}{|V_{\text{place}}|} \sum_{v \in V_{\text{place}}} y_v \log \sigma(p_v) + (1 - y_v) \log(1 - \sigma(p_v)) \quad \text{(C.1)}$$

Figure C.1: Overview of the discriminative model.
with \( V_{\text{place}} \) the set of all candidate places, \( p_v \) the probability of candidate place \( v \) and \( \sigma \) the logistic sigmoid function \( \left( \frac{e^x}{e^x + 1} \right) \), which convert the scores to probabilities. These probabilities can provide some interpretation on how sure the model is that a certain place should be present in the resulting Petri net. Minimizing \( L \) from Equation [C.1] causes the probabilities to move closer to the intended classes 0 and 1. Inference is done by simply rounding the probability values for each candidate place, i.e. putting the threshold on 0.5:

\[
C_v = \begin{cases} 
1 & \text{if } p_v \geq 0.5, \\
0 & \text{else} \end{cases} \quad \text{(C.2)}
\]

with \( C_v \) the choice on candidate place \( v \).

No direct relation is put between the places, causing the model to be prone to classifying false negatives and perhaps multiple joined/intertwined Petri nets that can produce the same event log. This problem is tackled by the sequential variant of our approach. Furthermore, since the candidate silent transitions are only added between already chosen candidate places, this variant is not able to model silent transitions in the Petri nets it produces.
Appendix D

Information propagation GCN

This chapter goes into more detail on what information is propagated through the graph as described in Chapter 7.1. Specifically, on what information from the event log reaches the candidate place nodes after a certain number of propagation steps.

Let us again use the definitions of a (directed) ball and (directed) sphere with center node $v$ and radius $r$ as introduced in Section 8.2.1:

$$B_{v,r} = \{ u \in V : \delta(v, u) \leq r \lor \delta(u, v) \leq r \}$$  \hspace{1cm} (D.1)

and directed balls $B_{v,r}^1$ and $B_{v,r}^{-1}$:

$$B_{v,r}^1 = \{ u \in V : \delta(u, v) \leq r \}, \quad B_{v,r}^{-1} = \{ u \in V : \delta(u, v) \leq r \}$$  \hspace{1cm} (D.2)

where $\delta(v, u)$ denotes the distance of the shortest path from $u$ to $v$, only using forward edges.

$$\partial B_{v,r} = \{ u \in V : \delta(u, v) = r \lor \delta(v, u) = r \}$$  \hspace{1cm} (D.3)

Directed spheres $\partial B_{v,r}^1$ and $\partial B_{v,r}^{-1}$ are equally defined as the directed balls.

To show what information is aggregated at a place node for a $n$-layer GCN, we dissect the ball $B_{v^{\text{place}},n}$ by looking at what nodes reside in the different spheres in the ball, i.e. $\partial B_{v^{\text{place}},r}^1$ and $\partial B_{v^{\text{place}},r}^{-1}$ for $1 \leq r \leq n$.

Spheres $B_{v^{\text{place}},1}$ and $B_{v^{\text{place}},1}^{-1}$ respectively contain the incoming and outgoing transition nodes:

$$\forall u \in \partial B_{v,1}^{\text{in}} \left[ v^{\text{place}} \in V_{\text{place}} : u \in V_{\text{transition}} \land u \in T_{\text{in}} \right]$$  \hspace{1cm} (D.4)

i.e. the nodes $\partial B_{v,1}^1$ are all incoming transition nodes of place node $v$.

$$\forall u \in \partial B_{v,1}^{-1} \left[ v^{\text{place}} \in V_{\text{place}} : u \in V_{\text{transition}} \land u \in T_{\text{out}} \right]$$  \hspace{1cm} (D.5)

i.e. the nodes $\partial B_{v,1}^{-1}$ are all outgoing transition nodes of place node $v$.

Then, we look at $\partial B_{v^{\text{place}},2}$ which is the smallest radius sphere containing event nodes:

$$\forall u, w \in B_{v,2} \left[ v \in V_{\text{place}} \land u \in V_{\text{transition}} \land w \in V_{\text{event}} : u.\text{label} = w.\text{label} \right]$$  \hspace{1cm} (D.6)
i.e. all event nodes in $B_{v,2}$ are those having the same label as one of the incoming or outgoing transition nodes of place node $v$. Note that $\partial B_{v(\text{place}),2}^{-1}$ does not contain any event nodes, since there are not edges from transition nodes to event nodes.

Spheres with larger radii contain information about the preceding and subsequent events. Starting with $\partial B_{v(\text{place}),3}^{1}$ which contain all event nodes which directly precede the event nodes from $B_{v,2}$:

$$\forall u, w \in V_{\text{event}} \left[ u \in \partial B_{v,2} \land w \in \partial B_{v,3} \land v \in V_{\text{place}} : \langle w.\text{label}, u.\text{label} \rangle \in L \right] \tag{D.7}$$

with $\langle w.\text{label}, u.\text{label} \rangle \in L$ meaning that the subtrace $\langle w.\text{label}, u.\text{label} \rangle$ exists in one of the traces of $L$.

$\partial B_{v(\text{place}),3}^{1}$ contains all event nodes which succeed the event nodes from $B_{v,2}$:

$$\forall u, w \in V_{\text{event}} \left[ u \in \partial B_{v,2} \land w \in \partial B_{v,3} \land v \in V_{\text{place}} : \langle u.\text{label}, w.\text{label} \rangle \in L \right] \tag{D.8}$$

By increasing the radius of the spheres we get the cases for $\partial B_{v(\text{place}),4}^{1}$ and $\partial B_{v(\text{place}),4}^{−1}$ which contain respectively all event nodes two steps before and after the event nodes from $B_{v,2}$. For $\partial B_{v(\text{place}),4}^{1}$ we have:

$$\forall u, w \in V_{\text{event}} \left[ u \in \partial B_{v,2} \land w \in \partial B_{v,4}^{1} \land v \in V_{\text{place}} : \langle w.\text{label}, *, u.\text{label} \rangle \in L \right] \tag{D.9}$$

with the wildcard $*$ event in subtrace $\langle w.\text{label}, *, u.\text{label} \rangle$ that could be any event. For $\partial B_{v(\text{place}),4}^{−1}$ we have:

$$\forall u, w \in V_{\text{event}} \left[ u \in \partial B_{v,2} \land w \in \partial B_{v,4}^{−1} \land v \in V_{\text{place}} : \langle u.\text{label}, *, w.\text{label} \rangle \in L \right] \tag{D.10}$$

So in general with $k \geq 3$ we get for $\partial B_{v(\text{place}),k}^{1}$:

$$\forall u, w \in V_{\text{event}} \left[ u \in \partial B_{v,2} \land w \in \partial B_{v,k}^{1} \land v \in V_{\text{place}} : \langle u.\text{label}, *, k^{-3}, w.\text{label} \rangle \in L \right] \tag{D.11}$$

where $*^{k−3}$ denotes a sequence of $k−3$ wildcard events and for $\partial B_{v(\text{place}),k}^{−1}$:

$$\forall u, w \in V_{\text{event}} \left[ u \in \partial B_{v,2} \land w \in \partial B_{v,k}^{−1} \land v \in V_{\text{place}} : \langle u.\text{label}, *, k^{-3}, w.\text{label} \rangle \in L \right] \tag{D.12}$$

One could see now that increasing the number of layers in the GCN increases the amount of past and future information of every activity occurrence that is aggregated at the candidate place nodes.
Appendix E

Conformance metrics results training phase

Figure E.1: Histogram showing the FScore metric for predictions and ground truths for training and test dataset.
Figure E.2: Histogram showing the fitness metric for predictions and ground truths for training and test dataset.

Figure E.3: Histogram showing the precision metric for predictions and ground truths for training and test dataset.