Process Mining

Kristian Wichmann

January 30, 2018

1 What is process mining?

Process mining is about discovering underlying procedures and causal relations in data. There is a multitude of models and algorithms that can be used for this. A number of these will be outlined here.

Process mining activities can be divided into three different "modes":

- *Play-in* is the term we use for the act of taking data and turning it into a process model.
- In *play-out* we take a process model and examine what kinds of behaviour it allows. This can be done through simulation.
- *Replay* is the most important part of process mining. Here, modelled and observed behaviours are compared. This is useful for (among other things) conformance checking, prediction, and bottleneck analysis.

2 Event logs

An *event log* is a common way to represent the raw data used in process mining. The term refers to a collection of data where each row contains at least the following:

- A *case id* corresponding to a particular unit whose processes we wish to discover.
- An *activity name*, to denote a specific kind of activity.
- A *timestamp*, which allows us to order the events.

The data may contain other columns (and these may be used in various ways as well), but this is the minimum.

2.1 Traces of activities

For each unique case id, we can look at the subset of the event log which matches it. This may then be sorted by the timestamp (if this is not already the case). This will yield a number of strings of activities known as *traces*. Examples of traces could be:

$$\langle abcd \rangle \quad \langle aaacccd \rangle \quad \langle abababab d \rangle \tag{2.1}$$

3 Process models

A mathematical *model* is an abstraction made to imitate som aspect of reality. Hence, a *process model* seeks to mimic a given process.

4 Evaluating process models

If we're building a binary classifier from a training set of labelled data, we can display the results of a model in a *confusion matrix* with four entries:

- True positives (TP): Actual positives predicted as positives.
- True negatives (TN): Actual negatives predicted as negatives.
- False positives (FP): Actual negatives predicted as positives.
- Fale negatives (FN): Actual positives predicted as negatives.

We obviously want to have many truthful predictions, and few falsy ones. There's a number of quantities which can be calculated - like precision and recall - to predict the quality of the model.

But for a process model, things are not so simple. There's a set of real behaviours, i.e. behaviours that are possible in reality, and a set of behaviours that is possible in the model. But the event log is only a subset of the real behaviours; we can't be sure to see everything possible. The situation is illustrated in figure 1. Which means that the event log consists of two regions: False negatives (FN') and true positives (TP').

So why can't we just define quantities corresponding to precision and recall, and so on? Because this is not a supervised learning problem! We have no idea what the set of real behaviours actually consists of, merely a subset - the event log.

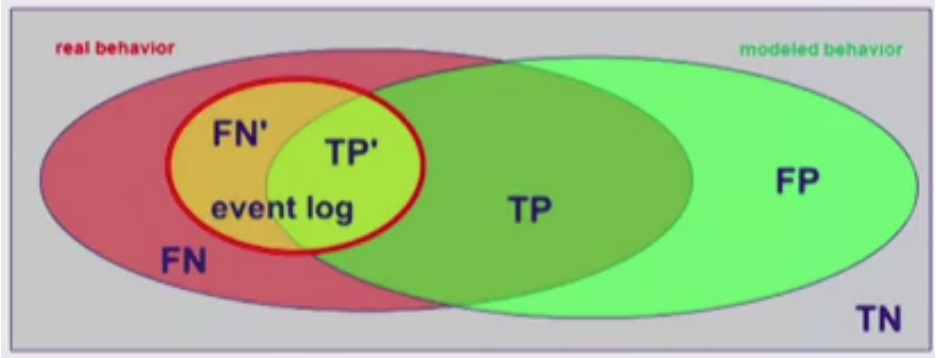

Figure 1: Venn diagram of behaviours.

One quantity that can be defined, though, is the *replay fitness*, the percentage of the event log allowed by the model:

$$\frac{\text{TP'}}{\text{TP'} + \text{FN'}} \tag{4.1}$$

But having this close to 1 is not enough to declare a model good: Fitness is only one part of the story, as we shall see below.

4.1 The four forces

Broadly speaking, there's four criteria for estimating the quality of a process model:

- Fitness: Ability to explain observed behaviours.
- Precision: Correspondence of model behaviours to observations.
- Simplicity: The model is parsimonious enough to be interpretable.
- Generalization: The model should not be overfit to the observed event log.

The first two bullet points might seem similar, so to clarify: High fitness means that all (or most) observed behaviours are possible in the model. High precision mean that all (or most) of the possible behaviours in the model are observed.

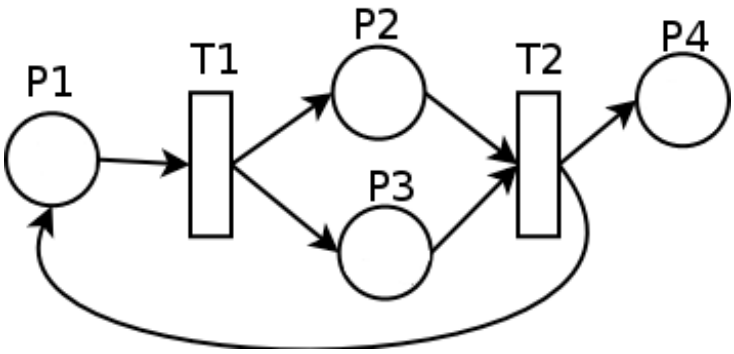

Figure 2: An example of a Petri net graph.

5 Model: Petri nets

5.1 Petri net graphs

A *Petri net graph* is a tuple (P, T, F). Here P is a collection of *places* and T a collection of *transitions*, P and T disjoint. The *flow relations* $F \subseteq (P \times T) \cup (T \times P)$ corresponds to place inputs ($I \subseteq P \times T$) and outputs ($O \subseteq T \times P$) to transitions.

The graphical representation of a petri net graph is to draw places as circles, transitions as thick lines or squares, and flow relations as arrows from places to transitions. As an example, figure 2 is the graphical representation of the following Petri net graph:

- $P = \{P1, P2, P3, P4\}$
- $T = \{T_1, T_2\}$
- $F = I \cup O$, where:

$$I = \{\{P1, T1\}, \{P2, T2\}, \{P3, T2\}\} \tag{5.1}$$
$$O = \{\{T1, P2\}, \{T1, P3\}, \{T2, P1\}, \{T2, P4\}\} \tag{5.2}$$

Sometimes, we may wish to allow more than one arrow between a place-transition/transition-place pair. In this case, F instead becomes a function:

$$F : (P \times T) \cup (T \times P) \to \mathbb{N}_0 \tag{5.3}$$

The value of a pair is then the number of arrows. Again, this could be split into input and output functions:

$$I : P \times T \to \mathbb{N}_0, \quad O : T \times P \to \mathbb{N}_0 \tag{5.4}$$

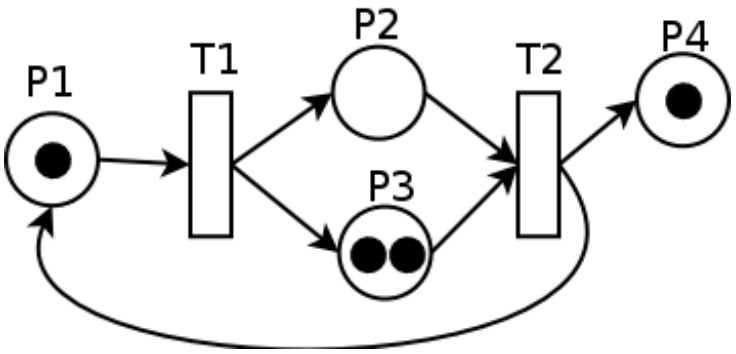

Figure 3: An example of a Petri net.

5.2 Markings and Petri nets

A *marking* M on a Petri net graph, is a function:

$$M : P \to \mathbb{N}_0 \tag{5.5}$$

This is interpreted as the number of *tokens* present at each place in the Petri net graph. For instance, figure 3 shows the following marking on the example Petri net graph from the section above:

$$P1 \mapsto 1, \quad P2 \mapsto 0, \quad P3 \mapsto 2, \quad P4 \mapsto 1 \tag{5.6}$$

A collection of a Petri net graph (P, T, F) and an initial marking M_0 on it is called a *Petri net*.

5.3 Rules of the "game"

From the initial marking, a Petri net can undergo the transitions in T under certain circumstances.

A transition $t \in T$ is *enabled*, if all the places which has an input flow relation pointing to it all has at least one token[1].

An enabled transition may *fire*, which means that the current marking changes, such that each token from the input places are removed, while all the output places gets a token[2].

Transitions always happens in a specific order, so two transitions cannot fire at the same time. This means that we may write a *trajectory* of the Petri

[1]If several arrows are allowed, each place has to hold at least a number of tokens corresponding to the number or arrows.

[2]Again, if several arrows are allowed, several tokens are removed/added according to the multiplicity of arrows.

net as:

$$M_0, M_1, M_2, \ldots, \quad \langle t_1, t_2, \cdots \rangle \tag{5.7}$$

Here, all the M_i's are markings, and between M_i and M_{i+1} transition t_{i+1} fires. Such a trajectory may or may not be finite in length.

The set of all markings that are *reachable* for a Petri net N is denoted $R(N)$. The elements of $R(N)$ forms a directed *reachability graph*, with transitions as edges.

5.4 Properties of Petri nets

A marking where no transitions are enable is called *deadlocked*. A Petri net with a deadlocked marking in its reachability graph is said to have a *potential deadlock*.

A transition in a Petri net that can never fire, i.e. is not an edge anywhere in the reachability graph is called *dead* or L_0-*live* (a designation that will become clear in a while). A transition that is not dead has some degree of being alive, as shown below:

- A transition is L_1-*live* if it is not dead, i.e. if is occurs as an edge somewhere in the reachability graph. This is also called **potentially fireable**.
- A transition is L_2-*live* if for any positive integer k there is a reachable marking in which it occurs at least k times on a trajectory to the marking. (The trajectories can be different for each k).
- A transition is L_3-*live* if there is a trajectory in which the transition occurs infinitely often.
- A transition is L_4-*live* - or simply *live*, if for any reachable marking, the transition is L_1-live. I.e. no matter where in the reachability graph we are, it is always possible to fire the transition some time in the future.

These conditions are progressively stronger, so usually only the highest index is used to refer to a given transition.

A place p in a Petri net is called k-*bounded* if all markings in the reachability graph has as most k tokens on it. A place that is 1-bounded is called *safe*. A place that is k-bounded for any positive value of k is called *bounded*.

A Petri net in which all places are (k-)bounded is called (k-)*bounded*. A Petri net in which all places are safe is called *safe*. A Petri net graph in which all possible initial markings are bounded is called *structurally bounded*.

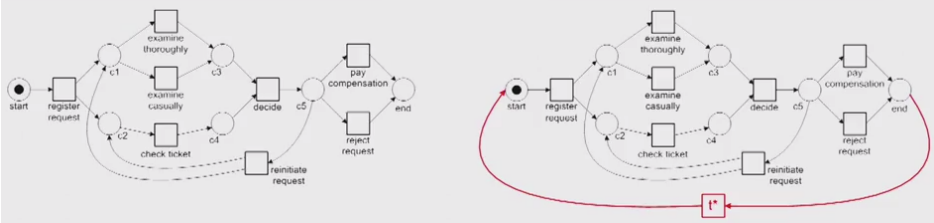

Figure 4: Adding the transition t^*.

5.5 Workflow nets

A *workflow net* (or WF-net for short) is a special kind of Petri net. It is used to model processes with transitions representing events.

A workflow net has a start or *source* place and an ending or *sink* place, so that the source has no inputs, and the sink no outputs. In addition, all other places should be on a trace from the source to the sink. The initial marking is one token at the source.

A workflow net is called *sound* if the following four criteria are satisfied:

1. Safeness: The WF-net is safe.
2. Proper completion: When a token is placed on the sink, all other places should be empty.
3. Option to complete: It is always possible to eventually get a token to the sink.
4. Absence of dead parts: All transitions are L_1-live.

Actually, this is somewhat redundant: Criterium 1 and 3 implies criterum 2:

Theorem 5.1. *For a safe workflow net, if there is option to complete, proper completion is also true.*

Proof. The statement can be written as (3) $\Rightarrow$ (2). This is equivalent to $\neg(2) \Rightarrow \neg(3)$. So assume that proper completion does not hold. I.e. there is a reachable marking with a token at the end place and at least one other place. But because any place in a WF-net is part of a trace from source to sink, we can eventually move that token to the sink. But this contradicts soundness. So completion is not possible. □

It turns out that there's a connection between sound workflow nets and a specific modification to the net: Given a workflow net, the so-called *short-circuited* Petri net is constructed by adding another transition t^* from the sink to the source. Figure 4 shows this process.

Theorem 5.2. *A workflow net is sound if and only if the corresponding short-circuited Petri net is live and bounded.*

Proof. There's two directions to show. (Coming later) □

6 The alpha algorithm

7 Model: Dependency graph

A *dependency graph* is a series of nodes - representing events - in which there are causal arrows between events. So as such, this is simply a directed graph.

Because such a graph does not contain any information about splits and joins, it is usually not used as is. Instead it is the basis for more complicated models.

8 Heuristics mining

9 Model: Causal nets

A *causal net* (or C-net for short) is a tuple (T, I, O) where T is a set of *tasks*. I and O denotes inputs and outputs respectively, in the following way:

$$I : T \to \mathcal{P}(\mathcal{P}(T)), \quad O : T \to \mathcal{P}(\mathcal{P}(T)) \tag{9.1}$$

In other words, to each task is associated a collection of subsets of T, both for input and output. For both inputs and outputs of a give task, we require, that if the empty set is in $I(t)$ (or $O(t)$), then it is the only element.

The interpretation is, that between the subsets in the collection is a logical and (AND) operation. Between elements in the subset is a logical exclusive or (XOR) operation.

For a $t \in T$, the *input and output tasks* for t are defined as follows:

$$\square t = \bigcup I(t), \quad t\square = \bigcup O(t) \tag{9.2}$$

Usually, we will consider C-nets with a source and a sink task, $t_i, t_e \in T$, respectively. We then require that $\square t_i = \varnothing$ and $t_e\square = \varnothing$.

10 Petri net instances

An *instance* of a Petri net (P, T, F, m_0) is a tuple $(PI, TI, FI, \rho, \varrho, m_0')$, so that PI is a set of *instance places*, TI a set of *instance transitions*, FI a set of *instace flow* relations between PI and TI. Furthermore ρ and ϱ are mappings into the original Petri net:

$$\rho : PI \to P, \quad \varrho : TI \to T \tag{10.1}$$

And m_0' is an initial marking of PI. For this structure to be an instance, we furthermore require:

- There can only be elements of instance flow FI where the corresponding elements (through ρ and ϱ) is in F. I.e. we require:

$$\forall p \in PI, t \in TI : (p, t) \in FI \Rightarrow (\rho(p), \varrho(t)) \in F \text{ and} \tag{10.2}$$
$$(t, p) \in FI \Rightarrow (\varrho(t), \rho(p)) \in F \tag{10.3}$$