panikzettel.htwr-aachen.de

Business Process Intelligence Panikzettel

Caspar Zecha, Philipp Schröer, Luca Oeljeklaus

Version 1 — 04.08.2024

Contents

1 Introduction 2

2 Data Mining 2
- 2.1 Decision-tree Learning . . . 2
- 2.2 Association Rule Learning . . . 3
- 2.3 Clustering . . . 4
- 2.4 Evaluating Mining Results . . . 4

3 Process Discovery 5
- 3.1 Petri-Nets . . . 5
- 3.2 Transition systems . . . 5
- 3.3 Workflow Nets . . . 6
- 3.4 The Alpha-Algorithm . . . 6
- 3.5 Limitations of the Alpha-Algorithm . . . 7
- 3.6 Quality Criteria for Process Discovery . . . 8
- 3.7 Business Process Modeling Notation (BPMN) . . . 8

4 Heuristic Mining 8
- 4.1 Dependency Graphs . . . 9
- 4.2 Causal Nets (C-Nets) . . . 9
- 4.3 Learning C-nets . . . 9

5 Region-based Mining 10
- 5.1 Learning Transition Systems . . . 10
- 5.2 State-based Region Mining . . . 10
- 5.3 Language-based Region Mining . . . 11

6 Inductive Mining 12

7 Conformance Checking 13
- 7.1 Causal footprints . . . 13
- 7.2 Token Replay . . . 13
- 7.3 Alignment-based Conformance Checking . . . 13

8 Decision Mining 14

9 Organisational Mining 14
9.1 Resource-activity Matrix 14
9.2 Social Networks 14

10 Process Mining Framework 15
10.1 Refined Process Mining Framework 15
10.2 L* Lifecycle for Process Mining 16
10.3 Mining Spaghetti Processes 16

11 Big Event Data 16

1 Introduction

This Panikzettel is about the lecture Business Process Intelligence by Prof. van der Aalst held in the summer semester 2018.

This Panikzettel is Open Source. We appreciate comments and suggestions at
`https://git.rwth-aachen.de/philipp.schroer/panikzettel`.

2 Data Mining

Our goal is to derive rules from data sets. That is, we have a bunch of datums with multiple variables. In *supervised learning*, we have a special *response variable* that labels each instance (= a result). We want the computer to predict the response variable from a bunch of *predictor variables*. If the response variable is categorical, then we use *classification techniques* (e.g. decision-tree learning). If it is numerical, we use *regression techniques*.

Unsupervised learning assumes unlabelled data, which means that variables are not split into response and predictor variables. Here we can use *clustering* (e.g. k-means clustering) or *pattern discovery*.

2.1 Decision-tree Learning

A *decision tree* is a tree that recursively splits data such that the variation in each subset becomes smaller. We usually use *information gain* to measure this.

The information gain is the difference between two *(information) entropy* values.

p_i is the proportion of examples where the current attribute has value c_i.

Definition: Information Entropy

$$E = -\sum_{i=1}^{k} p_i \cdot \log_2 p_i$$

Algorithm: Decision Tree-Learning

Input: A set of examples with attributes.

Output: A decision tree.

1. Create a new node x associated with all examples.
2. Compute a score $s^{old}(x)$ of x, e.g. based on entropy.
3. If the score $s^{old}(x)$ is too low, just `return` x.
4. Otherwise, find the attribute a with maximum improvement $s_a^{old}(x) - s^{new}(x)$.
5. If the improvement is too low, just `return` x.
6. Split based on a, recursively create sub-nodes.

2.2 Association Rule Learning

Association Rules are of the form $X \Rightarrow Y$. They indicate that usually elements with property Y are chosen when every element with property X is chosen.

To find good rules, we can use the metrics on the right.

Support indicates the importance of the rule. *Confidence* indicates the certainty of the rule. Support and Confidence range from 0 (bad) to 1 (good). *Lift* describes the correlation between X and Y. The number of Lift should be higher than 1.

Definition: Rule Metrics

$$\text{Support}(X \Rightarrow Y) = \frac{N_{X \wedge Y}}{N}$$

$$\text{Confidence}(X \Rightarrow Y) = \frac{N_{X \wedge Y}}{N_X}$$

$$\text{Lift}(X \Rightarrow Y) = \frac{N_{X \wedge Y} \cdot N}{N_X \cdot N_Y}$$

N is the number of instances and N_X is the number of instances covering X.

These equations can be used to filter the association rules apriori and rank them. This is necessary since there may be many rules.

Apriori algorithms filter by the frequency of X and its subsets.

Another example to filter is *FP-Growth* in *RapidMiner*. FP-Growth counts the overall frequency of items and projects them in order of global frequency into a tree. Then, infrequent leaves are repeatedly removed.

2.3 Clustering

Clustering is used to find groups of certain types.

Algorithm: k-Means Clustering

Input: A set of data points, the number k and optionally initial centroids.

Output: k-cluster with centroids.

Place k centroids randomly into the data set if not given.
Iterate until the clusters do not change:

1. Assign the instances to the closest centroid.
2. Recompute the center of the centroids as the average of every instance that belongs to this centroid.

2.4 Evaluating Mining Results

We can create a confusion matrix for decision trees and binary classification to measure the prediction of our data.

		Predicted class		
		+	-	
Actual class	+	**TP**	**FN**	**P=TP+FN**
	-	**FP**	**TN**	**N=FP+TN**
		P′=TP+FP	**N′=FN+TN**	**K=TP+FN+FP+TN**

We have different classes for our relation between predicted and actual class: There are True Positives (TP) and True Negatives (TN) for true predictions. There are also False Positives (FP) and False Negatives (FN). The sum of the actual (predicted) positives are P (P'). The sum of the actual (predicted) negatives are N (N'). The sum of overall data set is K.

The following equations are used to measure:

- *Error:* The rate of wrong predictions where the other class was predicted.
- *Accuracy:* The rate of right predictions where the correct class was predicted.
- *Precision:* The rate of how many of the positive predicted data is actually true positive.
- *Recall:* The rate of how many of the positive data was actually predicted true positive.
- *F1-Score:* It is the harmonic average of the precision and recall and a measure of the accuracy of the prediction.

$$\text{error} = \frac{FP + FN}{K}$$

$$\text{accuracy} = \frac{TP + TN}{K}$$

$$\text{precision} = \frac{TP}{P'} = \frac{TP}{TP + FP}$$

$$\text{recall} = \frac{TP}{P} = \frac{TP}{TP + FN}$$

$$\text{F1-Score} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Models can be *underfitted* when the model is too general and does not exploit the data. On the other side they can be *overfitted* when the model is too specific for the learning set and performs poorly on new sets. We can use *(k-fold) Cross Validation* and split the data set to have multiple training and test sets.

3 Process Discovery

We try to find processes in event data using *process discovery* methods. A case can be seen as a *trace* of events.

The same data can be interpreted in various ways under this model; multiple types of assignments are possible.

Definition: Event log

An *event log* contains a list of *events*, where each event has

• a *case*, • an *activity*, • a *timestamp*.

Lasagna processes are relatively well-structured and good for process mining. *Spaghetti processes* are less structured and not too good for process mining.

3.1 Petri-Nets

To find processes, we need a model for processes. *Petri-nets* are such a model.

We call a specific token allocation a *marking*. A marking is *reachable*, if there is a sequence of transitions that results in that marking.

A transition is called *enabled* if each input place contains a token. Then we can *fire* the transition, consuming and producing tokens.

We also define the *preset* and the *postset* as the set of input or output places respectively.

Definition: Petri-net

A *petri-net* consists of *places* (round nodes), *transitions* (boxed nodes), and *arcs* between them.

Each place can have an arbitrary amount of *tokens*.

There can be arbitrarily many arcs between places and transitions. Using a transition consumes a token from incoming arcs and produces a token along each outgoing arc.

A place in a petri-net is *k-bounded* if there is no reachable marking with more than k reachable tokens in that place. The complete petri-net is k-bounded if the property holds for each place. 1-bounded petri-nets are called *safe*.

A marking is *dead* if no transition is enabled in it. A *deadlock* (as opposed to *deadlock-free*) is possible if there is a reachable dead marking.

Further we call a transition *live* if from any reachable marking it is possible to reach marking that enables the transition.

Data-aware petri-nets use *guards* at the transition nodes to limit the behaviour. Guards do not have to be deterministic.

3.2 Transition systems

A *transition system* is a directed graph with *states* and *transitions*. There is at least one *initial states* and a set of *final states*.

The *reachability graph of a petri net* is a transition system: Nodes are markings encoded as multisets of nodes ($[q_1, q_1, q_3]$ means q_1 has two tokens, q_3 one token). Two nodes are connected if one marking is reachable from the other. Transition systems are more powerful than petri nets, but a corresponding transition system can be exponentially bigger than the petri net.

3.3 Workflow Nets

Workflow nets are petri-nets which have exactly one initial state and sink (final state), and every other state is on a path between these two.

Workflow nets have different properties:

- *Safeness:* Places can not hold multiple tokens at the same time.
- *Proper completion:* If the sink is marked, every other place is empty.
- *Option to complete:* It is always possible to reach a marking that just marks the sink.
- *Absence of dead parts:* For any transition there is a firing sequence to enable it.

Option to complete implies proper completion. When all properties are fulfilled the net is called *sound*.

A workflow net is sound iff the corresponding petri-net is *live* and *bounded*.

3.4 The α-Algorithm

The *α-Algorithm* discovers workflow nets from event logs. The algorithm only uses the case attributes of each event and the temporal ordering between events (but not concrete timestamps). We group transitions of a case into an ordered *trace*.

We define a few kinds of relations between cases.

- *Direct succession*: $x > y$ iff case x is directly followed by y.
- *Causality*: $x \rightarrow y$ iff $x > y$ and $y \not> x$.
- *Parallel*: $x || y$ iff $x > y$ and $y > x$.
- *Choice*: $x \# y$ iff $x \not> y$ and $y \not> x$.

Algorithm: α-Algorithm

Input: Multiset of traces σ.

Output: Workflow net α_L.

---

1. $T_L = \{t \in T \mid \exists_{\sigma \in L} t \in \sigma\}$
2. $T_I = \{t \in T \mid \exists_{\sigma \in L} t = \text{first}(\sigma)\}$
3. $T_O = \{t \in T \mid \exists_{\sigma \in L} t = \text{last}(\sigma)\}$
4. $X_L = \{(A, B) \mid A, B \subseteq T_L \wedge A, B \neq \emptyset \wedge \forall_{a \in A} \forall_{b \in B} a \rightarrow_L b \wedge \forall_{a_1, a_2 \in A} a_1 \#_L a_2 \wedge \forall_{b_1, b_2 \in B} b_1 \#_L b_2\}$
5. $Y_L = \{(A, B) \in X_L \mid \forall_{(A', B') \in X_L} A \subseteq A' \wedge B \subseteq B' \Rightarrow (A, B) = (A', B')\}$
6. $P_L = \{p_{(A,B)} \mid (A, B) \in Y_L\} \cup \{i_L, o_L\}$
7. $F_L = \{(a, p_{(A,B)}) \mid (A, B) \in Y_L \wedge a \in A\} \cup \{(p_{(A,B)}, b) \mid (A, B) \in Y_L \wedge b \in B\} \cup \{(i_L, t) \mid t \in T_I\} \cup \{(t, o_L) \mid t \in T_O\}$
8. $\alpha_L = (P_L, T_L, F_L)$

The algorithm is actually pretty simple. The interesting stuff happens in steps 4 and 5, where maximal sets of directly successive transitions are computed. The rest is basically just bookkeeping.

T_L is simply the set of all transitions available (step 1). T_I and T_O are sets of transitions occuring at the beginning respectively end of traces (steps 2 and 3).

In step 4, we compute X_L, which is a set of pairs of transitions sets. All transitions in A must cause (see above) all transitions in B, and transition pairs from A or B can't directly follow each other (choice). From all these pairs of sets, we only use the maxima with regards to set inclusion (Y_L, step 5).

Now we put the results into a workflow network. For this, step 6 computes P_L as the set of places in the network: Those between transitions of A and B, and an initial place i_L and a final place o_L. In the next step, we compute the arcs (edges) of the network. These go from A-transitions to places, and from those to B-transitions. Furthermore, we connect the initial and final states with first respectively last transitions.

The algorithm returns a workflow network $\alpha_L = (P_L, T_L, F_L)$: Places P_L, transitions T_L and arcs (edges) F_L.

3.5 Limitations of the α-Algorithm

We evaluate the α-algorithm by looking at the output the algorithm produces on some traces of constructed networks.

- **Implicit Places**: The α-algorithm can generate places which can be simply discarded (that is they and arcs to/from them are removed) without changing the net's behaviour. These implicit places are only connected to transitions whose activations are implied by other places.
- **Loops of lengths 1 and 2**: On traces of networks with loops of length 1 (a place is connected to both the input and the output of a transition) and loops of length 2, the algorithm will not rediscover the loops, but will instead connect one transition from the loop with nothing at all.
- **Non-local dependencies**: This is analogous to implicit places: If the original network contains places which are implied non-locally (over at least one transition), the algorithm may not discover those connections. The problem is not specific to the α-algorithm.
- **Representational bias**: Petri nets do not allow transitions with duplicate or invisible labels, so the algorithm cannot discover these.
- **The model is not necessarily sound**: The petri net we get from the algorithm is not always sound, which is bad.
- **Noise is not filtered**: The algorithm does not account for noise in the log, so the model may be pretty complex.
- **Incompleteness**: The log may not account for all real-world behaviour, so the algorithm cannot model this. This is also not specific to the α-algorithm.

3.6 Quality Criteria for Process Discovery

We want to evaluate whether a process model is a correct reflection of the "real process".

We use four criteria to evaluate a process model:

- *Fitness:* The model is able to explain the observed behaviour.
- *Simplicity:* The model uses as few as assumptions as possible.
- *Precision:* The model avoids underfitting.
- *Generalisation:* The model avoids overfitting.

We can use several metrics for all four dimensions.

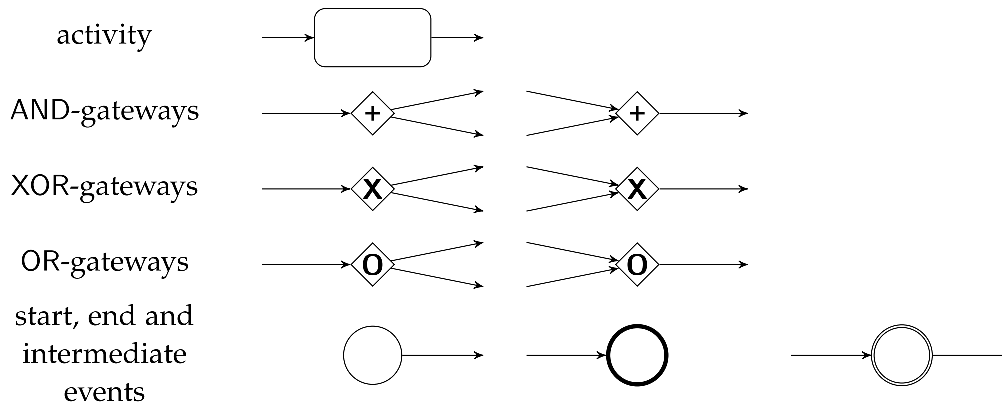
Different modelling languages provide a different "bias". This bias impacts the search space. The α-Algorithm for example will not discover a model where there are k optional activities in between. Often the discovery process is guided by the representational bias.

Process discovery is difficult. Different aspects need to be considered carefully:

- No negative examples: A log does not show what can not happen.
- Search space with complex structure: Concurrency, loops and choices are very complicated.
- Typically the log contains only a fraction of all possible behaviours.
- No clear relation between the size of a model and its behaviour.

3.7 Business Process Modeling Notation (BPMN)

BPMN is another notation for process models.



4 Heuristic Mining

Instead of using all events in a log, we use a *heuristic* which filters events that are used to generate our process model. This is an important technique in practice to prevent infrequent traces ruining our process model into overspecialisation.

The technique here has two phases: Learning dependency graph and then adding bindings to the dependency graph, creating a C-net.

4.1 Dependency Graphs

The *directly follows matrix* is the computation of how many times any activity is followed by another one for a given log. If the matrix on the right were the directly follows matrix of our log, this would mean that a_2 is followed by a_1 three times: $|a_2 >_L a_1| = 3$.

$\|>_L\|$	a_1	a_2	a_3	a_4
a_1	1	0	0	0
a_2	3	4	0	0
a_3	0	0	0	2
a_4	0	0	1	0

The *dependency measure* $|a \Rightarrow_L b|$ is given by:

$$|a \Rightarrow_L b| = \begin{cases} \frac{|a>_L b| - |b>_L a|}{|a>_L b| + |b>_L a| + 1} & \text{if } a \neq b \\ \frac{|a>_L a|}{|a>_L a| + 1} & \text{otherwise} \end{cases}$$

The corresponding *dependency measure matrix* contains the dependency measure for each activity pair.

A *dependency graph* connects activities a and b where their directly follows count and dependency measure meet certain thresholds.

Edges in the graph are labelled as $x(y)$ (that's not multiplication!), where $x = |a >_L b|$ and $y = |a \Rightarrow_L b|$.

Definition: Dependency graph

A dependency graph is a labelled graph (V, E, L) where

- $V = L(\sigma)$ (activities),
- $E = \{ (a, b) \mid a >_L b \}$ (successors),
- $L((a, b)) = |a >_L b|(|a \Rightarrow_L b|)$.

4.2 Causal Nets (C-Nets)

Causal nets (C-nets) are process representations which allow simple modelling of *XOR*, *AND* and *OR* conditions. They are strictly more expressive than petri-nets.

In C-nets, activities have sets of potential input and output bindings. Each set of connected dots shows such a binding. An execution of a C-net (a sequence of bindings) creates and removes *obligations* (pairs of activities), and it is only valid if all created obligations are removed again. Output bindings create obligations and input bindings remove them. This also means that we can traverse the network in any order we like as long as obligations are fulfilled.

4.3 Learning C-nets

This is the second phase heuristic mining: After constructing the dependency graph, we add bindings, by discovering splits and joins. There are two approaches to discover these.

The *heuristic* approach evaluates all possible input-output bindings of each activity a in the directly-follows graph (this is exponential). Then, the log is processed and each occurrence of the binding is counted: One sets a window size w and a binding occurs if bound activities occur in the w events before respectively after a.

The *optimisation* process evaluates all possible input-output bindings of an activity. For a selected binding we evaluate it with replaying and looking at fitness, precision, generalisation and simplicity. The best binding for an activity is then chosen. This approach can be intense and time consuming. It can be optimised with randomness or genetic algorithms.

Genetic Mining is an approach to find models by evolution. First we create an initial set of models from a given log. In the next step, we evaluate the models based on fitness. Then, we keep the best models in the pool. The worst models are sorted out. The rest of the models are *crossed over* and *mutated* to create diversity in the pool. The models in the pool are evaluated again and this loop goes on until it terminates when a satisfactory model is found. This method is possible for different net types.

5 Region-based Mining

Region-based mining is another approach to discover processes. The idea is to find regions that correspond to places in the resulting Petri net. We have two variants: *State-based region mining* and *Language-based region mining*.

5.1 Learning Transition Systems

The state-based region miner has two steps, and it learns a transition system from a log first. This task is trivial. Start with the trivial transition system (one state) and then walk through each case. For each case, extend the transition system and add transitions and places so that the transition system accepts the case.

There are a few variants of this process:

- *Past without abstraction*: Start at the first event in each trace, extending the system left-to-right. You'll end up with a tree-shaped system.
- *Future without abstraction*: As above, but start with the last events, extending right-to-left.
- *Past with multiset abstraction*: Here, states in the transition systems are named by multisets of events. That means two traces $< a, b, c >$ and $< c, b, a >$ lead to the same state $[a, b, c]$, but they take different paths. Thus the abstraction leads to a much smaller, but possibly more imprecise transition network that is not necessarily a tree anymore.

5.2 State-based Region Mining

As the second step, the transition system converted to a Petri net using regions. A region corresponds to a place in the Petri net. We are of course interested in minimal (but not empty) regions to get the most precise Petri net.

Regions are easily transformed into Petri nets: A region R becomes a place p_R, entering activities become input transitions of p_R, and similarly exiting transitions become output transitions of p_R. Finally we add a token to the place corresponding to the initial region. We call the result the *minimal saturated net*.

Definition: Region

A *region* R is a subset of states in the transition system where for each activity a (exactly) one of the following conditions hold for all transitions (s_1, a, s_2):

- All transitions *enter* R: $s_1 \notin R$ and $s_2 \in R$.
- All transitions *exit* R: $s_1 \in R$ and $s_2 \notin R$.
- All transitions *do not cross* R: $s_1, s_2 \in R$ or $s_1, s_2 \notin R$.

5.3 Language-based Region Mining

Language-based region mining does not use a transition system, but uses the log directly (the language). Valid regions are represented by solutions of a linear program.

Intuitively, a region R is valid if a place p_R can be added to a Petri net (restricting its behaviour), without disabling any traces in the log.

Definition: (Possible) Language-based Region

$R = (X, Y, c)$ is a possible region where

- X is the set of input transitions,
- Y is the set of output transitions,
- $c \in \{0, 1\}$ is the initial marking of R.

More formally, this means adding a place p_R with input transitions from X and output transitions to Y and c initial tokens may not prevent execution of any traces in the log. This only happens if p_R is required for a transition activation, but has no tokens. So our equations ensure that, for each activity a at every position in a each trace, p_R has a non-negative token count just after a occurred.

For a trace σ_i, we define before_k as the partial trace of events before event at position k and upto_k as the partial trace that also includes the event at position k.

$$\sigma_i = \underbrace{\overbrace{< e_1 \dots e_{k-1}}^{\text{before}_k}\, e_k}_{\text{upto}_k} \dots e_n >$$

The algorithm on the right produces a set of valid regions, but they need not be minimal.

Algorithm: Language-based Region Mining

Input: An event log L with traces σ_i.

Output: A set of valid regions for L.

1. We build an integer linear program (ILP), with variables $x_a, y_a, c \in \{0, 1\}$ for all activities a.
   - For each trace σ_i:
     - For each position $1 \leq k \leq |\sigma_i|$ in σ_i:

$$\underbrace{c}_{\substack{\text{number of}\\\text{initial tokens}\\\text{in } p_R}} + \underbrace{\sum_{a \in \text{before}_k} x_a}_{\substack{\text{tokens produced}\\\text{for } p_R}} - \underbrace{\sum_{a \in \text{upto}_k} y_a}_{\substack{\text{tokens comsumed}\\\text{from } p_R}} \geq 0$$

2. For each solution (X, Y, c), return valid region $R = (\{ a \mid x_a = 1 \}, \{ a \mid y_a = 1 \}, c)$.

Consider an example: Let

$$L = [\langle a, b, c \rangle, \langle a, c \rangle]$$

be a log. The corresponding ILP has the following equations:

$$\begin{array}{llll} c & & -y_a & \geq 0 \\ c & +x_a & -y_b & \geq 0 \\ c & +x_a + x_b & -y_c & \geq 0 \\ c & & -y_a & \geq 0 \quad \text{(duplicate)} \\ c & +x_a & -y_c & \geq 0 \end{array}$$

One solution is:

$$c = 0, x_a = 1, x_b = 0, y_a = 0, y_b = 1, y_c = 1$$
$$\downarrow$$
$$R = (\{ a \}, \{ b, c \}, 0)$$

6 Inductive Mining

The *Inductive Miner* recursively *cuts* logs into smaller *sublogs*, based on patterns in their directly-follows graph. These patterns are checked in a fixed order (see below). A *cut* splits a log L into pairwise disjoint sets $A_1, \ldots, A_n$.

The Inductive Miner outputs a *process tree*. In its textual representation, an example looks like this: $\circlearrowleft(a, \times(b, c))$. An equivalent regular expression: $(a\ (b + c))^*(a)$.

Definition: Process tree operators

exclusive choice	$\times$
sequential composition	$\rightarrow$
parallel composition	$\bigwedge$
redo loop	$\circlearrowleft$
normal activity	a
silent activity	τ

Definition: 1. Exclusive-choice Cut

$$\times(A_1, \ldots, A_n)$$

Nodes from different partitions may not be connected (directly).

Definition: 2. Sequence Cut

$$\rightarrow(A_1, \ldots, A_n)$$

All nodes from A_i must have a path (not necessarily a direct connection) to all nodes in A_j, where $i < j$.

Definition: 3. Parallel Cut

$$\bigwedge(A_1, \ldots, A_n)$$

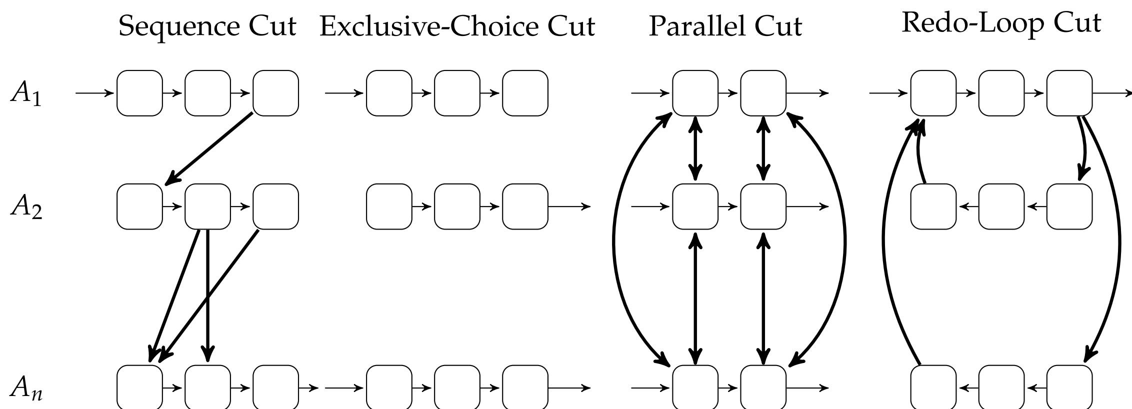
All partitions must have at least one start and end node. All nodes from different partitions are directly connected.

Definition: 4. Redo-loop Cut

$$\circlearrowleft(A_1, \ldots, A_n)$$

The intuition is a redo-loop is like a `while`-loop, except one can abort at each step of the loop and start over.

- A_1 must contain all start and end activities.
- Arrows going out of A_1 must start from end activities.
- Arrows into A_1 must go to a start activity.
- Partitions $A_{i \geq 2}$ do not have any direct connections.
- All nodes in $A_{i \geq 2}$ which have a connection from an end activity must have a connection to **all** end activities.
- All nodes in $A_{i \geq 2}$ which have a connection back to a start activity must be connected to **all** start activities.



The Inductive Miner algorithm has some nice properties: The resulting model is sound, has perfect fitness, and can handle infrequent or incomplete behaviour.

7 Conformance Checking

Conformance checking is all about verifying that a model fits the data. We have three main use cases: Compliance checking (detecting fraud etc.), verifying process discovery results, and checking for conformance to a specification.

7.1 Causal footprints

This method compares the footprint of the event log and the footprint of the model. We can calculate a value for the *footprint-based conformance* by dividing the number of matches between the footprints by the overall number of activity pairs.

$$\frac{|\text{same footprint entries}|}{|\text{activities}|^2}$$

7.2 Token Replay

Token replay measures fitness of a model as a proportion of behaviour in the event log according to the model. Formally, we calculate this value by executing our (WF-net N) model on a trace σ and tracking four values:

- p: Number of produced tokens,
- c: Number of consumed tokens,
- m: Number of missing tokens (consumed, but not actually existing),
- r: Number of remaining tokens (produced, but not consumed).

Note that the environment produces the first and consumes the last token. The fitness is then defined as: $fitness(\sigma, N) = \frac{1}{2}\left(1 - \frac{m}{c}\right) + \frac{1}{2}\left(1 - \frac{r}{p}\right)$. To compute the fitness for an entire log, simply track the respective sums of all four values for each trace in the log.

Token replay imposes an important restriction on the fitness value: It is limited by the notation of the model (WF-nets). Therefore token replay cannot be used to measure conformance of models with silent transitions or models with two transitions with the same label.

7.3 Alignment-based Conformance Checking

Alignment-based conformance checking is more general than token replay and measures, as the name implies, *alignments*.

An *alignment* is a pair of sequences of moves. Moves are either transitions or the "no move" symbol $\gg$. Both move sequences in the alignment should be equal ignoring $\gg$ symbols. The first move sequence corresponds to a trace in the event log, the second one to a run of the model.

a	b	b	b	$\gg$
a	b	$\gg$	b	a

An example alignment with a $\gg$ in both move sequences that aligns *abbb* and *abba*.
It is not unique.

Many metrics are possible to compute the fitness of an alignment. The *standard cost function* simply counts the number of $\gg$ symbols in an alignment. Of course this means that many optimal alignments are possible with this metric.

8 Decision Mining

Decision mining can use data-aware petri-nets to give a more detailed model with conditions (*guards*) at transitions at so-called *decision points*. Decision points are the places in networks with multiple outgoing transitions. Decision trees can be used to find these conditions: Use the next activity as the response variable and, for example, attributes of the previous activities as the predictor variables.

9 Organisational Mining

9.1 Resource-activity Matrix

The *resource-activity matrix* provides information as to which of the involved employees performs which activities how often. To compute it, one looks at each (*resource*, *activity*)-pair, computes how often it comes up in the log and divides it by the number of traces.

$$\frac{instances\ of\ a_i\ performed\ by\ resource\ r_j}{amount\ of\ traces}$$

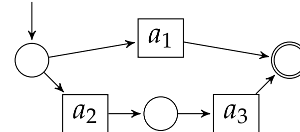
$$\langle a_1^{r_1} \rangle$$
$$\langle a_2^{r_3}, a_3^{r_4} \rangle$$

	a_1	a_2	a_3	a_4
r_1	0.5	0	0	0
r_2	0	0	0	0
r_3	0	0.5	0	0
r_4	0	0	0.5	0

As an example, on the right, we have a log, and below we have its resource activity matrix. What it says is that r_1 performs a_1 on average once every two traces, and that r_3 and r_4 respectively perform a_2 and a_3 on average every two traces.

9.2 Social Networks

Handover of work Matrix The *handover of work matrix* provides information as to which employees transfer work to which other employees. To compute it, you need a process model in the form of a petri net.
On the right we have a petri net for the log shown above. This is important as handover of work only happens between successive activities, and especially not between concurrent activities.



	r_1	r_2	r_3	r_4
r_1	0	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0.5
r_4	0	0	0	0

What we see in this matrix is that once every two cases on average, r_3 hands over work to r_4. This would not be the case for example if we had a model in which a_2 and a_3 were concurrent.

To compute this matrix, use the formula on the right for each ordered pair (r_i, r_j). The corresponding graph can be found on the next page.

$$\frac{times\ r_i\ hands\ over\ work\ to\ r_j}{amount\ of\ traces\ in\ log}$$

Subcontracting matrix The *subcontracting matrix* provides information about which employees outsource work to other employees before taking back control.

$$\langle a_2^{r_3}, a_3^{r_4}, a_4^{r_3} \rangle$$

In the log on the right, you have r_3 outsourcing work to r_4. That is, r_4 executes an activity between two activities from r_3. To compute the matrix, for each ordered pair (r_i, r_j), compute the formula on the right.

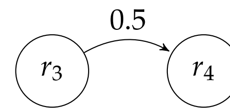
$$\frac{times\ r_i\ outsources\ work\ to\ r_j}{amount\ of\ traces\ in\ log}$$

Working Together Matrix The *working together matrix* indicates how much employees cooperate on cases. For example, on the right, you have r_1, r_3 and r_4 working together. To compute the matrix, compute for each pair (r_i, r_j) the formula on the right.

$$\langle a_2^{r_3}, a_3^{r_4}, a_4^{r_1} \rangle$$

$$\frac{traces\ in\ which\ r_i\ and\ r_j\ both\ work}{amount\ of\ traces\ in\ log}$$

Social Network Graph Any of these matrices can be turned into *social network graphs* by using the resources as nodes and non-zero matrix entries as edges with their value as a tag. (note: you might have a lower threshold to prune outliers)



10 Process Mining Framework

Pre mortem event data is data about cases that are still running, while *post mortem data* is only about dead cases.

10.1 Refined Process Mining Framework

1. *Cartography*: Process models as maps
   a) *Discover*: Find process models.
   b) *Enhancer*: Extend/repair models with event logs.
   c) *Diagnose*: Analyse models without event logs.
2. *Auditing*: Confronting model and reality
   a) *Detect*: Using the models, find violations of current real-world rules of currently running cases (pre mortem).
   b) *Check*: Find weird behavior in the model (deviations, compliance).
   c) *Compare*: On post mortem data, find deviations of current rules.
   d) *Promote*: Create new rules from the models.
3. *Navigation*: Supporting process execution
   a) *Explore*: Explore currently running processes using current data and models.
   b) *Predict*: Predict facts like runtime or success probability for running cases.
   c) *Recommend*: Recommend good choices for current cases.

Operational support, that is supporting current cases, is is done by the Detect, Predict and Recommend actions.

10.2 L^* Lifecycle Model for Process Mining

Stage 0 – *Plan and justify*: Plan the project and justify the project. Is the project data-driven, question-driven or goal-driven?
Stage 1 – *Extract*: Actually get some event data.
Stage 2 – *Create control-flow model*: Find a control-flow model for the available data. Use conformance checking and alignments.
Stage 3 – *Create integrated process model*: Replay the event data on the control-flow model to learn about e.g. time, data, resources. Merge this into a greater model.
Stage 4 – *Operational support*: Use the model for detection, predictions and recommendations (operational support). This only works for structured processes ("Lasagna processes").

10.3 Mining Spaghetti Processes

Because spaghetti processes are less structured than lasagna processes, we need to focus on some subset of the data.

- *Subset of activities*: E.g. most frequent, or some region.
- *Subset of cases*: Choose a homogenous group of cases (by doing clustering, for example), or subclasses.
- *Subset of paths*: E.g. most frequent.

11 Big Event Data

ProTip: Use MongoDB, it's web scale. Because it uses sharding. And has no joins.

We distuingish *data warehouses*, containing structured and cleaned up data, from *data lakes*, which contain unstructured data.

Very large amounts of event data may be too large to be saved, so a *streaming* approach is useful: Using algorithms that do not need the complete event log, but process a stream.

Another way to make event logs smaller is by *decomposition*. This can be done on activities, or on cases, whatever makes most sense in the application context. Conformance checking and Discovery (if it's only counting) can be easily decomposed on a case-level and distributed, but activity-based distribution is usually harder.

This decomposition can be done with *MapReduce*: Worker nodes apply a `map` function to each datum (e.g. case). The output is a collection of key-value pairs. The *shuffle* phase redistributes data so that pairs with the same key end up at the same worker. Then, a `reduce` function is used that processes each group.

One can refer to a decomposition of an event log as a *process cube*, because events are split along some axes into (hyper-)cubes. A comparison between cubes and e.g. their models is called *comparative process mining*.