Weighing the Pros and Cons: Process Discovery with Negative Examples

Tijs Slaats[1(✉)], Søren Debois[2,3], and Christoffer Olling Back[1]

[1] Department of Computer Science, University of Copenhagen, Copenhagen, Denmark
{slaats,back}@di.ku.dk

[2] IT University of Copenhagen, Copenhagen, Denmark
debois@itu.dk

[3] DCR Solutions A/S, Copenhagen, Denmark

Abstract. Contemporary process discovery methods take as inputs only *positive* examples of process executions, and so they are *one-class classification* algorithms. However, we have found *negative* examples to also be available in industry, hence we propose to treat process discovery as a *binary classification* problem. This approach opens the door to many well-established methods and metrics from machine learning, in particular to improve the distinction between what should and should not be allowed by the output model. Concretely, we (1) present a formalisation of process discovery as a binary classification problem; (2) provide cases with negative examples from industry, including real-life logs; (3) propose the Rejection Miner binary classification procedure, applicable to any process notation that has a suitable syntactic composition operator; and (4) apply this miner to the real world logs obtained from our industry partner, showing increased output model quality in terms of accuracy and model size.

Keywords: Process mining · Binary classification · Negative examples · Labelled event logs

1 Introduction

From the perspective of machine learning, process discovery [1] sits uneasily in the gap between unary and binary classification problems [21,31]. Popular contemporary miners, e.g. [5,23], approach process discovery as unary classification: given only positive examples (the input log) they generate a classifier (the output model) which recognizes traces (adhering to the output model) that resemble the training data. However, a process model is really a binary classifier: it classifies traces into those it accepts (desired executions of the process) and those it does not (undesired executions of the process).

Binary classification in machine learning relies on having access to examples of both classes. For process discovery, this means having not only positive examples of desired behaviour to be accepted by the output model, but also negative examples of undesired behaviour that should be rejected.

Negative examples also underpin a substantial part of the mechanics and theory of machine learning, in particular on model evaluation. Output models are evaluated on measures comparing ratios of true and false positives and negatives; however, absent negative examples, it is impossible to apply such measures. Accordingly, in process discovery, we use measures based only on true positive answers, such as *recall*; we are deprived of more fine-grained measures involving true negative or false positive answers such as *accuracy*.

In practical process discovery, negative examples would help distinguish between incidental correlation and actual rules. For instance, suppose that in some log, whenever we see an activity B, that B is preceded by an activity A. Does that mean that we can infer the declarative rule $A \rightarrow\bullet\ B$, that A is required before B may happen? In general, no: making this distinction requires domain knowledge. E.g., if A is "call taxi" and B is "file minutes from weekly status meeting"; by coincidence, we always call a taxi in the morning the day we file minutes, but clearly there is no rule that we must call a taxi before filing minutes. Conversely, if A is "approve payment" and B is "execute payment", very likely it is a rule that B must be preceded by A.

A mining algorithm does not possess domain knowledge, and so must have help to make such distinctions, to decide whether to add a rule $A \rightarrow\bullet\ B$ to its output model. Negative examples potentially help here: If BA is in the set of negative examples, adding the rule $A \rightarrow\bullet\ B$ is justified, as it rejects this trace. Conversely, if a rule rejects no trace from the negative examples, it is not necessary but *discretionary* for the miner to leave out or keep in. In the case of our examples, we would expect to find ample evidence in our negative examples that executing a payment before approving it is bad, whereas we would expect to find little to no evidence that filing minutes before calling a taxi is undesired.

As shown by [28] negative examples *do* exist in practice, some mining algorithms that include negative examples have been proposed, e.g. [22,28], and interestingly recent editions of the process discovery contest[1] have moved towards using labelled test logs (but not training logs) to rank submissions. In this paper we add to these developments with the following contributions:

1. We formalize process discovery as a binary classification problem, and show that not all process notations can express complete solutions to this problem (Sect. 3).
2. We propose the *Rejection Miner*, a notation-agnostic binary mining procedure applicable to *any* process notation with a syntactic composition operator Sect. 4.
3. We describe two cases where negative examples were encountered in industry and provide data sets [34] (Sect. 5).
4. We implement a concrete Rejection Miner and apply it to these data sets, comparing exploratively to contemporary unary miners (Sect. 6). The miner has been integrated in the commercial dcrgraphs.net modelling tool.

[1] https://icpmconference.org/2019/process-discovery-contest/
https://icpmconference.org/2020/process-discovery-contest/.

For the latter experiments, do note that the contemporary unary miners with which we compare do not take into account the negative examples. They must guess from the positive examples which traces to reject, whereas the Rejection Miner has the negative examples to guide it. We find that the Rejection Miner achieves noticeably better accuracy, in particular on out-of-sample tests, and produces models that are orders-of-magnitude smaller than the unary miners. We also note that we chose not to compare to other binary miners, as we did not aim to show the merits of the Rejection Miner in particular, but of binary mining in general. We chose the Rejection Miner as representative for binary mining as it allows us to build DCR Graphs, which were requested by the industry partner. The implementation of the Rejection Miner is available on-line [33].

Related Work. There have been several earlier works framing process mining as a binary classification task. [22] formulates constraints as Horn clauses and uses the ICL learning algorithm to successively find constraints which remove negative examples, stopping when there are no negative examples left. They translate these generated clauses to DECLARE. The Rejection Miner generalises this approach in that (a) it replaces the horn clauses with a generic notion of "model" for notations with composition (or synchronous product of models), and thus applies directly to a plethora of languages such as DECLARE and DCR Graphs, (b) the Rejection Miner leaves the choice of which clauses to prune until after a set of constraints ruling out all negative constraints is found, opening the door to non-greedy minimisation, and most importantly (c) we prove correctness for the Rejection Miner. [28] proposes an approach where traces are represented as points in an n-dimensional space (n being the number of unique event classes of the log), each point representing the multiplicity of the event classes in that trace. Finding a model is then reduced to the problem of finding a convex hull for the points such that positive points are included and negative points excluded. Whereas the work only considers the multiplicity of event classes in negative traces, the Rejection Miner is able to also consider the temporal ordering of individual events, while the former works well for the generation of Petri net models, it is less suitable for declarative notations. In [7,18], the authors artificially generate negative labels, but at the level of individual events rather than traces. The authors also defined process mining oriented metrics based on the resulting true positive/negative labels at the level of events. In [29] the development of binary process discovery algorithms was identified as a key open challenge for the field of declarative process discovery. Our work is also closely related to the work on vacuity detection in declarative process mining [15,25] which considers techniques for selecting the most relevant discovered constraints. However, they only consider logs with positive examples. The use of labelled input data is also well-accepted in the field of predictive process monitoring [16,32]. Finally, our test-driven modelling use case presented in Sect. 5.1 is similar to the scenario-based modelling approach introduced in [17], where (potentially negative) scenarios are modelled as small Petri nets which can then be synthesised into a single larger model. Contrary to this approach we input positive and negative scenarios as traces and learn a declarative model from these.

2 Process Notations and Unary Discovery

We recall the traditional definitions of event logs etc. [1].

Definition 1 (Events, traces, logs). *Assume a countably infinite universe $\mathcal{A}$ of all possible activities. As usual, an* alphabet *$\Sigma \subseteq \mathcal{A}$ is a set of activities, and the Kleene-star $\Sigma^\star$ denotes the countably infinite set of finite strings or sequences over Σ; we call such a string a* trace*. A* log *L is a multiset of occurrences of traces $L = \{t_1^{m_1}, \ldots, t_n^{m_n}\}$ where $m_k > 0$ is the multiplicity of the trace $t_k \in \Sigma$. We write $\mathcal{L}_\Sigma$ for the set of all event logs over alphabet Σ.*

When convenient, we treat an event log L also as simply a set of traces by ignoring multiplicities.

When we discuss unary and binary process discovery in the abstract in later sections, we will be interested in applying discovery to a variety of process notations; and we shall propose a miner which can be instantiated to any notation with a suitable composition operator. To make such statements formally, we need a formal notion of process notation. We use $\mathcal{P}(S)$ for the power set of S.

Definition 2 (Process notation). *A* process notation *for an alphabet Σ comprises a set of* models $\mathbf{M}$ *and an interpretation function $[\![-]\!] : \mathbf{M} \to \mathcal{P}(\Sigma^\star)$ assigning to each individual model m the set of traces $[\![m]\!]$ accepted by that model. For a set $S \subseteq \Sigma^\star$, we write $m \models S$ iff $S \subseteq [\![m]\!]$.*

While a process notation comprises the three components Σ, $\mathbf{M}$, and $[\![-]\!]$, when no confusion is possible we shall allow ourselves to say "consider a process notation $\mathbf{M}$", understanding the remaining two components to be implicit.

Example 3. Here is a toy declarative formalism which allows exactly the condition constraint of DECLARE [2,27] or DCR [12,19] over a countably infinite alphabet $\Sigma = \{A, B, C, \ldots\}$. A "model" is any finite set of pairs $(x, y) \in \Sigma \times \Sigma$, and we interpret each such pair as a condition from x to y. Formally:

$$\begin{aligned} \mathbf{M}_{\mathsf{cond}} &= \{C \subseteq \Sigma \times \Sigma \mid C \text{ finite}\} \\ [\![C]\!] &= \{t \in \Sigma^\star \mid \forall (x,y) \in C.\ \text{each } y \text{ in } t \text{ is preceded by } x\} \end{aligned}$$

For instance, $\{(A, B)\} \in \mathbf{M}_{\mathsf{cond}}$ is a model consisting of a single condition from A to B. In DECLARE or DCR, we would write this model "$A \to\!\bullet\ B$". Just as in DECLARE or DCR, this model admits all traces in which any occurrence of B is preceded by an occurrence of A. That is, this model admits the trace AB, but not B or $BABA$. Formally, we write

$$\begin{array}{rcl} AB \in [\![\{(A,B)\}]\!] & \text{or} & \{(A,B)\} \models \{AB\} \\ \{B, BABA\} \not\subseteq [\![\{(A,B)\}]\!] & \text{or} & \{(A,B)\} \not\models \{B, BABA\} \end{array}$$

Any process modelling formalism with trace semantics is a process notation in the above sense; such formalims include DECLARE, DCR, and Workflow Nets [3] (see also [1]).

We conclude this Section by pinning down process discovery: a procedure which given an event log produces a process model which admits that log. Assume a fixed alphabet Σ, and write $\mathcal{L}_\Sigma$ for the set of all valid event logs over Σ.

Definition 4 (Unary process discovery). *A unary process discovery algorithm γ for a process notation $(\mathbf{M}, [\![-]\!])$ over Σ is a function $\gamma : \mathcal{L}_\Sigma \to \mathbf{M}$. We say that γ has* perfect fitness *iff for all $L \in \mathcal{L}_\Sigma$ we have $\gamma(L) \models L$.*

Anticipating our binary miners, we shall refer to "perfect fitness" also as *positive soundness* of the miner.

3 Process Discovery as Binary Classification

We proceed to consider process discovery a binary classification problem. This approach presumes that we have not only positive examples (the set L in Definition 4), which the output model must accept, but also a set of negative examples, which the output model must reject.

Example 5. Consider again the condition models $\mathbf{M}_{\mathsf{cond}}$ of Example 3. Take as positive set of examples the singleton set $\{AB\}$, and take as negative examples the set $\{BA, B\}$. One model which accepts the positive example and rejects the negative ones is the singleton condition $\{(A, B)\}$. This model admits the positive example AB, because B is preceded by A; and it rejects the negative examples, because in both of the traces B and BA, the initial B is *not* preceded by A.

The negative examples here help solve the relevancy problem that plagues unary miners for declarative formalisms: The positive example AB clearly supports the constraint "A is a condition for B", however, as we saw in the introduction, with only positive examples and without domain knowledge, we cannot know whether this is a coincidence or a hard requirement. In the present example, the negative examples tell us that our model must somehow reject the trace BA, encouraging us to include the condition $A \rightarrow\bullet B$.

Unfortunately, a model accepting a given set P of positive examples and rejecting a given set N of negative ones does not necessarily exists: At the very least, we must have P and N disjoint. To cater to such ambiguous inputs, we allow a binary miner to refuse to produce a model.

Definition 6 (Binary process discovery). *Let $\mathbf{M}$ be a process notation for an alphabet Σ. A* binary-classification process discovery algorithm *("binary miner") is a partial function $\eta : \mathcal{L}_\Sigma \times \mathcal{L}_\Sigma \rightharpoonup \mathbf{M}$, taking sets of positive and negative examples P, N to a model $\eta(P, N)$. We require that $\eta(P, N)$ is defined whenever P, N are disjoint.*

In the rest of this paper, unless otherwise stated, we shall implicitly assume that examples P, N are disjoint. We proceed to generalise the notion of fitness from unary mining.

Definition 7 (Soundness, perfection). *Let $P, N \subseteq \mathcal{L}_\Sigma$ be positive and negative examples. We say that a binary miner η is* positively sound at P, N *iff $\eta(P, N) \models P$. Similarly, we say that η is* negatively sound at P, N *iff $N \cap [\![\eta(P, N)]\!] = \emptyset$. We say that η is* perfect *iff for any disjoint P, N it is defined and both positively and negatively sound.*

In other words: A perfect binary miner produces an output whenever its positive and negative examples are not in direct conflict, and that output admits all positive examples and none of the negative examples provided as input.

Over-and Underfitting of Out-of-Sample Data. A perfect binary miner has no choice in how it treats the elements of P and N: it must admit its positive examples P and reject its negative examples N. It is the remaining *undecided* traces where it has a choice. In the limits, we have the overfitting "maximally rejecting miner", whose output always accepts exactly P and nothing else; and the underfitting "maximally accepting miner", whose output rejects exactly N and nothing else.

However, unlike the unary case, where perfect fitness miners are generally quite easy to come by, **perfect binary miners do not necessarily exist**, and helpful ones may in practice be quite hard to come by.

First, let us try to use a unary miner as a binary one. We do so by simply ignoring the negative examples and applying our unary miner to the positive ones. In this case, it is easy to show that for any unary miner (for any notation) which never returns exactly its input log, we can construct a negative example which will be accepted by the output model of that miner for those positive examples:

Proposition 8. *Let γ be a unary miner for a notation $\mathbf{M}$ over alphabet Σ, and assume that for all L we have $[\![\gamma(L)]\!] \neq L$. Then for all $P \in \mathcal{L}_\Sigma$ there exists a $N \in \mathcal{L}_\Sigma$ s.t. N and P are disjoint, yet N is accepted by the output model $\gamma(L)$.*

So in this sense, **non-trivial unary miners *never* generalise to binary ones.** This is perhaps not entirely surprising. Much less obvious, and a core difference between binary and unary mining, we find that some *notations* cannot express distinctions fine enough to distinguish between positive and negative examples. This is in stark contrast to the unary case, where essentially all notations have a model accepting all traces (the "flower model"); moreover, all commonly accepted notations are able to express any finite language, and so for any input log (finite language), a perfectly fitting model must exist.

However, in the binary case, even though our example notation admits the "flower model", it is still too coarse to admit a perfect binary miner.

Lemma 9. *In $\mathbf{M}_{\mathsf{cond}}$, take positive examples $P = \{ABC\}$, and negative examples $N = \{AB\}$. Then no model $m \in \mathbf{M}_{\mathsf{cond}}$ exists such that $m \models P$ yet $m \not\models N$.*

Proof. Suppose m is a model with $m \not\models \{AB\}$. Then m requires something preceding either A or B, something which is apparently not there. But then that something is missing also from ABC.

In fact, we prove below that **no perfect binary miner can exist in any notation that has only finitely many possible models.** To understand the ramifications of this Theorem, consider again DCR and DECLARE. For DCR or DECLARE models over a fixed finite alphabet (e.g., the set of tasks present in a given log), DCR has infinitely many such models (with distinct semantics), whereas DECLARE has only finitely many. To see this, note that in DCR, because labels and events are not one-one, we can keep adding events that do affect behaviour, while remaining within a finite set of observable tasks. In DECLARE, if there are n activities to choose from, you can populate only finitely many DECLARE templates with those finitely many tasks. Since the arity of DECLARE templates is bounded, and current DECLARE miners are bounded to a finite set of input templates, you are left with only finitely many models.

Note the following consequence for DECLARE: **any binary miner for DECLARE has inputs P, N for which the output a model has either false positives or false negatives**.

Theorem 10. *No perfect binary miner exists for any process notation that has only finitely many possible models* $\mathbf{M}$ *over any non-empty finite alphabet* Σ.

Proof. We construct finite positive and negative examples P and N such that no model accepts P and rejects N. First, we construct N. Let I^+ as the subset of models that accepts infinitely many traces, i.e., $I^+ = \{m \in \mathbf{M} \mid [\![m]\!] \text{ infinite}\}$. Since there are only finitely many models, I^+ is finite, and without loss of generality write it $I^+ = \{m_1, \ldots, m_n\}$. For each m_i, choose a $t_i \in [\![m_i]\!]$, and define $N = \{t_1, \ldots, t_n\}$. Next, we construct P. Let $\complement([\![m]\!])$ be the complement of the traces generated by a model m and I^- the subset of models which reject infinitely many traces, i.e., $I^- = \{m \in \mathbf{M} \mid \complement([\![m]\!]) \text{ infinite}\}$. Again I^- is finite and we write it without loss of generality $I^- = \{p_1, \ldots, p_k\}$. For each of p_j, pick a trace s_j such that $s_j \notin [\![p_j]\!]$ and $s_j \notin N$-this is always possible because $\complement[\![m_j]\!]$ is infinite and N finite. Then define $P = \{s_1, \ldots, s_k\}$. Note that by construction P and N are disjoint. Finally, let $m \in \mathbf{M}$ be a model. At least one of $[\![m]\!]$ and $\complement[\![m]\!]$ must be infinite; we show that in neither case can m be the output of a perfect binary miner applied to P, N. If $[\![m]\!]$ is infinite, then $m \in I^+$, say $m = m_i$, and it follows that $m \models t_i \in N$; hence m fails to reject all negative examples. If on the other hand $\complement([\![m]\!])$ is infinite, then $m \in I^-$, say $m = p_j$ and it follows that $s_j \notin [\![p_j]\!] = [\![m]\!]$; hence m fails to accept the positive example $s_j \in P$.

Alternatively, the above proof can possibly be used to show that there are infinitely many problems P, N with pairwise distinct solutions; the Theorem then follows from the Vapnik-Chervonenkis dimension [4] of the set of interpretations of the finite set of models being necessarily finite, and so unable to shatter this infinite set of distinct solutions.

In unary mining, we may construct a perfect fitness miner like this: As notation, pick simply finite sets of traces, and let the semantics of the notation be that a model (set of traces) T accepts a trace t iff $t \in T$. Then the function $\eta(P) = P$ is a perfect fitness miner. This generalises to any notation strong enough to characterise exactly a given set of T of traces. Obviously, this unary miner has little practical relevance.

It is interesting to note that a similar perfect binary miner exists. Pick as notation pairs of sets of traces T, U, with semantics that T, U accepts t iff $t \in T$ and $t \notin U$. Clearly the function $\eta(P, N) = (P, N)$ is a perfect binary miner, although again, not a particularly helpful one. However, the construction shows that a perfect binary miner exists for any notation strong enough to exactly characterise membership resp. non-membership of finite sets of traces. Notable examples here are Petri-nets and BPMN (through an exclusive choice over the set of positive traces); so it follows that a (trivial) binary miner exists for these notations.

4 Rejection Miners

We proceed to construct a family of binary miners we call "Rejection miners", defined for *any* process notation which has a behaviour-preserving syntactic model composition. Rejection miners are parametric in a "pattern oracle" which selects a set of patterns for consideration; if the patterns selected allow it, the output of the Rejection Miner is perfect. When they do not, the miner does a greedy approximation to optimise for accuracy (i.e., maximising the ratio of true positives and negatives to all inputs).

Definition 11 (Additive process notation). *We say that a process formalism* $\mathbf{M}$ *over* Σ *is* additive *if it comes equipped with a commutative monoid* $(\oplus, \mathbf{1})$ *on* $\mathbf{M}$ *such that*

$$[\![\mathbf{1}]\!] = \Sigma^\star \tag{1}$$

$$[\![m \oplus n]\!] = [\![m]\!] \cap [\![n]\!] \tag{2}$$

We lift the monoid operator to sequences and write $\bigoplus_{i<n} m_i = m_1 \oplus \cdots \oplus m_{n-1}$,

That is, an additive formalism has a flower model $\mathbf{1}$ and a model combination operator $\oplus$. This operator combines two models into a compound one, such that this compound model accepts *exactly* the traces accepted by both of the two original models. DECLARE is an additive formalism: A DECLARE model is a finite set of constraints; the empty such set accepts all traces ($\mathbf{1}$), and the union of two such sets is again such a set, with exactly the desired semantics ($\oplus$). DCR also has a model composition, where the composite model is the union of events, markings, and constraints [11,20]. However, this composition does not preserve semantics in the general case.

In practice, any process notation can be considered additive by forming the synchronous product of models: To check whether a given trace t conforms to a composite model $m \oplus n$, we simply check whether $m \models t$ and $n \models t$. Incidentally, this is a popular implementation mechanism for DECLARE constraints (see, e.g., [10,14]).

The key property of additive process notations used for Rejection Miners is that in such a notation, we can think about models as being the sum of their parts, and the problem of mining can then be reduced to finding suitable such parts. For this approach to be able to generate all models, we would also need to know a subset $\mathbf{S} \subseteq \mathbf{M}$ which generates $\mathbf{M}$ under the model composition operator $-\oplus-$. DECLARE and DCR clearly has such subsets. In keeping with declarative notations and nomenclature, we will refer to such part models as "constraints" in the sequel, however, we emphasise that there is nothing special about them: A constraint m is just another model $m \in \mathbf{M}$.

A rejection miner is parametric in two sub-components: A *pattern oracle*, which given positive and negative examples produces a finite set of (hopefully) relevant constraints; and a *constraint minimiser*, which given a sequence of constraints known to fully reject a set of negative examples selects a subset still fully rejecting those examples.

Definition 12 (Rejection miner components). *Let* $\mathbf{M}$ *be a process notation over an alphabet* Σ*. A* pattern oracle *is a function* $\mathsf{patterns} : \mathcal{L}_\Sigma \times \mathcal{L}_\Sigma \to \mathbf{M}^\star$*. A* minimiser *is a function* $\mathsf{minimise} : \mathbf{M}^\star \times \mathcal{L}_\Sigma \to \mathbf{M}^\star$ *satisfying:*

1. *if* $\sigma \in \mathbf{M}^\star$ *fully rejects* L*, then also* $\mathsf{minimise}(\sigma, L)$ *fully rejects* L*; and*
2. $\mathsf{minimise}(\sigma, L)$ *contains only elements from the input sequence* σ*.*

An example pattern oracle for DECLARE would be the function that produces all possible instantiations of all templates with activities observed in either of its input logs. An example minimiser is the *greedy minimiser* which, starting from the left of the list of constraints, removes those constraints which reject only traces in N that are already rejected by preceding constraints.

Algorithm 13 (Rejection miner). Let $\mathbf{M}$ be an additive notation over Σ, let patterns be a pattern oracle and let minimise be a minimiser.

```
1:  procedure RejectionMiner($P, N$)
2:      $[m_1, \ldots, m_n] \leftarrow \mathsf{patterns}(P, N)$
3:      $\sigma \leftarrow [m_i \mid m_i \models P]$                    ▷ remove $m_j$ where $m_j \not\models P$
4:      $\sigma \leftarrow \bigoplus \mathsf{minimise}(\sigma, N)$
5:      if $[\![\bigoplus \sigma]\!] \cap N \neq \emptyset$ then          ▷ are any negative examples not rejected?
6:          $\delta \leftarrow 1, \sigma_2 \leftarrow []$
7:          while $\delta > 0$ and $|\sigma_2| < |\sigma|$ do
8:              $N' \leftarrow \{n \in N \mid \bigoplus \sigma_2 \models n\}$   ▷ negative examples not yet rejected
9:              $P' \leftarrow \{p \in P \mid \bigoplus \sigma_2 \models n\}$ ▷ positive examples currently accepted
10:             $m, \delta \leftarrow \max_{m_j \in \sigma \setminus \sigma_2}(|\{n \in N' \mid m_j \not\models n\}| - |\{p \in P' \mid m_j \not\models p\}|)$
11:             if $\delta > 0$ then
12:                 $\sigma_2 \leftarrow \sigma_2, m$
13:             end if
14:         end while
15:     end if
16: end procedure
```

A brief explanation: On line 2, the pattern oracle is invoked to produce a finite list $[m_1, \ldots, m_n]$ of relevant constraints. On Line 3, those constraints *not* modelling the positive examples P are filtered out; only the constraints m_i which *do* model P are retained; we assign the resulting list to σ. We then apply the minimiser in Line 4, which by Definition 12 at most removes constraints. On Line 5, we check whether all negative examples are rejected; if so, we have found a perfect model and return it.

Otherwise, we turn to approximation. In the loop in Line 7 to 13, we repeatedly compute the set N' of negative examples not yet rejected and P' of positive examples currently accepted. In Line 10, we iterate over the constraint m_j of the original pattern oracle and compute for each the difference δ_j between how many additional negative examples m_j rejects (wins) and how many already accepted positive examples m_j rejects (losses); we then pick the m_j with the maximum δ_j. If $\delta > 0$, adding the constraint m_j will improve accuracy, and we add it to the set of output constraints. If $\delta \leq 0$, we cannot improve accuracy by including any more constraints, and the loop terminates.

Recall from the previous section the notions of maximally accepting or maximally rejecting perfect binary miners. The minimiser provides a handle for pushing the Rejection Miner towards either of these extremes. Using the identity function as the minimiser will retain all constraints, and so reject the most undecided traces. Conversely, using a minimiser which finds a least subset of constraints rejecting N will remove more constraints, accepting more undecided traces.

The Rejection Miner is not in general a perfect binary miner: The patterns σ provided to it by the $\mathsf{patterns}$ might not, even if all of them were retained, be strong enough to fully reject the set N of negative examples while retaining the positive ones. Moreover, while the Rejection Miner in practice produces decent results, its approximation phase does not find a subset of patterns with optimal accuracy because of its greedy nature.

However, the Rejection Miner will *always* accept all the positive examples; and if the selected patterns σ has any subset σ' which accepts P and rejects N, the Rejection Miner will find such a subset.

Proposition 14. *Let* $\mathsf{patterns}$ *be a pattern oracle, let* $\mathsf{minimise}$ *be a minimiser, and let* P, N *be disjoint sets of positive and negative examples. Then the Rejection Miner for this oracle and minimiser has positive soundness at* P, N. *Moreover if there exists* $\sigma \subseteq \mathsf{patterns}(P, N)$ *such that* σ *accepts* P *and fully rejects* N, *then the Rejection Miner also has negative soundness at* P, N.

Proof (sketch). The former is immediate from line 4; the latter is immediate by the requirements 1 and 2 of Definition 12.

That is: On all inputs where the pattern oracle produces patterns strong enough to make the distinction, the Rejection Miner will exhibit neither false negatives nor positives. Note that this is not in contradiction to Theorem 10: the Rejection Miner is not a perfect miner in general, but if a perfect model exists for a given input and pattern oracle, then it will find such a model.

5 Cases with Negative Examples

The development of the Rejection Miner was not just motivated by academic, but also industrial interest. When pursuing process mining activities in practice we regularly see opportunities to label data and in some cases we have even been asked directly by commercial partners to include counter examples in the construction of models. In this section we discuss the two most developed cases we have encountered, where we both had the opportunity to extract labelled data and publish it in an anonimyzed format. The negative examples in these cases arise from test-driven development and as failures in process engineering.

5.1 DCR Solutions: Test-Driven Modelling

A Danish vendor of adaptive case-management systems, *DCR Solutions*, offers the on-line process modelling portal dcrgraphs.net. In this tool, modellers define *required* (positive) resp. *forbidden* (negative) test cases (traces), expected to be accepted resp. rejected by the model under development. The test cases are also used as input to a process discovery algorithm, which dynamically recommends new constraints to modellers [6]. However, the algorithm used only the positive test-cases, ignoring the negative ones. The extension to consider also those negative ones has been repeatedly requested by the developers of the portal and was implemented as part of this paper. DCR Solutions has kindly allowed us to make the entire data set of test-cases produced in the portal available in an anonymized form [34].

5.2 Dreyer Foundation: Process Engineering

The Danish *Dreyer Foundation* supports budding lawyers and architects, and has previously released an anonymised log of casework [13]. This log documents also testing and early stages of deployment of the system. In a number of cases, process instances that had gone astray were reset to their starting state and partially replayed. The log contains reset markers, and so provides clear negative examples: those prefixes that ended in a reset. We make available here also this partitioning into positive and negative examples [34].

6 Experimental Results

We report on exploratory experiments applying an instantiation of the Rejection Miner to the data sets of Sect. 5, comparing results to current major unary miners.

Data Sets. The DCR Solutions case (Sect. 5.1) comprises 215 logs, each containing at least one negative example, and each produced by users of the portal to codify what a single model should or should not do. The logs contain 7030 events, 1681 unique activities, 589 negative and 705 positive traces. Logs vary enormously in size: the largest log contains 1162 events, 19 activities, 98 negative and 14 positive traces; the smallest log contain but one negative trace of 3 events. Log size distribution is visualised in Fig. 1. The Dreyer case ((Sect. 5.2)) comprises a single log of 10177 events, 33 unique activities, 492 positive and 208 negative traces. The mean trace length is 15 (1–46), and the mean number of activities per trace is 12 (1–24). Both data sets are available on-line [34].

Both data sets were pre-processed to remove any conflicting traces (i.e. that were both marked as positive and negative for the same log). In addition the DCR Solutions data set had a notion of "optional" traces, but what this meant was not well-defined, therefore these were also removed.

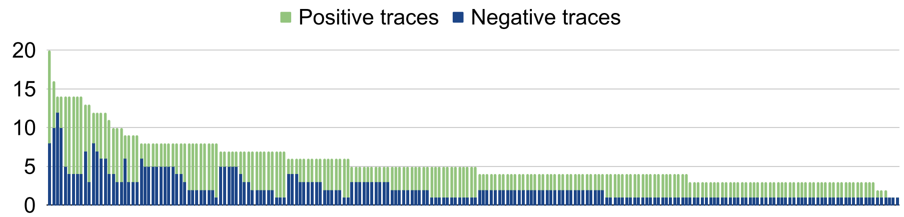

Fig. 1. DCR Solutions data set log size distribution. The largest log of 98 negative and 14 positive traces has been omitted from the diagram.

Metrics. Binary classification mining allows us to rely on traditional machine learning metrics [35] of relative misclassification (true and false positives, TP and FP, and true and false negatives, TN and FN). We use in particular the true positive rate (TPR), true negative rate (TNR), accuracy (ACC), balanced accuracy (BAC), positive predictive value (PPV), and F1-score (F1). We recall their definitions in Table 1. These particular measures demonstrate the difference between what can be measured in the unary and binary settings. In the setting of unary-classification miners, where we do not have negative examples, we can count only TP and FN. In that setting, we can only measure the true positive rate (TPR)-known as "fitness" in the process mining community-but none of the other measures[2]. But in the setting of binary-classification miners, we can measure also how well the output model recognizes negatives (TNR), how reliable a positive classification is (PPV), and generally how accurately both positive and negative traces are classified (ACC, which counts each trace equally and BAC, which balances between positive and negative traces).

Finally, one goal particular to process discovery is to produce output models that are understandable by humans: Output models are not mere devices for classification; they are vehicles for humans to understand the reasons and structure behind that classification. To this end, smaller models are more helpful, so we calculate also the size of the models, dependent on their notation.

[2] The name "F1" is used for a metric of unary miners defined like F1 here, except using the escaping-edges notion of precision [8] *en lieu* of the PPV.

For the pattern-based notations such as DECLARE, we use the number of such patterns; for DCR models the number of relations; and for workflow nets the number of edges and places. Of course sizes for models in different notations are not directly comparable, but they give us an insight in the number of elements that need to be processed by the reader and give a rough indication of relative complexity.

Table 1. Confusion matrix for binary mining

	Log classification		
Model class.	Pos.	Neg.	$\text{ACC} = \frac{\text{TP+TN}}{\text{TP+FP+TN+FN}}$
Pos.	TP	FP	$\text{PPV} = \frac{\text{TP}}{\text{TP+FP}}$
Neg.	FN	TN	$\text{BAC} = \frac{\text{TPR+TNR}}{2}$
	$\text{TPR} = \frac{\text{TP}}{\text{TP+FN}}$	$\text{TNR} = \frac{\text{TN}}{\text{FP+TN}}$	$\text{F1} = 2 \cdot \frac{\text{PPV}\cdot\text{TPR}}{\text{PPV+TPR}}$

Rejection Miner. We provide a JavaScript implementation of the Rejection Miner, available at [33]. We use a pattern oracle which simply instantiates the following list of DECLARE-like patterns at all activities seen in the log: Existence(x), Absence(x), Absence2(x), Absence3(x), Condition(x, y), Response(x, y), NotSuccession(x, y), AlternatePrecedence(x, y), DisjunctiveResponse$(x, (y, z))$, and ConjunctiveResponse$((x, y), z)$. The oracle outputs patterns sorted by how many negative examples they exclude. Ties are broken by sorting the disjunctive and conjunctive responses last, to de-emphasise these relatively more complex patterns.

We emphasise the flexibility of the oracle and minimizer selection: if one wants to include more patterns, one simply extends the oracle; if one wants to have a more restrictive model, or a different prioritization of constraints, one simply replaces the minimizer. One can also produce models that sacrifice TPR for accuracy by creating a minimizer that accepts constraints excluding some positive examples, but also excluding many negative examples.

Other Miners. We compare the Rejection Miner (RM) to flagship miners for three major process notations. For DCR graphs [12,19], we use DisCoveR [26]. DisCoveR is used commercially for model recommendation by DCR solutions. We consider DisCoveR with two settings, the default one (intended to emphasise precision, denoted D), and a "light" version intended to emphasise simplicity (DL). For DECLARE [2,27], we use MINERful [9] and consider three settings, (M1) the most restrictive setting where support = 1.0, confidence = 0.0, and interest factor = 0.0; (M2) a less restrictive setting (likely outputting smaller models) with support = 1.0, confidence = 0.5, and interest factor = 0.25; and (M3) with support = 1.0, confidence = 0.75, and interest factor = 0.5. Finally, for Workflow Nets [3], we use the Inductive Miner [1,23], with a noise threshold of 0.0 (IM) and 0.2 (IMf) respectively.

6.1 Results

We performed both in-sample and out-of-sample testing. For the latter we performed 10-fold validation [30] and calculated our measures as the mean values across 10 randomized attempts. The results are shown in Table 2. For the DCR Solutions data set each value is calculated as the mean over all 215 logs. Because of the limited size of most of the logs, we only tested on in-sample data for this case, however, since the primary goal for the company is to find models that accurately fit the training data, in-sample accuracy is highly relevant.

Table 2. Experiment results

Miner	TPR	TNR	ACC	BAC	PPV	F1	Size
DCR Solutions Data set (Sect. 5.1) In-sample							
Rejection (RM)	1.000	1.000	1.000	1.000	1.000	1.000	1.5
DisCoveR (D)	1.000	0.927	0.976	0.964	0.971	0.983	24.8
- light (DL)	1.000	0.921	0.974	0.948	0.967	0.981	19.6
MINERful (M1)	1.000	0.881	0.958	0.941	0.949	0.970	120.5
- 0.5/0.25 (M2)	0.997	0.841	0.942	0.919	0.930	0.957	77.6
- 0.75/0.5 (M3)	0.961	0.657	0.848	0.809	0.850	0.877	37.8
Inductive (IM)	1.000	0.860	0.946	0.930	0.932	0.960	22.1
- 0.2 noise (IMf)	1.000	0.860	0.946	0.930	0.932	0.960	22.1
Dreyer Foundation Data set (Sect. 5.2) In-sample							
Rejection	1.000	0.928	0.979	0.964	0.970	0.985	6.0
DisCoveR	1.000	0.048	0.717	0.524	0.713	0.832	125.0
- light	1.000	0.048	0.717	0.524	0.713	0.832	71.0
MINERful	1.000	0.067	0.723	0.534	0.717	0.835	1124.0
- 0.5/0.25	1.000	0.0288	0.711	0.514	0.709	0.830	174.0
- 0.75/0.5	1.000	0.005	0.704	0.502	0.704	0.826	102.0
Inductive	1.000	0.019	0.709	0.510	0.707	0.828	160.0
- 0.2 noise	1.000	0.019	0.709	0.510	0.707	0.828	160.0
Dreyer Foundation Data set (Sect. 5.2) Out-of-sample							
Rejection	0.985	0.914	0.964	0.950	0.965	0.975	6.2
DisCoveR	0.962	0.362	0.692	0.662	0.706	0.814	127.6
- light	0.968	0.447	0.697	0.707	0.708	0.817	72.9
MINERful	0.906	0.231	0.659	0.569	0.698	0.787	1128.4
- 0.5/0.25	0.962	0.270	0.685	0.616	0.701	0.810	176.4
- 0.75/0.5	0.970	0.081	0.684	0.525	0.698	0.810	104.9
Inductive	0.981	0.339	0.696	0.660	0.703	0.818	158.9
- 0.2 noise	0.983	0.359	0.698	0.671	0.704	0.819	157.8

DCR Solutions. First, on in-sample test data, the Rejection Miner mines perfectly accurate models on every log. This is a small, but meaningful, improvement over the 0.967 accuracy achieved by DisCoveR light, which is currently used for this task. In practice this means that, given a mapping from the Declarative patterns to DCR Graphs, the Rejection Miner will allow the portal to recommend perfectly accurate models for all test cases that have been defined to-date. Secondly, there is an order-of-magnitude gain in simplicity for the Rejection Miner compared to all other miners: the Rejection Miner requires only 1.5 constraints on average per model. We conjecture that this gain is achieved because knowing what behaviour should be forbidden allows the miner to find precisely the constraints we need, instead of having to propose many constraints to forbid all behaviour that was not explicitly seen in the positive samples. This gain in simplicity also directly benefits the business case, as the industry partners have repeatedly voiced a strong preference for fewer, but more relevant, recommended relations. As a result, the Rejection Miner has already been integrated into the portal by the company.

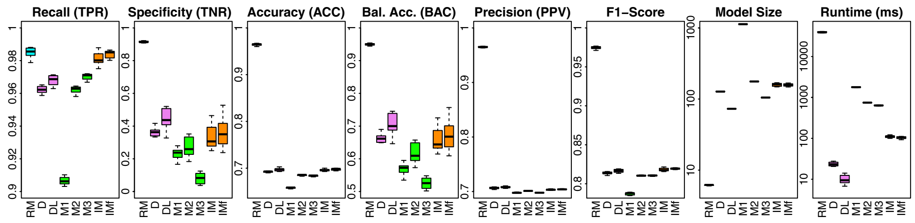

Fig. 2. Boxplots illustrating the distribution of mean performance of various miners across 10 runs of 10-fold cross validation on the Dreyers log.

Dreyers Foundation. The results show that the Rejection Miner once again provides high levels of accuracy while requiring only a small model. Of most interest are the out-of-sample results, shown in more detail in the boxplots of Fig. 2, which indicate that the models found by the Rejection Miner are not only accurate for the training data, but also for unseen test data. In other words, providing the miner with *some* negative examples allows it to accurately predict what *other* negative examples may be seen in the future. In addition there is very little variance in the results of the Rejection Miner, with model size and accuracy scores remaining close to the mean for each randomized run of the 10-fold validation. We also included measures of the run-time performance in Fig. 2, showing that the Rejection Miner is several orders of magnitude slower than the other miners (requiring on average 39.3 s to mine the Dreyers log). We stress however that good run-time performance was never a goal for the current prototype, that there are known methods for improving the run-time performance through a more intelligent initial selection of relevant patterns by the oracle [6,24], and that the results do show that the miner is computationally viable for the experimental data.

7 Conclusion

We propose approaching process discovery as a binary classification problem. We provided a formal account of when binary miners exist; proposed the Rejection Miner; introduced real-world cases of negative examples; and compared the Rejection Miner to contemporary miners for various notations, finding an increase in accuracy and, in particular, output model simplicity.

In future work we will optimize the run-time performance, for example through a more intelligent pattern oracle based on the Declare miner [24] or DisCoveR [6]. We will also pursue additional experiments through labelled real-world logs, such as the PDC datasets and novel use cases from industrial partners.

References

1. Aalst, W.: Process Mining. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49851-4_1
2. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: towards a truly declarative service flow language. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 1–23. Springer, Heidelberg (2006). https://doi.org/10.1007/11841197_1
3. Aalst, W.M.P.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63139-9_48
4. Abu-Mostafa, Y.S., Magdon-Ismail, M., Lin, H.: Learning from Data: A Short Course. AML (2012)
5. Augusto, A., Conforti, R., Dumas, M., La Rosa, M., Polyvyanyy, A.: Split miner: automated discovery of accurate and simple business process models from event logs. Knowl. Inf. Syst. **59**(2), 251–284 (2019)
6. Back, C.O., Slaats, T., Hildebrandt, T.T., Marquard, M.: DisCoveR: accurate & efficient discovery of declarative process models. Presented at the (2021)
7. Broucke, S.V.: Advances in process mining: artificial negative events and other techniques (2014)
8. Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: On the role of fitness, precision, generalization and simplicity in process discovery. In: Meersman, R., et al. (eds.) OTM 2012. LNCS, vol. 7565, pp. 305–322. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33606-5_19
9. Ciccio, C.D., Mecella, M.: A two-step fast algorithm for the automated discovery of declarative workflows. In: CIDM 2013, pp. 135–142, April 2013
10. de Leoni, M., Maggi, F.M., van der Aalst, W.M.P.: An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data. Inf. Sys. **47**, 258–277 (2015). https://doi.org/10.1016/j.is.2013.12.005
11. Debois, S., Hildebrandt, T., Slaats, T.: Hierarchical declarative modelling with refinement and sub-processes. In: Sadiq, S., Soffer, P., Völzer, H. (eds.) BPM 2014. LNCS, vol. 8659, pp. 18–33. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10172-9_2
12. Debois, S., Hildebrandt, T.T., Slaats, T.: Replication, refinement & reachability: complexity in dynamic condition-response graphs. Acta Informatica **55**(6), 489–520 (2018). https://doi.org/10.1007/s00236-017-0303-8

13. Debois, S., Slaats, T.: The analysis of a real life declarative process. In: SSCI/CIDM 2015, pp. 1374–1382. IEEE (2015)
14. Di Ciccio, C., Bernardi, M.L., Cimitile, M., Maggi, F.M.: Generating event logs through the simulation of declare models. In: Barjis, J., Pergl, R., Babkin, E. (eds.) EOMAS 2015. LNBIP, vol. 231, pp. 20–36. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24626-0_2
15. Di Ciccio, C., Maggi, F.M., Montali, M., Mendling, J.: On the relevance of a business constraint to an event log. Inf. Syst. **78**, 144–161 (2018)
16. Di Francescomarino, C., Dumas, M., Maggi, F.M., Teinemaa, I.: Clustering-based predictive process monitoring. IEEE Trans. Serv. Comput. **12**(6), 896–909 (2016)
17. Fahland, D.: Oclets – scenario-based modeling with petri nets. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 223–242. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02424-5_14
18. Goedertier, S., Martens, D., Vanthienen, J., Baesens, B.: Robust process discovery with artificial negative events. J. Mach. Learn. Res. **10**, 1305–1340 (2009)
19. Hildebrandt, T., Mukkamala, R.R.: Declarative event-based workflow as distributed dynamic condition response graphs. In: PLACES 2010. EPTCS, vol. 69, pp. 59–73 (2010). https://doi.org/10.4204/EPTCS.69.5
20. Hildebrandt, T., Mukkamala, R.R., Slaats, T.: Safe distribution of declarative processes. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 237–252. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24690-6_17
21. Khan, S.S., Madden, M.G.: A survey of recent trends in one class classification. In: Coyle, L., Freyne, J. (eds.) AICS 2009. LNCS (LNAI), vol. 6206, pp. 188–197. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17080-5_21
22. Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Inducing declarative logic-based models from labeled traces. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 344–359. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75183-0_25
23. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs - a constructive approach. In: Colom, J.-M., Desel, J. (eds.) PETRI NETS 2013. LNCS, vol. 7927, pp. 311–329. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38697-8_17
24. Maggi, F.M., Bose, R.P.J.C., van der Aalst, W.M.P.: Efficient discovery of understandable declarative process models from event logs. In: Ralyté, J., Franch, X., Brinkkemper, S., Wrycza, S. (eds.) CAiSE 2012. LNCS, vol. 7328, pp. 270–285. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31095-9_18
25. Maggi, F.M., Montali, M., Di Ciccio, C., Mendling, J.: Semantical vacuity detection in declarative process mining. In: La Rosa, M., Loos, P., Pastor, O. (eds.) BPM 2016. LNCS, vol. 9850, pp. 158–175. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45348-4_10
26. Nekrasaite, V., Parli, A.T., Back, C.O., Slaats, T.: Discovering responsibilities with dynamic condition response graphs. In: Giorgini, P., Weber, B. (eds.) CAiSE 2019. LNCS, vol. 11483, pp. 595–610. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21290-2_37
27. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: full support for loosely-structured processes. In: EDOC 2007, p. 287 (2007)
28. Ponce de León, H., Nardelli, L., Carmona, J., vanden Broucke, S.K. : Incorporating negative information to process discovery of complex systems. Inf. Sci. **422**, 480–496 (2018)

29. Slaats, T.: Declarative and hybrid process discovery: recent advances and open challenges. J. Data Semant. **9**(1), 3–20 (2020). https://doi.org/10.1007/s13740-020-00112-9
30. Stone, M.: Cross-validatory choice and assessment of statistical predictions. J. R. Stat. Soc. Ser. B (Methodol.) **36**(2), 111–133 (1974)
31. Tax, D.M.J.: One-class classification: Concept learning in the absence of counter-examples (2002)
32. Tax, N., Teinemaa, I., van Zelst, S.J.: An interdisciplinary comparison of sequence modeling methods for next-element prediction. Softw. Syst. Model. **19**(6), 1345–1365 (2020)
33. Slaats, T., Debois, S.: The Rejection Miner, July 2020. https://github.com/tslaats/RejectionMiner
34. Slaats, T., Debois, S., Back, C.O.: Data Sets: DCR Solutions and Dreyers Foundation logs, July 2020. https://github.com/tslaats/EventLogs
35. Witten, I., Frank, E., Hall, M., Pal, C.: Data Mining: Practical Machine Learning Tools and Techniques, 4th edn. Morgan Kaufmann, Burlington (2016)