Using logical decision trees to discover the cause of process delays from event logs

Diogo R. Ferreira[a,*], Evgeniy Vasilyev[b]

[a]*Instituto Superior Técnico, University of Lisbon, Portugal*
[b]*GREE Inc., Tokyo, Japan*

Abstract

In real-world business processes it is often difficult to explain why some process instances take longer than usual to complete. With process mining techniques, it is possible to do an *a posteriori* analysis of a large number of process instances and detect the occurrence of delays, but discovering the actual cause of such delays is a different problem. For example, it may be the case that when a certain activity is performed or a certain user (or combination of users) participates in the process, the process suffers a delay. In this work, we show that it is possible to retrieve possible causes of delay based on the information recorded in an event log. The approach consists in translating the event log into a logical representation, and then applying decision tree induction to classify process instances according to duration. Besides splitting those instances into several subsets, each path in the tree yields a rule that explains why a given subset has an average duration that is higher or lower than other subsets of instances. The approach is applied in two case studies involving real-world event logs, where it succeeds in discovering meaningful causes of delay, some of which having been pointed out by domain experts.

Keywords: Process mining, Performance analysis, Logical decision trees, Regression trees, Root cause analysis

1. Introduction

Why does a business process become delayed? There may be many possible reasons, but here we focus on causes that can be inferred from run-time data about the past executions of the process. Such data is usually recorded in the form of an event log [1, 2], which can be analyzed from a number of different perspectives, namely the control-flow perspective, the organizational perspective, and the performance perspective:

- In the control-flow perspective, the goal is to extract a process model from the sequence of tasks recorded in the event log (examples of techniques are the α-algorithm [1], the heuristics miner [3], and the genetic miner [4]).
- In the organizational perspective, there are techniques to analyze the interaction between process participants and to extract a social network from the event log [5, 6].
- In the performance perspective, there are ways to calculate performance indicators and to detect bottlenecks based on the timestamp of events [7, 8].

A practical case study that includes these three different perspectives can be found in [9]. Here, we focus mainly on the performance perspective, but we also seek causes of delay that are possibly related to the control-flow and to the organizational perspectives. Since an event log contains information about the tasks that have been performed and the users who have performed them [10], it should be possible to identify causes of delay such as:

– when a certain activity is executed;
– when a certain user participates in the process;
– when a certain user performs a certain activity;
– when a certain activity follows another activity;
– when a specific group of users participate in the process;
– etc.

The causes that can be considered are only limited by the type of data recorded in the event log. If the event log includes information about the data perspective (as in e.g. [11]), it may be possible to find causes based on data properties as well.

As a baseline, we assume that the event log has some minimal information about each event, namely: a case id (i.e. the process instance identifier), a task, a user, and a timestamp. The analysis is based on the total time that each process instance takes to complete. This will be referred to as *duration* and it is measured from the timestamp of the first event to the timestamp of the last event recorded for a given process instance.[1] An instance is said to be delayed if it takes longer than usual to complete, meaning that it takes longer than the average duration of all instances recorded in the event log.

We transform the event log into a logic representation, so that it becomes possible to use inductive reasoning to find the cause of delays. In particular, we capture the information in the event log as a set of first-order logical predicates, and we feed this knowledge base to a decision tree learner which induces a

*Corresponding author
Email addresses: `diogo.ferreira@tecnico.ulisboa.pt` (Diogo R. Ferreira), `vasiliev.evgeni@gmail.com` (Evgeniy Vasilyev)

[1]The concept of duration will be discussed in more detail in Section 2.2.

logical decision tree [12, 13]. This decision tree splits the process instances into a set of classes according to their duration. Each class is defined by a specific rule, which is expressed as a conjunction of predicates (e.g. "the instances where task a is performed and user u_1 participates have an average duration of 252.4 hours"). This rule can be interpreted as the reason why that group of instances has such duration. For those classes of instances which take longer to complete, we take the rule as an indication of a possible reason of delay.

The main challenge in this approach is not in inducing the logical decision tree (an algorithm for that purpose, known as Tilde [12], already exists), but in defining the predicates that will appear in such tree. In essence, those are the predicates that will be used to express the cause of delays. The approach we present here is based on two layers of predicates:

- The first layer comprises a small set of *base predicates* that are used to create a logical representation of events as they have been recorded in the event log.
- The second layer consists in a set of *rules* that define new predicates in terms of the base predicates. These new predicates are meant to represent high-level concepts such as the flow between tasks or the handover of work between users, and they can be automatically inferred from the logical representation of the event log.

In real-world applications, it is possible for the analyst to use custom predicates to focus on domain-specific issues or to analyze specific causes of delay. Therefore, the main goal of this work is two-fold: on one hand, we aim to show how effective the use of logical decision trees can be in discovering the causes of delay and, on the other hand, we intend to illustrate how the analyst may use of custom predicates to analyze specific causes of delay. These goals are better achieved by resorting to concrete examples, so we present two case-study applications involving real-world event logs. The approach itself is also presented by means of a simple example.

The structure of the paper is as follows: Section 2 develops the base predicates and rules that lay the foundation for a logical representation of the event log. Section 3 discusses the induction of logical decision trees and the use of regression to support continuous variables, as is the case with duration. A first example of a logical decision tree which captures a cause of delay is also introduced in Section 3. Then Section 4 presents two case studies using real-world event logs. The second case study illustrates the use of custom predicates. Finally, the paper ends with an overview of some related works.

2. Event log representation

Consider a simple purchase process which can be described as follows:

An employee fills out a requisition form and sends it to a manager for approval. If the requisition is not approved, it is archived and the process ends. Otherwise, the requisition is approved, and the requested product is ordered from a supplier. Then two things will happen in parallel: the warehouse receives the product and updates the stock, and the accounting department takes care of payment to the supplier. When these tasks are complete, the requisition is closed, and the process ends.

A model for this process, using the BPMN language[2], is shown in Figure 1. Each task in this process is assigned to some user (e.g. u_1, u_2, etc.). There may be several users who are able to perform the same task, but only one user will be selected to perform the task for a given process instance. On the other hand, each user may be assigned multiple tasks, either from the same process instance or from different process instances. The result is that each user has its own list of work assignments (a.k.a. "task list", "work list", or "to-do list"), and it is assumed that users carry out their assignments one at a time.

Table 1 shows an excerpt of a sample event log that can be generated from such process. This is similar to the event logs that are typically used for process mining (see e.g. [14], [15]). In the excerpt of Table 1 there are only three process instances, but it is possible to recognize several features of the process. For example, case 1 has a trace in the form $abdefgh$, while case 2 has a trace in the form $abdgefh$, which comes as a result of the parallelism between g and the branch ef. Also, in cases 1 and 2 the requisition is approved, while in case 3 (trace abc) it is archived. Furthermore, it is possible to see different users performing the same task (e.g. u_1 and u_2 performing task a) as well as the same user performing different tasks (e.g. u_2 performing tasks a and h).

Table 1: Excerpt of a generated event log

case id	task	user	timestamp
1	a	u_1	2014-06-09 17:36:47
1	b	u_3	2014-06-11 09:11:13
1	d	u_6	2014-06-12 10:00:12
1	e	u_7	2014-06-12 18:21:32
1	f	u_8	2014-06-13 03:27:41
2	a	u_2	2014-06-14 08:56:09
2	b	u_3	2014-06-14 09:36:02
2	d	u_5	2014-06-15 00:16:40
1	g	u_6	2014-06-16 19:14:14
2	g	u_6	2014-06-19 15:39:15
1	h	u_2	2014-06-19 20:48:16
2	e	u_7	2014-06-20 15:39:45
2	f	u_8	2014-06-22 03:16:16
2	h	u_1	2014-06-23 07:39:24
3	a	u_2	2014-06-25 21:19:46
3	b	u_4	2014-06-29 03:56:14
3	c	u_1	2014-06-30 03:41:22
...	...	...	...

Here we are looking for a different representation of the event log because our aim is to capture the knowledge contained

[2]http://www.omg.org/spec/BPMN/2.0/

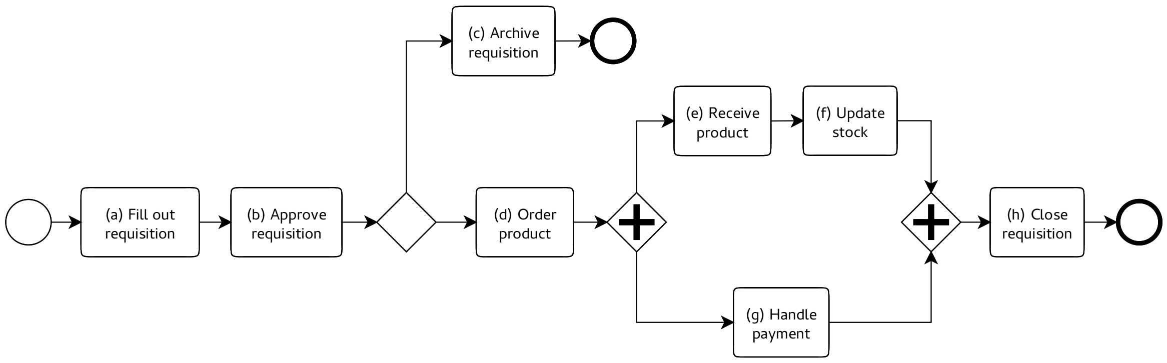


Figure 1: BPMN model of a simple purchase process

therein as a set of logical facts. At the simplest level, these logical facts can be used to express that certain events occurred, and that they occurred in a certain order. This is essentially the same kind of information as presented in Table 1. However, the distinct advantage of using a logical representation for the event log is that simple facts can be used to derive other facts which may represent concepts at a higher level of abstraction.

For example, suppose that we use a logical predicate $event/3$ to capture the first two events in Table 1 as $event(1, a, u_1)$ and $event(1, b, u_3)$.[3] Also, we need to say that the second event follows the first, so we could use a predicate $follows/5$ to express that fact in the following way: $follows(1, a, u_1, b, u_3)$.

Now, suppose that we want to determine whether two users have worked together in the same case. We can express this with the following rule:

$$together(I, X, Y) \text{ :- } event(I, \_, X), event(I, \_, Y).$$

Here, $together/3$ is a new predicate, where I is a variable that stands for a case id, and X and Y are variables that stand for any users.[4] The rule above says that users X and Y work together in case I if there is an event with user X and an event with user Y recorded for that case. The actual tasks in those events do not matter, so we leave them as anonymous variables, i.e. using an underscore, which stands for "any term".

A predicate such as $together/3$ would be useful to study the organizational perspective, e.g. by mining the social network of users according to the "working together" metric, as defined in [5]. Additional predicates can be defined to support other metrics and also other analysis perspectives, namely the control-flow perspective. For example, suppose we want to capture the fact that task b is executed after task a. In other words, we say that there is a "flow of work" [16] from task a to task b. This could be represented with the following rule:

$$flow(I, A, B) \text{ :- } follows(I, A, \_, B, \_).$$

This rule states that there is a flow from task A to task B if an event with task B follows an event with task A. Note that A and B are variables, and therefore they can stand for any task that appears in the event log. In particular, if we have $follows(1, a, u_1, b, u_3)$ (note the use of lowercase letters to represent concrete terms rather than variables) then the fact $flow(1, a, b)$ can be derived.

Above, the predicates $together/3$ and $flow/2$ allow us to derive new facts from a knowledge base containing just facts expressed with $event/3$ and $follows/5$. The interesting point is that whereas all the base facts expressed with $event/3$ and $follows/5$ must be enumerated exhaustively, the new predicates ($together/3$, $flow/2$, and possibly others as well) can be defined as general rules. In this work, the purpose of these general rules is to define new predicates that can be used to explain the cause of delays.

For example, suppose that the process is delayed whenever users u_1 and u_3 work together in the same instance and activity g occurs after f in that instance. Then the rule that explains this cause of delay could be written as:

$$delayed(I) \text{ :- } together(I, u_1, u_3), flow(I, f, g).$$

In this work, our goal is to find such rules through induction over the available facts and predicates. Also, the rule above yields a result of true or false (i.e. either the process instance is delayed or not). However, a delay is a continuous variable and there may be different amounts of delay. Therefore, we need to consider the actual duration of process instances, which can be calculated from the timestamp of events. For this reason, we need to include the timestamps in the base predicates, as explained in the following subsections.

2.1. Base predicates

There are two base predicates that we will use to capture the information in an event log:

- $event/4$ – this predicate captures the fact that a certain task has been completed by a certain user at a certain time, for a certain process instance.
- $follows/7$ – this predicate captures the fact that two events follow each other within the same process instance.

These two predicates are similar to the ones described before, except that now they include the timestamp of each event.

[3]We use the Prolog notation *predicate/arity* to indicate the number of arguments for each predicate.

[4]Here we adopt the Prolog convention that variables begin with an uppercase letter or underscore.

For convenience, the timestamp will be represented as a floating-point number rather than a date and time. The timestamp of the first event in the event log will be set to 0.0 and will be used as reference for the remaining events. Subsequent events will have a timestamp that denotes the time elapsed (in hours) since that first event. Following these conventions, the event log in Table 1 can be represented as follows:

$event(1, a, u_1, 0.000000).$
$event(1, b, u_3, 39.573889).$
$event(1, d, u_6, 64.390278).$
$event(1, e, u_7, 72.745833).$
$event(1, f, u_8, 81.848333).$
$event(1, g, u_6, 169.624167).$
$event(1, h, u_2, 243.191389).$
$follows(1, a, u_1, 0.000000, b, u_3, 39.573889).$
$follows(1, b, u_3, 39.573889, d, u_6, 64.390278).$
$follows(1, d, u_6, 64.390278, e, u_7, 72.745833).$
$follows(1, e, u_7, 72.745833, f, u_8, 81.848333).$
$follows(1, f, u_8, 81.848333, g, u_6, 169.624167).$
$follows(1, g, u_6, 169.624167, h, u_2, 243.191389).$

$event(2, a, u_2, 111.322778).$
$event(2, b, u_3, 111.987500).$
$event(2, d, u_5, 126.664722).$
$event(2, g, u_6, 238.041111).$
$event(2, e, u_7, 262.049444).$
$event(2, f, u_8, 297.658056).$
$event(2, h, u_1, 326.043611).$
$follows(2, a, u_2, 111.322778, b, u_3, 111.987500).$
$follows(2, b, u_3, 111.987500, d, u_5, 126.664722).$
$follows(2, d, u_5, 126.664722, g, u_6, 238.041111).$
$follows(2, g, u_6, 238.041111, e, u_7, 262.049444).$
$follows(2, e, u_7, 262.049444, f, u_8, 297.658056).$
$follows(2, f, u_8, 297.658056, h, u_1, 326.043611).$

$event(3, a, u_2, 387.716389).$
$event(3, b, u_4, 466.324167).$
$event(3, c, u_1, 490.076389).$
$follows(3, a, u_2, 387.716389, b, u_4, 466.324167).$
$follows(3, b, u_4, 466.324167, c, u_1, 490.076389).$

The decision to include the timestamps also in the $follows/7$ predicate has to do with the fact that, depending on the business process, the same task may be performed more than once in the same process instance, so the inclusion of timestamps allows to distinguish between those different occurrences.

2.2. Duration

The total length or duration of a process instance can be calculated as the difference in timestamps between the last event and the first event recorded for that process instance. In some cases, this may not be perfectly accurate, but it will suffice when the goal is just to compare the duration of different process instances. For example, consider the following scenarios:

- If each event in the event log marks the completion of a task, and we define the duration of a process instance to be the difference between the last event and the first event recorded for that instance, then the time it took to perform the first task will not be taken into account. (An extreme example: if a process instance consists of a single task, its duration will be zero.) However, this is regarded as a problem of event logging. If both "start" and "complete" events [17] are recorded for each task, then it is possible to overcome this problem, but for the sake of generality we assume that only "complete" events are available.
- In a real-world event log, some process instances may have been incompletely recorded. This may happen because the instance was already running when event logging was started (in this case, events that occurred earlier than that have been missed) or because the instance was still running when event logging was stopped (in this case, events that occurred after that have been missed). Again, this is a problem of event logging. If it is possible to distinguish between the complete instances and the incomplete ones, then the analysis can focus on the complete ones only. Otherwise, the analysis can be carried out anyway, but the analyst should have in mind that the relatively short duration of some instances may be due to the fact that they are incomplete.

To calculate the duration of each process instance, we introduce the following rule:

$$\begin{aligned} duration(I, D) :- \; & event(I, A, X, T_1), \\ & not(follows(I, \_, \_, \_, A, X, T_1)), \\ & event(I, B, Y, T_2), \\ & not(follows(I, B, Y, T_2, \_, \_, \_)), \\ & D \; is \; T_2 - T_1. \end{aligned}$$

This rule calculates the difference D between two timestamps T_2 and T_1, where T_2 is the timestamp of the last event and T_1 is the timestamp of the first event in process instance I. The first event is the one which does not follow any other, hence the use of $not(follows(I, \_, \_, \_, A, X, T_1))$. The last event is the one which is not followed by any other, which is ensured by $not(follows(I, B, Y, T_2, \_, \_, \_))$. Since it is possible that multiple events have the same timestamp, it is necessary to include all attributes to identify each event.

The application of the above rule to the logical representation of the event log in the previous subsection yields:

$duration(1, 243.191389).$
$duration(2, 214.720833).$
$duration(3, 102.36).$

2.3. Other rules

Besides $duration/2$ introduced above, there are other predicates which can be useful to explain the cause of delays. Earlier, we have introduced some examples, such as $together/3$ and $flow/3$. Here we are going to add some more predicates and also redefine those earlier ones in order to take into account that the base predicates $event/4$ and $followed/7$ now include timestamps.

The first two predicates to be considered are:

- $task/2$ – this predicate captures the fact that a certain task appears in a certain process instance. It is defined as:

$$task(I, A) \coloneq event(I, A, \_, \_).$$

- $user/2$ – this predicate captures the fact that a certain user participates in a process instance. It is defined as:

$$user(I, X) \coloneq event(I, \_, X, \_).$$

With these predicates, it is possible to find causes for delay based on the fact that some particular task is performed or some particular user is involved.

Additionally, we defined the predicate:

$$performs(I, A, X) \coloneq event(I, A, X, \_).$$

to express the fact that a certain user performs a specific task, and this may also be a cause of delay.

Now we introduce some predicates that represent common concepts in the field of process mining.

First, we address the control-flow perspective by introducing the following predicate:

$$flow(I, A, B) \coloneq follows(I, A, \_, \_, B, \_, \_).$$

This expresses the fact that there is a flow from some task A to some task B in instance I. In the field of process mining, such concept is useful to identify constraints in the control flow, such as ordering relations [1] or dependencies [3] between tasks, which are the basis for several process mining algorithms. With this predicate, we can find causes of delay that are related to the flow between tasks.

Second, we address the organizational perspective by introducing a predicate to capture the fact that there is handover of work between users:

$$handover(I, X, Y) \coloneq follows(I, \_, X, \_, \_, Y, \_).$$

and another predicate to capture the fact that users work together on the same process instance:

$$together(I, X, Y) \coloneq event(I, \_, X, \_), event(I, \_, Y, \_).$$

The concepts of handover of work and working together are usually employed to extract the social network of interactions between users in the process [5]. For that purpose, one usually counts the number of times that a user hands over work to another user, or the number of cases in which two users work together. In contrast, here we simply collect those facts in order to check whether some particular interaction between users could be the cause of delay.

There could be additional predicates to capture other concepts. For example, in declarative process mining [18, 19, 20] there are concepts such as co-existence and responded existence constraints between tasks; these concepts can also be captured with the introduction of new predicates. For illustrative purposes, we find it convenient to focus on a concrete set of predicates, and therefore we use the predicates above whenever possible. However, the reader should keep in mind that there is large degree of flexibility in defining these predicates, and that it is possible to use a different set of predicates without fundamentally changing the approach.

3. Logical decision trees

In this work, we use logical decision trees [12, 13] to classify process instances based on their duration, and according to the logical facts that are known about those instances. Logical decision trees use first-order logical predicates to express the branching conditions in the tree.

Figure 2 illustrates the difference between logical decision trees and common decision trees:

- In a common decision tree, at each decision there are as many output branches as possible (discrete) values for a selected data attribute, e.g. x in Figure 2(a). Branching conditions using continuous variables are also possible. The subset of data below a given branch complies with the condition associated with that branch, and with any upper branches that it descends from.

- In a logical decision tree, branching conditions are based on logical predicates and there are only two possible outcomes from each decision. The subset of data that is below a given branch is characterized by the fact that the selected predicate (e.g. P_1 in Figure 2(b)) holds (or does not hold) for all instances in the subtree under that branch.

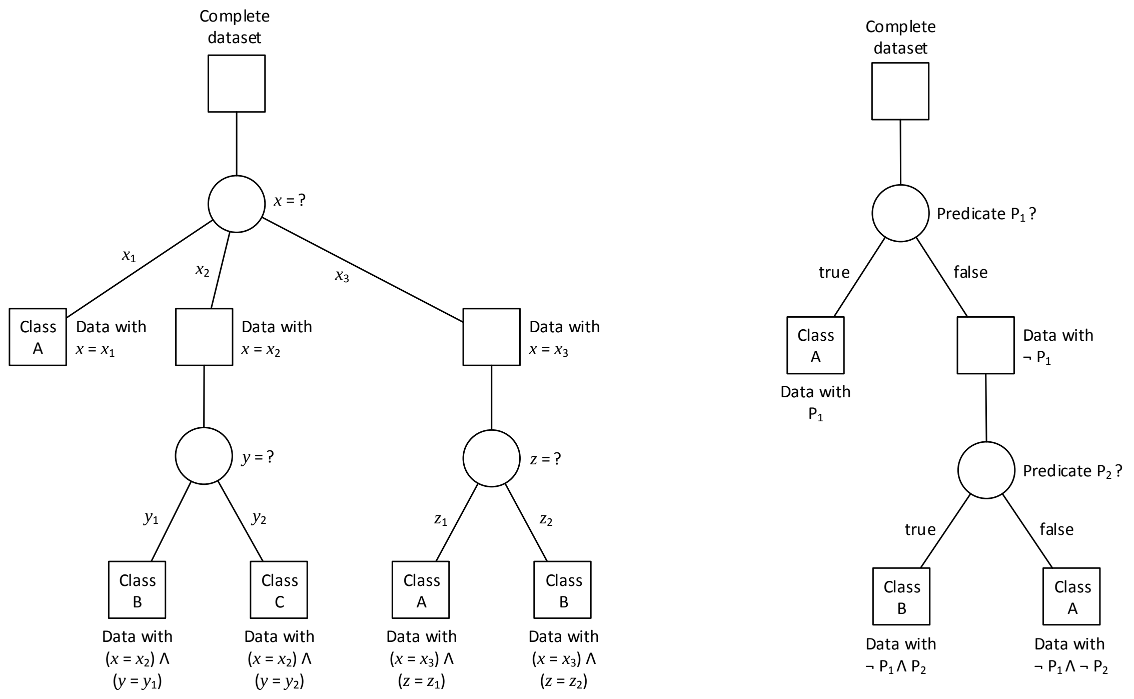
In both types of decision tree, the initial dataset is successively divided into different subsets, until eventually reaching the leaf nodes, where each leaf represents a class of data. Typically, a class is associated with a certain value for the target variable. If the target variable is continuous, as is the case with duration, then each class corresponds to an average value of the target variable for that subset of data.

For any given leaf node, the path in the tree (from the root to that leaf node) is a conjunction of all the conditions that are found along that path (e.g. $x = x_2 \wedge y = y_1$), and this can be interpreted as a rule that explains why that subset of data (in the leaf node) belongs to a certain class. In the context of this work, such rule (e.g. $\neg\mathsf{P}_1 \wedge \mathsf{P}_2$) is interpreted as the reason why a group of instances have a certain duration.

3.1. Tree induction

Typically, decision tree induction is based on a top-down procedure which, at each step, divides the dataset into disjoint subsets according to a *splitting criterion* [21]. This splitting criterion consists in selecting the attribute (or predicate, in the case of logical decision trees) that is the most appropriate to split the data, in the sense that the resulting subsets should be as homogeneous as possible with respect to the target variable. There are several ways to measure the homogeneity of the resulting subsets. For example, ID3 [22] uses information entropy [23], C4.5 [24] uses the normalized information gain [25], and another popular criterion is the Gini index [26].

Besides the splitting criterion, an important feature of an induction algorithm is the *stopping criterion*, which determines when a node should *not* be further divided into subsets. Such node becomes a leaf in the tree. Examples of common stopping criteria are: when every instance in the subset belongs to the



(a) Attribute values

(b) Logical predicates

Figure 2: Decision trees with different types of branching conditions

same class; when there are no more attributes (or predicates) to consider for branching conditions; or when the number of instances in the subset is smaller than a given threshold. Another option is to use a stopping criterion based on an F-test [27], which checks whether there is a statistically significant change in variance after a split.

In summary, an induction algorithm is characterized by having certain splitting and stopping criteria. Apart from these criteria, the structure of any top-down induction algorithm is similar to the one described in Algorithm 1.

Algorithm 1 Tree induction algorithm for a given dataset S, a set of data attributes $\mathbb{A} = \{x, y, z, \ldots\}$ and a target variable t.

1. Create the root node of the tree with the full dataset S.
2. If the current node complies with the *stopping criterion*, mark it as a leaf and label it with the most common (or average) value for t within that (sub)set of data. Otherwise, proceed to step 3.
3. Find the attribute from $\mathbb{A}$ which yields the best split according to the *splitting criterion*. (Some algorithms do not allow the same attribute to be used more than once; if this is the case, then pick the best attribute which has not been used before.) Split the data into subsets according to the possible values for the chosen attribute, and create a child node in the tree for each of these subsets. For each of these child nodes, apply recursively step 2 above.

This procedure can be adapted to logical decision trees by considering the set of available predicates instead of the set of data attributes $\mathbb{A}$. This is the main idea behind Tilde [12], which is equivalent to C4.5 with binary attributes.[5] When the target variable is continuous, Tilde uses a splitting criterion based on the sum of squares, as explained next.

3.2. Regression trees

An advantage of being based on C4.5 is that Tilde is able to handle continuous target variables as well, as is the case with duration. Such type of decision tree is usually referred to as a *regression tree* [28], since the homogeneity of each subset is measured using the sum of squares.

For any given set of data C (which can be the whole dataset or any of its subsets arising from a split), the sum of squares is calculated as $E(C) = \sum_{i \in C}(t_i - \bar{t})^2$ where t_i is the value of the target variable for each element in C, and $\bar{t}$ is the average value of the target variable across all elements in C.

If C is (further) split into subsets $\{C_1, C_2, \ldots\}$ then the sum of squares after such split can be computed as $E'(C) = \sum_k E(C_k)$. The splitting criterion for a regression tree consists in selecting the attribute or predicate which yields the largest decrease in the sum of squares, i.e. the attribute or predicate which maximizes the difference $E(C) - E'(C)$.

3.3. An example

To illustrate the results that can be obtained with Tilde, we generated an event log from the purchase process described in

[5]Tilde was developed by K. U. Leuven and it is part of the ACE data mining system: `https://dtai.cs.kuleuven.be/ACE/`

Section 2. For this example, the event log had 1000 instances, and we modified it by inserting an artificial delay in every instance where users u_1 and u_3 work together and also where task g follows task f.[6]

We converted the event log to the logical representation of Section 2.1, using the base predicates $event/4$ and $follows/7$. The event log was then provided as input to TILDE, together with the predicates defined in Sections 2.2 and 2.3. Figure 3 shows a graphical representation of the output produced by TILDE.

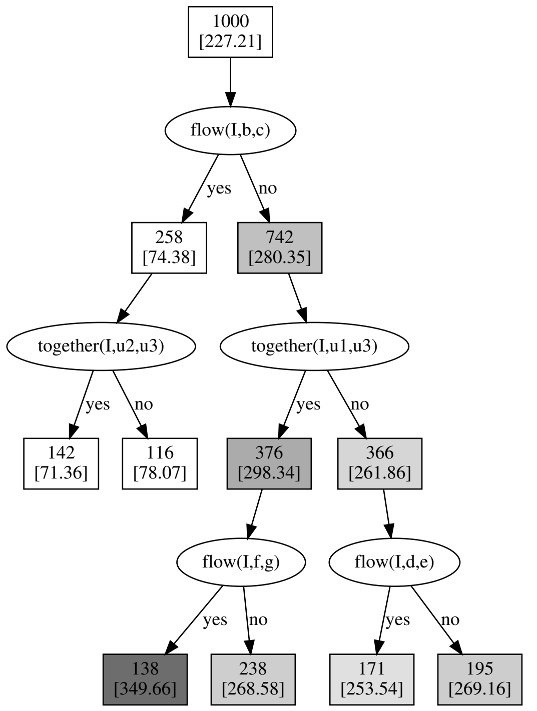


Figure 3: Example of a logical regression tree induced by TILDE

The root node of the tree indicates that there is a total of 1000 instances with an average duration of 227.21 hours. The first decision, based on $flow(I, b, c)$, separates those instances into two subsets depending on whether there is a flow from task b to c or not. A quick look at Figure 1 reveals that the flow from b to c happens when the process ends prematurely; therefore, these instances are noticeably shorter than the rest, as is indicated by their average duration (74.38 hours).

As for the longer instances (with an average duration of 280.35 hours), these are further subdivided into two subsets depending on whether u_1 and u_3 work together or not. Subsequently, the instances where u_1 and u_3 work together are split into those where there is a flow from f to g, and those where such flow does not take place. The leaf node which contains the longest instances of all (with an average duration of 349.66 hours) is highlighted with a dark shade of gray and is characterized by the following rule:

$$\neg\, flow(I,b,c) \;\wedge\; \underbrace{together(I,u_1,u_3) \;\wedge\; flow(I,f,g)}$$

[6]Since the process has an average duration of 216.95 hours and a standard deviation of 98.18 hours, we inserted an amount of delay equal to one standard deviation.

In this rule, the underlined sub-expression is exactly the cause of delay that was inserted in the event log.

The tree contains other splits, namely those based on the predicates $together(I, u_2, u_3)$ and $flow(I, d, e)$, which do not contribute to provide much of a distinction between the resulting subsets, in terms of average duration. The presence of these additional splits is due to the fact that we specified, as a stopping criterion, that no leaf should have less than 100 instances. (It can be seen in Figure 3 that this is indeed the case.) In practice, the analyst can adjust this parameter to control how deep the tree will grow.

4. Case Studies

In this section we describe how the approach has been applied in two case studies involving real-world and publicly available event logs. The two case studies were selected due to their complementary characteristics. The first event log comes from a structured business process and there is no prior information about which factors have an influence on the duration of the process. In contrast, the second event log comes from a case-handling scenario where the process is much less structured, but there are some initial clues about the possible causes of delay.

In both case studies, a few adaptations were required in order to represent the event log in terms of the base predicates defined in Section 2.1. To the furthest extent possible, we tried to use the same predicates so as not to confuse the reader with the introduction of new predicates. However, we recall that, in practice, the analyst may choose to use different or additional predicates to capture the cause of delays. We illustrate this possibility in the second case study.

4.1. A loan application process

The event log used in this case study was originally released in the context of the *BPI Challenge 2012*.[7] The event log was provided by a Dutch financial institution and it concerns a loan application process. Originally, the event log was provided without a specific analysis goal, and participants were free to analyze it using any tools and techniques available. This resulted in a number of submissions[8] focusing on different aspects of the event log, with some of them focusing on the performance and duration of the process, but not on the cause of delays as we do here.

The loan application process that generated the event log can be described as follows:

1. A loan application is submitted through a website, which performs some preliminary checks.
2. An employee gets in contact with the customer by phone in order to gather additional information.
3. If the applicant is eligible, then an offer is sent to the customer by mail.

[7]More information about the BPI Challenge 2012 can be found at: `http://www.win.tue.nl/bpi/2012/challenge`

[8]The submissions are available at: `http://www.win.tue.nl/bpi/2012/challenge`

4. When the offer is returned by the customer, it is assessed.
5. If the offer is incomplete, the missing information is added by contacting the customer again.
6. When the final assessment is done, the loan application is approved and activated.

Table 2 shows the events recorded for the first loan application that appears in the event log. In the second column, it is possible to see that the event log comprises three main kinds of events. The events labeled with the prefix ‘a_’ denote the state of the loan application, the events labeled with ‘o_’ denote the state of the offer that is exchanged with the customer, and the events labeled with ‘w_’ concern the work items (i.e. tasks) that are performed by employees. According to the information released together with the event log, the process is actually a merge of these three sub-processes.

In the third column of Table 2 it can be seen that while the ‘a_’ and ‘o_’ events are one-time occurrences (only *complete* events are recorded), the ‘w_’ events represent activities that are *scheduled*, *started*, and eventually *completed* by an employee. Scheduling an activity means adding it as a new item to the work list of some employee. The activity *starts* when the employee fetches the work item from the list, and it *completes* when the employee either releases the work item, puts it back in the queue, or transfers it to another queue.

In particular, the ‘w_’ activities that appear in Table 2 can be interpreted as follows: *“completeren aanvraag”* means completing the loan application with the additional information provided by the customer; *“nabellen offertes”* consists in calling the customer after the offer is sent; and *“valideren aanvraag”* consists in validating or assessing the loan application.

Given that the event log is a merge of three sub-processes, when representing the event log in terms of the base predicates *event*/4 and *follows*/7, we consider that the *follows* relationship applies separately to the events of each sub-process (e.g. ‘a_’ events follow ‘a_’ events, and ‘w_’ events follow ‘w_’ events, but ‘a_’ events do not follow ‘w_’ events.).

Also, we consider only the *complete* events, since it is important to know that one activity follows another, but it is not so relevant to include the behavior of *schedule*, *start* and *complete* events, as these occur always in the same strict order.

Therefore, the sequence of events in Table 2 can be represented as follows (with timestamps converted to hours since the beginning of the event log):

```
event(173688, a_submitted, 112, 0.0).
event(173688, a_partlysubmitted, 112, 0.0).
event(173688, a_preaccepted, 112, 0.014).
event(173688, a_accepted, 10862, 11.066).
event(173688, o_selected, 10862, 11.106).
event(173688, a_finalized, 10862, 11.106).
event(173688, o_created, 10862, 11.107).
event(173688, o_sent, 10862, 11.107).
event(173688, w_completeren_aanvraag, 10862, 11.108).
event(173688, w_nabellen_offertes, 10862, 11.640).
event(173688, w_nabellen_offertes, 10913, 183.887).
event(173688, o_sent_back, 11049, 226.905).
event(173688, w_nabellen_offertes, 11049, 226.905).
event(173688, o_accepted, 10629, 297.979).
event(173688, a_approved, 10629, 297.979).
event(173688, a_registered, 10629, 297.979).
event(173688, a_activated, 10629, 297.979).
event(173688, w_valideren_aanvraag, 10629, 297.981).
follows(173688, a_submitted, 112, 0.0,
                a_partlysubmitted, 112, 0.0).
follows(173688, a_partlysubmitted, 112, 0.0,
                a_preaccepted, 112, 0.014).
follows(173688, a_preaccepted, 112, 0.014,
                a_accepted, 10862, 11.066).
follows(173688, a_accepted, 10862, 11.066,
                a_finalized, 10862, 11.106).
follows(173688, a_finalized, 10862, 11.106,
                a_approved, 10629, 297.979).
follows(173688, a_approved, 10629, 297.979,
                a_registered, 10629, 297.979).
follows(173688, a_registered, 10629, 297.979,
                a_activated, 10629, 297.979).
follows(173688, o_selected, 10862, 11.106,
                o_created, 10862, 11.107).
follows(173688, o_created, 10862, 11.107,
                o_sent, 10862, 11.107).
follows(173688, o_sent, 10862, 11.107,
                o_sent_back, 11049, 226.905).
follows(173688, o_sent_back, 11049, 226.905,
                o_accepted, 10629, 297.979).
follows(173688, w_completeren_aanvraag, 10862, 11.108,
                w_nabellen_offertes, 10862, 11.640).
follows(173688, w_nabellen_offertes, 10862, 11.640,
                w_nabellen_offertes, 10913, 183.887).
follows(173688, w_nabellen_offertes, 10913, 183.887,
                w_nabellen_offertes, 11049, 226.905).
follows(173688, w_nabellen_offertes, 11049, 226.905,
                w_valideren_aanvraag, 10629, 297.981).
```

In this representation there are three separate sequences of events, because the relationships established by the predicate *follows*/7 for ‘a_’, ‘o_’, and ‘w_’ events are disconnected from each other. Therefore, the duration of a process instance can no longer be computed with the rule defined in Section 2.2; instead, it must be determined by the more simple procedure of subtracting the largest timestamp from the smallest timestamp for a given process instance. For the example above, we have:

```
duration(173688, 297.981).
```

As for the predicates that will be used as branching conditions in the decision tree, these are the same as those defined in Section 2.3, namely: *task*/2, *user*/2, *performs*/3, *flow*/3, *handover*/3 and *together*/3.

Figure 4 shows an excerpt of decision tree produced by Tilde. As before, the darker nodes indicate longer durations, and these appear mainly on the left-hand side of the tree. At the root of the tree there is a split based *task*(*I*, *a_finalized*). This means that instances can be divided into two major groups, depending on whether the loan application reached the *a_finalized* state or not, with the ones that have reached such state having a longer duration, on average, than the ones which have not.

At the second level in the tree, both splits use the predicate *performs*(*I*, *a_cancelled*, 112) to check whether the loan application has been canceled by the system (the 3-digit username 112 is assigned to the system; employees have a 5-digit username). In general, a loan application is automatically canceled by the system if no approval decision has been reached within one month after the application was originally submitted. This

Table 2: The first instance recorded for the loan application process

case id	activity	event type	user	timestamp
173688	a_submitted	complete	112	2011-10-01 00:38:44
173688	a_partlysubmitted	complete	112	2011-10-01 00:38:44
173688	a_preaccepted	complete	112	2011-10-01 00:39:37
173688	w_completeren_aanvraag	schedule	112	2011-10-01 00:39:38
173688	w_completeren_aanvraag	start	112	2011-10-01 11:36:46
173688	a_accepted	complete	10862	2011-10-01 11:42:43
173688	o_selected	complete	10862	2011-10-01 11:45:09
173688	a_finalized	complete	10862	2011-10-01 11:45:09
173688	o_created	complete	10862	2011-10-01 11:45:11
173688	o_sent	complete	10862	2011-10-01 11:45:11
173688	w_nabellen_offertes	schedule	10862	2011-10-01 11:45:11
173688	w_completeren_aanvraag	complete	10862	2011-10-01 11:45:13
173688	w_nabellen_offertes	start	10862	2011-10-01 12:15:41
173688	w_nabellen_offertes	complete	10862	2011-10-01 12:17:08
173688	w_nabellen_offertes	start	10913	2011-10-08 16:26:57
173688	w_nabellen_offertes	complete	10913	2011-10-08 16:32:00
173688	w_nabellen_offertes	start	11049	2011-10-10 11:32:22
173688	o_sent_back	complete	11049	2011-10-10 11:33:03
173688	w_valideren_aanvraag	schedule	11049	2011-10-10 11:33:04
173688	w_nabellen_offertes	complete	11049	2011-10-10 11:33:05
173688	w_valideren_aanvraag	start	10629	2011-10-13 10:05:26
173688	o_accepted	complete	10629	2011-10-13 10:37:29
173688	a_approved	complete	10629	2011-10-13 10:37:29
173688	a_registered	complete	10629	2011-10-13 10:37:29
173688	a_activated	complete	10629	2011-10-13 10:37:29
173688	w_valideren_aanvraag	complete	10629	2011-10-13 10:37:37

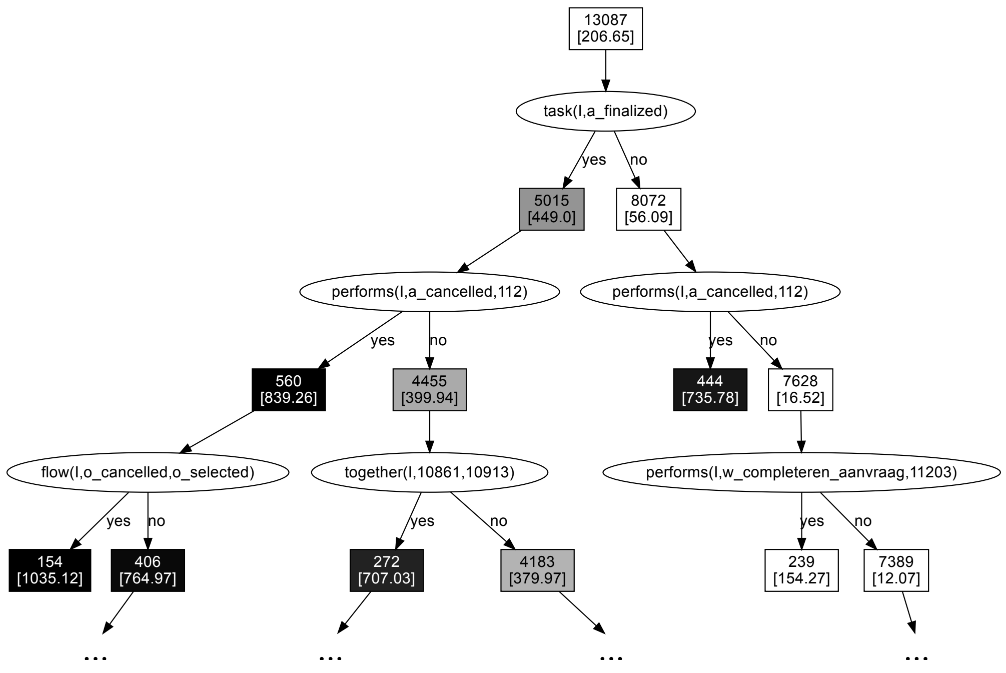


Figure 4: Decision tree induced from the event log of the loan application process

usually happens due to lack of response from the customer: either the application needs to be completed with additional info and the customer does not provide such info, or an offer is sent to the customer but the customer does not reply.

At the third level in the tree we find a split with the predicate $flow(I, o\_cancelled, o\_selected)$ which separates cancelled instances into two further subgroups. As it happens, even after the loan application has been canceled there are cases when an employee selects a timed-out offer and tries to contact the customer again. These instances will run some weeks longer. At the third level in the tree, we also find a split based on the predicate $together(10861, 10913)$, indicating that those instances in which these two employees participate last longer.

In summary, we find that there is at least one external cause and one internal cause of delay in this process. The external cause has to do with the fact that customers often do not provide feedback in a timely fashion (be it additional info, or a reply to an offer). The internal cause is that there is an interaction between users 10861 and 10913 which should be further investigated by the company. Possibly, this interaction can be made more efficient in order to avoid unnecessary delays.

4.2. An incident management process

The event log used in this case study concerns an incident management process at Volvo IT, and it was made available in the context of the BPI Challenge 2013 [29]. Incident management is a well-known business process that deals with unanticipated problems in the operation of a product or service. The goal of incident management is to restore the normal operation of the product or service as soon as possible [30]. The event log was collected from an information system (called VINST) that supports the incident management process in several countries where Volvo IT operates.

An interesting feature of this case study is that some possible causes of delay have been identified beforehand, namely:

- *Ping-pong behavior* – ideally, each incident should be solved quickly and the number of support teams involved should be small. However, it often happens that support teams keep sending incidents back and forth to each other, which is an unwanted behavior. It is suspected that there is a correlation between this ping-pong behavior and the total lifetime of an incident.

- *Wait-user status* – given that there are different ways to measure the total lifetime of an incident, sometimes the support teams will try to find a workaround to stop the clock from ticking. One way to do this is to manually set a *"wait user"* status for the incident. Although there are guidelines for not doing so (unless someone is really waiting for input from a user), it is known that some people are deviating from this guideline.

Our main goal is to check whether (and if so, how) these two factors appear in the decision tree.

As we did in the previous case study, Table 3 shows the sequence of events for a sample instance that appears in the event log. The first column is the incident number, which is equivalent to the case id. The second column (status) denotes the changes in the state of the incident; in a traditional event log, this would denote the task or activity that was performed on the incident. The third column indicates the support team that is trying to resolve the incident.

Here we can see an important feature of this event log: some support teams have a suffix ('2nd' or '3rd') to indicate higher lines of support. The teams without a suffix are assumed to be first-line. The meaning is as follows: the first line of support is the service helpdesk that is the entry point for end-user support; the second line takes care of those incidents which cannot be resolved in the first line; and the third line involves product experts who are called in when the incident cannot be resolved in the second line. Ideally, incidents should be handled at the first line, and they should be sent to the second line or to the third line only if they cannot be solved at the first line (this is referred to as the *push-to-front* principle).

The *ping-pong* behavior refers to the sending of incidents back and forth between support teams. As can be seen in the example of Table 3, an incident may go across multiple support teams and different support lines. In particular, this incident goes from first line (S42) to second line (N52), then to third line (O3), then back to second line (G140, N52, and G137), then back to first line (M25), then again to second line (G142), and finally back to the first line (M25 and S42).

There are many incidents with a similar behavior but involving a different set of teams. Also, some teams are functionally equivalent (e.g. G137, G140, and G142 in Table 3). Therefore, to analyze this behavior we will abstract from the actual names of the support teams and instead we will focus on whether it is a first-line, second-line, or third-line support team. The handover of work back and forth between these support lines will be taken as an indication of ping-pong behavior. The sequence of events in Table 3 can then be represented as follows:

```
event(467153946, in_progress, first_line, 7320.800).
event(467153946, in_progress, first_line, 7320.804).
event(467153946, awaiting_assignment, second_line, 7320.805).
event(467153946, in_progress, second_line, 7320.875).
event(467153946, wait_user, second_line, 7320.943).
event(467153946, awaiting_assignment, third_line, 7392.686).
...
follows(467153946, in_progress, first_line, 7320.800,
                   in_progress, first_line, 7320.804).
follows(467153946, in_progress, first_line, 7320.804,
                   awaiting_assignment, second_line, 7320.805).
follows(467153946, awaiting_assignment, second_line, 7320.805,
                   in_progress, second_line, 7320.875).
follows(467153946, in_progress, second_line, 7320.875,
                   wait_user, second_line, 7320.943).
follows(467153946, wait_user, second_line, 7320.943,
                   awaiting_assignment, third_line, 7392.686).
...
```

This representation allows us to use the same predicates as defined in Section 2.3, and also the predicate $duration/2$ as defined in Section 2.2. It should be noted that here the concept of "task" corresponds to the status of the incident (i.e. $in\_progress$, $awaiting\_assignment$, $wait\_user$, etc.) and the possible "users" are $first\_line$, $second\_line$ and $third\_line$.

Table 3: An instance recorded for the incident management process

incident no.	status	support team	timestamp
467153946	In Progress	S42	2011-01-31 11:12:22
467153946	In Progress	S42	2011-01-31 11:18:44
467153946	Awaiting Assignment	N52 2nd	2011-01-31 11:19:05
467153946	In Progress	N52 2nd	2011-01-31 12:59:46
467153946	Wait - User	N52 2nd	2011-01-31 14:37:55
467153946	Awaiting Assignment	O3 3rd	2011-02-03 08:28:58
467153946	In Progress	O3 3rd	2011-02-07 12:37:33
467153946	Wait - Implementation	O3 3rd	2011-02-07 12:38:25
467153946	In Progress	G140 2nd	2011-03-09 11:08:06
467153946	Wait - Implementation	G140 2nd	2011-03-09 11:27:05
467153946	In Progress	G140 2nd	2011-03-10 11:53:10
467153946	Assigned	G140 2nd	2011-03-10 11:53:22
467153946	In Progress	G140 2nd	2011-03-10 12:17:22
467153946	Wait - User	G140 2nd	2011-03-10 12:21:51
467153946	Wait	G140 2nd	2011-03-10 14:45:27
467153946	Assigned	G140 2nd	2011-03-14 07:56:20
467153946	In Progress	G140 2nd	2011-03-14 07:58:01
467153946	Wait	G140 2nd	2011-03-14 07:58:55
467153946	In Progress	G140 2nd	2011-04-05 16:59:52
467153946	Awaiting Assignment	N52 2nd	2011-04-05 17:02:01
467153946	In Progress	N52 2nd	2011-04-06 09:32:07
467153946	Assigned	N52 2nd	2011-04-06 10:47:50
467153946	In Progress	N52 2nd	2011-04-06 10:48:33
467153946	Wait - User	N52 2nd	2011-04-06 11:08:47
467153946	In Progress	N52 2nd	2011-04-07 12:36:47
467153946	Wait	N52 2nd	2011-04-07 13:06:06
467153946	In Progress	N52 2nd	2011-04-08 13:10:59
467153946	Wait	N52 2nd	2011-04-08 13:17:44
467153946	Awaiting Assignment	G137 2nd	2011-04-08 13:20:00
467153946	In Progress	G137 2nd	2011-04-12 14:16:47
467153946	In Progress	M25	2011-08-08 14:28:38
467153946	Assigned	M25	2011-09-12 10:22:38
467153946	In Progress	M25	2011-11-21 13:52:19
467153946	Wait - Implementation	M25	2011-11-21 13:52:57
467153946	Awaiting Assignment	G142 2nd	2012-01-25 15:50:41
467153946	In Progress	G142 2nd	2012-01-25 15:53:07
467153946	Awaiting Assignment	M25	2012-01-25 15:54:00
467153946	In Progress	S42	2012-05-15 16:56:51
467153946	Resolved	S42	2012-05-15 16:57:39
467153946	Closed	S42	2012-05-23 00:22:25

Figure 5 shows an excerpt of the decision tree that has been obtained from the complete event log. At the root of the tree, it is possible to see that there is a first split between long and very short instances depending on whether the incident has reached the *resolved* state. The incidents where the resolved state does not occur are those which are solved at the helpdesk, while in contact with the customer. Naturally, these are the shortest ones in the event log, since there is no transfer of the incident to another support team.

If we follow those incidents which do have to be transferred to another team in order to be solved (left branch), then the second-level split that appears in the tree is based on the presence of a handover of work from the second line to the first line. This is already an indication of ping-pong behavior because, since incidents start in the first line, a handover of work from second line to first line means that the incident is being sent *back* to the first line.

The ping-pong behavior is more apparent when we consider the third-level split based on $performs(I, closed, first\_line)$, where we see that there are some incidents which, despite being sent back from second line to first line, are not closed in the first line. This means that these incidents will be sent yet again to the second line or to the third line to be closed.

In the other subtree we see the effect of the wait-user status. Basically, those incidents which have a wait-user status last longer (on average, 427.5 hours) than those which do not enter such state (250.96 hours). However, the effects of the wait-user status do not seem to have as much impact in the duration of the process as those of ping-pong behavior.

4.3. Using a custom predicate

To further investigate ping-pong behavior as a cause of delay in the previous case study, it would be useful to have a special-purpose predicate to count the number of times an incident is handed over between different support lines.

For this purpose, we redefine the $handover/3$ predicate:

$$\begin{aligned} handover(I, X, Y) :- \; & follows(I, \_, X, \_, \_, Y, \_), \\ & not(X = Y). \end{aligned}$$

where $not(X = Y)$ is the condition that ensures that the support lines are indeed different.

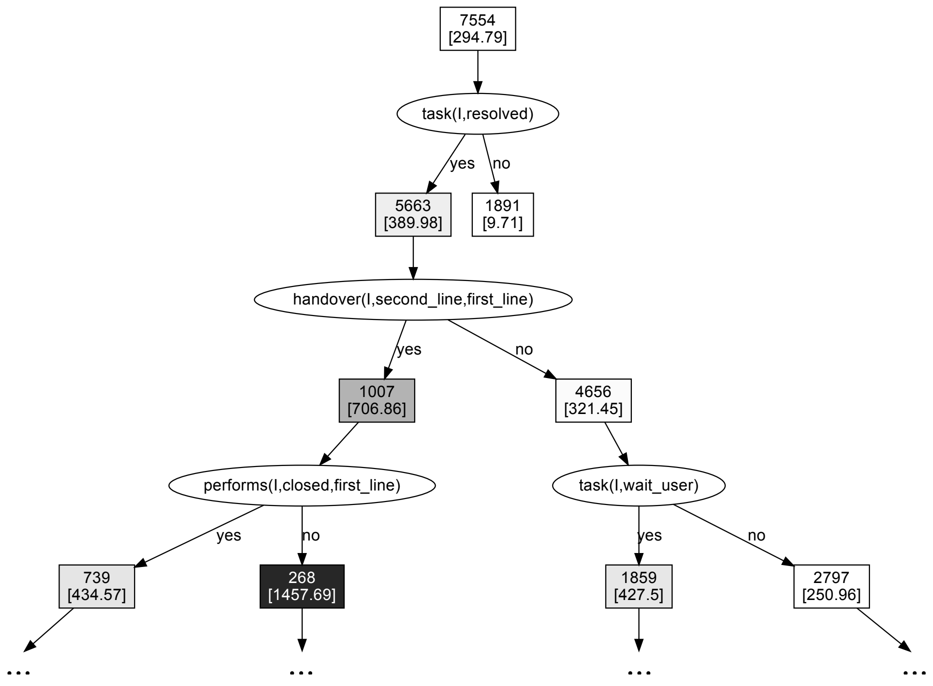


Figure 5: Decision tree induced from the event log of the incident management process

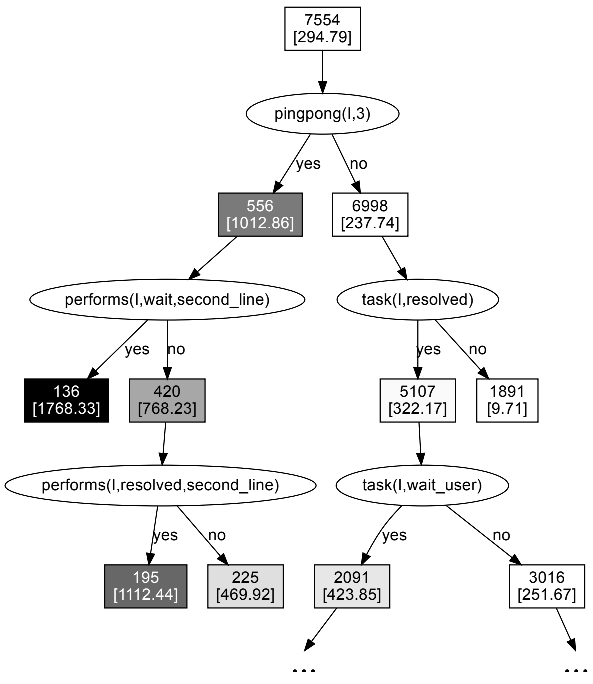


Figure 6: A second decision tree induced from the event log of the incident management process

The new predicate can then be defined as follows:

$$\begin{array}{l} pingpong(I,N) \text{ :-} \\ \quad aggregate\_all(count, handover(I,\_,\_), M), \\ \quad M >= N. \end{array}$$

which yields true if incident I contains at least N handovers of work between different support lines.

The predicate *aggregate_all*/3 is an aggregation operator defined in some PROLOG implementations (namely SWI-PROLOG [31]) which is being used here to count how many facts in the form $handover(I,\_,\_)$ can be inferred from the event log. The result is stored in the variable M, which is compared to N. The condition $M >= N$ guarantees that $pingpong(I,N)$ is true if there are N or more handovers of work.

Figure 6 shows the new decision tree induced after replacing *handover*/3 with the new version and adding *pingpong*/2 to the set of available predicates. In comparison with Figure 5, the most important difference is right at the top of the tree, where we have $pingpong(I,3)$ instead of $task(I,resolved)$.

The split based on $pingpong(I,3)$ divides incidents into those that have less than three handovers, and those which have three handovers or more. Since there are only three support lines, and every incident begins in the first line, an incident with less than three handovers must have been sent from first line to second line and back, or from first line to second line and then on to third line, for example. On the other hand, an incident with three handovers must have visited the second line or the third line several times, meaning that there was definitely some amount of ping-pong behavior involved.

The predicate $task(I,resolved)$ can still be found in Figure 6, but it is now at the second level in the tree. This means that ping-pong behavior has such a large impact on the lifetime of incidents that any other factor becomes of second importance, including the effect of the wait-user status, which appears only at the third level in the tree.

4.4. Finding new custom predicates

In this case study we were able to verify certain causes of delay which were already known from the start. In particular, with the custom predicate *pingpong*/2 it became evident that ping-pong behavior is a major cause of delay. However, one might wonder how the analysis would have been carried out if these causes of delay had not been provided beforehand. For example, how could the analyst ever come up with the idea of ping-pong behavior if this had not been suggested before? This section explains how the analyst could have proceeded to find new custom predicates for such high-level concepts.

In the absence of any clue about the possible reasons of delay, the analyst could start with the predicates defined in Section 2.3, but instead of using them all at once (which would yield the tree in Figure 5), it is possible to try them one at a time to gain a feeling for which features have the largest impact on the duration of the process. For example, by starting with the *task*/2 predicate, the tree in Figure 7 would immediately raise suspicion about the "wait" and "wait user" status.

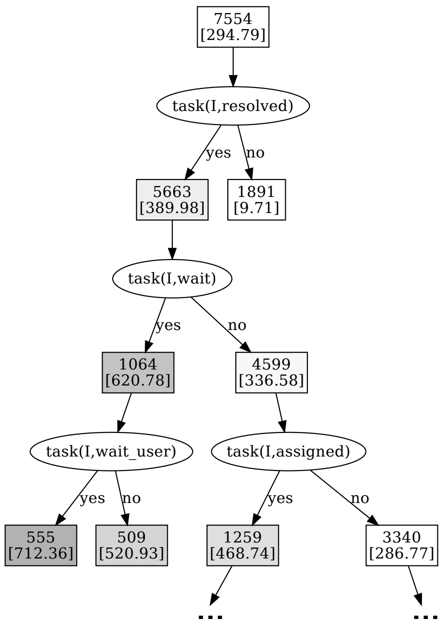


Figure 7: Decision tree induced with the *task*/2 predicate only

The analyst could therefore define a custom predicate,

$$\begin{array}{l} has\_wait(I) \text{ :- } task(I,wait). \\ has\_wait(I) \text{ :- } task(I,wait\_user). \\ has\_wait(I) \text{ :- } task(I,wait\_implementation). \end{array}$$

to capture the fact that a given instance has some kind of wait status (*wait* OR *wait_user* OR *wait_implementation*). Here, the custom predicate involves multiple statements of the same predicate (*task*/2), but it might as well combine different predicates into an OR-clause.

When the analyst considers the predicate *together*/3 in isolation, the tree in Figure 8 immediately points to a suspicious delay when the three support lines are involved.

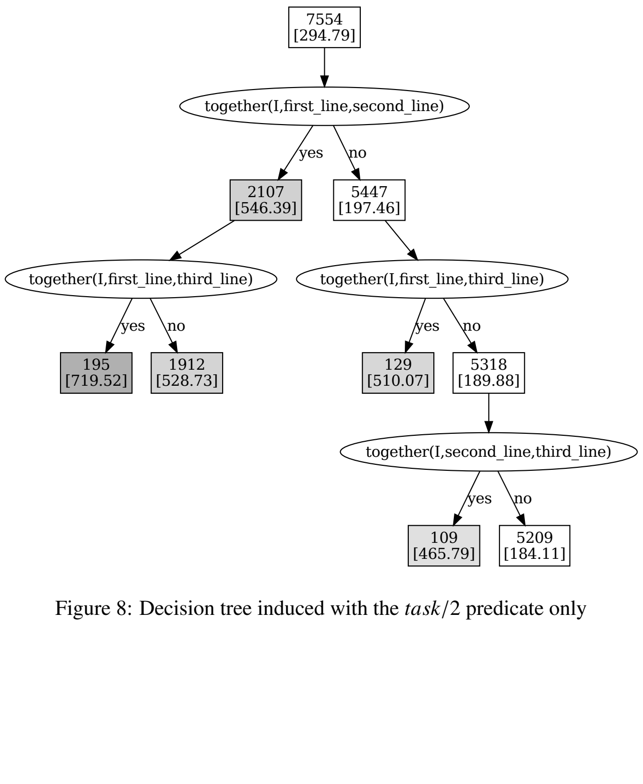

7554 [294.79]
together(I,first_line,second_line)
yes
no
2107 [546.39]
5447 [197.46]
together(I,first_line,third_line)
together(I,first_line,third_line)
yes
no
yes
no
195 [719.52]
1912 [528.73]
129 [510.07]
5318 [189.88]
together(I,second_line,third_line)
yes
no
109 [465.79]
5209 [184.11]

Figure 8: Decision tree induced with the *task*/2 predicate only

The analyst could therefore define a custom predicate,

$$\begin{aligned} all\_three\_lines(I) \text{ :- } & user(I, first\_line), \\ & user(I, second\_line), \\ & user(I, third\_line). \end{aligned}$$

to capture the fact that the three lines work together in the same incident (*first_line* AND *second_line* AND *third_line*). Again, the custom predicate involves multiple statements of the same predicate (*user*/2), but it might as well combine different predicates into an AND-clause.

Such kind of OR- and AND-clauses are the building blocks to create new custom predicates that capture high-level concepts or domain-specific issues concerning the scenario at hand.

After considering the *handover*/3 predicate in isolation, the analyst would realize that the delay is related not only to the fact that the three support lines work together, but also to the fact that they keep handing over incidents back and forth between each other. At that point, the analyst would be led to define some sort of custom predicate to capture that ping-pong behavior, as we did in Section 4.3.

On a final note, the discovery of such new custom predicates should be validated by a domain expert or process owner with a profound understanding of the business process, since such knowledge could help simplify the generalization of observations (e.g. moving from handover and together predicates towards a new ping-pong predicate). This validation could even be crucial as the proposed generalization might not be appropriate without such key knowledge or understanding.

5. Related work

The induction of logical decision trees can be seen as a form of Inductive Logic Programming (ILP) [32, 33]. In ILP terms, the logical representation of the event log that we introduced in Section 2.1 would be called the *input examples*, and the rules that we introduced in Sections 2.2 and 2.3 would be called the *background knowledge*. The top-down induction of a logical decision tree is similar to the way an ILP algorithm searches for an hypothesis that covers the input examples.

ILP techniques have been used before in the area of process mining, but not for the purpose we describe here. For example, [34] uses ILP and Tilde to learn the preconditions of each activity from the event log and from a set of negative events that are generated artificially to discover the process model. Ref. [20] uses a related technique called *inductive constraint logic* [35] to learn declarative models from an event log containing traces labeled as compliant or non-compliant. Also, [36] uses ILP combined with a planning algorithm in a framework that supports a continuous learning of business processes.

Some recent works [37, 38, 39, 40] are more closely related to our approach in the sense that they use classification techniques for the analysis of performance and other perspectives. Ref. [37] proposes an architecture for a business process intelligence tool that makes use of classification techniques for the analysis of process behavior. Ref. [38] analyzes how case data affects the routing of process instances with the help of decision trees. Ref. [39] describes an approach that uses decision trees to find out whether there is a relationship between the workload of resources and the fact that the process gets overdue (i.e. duration exceeds a threshold); for this purpose, the authors make use of an event log that is enriched with workload information. Finally, ref. [40] uses decision trees to answer general questions based on characteristics extracted from the event log; although process duration could be one of those characteristics, the framework requires characteristics to be discretized and this discretization may have large repercussions in the results.

Our present approach differs from those previous works in that we do not place any special requirements on the event log. Rather, we translate the event log, as is, into a first-order logic representation using a set of predicates, and we provide the analyst with the possibility of discovering the cause of delays using high-level concepts defined as rules over those predicates. Also, by using regression trees, process instances are classified based on their actual duration (i.e. using time as a continuous variable), rather than on discrete categories or intervals as in [37, 39, 40]. This allows us to analyze different amounts of delay without the need to define a strict borderline between classes of delayed and non-delayed instances.

6. Conclusion

Typically, the event logs used for process mining contain information about the tasks that have been performed and the users who have performed them. In addition, from the timestamps of events it is possible to calculate the duration of each process instance. For those instances which take longer to complete, there may be several possible reasons for the delay. In this work we have shown that if the delay is caused by specific tasks, by specific users, or by some relationship between them, then it is possible to discover such cause of delay from the simple facts recorded in the event log.

Logical decision trees are a powerful and effective tool for this purpose. As a classifier, the logical decision tree picks up the most relevant predicates to split the input data into groups according to the desired target variable (i.e. duration). These predicates may represent common concepts in the process mining domain, such as the flow between tasks, the handover of work between users, etc., but it is also possible to define custom predicates to analyze specific causes of delay.

Overall, we may have given the impression that the approach is straightforward. However, applying the approach in practice is not without some ingenuity, since translating the event log into a logical representation may require some adjustments, as it happened in the first case study. Furthermore, even when everything has been done correctly, the analysis may depend on a correct interpretation of the decision tree, together with an assessment of whether custom predicates should be introduced, as we did in the second case study.

For these reasons, in future work we will be looking into the possibility of using heuristics or other mechanisms to assist in the translation of the event log, and to provide the analyst with some indication as to whether the use of standard predicates alone provides a good classification of process instances.

Acknowledgements

We would like to thank Prof. Hendrik Blockeel at KU Leuven for providing us with access to ACE/Tilde 1.2.20, which we used in this work.

We would also like to thank Prof. Junichi Iijima at the Tokyo Institute of Technology for an interesting exchange of ideas in a preliminary stage of this work.

References

[1] W. M. P. van der Aalst, A. J. M. M. Weijters, L. Maruster, Workflow mining: Discovering process models from event logs, IEEE Transactions on Knowledge and Data Engineering 16 (2004) 1128–1142.

[2] W. M. P. van der Aalst, A. J. M. M. Weijters, Process mining: a research agenda, Computers in Industry 53 (3) (2004) 231–244.

[3] A. J. M. M. Weijters, W. M. P. van der Aalst, A. K. A. de Medeiros, Process mining with the HeuristicsMiner algorithm, Tech. Rep. WP 166, Eindhoven University of Technology (2006).

[4] A. K. A. de Medeiros, A. J. M. M. Weijters, W. M. P. van der Aalst, Genetic process mining: an experimental evaluation, Data Mining and Knowledge Discovery 14 (2) (2007) 245–304.

[5] W. M. P. van der Aalst, M. Song, Mining social networks: Uncovering interaction patterns in business processes, in: Business Process Management, Vol. 3080 of LNCS, Springer, 2004, pp. 244–260.

[6] W. M. P. van der Aalst, H. A. Reijers, M. Song, Discovering social networks from event logs, Computer Supported Cooperative Work 14 (6) (2005) 549–593.

[7] P. Hornix, Performance analysis of business processes through process mining, Master's thesis, Eindhoven University of Technology (2007).

[8] R. S. Mans, Workflow support for the healthcare domain, Ph.D. thesis, Eindhoven University of Technology (2011).

[9] R. S. Mans, M. H. Schonenberg, M. Song, W. M. P. van der Aalst, P. J. M. Bakker, Process mining in healthcare – a case study, in: Proceedings of the 1st International Conference on Health Informatics, Vol. 1, INSTICC, 2008, pp. 118–125.

[10] W. M. P. van der Aalst, Process Mining: Discovery, Conformance and Enhancement of Business Processes, Springer, 2011.

[11] A. Rozinat, R. S. Mans, M. Song, W. M. P. van der Aalst, Discovering simulation models, Information Systems 34 (3) (2009) 305–327.

[12] H. Blockeel, L. D. Raedt, Top-down induction of first order logical decision trees, Artificial Intelligence 101 (1-2) (1998) 285–297.

[13] H. Blockeel, Top-down induction of first order logical decision trees, Ph.D. thesis, Department of Computer Science, K.U. Leuven, Belgium (December 1998).

[14] W. M. P. van der Aalst, H. A. Reijers, A. J. M. M. Weijters, B. F. van Dongen, A. K. A. de Medeiros, M. Song, H. M. W. Verbeek, Business process mining: An industrial application, Information Systems 32 (5) (2007) 713–732.

[15] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, W. M. P. van der Aalst, The ProM framework: A new era in process mining tool support, in: Applications and Theory of Petri Nets 2005, Vol. 3536 of LNCS, Springer, 2005, pp. 444–454.

[16] W. M. P. van der Aalst, Making work flow: On the application of Petri nets to business process management, in: Application and Theory of Petri Nets 2002, Vol. 2360 of LNCS, Springer, 2002, pp. 1–22.

[17] L. Wen, J. Wang, W. M. P. van der Aalst, B. Huang, J. Sun, A novel approach for process mining based on event types, Journal of Intelligent Information Systems 32 (2) (2009) 163–190.

[18] M. Pesic, W. M. P. van der Aalst, A declarative approach for flexible business processes management, in: Business Process Management Workshops, Vol. 4103 of LNCS, Springer, 2006, pp. 169–180.

[19] M. Pesic, H. Schonenberg, W. van der Aalst, DECLARE: Full support for loosely-structured processes, in: 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), IEEE Computer Society, 2007, pp. 287–298.

[20] F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, S. Storari, Exploiting inductive logic programming techniques for declarative process mining, in: Transactions on Petri Nets and Other Models of Concurrency II, Vol. 5460 of LNCS, Springer, 2009, pp. 278–295.

[21] L. Rokach, O. Maimon, Data Mining with Decision Trees: Theory and Applications, World Scientific, 2008.

[22] J. R. Quinlan, Induction of decision trees, Machine Learning 1 (1) (1986) 81–106.

[23] T. M. Cover, J. A. Thomas, Elements of Information Theory, 2nd Edition, Wiley, 2006.

[24] J. R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann, 1993.

[25] J. R. Quinlan, Decision trees and multi-valued attributes, in: J. E. Hayes, D. Michie, J. Richards (Eds.), Towards an Automated Logic of Human Thought, Vol. 11 of Machine Intelligence, Clarendon Press, 1988, Ch. 13, pp. 305–318.

[26] S. Gelfand, C. S. Ravishankar, E. Delp, An iterative growing and pruning algorithm for classification tree design, IEEE Transactions on Pattern Analysis and Machine Intelligence 13 (2) (1991) 163–174.

[27] H. Blockeel, L. D. Raedt, J. Ramon, Top-down induction of clustering trees, in: Proceedings of the 15th International Conference on Machine Learning (ICML'98), Morgan Kaufmann, 1998, pp. 55–63.

[28] L. Breiman, J. Friedman, C. J. Stone, R. A. Olshen, Classification and Regression Trees, Chapman and Hall, 1984.

[29] B. van Dongen, B. Weber, D. R. Ferreira, J. D. Weerdt (Eds.), Proceedings of the 3rd Business Process Intelligence Challenge, Vol. 1052 of CEUR Workshop Proceedings, 2013.

[30] J. van Bon, M. Pieper, A. van der Veen, T. Verheijen (Eds.), Foundations of IT Service Management based on ITIL, Van Haren, 2005.

[31] J. Wielemaker, T. Schrijvers, M. Triska, T. Lager, Swi-prolog, Theory and Practice of Logic Programming 12 (Special Issue 1-2) (2012) 67–96.

[32] S. Muggleton, Inductive logic programming, New Generation Computing 8 (4) (1991) 295–318.

[33] N. Lavrac, S. Dzeroski, Inductive Logic Programming: Techniques and Applications, Ellis Horwood, New York, 1994.

[34] S. Goedertier, D. Martens, B. Baesens, R. Haesen, J. Vanthienen, Process mining as first-order classification learning on logs with negative events, in: Business Process Management Workshops, Vol. 4928 of LNCS, Springer, 2008, pp. 42–53.

[35] L. D. Raedt, W. V. Laer, Inductive constraint logic, in: Algorithmic Learning Theory, Vol. 997 of LNCS, Springer, 1995, pp. 80–94.

[36] H. M. Ferreira, D. R. Ferreira, An integrated life cycle for workflow management based on learning and planning, International Journal of Cooperative Information Systems 15 (4) (2006) 485–505.

[37] D. Grigori, F. Casati, M. Castellanos, U. Dayal, M. Sayal, M.-C. Shan, Business process intelligence, Computers in Industry 53 (3) (2004) 321–343.

[38] A. Rozinat, W. M. P. van der Aalst, Decision mining in ProM, in: Business Process Management, Vol. 4102 of LNCS, Springer, 2006, pp. 420–425.

[39] S. Suriadi, C. Ouyang, W. M. P. van der Aalst, A. ter Hofstede, Root cause analysis with enriched process logs, in: Business Process Management Workshops, Vol. 132 of LNBIP, Springer, 2013, pp. 174–186.

[40] M. de Leoni, W. M. P. van der Aalst, M. Dees, A general framework for correlating business process characteristics, in: Business Process Management, Vol. 8659 of LNCS, Springer, 2014, pp. 250–266.