Knowledge and Information Systems (2020) 62:2539–2575
https://doi.org/10.1007/s10115-019-01430-6

REGULAR PAPER

Case notion discovery and recommendation: automated event log building on databases

E. González López de Murillas[1] · H. A. Reijers[1,2] · W. M. P. van der Aalst[1,3]

Received: 24 August 2018 / Revised: 4 December 2019 / Accepted: 8 December 2019 /
Published online: 31 December 2019

Abstract
Process mining techniques use event logs as input. When analyzing complex databases, these event logs can be built in many ways. Events need to be grouped into traces corresponding to a case. Different groupings provide different views on the data. Building event logs is usually a time-consuming, manual task. This paper provides a precise view on the case notion on databases, which enables the automatic computation of event logs. Also, it provides a way to assess event log quality, used to rank event logs with respect to their interestingness. The computational cost of building an event log can be avoided by predicting the interestingness of a case notion, before the corresponding event log is computed. This makes it possible to give recommendations to users, so they can focus on the analysis of the most promising process views. Finally, the accuracy of the predictions and the quality of the rankings generated by our unsupervised technique are evaluated in comparison to the existing regression techniques as well as to state-of-the-art learning to rank algorithms from the information retrieval field. The results show that our prediction technique succeeds at discovering interesting event logs and provides valuable recommendations to users about the perspectives on which to focus the efforts during the analysis.

Keywords Process mining · Event log · Database · Case notion · Recommendation · Ranking

✉ E. González López de Murillas
e.gonzalez@tue.nl; edu.gonza.lopez@gmail.com

H. A. Reijers
h.a.reijers@tue.nl; h.a.reijers@uu.nl

W. M. P. van der Aalst
w.m.p.v.d.aalst@tue.nl; wvdaalst@pads.rwth-aachen.de

1 Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands

2 Department of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands

3 Department of Computer Science, RWTH Aachen University, Aachen, Germany

1 Introduction

Process mining [1] is a field of data science devoted to the analysis of process behavior. This data-driven analysis makes it possible to discover models, analyze performance, detect deviations, identify bottlenecks and inefficiencies, make improvements, monitor the behavior, and make predictions, all related to business processes in a large variety of domains. To perform these kinds of analyses, process mining techniques require event logs as input. An event log is a set of process instances or traces, each of which contains a set of events. Events represent occurrences of process tasks or activities at a certain point in time.

Obtaining event logs is not a trivial matter. Data extraction and preparation are, very often, the most time-consuming tasks (around 80% of the time) and one of the most costly (around 50% of the cost) in data analysis projects [2]. This is due to the fact that data come in many forms, while a lot of manual work and domain knowledge is needed to obtain meaningful event logs from it. Additionally, not all systems worth analyzing are process-aware information systems (PAIS), i.e., event data are not explicitly recorded as a first-class citizen within the system. If that is the case, additional work needs to be performed to obtain the events required to build logs for analysis. Another reason for the high cost in time and effort of the event log building phase is that, in many cases, domain knowledge about the system at hand is simply not available. Analysts need to interview the business owners and database managers to understand what parts of the event data can be interesting to look into. This interaction often requires several iterations and a large time investment from all parties.

The principal idea behind log building is to correlate events in such a way that they can be grouped into traces to form event logs. Classical approaches would use a common attribute to correlate events. This is a valid method in scenarios where the data schema has a star shape [3] (Fig. 1a): there is a central table and the rest are directly related to it, with at least one column in common, which can be used as a case notion. However, we consider the scenario in which some pairs of events may not have any attribute in common. This is the case for a snowflake schema [3] (Fig. 1b), which resembles the shape of a star schema, with the difference that, at the points, we find tables that only hold a transitive relation with the central table. In practice, we often find databases which schema presents a higher complexity than a star or snowflake structure (Fig. 1c). In that case, there are many combinations in which events can be grouped. These combinations cannot be arbitrary, but must obey some criteria with a business meaning, e.g., group the *invoice* and *delivery* events by means of the *invoice_id* field present in the former ones. Also, more complex combinations can be defined when transitive relations are considered for the grouping, e.g., group the *invoice*, *delivery*, and *bill* events according to the field *invoice_id* in delivery events and the field *delivery_id* in the bill events. Each of these examples captures what we will refer to as a *case notion*, i.e., a way to look at event data from a specific perspective.

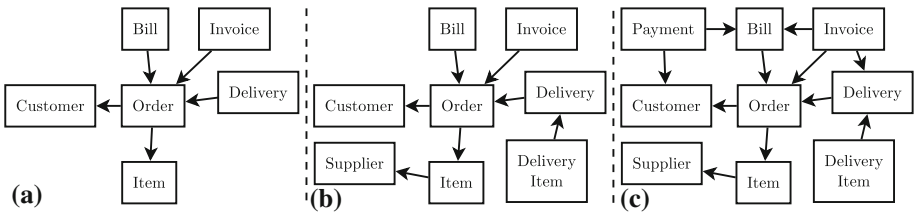


Fig. 1 Example of database schema types: **a** star, **b** snowflake and **c** arbitrary

When dealing with vast datasets from complex databases, the existence of many potential case notions is evident. Enterprise resource planning (SAP, Oracle EBS, Dolibarr), hospital information systems (ChipSoft, GE Centricity, AGFA Integrated Care), and customer relationship management (Salesforce, MS Dynamics, SugarCRM) are examples of systems powered by large databases where multi-perspective analysis can be performed. According to different case notions, many different event logs can be built. *The research problem we tackle in this paper is how to choose the right perspective on the data, which is a crucial step in order to obtain relevant insights*. It is common practice to perform this selection by hand-written queries, usually by an analyst with the right domain knowledge about the system and process under study. However, when facing complex data schemas, writing such queries can become a very complicated task, especially when many tables are involved.

A naive way to tackle the exploration of complex databases is to automatically generate all the possible case notions as combinations of tables. This can lead to many event log candidates, even for a small database. The combinatorial problem is aggravated in more complex scenarios, i.e., with hundreds of tables involved. Given a weakly connected[1] data schema of 90 tables, there exist 4005 combinations of pairs of tables.[2] If we consider combinations of 3 tables instead, the number increases to 117,480, even before considering the many different paths that could connect the tables in each combination. In such cases, the automated building of logs for all possible table combinations may still be possible, but has proven to be computationally very expensive: in the hypothetical case that building an event log would take 4 s on average, building the event logs for a data schema with 90 tables and 10,000 possible case notions would take approximately 11 h. Even if we spend the time to compute all of them, we still need to inspect 10,000 event logs to find out which perspective is both meaningful and interesting.

A way to mitigate the combinatorial explosion is to reduce the case notion search space as much as possible. Identifying the most interesting event logs would help to prioritize the most promising views on the data for its analysis. The challenge of identifying the most promising views is related to the log quality problem. The *log quality problem* is concerned with identifying the properties that make an event log more suitable to be analyzed, i.e., the characteristics that increase the probability of obtaining valuable insights from the analysis of such an event log. The choices made during the log building process have an effect on the log quality [4]. Also, metrics to assess structural log properties have been proposed by some authors [5], which may be important to assess log quality.

The main contributions of this work are: (a) formally defining complex case notions to adopt different perspectives on event data; (b) automatically generating candidate case notions on a dataset; (c) assessing the quality of the resulting event logs; (d) automatically predicting an event log's quality before it is built; (e) sorting the case notions according to their relative quality from the analysis point of view. This drastically reduces the computational cost avoiding the generation of uninteresting event logs. In order to achieve these goals, data must be extracted from the original system and transformed to fit into a certain structure. This structure should be able to capture both the process and the data sides of the system under study. The techniques proposed in this paper have been implemented in a framework and evaluated with respect to related ranking algorithms. The approach yields promising results in terms of performance and accuracy on the computation of event log rankings.

[1] Weakly connected graph: a directed graph such that, after replacing all of its directed edges with undirected ones, it produces a connected graph. A connected graph is one such that, for any pair of nodes (a, b), there is a path from a to b.

[2] For a set of n elements (n tables), the number of k-combinations (combinations of k tables) is $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

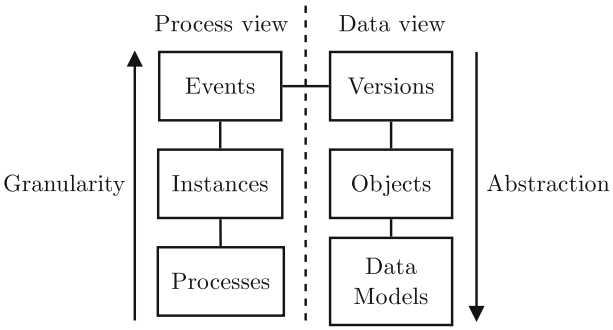


Fig. 2 High-level structure of the OpenSLEX meta-model

The paper is structured as follows. Section 2 introduces some preliminary concepts about how information contained in databases can be extracted and structured. Section 3 introduces a running example. Section 4 defines the concept of case notion and proposes a formalized way to build event logs. Section 5 provides a way to automatically assess the quality of event logs. Section 6 proposes a technique to predict the quality of an event log before it is computed, reducing the computation time several orders of magnitude. Section 7 presents the implementation of all the techniques described in this work. The result of the evaluation is presented in Section 8. Related work is discussed in Sect. 9. Lastly, Sect. 10 presents the conclusions of this study.

2 Preliminaries

To enable the application of process mining and the techniques proposed in this work, we need access to the database of the system under study. This information should be extracted and transformed to fit into a specific data structure. An appropriate structure has been previously defined as a meta-model [6] and implemented in a *queryable* file format called OpenSLEX. Figure 2 shows a high-level view of the meta-model that describes the OpenSLEX format. The meta-model captures all the necessary aspects to enable the application of our techniques. This section describes the structure of OpenSLEX and provides the necessary background to understand the techniques proposed in the coming sections.

Standards of reference like XES [7] are focused on the process view (events, traces, and logs) of systems. OpenSLEX supports all concepts present in XES, but in addition, also considers the data elements (data model, objects, and versions) as an integral part of its structure. This makes it more suitable for database environments where only a small part of the information is process oriented (i.e., events) with respect to the rest of data objects of different classes that serve as an augmented view of the process information. The OpenSLEX format is supported by a meta-model that considers *data models* and *processes* as the entities at the highest abstraction level. These entities define the structure of more granular elements like *logs*, *cases*, and *activity instances* with respect to processes, and *objects* with respect to classes in the data model. Each of these elements at the intermediate level of abstraction can be broken apart into more granular pieces. This way, *cases* are formed by *events*, and *objects* can be related to several *object versions*. Both *events* and *object versions* represent different states of a higher-level abstraction (*cases* or *objects*) at different points in time.

Figure 3 depicts the entity-relation diagram of the OpenSLEX format. Some elements of the meta-model have been omitted from the diagram for the sake of simplicity. A full version of the ER diagram is available online.[3] Each of the entities in the diagram, as represented by a square, corresponds to the basic *entities* of the meta-model as formalized in Definition 2. Also, these entities, together with their relations (diamond shapes), have been grouped in areas that we call *sectors* (delimited by dashed lines). These sectors are: *data models*, *objects*, *versions*, *events*, *cases*, and *process models*. These tightly related concepts provide an abbreviated representation of the meta-model. As can be observed, the entity-relation diagram is divided into six sectors. The purpose of each of them is described below:

- *Data models* this sector is formed by concepts needed to describe the structure of any database system. Many data models can be represented together in this sector, whose main element is the *data model* entity. For each data model, several *classes* can exist. These classes are abstractions of the more specific concept of table, which is commonly found in RDBMSs. Classes contain *attributes*, which are equivalent to table columns in modern databases (e.g., *id*, *name*, *address*, etc.). The references between classes of the same data model are represented with the *relationship* entity. This last entity holds links between a source and a target class.
- *Objects* the *object* entity, part of the objects sector, represents each of the unique data elements that belong to a class. An example of this can be a hypothetical customer with *customer_id* = 75. Additional details of this object are omitted, given that they belong to the next sector.
- *Versions* for each of the unique object entities described in the previous sector, one or many *versions* can exist. A version is an instantiation of an object during a certain period of time, e.g., the customer object with id 75, existed in the database, during a certain period of time, for example from "2015-08-01 14:45:00" to "2016-09-03 12:32:00." During that period of time, the object had specific values for the attributes of the customer class that it belongs to. Therefore, there is a version of customer 75, valid between the mentioned dates, with name "John Smith," address "45, 5th Avenue," and birth date "1990-01-03." If at some point, the value of one of the attributes changed (e.g., a new address), the end timestamp of the previous version would be set to the time of the change, and a new version would be created with the updated value for that attribute, and a start timestamp equal to the end of the previous version, e.g., version_1 = {object_id = 75, name = "John Smith," address = "45, 5th Avenue," birth_date = "1990-01-03," start_timestamp = "2015-08-01 14:45:00," end_timestamp = "2016-09-03 12:32:00"}, and version_2 = {object_id = 75, name = "John Smith," address = "floor 103, Empire State Building," birth_date = "1990-01-03," start_timestamp = "2016-09-03 12:32:00," end_timestamp = NONE }. Note that the value of *end_timestamp* for the newly created object version (*version_2*) is NONE. That means that it is the current version for the corresponding object (*object_id* = 75). Another entity reflected in this sector is the concept of *relation*. A relation is an instantiation of a relationship and holds a link between versions of objects that belong to the source and target classes of the relationship. For example, a version of a booking object can be related to another version of a customer object by means of a relation instance, as long as a relationship exists from class *booking* to class *customer*.
- *Events* this sector collects a set of events, obtained from any available source (database tables, redo-logs, change records, system logs, etc.). In this sector, events appear as a collection, not grouped into traces (such grouping is reflected in the next sector). In order to keep process information connected to the data side, each event can be linked to one or

[3] https://github.com/edugonza/OpenSLEX/blob/master/doc/meta-model.png.

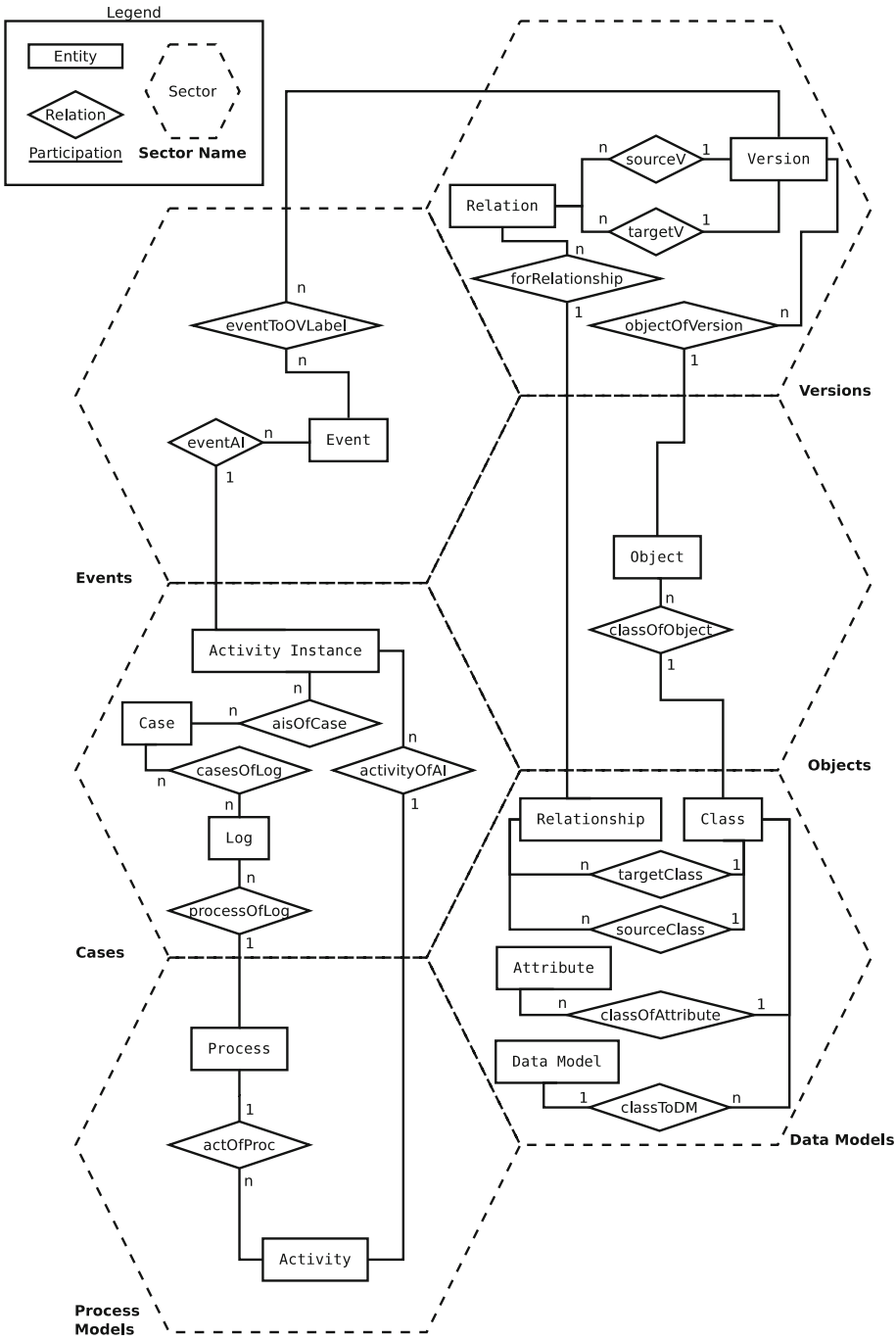


Fig. 3 ER diagram of the OpenSLEX meta-model. The entities have been grouped into sectors, delimited by the dashed lines

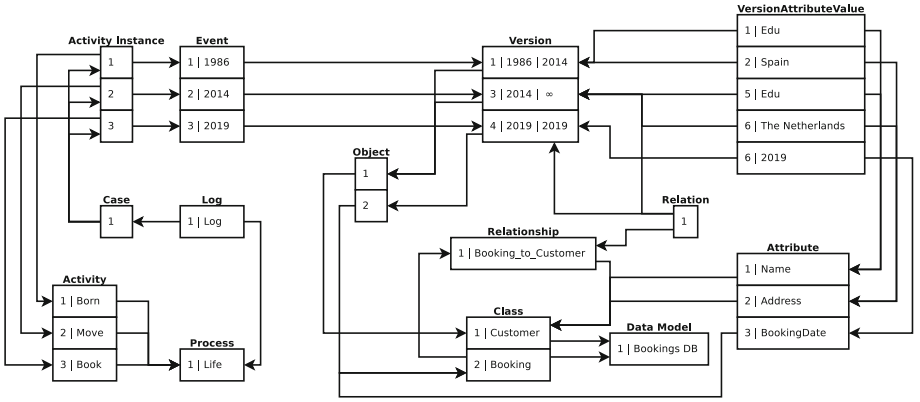


Fig. 4 Diagram of an instance of the OpenSLEX meta-model

many object versions by means of a label (*eventToOVLabel*). This label allows specifying what kind of interaction exists between the event and the referred object version, e.g., *insert*, *update*, *delete*, *read*, etc. Events hold details such as *timestamp*, *life cycle*, and *resource* information, apart from an arbitrary number of additional event attributes.

– *Cases and instances* the entities present in this sector are very important from the process mining point of view. The events by themselves do not provide much information about the control flow of the underlying process, unless they are correlated and grouped into traces (or cases). First, the *activity instance* entity should be explained. This entity is used to group events that refer to the same instance of a certain activity with different values for its life cycle, e.g., the execution of an activity generates one event for each phase of its life cycle. Both events, referring to the same execution of an activity, are grouped into the same activity instance. Next, as in any other event log format, activity instances can be grouped in *cases*, and these cases, together, form a *log*.
– *Process models* the last sector contains information about *processes*. Several processes can be represented in the same meta-model. Each process is related to a set of *activities*, and each of these activities can be associated with several activity instances, contained in the corresponding *cases and instances* sector.

Figure 4 shows an example of an instance of the OpenSLEX meta-model. For the sake of clarity, the model has been simplified, but the main structure remains. We see that there is a global *data model*. All the *classes* belong to it: "Customer" and "Booking." Also, there are three *attributes*: "Name," "Address," and "BookingDate." The first two attributes belong to the class "Customer." The third one belongs to "Booking." There is a *relationship* connecting bookings to customers named "Booking_to_Customer." Two *objects* exist. The first object has two *versions*. Each version of the customer object has *values* for the corresponding attributes. We see that the first customer version corresponds to a customer named "Edu" while he lived in "Spain," from 1986 to 2014. The second version corresponds to the same customer, while he lived in "The Netherlands" from 2014 until the present. There is another object version that belongs to the second object, a booking object. The "BookingDate" value of this version is "2019." There is a *relation* (an instance of the relationship "Booking_to_Customer"), that connects the second object version of customer *1* to the first object version of booking *1*. On the left side of the figure, we see that three *events* exist. The first event, related to the first version of customer *1*, is linked to the *activity* "Born," and happened in 1986. The second

event, linked to the activity "Move," happened in 2014 and is related to the second version of the same customer. Finally, the third event is linked to the activity "Book" and is linked to the first version of booking *1*. Each event belongs to its own *activity instance*. All activity instances belong to one *case*. This case belongs to a *log* of the *process* "Life."

The OpenSLEX format makes use of a SQL schema to store all the information, and a Java API[4] is available for its integration in other tools. An evaluation of the use of OpenSLEX [6] in several environments tackles the data extraction and transformation phase and demonstrates its flexibility and potential to enable standard querying and advanced data analyses. To keep this paper self-contained and to provide the necessary background for the understanding of this work, a simplified version of the meta-model is formally presented below. Every database system contains information structured with respect to a data model. Definition 1 provides a formalization of a data model in the current context.

Definition 1 (*Data model*) A data model is a tuple $DM = (CL, AT, classOfAttribute, RS, sourceClass, targetClass)$ such that

- CL is a set of class names,
- AT is a set of attribute names,
- $classOfAttribute \in AT \rightarrow CL$ is a function that maps each attribute to a class,
- RS is a set of relationship names,
- $sourceClass \in RS \rightarrow CL$ is a function mapping each relationship to its source class,
- $targetClass \in RS \rightarrow CL$ is a function mapping each relationship to its target class.

Data models contain classes (i.e., tables), which contain attribute names (i.e., columns). Classes are related by means of relationships (i.e., foreign keys). Definition 2 formalizes each of the entities of the meta-model and shows the connection between them.

Definition 2 (*Connected meta-model*) Let V be some universe of values and TS a universe of timestamps. A connected meta-model is defined as a tuple $CMM = (DM, OC, classOfObject, OVC, objectOfVersion, EC, eventToOVLabel, IC, eventAI, PMC, activityOfAI, processOfLog)$ such that

- $DM = (CL, AT, classOfAttribute, RS, sourceClass, targetClass)$ is a data model,
- OC is an object collection,
- $classOfObject \in OC \rightarrow CL$ is a function that maps each object to a class,
- $OVC = (OV, attValue, startTimestamp, endTimestamp, REL)$ is a version collection where OV is a set of object versions, $attValue \in (AT \times OV) \nrightarrow V$ is a mapping of pairs of object version and attribute to a value, $startTimestamp \in OV \rightarrow TS$ is a mapping between object versions and start timestamps, $endTimestamp \in OV \rightarrow TS$ is a mapping between object versions and end timestamps, and $REL \subseteq (RS \times OV \times OV)$ is a set of triples relating pairs of object versions through a specific relationship,
- $objectOfVersion \in OV \rightarrow OC$ is a function that maps each object version to an object,
- EC is an event collection such that $EC = (EV, EVAT, eventTimestamp, eventLifecycle, eventResource, eventAttributeValue)$ where EV is a set of events, $EVAT$ a set of event attribute names, $eventTimestamp \in EV \rightarrow TS$ maps events to timestamps, $eventLifecycle \in EV \rightarrow \{start, complete, ...\}$ maps events to life cycle attributes, $eventResource \in EV \rightarrow V$ maps events to resource attributes, and $eventAttributeValue \in (EV \times EVAT) \nrightarrow V$ maps pairs of event and attribute name to values,

[4] https://github.com/edugonza/openslex.

- $eventToOVLabel \in (EV \times OV) \nrightarrow V$ is a function that maps pairs of an event and an object version to a label. The existence of a label associated with an event and an object version, i.e., $(ev, ov) \in dom(eventToOVLabel)$, means that both event and object version are linked. The label defines the nature of the link, e.g *"insert"*, *"update"*, *"delete"*, etc,
- $IC = (AI, CS, LG, aisOfCase, casesOfLog)$ is an instance collection where AI is a set of activity instances, CS is a set of cases, LG is a set of logs, $aisOfCase \in CS \rightarrow \mathcal{P}(AI)$ is a mapping between cases and sets of activity instances,[5] and $casesOfLog \in LG \rightarrow \mathcal{P}(CS)$ is a mapping between logs and sets of cases,
- $eventAI \in EV \rightarrow AI$ is a function that maps each event to an activity instance,
- $PMC = (PM, AC, actOfProc)$ is a process model collection where PM is a set of processes, AC is a set of activities, and $actOfProc \in PM \rightarrow \mathcal{P}(AC)$ is a mapping between processes and sets of activities,
- $activityOfAI \in AI \rightarrow AC$ is a function that maps each activity instance to an activity,
- $processOfLog \in LG \rightarrow PM$ is a function that maps each log to a process.

A connected meta-model provides the functions that make it possible to connect all the entities in the meta-model. However, some constraints must be fulfilled for a meta-model to be considered a valid connected meta-model (e.g., versions of the same object do not overlap in time). The details about such constraints are out of the scope of this paper, but their description can be found in [6]. From now on, any reference to input or extracted data will assume to be in the form of a valid connected meta-model. As we have seen, according to our meta-model description, *events* can be linked to *object versions*, which are related to each other by means of *relations*. These *relations* are instances of data model *relationships*. In database environments, this would be the equivalent of using foreign keys to relate table rows and knowing which events relate to each row. For the purpose of this work, we assume that pairwise correlations between events, by means of related object versions, are readily available in the input meta-model. This means that, prior to the extraction, we know the data schema, i.e., primary and foreign keys, and how events are stored in each table, e.g., which columns contain the timestamp and activity name of each event. The first precondition (knowing the data schema) is fair to assume in most real-life environment. Given the lack of automated approaches in the literature that tackle the challenge of event data discovery, the second precondition (knowing the events) requires having the right domain knowledge in order to extract events. The presented meta-model formalization sets the ground for the definition of *case notion* and *log* that will be presented in the coming sections.

3 Running example

Extracting data contained in an information system's database is a complex task. Very often, we lack the domain knowledge needed to identify business objects and meaningful case notions. Also, understanding complex data schemas can be challenging when the number of tables is beyond what can be plotted and explored intuitively. Consider for example the SAP ERP system. This widespread ERP system is often a target for process mining analysis, as it is used in a multitude of organizations and contains a huge amount of functionalities by means of configurable modules. SAP can run on different database technologies. And its instances always maintain a common data model, which is well known for its complexity. SAP represents a prime example because it is a widely used system. Nevertheless, the approach is highly generic and can be applied in different environments, e.g., alternative ERP tools such

[5] $\mathcal{P}(X)$ is the powerset of X, i.e., $Y \in \mathcal{P}(X)$ if $Y \subseteq X$.

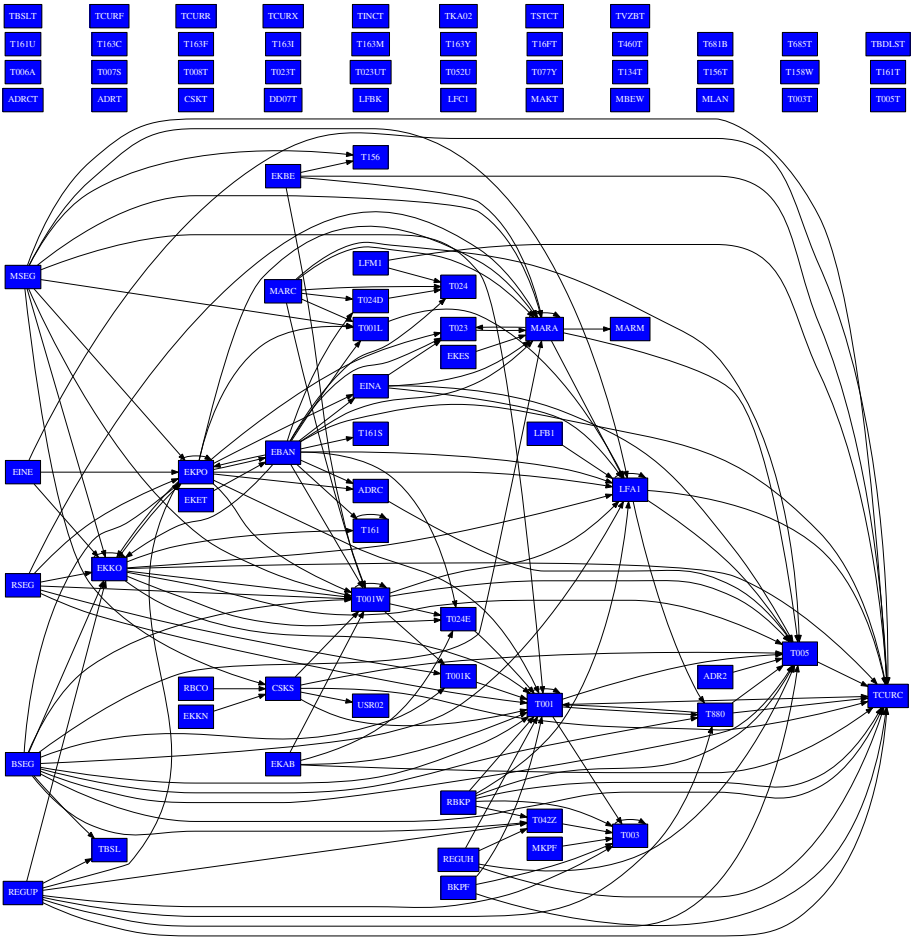


Fig. 5 General view of the data model of the SAP dataset (the table attributes have been omitted)

as Oracle EBS, HIS solutions such as ChipSoft, and CRM systems like Salesforce. Figure 5 depicts the data model of a sample SAP dataset. This dataset, belonging to SAP IDES (Internet Demonstration and Evaluation System), is an instance of a fictitious organization. It contains more than 7M data objects of 87 different classes and more than 26k events corresponding to changes for a subset of the objects present in the database. In the diagram, classes are represented by squares, while edges show the relationships between classes. Table names in SAP are codified in such a way that it is not easy to identify what these classes mean without further documentation. Also, most of the relevant classes are connected to many other. This makes it very difficult to plot the graph in such a way that clusters of classes can be easily identified.

Figure 6 shows in detail a small portion of the graph, where we observe that the *EKKO* (Purchasing Document Header) class is linked, among others, to the *EKPO* (Purchasing Document Item) class. Also, the *EBAN* (Purchase Requisition) class is connected to both. Additionally, the class *EKET* (Scheduling Agreement Schedule Lines) is linked to *EBAN*. According to the official documentation, both *EKKO* (header table) and *EKPO* (item table)

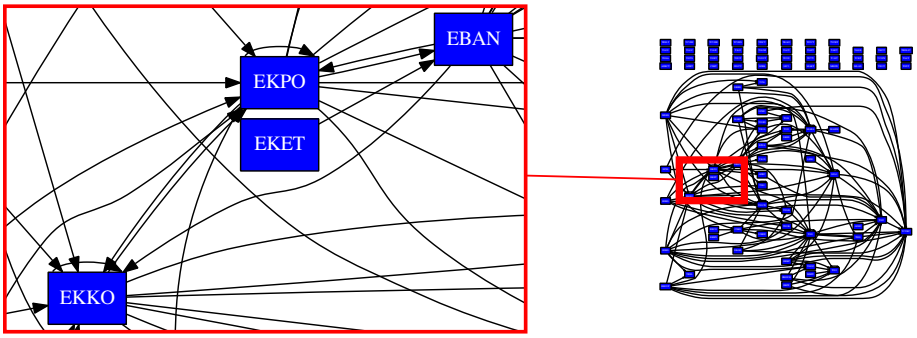


Fig. 6 Detail of the data model of the SAP dataset. *EKKO* and *EKPO* tables refer to purchase documents, while *EBAN* contains information about purchase requisitions

refer to purchasing documents. The *EBAN* class contains information about purchase requisition, and the *EKET* class contains schedule lines related to a scheduling agreement. This could very well be a valid case notion, if we use the connection between the four tables to correlate the corresponding events in traces. However, there are many ways in which this correlation could be constructed. One-to-many relationships can exist between classes, which leads to the well-known problems of *data divergence* (several events of the same type are related to a single case) and *data convergence* (one event is related to multiple cases), as described in [8]. This means that the combination of a subset of classes can yield several, different event logs, depending on the choices made to correlate the events. Should all the purchase items or the same purchase requisition be grouped in the same trace? Should one trace per purchase item exist? Would that mean that the same purchase requisition events would be duplicated in different traces? The fact that these choices exist makes the process of log building a non-trivial task. Section 4 provides a definition of case notion and presents a framework to build event logs effectively, taking into account the aforementioned choices in a formal manner.

4 Case notions and log building

As we have discussed earlier, event log building is a job that has been traditionally performed by analysts. It remains a manual and tedious task, and the time dedicated to it has a large impact on the cost of process mining projects, especially at the start, when the explorative analysis is performed.

When applying the traditional approach to event extraction and event log building, analysts need to perform several manual tasks (Fig. 7). First, a query will be written to extract events from the dataset, selecting a set of required attributes (timestamp, activity name, case identifier), and additional attributes (e.g., resource, life cycle, etc.). These events are then grouped in traces with respect to the value of the chosen case identifier. This method works well in situations when the case notion is clear, and all the events share a common field as case identifier. This is the case, for example, in databases with a star schema [9], where a factual table is at the center, being connected to other dimensional tables in a star-like shape. However, more complex database schemas, like the one exposed in Sect. 3, may lack a common case-identifying attribute between all the events. In that case, transitive relationships between data elements need to be pursued in order to correlate events that are not directly

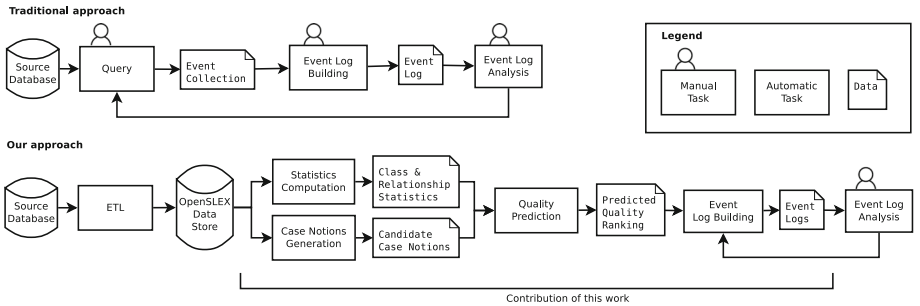


Fig. 7 Overview of the approach for case notion discovery and recommendation

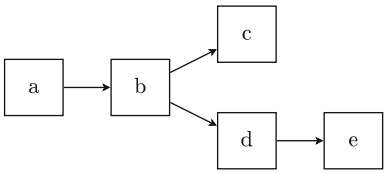


Fig. 8 Simple data schema with 5 nodes (tables) and 4 edges (relationships)

linked (e.g., invoices related to orders that are related to customers). In this situation, queries to extract and correlate events become increasingly complex with respect to the number of tables involved.

Additionally, it may be that we lack the right domain knowledge about the process to be able to identify the correct case notion. When this happens, analysts are forced to approach the data in an explorative way. This means applying a trial and error approach, selecting a specific case notion, building the log, inspecting the result and, if it is not satisfying, repeating the process from a different perspective. The problem of this approach is that, in complex scenarios, it can be extremely time-consuming. Consider the data schema in Fig. 8, where nodes represent tables and edges relationships (foreign keys) between tables. With only 5 tables and 4 relationships, 17 different combinations, or subgraphs, exist: $\{a, b, c, d, e, ab, abc, abcd, abcde, abd, abde, bc, bcd, bcde, bd, bde, de\}$

The approach to event log building presented in this work aims at automating the process as much as possible. As shown in Fig. 7, the goal is to provide input event logs to the user to be analyzed during the explorative phase of a process mining project, while reducing the time spent performing manual tasks. First, we rely of previous work [6] to extract the data from the source database, transforming and storing it in a format suitable for automated analysis. Then, we collect several statistics on different dimensions. These statistics will help us assess which perspectives (case notions) on the data look more interesting and are sorted in a ranking. Finally, based on the ranking, the user can choose which of the suggested case notions to use to automatically obtain an event log for analysis. The methodology that we propose for event log building is explained in detail along the present and coming sections.

The focus of this section is on defining what a case notion is, in order to build logs from event data. Relying on the meta-model structure to correlate events gives us the freedom to apply our log building technique to data coming from different environments, where SAP is just an example. As long as the existing data elements can be matched to the class, object and event abstractions, event correlation will be possible. Therefore, our log building technique will be feasible. The fact that this kind of data and correlations can be obtained in real-life environments has been previously demonstrated in [6]. Our approach defines case notions

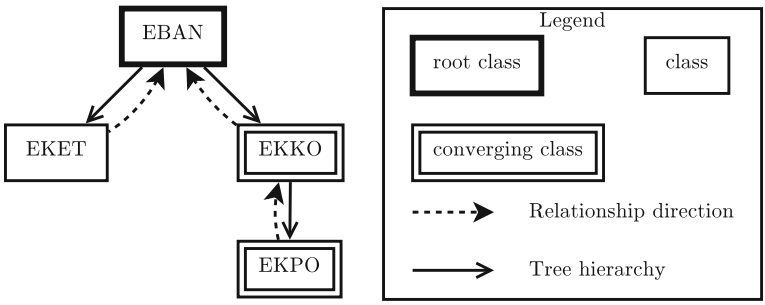


Fig. 9 Sample of a case notion, represented as an annotated rooted tree

based on the data model of the dataset (classes and relationships) and projects the data onto it (objects, object versions, and events) to find build traces with correlated events.

4.1 Defining case notions

We define a case notion (Definition 3) as an annotated rooted tree in which there is always a root node (root class of the case notion). There can be a set of additional regular class nodes, together with some converging class nodes, as children of the root node or other nodes of the subtrees. The root node is the main class of the case notion and triggers the creation of a new case identifier for each object that belongs to it (e.g., a case identifier for a purchase order). Regular nodes will force the creation of a new case identifier when several of its objects relate to one root or regular object (e.g., several deliveries of the same order will result in one case identifier for each delivery). Converging nodes are the ones that allow one case identifier to refer to objects of that same class (e.g., several delivery items linked to the same delivery will be grouped in under the same case identifier).

Definition 3 (*Case notion*) Let us assume a data model $DM = (CL, AT, classOfAttribute, RS, sourceClass, targetClass)$. We define a case notion as a tuple $CN = (C, root, children, CONV, IDC, rsEdge)$ such that:

- $C \subseteq CL$ is the set of classes involved in the case notion,
- $root \in C$ is the root class in the case notion tree,
- $children \in C \rightarrow \mathcal{P}(C)$ is a function returning the children of a class in the case notion tree,
- $CONV \subseteq C$ is the set of classes of the case notion for which convergence is applied. If a class c belongs to $CONV$, all the members of the subtree of c must belong to this set, i.e., $\forall c \in CONV : children(c) \subseteq CONV$,
- $IDC = C \backslash CONV$ is the set of identifying classes that will be used to uniquely identify cases of this case notion,
- $rsEdge \in (C \times C) \rightarrow RS$ is a function returning the relationship of the edge between two classes in the tree such that, $\forall c \in C : \forall c' \in children(c) : \exists rs \in RS : \{c, c'\} = \{sourceClass(rs), targetClass(rs)\} \wedge rsEdge(c, c') = rs$.

Figure 9 shows an example of a case notion combining classes *EBAN*, *EKET*, *EKKO*, and *EKPO*. The class *EBAN* is the root of the case notion. The class *EKET* is a regular child of the root node, while the child node *EKKO* is a converging class. By inheritance, the node *EKPO* is a converging class as well, given that it belongs to a

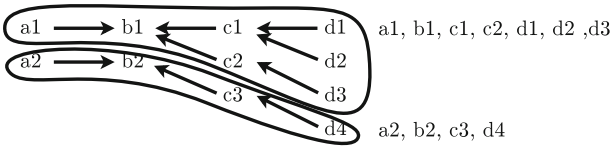


Fig. 10 Links between objects of classes EKET (a1, a2), EBAN (b1, b2), EKKO (c1, c2, c3), and EKPO (d1, d2, d3, d4). The objects have been grouped in two sets, corresponding to the case identifiers computed for the case notion of Fig. 9

subtree of the converging class *EKKO*. Therefore, Fig. 9 is the graphical representation of the case notion *cn* for which $C = \{EBAN, EKET, EKKO, EKPO\}$, $root = EBAN$, $CONV = \{EKKO, EKPO\}$, $IDC = \{EBAN, EKET\}$, $children \in C \rightarrow \mathcal{P}(C)$ such that $children(EBAN) = \{EKET, EKKO\}$, $children(EKKO) = \{EKPO\}$, $children(EKPO) = \emptyset$, and $children(EKET) = \emptyset$, and $rsEdge \in (C \times C) \rightarrow RS$ such that $rsEdge(EKET, EBAN) =$ *fk_eket_to_eban*,[6] $rsEdge(EKKO, EBAN) =$ *fk_ekko_to_eban*, and $rsEdge(EKPO, EKKO) =$ *fk_ekpo_to_ekko*. According to this case notion, each trace will contain events belonging only to one *EBAN* object, only one *EKET* object, but to any *EKKO* or *EKPO* objects that hold a relation with the *EBAN* object represented by the trace. This is due to the fact that *EKKO* and *EKPO* are defined as converging classes in our case notion. The log building process is described in greater detail below.

4.2 Building a log

The process of building an event log can be seen as the projection of a dataset on a certain case notion. First, a set of case identifiers will be constructed, which will determine the objects that will be correlated per trace. Definition 4 describes in more detail how this set of case identifiers is generated. Figure 10 will be used in this section as an example to illustrate the method.

Definition 4 (*Case identifiers*) Let us assume a valid connected meta-model *CMM* and a case notion $CN = (C, root, children, CONV, IDC, rsEdge)$. We define CI as the maximal set[7] of case identifiers such that, each case identifier $ci \in CI$ is a set of objects $ci = \{o \in OC \mid classOfObject(o) \in C\}$ and the following properties apply:

- $\forall o \in ci : classOfObject(o) \in IDC \Rightarrow (\exists o' \in ci : classOfObject(o') = classOfObject(o) \Rightarrow o' = o)$, i.e., cannot exist two objects per identifying class in each case identifier,
- $\exists o \in ci : classOfObject(o) = root$, i.e., one object of the case identifier belongs to the root,
- $R \subseteq (ci \times ci) = \{(o, o') | \exists (rs, ov, ov') \in REL : c = classOfObject(o) \wedge c' = classOfObject(o') \wedge objectOfVersion(ov) = o \wedge objectOfVersion(ov') = o' \wedge rs = rsEdge(c, c') \wedge sourceClass(rs) = c \wedge targetClass(rs) = c'\}$, i.e., R is a relation between two objects of the case identifier such that both objects have at least one link in the original data for a relationship considered in the case notion. To improve readability, we can say that $oRo' \iff (o, o') \in R$,

[6] *fk_** stands for *"foreign key"*, e.g., *fk_eket_to_eban* represents a foreign key from table *EKET* to table *EBAN*.

[7] A is a maximal set for property P if: (a) A satisfies property P and (b) $\forall B \supseteq A$ satisfying property P: $B = A$.

Table 1 Sample object, version, and event identifiers for the classes involved in the case notion

Class	ObjectID	VersionID	EventID	RelationID
EKET	a1	av1	ae1	bv1
EKET	a1	av2	ae2	bv2
EKET	a2	av3	ae3	bv3
EBAN	b1	bv1	be1	–
EBAN	b1	bv2	be2	–
EBAN	b2	bv3	be3	–
EKKO	c1	cv1	ce1	bv2
EKKO	c2	cv2	ce2	bv2
EKKO	c3	cv3	ce3	bv3
EKPO	d1	dv1	de1	cv1
EKPO	d2	dv2	de2	cv1
EKPO	d3	dv3	de3	cv2
EKPO	d4	dv4	de4	cv3

– $|ci| > 1 \Rightarrow \forall (o, o') \in (ci \times ci) : oR^+o'$, i.e., as long as the case identifier contains more than one object, any pair of objects must belong to the transitive closure[8] of the relation R, i.e., directly or transitively related through objects of the case identifier.

Let us consider the sample dataset in Table 1. It corresponds to the tables *EBAN*, *EKET*, *EKKO*, and *EKPO*. In total, there are 11 objects ($\{a1, a2, b1, b2, c1, c2, c3, d1, d2, d3, d4\}$), 13 object versions ($\{av1, av2, av3, bv1, bv2, bv3, cv1, cv2, cv3, dv1, dv2, dv3, dv4\}$), and 13 events ($\{ae1, ae2, ae3, be1, be2, be3, ce1, ce2, ce3, de1, de2, de3, de4\}$). Additionally, there are 10 relations between object versions ($\{av1 \rightarrow bv1, av2 \rightarrow bv2, av3 \rightarrow bv3, cv1 \rightarrow bv2, cv2 \rightarrow bv2, cv3 \rightarrow bv3, dv1 \rightarrow cv1, dv2 \rightarrow cv1, dv3 \rightarrow cv2, dv4 \rightarrow cv3\}$).

The first step to build the event log corresponding to the case notion in Fig. 9 is to build the set of case identifiers. First, we have to find the maximal set of case identifiers that comply with the constrains set by the case notion at hand, i.e., (a) all the objects must belong to the classes in the case notion, (b) at least one object per case identifier must belong to the root class of the case notion, (c) two objects of the same case identifier cannot belong to the same identifying class of the case notion, and (d) all the objects in the same case identifier must be related, either directly or transitively, by means of the relationships specified in the case notion.

Going back to our example, we will construct the set of case identifiers by looking at Fig. 10. In it, we see the relations between objects. Knowing that $\{b1, b2\}$ are the objects belonging to the *EBAN* class and that *EBAN* is the root class of the case notion, we know that exactly one of these objects must be in each of the resulting traces. That means we will generate, at least, two traces. Objects $\{a1, a2\}$ belong to the class *EKET*, which is the other identifying class of the case notion. Only one of these objects is allowed per trace. In this case, each one of them is related to a different *EBAN* object. Because *EKET* and *EBAN* are the only identifying classes of the case notion, we can combine their objects already to create

[8] R^+ is the transitive closure of a binary relation R on a set X if it is the smallest transitive relation on X containing R.

a (non-maximal) set of case identifiers $CI' = \{ci1', ci2'\}$:

$$ci1' = \{a1, b1\}$$
$$ci2' = \{a2, b2\}.$$

The next class to look at in the case notion hierarchy is *EKKO*. There are three objects ($\{c1, c2, c3\}$) belonging to this class. Two of them ($\{c1, c2\}$) are related to the *EBAN* object $b1$. Given that it is a converging class, we can put them in the same case identifier, in this case $ci1'$. The other object ($c3$) is related to the *EBAN* object $b2$. Therefore, it will be inserted in the case identifier $ci2'$. We proceed analogously with the *EKPO* objects $\{d1, d2, d3, d4\}$, given that *EKPO* is a converging class in our case notion as well. Finally, the maximal case identifiers set $CI = \{ci1, ci2\}$ is:

$$ci1 = \{a1, b1, c1, c2, d1, d2, d3\}$$
$$ci2 = \{a2, b2, c3, d4\}.$$

Once the case identifiers have been generated, it is possible to build the log in its final form. First we introduce some useful notation in Definition 5.

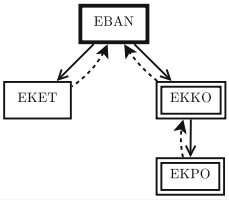
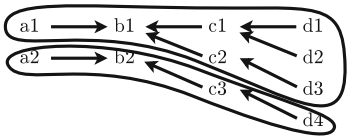
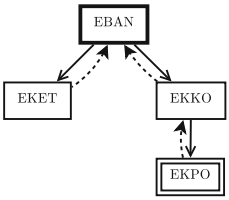
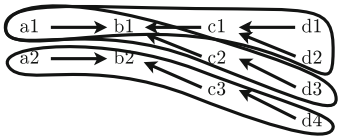
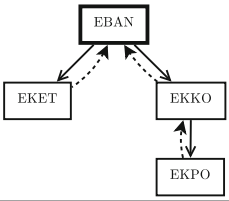
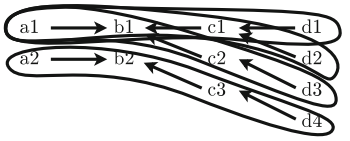
Definition 5 (*Shorthands I*) Given a valid connected meta-model *CMM*, a case notion $CN = (C, root, children, CONV, IDC, rsEdge)$ and a maximal set of case identifiers CI, we define the following shorthands:

- $Act_o = \{act \in AC \mid \exists (e, ov) \in dom(eventToOVLabel) : objectOfVersion(ov) = o \wedge activityOfAI(eventAI(e)) = act\}$, i.e., the set of activities of the activity instances related to an object through its versions and events,
- $ActC_c = \{act \in AC \mid \exists (e, ov) \in dom(eventToOVLable) : objectOfVersion(ov) = o \wedge activityOfAI(eventAI(e)) = act \wedge classOfObject(o) = c\}$, i.e., the set of activities related to a class through its activity instances, events, versions, and objects,
- $O_c = \{o \in OC \mid classOfObject(o) = c\}$, i.e., the set of objects of a certain class $c \in C$,
- $EvO_o = \{e \in EV \mid \exists (e, ov) \in dom(eventToOVLabel) : objectOfVersion(ov) = o\}$, i.e., the set of events of a certain object $o \in OC$,
- $EvC_c = \{e \in EV \mid \exists (e, ov) \in dom(eventToOVLabel) : classOfObject(objectOfVersion(ov)) = c\}$, i.e., set of events of a certain class $c \in C$,
- $E_{ai} = \{e \in EV | ai \in AI \wedge eventAI(e) = ai\}$, i.e., set of events of a certain activity instance $ai \in AI$.

In order to build the final log, we will map a set of activity instances to each object and group them per case identifier to form traces. According to the definition of the OpenSLEX meta-model, an activity instance is a set of events that belong to the same activity and case, e.g., correlated events with different life cycle of the same activity (*start* and *complete* events). In our example, for the sake of clarity, we assume that each activity instance is a singleton with a single event. In fact, we will represent traces as a set of events. Definition 6 provides a formal description of a log and how to build it from a maximal set of case identifiers.

Definition 6 (*Log*) Given a valid connected meta-model *CMM*, a case notion $CN = (C, root, children, CONV, IDC, rsEdge)$ and a maximal set of case identifiers CI, we define a log $l \in CI \rightarrow \mathcal{P}(AI)$ as a deterministic mapping between the set of case identifiers and the powerset of activity instances, such that each of the activity instances in the mapped set is linked to at least one object of the case identifier, i.e., for all $ci \in CI : l(ci) = \{ai \in AI \mid \exists e \in EV : ai = eventAI(e) \wedge \exists ov \in OV : (e, ov) \in dom(eventToOVLabel) \wedge objectOfVersion(ov) \in ci\}$.

Table 2 Case identifiers and final traces built from the sample dataset, according to each of the three case notions

ID	Case notion	Case identifiers and traces
a		 Trace 1: $\{ae1, ae2, be1, be2, ce1, ce2, de1, de2, de3\}$ Trace 2: $\{ae3, be3, ce3, de4\}$
b		 Trace 1: $\{ae1, ae2, be1, be2, ce1, de1, de2\}$ Trace 2: $\{ae1, ae2, be1, be2, ce2, de3\}$ Trace 3: $\{ae3, be3, ce3, de4\}$
c		 Trace 1: $\{ae1, ae2, be1, be2, ce1, de1\}$ Trace 2: $\{ae1, ae2, be1, be2, ce1, de2\}$ Trace 3: $\{ae1, ae2, be1, be2, ce2, de3\}$ Trace 4: $\{ae3, be3, ce3, de4\}$

Assuming that, in our example, each activity instance is represented by a single event, we can build the final log l as the following mapping:

$$\begin{aligned} CI &\rightarrow \mathcal{P}(AI) \\ l : ci1 &= \{ae1, ae2, be1, be2, ce1, ce2, de1, de2, de3\} \\ ci2 &= \{ae3, be3, ce3, de4\} \end{aligned}$$

Of course, different variations of case notions will lead to different event logs, given that the grouping rules will change. Table 2 shows three different case notions, as well as the corresponding case identifiers and final traces. The first row (a) is based on the case notion in Fig. 9, representing the same example we have just analyzed. Case notions (b) and (c) are variations of the case notion (a). In (b), the *EKKO* class has been promoted to be an identifying class. This provokes the generation of an additional case identifier, since objects $\{c1, c2\}$ cannot coexist in the case identifier anymore. In (c), also the *EKPO* class has been transformed into an identifying class. This triggers the creation of another case identifier, since the objects $\{d1, d2, d3, d4\}$ cannot belong to the same case identifier either. These examples show the impact of converging and identifying classes in the output of the log building process.

These definitions make it possible to create specialized logs that capture behavior from different perspectives. If all the possible case notions for a data model are generated, automated analysis techniques could be applied to each of the resulting logs, relieving users from tedious analysis tasks and enabling process mining on a large scale. However, the combinatorial explosion problem makes it practically impossible to explore all the case notions for large and complex data models. Even if the search space could be reduced to discard irrelevant case notions, the remaining number would be too high in order for humans to interpret the insights for each of the resulting event logs. This means that we must focus our efforts on the most interesting perspectives to obtain insights without being overwhelmed by excessive amounts of information. The following section proposes a set of metrics to assess the interestingness of a case notion, based on measurable quality features of the resulting event log.

5 Log quality: is my log interesting?

The log quality problem concerns the identification of characteristics that make event logs interesting to be analyzed. This problem is not new to the field. Some authors have studied how the choices made during the log building process can affect the log quality [4] and have developed procedures to minimize the negative impact. Other authors have tried to define metrics to assess different log properties from the structural point of view [5]. In this work, we aim at assessing the quality of an event log in an automated way. For that purpose, we adopt some metrics from [5] that will give us an idea of the structural and data properties that a log should possess in order to be an interesting candidate. In the scope of our meta-model and the logs we are able to build, we need to adapt these concepts to be able to compute them based on our input data, an OpenSLEX file. Considering a valid connected meta-model CMM, a case notion CN, a set of case identifiers CI, and a log l, we adapt the following three metrics to match the structure of our meta-model:

Support (SP) (Eq. 1): number of traces present in an event log:

$$SP(l) = |dom(l)| = |CI| \tag{1}$$

Level of detail (LoD) (Eq. 2): average number of unique activities per trace:

$$LoD(l) = \frac{\sum_{ci \in CI} \left| \bigcup_{ai \in l(ci)} activityOfAI(ai) \right|}{SP(l)} = \frac{\sum_{ci \in CI} \left| \bigcup_{o \in ci} Act_o \right|}{|CI|} \tag{2}$$

Average number of events (AE) (Eq. 3): average number of events per trace:

$$AE(l) = \frac{\sum_{ci \in CI} \left| \bigcup_{ai \in l(ci)} E_{ai} \right|}{SP(l)} = \frac{\sum_{ci \in CI} \left| \bigcup_{o \in ci} EvO_o \right|}{|CI|} \tag{3}$$

When analyzing processes, intuitively, it is preferable to have event logs with as many cases as possible, i.e., higher support (Eq. 1), but not too many activities per case, i.e., reasonable level of detail (Eq. 2). The reason for this is that the complexity of the resulting model, and therefore its interpretation, is closely related to the amount of activities it needs to represent. However, too few activities result in very simple models that do not capture any interesting patterns we want to observe. Also, we try to avoid cases with extremely long sequences of events, i.e., large average number of events per trace (Eq. 3), because of the difficulty to interpret the models obtained when trying to depict the behavior. However, too short sequences of events will be meaningless if they represent incomplete cases.

Table 3 Default parameters used to configure the scoring function for case notions

Metric	Parameter	Value	Description
Support	SP_{mode}	–	Mode of the beta pdf used to score the *support* (number of cases). Default is null, since we try to maximize *sp*
	SP_{max}	∞	Highest value of the desired range used to score the *support* value
	SP_{min}	0	Highest value of the desired range used to score the support value
Level of detail	LoD_{mode}	4	Mode of the beta pdf used to score the *lod* (level of detail) value
	LoD_{max}	10	Highest value of the desired range used to score the *lod* value
	LoD_{min}	2	Lowest value of the desired range used to score the *lod* value
Average number of events	AE_{mode}	8	Mode of the beta pdf used to score the *ae* (average number of events per trace) value
	AE_{max}	30	Highest value of the desired range used to score the *ae* value
	AE_{min}	4	Lowest value of the desired range used to score the *ae* value
Global score	w_{sp}	0.33	Weight of the *support* score on the final global score
	w_{lod}	0.33	Weight of the *lod* score on the final global score
	w_{ae}	0.33	Weight of the *ae* score on the final global score

Therefore, while we would like to maximize the support value (1), i.e., give priority to logs with a higher number of traces, we cannot say the same for the level of detail (2) and average number of events per case (3). These last two metrics will find their optimality within a range of acceptable values, which will depend on the domain of the process and taste of the user, among other factors. Given the differences between the pursued optimal values for each of the metrics, the need for a scoring function becomes evident. It is required to be able to effectively compare log metrics. A candidate is the beta distribution. The reason for our choice is that the beta distribution has two parameters to control its shape, and this gives us additional freedom to customize the scoring function. Choosing the right values for the parameters of the distribution can seem daunting at first. However, it is possible to estimate their value based on more intuitive parameters that describe the shape of the resulting distribution, e.g., mode and inflection points of the curve. In practice, the technique yields satisfactory results using the default parameters (Table 3), and only the advanced user might need to modify them. Note that the choice of the scoring function is not restricted by the approach and could be replaced by any distribution more appropriate to the setting of application.

The beta distribution is defined on the interval [0, 1] and has two shape parameters, α and β. The values of these two parameters determine the shape of the curve, its mean, mode, variance, etc. Also, the skewness of the distribution can be shaped choosing the right

combination of parameters (see Fig. 11). This allows one to define a range of values for which the *probability density function* (PDF) of the beta distribution (Eq. 4) will return higher scores as they approximate to the mode.

$$Beta_{PDF}(x;\alpha,\beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}, \quad \text{where } B(\alpha,\beta) \text{ is the Euler beta function.} \tag{4}$$

The input values will get a lower score as they get farther from the mode. One advantage of this distribution is that it is possible to define a mode value different from the mean, i.e., to shape an asymmetric distribution. Figure 11 shows examples of beta distributions for different values of α and β.

The parameters α and β can be estimated based on the mode and approximate inflection points of the desired PDF [10]. We show an example considering only the mode. If we are interested on event logs with a *level of detail* close to 7, we need to estimate the values of α and β to obtain a PDF with mode 7. First we scale the value. If the minimum and maximum values for LoD are 1 and 20, then the scaled mode is 0.32. Assuming that we are after a unimodal PDF and $\alpha, \beta > 1$, we use Eq. (5) to compute the mode:

$$mode = \frac{\alpha - 1}{\alpha + \beta - 2} \quad \text{for } \alpha, \beta > 1. \tag{5}$$

Given the desired $mode$, we can fix the value of one of the shape parameters and estimate the other one using Eq. (5):

$$est(mode) = \begin{cases} \beta = 2, \alpha = \frac{1}{1-mode}, & \text{if } mode < 0.5 \Rightarrow \text{positively skewed} \\ \alpha = 2, \beta = \frac{1-4mode}{mode}, & \text{if } mode > 0.5 \Rightarrow \text{negatively skewed} \\ \alpha, \beta = 2, & \text{if } mode = 0.5 \Rightarrow \text{symmetric.} \end{cases} \tag{6}$$

Therefore, for the mode 0.32, the PDF is positively skewed. Using Eq. (6), we evaluate $est(0.32)$ to obtain the values $\beta = 2$ and $\alpha = 1/(1-0.32) = 1.47$. The resulting PDF can be observed in Fig. 11 (dotted curve). This is a basic yet effective method to set the shape parameters of the beta function using domain knowledge, i.e., the optimal value that we desire to score higher. Once the parameters α and β have been selected, we can compute the scores of the previous log metrics. To do so, we provide a score function:

$$score(f, x_i, X, , fi) = Beta_{PDF}(scaled(f, x_i, X); \alpha, \beta) \tag{7}$$

Here, f is a function to compute the metric to be scored (e.g., SP, LoD, or AE), x_i is the input of function f (e.g., a log l), X is the set of elements with respect to which we must scale the value of $f(x_i)$ (e.g., a set of logs L), α and β are the parameters of the beta probability distribution function, and $scaled(f, x_i, X)$ is a rescaling function such that:

$$scaled(f, x_i, X) = \frac{f(x_i) - \min_{x_j \in X}\{f(x_j)\}}{\max_{x_j \in X}\{f(x_j)\} - \min_{x_j \in X}\{f(x_j)\}}. \tag{8}$$

With the *score* function in Eq. (7), first we perform feature scaling (Eq. 8). Next, we apply the beta distribution function (Eq. 4) with the corresponding α and β parameters. With respect to the **support** of the log, the score will be the result of scaling the support feature ($SP(l)$) with respect to the set of possible logs L and applying the beta probability distribution function. As the purpose, in this case, is to give a higher score to higher support values, we will set the parameters α_{SP} and β_{SP} such that the probability distribution function resembles an ascending line (e.g., $\alpha = 2$ and $\beta = 1$ in Fig. 11):

$$s_{sp}(l, L) = score(SP, l, L, {}_{SP}, fi_{SP}). \tag{9}$$

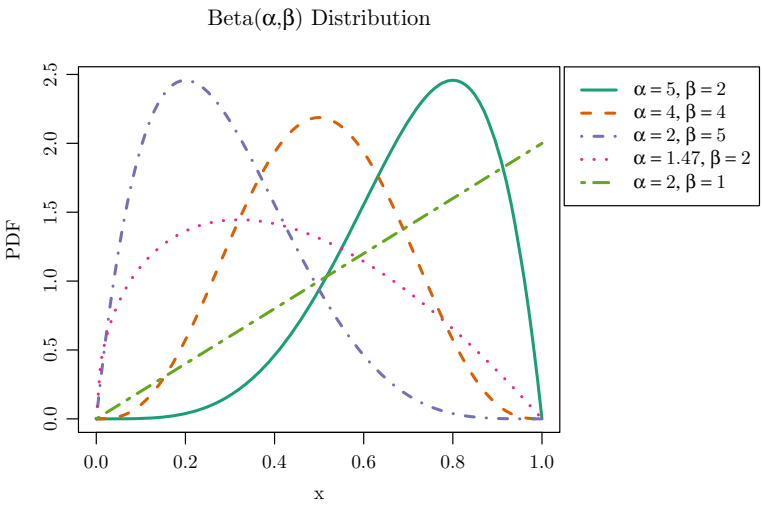


Fig. 11 Sample of beta distribution curves for different values of the α and β parameters

To score the **level of detail**, we let the parameters α_{LoD} and β_{LoD} to be tuned according to the preference of the user:

$$s_{lod}(l, L) = score(LoD, l, L, {}_{LoD}, fi_{LoD}). \tag{10}$$

The score of the **average number of events** per case is computed in the same way, using the appropriate values for the parameters α_{AE} and β_{AE}:

$$s_{ae}(l, L) = score(AE, l, L, {}_{AE}, fi_{AE}) \tag{11}$$

The interestingness of a log l with respect to all the logs L can be defined by the combination of the score values for each of the previous metrics. In order to combine the scores for each log metric, a global scoring function $gsf \in L \times \mathcal{P}(L) \Rightarrow \mathbb{R}$ can be used, which takes a log l and a set of logs L and returns the score of l with respect to L. The approach does not depend on the choice of this function, and it can be replaced by any custom one. For the purpose of demonstrating the feasibility of this approach, we define the global scoring (or "log interestingness") function as the weighted average of the three previous scores. The weights (w_{sp}, w_{sp}, w_{sp}) and the parameters of the beta distribution (α_{SP}, β_{SP}, α_{LoD}, β_{LoD}, α_{AE}, β_{AE}) can be adjusted by the user to balance the features according to their interest.

$$gsf(l, L) = w_{sp} \cdot s_{sp}(l, L) + w_{lod} \cdot s_{lod}(l, L) + w_{ae} \cdot s_{ae}(l, L). \tag{12}$$

It must be noted that it is not necessary to set custom values for the parameters of our scoring function every time that we want to tackle a different dataset. In most of the cases, it will be enough to apply the technique using the default parameters in Table 3.

The "log interestingness" scoring function (Eq. 12) proposed in this section aims at giving an indication of how likely it is that a log will be of interest, with respect to the other candidates, given a set of parameters. Table 4 shows the top 8 discovered case notions of the sample SAP dataset, according to the computed score. We see that the tables involved in the purchase requisition process represent a relevant case notion candidate for this specific dataset. The main contribution until now is not the specific scoring function, but the framework that enables the assessment and its configuration.

Table 4 Top 8 discovered case notions, sorted by score with parameters ($\alpha_{SP} = 2$, $\beta_{SP} = 1$, $\alpha_{LoD} = 4.16$, $\beta_{LoD} = 1$, $\alpha_{AE} = 1.28$, $\beta_{AE} = 1.53$, $w_{sp} = 0.3$, $w_{lod} = 0.3$, and $w_{ae} = 0.3$)

	Root	Tables	SP′	LoD′	AE′	Score
1	EBAN	EKPO, EINE, EBAN, EKKO, LFA1	0.54	1.00	0.60	1.90
2	EINE	EKPO, EINE, EBAN, EKKO, LFA1	0.70	0.95	0.65	1.79
3	EBAN	EKPO, EINE, EBAN, MARA	0.28	1	0.69	1.73
4	EKPO	EKPO, EINE, EBAN, EKKO, LFA1	0.80	0.87	0.63	1.60
5	EKKO	EKPO, EINE, EBAN, EKKO, LFA1	0.55	0.88	0.47	1.53
6	EINE	EKPO, EINE, EBAN, EKKO	0.70	0.85	0.56	1.52
7	EBAN	EKPO, EINE, EBAN, EKKO	0.54	0.87	0.48	1.51
8	EINE	EKPO, EINE, EBAN, MARA	0.45	0.89	0.71	1.44

The α and β parameters have been estimated based on desired min, max, and mode values for the corresponding beta distribution ($LoD_{min} = 2$, $LoD_{max} = 10$, $LoD_{mode} = 4$, $AE_{min} = 4$, $AE_{max} = 30$, and $AE_{mode} = 8$). The values for SP, LoD, and AE have been scaled

The metrics that we chose (support, level of detail, and average number of events per trace) represent a baseline set of key indicators to compute an interestingness score per event log. It can be the case that, in certain scenarios, assessing the potential interestingness of an event log requires the use of different metrics, e.g., the variety of trace types, some structural property of a discovered process model, or the fitness score with respect to a normative model. The framework proposed in this work allows the user to define any custom metric and/or global score to be computed for each candidate event log.

However, this framework still requires a log to be generated in order to be subjected to evaluation. Taking into account that the final goal is to automatically assess log interestingness at a large scale, we need better ways to score case notions before the corresponding logs are built. The following section explores this idea, proposing a method to predict log interestingness based on our baseline metrics and score function.

6 Predicting log interestingness

If an event log is completely created from an extracted dataset, then it is straightforward to assess the actual interestingness. However, as explained before, for large databases, it is infeasible to compute all candidates. In order to mitigate this problem and save computation time, we aim at approximating the value of the metrics considered in Sect. 5 for a certain case notion, before the log is computed. To do so, it is important to define bounds for the log metrics, given a certain case notion. The purpose is to restrict the range of uncertainty and improve the prediction accuracy. In fact, at the end of this section, the bounds will be used to define a custom predictor for each of the log metrics.

As we mentioned in the previous section, the framework is extensible, allowing the user to define additional metrics when necessary. Any additional metric used to assess log interestingness will need to be taken into account in the global scoring function (Eq. 12). Also, in order to take advantage of the log interestingness prediction method, an approximation function must be provided for any additional metric that the user defines. The approximation function for a certain metric must be able to compute an approximated value for a metric, given a certain case notion and the extracted data, without the need to compute the corre-

sponding event log. As an example, in this section, we present upper and lower bounds of the baseline metrics used in our global scoring function.

First, we try to set bounds to the support of a log. From Eq. (1), we see that the support of a log is equal to the domain of the mapping, i.e., the amount of case identifiers of the log. Definition 4 shows that the amount of case identifiers depends on the combinations of objects belonging to the identifying classes of the case notion (*IDC*). Given that every case identifier must contain one object of the root class, that only one object of the root class is allowed per case identifier, and that the set of case identifiers is a maximal set, we can conclude that the set of case identifiers will contain at least one case identifier per object in the root class:

Bound 1 (Lower bound for the support of a case notion) *Given a valid connected meta-model CMM, a case notion* $CN = (C, root, children, CONV, IDC, rsEdge)$*, a maximal set of case identifiers CI, and the corresponding log l we see that* $\forall ci \in CI : \exists o \in ci : classOfObject(o) = root \iff \forall o \in O_{root} : \exists ci \in CI : o \in ci \Rightarrow |CI| \geq |O_{root}|$*. Therefore, we conclude that:* $SP(l) \geq \lfloor SP(CN) \rfloor = |O_{root}|$

For a case identifier to be transformed into an actual trace, at least an event must exist for the root object involved in it. For the sake of simplicity, Bound 1 assumes that at least one event exists for every object in the root class. This has been taken into account in the implementation, considering only objects of the root class that contain at least one event.

Each of the case identifiers is a combination of objects. Also, exactly one object of the root class and no more than one object of each identifying class (classes in *IDC*) can exist per case identifier. This leads to the following upper bound for support:

Bound 2 (Upper bound for the support of a case notion) *Given a valid connected meta-model CMM, a case notion* $CN = (C, root, children, CONV, IDC, rsEdge)$*, a maximal set of case identifiers CI, and the corresponding log l, we define a maximal set* CI' *for which the following properties hold:*

(a) $\forall ci \in CI' : \forall o \in ci : classOfObject(o) \in IDC \Rightarrow \exists o' \in ci : classOfObject(o) = classOfObject(o') \iff o = o'$*, i.e., only one object per class belongs to the case identifier,*

(b) $\forall ci \in CI' : \exists o \in ci : classOfObject(o) = root$*, i.e., one object of the root class must always belong to the case identifier.*

This implies that CI' *contains all the possible combinations of one or zero objects of each class in IDC, except for the root class, that must always be represented by an object in the case identifier. That means that* $|CI'| = |O_{root}| \cdot \prod_{c \in \{C \setminus root\}} (|O_c| + 1)$*. Given that* CI' *is less restrictive than CI, we know that* $CI' \supseteq CI \Rightarrow |CI'| \geq |CI|$*. Therefore,* $SP(l) \leq \lceil SP(CN) \rceil = |O_{root}| \cdot \prod_{c \in \{C \setminus root\}} (|O_c| + 1)$

Following the same logic to set a lower bound for support, we know that all the objects that belong to the root class will be involved in at least one case identifier. However, the number of traces is still unknown if the log has not been built and we can only consider it as the maximum possible, i.e., the upper bound of the support. Therefore, a lower bound for the level of detail will be given by the sum of the unique activities per object of the root class divided by the maximum number of case identifiers. If we consider that the additional case identifiers (beyond the number of objects of the root class) will, at least, add a unique number of activities equal to the minimum number of activities per object of the root class, we can get a better lower bound as described below:

Bound 3 (Lower bound for the LoD of a case notion) *Given a valid connected meta-model CMM, a case notion* $CN = (C, root, children, CONV, IDC, rsEdge)$*, a maximal set of case identifiers CI, and the corresponding log l, we see that* $\forall ci \in CI : \exists o \in ci : classOfObject(o) = root \iff \forall o \in O_{root} : \exists ci \in CI : o \in ci \Rightarrow \forall ci \in CI : \bigcup_{o \in ci} Act_o \supseteq \bigcup_{o \in (ci \cap O_{root})} Act_o$*. Additionally, we know that* $\sum_{ci \in CI} |\bigcup_{o \in (ci \cap O_{root})} Act_o| \geq (\sum_{o \in O_{root}} |Act_o|) + (|CI| - |O_{root}|) \cdot \min_{o \in O_{root}} \{|Act_o|\}$*. Therefore,*

$$LoD(l) \geq \lfloor LoD(CN) \rfloor = \frac{\left(\sum_{o \in O_{root}} |Act_o|\right) + \left(\lceil SP(CN) \rceil - |O_{root}|\right) \cdot \min_{o \in O_{root}} \{|Act_o|\}}{\lceil SP(CN) \rceil}.$$

A lower bound for LoD is given by the lower bound of the sum of the unique activities per case, divided by the upper bound on the number of cases. We know that, at least, one case will exist per object belonging to the root class. That is why the sum of the unique activities per objects of root is added on the top part of the formula. Also, because these objects could be involved in more than one case, to a maximum of $\lceil SP(CN) \rceil$ cases, we add the minimum number of unique activities they could have and multiply it by the maximum number of additional case identifiers. This will always be a lower bound given that the number of activities we add at the upper part for the additional case identifiers will always be equal or lower than the average. Not adding these extra case identifiers would still result in a lower bound, but an extremely low one since the divisor is usually an overestimation for the number of possible case identifiers.

With respect to the upper bound for the level of detail, we need to consider the most extreme situation. This is caused by a case identifier that contains one object per identifying class and one or more objects per converging class, such that, for each object, the events related to them represent all the possible activities. For this case identifier, the number of unique activities will be the sum of the number of unique activities per class involved. However, there is a way to restrict this bound. If we count the number of unique activities for the events of each object, and find the maximum per class, the upper bound will be given by the sum of the maximum number of unique activities per object for all the identifying classes, plus the total of unique activities per converting class involved in the case notion:

Bound 4 (Upper bound for the LoD of a case notion) *Given a valid connected meta-model CMM, a case notion* $CN = (C, root, children, CONV, IDC, rsEdge)$*, a maximal set of case identifiers CI, and the corresponding log l, we know that,* $\forall c \in C : \forall o \in O_c : |Act_o| \leq \max_{o' \in O_c} \{|Act_{o'}|\}$*. This implies that,* $\forall ci \in CI : |\bigcup_{o \in ci} Act_o| \leq \sum_{c \in IDC} \max_{o \in O_c} \{|Act_o|\} + \sum_{c\ inCONV} |ActC_c|$*. Therefore,*

$$\begin{aligned} LoD(l) \leq \lceil LoD(CN) \rceil &= \frac{|CI| \cdot \left(\sum_{c \in IDC} \max_{o \in O_c} \{|Act_o|\} + \sum_{c\ inCONV} |ActC_c|\right)}{|CI|} \\ &= \sum_{c \in IDC} \max_{o \in O_c} \{|Act_o|\} + \sum_{c\ inCONV} |ActC_c|. \end{aligned}$$

The same reasoning used to obtain a lower bound for the level of detail can be applied in the case of the average number of events per trace. Only that, in this case, instead of counting the number of unique activities, we count the number of events per object:

Bound 5 (Lower bound for the AE of a case notion) *Given a valid connected meta-model CMM, a case notion* $CN = (C, root, children, CONV, IDC, rsEdge)$*, a maximal set of case identifiers CI, and the corresponding log l, we see that* $\forall ci \in CI : \exists o \in ci : classOfObject(o) = root \iff \forall o \in O_{root} : \exists ci \in CI : o \in$

$ci \Rightarrow \forall ci \in CI : \bigcup_{o \in ci} EvO_o \supseteq \bigcup_{o \in (ci \cap O_{root})} EvO_o$. *Additionally, we know that* $\sum_{ci \in CI} |\bigcup_{o \in (ci \cap O_{root})} EvO_o| \geq (\sum_{o \in O_{root}} |EvO_o|) + (|CI| - |O_{root}|) \cdot \min_{o \in O_{root}} \{|EvO_o|\}$. *Therefore,*

$$AE(l) \geq \lfloor AE(CN) \rfloor = \frac{\left(\sum_{o \in O_{root}} |EvO_o|\right) + \left(\lceil SP(CN) \rceil - |O_{root}|\right) \cdot \min_{o \in O_{root}} \{|EvO_o|\}}{\lceil SP(CN) \rceil}.$$

A lower bound for AE is given by the lower bound of the sum of the events per case, divided by the upper bound on the number of cases. At least one case will exist per object of the root class. Therefore, we consider the sum of the number of events per object. These objects could be involved in more than one case, to a maximum of $\lceil SP(CN) \rceil$ cases. So, we add the minimum number of events they could have, multiplied by the maximum number of additional case identifiers. This is a lower bound given that the number of events added at the upper part for the additional case identifiers is equal or lower than the average. Not adding these extra case identifiers would still result in a lower bound, but an extremely low one since the divisor is usually an overestimation on the number of possible case identifiers.

To define an upper bound for AE, we use an approach similar to the one used to compute an upper bound for LoD. We need to consider the most extreme case, the case in which the maximum number of events per object (for the identifying classes) could be included in the final trace. However, if the case notion has converging classes, the most extreme case is the one in which all the objects of such classes are contained in the case identifier, therefore all the events belonging to the converging classes would be inserted in the trace:

Bound 6 (Upper bound for the AE of a case notion) *Given a valid connected meta-model CMM, a case notion* $CN = (C, root, children, CONV, IDC, rsEdge)$*, a maximal set of case identifiers CI, and the corresponding log l, we know that,* $\forall c \in C : \forall o \in O_c : |EvO_o| \leq \max_{o' \in O_c} \{|EvO_{o'}|\}$. *This implies that,* $\forall ci \in CI : |\bigcup_{o \in ci} EvO_o| \leq \sum_{c \in IDC} \max_{o' \in O_c} \{|EvO_{o'}|\} + \sum_{c \in CONV} |EvC_c|$. *Therefore,*

$$\begin{aligned} AE(l) \leq \lceil AE(CN) \rceil &= \frac{|CI| \cdot \left(\sum_{c \in IDC} \max_{o' \in O_c} \{|EvO_{o'}|\} + \sum_{c \in CONV} |EvC_c|\right)}{|CI|} \\ &= \sum_{c \in IDC} \max_{o' \in O_c} \{|EvO_{o'}|\} + \sum_{c \in CONV} |EvC_c|. \end{aligned}$$

These bounds define the limits for our prediction. For each metric ($SP(l)$, $LoD(l)$ and $AE(l)$), either the lower or upper bound could be a prediction. However, a better heuristic can be designed. We defined equations to predict the values as the weighted average of the corresponding bounds (Eqs. 13, 14, 15). Given a valid connected meta-model *CMM* and a case notion *CN*, our prediction for each metric is given by the following heuristics:

$$\widehat{SP}(CN) = w_{lbsp} \cdot \lfloor SP(CN) \rfloor + w_{ubsp} \cdot \lceil SP(CN) \rceil \tag{13}$$

$$\widehat{LoD}(CN) = w_{lblod} \cdot \lfloor LoD(CN) \rfloor + w_{ublod} \cdot \lceil LoD(CN) \rceil \tag{14}$$

$$\widehat{AE}(CN) = w_{lbae} \cdot \lfloor AE(CN) \rfloor + w_{ubae} \cdot \lceil AE(CN) \rceil . \tag{15}$$

From these equations we see that, in order to calculate the heuristics for each metric, we need to collect some features. These features (Table 5) can be easily computed once for each class $c \in CL$ in the dataset and be reused for every case notion *CN* we want to assess.

Finally, in order to score the predicted values of each metric, the scoring function previously used (Eq. 7) must be individually applied. The input parameters are two: a case notion

Table 5 Features used to compute upper and lower bounds for each log metric

	Feature	Description
1	$MaxEvO_c = \max_{o \in O_c}\{\lvert EvO_o \rvert\}$	Maximum # of events per object of a class c
2	$MaxAct_c = \max_{o \in O_c}\{\lvert Act_o \rvert\}$	Maximum # of activities per object of a class c
3	$MinEvO_c = \min_{o \in O_c}\{\lvert EvO_o \rvert\}$	Minimum # of events per object of a class c
4	$MinAct_c = \min_{o \in O_c}\{\lvert Act_o \rvert\}$	Minimum # of activities per object of a class c
5	$\lvert EvC_c \rvert$	# of events per class c
6	$\lvert ActC_c \rvert$	# of unique activities per class c
7	$SumEvO_c = \sum_{o \in O_c} \lvert EvO_o \rvert$	Total # of events per object for a class c
8	$SumAct_c = \sum_{o \in O_c} \lvert Act_o \rvert$	Total # of unique activities per object for a class c
9	$\lvert O_c \rvert$	# of objects of a class c

CN, and a set of case notions CNS to compare to. Equations (16), (17), and (18) provide the scores for the predicted metrics given a case notion CN and a set of case notions CNS.

$$\widehat{s_{sp}}(CN, CNS) = score(\widehat{SP}, CN, CNS, {}_{SP}, fi_{SP}) \tag{16}$$

$$\widehat{s_{lod}}(CN, CNS) = score(\widehat{LoD}, CN, CNS, {}_{LoD}, fi_{LoD}) \tag{17}$$

$$\widehat{s_{ae}}(CN, CNS) = score(\widehat{AE}, CN, CNS, {}_{AE}, fi_{AE}). \tag{18}$$

Next, a global scoring function is defined to combine the three of them. We will call this function the *predicted global scoring function*, $pgsf \in CNS \times \mathcal{P}(CNS) \rightarrow \mathbb{R}$ and it is the weighted average of the scores of each of the three predicted values:

$$pgsf(CN, CNS) = w_{sp} \cdot \widehat{s_{sp}}(CN, CNS) + w_{lod} \cdot \widehat{s_{lod}}(CN, CNS) + w_{ae} \cdot \widehat{s_{ae}}(CN, CNS) \tag{19}$$

This function represents our *custom predictor* for log interestingness. The accuracy of the predictor will be evaluated in Sect. 8, where it will be compared to alternative techniques.

7 Implementation

All the techniques proposed in this paper are part of the Event Data Discovery Tools package (*eddytools*[9]). This tool assists the user at every step from data extraction to event log building. The eddytools Python package provides six commands that cover the main steps (some of them out of the scope of this paper) of the data extraction and preparation phase. These steps and their purpose are described below:

1. *Data exploration* to get a feeling of the size and dimension of the data. Also, to look for any high-level structure that can be extracted from it.
2. *Data schema discovery* to discover the data relations (primary, unique, and foreign keys) in order to be able to correlate data objects in future steps.
3. *Data extraction* to obtain an off-line copy of the data that we can transform into a format suitable for analysis. Also, this allows us to complete the data once a schema has been discovered.
4. *Event data discovery* event data might be implicitly stored within or across different tables in the dataset. We need to discover the events and make them explicit.

[9] https://github.com/edugonza/eddytools.

Table 6 Details about the SAP dataset used during the evaluation

Tables	87	Case notions	10,622
Objects	7,339,985	Non-empty logs	5180
Versions	7,340,650	Total log building time	13 h 57 m
Events	26,106	Average log building time	4.7 s
		Features computation time	2 m

5. *Case notion discovery* defining a case notion allows us to correlate events into traces. Many alternative case notions can be defined depending on the perspective we want to take.
6. *Event log building* from the discovered events and a case notion we can build an event log. Many case notions can be defined, and the corresponding event logs can be constructed in order to analyze different coexisting processes, or the same process from different perspectives.

We claim that these steps can be executed in a semiautomatic way, given that they allow for a certain customization depending on the characteristics of the environment to analyze. In [11] (Chapter 8), we provide additional details on the use of the tool in a real-life case study.

8 Evaluation

So far, we proposed a set of metrics to assess the interestingness of an event log once it has been constructed. Also, we provided predictors for these metrics based on (a) the characteristics of the case notion being considered and (b) features of the dataset under study. The aim of this section is twofold. (1) To find out how good our predictors are at estimating the value of each log characteristic. (2) To evaluate the quality of the rankings of case notions, based on their potential interestingness according to certain log metrics, using our custom predictor and compare them to existing learning to rank algorithms.

The evaluation was carried out on a SAP sample dataset (Table 6). It contains the data model, objects, object versions, and events of 87 SAP tables. The following steps were executed using the open source software package *eddytools*. First, a set of candidate case notions was generated. To do so, each one of the tables in the data model was taken as the root node of a potential case notion. Next, for each of them, all the possible simple paths following outgoing arcs were computed, yielding a result of 10,622 case notion candidates. For each of the candidates, the corresponding event log was generated and the metrics presented in Sect. 5 were computed. This set of logs and metrics represent the ground truth. Given that we want to predict the metrics in the ground truth set, we need to measure the features that our predictors require. The following section describes these features.

8.1 Features for log quality prediction

Section 6 presented our predictors for each of the log characteristics. These predictors estimate the values of the *support* (*SP*, Eq. 13), *level of detail* (*LoD*, Eq. 14), and *average number of events per trace* (*AE*, Eq. 15) of a log, given the corresponding case notion and a set of features. This subsection describes the features used during the evaluation which are (a) the

Table 7 Features used to predict log interestingness

	Feature	Description
1	$\lfloor SP(CN) \rfloor$	Lower bound for the support
2	$\lceil SP(CN) \rceil$	Upper bound for the support
3	$\lfloor LoD(CN) \rfloor$	Lower bound for the level of detail
4	$\lceil LoD(CN) \rceil$	Upper bound for the level of detail
5	$\lfloor AE(CN) \rfloor$	Lower bound for average number of events per trace
6	$\lceil AE(CN) \rceil$	Upper bound for average number of events per trace
7	$\lvert C \rvert$	Number of classes in the case notion
8	$\lvert E(CN) \rvert$	Total number of events of all the classes in the case notion
9	$IR(CN)$	Average number of events per object

lower and upper bounds of each log property as listed in Sect. 6 and (b) additional features used to improve the accuracy of the regressors we will compare to.

Given a valid connected meta-model *CMM* (i.e., a dataset stored in the OpenSLEX format containing events, objects, versions, and a data model) and a specific case notion *CN*, we can measure the features enumerated in Table 7. The log associated with such case notion *does not* need to be built in order to compute these features. Actually, many of the features are the result of an aggregation function over a class property. Once the class properties have been computed, the complexity of calculating these case notion metrics is linear with respect to the number of classes involved.

8.2 Evaluation of predictors' accuracy

In Sect. 6, upper and lower bounds were given for each log property given a case notion (CN). These bounds have been used to estimate the value of such log properties by means of three predictors (one per log property), before the log is actually built. Now it is time to evaluate the accuracy of these predictors. To do so, we compared the predicted value for each log property (*SP*, *LoD*, and *AE*) with the actual values in the ground truth dataset. This was done for the predictors for each log property as defined in Sect. 6 (Eqs. 13, 14, 15). The combination of the scores of the three individual predictors (Eqs. 16, 17, 18) in a single scoring function of log interestingness (Eq. 19) is what we call our *Custom Predictor (CP)*. Additionally, we compared the accuracy of the individual predictors to three different regressors: (a) Multiple Linear Regressor (MLP), (b) Quantile Regressor (QR) [12], and (c) Neural Network Regressor (NN). Each of them where trained and tested using the features in Table 7. A fivefold cross-validation was performed in order to determine the accuracy of the predictors (our predictors, MLP, QR, and NN). To avoid underestimation of the prediction error, empty logs where filtered out of the dataset, using only 5180 case notions from the original 10,622.

Figure 12 shows the mean absolute error (MAE) measured per normalized property for each predictor. We see that our predictors do not perform really well, presenting an average error of around 1.0 when predicting LoD or AE and around 1.1 when predicting SP. In comparison, the regressors perform better, in particular the Quantile regressor with an average error of around 0.8 for SP and LoD, and around 0.9 for AE. This figure, however, could be misleading, given that the MAE is computed on all the predictions, regardless of the existence

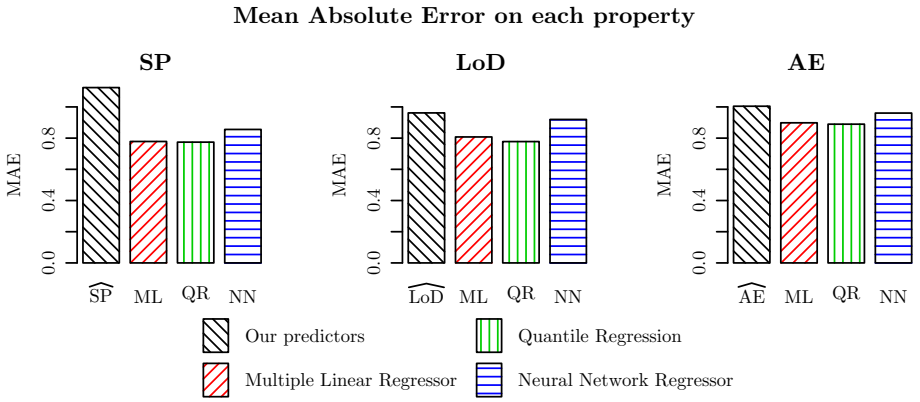


Fig. 12 Comparison of mean absolute error for the predictors on the three normalized log properties

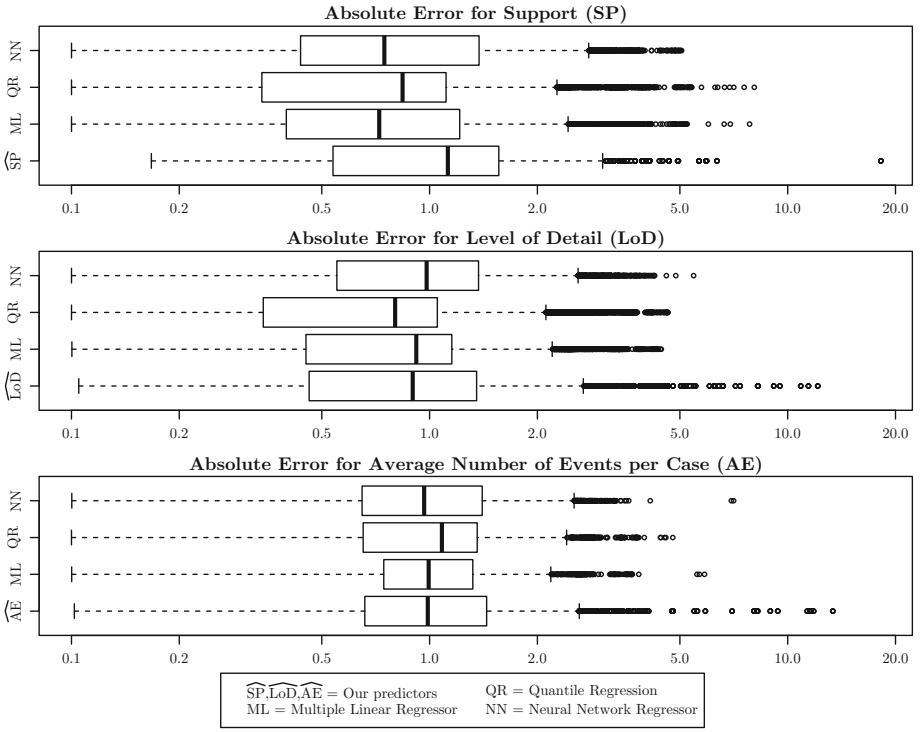


Fig. 13 Comparison of absolute error for the three normalized log properties per predictor. The scale is logarithmic

of outliers. To get a better idea of the influence of extremely bad predictions on the overall performance, we include Fig. 13, which shows box-plots for each log property per predictor. It is important to notice that a logarithmic scale has been used, in order to plot extreme outliers and still be able to visualize the details of each box.

We see that our predictors ($\widehat{SP}$, $\widehat{LoD}$, and $\widehat{AE}$) are the worst performing ones, especially when it comes to SP. Also, they are the ones presenting the most extreme outliers for the

three log properties. Quantile Regression and Neural Network regressors present the most consistent results, with the least extreme outliers. These results show that there is considerable room for improvement to predict SP, LoD, and AE accurately. This can be achieved, for example, by selecting additional features that have a stronger correlation with the properties we aim to predict. It must be noted that our predictors are unsupervised, i.e., do not need a training set. This represents an advantage with respect to the regressors, since they can generate predictions on the absence of training data. Despite the inaccuracy of our predictors, their usefulness is yet to be determined. The aim of the prediction is to build a ranking of case notions based on their interestingness (Eq. 19). This means that, as long as the relative interestingness is preserved, the ranking can be accurate. The following section will address this issue, using a metric to evaluate the quality of the rankings.

8.3 Evaluation of ranking quality

Until now, we have evaluated the accuracy of our predictors and compared them to other existing regressors. However, the goal of predicting log properties is to assess the interestingness of the log *before* it is built. If we are able to predict the interestingness of the logs for a set of case notions, we can rank them from more to less interesting and provide a recommendation to the user. In this section we evaluate how good the predictors are at ranking case notions according to their interestingness. To do so, we use the metrics on the resulting event logs as the ground truth to elaborate an ideal ranking (Eq. 12). Then, a new ranking is computed using our custom predictor (Eq. 19) and it is compared to the ideal one. This comparison is done by means of the metric *normalized discounted cumulative gain at p* ($nDCG_p$), widely used in the information retrieval field.

$$DCG_p = \sum_{i=1}^{p} \frac{rel\_score_i}{\log_2(i+1)} = rel\_score_1 + \sum_{i=2}^{p} \frac{rel_i}{\log_2(i+1)} \tag{20}$$

$$IDCG_p = \sum_{i=1}^{|REL\_SCORES|} \frac{rel\_score_i}{\log_2(i+1)} \tag{21}$$

$$nDCG_p = \frac{DCG_p}{IDCG_p} \tag{22}$$

The *normalized discounted cumulative gain at p* (Eq. 22) is a metric that assumes the existence of a relevance score for each result, penalizing the rankings in which a relevant result is returned in a lower position. This is done by adding the graded relevance value of each result, that is logarithmically reduced proportional to its position (Eq. 20). Next, the accumulated score is normalized, dividing it by the ideal score in case of a perfect ranking (Eq. 21). This means that the ranking $\langle 3, 1, 2\rangle$ will get a lower score than the ranking $\langle 2, 3, 1\rangle$ for an ideal ranking $\langle 1, 2, 3\rangle$ and a relevance per document of $\langle 3, 3, 1\rangle$.

When it comes to ranking, there is a large variety of *learning to rank* (LTR) algorithms in the information retrieval field [13]. These algorithms are trained on ranked lists of documents and learn the optimal ordering according to a set of features. A fivefold cross-validation was performed on the unfiltered set of case notions (10,622 candidates) comparing the implementation[10] of 10 learning to rank algorithms (MART, RankNet, RankBoost, AdaRank, Coordinate Ascent, LambdaRank, LambdaMART, ListNet, Random Forest, and Linear Regression) with the predictors evaluated in Sect. 8.2 (Quantile Regression, Multiple

[10] https://sourceforge.net/p/lemur/wiki/RankLib/.

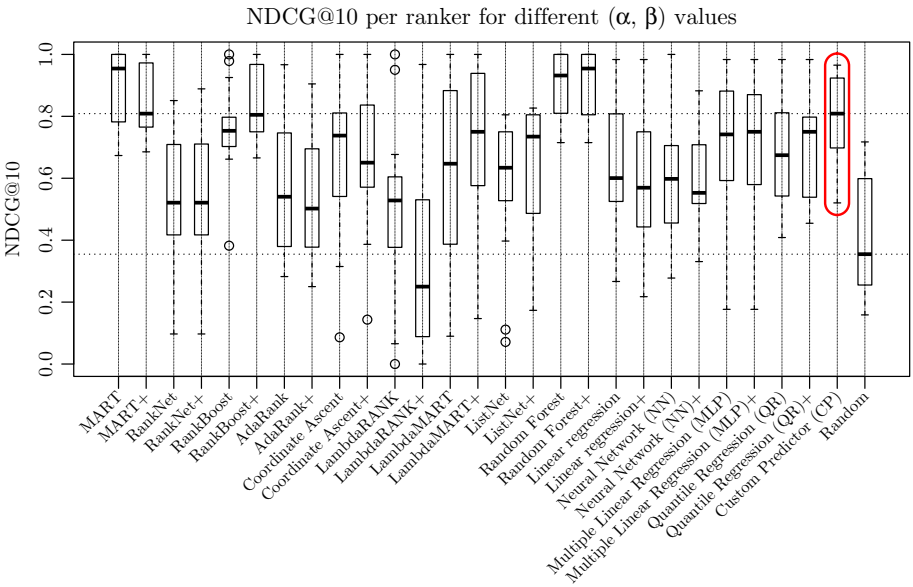


Fig. 14 NDCG@10 per ranker given different combinations of α and β values. The box-plot corresponding to our custom predictor has been highlighted in red (second box-plot from the right)

Linear Regression, Neural Network Regressor, and our custom predictor). Two models were trained for each algorithm: one with the 9 input features in Table 7 and another one with 4 extra features (the estimated value for SP, LoD, AE, i.e., Eqs. 13, 14, and 15). The purpose of adding these extra features is to find out how the estimations made by our predictors affect the predictions of the other algorithms.

Figure 14 shows the result of the evaluation. The 13 algorithms (10 LTR + 3 regressors) were trained on two different sets of features (9 and 13 input features), 3 different combinations of α and β values for the log quality function ($(\alpha, \beta) \in \{(2, 5), (5, 2), (2, 1)\}$), and with equal weight for the three metrics. That makes a total of 78 models ($(10 + 3) \times 2 \times 3$). The NDCG@10 metric was measured for each model and the results were grouped per algorithm and feature set. That resulted in 27 categories ((10 LTR algorithms $\times$ 2 sets of features) + (3 regressors $\times$ 2 sets of features) + our custom predictor) with 15 NDCG@10 values each (5 folds $\times$ the 3 combinations of α and β values). The models trained with 13 features are represented in the figure with the symbol + at the end of their name. Additionally, the NDCG@10 was calculated for a set of random rankings, in order to set a baseline. In the case of our custom predictor, given that it only takes 6 features (the lower and upper bounds for SP, LoD, and AE) and that it does not need training, only three NDCG@10 values were computed, one for each pair of values for the α and β parameters. The horizontal dashed lines drawn in Fig. 14 represent the median of the NDCG@10 for our custom predictor (upper) and the random ordering (lower). Any algorithm whose median is above the upper line will perform better than our custom predictor at least 50% of the time. Any algorithm whose median is above the lower line, will perform better than random at least 50% of the time. Most of the algorithms perform better than random. But only two have the median above the upper line: MART, and Random Forest. When trained with 9 input features, both MART and Random Forest show very similar behavior. However, when considering 13 input features, MART's

median is lower. In the case of Random Forest, using 13 features is better than using 9 in every aspect.

8.4 Discussion

The aim of this evaluation has been twofold. First, to assess the precision of our predictors at estimating the value of each log characteristic. Second, to evaluate the quality of the rankings of case notions, based on their potential "interestingness," using our custom predictor and compare them to LTR algorithms. The results (Figs. 12, 13) show that our predictors are not very good at predicting log characteristics with precision. Other regressors, like Quantile Regression, have shown better results in this aspect. However, when it comes to ranking quality, the precision in the prediction of the log characteristics is of less importance than the relative differences between predictions for several case notions (i.e., it is not so important to predict accurately the log quality of case notions a and b, as long as we can predict that a will be more interesting than b). In fact, the results obtained from the ranking quality evaluation (Fig. 14) show that our custom predictor performs better, on average, than other regressors, even though they showed better prediction accuracy.

We conclude that for the purpose of predicting accurately the value of log characteristics and when training data are available, the use of regressors such as QR is the best option. When it comes to ranking candidates, LTR algorithms such as Random Forest and MART provide better results. However, unlike our custom predictor, all these techniques require the existence of training data to build the models. Therefore, in the absence of such data, the proposed custom predictor provides close-to-optimal results when it comes to rankings and indicative values for the prediction of log characteristics.

9 Related work

The field of process mining is dominated by techniques for process discovery, conformance, and enhancement. Yet event correlation and log building are crucial since they provide the data that other process mining techniques require to find insights. In fact, the choices made during the log building phase can drastically influence the results obtained in further phases of a process mining project. Therefore, it is surprising that there are only a few papers on these topics. Works like the one presented in [4] analyze the choices that users often need to make when building event logs from databases. Also, it proposes a set of guidelines to ensure that these choices do not negatively impact the quality of the resulting event log. It is a good attempt at providing structure and a clear methodology to a phase typically subject to experience and domain knowledge of the user. However, it does not aim at enabling automated log building in any form. It has been shown that extracting event logs from ERP systems like SAP is possible [14]. However, the existing techniques are ad-hoc solutions for ERP and SAP architectures and do not provide a general approach for event log building from databases. Another initiative for event log extraction is the *onprom* project [15–17]. The focus is on event log extraction by means of ontology-based data access (OBDA). OBDA requires to define mappings between the source data source and a final event log structure using ontologies. Then, the *onprom* tools perform an automatic translation from the manually defined mappings to the final event log.

Event log labeling deals with the problem of assigning case identifiers to events from an unlabeled event log. Only a few publications exist that address this challenge. In [18], the

authors transform unlabeled event logs into labeled ones using an Expectation–Maximization technique. In [19], a similar approach is presented, which uses sequence partitioning to discover the case identifiers. Both approaches aim at correlating events that match certain workflow patterns. However, they do not handle complex structures such as loops and parallelism. The approach proposed in [20] makes use of a reference process model and heuristic information about the execution time of the different activities within the process in order to deduct case ids on unlabeled logs. Another approach called Infer Case Id (ICI) is proposed in [21,22]. The ICI approach assumes that the case id is a hidden attribute inside the event log. The benefit of this approach is that it does not require a reference process model or heuristics. The approach tries to identify the hidden case id attribute by measuring control-flow discovery quality dimensions on many possible candidate event logs. Its goal is to select the ones with a higher score in terms of fitness, precision, generalization, and simplicity. The mentioned approaches for event log labeling are clearly related to the problem we try to solve. However, they ignore the database setting, where event correlations are explicitly defined by means of foreign keys. This means that case identifiers do not need to be discovered. Therefore, the challenge of identifying interesting event logs remains open. Only the ICI approach tackles this issue by measuring control-flow metrics to select the best event log. This is similar to our idea of measuring log "interestingness." However, the ICI approach requires to build all the candidate event logs in order to measure such properties. Our approach is able to reduce the computational cost by predicting interestingness properties before the log is built.

Other authors have already considered the idea of evaluating event log characteristics. The metrics proposed in [5] aim at discovering the structural properties of event logs without actually mining the behavior. These metrics have proven to be of great value in order to develop our automated approach. The approach in [23] focuses on event correlation for business processes in the context of Web services. Additionally, it proposes semiautomatic techniques to generate process views with a certain level of "interestingness." Instead of focusing on what is interesting, it discards uninteresting correlations based on the variability of values on the correlating attributes, or on the ratio of process instances per log. The approach is certainly of value in the area of event correlation. On the other hand, it does not provide a framework for automatic case notion discovery. Also, the approach chosen by the authors to deal with the combinatorial explosion problem is search space pruning, which still requires to compute the event logs, but for a smaller set of candidates.

When it comes to computing rankings, in our case rankings of event logs or case notions, we must consider *learning to rank* (LTR) algorithms from the information retrieval field. These algorithms are able to learn an optimal ordering of documents with respect to certain features. Three main categories can be distinguished among them: pointwise, pairwise, and listwise. Pointwise algorithms try to predict the relevance score of each candidate, one by one. These algorithms are able to give a prediction of the score, but do not consider the position of a document in the ranking. Examples of pointwise algorithms are Random Forest [24], Linear regression [25], the predictors evaluated in Sect. 8.2, and any other algorithm that applies regression in general. Pairwise algorithms take pairs of candidates and predict which candidate ranks higher. In this case, the relative position of documents is taken into account. Examples of pairwise algorithms are MART [26], RankNet [27], RankBoost [28], and LambdaRANK [29]. Listwise algorithms take lists of candidates and learn to optimize the order. A disadvantage of this type of approach is the difficulty to obtain training sets of full ranked lists of candidates. Examples of listwise algorithms are AdaRank [30], Coordinate Ascent [31], LambdaMART [32], and ListNet [26].

As a summary, event correlation, log building, and process view "interestingness" are known topics in the field. Despite the attempts of authors, none of the approaches succeeded at

reaching a satisfactory level of automation. Also, none of them proposes a way to recommend process views to the user, neither to rank them by interests.

10 Conclusion

Applying process mining in environments with complex database schemas and large amounts of data becomes a laborious task, specially when we lack the right domain knowledge to drive our decisions. This work attempts to alleviate the problem of event log building by automatically computing case notions and by recommending the interesting ones to the user. By means of a new definition of case notion, events are correlated to construct the traces that form an event log. The properties of these event logs are analyzed to assess their interestingness. Because of the computational cost of building the event logs for a large set of case notion candidates, a set of features was defined based on the characteristics of the case notion and the dataset at hand. Next, a custom predictor estimates the log metrics used to assess the interestingness. This allows one to rank case notions even before their corresponding event logs are built. Finally, an extensive evaluation of the custom predictor was carried out, comparing it to different regressors and to state-of-the-art learning to rank algorithms. We believe that evaluating the approach in comparison to techniques from the information retrieval field has not been considered before in the process mining discipline.

To conclude, this work proposes a framework that covers the log building process from the case notion discovery phase, to the final event log computation, providing the tools to assess its interestingness based on objective metrics. This assessment can be done on the case notion itself before the event log is generated. The result of this assessment is used to provide recommendations to the user.

Our framework presents several limitations, however. The most important one has to do with log interestingness. We are aware that the notion of log "interestingness" proposed in this work is somewhat superficial. Only certain structural properties of the log (*level of detail*, *support*, *average number of events per trace*) are taken into account when evaluating event logs. The current notion of log "interestingness" ignores other important aspects such as the relevance of the log semantics at the business level, how meaningful the activities are with respect to the process, as well as the homogeneity of behavior captured in the event log. Our definition of log "interestingness" is a first attempt at providing an objective score to rank event logs. However, the relation of the proposed "interestingness" metric with respect to a subjective interestingness score provided by users has not been evaluated. A study should be carried out involving real business analysts and domain experts to evaluate the suitability of the metric when applied to different datasets and contexts. Also, this study would be valuable to identify additional measurable aspects that contribute to the notion of log "interestingness" and have not been considered by our definition.

Another limitation has to do with our prediction results. We proposed certain predictors for the event log metrics used to assess log "interestingness." It has been shown that the resulting ranking based on predicted scores resembles, at an acceptable level of accuracy, the ranking based on the actual metrics. However, the individual predictions for each log metric lack accuracy. Relative assumptions can still be made, e.g., log A has higher support than log B. However, accurate predictions would make the technique more robust to outliers, and benefit the overall quality of the log "interestingness" assessment. Finding stricter upper and lower bounds and designing more accurate predictors for each log metric would help to improve the quality of event log "interestingness" rankings and provide better recommendations to

the analyst. This could be combined with sampling techniques that combine predicted scores on candidate case notions with actual scores on computed event logs. This would allow to compute event logs only for a limited number of case notions, while increasing ranking quality introducing some certainty in the scores.

Additionally, processing queries expressed on natural language would be a great addition to the framework, allowing the user to refine the search and insert domain knowledge in the recommendation process. Also, interactive approaches based on feedback provided on example logs would allow to guide the search using domain knowledge.

References

1. van der Aalst WMP, Adriansyah A, de Medeiros AKA, Arcieri F et al (2012) Process mining manifesto. Springer, Berlin, pp 169–194. https://doi.org/10.1007/978-3-642-28108-2_19
2. Watson HJ, Wixom BH (2007) The current state of business intelligence. Computer 40(9):96–99. https://doi.org/10.1109/MC.2007.331
3. Gopalkrishnan V, Li Q, Karlapalem K (1999) Star/snow-flake schema driven object-relational data warehouse design and query processing strategies. In: International conference on data warehousing and knowledge discovery. Springer, Berlin, pp 11–22
4. Jans M, Soffer P (2017) From relational database to event log: decisions with quality impact. In: BPM workshops. Springer, Berlin
5. Gunther C (2009) Process mining in flexible environments. Ph.D. thesis, Eindhoven University of Technology
6. López González, de Murillas E, Reijers HA, van der Aalst WMP (2019) Connecting databases with process mining: a meta model and toolset. Softw Syst Model 18:1209–1247. https://doi.org/10.1007/s10270-018-0664-7
7. XES Working Group (2016) IEEE standard for eXtensible event stream (XES) for achieving interoperability in event logs and event streams. https://doi.org/10.1109/IEEESTD.2016.7740858
8. Lu X, Nagelkerke M, van de Wiel D, Fahland D (2015) Discovering interacting artifacts from ERP systems. IEEE Trans Serv Comput 8(6):861–873
9. Giovinazzo WA (2000) Object-oriented data warehouse design: building a star schema. Prentice Hall PTR, Upper Saddle River
10. Panik MJ (2005) Advanced statistics from an elementary point of view, vol 9. Academic Press, Cambridge
11. González López de Murillas E (2019) Process mining on databases: extracting event data from real-life data sources. Ph.D. thesis, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven
12. Koenker R (2005) Quantile regression. Econometric society monographs. Cambridge University Press, Cambridge
13. Tax N, Bockting S, Hiemstra D (2015) A cross-benchmark comparison of 87 learning to rank methods. Inf Process Manag 51(6):757–772
14. Ingvaldsen JE, Gulla JA (2008) Preprocessing support for large scale process mining of SAP transactions. In: BPM workshops. Springer, pp 30–41
15. Calvanese D, Kalayci TE, Montali M, Santoso A (2017) Obda for log extraction in process mining. In: Reasoning web international summer school. Springer, pp 292–345
16. Calvanese D, Kalayci TE, Montali M, Santoso A (2017) The onprom toolchain for extracting business process logs using ontology-based data access. In: Proceedings of the BPM demo track and BPM dissertation award. CEUR-WS.org

17. Calvanese D, Kalayci TE, Montali M, Tinella S (2017) Ontology-based data access for extracting event logs from legacy data: the onprom tool and methodology. In: International conference on business information systems. Springer, pp 220–236
18. Ferreira DR, Gillblad D (2009) Discovering process models from unlabelled event logs. In: International conference on business process management. Springer, pp 143–158
19. Walicki M, Ferreira DR (2011) Sequence partitioning for process mining with unlabeled event logs. Data Knowl Eng 70(10):821–841
20. Bayomie D, Helal IM, Awad A, Ezat E, ElBastawissi A (2015) Deducing case IDs for unlabeled event logs. In: International conference on business process management. Springer, pp 242–254
21. Andaloussi AA, Burattin A, Weber B (2018) Toward an automated labeling of event log attributes. In: Halpin T, Krogstie J, Nurcan S, Proper E, Schmidt R, Ukor R (eds) Enterprise, business-process and information systems modeling. Springer, Berlin, pp 82–96
22. Burattin A, Vigo R (2011) A framework for semi-automated process instance discovery from decorative attributes. In: IEEE symposium on computational intelligence and data mining (CIDM). IEEE, pp 176–183
23. Motahari-Nezhad HR, Saint-Paul R, Casati F, Benatallah B (2011) Event correlation for process discovery from web service interaction logs. VLDB J 20(3):417–444. https://doi.org/10.1007/s00778-010-0203-9
24. Breiman L (2001) Random forests. Mach Learn 45(1):5–32
25. Ng AY (2004) Feature selection, l1 vs. l2 regularization, and rotational invariance. In: Proceedings of the twenty-first international conference on machine learning. ACM, p 78. https://doi.org/10.1145/1015330.1015435
26. Friedman JH (2001) Greedy function approximation: a gradient boosting machine. Ann Stat 29:1189–1232
27. Burges C, Shaked T, Renshaw E, Lazier A, Deeds M, Hamilton N, Hullender G (2005) Learning to rank using gradient descent. In: Proceedings of the 22nd international conference on machine learning. ACM, pp 89–96
28. Freund Y, Iyer R, Schapire RE, Singer Y (2003) An efficient boosting algorithm for combining preferences. J Mach Learn Res 4:933–969
29. Burges CJ, Ragno R, Le QV (2007) Learning to rank with nonsmooth cost functions. In: Gretton A, Borgwardt KM, Rasch M, Schölkopf B, Smola AJ (eds) Advances in neural information processing systems. MIT Press, Cambridge, pp 193–200
30. Xu J, Li H (2007) Adarank: a boosting algorithm for information retrieval. In: SIGIR. ACM, pp 391–398
31. Metzler D, Croft WB (2007) Linear feature-based models for information retrieval. Inf Retr 10(3):257–274
32. Wu Q, Burges CJ, Svore KM, Gao J (2010) Adapting boosting for information retrieval measures. Inf Retr 13(3):254–270

E. González López de Murillas obtained his Ph.D. at the Eindhoven University of Technology, the Netherlands, in 2019. His research interests include process mining, data extraction and transformation, data querying, automated event log building, and business process management. Currently, he works as a Machine Learning Engineer at Accha.nl, where he develops solutions to optimize manual processes using techniques from different fields such as NLP and information retrieval.

H. A. Reijers is a full professor in the Department of Information and Computing Sciences of Utrecht University, where he holds the chair in Business Process Management and Analytics. He is also a part-time, full professor in the Department of Mathematics and Computer Science of Eindhoven University of Technology, as well as an adjunct professor in the School of Information Systems of Queensland University of Technology. Previously, he headed a research unit within Lexmark and led IT projects as a management consultant for Accenture and Deloitte. The focus of his research is on business process innovation, process analytics, robotic process automation, and enterprise IT. On these and other topics, he published over 200 scientific papers, book chapters, and professional publications. His latest research is concerned with how to let people and computer systems work together gracefully within business processes.

W. M. P. van der Aalst is a full professor at RWTH Aachen University leading the Process and Data Science (PADS) group. He is also part-time affiliated with the Fraunhofer-Institut für Angewandte Informationstechnik (FIT) where he leads FIT's Process Mining group. His research interests include process mining, Petri nets, business process management, workflow management, process modeling, and process analysis. Next to serving on the editorial boards of over ten scientific journals, he is also playing an advisory role for several companies, including Fluxicon, Celonis, and Processgold. Van der Aalst received honorary degrees from the Moscow Higher School of Economics (Prof. h.c.), Tsinghua University, and Hasselt University (Dr. h.c.). He is also an elected member of the Royal Netherlands Academy of Arts and Sciences, the Royal Holland Society of Sciences and Humanities, and the Academy of Europe. In 2018, he was awarded an Alexander-von-Humboldt Professorship.