Unsupervised Task Recognition from User Interaction Streams

Adrian Rebmann and Han van der Aa

Data and Web Science Group, University of Mannheim, Mannheim, Germany
{rebmann|han.van.der.aa}@uni-mannheim.de

Abstract. User interaction events can give an accurate picture of tasks executed in a process, since they capture work performed across applications in a detailed manner. However, such data is too low level to be used for process analysis directly, since the underlying tasks are typically not apparent from individual events. Therefore, several task-recognition techniques were recently proposed that are able to abstract user interaction data to a higher level. However, these techniques work in an offline manner, requiring user interaction data to be stored in event logs. Such storage is often infeasible, though, due to the data's sheer volume and its privacy-sensitive nature. While this can be avoided by analyzing user interaction data in a streaming manner, existing task-recognition techniques cannot be applied to such settings, since they require multiple, post-hoc passes over the entire data collection. To overcome this, we propose the first approach for unsupervised task recognition from user interaction streams. For a given stream, our approach continuously identifies task instances, groups them according to their type, and emits task-level events to an output stream. Our evaluation demonstrates our approach's efficacy and shows that it outperforms two baseline approaches.

Keywords: Streaming process mining · User interaction data · Task recognition.

1 Introduction

Process mining comprises methods for the analysis of event data recorded by information systems in order to gain insights into the true behavior of organizational processes [1]. To be able to apply these methods, events that correspond to the execution of tasks have to be recorded by the systems supporting a process. However, any tasks performed outside of those systems, e.g., involving email, spreadsheets, or web applications, will not be included in such data, and, therefore, will not be taken into account when applying process mining methods.

By contrast, user interaction data [3,19] has the potential to give an accurate picture of such tasks, since it records all actions performed by a user across applications, at a detailed level and with no need to extract or integrate data from heterogeneous systems. Yet, the fine-granular nature of this data is both a blessing and a curse: raw interaction events are too low-level to be used for process analysis directly, since the purpose of the individual events from a process perspective is typically not apparent. For instance, what was the role of *Click button* and *Edit field* events in a purchase-to-pay setting? Therefore, so-called *task recognition* is required to infer which higher-level tasks are described in low-level user interaction data. Though a challenging problem in an unsupervised setting, some initial approaches to address it [18,23] were recently proposed.

However, these existing approaches are designed for offline analysis of stored user interaction logs, whereas we argue that there are two key benefits to performing task recognition in an online manner, i.e., by analyzing streams of user interaction events. First, offline analysis requires large amounts of low-level data to be stored, including data on events that are not relevant from the perspective of process analysis (e.g., a user logging into an SAP system) or that contain personal information (e.g., records of visits to news websites or private communication). By contrast, performing task recognition in an online manner means that only high-level, process-relevant event data is emitted for further analysis or storage. Second, offline task recognition means that downstream process analysis can only give post-hoc insights into a process, whereas performing it an online manner enables the subsequent application of streaming process mining techniques [9], which can provide a timely understanding of current process behavior and enable process predictions on-the-fly.

Because of these benefits of online task recognition and because existing approaches cannot be applied in online settings, since they perform multiple passes over an entire data collection, we use this paper to propose a novel approach for unsupervised task recognition based on user interaction streams. Our approach continuously identifies task instances and categorizes them according to their type, while adhering to the constraints of the streaming setting it operates in. In this manner, we turn a stream of user interaction events into a stream of task-related events, ready for downstream analysis.

In the following, Section 2 illustrates the challenges of unsupervised task recognition over streams and Section 3 presents key definitions. Section 4 presents our approach itself, which is evaluated in Section 5. Finally Section 6 summarizes related work and Section 7 discusses limitations and concludes.

2 Problem Illustration

Our work deals with task recognition in situations where user interaction events arrive in a stream and there is no information available about the relation between user interactions and high-level tasks. This section illustrates the problem of such unsupervised task recognition from events in general, before introducing the additional challenges of doing this in a streaming setting.

Unsupervised task recognition. To illustrate the goal of (unsupervised) task recognition, consider the excerpt of an event stream in Table 1, where the events record how a user handles requests related to orders. Although the user interaction events show what a user does at a low level (e.g., which buttons are clicked), it fails to give a clear impression about the actual process that is executed. For example, it is hard to recognize that events *u1*–*u8* correspond to the execution of a specific task, i.e., creating a new order, and events *u9*–*u16* to a different one, i.e., updating an existing order after a change request was made. Task recognition aims to extract such insights from user interaction data, which involves two parts:

1. *Task identification.* The first step in task recognition is to identify sequences of user interaction events that together form individual tasks, also referred to as *segmen-*

Table 1: An excerpt of a user interaction stream recording the execution of two tasks.

ID	Action	Application	Timestamp	Element	Label	Value
...	...	...	...	...	...	...
u1	click	Mail	15:41:32	list	Order	-
u2	input	Chrome	15:42:10	field	Login	-
u3	input	Chrome	15:42:26	field	Password	-
u4	click	Chrome	15:42:31	button	ok	-
u5	click	Chrome	15:43:01	button	Create order	-
u6	input	Chrome	15:43:29	field	Search	Pete Miller
u7	input	Chrome	15:43:43	field	Customer	C0075
u8	click	Chrome	15:43:58	button	Save	O007501
u9	click	Mail	15:44:32	list	Change request	-
u10	input	Chrome	15:44:41	field	Search	C0081
u11	click	Chrome	15:45:39	button	Edit	O008102
u12	input	Chrome	15:45:48	field	Quantity	4
u13	click	Chrome	15:46:05	button	Save	O008102
u14	click	Chrome	15:46:39	button	Edit	O008102
u15	input	Chrome	15:46:48	field	Quantity	5
u16	click	Chrome	15:46:55	button	Save	O008102
...	...	...	...	...	...	...

tation [18]. Working under the assumption that a user performs one task before moving to the next, this involves the identification of points in the data where one task completes and the next one starts. In the given example, this is the case after events *u8* and *u16*, which denote the completion of two higher-level tasks. The difficulty here is that such end points are not explicitly indicated in the data. For instance, although *u8* ends the first task by the press of a *Save* button, event *u13* involves such a button as well, even though it occurs only halfway through the execution of the second task. As such, task identification must infer when execution has moved to the next task, based on clues from the context and attributes of events.

2. *Task categorization.* Having identified individual tasks, task categorization strives to recognize which tasks correspond to the same type of task (such as *creating an order* or *updating a quantity*), and which to different ones. Due to variability, such categorization is difficult, though, since the same process-level task may be executed by performing different sequences of user interaction events. For instance, the *create order* task (*u1*–*u8*) could also be executed without first logging in (*u2*–*u4*) or by having to search multiple times (*u6*) until the right customer is found.

Challenges of the streaming setting. Performing task recognition over a user interaction stream is more complex than doing it in an offline manner using an event log, due to the general constraints of streaming settings [8]. Specifically, we have to identify tasks and categorize them as they are observed, using just a limited buffer to temporarily store a small number of events. This leads to two main difficulties:

1. *Single-pass processing.* Offline task-recognition approaches can do multiple passes over an entire collection of events, allowing them to use global information such as co-occurrence counts and similarity scores [23]. In a streaming setting, decisions have to be made as events are recorded, which means that they can only consider

local information. For example, we have to decide if events *u1* to *u16* form one or more tasks in the moment, without being able to revise this decision later.

2. *Adapting to changes over time.* An associated issue is that when dealing with streams, decisions have to be made without knowing what kind of events will arrive in the future. For example, while offline task-recognition approaches can be certain that all types of tasks they need to identify and categorize are already available, this is not the case in a streaming setting. At any point in time, events corresponding to new kinds of applications, actions, or task types may be observed. For instance, for the running example, events *u9–u16* must be properly analyzed, even if no such *update order* task has been seen before, which requires on-the-fly updating of the task-identification and categorization mechanisms.

3 Preliminaries

User interactions and user interaction events. A *user interaction* is a manual activity performed on a user interface, e.g., clicking a button or entering a value into a text field [2]. In line with the definitions of Leno et al. [18], a *user interaction event* (simply *event* in the remainder) $ue = (id, ts, P, V)$ is a tuple that records a user interaction, with $\mathcal{E}$ the universe of all events. Each event has a unique identifier $ue.id$, a timestamp $ue.ts$, a set of context attribute values $ue.P$, capturing the interaction type and information about the affected user interface element, and a set of data attribute values $ue.V$, capturing data associated with an interaction, e.g., what the user typed into a field. For instance, *u6*=(*u6*, *15:43:29*,{*input*, *Chrome*, *field*, *Search*}, {*Pete Miller*}).
Event class. Given an event, we let its context attributes values, i.e., $ue.P$, define its event class. For instance, the event class of *u6* is given as {*input*, *Chrome*, *field*, *Search*}.
User interaction stream. A user interaction stream S_E is a potentially infinite sequence of events recorded during task execution, i.e., $S_E \in \mathcal{E}^* \forall_{1 \leq i < j \leq |S_E|} S_E(i) \neq S_E(j)$.
Tasks and task-level events. A task is a single unit of work that is part of an organizational process. A *task-level event* $te = (id, type, ts, D)$ is a tuple that corresponds to the execution of a task, with $\mathcal{T}$ the universe of all task-level events. Each task-level event has a unique identifier $te.id$, relates to a task type $te.type$, has a timestamp $te.ts$, and has optional information captured in its set of attribute-value pairs $te.D$, such as life cycle information that, e.g., indicates whether the event corresponds to the start or completion of a task. For instance, the start of the task that corresponds to *u9–u16*, is given by te_1 =(*1*, *Change order*, *15:44:32*, {(`lifecycle`, *start*)}).
Task-level event stream. A task-level event stream S_T is a potentially infinite sequence of task-level events, i.e., $S_T \in \mathcal{T}^* \forall_{1 \leq i < j \leq |S_T|} S_T(i) \neq S_T(j)$.

4 Approach

Fig. 1 provides a high-level overview of our approach, which we complement with a formalization in Algorithm 1. As depicted, our approach collects user interaction events from a stream S_E into a buffer B. In this work, we assume that the buffer's size is large enough to store the events corresponding to a single task instance. Given the events collected in the buffer, task recognition consists of two components: the *task-identification* *component* identifies sequences of events that correspond to individual tasks, whereas the *task-categorization component* subsequently assigns a type to them. Finally, for each task recognized in this manner, our approach emits a start and a completion event to a task-level event stream S_T.

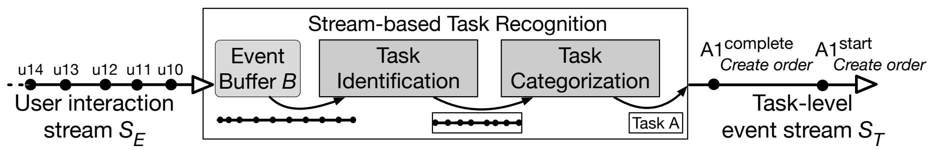

Fig. 1: Overview of our task-recognition approach.

Algorithm 1 Stream-based task recognition

Input S_E: User interaction stream, *b*: Maximum buffer size
Output S_T: Task-level event stream

▷ **Initialize buffer B, clustering model M, and chunk list C**
$B \leftarrow$ new FIFOQueue(b), $M \leftarrow$ new OnlineClusteringModel(), $C \leftarrow []$
loop forever
  $e \leftarrow S_E$.observeEvent() ▷**A new event is consumed from the stream**
  B.insert(e) ▷**Add the new event to the buffer**
▷ **Task identification**
  if completesChunk(e) **then**
    C.add(B.getEventsSinceLastChunk(C)) ▷**Create and store new chunk**
  if $|C| \geq 3$ **then** ▷**Check if enough chunks available**
    $c_i, c_{i+1} \leftarrow C[-3], C[-2]$ ▷**Get chunks to be checked**
    if endsTask(B, C, c_i, c_{i+1}) **then** ▷**Check if c_i completes a task**
      $task \leftarrow$ new Task(B.dequeueUpThrough(c_i)) ▷**De-queue events to create task**
      $C \leftarrow C$.removeRange($C[0], c_i$) ▷**Remove chunks that are part of new task**
▷ **Task categorization**
      $v \leftarrow$ vectorize($task$) ▷**Create a feature vector of the task**
      M.update(v) ▷**Update the clustering model**
      $task.type \leftarrow M$.categorizeTask(v) ▷**Assign a type to the task**
▷ **Emit task-level events**
      emit(S_T, $task$.startEvent(), $task$.completionEvent())

4.1 Task Identification

The task-identification component identifies sequences of events from the stream that correspond to individual tasks. It consists of two main operations, as visualized in Fig. 2. Here, *chunking* identifies sequences of observed events that represent sub-tasks, such as filling in a form or sending an e-mail, whereas *segmenting* determines if consecutive

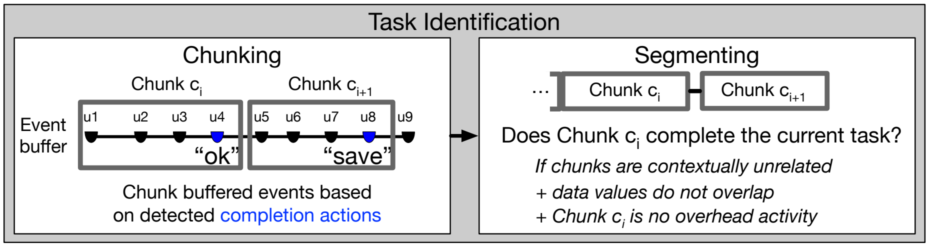

Fig. 2: Task-identification component.

sub-tasks corresponds to the same process-level task, or rather to different ones. Once such a transition from one task to the next has been detected, we forward the segment that corresponds to the completed task to our task-categorization component.

Chunking. We recognize sub-tasks by looking for common keywords in user interaction data that indicate the conclusion of an interaction sequence, achieved through the `completesChunk` function in Algorithm 1 (line 5). To operationalize this function, we established a set of completion actions K_A, which consists of 20 keywords stemming from design guidelines for user interfaces by IBM [17], covering typical terms that indicate the conclusion of a smaller part in a process, such as *ok* (to go to the next step in a user interface), *submit* (for a form), *send* (e-mail), or *save* (changes).[1]

For an event e, `completesChunk`(e) returns true if e's event class contains a mention of an action in K_A. Based on the events stored in B, a sub-task is formed by the events that occurred since the last completed chunk in C (line 6). For instance, for the running example, *u4*, *u8*, *u13*, and *u16* complete chunks (due to their *ok* and *save* labels), which results in *u1–u4*, *u5–u8*, *u9–u13*, and *u14–u16* as chunks.

Segmenting. The segmenting operation aims to decide whether a chunk c_i corresponds to the end of a task or if it continues with the next chunk, c_{i+1} (function `endsTask` in line 9). Specifically, as shown in Fig. 2, `endsTask` identifies c_i as finalizing a task if: (1) the chunks are contextually unrelated to each other, (2) the chunks have no overlap in data values, and (3) c_i does not represent an overhead activity. Otherwise, c_i and c_{i+1} are considered to belong to the same task.

(1) Assessing contextual relatedness. Our approach first checks if c_i and c_{i+1} are contextually related. We do this by lifting the notion of contextual relatedness proposed by Urabe et al. [23], which targets offline segmentation, to our setting. The idea is to check if the event classes contained in c_i and c_{i+1} commonly co-occurred so far (indicating a shared context) or not (suggesting that the chunks are part of different tasks).

As illustrated in Fig. 3, contextual relatedness is quantified on the basis of a global co-occurrence matrix, which tracks how often pairs of event classes have been observed to be part of the same chunk. Based on the global counts, we obtain the co-occurrence vectors of the event classes per chunk (i.e., rows in the co-occurrence matrix) and compute their centroid. Then, we compute the similarity score $\mathtt{sim}(c_i, c_{i+1})$ as the cosine similarity between the centroids of c_i and c_{i+1}.

[1] We refer to our repository for the full list of keywords, though K_A can naturally be extended with, e.g., self-defined keywords or other languages.

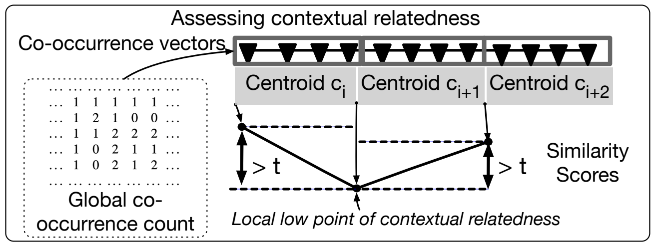

Fig. 3: Adapted contextual-relatedness approach by Urabe et al. [23]

Finally, we place $\mathtt{sim}(c_i, c_{i+1})$ into the context of other scores to determine if the chunks are contextually related or not. Specifically, we check if $\mathtt{sim}(c_i, c_{i+1})$ represents a local minimum by comparing it to the highest similarity score observed since the last identified task ($prev_{max}$) and to the next similarity score ($\mathtt{sim}(c_{i+1}, c_{i+2})$). Given a similarity threshold t,[2], if $prev_{max} - \mathtt{sim}(c_i, c_{i+1}) > t$ and $\mathtt{sim}(c_{i+1}, c_{i+2}) - \mathtt{sim}(c_i, c_{i+1}) > t$, chunks c_i and c_{i+1} are considered contextually unrelated, since their contextual-relatedness score negatively stands out.

In this manner, given the four chunks identified in the previous operation, we would determine that the transitions from *u1–u4* (logging in) to *u5–u8* (creating an order), and from *u5–u8* (creating an order) to *u9–u13* (updating a quantity) are both clear changes in context. By contrast, the transition from *u9–u13* to *u14–u16* occurs within the same context (updating and fixing an order quantity), due to its strongly related event classes.

(2) Checking for data value overlap. Next, we recognize that sub-tasks may be part of the same tasks, even when they relate to different contexts, such as opening a request sent by a customer per e-mail and subsequently updating one of their orders in a system. Therefore, we check if events belonging to chunks c_i and c_{i+1} share particular attribute values, such as a customer name or an order ID. Specifically, we check the last two events of c_i and the first two of c_{i+1} for exact matches in their attribute sets V, and if these are present, determine that there should be no segmentation between c_i and c_{i+1}.

In this manner, we would, for instance, recognize that chunks *u9–u13* and *u14–u16* also relate to each other in terms of their data attributes, because events *u13* and *u14* both refer to order number *O008102*, thus avoiding segmentation here.

(3) Checking for overhead sub-tasks. Finally, we check if c_i actually corresponds to a sub-task performed for a particular process instance or that it, rather, corresponds to overhead being performed. Common examples of this include logging into a system, launching an application, or visiting non-work related websites. If c_i represents such an overhead sub-task, we do not want to treat this chunk as a distinct task on a process level, which is why we would not segment after c_i (even though contextual relatedness or shared data values between c_i and c_{i+1} are unlikely).

To operationalize this final check, we established a set K_O of overhead keywords based on the guidelines [17] we also use for chunking, including *log in*, *sign up*, *reload*, and *open*. Using this set, we check if a member of the last two event classes of c_i is contained in K_O and, if so, avoid segmentation. In this manner, we, e.g., recognize

[2] t is configurable, yet, we set it to 0.1 as done by the authors of the offline approach [23].

that the first sub-task in our running example (*u1*–*u4*), which corresponds to the user logging into a web app, belongs to the same task as the next chunk *u5*–*u8*, where the same system is used to create an order.

Post processing. When our approach has detected that c_i represents the final chunk of a task (line 9), this means that all events currently in the buffer, up to and including the final event of c_i, together form a task. This completed task is then forwarded to the categorization component, whereas its individual events are removed from buffer B and chunk list C (lines 10–11), so that the first event in B is the first event of the next task.

4.2 Task Categorization

The task-categorization component assigns a type to tasks identified by the previous component. Given that we cannot store identified tasks in a streaming setting, we categorize them directly after identification. This is complex, though, because it means we may not yet have observed all possible task types.

To deal with this challenge, we perform task categorization on the basis of an online clustering model M, which is incrementally updated as new task instances arrive. As shown in Fig. 4, this involves the transformation of a task into a feature vector, updating the model M, then using it to assign a cluster and a corresponding cluster explanation. The latter part is necessary, because the clustering only indicates that certain tasks belong to the same type (e.g., *Task A*), but not what this type entails [22].

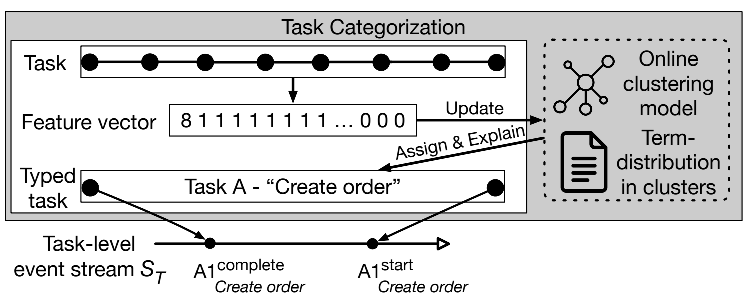

Fig. 4: Task-categorization component.

Establishing feature vectors of tasks. Given an identified task, we first transform its contents into a feature vector that can be used for clustering (line 12). We use a vector encoding that accounts for variability in the executions of tasks of the same type, such as tasks that consist of slightly different sets of events or that are performed in a different order. Therefore, we capture the number of unique event classes (as an indicator of a task's complexity) and the frequency of each event class (to capture its contents) as features. For instance, the task *u1*–*u8* in our running example consists of eight event classes, all performed once, resulting in a vector $\langle 8, 1, 1, 1, 1, 1, 1, 1, 1, \ldots, 0, 0, 0\rangle$.

Here, the zeros at the end are used to ensure that vectors remain of the same size s_v throughout the stream[3], accounting for a number of event classes not seen so far.

Clustering tasks. We use an online clustering model M to recognize that tasks are of the same type, based on their vector representation. Specifically, we use *DenStream* [10], a density-based online clustering technique building on the DBSCAN algorithm [15]. It dynamically creates, updates, and deletes so-called micro-clusters in the online feature space it maintains. This technique has two key benefits. First, the technique is highly memory efficient, since it only stores summaries of vector sets (the micro-clusters), rather than the vectors themselves. Second, unlike many other clustering techniques, it does not depend on a user-defined number of desired clusters.

As shown in Fig. 4 and Algorithm 1, we update the clustering model M with the vector v that corresponds to an identified task (line 13), before using it to assign the task to a cluster (line 14). For instance, due to the distinct features of the two tasks in our running example, these are assigned to different clusters (e.g., types *A* and *B)*.

Providing type explanations. After using clustering to recognize tasks that belong to the same or to different types, we aim to provide useful indicators of what these types (like *A* or *B*) actually mean. For this, we detect parts of event classes that are distinctive for the detected clusters, e.g., that tasks of type *A* uniquely involve interactions with a *Create order* button or that type *B* involves the editing of a *Quantity* field.

To this end, we use the well-known *term frequency–inverse document frequency* (tf-idf) score. For this, we use a term dictionary to keep track of the frequency of terms used in the event classes within tasks of a specific type, where a *term* is an attribute value part of P, such as *click*, *Chrome*, *button*, or *Create order*. Using $M.types$ to refer to the types (i.e., clusters) currently recognized by the clustering model M, we write $\texttt{tf-idf}(x, type)$ as the score for a term x and a type $type \in M.types$.

Then, we set the *explanation* of *type* as the term x with the highest $\texttt{tf-idf}(x, type)$, e.g., *Create order* for type *A*. If multiple types are assigned the same explanation (e.g., if *Quantity* is the most distinctive term for types *B* and *C*), we add the term with the next highest `tf-idf` score to each explanation, until they are all unique. For example, *B* may get *Quantity, update*, while *C* gets *Quantity, Confirm*.

4.3 Output

The output of our approach is a stream of task-related events based on the identified and categorized tasks. For each task, we emit a start event with the timestamp of its first user interaction event and a completion event with the last timestamp (line 15). Both task-level events are assigned a unique identifier, as well as the type and explanation assigned by the categorization component. For instance, for *u1–u8*, we emit:

$$te_1 = \{\mathit{1}, \mathit{A}, \mathit{15{:}41{:}32}, \{(\texttt{lifecycle}, \mathit{start}), (\texttt{explain}, \mathit{Create\ order})\}\}$$
$$te_2 = \{\mathit{2}, \mathit{A}, \mathit{15{:}43{:}58}, \{(\texttt{lifecycle}, \mathit{complete}), (\texttt{explain}, \mathit{Create\ order})\}\}$$

For *u9–u16* we emit:

[3] Given that the final number of event classes is unknown, s_v should be set sufficiently large. We set 1,000 as the default for s_v, which already covers more than 6 times the total number of event classes in our evaluation data.

$te_3 = \{3, B, 15{:}44{:}32, \{(\texttt{lifecycle}, start), (\texttt{explain}, Edit,\ Quantity)\}\}$
$te_4 = \{4, B, 15{:}46{:}55, \{(\texttt{lifecycle}, complete), (\texttt{explain}, \{Edit,\ Quantity\})\}\}$

Compared to the fine-grained input events in our running example, our approach thus only emits events that are important for process analysis, whereas potentially sensitive or irrelevant user behavior and information is omitted from consideration.

5 Experimental Evaluation

We describe the data collection used in our experiments in Section 5.1 and the setup in Section 5.2. In Section 5.3 we present the evaluation results showing our approach's capability to recognize tasks in a stream of events and compare it against two baselines. The implementation, data collection, evaluation pipeline, and raw results are all available in our repository.[4]

5.1 Data Collection

There are no publicly available event logs (let alone streams) that contain interaction data related to different task types, associated with a necessary gold standard. Therefore, we follow the idea of Urabe et al. [23] and take available task logs, each recording various instances of the same task type, and combine these into three evaluation logs, which thus cover multiple different task types, in various orders. Finally, we use these event logs as a basis to simulate event streams.

Task logs. As depicted in Table 2, we have eight task logs available, stemming from three sources. The tasks can be divided into two groups, with types 1–4 involving *copying data* from spreadsheets to web forms, filing *reimbursement* requests, entering *student records* into a web-based app, and filling out *travel requests*, whereas types 5–8 all relate to the creation of informational objects in an SAP system. Six out of these eight task logs contain overhead tasks such as logging into a system, starting an application, or opening a file. As shown in the table, the logs also differ considerably in their variation and task lengths.

User interaction streams. We established three evaluation logs (available on our repository) by combining and shuffling the tasks contained in individual task logs, which we use to simulate three streams (S_{E1}–S_{E3}):

- S_{E1} consists of 200 tasks, covering types 1–3, with a total of 6,114 events. This stream includes the same task types as used in the evaluation of Urabe et al. [23].
- S_{E2} consists of 240 tasks, covering types 1–4, with a total of 9,054 events. It is an extension of *L1*, based on the recently released task log of type 4.
- S_{E3} consists of 120 tasks, covering types 5–8, with a total of 1,386 events. We use the task types included here in isolation from the others, since their event attributes differ considerably from those in the other task logs, which would bias the results.

We unified the data structure across the task logs as much as possible before merging them, such that attribute names are the same for all task types.

[4] https://gitlab.uni-mannheim.de/processanalytics/task-recognition-from-event-stream

Table 2: Characteristics of the task logs used in our experiments

Type	Source	Description	#Tasks	Avg. length	#Variants	#Events	#Classes
1	[18]	Copy data	100	14.5	7	1,462	15
2	[18]	Request reimbursement	50	2.3	2	3,113	32
3	[18]	Enter student records	50	30.8	1	1,539	23
4	[4]	Fill in travel request	40	73.5	2	2,940	48
5	[3]	Create group	30	10.9	30	331	9
6	[3]	Create keyword	30	10.1	30	425	12
7	[3]	Create version	30	14.2	30	304	8
8	[3]	Create document	30	11.0	30	326	9

5.2 Setup

Environment. We implemented our approach in Python and ran our experiments single-threaded on a laptop with a 2 GHz Intel Core i5 processor and 16GB of memory.

Configurations. Our approach requires a buffer size b that can store all events belonging to a single task. We report on the results using a buffer size of 250 events (also for the baselines), which covers $3\times$ the number events of the longest task in our data. Furthermore, we consider that two parts of our approach need to be initialized at the beginning of a stream and populated over time: the global co-occurrence matrix used during task identification and the online clustering model used for task categorization. Given that the accuracy of these two parts may improve as more events are observed, we test the value of a *warm-up phase*, where our approach populates the matrix and clustering model using the first 0, 100, 250, 500, or 1,000 events, before starting to perform identification and categorization on them.

Baselines. Because there are currently no techniques capable of recognizing tasks based on a stream of user interactions, we established two baselines by adapting existing works. The task-identification component of each baseline consists of an existing, offline identification technique, lifted to an online setting (as described below). Their task-categorization components, by contrast, are operationalized with the same clustering technique used in our proposed approach, which is necessary because offline categorization techniques cannot be lifted to an online setting.

- *BL1: Back-edge-based identification.* Leno et al. [18] proposed a log segmentation technique based on back-edges identified from a directly-follows graph (DFG). We adapted the technique to build the DFG incrementally using the same event classes as available to our approach and apply the authors' back-edge detection method periodically, after every b events (i.e., each time the buffer is full).
- *BL2: Co-occurrence-based identification.* The approach by Urabe et al. [23], which also inspired parts of our work, leverages co-occurring event classes in fixed windows to segment a log. We adapted it to count co-occurrence incrementally and compute similarities on a buffer of events. We use the same parameter configurations as reported in the original paper.

Measures. We use the following measures for quality and efficiency in our experiments.

Identification quality. We assess identification quality by comparing the identified task segments to those of the corresponding gold standard, for which we use two measures:
- *# of tasks.* To assess if an approach makes the right amount of segmentation decisions, we compare the numbers of identified and gold-standard tasks.
- *Normalized edit distance* (*nor.ED*). To quantify how similar the identified tasks are in comparison to the gold standard, we calculate the average normalized edit distance between identified tasks and their closest task in the gold standard.

Categorization quality. We assess categorization quality through measures for cluster quality, by comparing the tasks that are assigned to the same category to the gold-standard categorization (i.e., task types):
- *Rand index* (*R*). We compute the Rand index, which considers the fraction of pairs (tasks at macro level, events at micro level) that are correctly assigned to the same or to different categories, i.e., $^{(TP+TN)}/_{(TP+TN+FP+FN)}$, where a true positive (TP) indicates that two tasks/events are correctly assigned to the same category.
- *Jaccard index* (*J*). We also compute the weighted average Jaccard index to quantify the similarity between identified clusters and the gold-standard clusters, which is given as $^{A \cap B}/_{A \cup B}$ per cluster, with A a cluster's identified contents and B its gold-standard contents (i.e., tasks at the macro level, events at the micro level).

Efficiency. Finally, we assess the memory and response time efficiency of our approach:
- *Memory consumption.* We measure the maximum memory required by our approach, which is the sum of the largest buffer size during runtime, the final size of the global co-occurrence matrix, and the final size of the clustering model.
- *Response time.* We measure how long it takes our approach to perform task identification after an event arrives, i.e., determining if a task has been completed, as well as how long it takes to categorize an identified task.

5.3 Results

We first present the overall results of our approach compared to the baselines, before discussing the impact of the warm-up phase, memory consumption, and response time.

Overall results. Table 3 shows the results obtained using our approach (with a warm-up of 250 events), the two baselines, and a perfect identification strategy (to show the quality of our categorization step independently of identification quality).

Identification results. Our approach achieves highly accurate identification results for S_{E1} and S_{E2}, identifying approximately the same numbers of tasks as in the gold standard (198 vs 200 and 231 vs 240), to which they are very close in terms of contents, yielding edit distances of just 0.04 and 0.06. Stream S_{E3} is more challenging, though. Our approach overestimates the total number of tasks (138 versus 120), and achieves an edit distance of 0.23. Taking an in-depth look at the results, we find that our approach occasionally fails to recognize that certain sub-tasks belong to the same gold-standard task, since they lack contextual relatedness and overlapping data values.

Compared to the baselines, our approach consistently obtains better results in terms of edit distances. This indicates that the tasks they identify differ more from their gold-standard counterparts than the ones identified by our approach. Both baselines often miss segmentation points, resulting in much lower numbers of identified tasks than

Table 3: Results of our approach and the baselines (warm-up of 250 events). ↑ and ↓ indicate the desired direction per measure.

		Identification		Categorization			
Stream	**Approach**	**# tasks**	**nor.ED ↓**	**R(mi) ↑**	**R(ma) ↑**	**J(mi) ↑**	**J(ma) ↑**
S_{E1}	*Our approach*	**198**	**0.04**	**0.94**	**0.95**	**0.92**	**0.92**
	BL1	202	0.83	0.58	0.82	0.38	0.89
	BL2	159	0.33	0.74	0.80	0.64	0.71
	Perfect ident.	200	0.00	1.00	1.00	1.00	1.00
S_{E2}	*Our approach*	**231**	**0.06**	**0.89**	**0.95**	**0.82**	**0.89**
	BL1	99	0.32	0.42	0.80	0.24	0.56
	BL2	198	0.33	0.69	0.71	0.50	0.51
	Perfect ident.	240	0.00	0.97	0.97	0.95	0.95
S_{E3}	*Our approach*	**138**	**0.23**	**0.87**	**0.88**	**0.76**	**0.76**
	BL1	29	0.58	0.34	0.49	0.16	0.35
	BL2	58	0.37	0.45	0.57	0.31	0.46
	Perfect ident.	120	0.00	1.00	1.00	1.00	1.00

contained in the gold standard. *BL1*, specifically, only finds 99 tasks for S_{E2} (out of 240) and 29 for S_{E3} (out of 120). Although *BL2* generally performs better than *BL1*, we find that its results are heavily dependent on the selection of two parameter values, with the edit distances differing by up to 0.5 across configurations.[5]

Overall, these results indicate that our approach, which takes the semantic and data perspectives into account, on top of the control-flow perspective also considered by the baselines, leads to more accurate task identification, whereas our approach also does not depend on user-defined parameters (unlike *BL2*).

Categorization results. Our approach achieves high macro Rand scores of 0.95 for S_{E1} and S_{E2} and 0.87 for S_{E3}, which shows that it accurately assigns pairs of tasks to the same category as their gold-standard counterparts. The comparable micro-level scores show that this categorization quality generally also holds for pairs of events, which thus accounts for tasks of different lengths. The Jaccard index, which provides insights into the quality per cluster, rather than per task (or event) pair, confirms the accurate categorization quality, achieving macro scores of 0.92 for S_{E1}, 0.89 for S_{E2} and 0.76 for S_{E3} and the comparable scores on the micro level (0.92, 0.82, and 0.76).

The lower scores for S_{E3} can largely be attributed to the more challenging nature of this stream when it comes to task identification, as evidenced by the results obtained when using our categorization component on perfectly identified tasks (gray row in Table 3). The results reveal that categorization itself is highly accurate, achieving perfect scores for S_{E1} and S_{E3}, and near-perfect ones (≥ 0.95) for S_{E2}. As also confirmed by the results of the baselines, which use the same categorization technique as our approach, it is thus clear that lower identification quality leads to lesser categorization results.

Impact of warm-up phase. Table 4 shows the results of our approach for warm-up phases of 0, 250, 500, and 1,000 events. We find that there is little to no impact on

[5] See our repository for detailed experiments regarding *BL2*'s parameter configurations.

Table 4: Results of our approach with warm-up phases of 0, 250, 500, and 1,000 events. ↑ and ↓ indicate the desired direction per measure.

Stream	Warm-up	Identification		Categorization			
		# tasks	nor.ED ↓	R(mi) ↑	R(ma) ↑	J(mi) ↑	J(ma) ↑
S_{E1}	*0 events*	198	0.04	0.89	0.84	0.84	0.79
	250 events	198	0.04	0.94	0.95	0.92	0.92
	500 events	198	0.04	**0.96**	**0.96**	**0.94**	**0.94**
	1,000 events	198	0.04	0.96	0.96	0.94	0.94
S_{E2}	*0 events*	231	0.06	0.89	0.95	0.82	0.89
	250 events	231	0.06	0.89	0.95	0.82	0.89
	500 events	231	0.06	0.89	0.95	0.82	0.89
	1,000 events	**236**	**0.05**	**0.95**	**0.96**	**0.92**	**0.94**
S_{E3}	*0 events*	138	0.23	0.76	0.79	0.57	0.57
	250 events	138	0.23	0.87	0.88	0.76	0.76
	500 events	138	0.23	**0.90**	0.91	**0.83**	0.82
	1,000 events	138	0.23	0.90	**0.94**	0.83	**0.88**

identification quality, since only for S_{E2} the number of identified tasks and edit distance improve slightly (by 5 resp. 0.01), when setting the warm-up to 1,000 events compared to no warm-up at all. For categorization quality, the benefit of a warm-up phase becomes clear, though. While for 0 warm-up events, we achieve a macro Rand score of 0.84 and Jaccard score of 0.79 for S_{E1}, setting the warm-up phase to 500 events increases the scores by 0.12 resp. 0.15. A further increase to 1,000 has no substantial impact, though, only leading to improvements for S_{E2}. Overall, these results show that a warm-up phase is not necessary for task identification, but that it is beneficial for task categorization, if an application context allows for it. However, its length can be relatively short (e.g., just one to two times the buffer size).

Efficiency. We find that our approach requires less than 1% of the memory that would be needed to store all events from the streams, thus clearly demonstrating its memory efficiency. When it comes to response time, we find that our approach requires between 2 and 4 ms for task identification and between 40 to 150 ms for task categorization. Given that the average time between user interactions is over 2.5 seconds in the available data, this means that our approach can easily keep up in terms of responses.

6 Related Work

Our work primarily relates to task recognition from user interaction logs and other low-level event data as well as pre-processing techniques for stream-based process mining.

Urabe et al. [23] segment logs in a post-hoc manner based on contextual relatedness, which we lift to an online setting as part of our identification component and use as a baseline in our evaluation. Other offline approaches [12, 18] segment logs by focusing on frequent execution patterns in the logs assuming that task execution is deterministic. Therefore, these approaches cannot handle cases with multiple execution variants well. Agostinelli et al. [4, 5] take a supervised approach based on the computation of alignments between user interaction logs and task models that they require as input. Finally, Linn et al. [20] combine transactional data recorded by information systems with user interaction logs, to integrate interaction data with traditional process mining. Beyond user interaction events, related approaches recognize process-related tasks from ambient or wearable sensors [11, 21] or network traffic data [14]. However, due to the low-level, highly abstract nature of the data used by these approaches, they depend on supervised recognition strategies, as opposed to our unsupervised approach.

Research on stream pre-processing in process mining mostly focuses on cleaning noisy event streams. Van Zelst et al. [24] filter a stream based on estimates of how likely new events belong to real process behavior, whereas Hassani et al. [16] filter noise by extracting frequent sequential patterns from an event stream before applying streaming process discovery. Finally, Awad et al. [6] propose an approach to resolve situations in which events arrive in an incorrect order on a stream. However, these techniques assume arriving events to be on the task level, even though, in practice, streaming data is commonly at a lower-level of abstraction, such as taken into account by our approach.

7 Conclusion

In this paper, we proposed an approach to recognize tasks from a stream of user interaction events in a fully unsupervised manner. To this end, our approach continuously segments the stream to identify task instances and categorizes these according to their type. The output is a stream of task-related events, to which subsequent streaming analysis techniques can subscribe. We demonstrated our approach's efficacy in an experimental evaluation on real data and showed that it outperforms two baseline approaches.

In its current form, our work has limitations that we aim to address in the future. With respect to our evaluation, we acknowledge that, although the considered data includes a variety of task types, it does not capture a user's real sequence of process-related and overhead tasks conducted during a workday. Therefore, we plan to conduct further experiments as soon as more suitable data becomes available. As for our approach, we recognize that it currently does not consider event timestamps. It has been shown that these can help make correct segmentation decisions when there are large time differences between events, though [7]. Therefore, we aim to incorporate a strategy based on time differences into our identification component. Furthermore, our approach is currently limited to task recognition, so it does not identify information about the process instance or business objects to which a task relates. In the future, we aim to tackle this by building on existing work on case correlation [13] and leveraging data values associated with events (such as identifiers of orders and customers). Finally, our approach (and other unsupervised task-recognition approaches) so far assumes that tasks are executed sequentially, i.e., one task must be completed before another one is started. In the future, we aim to loosen this assumption, by also dealing with interleaving task executions, which again largely depends on the detection of data values that allow us to infer inter-relations between non-consecutive events.

References

1. van der Aalst, W.M.P.: Process Mining: Data Science in Action. Springer (2016)
2. Abb, L., Bormann, C., van der Aa, H., Rehse, J.R.: Trace clustering for user behavior mining. In: ECIS 2022 Research Papers. 34 (2022)
3. Abb, L., Rehse, J.R.: A reference data model for process-related user interaction logs. In: BPM. pp. 57–74. Springer (2022)
4. Agostinelli, S.: Automated segmentation of user interface logs using trace alignment techniques. In: ICPM Doctoral Consortium/Tools. pp. 13–14 (2020)
5. Agostinelli, S., Marrella, A., Mecella, M.: Automated segmentation of user interface logs. In: Robotic Process Automation, pp. 201–222. De Gruyter Oldenbourg (2021)
6. Awad, A., Weidlich, M., Sakr, S.: Process mining over unordered event streams. In: ICPM. pp. 81–88. IEEE (2020)
7. Bernard, G., Senderovich, A., Andritsos, P.: Cut to the trace! Process-aware partitioning of long-running cases in customer journey logs. In: CAiSE. pp. 519–535. Springer (2021)
8. Bifet, A., Gavalda, R., Holmes, G., Pfahringer, B.: Machine learning for data streams: with practical examples in MOA. MIT press (2018)
9. Burattin, A.: Streaming process mining. Process Mining Handbook, Springer (2022)
10. Cao, F., Ester, M., Qian, W., Zhou, A.: Density-based clustering over an evolving data stream with noise. In: International Conference on Data Mining. pp. 328–339. SIAM (2006)
11. Chen, L., Hoey, J., Nugent, C.D., Cook, D.J., Yu, Z.: Sensor-based activity recognition. IEEE Transactions on Systems, Man, and Cybernetics **42**(6), 790–808 (2012)
12. Dev, H., Liu, Z.: Identifying frequent user tasks from application logs. In: Proceedings of the 22nd international conference on intelligent user interfaces. pp. 263–273 (2017)
13. Diba, K., Batoulis, K., Weidlich, M., Weske, M.: Extraction, correlation, and abstraction of event data for process mining. Wiley Interdisciplinary Reviews **10**(3), 1–31 (2020)
14. Engelberg, G., Hadad, M., Soffer, P.: From network traffic data to business activities: A process mining driven conceptualization. In: BPMDS 2021. pp. 3–18. Springer (2021)
15. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: KDD. p. 226–231. AAAI Press (1996)
16. Hassani, M., Siccha, S., Richter, F., Seidl, T.: Efficient process discovery from event streams using sequential pattern mining. In: SSCI. pp. 1366–1373. IEEE (2015)
17. IBM: Carbon Design System - Action Labels (2022), https://carbondesignsystem.com/guidelines/content/action-labels/
18. Leno, V., Augusto, A., Dumas, M., La Rosa, M., Maggi, F.M., Polyvyanyy, A.: Identifying candidate routines for robotic process automation from unsegmented UI logs. In: ICPM. pp. 153–160. IEEE (2020)
19. Leno, V., Polyvyanyy, A., Dumas, M., La Rosa, M., Maggi, F.M.: Robotic process mining: vision and challenges. Business & Information Systems Engineering **63**(3), 301–314 (2021)
20. Linn, C., Zimmermann, P., Werth, D.: Desktop activity mining-a new level of detail in mining business processes. In: Workshops der INFORMATIK 2018-Architekturen, Prozesse, Sicherheit und Nachhaltigkeit. Köllen Druck+ Verlag GmbH (2018)
21. Rebmann, A., Emrich, A., Fettke, P.: Enabling the discovery of manual processes using a multi-modal activity recognition approach. In: BPM Workshops. Springer (2019)
22. Rebmann, A., Pfeiffer, P., Fettke, P., van der Aa, H.: Multi-perspective identification of event groups for event abstraction. In: ICPM Workshops. Springer, to appear (2022)
23. Urabe, Y., Yagi, S., Tsuchikawa, K., Oishi, H.: Task clustering method using user interaction logs to plan RPA introduction. In: BPM. pp. 273–288. Springer (2021)
24. van Zelst, S., Fani Sani, M., Ostovar, A., Conforti, R., Rosa, M.L.: Filtering spurious events from event streams of business processes. In: CAiSE. pp. 35–52. Springer (2018)