Log-Based Concept Drift Detection over Event Streams

Huete Guzmán, Jesús Santiago

Award date:
2022

Department of Mathematics and Computer Science
Process Analytics Research Group

Log-Based Concept Drift Detection over Event Streams

Master Thesis

Jesús Santiago Huete Guzmán

Supervisors:
dr. ing. Marwan Hassani - M.Hassani@tue.nl

Eindhoven, August 2021

Abstract

Process mining is an emerging field which focuses on applying data mining techniques over business process data. Recently, with the improvements in sensoring, collection, and storing of data technologies, a big demand for both shorter mining time and adaptive models of streaming process events arose. This initiated the field of stream process mining. Decision makers have placed great interest in drifts of the underlying concepts of the business processes. Stream process mining techniques have the ability to detect such drifts in real-time, compared to static approaches. This thesis contributes an unsupervised-learning novel approach for online concept drift detection over event streams. The intention is for it to address the limitations of the current state-of-the-art techniques, outperform the existing unsupervised-learning techniques, and provide useful insights on the drifts detected to understand the cause of such. The framework relies on data structures previously defined by a nobel process discovery algorithm [1] called prefix-trees. With the events from the stream stored in these trees, the data is converted from discrete to continuous using a principal component analysis technique [2] over trees. Finally, an adaptive window is used to process the data and perform a well-known concept drift detection technique [3]. The framework was extensively tested over six artificial logs and three real-world logs with different parameters to identify the best performing for each type of log. Later, such results were compared to those of three nobel concept drift detection techniques. The results showed a significant advantage of the proposed framework in terms of quality and delay of detection over the artificial logs. The three chosen techniques were not able to process the real-world datasets, while the proposed algorithm demonstrated positive results. Also, the framework provided meaningful insights on the cause for drifts identified in one of the real-world datasets, and produce sub-logs from these drifts to allow users to perform further process analysis.

Acknowledgements

This thesis represents the last milestone of the Erasmus Mundus Joint Masters Degree in Big Data Management and Analystics (Université Libre de Bruxelles, Universitat Politècnica de Catalunya, Technische Universiteit Eindhoven).

I would like to thank all the people that contributed to this achievement. From my supervisor Marwan, who guided and supported me over these six months, to all the professors and colleagues from the masters programme.

A special thanks goes to my parents, family, and friends. Thank you for encouraging me to never give up and always believing in me.

Jesús Santiago Huete Guzmán

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Business processes are a set of tasks performed by resources of an organization in order to achieve a defined business outcome. These processes need to continuously change in response to external factors. Example of such can be variations in demand or supply, customer expectations and governmental regulations, and seasonal factors. These are reflected in the underlining process by variations in the workload, replacement of resources, modifications on the sequence of tasks performed, etc. Furthermore, in some scenarios these changes are planned and documented, but others occur unexpectedly and remain unnoticed by the process owners. Unexpected changes are of importance to the organization due to the unknown impact they can have over their objectives. To cope with these, business process managers require tools and methods that allow them to detect and pinpoint them as early and accurately as possible. Here is where process mining comes into place. Process mining is an emerging data mining task of gathering valuable knowledge out of the huge amount of business operation data. It implements methods from fields like data mining and analyzes logs of process information to derive insights about the business process.

One field of process mining, which is focused on identifying points in time when the behavior of recent executions of the process differs significantly from that of an older one, is called concept drift detection (CDD). Techniques developed on this context use an event log or stream as an input and attempt to detect the points in time where the process deviated from its normal behavior. A group of techniques focuses on extracting a set of features from the stream and compare their distribution over a defined period of time [11] [9] [10] [12], while others focus on discovering business process models and applying conformance checking or a distance function over the event stream [8] [13]. Moreover, some techniques focus on the control-flow of the sequence of activities [14] [15], whereas others look at the performance or resources used by the process [16] [17]. Another aspect is whether the techniques can detect the drifts in a real-time scenario [18] [19] or if they require the processing of the whole event log before signaling the drifts [20]. All of these achieve the objective of identifying points in time where the process changed significantly, but present the following limitations:

(i) Require some a-priori knowledge of the nature of the drifts.

(ii) Data structures used limit the characteristics of the drifts they can identify.

(iii) Perform poorly with real-world datasets.

(iv) Rely on process discovery techniques, which delays the speed of detection and adds extra weight over the system requirements.

To address these limitations, this thesis proposes a new unsupervised-learning online concept drift detection framework. The goal is to provide a tool which is accurate in detecting drifts from both artificial and real-world datasets, while also being efficient from a time and memory perspective. The framework relies on the previous work [8] which uses a unique data structure to store and process an event stream. We attempt to improve such work by applying well-known drift detection techniques over this data structure rather than relying on a process model discovered. To prove this, several experiments were done with a set of artificial logs used in the literature [10] and a set of real-world logs from a well-known business process challenge. Each of them containing a variety of drifts from different characteristics. They were all tested in an online fashion by generating an event stream out of the event logs, or using an external tool which generates an artificial event stream out of a process model [21]. A data exploration procedure was performed to define the best parameters to use in different scenarios. Furthermore, the framework provides characterization of the drifts in order to answer the question of "What caused the change?". The outcome obtained was compared against three state-of-the-art techniques. This showed promising results of improvement in all aspects. To conclude, this thesis addresses three main research questions:

1. What concept drift detection framework would be able to address the limitations aforementioned?
2. Can the proposed framework outperform existing unsupervised-learning online concept drift detection techniques over event streams?
3. Can the proposed framework provide insights regarding the process besides the time at which the drifts happened for real-world scenarios?

Chapter 2

Background

This chapter will introduce the theoretical background and literature required to understand the framework proposed in this thesis. Section 2.1 will give an overview of process mining and related areas of such field. Later, Section 2.2 will focus on describing two stream generation tools used for the development and experimental phase of the project. Finally, Section 2.3 describes the related and reference work.

2.1 Preliminaries

2.1.1 Process Mining

Process mining serves as a bridge between data mining and business process modeling and analysis. The idea of process mining is to discover, monitor and improve real processes (i.e., not assumed processes) by extracting knowledge from event logs readily available in today's information systems. As stated in the Process Mining Manifesto [4], process mining techniques offer means to a more rigorously check compliance and ascertain the validity and reliability of information about an organization's core processes. The three main types of process mining are:

- *Process Discovery*: Extracts a process model from the event log without using a-priori information.
- *Conformance Checking*: Monitoring deviations between a model and the event log.
- *Process Enhancement*: Extend or improve an existing process model with information recorded in the event log.

Event data is basis for all process mining analysis. Most process mining techniques assume that their input is in the form of an event log. Following are the formal definitions of all components of an event log as taught in [22].

Definition 2.1.1 (Event) *An event $e \in \varepsilon$ describes that a specific discrete observation has been made (by a sensor, a system, a human observer, etc.). The observation itself is described by attribute-value pairs through the partial function $\pi : \varepsilon \times AN \mapsto Val$. For each event $e \in \varepsilon$ and each attribute name $a \in AN, \pi(e, a) = v$ defines the value v of attribute a. We write $\pi(e, a) = \perp$ if attribute a is undefined for e (has no value). We also write $\pi_a(e) = v$ or $e.a = v$ for $\pi(e, a) = v$. In the literature, you also find the notation $\#_a(e) = v$. For each event e, the attribute time is defined, i.e., $\pi_{time}(e) \neq \perp$. Further, each event e carries a value $\pi_a(e) \neq \perp$ for some other attribute $a \in AN$.*

We use the term *Case* to refer to an *entity* or *object* that we are "tracking" over time in terms of the events in which this entity is involved. To obtain a structured event log from an event table, we have to recognize from all attribute names the entity types, and then select one of these

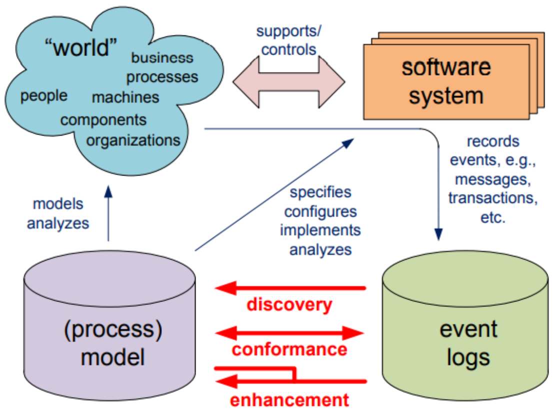


Figure 2.1: Overview of the three main types of process mining [4].

attribute names as the *case identifier* attribute. The attribute values are the different cases under this case identifier.

Definition 2.1.2 (Case) *Let $ET = \langle e_1, ..., e_n \rangle$ be an event table. The set of attribute names in ET is*

$$AN(ET) = \{a \in AN \mid \exists e_i \in ET, \pi_a(e_i) \neq \perp\} \tag{2.1}$$

If we select an attribute $id \in AN(ET)$ as case identifier, then

$$Cases(ET, id) = \{\pi_{id}(e_i) \mid e_i \in ET\} \tag{2.2}$$

Note that Definition 2.1.2 allows to pick any attribute as case identifier, not just those that refer to entity types. It is implicitly required that each event e has *three* mandatory attributes that are different from each other:

1. the *timestamp* $\pi_{time}(e)$
2. a recorded action or *activity* $\pi_a(e)$
3. a *case identifier* $\pi_c(e)$

If an attribute id is selected as case identifier, then it is said that an event e is *correlated* to a case c if its id-attribute refers to c, i.e., $\pi_{id}(e) = c$. A *trace* is the sequence of events correlated to a case and ordered by time.

Definition 2.1.3 (Trace) *Let $ET = \langle e_1, ..., e_n \rangle$ be an event table. Let $id \in AN(ET)$ be the selected case identifier.*

A sequence $\langle e_1, ..., e_n \rangle$ of events is a trace of a case $c \in Cases(ET, id)$ iff

1. *$\{e_1, ..., e_k\}$ = corr(ET,id,c), i.e., it consists of all events of ET correlated to c, and*
2. *for each i = 1,...,k-1 holds $\pi_{time}(e_{i+1}) = \pi_{time}(e_{i+1})$, i.e., events are ordered by time.*

A structured event log is a set of cases where each case is associated with exactly one trace for this case as a case attribute.

Definition 2.1.4 (Event Log) *Let $ET = \langle e_1, ..., e_n \rangle$ be an event table. Let $id \in AN(ET)$ be the selected case identifier.*

The structured event log L is the set L = Cases(ET,id) of cases for case identifier id so that additionally each case $c \in L$ gets assigned a trace $\langle e_1, ..., e_n \rangle$ of c as trace attribute $\pi_{trace}(c) = \langle e_1, ..., e_n \rangle$.

Timestamp	Case Identifier	Activity	Resource
20/12/2018 11:02	23	Receive Order	System
20/12/2018 12:09	23	Pack Order	Alice
20/12/2018 12:10	23	Add Item	Bob
20/12/2018 12:15	23	Ship Parcel	Charles
20/12/2018 11:03	41	Receive Order	System
23/12/2018 23:11	41	Pack Order	Bob
23/12/2018 23:49	41	Ship Parcel	Charles
20/12/2018 12:13	36	Receive Order	System

Table 2.1: Sample structured event log.

A structured event log has a simple hierarchical structure. At the top-level are the cases $L = \{c_1, .., c_l\} = Cases(ET, id)$. Each case has case attributes as "children", one of them is the trace $\pi_{trace}(c)$. Each event e in a trace has event attributes as children, including $\pi_{time}(e)$ (timestamp), $\pi_a(e)$ (the observed activity), and $\pi_c(e)$ (the case identifier). This hierarchical structure is formalized in the XES-standard[1]

2.1.2 Stream Process Mining

Advances in information technology have lead to the generation of large flows of data. These large volumes can be mined for interesting and relevant information in a variety of applications. This same volume component leads to a number of computational and mining challenges according to [23]: with increasing volume of data, it is no longer possible to process the data efficiently by using multiple passes. Rather, one can process a data item at most once. This leads to constraints on the implementation of the underlying algorithms. Moreover, there is an inherent temporal component to the stream mining process. This is because the data may evolve over time. This behavior of data streams is referred to as *temporal locality.* An adaption of one-pass mining algorithms may not be an effective solution to the task. The stream mining algorithms need to be designed with a focus on the evolution of the underlying data.

In [24] a data stream is defined as an unbounded sequence of data items with very high throughput. In addition, the following assumptions are typically made regarding the data stream:

- Data is assumed to have a small and fixed number of attributes
- Mining algorithms should be able to process an infinite amount of data, without exceeding memory limits or otherwise fail, no matter how many items are processed
- For classification tasks, data has a limited number of possible class labels
- The amount of memory available to a learning/mining algorithm is considered finite, and typically much smaller than the data observed in a reasonable span of time
- There is a small upper bound on the time allowed to process an item (one pass of the data)
- Stream "concepts" are assumed to be stationary or evolving.

An event stream then would be a specific type of data stream, where each data point refers to an event generated by an underlying process or system.

[25] argues that due to these requirements, algorithms for data stream mining are divided into two categories: data and task-based. Data-based techniques refer to summarizing the whole (Sampling, load shedding and sketching) or a subset (Synopsis data structures and aggregation) of the dataset from the incoming stream to be analyzed. Task-based techniques are those methods which modify existing techniques or invent new ones in order to address the computational

[1]XES Standard can be found at `http://www.xes-standard.org/`

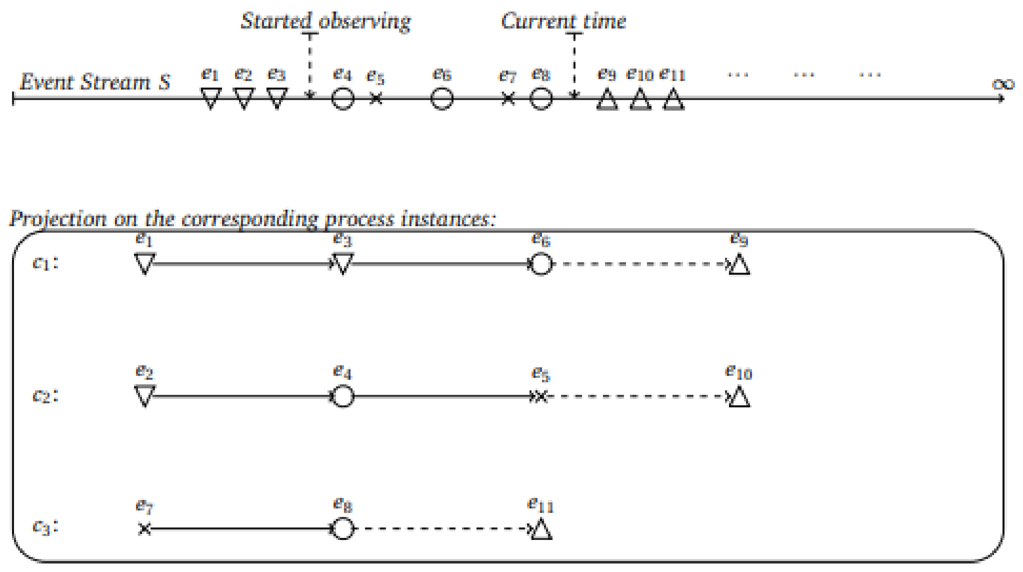

Started observing
Current time
Event Stream S
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9 e_{10} e_{11}
... ...
∞
Projection on the corresponding process instances:
c_1:
c_2:
c_3:

Figure 2.2: Schematic overview of the challenges of handling streaming data in the context of process mining. A ▽-symbol indicates an unobserved event, i.e. executed prior to observing the stream, a O-symbol indicates a stored event, a ×-symbol indicates a dropped event, i.e. it is not stored, and, a △-symbol indicates a future event [5]

.

challenges of data stream processing. Approximation algorithms, sliding window and algorithm output granularity represent this category. In the context of this thesis, we are working with task-based techniques which utilize a sliding window to process the event stream. More details on how the technique works will be given in Chapter 4

2.1.3 Event Stream Characteristics

In Subsection 2.1.2 several details on how a data stream is defined were described. Knowing that we are working with a specific type of data stream called *event stream*, it is important to define what are the components of the streams this framework focuses on. The characteristics which the framework presented in this thesis project can handle are:

- **Multidimensional stream of events:** Each event arrives with three components of information as described earlier (c, a, t). Therefore, the algorithm need to handle a multivariate data stream.

- **Parallel running cases:** Systems often generate a stream of events while running multiple cases at the same time (think of a purchase-to-pay scenario where each order is in a different step of the process, e.g. Create Order, Send Order, Receive Goods, Receive Invoice, etc.). We should not assume then that each event received will be corresponding to the same case. There may be multiple cases executing at the same time and finishing at different intervals as well.

- **Small alphabet of activities:** In a real-life scenario, a process would not contain an infinite set of activities. Though new activities may be introduced at any point in time, the alphabet of these is finite and usually known by the process owner and parties involved.

An algorithm which could cope with these set of requirements had to be designed, while also considering an efficient memory usage and processing time.

2.2 Event Stream Generation Tools

Burratin, A. [21] argues that researchers and practitioners frequently have to face the general data availability problem which could be decomposed into several research challenges:

1. Build large repositories of randomly created process models with control-flow and data perspectives.
2. Obtain realistic (e.g. noisy) multi-perspective event logs, which are referring to a model already known (i.e. the gold standard), to test process mining algorithms.
3. Generate potentially infinite multi-perspective streams of events starting from process models. These streams have to simulate realistic scenarios, e.g. they could contain noise, fluctuating event emission rates, and concept drifts.

While some of there are required in the testing phase to avoid over-fitting (tailored approaches that perform well on particular data, but lack in abstraction), others are key due to the emerging importance of big data analysis.

This section will present two tools available which allowed the generation of an event stream out of an event log itself, or business process model in certain formats. These were utilized during the complete development and testing phase, as well as through the experimental evaluation.

2.2.1 PM4PY

PM4PY[2] is a software product, developed by the process mining group of Fraunhofer Institute for Applied Information Technology (FIT). It includes a set of open-source Python libraries focused on different aspects of the three main types of process mining (2.1.1) and more.

One of the packages available, which supports streaming process mining, includes a module for importing an event log in both XES and CSV and generating an event stream. The importer allows to import the log in a trace-by-trace or an event-by-event format. A trace-by-trace would allow an analysis focused on cases, where each case would be processed completely before the next case arrives. This works well with algorithms that are focused on handling only close-ended traces, meaning that the whole sequences of events for that case has been seen already. Though it may be useful in some scenarios, our approach focuses on handling event streams with parallel running cases. This means that it is unknown at which point in time a case will finish and there is no certainty that it will ever do during the time of observation. An event-by-event importing, on the other hand, would create an iterator which extracts a single event within a trace based on the timestamp (time of arrival) in a chronological order. The iteration then would disregard if the next event seen is from the same or a different trace. This mimics in a better way how a real-world system generates a stream of events.

While iterating over the stream events, the importer extracts a set of attributes from the event log and makes them available in a dictionary per event. An example list of default attributes provided per event are:

- *id* (unique identifier)
- *concept:name* (activity label)
- *time:timestamp* (time-stamp)
- *case:id* (case unique identifier)
- *case:concept:name* (case label)
- *org:resource* (event resource)

Besides these, if any, the importer will provide the set of all case and event attributes. The iterator will go through all events within the event log and stop once the whole log has been processed. In a real-life scenario, this stoppage is assumed never to happen. This tool allowed the conversion of multiple real-life and artificial event logs into event streams.

[2]Details about PM4PY and full documentation available at `https://pm4py.fit.fraunhofer.de/documentation`

2.2.2 PLG2

PLG2[3] [21] is a standalone open-source Java application which utilizes a set of Application Programming Interfaces (APIs) which is able to:

- Generate random process models with additional data perspective (or import existing ones).
- Have detailed control over the data attributes (e.g. control values via scripts).
- Have detailed control over the time perspective (controlled via scripts).
- Evolve a process model, by randomly changing some of its features (e.g. adding/removing/replacing subrpocesses).
- Generate a realistic multi-perspective event log, with executions of a process model and noise addition (with probabilities for different noise behaviors).
- Generate a stream of multi-perspective events referring to process models that could change over time with customizable output ratio.

The last point is of special interest in the context of this thesis, as it allows the execution of an event stream generated out of a process model (either randomly generated or provided). Not only that, but such model can be updated during the stream execution to mimic a "change" in the process, and therefore introducing a concept drift. This thesis will not cover aspects related to the data-objects components of the events mentioned in [21]. Such are a dimension of process mining and concept drift detection that this framework is not focused on. We will cover two functionalities of the tools which were of use during this thesis project.

Random Generation of Business Processes

The generation of business process models is a useful tool for testing and describing different scenarios of interest in process behavior. PLG2 incorporates different well known workflow control-flow patterns that help to better describe a process. Such patterns are:

- *sequence*: direct succession of two activities ($a > b$).
- *parallel split*: parallel execution of activities (both a and b are forked into parallel branches, each executing concurrently).
- *synchronization*: synchronization of parallel branches (only after both a and b have both been executed may the process flow continue).
- *exclusive choice*: mutual exclusion of activities (either a or b executes, but not both).
- *simple merge*: convergence of branches (after a split by an exclusive choice, the branches merge to continue the flow).
- *structured loop*: execute sub-processes repeatedly (after executing a, there's another process branch that requires the re-execution of that same a).

Important to note that these patterns do not describe all possible behaviors that could be modeled in reality. These all are used in a combination to build a complete process. [21] describes the set of rules used to combine such patterns. After creating the desired random process model, users are able to use it for generating an event log or simulating an event stream.

[3]Application and documentation available at `https://github.com/delas/plg`

Noise Addition

For the data to be more realistic, noise can be introduced into the event log or event stream. This would simulate real-life scenarios where some deviations from the "expected" flow of the process occur. PLG2 provides a module to add noise both to a generated event log or a simulated event stream. It is able of applying noise at three different "levels": *trace-level* (noise related to the execution of the trace), *event-level* (noise that involves events on the control-flow perspective), and *data-object-level* (noise associated to the attributes stored within the event). The noise generation is driven by a set of parameters where the probability of applying a particular type of noise is defined. If all values are set to zero, it implies that traces have no noise. Examples of such noise phenomena are:

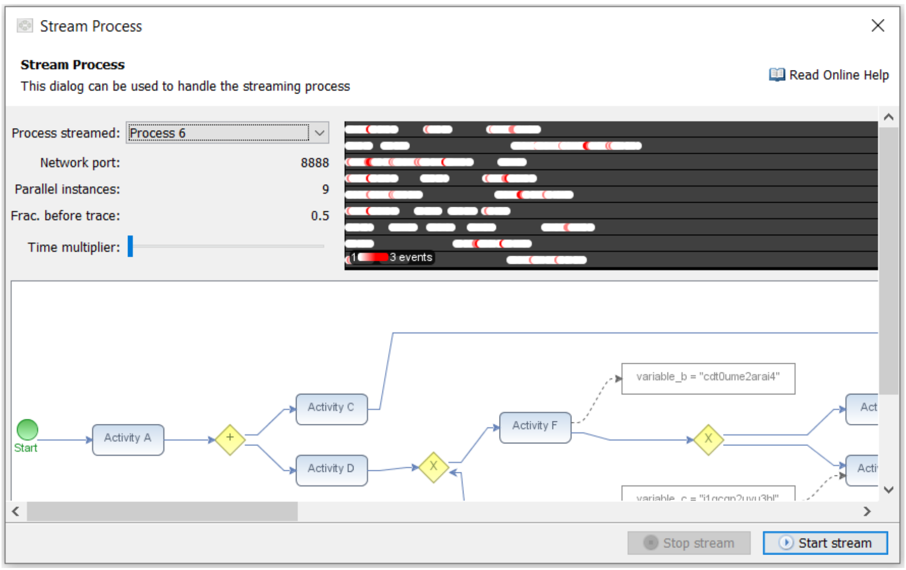
- Trace missing *head*. This means that the first events of the trace are not present. User must also define the size of the head.
- Trace missing *tail*. This means that the last events of the trace are not present. Again, the user must define the size of the tail.
- Trace missing *episode*. This means the trace misses a sequence of contiguous events. Again, user must specify the size of the episode.
- Alien event introduced in a random position with random attributes.
- Doubled event on the trace.
- Change of an activity name of an event. This means that an event which normally would have a label 'X' now receives a random label 'Y' instead.
- Change in the order between two events of a trace. This involves changing the time-stamps two events to represent an incorrect order of execution.

As explained previously, the *data-object-level* noise will be excluded from this thesis. All these noise introduction capabilities provide a very complete setup to generate testing scenarios of process behavior. When working with real-life systems, issues may arise which produce loss or incorrect data. This would result on deviations from the "gold-standard" model. Therefore, such need to be introduced within the experimental evaluation to make sure the algorithm is able to handle them properly, without resulting in inaccurate outcomes.

Stream Simulation

The stream simulation component of the tool creates a socket, which runs in your local environment, that accepts connections from external clients. PLG2 then "emits" (i.e. writes on the socket) events which are generated from the process selected. This would continuously send events for a potentially infinite amount of time (until the script is stopped). The tool asks for two parameters in order to run the simulation: maximum number of parallel instances running at the same time, and the "time scale". First one populates the data structures required for the generation of the stream so that multiple cases can be "running" in parallel. Second one allows the users to scale the emission time to simulate a real-time system emission. Values expected for the time scaling are defined in $(0,\infty]$. While the simulation is running, the user can change the process for the simulation without stopping the current stream emission. This would generate the concept drift mentioned previously.

Figure 2.3 shows the "Stream Process" window which has the process selected, network port for emission, number of parallel instances, the time-scale adjustment selected (which changes the emission rate for the events), and a process model diagram from the process selected. Once the user starts the stream, a connection must be made to the defined port to "receive" the event stream data. Listing 2.2.2 shows an example XML generated by PLG2 and received through a Telnet connection. The XML shows details of a single event emitted by the tool, which includes:



Stream Process

Stream Process

This dialog can be used to handle the streaming process

Process streamed: Process 6

Network port: 8888

Parallel instances: 9

Frac. before trace: 0.5

Time multiplier:

Figure 2.3: Stream Process window in PLG2 showing details for starting the stream emission of a process model.

two "concept:name" attributes (higher-level for the case label and lower-level for the activity label) and a "time:timestamp" attribute (event time-stamp) with their corresponding values.

```
<org.deckfour.xes.model.impl.XTraceImpl>
  <log openxes.version="1.0RC7" xes.features="nested-attributes" xes.version="1.0"
    xmlns="http://www.xes-standard.org/">
    <trace>
      <string key="concept:name" value="case_9"/>
      <event>
        <string key="concept:name" value="Activity Q"/>
        <date key="time:timestamp" value="2021-07-20T12:32:30.433+02:00"/>
      </event>
    </trace>
  </log>
</org.deckfour.xes.model.impl.XTraceImpl>
```

Listing 2.1: XML example generated from PLG2 stream simulator

2.2.3 Concept Drift Detection (CDD)

Previously mentioned algorithms for data stream mining have different applications within process mining. One important application over event streams which has been recently introduced is related to change/drift detection. As stated in [26], processes can change with respect to three main process perspectives: control-flow, data, and resource. Such changes introduce a drift in the concept (process behavior). This for example can be related to the way in which activities are executed (when, how, and by whom). The basic premise in handling concept drifts is that *the characteristics of the traces before the change point differ from the characteristics of the traces after the change point.* The problem is then to identify the points in time when the process has changed (if any). [26] argues that it involves two primary steps:

1. Capturing the characteristics of the traces.
2. Identifying when these characteristics change.

This thesis will only focus on those process changes related to the control-flow perspective. This perspective of a process characterizes the relationships between activities. Such relationships or dependencies are captured and expressed using *causal footprints*, which implies that an activity follows or precedes another one. Change detection is done then by analyzing a series of successive populations, which can be generated by splitting your event stream with a sliding window approach for example. A comparison is then done between two successive populations to investigate if there is a "significant" difference between them. *Significant* becomes then an important word, as each change detection technique defines itself what becomes relevant for them to detect a change.

According to [6] we can classify changes into *momentary* and *permanent*. While the momentary changes are short lived and affect only a very few cases, permanent changes are persistent and stay for a while. In the context of this thesis we focused solely on permanent changes, as momentary changes often cannot be discovered because of insufficient data. This leads us to four different classes of drifts:

1. *Sudden drift*: correspond to a substitution of an existing process with a new process. In other words, all cases from the new process emanate on an instant.

2. *Gradual drift*: refer to the scenario where a current process is replaced by a new process. Unlike sudden drift, here both processes coexist for some time, where the first one is gradually discontinued.

3. *Recurring drift*: correspond to a scenario where a set of processes reappear after some time (substituted back and forth). They show somewhat a seasonal influence or behavior, where this recurrence may be periodic or not periodic.

4. *Incremental drift*: refer to the substitution between two processes done via smaller incremental changes. They normally happen within processes that are undergoing sequences of quality improvement initiatives.

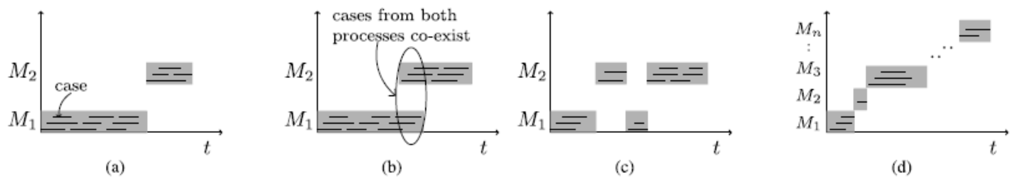


Figure 2.4: Different types of drifts. x-axis: time. y-axis: process variants. Shaded rectangles: process instances. (a) Sudden drift. (b) Gradual drift. (c) Recurring drift. (d) Incremental drift [6]

.

While it would be beneficial that all change detection techniques were able to identify the four types of drifts, most techniques are adapted to work for one or two of these. This due to the characteristics of the log which each technique is designed to track. While some focus on the distribution of the population's features (mean, variance, etc.), others use principal components or so to track some other relevant features. The technique presented within this thesis focuses on identifying sudden drifts.

The most well-known change detection techniques are the Page-Hinckley Test [27], which computes the observed values and their mean up to the current moment and will detect a concept drift if the observed mean at some instant is greater than a threshold value lambda; Cumulative Sum (CUSUM) [27], which uses the cumulative sum of a parameter of the probability distribution (e.g. the mean) to investigate whether a sequence of these can be modeled as random; and Adaptive Windowing (ADWIN)[3], which is an adaptive sliding window of variable size that keeps updated statistics about the data stream. The algorithm will decide the size of the window by cutting the statistic's window at different points and analyzing the average of some statistic over these two windows. If the value of the difference between the two averages surpasses a pre-defined threshold, a change is detected. There are far more methods developed which utilize multivariate data streams. These will be covered further in 2.3

2.2.4 Supervised vs Unsupervised Stream Learning

Similar to data mining, stream learning techniques can also be split into supervised and unsupervised learning. While supervised algorithms require some apriori knowledge of the event log to define a class label (e.g. prediction of the next activity), unsupervised algorithms are used without any external knowledge and rely solely on characteristics of the stream as it is being processed.

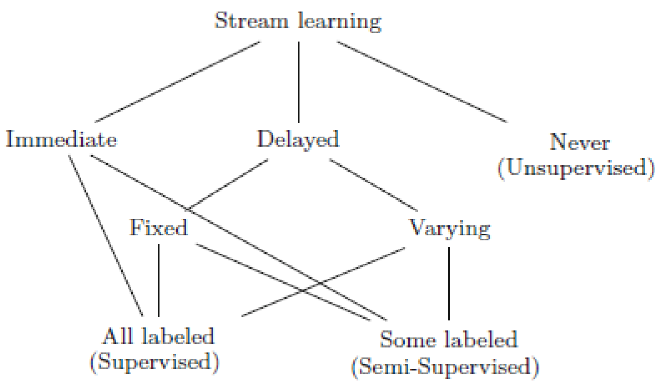


Figure 2.5: Stream learning according to labels arrival time [7]

.

The problem with supervised learning approaches according to [28], is that they assume that the true labels are readily available after the prediction. In reality, this is rarely the case. In most real-world scenarios where a process generates these labels, they can take too long to be available, if ever. A predictive model is built using an initial batch of training data, whose labels are available. This model is then deployed with a test set and CDD is carried out in an unsupervised or supervised manner. In contrast, unsupervised learning does not require labels to build a model. Drift detection is carried out using different strategies than monitoring the prediction loss. They track, for example, the output probability of models or the unconditional probability distribution. The next section will provide an overview of novel CDD techniques from both supervised and unsupervised learning approaches.

2.3 Related Work

2.3.1 Supervised-Learning CDD Techniques over Data Streams

As mentioned previously, supervised learning approaches require a true label to be provided by the event log or an external knowledge entity just after data arrives. There is extensive work done in the data stream field focused on developing supervised learning techniques. Most methods are focused on tracking the performance of the classifier over time. When there is a significant drop or increase in the loss of the prediction, a drift is signaled.

The Drift Detection Method (DDM)[29] monitors the error rate of the base learner. The method assumes that the error rate will decrease while the number of instances increases if the data distribution is stationary. The algorithm relearns a new model using only the stored instances since the warning state was reached. The Early Drift Detection Method (EDDM)[30] estimates the average distance between two adjacent errors and its standard deviation at a specific. Again, if a drift is detected the algorithm relearns a new model from the stored instances. The Fast Hoeffding Drift Detection Method (FHDDM)[31] uses a constant sized sliding window to estimate the accuracy of the predictive model and keeps the maximum accuracy until current time. A drift is isgnaled when the difference between current accuracy and maximum accuracy is greater than a threshold determined by using Hoeffding's inequality. The Accurate Concept Drift Detection Method (ACDDM)[32] detects concept drifts by analyzing the status of prequential error rates using the Hoeffding's inequality to observe inconsistency of error rates. It will signal possible concept drifts based on the current error rate of the base learner. The Hoeffding Adaptive Tree (HAT)[33] method places instances of estimators of frequency statistics at every node of a Hoeffding Tree. Each of these detectors will raise a flag if the statistics within the node changes. It uses a similar sliding window approach as ADWIN. In [34] drift is measured by continuously tracking Total Variation Distance (TVD) metric between probability distributions estimated from two sample windows preceding a time point. High values for the difference metric indicates that concept drift has occurred, and the model must be adapted. Adaptation is done by training a new model for the drifted process, and adding it to an ensemble of models. Previously trained models are retained, and their weights in the ensemble are adjusted to reflect similarity with the current probability distribution of the process. The Adaptive Random Forest (ARF)[35] algorithm is an adaptation of the well-known Random Forest algorithm. It combines the batch algorithm traits with dynamic update methods to deal with evolving data streams. It applies a theoretically sound re-sampling method based on online bagging and an updated adaptive strategy based on using drift monitors per tree to track the drifts and also train new trees in the background before replacing them. The Self-Adjusting Memory Ensemble (SAM-E)[36] combines the advantages of the Bagging-ensembles with the SAM algorithm to boost performance further. These SAMs ensemble deal with concept drift by means of the local adaptation of its base models. The drift detection replaces unsuitable base learners, which does not increase the adaptation speed of the overall model. It is implemented in parallel to optimize performance and facilitate comparison.

2.3.2 Unsupervised-Learning Process Mining Algorithms for Concept Drift Detection

The focus of this thesis is over unsupervised learning techniques for data streams. Unsupervised window-based change detection, according to [18], is based on comparing the distribution in a current stream window with a reference distribution, where density estimation techniques and divergence metrics are essential to model and compare the distributions. Modeling the data distribution becomes a challenging when working with multidimensional data. Also, as data dimensionality increases, techniques become more inaccurate and the computational costs increase as well.Unsupervised-learning algorithms used for concept drift detection in Process Mining can further be classified into two classes depending on how they handle the event log: online and offline [11].

Offline Analysis

Offline analysis refer to those scenarios where the presence of changes need not to be uncovered in real time. Therefore, the detection of changes is mostly used postmortem. These results are normally used when designing or improving processes for a later deployment. Many techniques of this branch process the complete event stream or event log before signaling a drift or change.

The approach in [11] extracts a set of relation-based features from the event stream, over which the J measure of dissimilarity between the *a priori* and *a posteriori* frequencies is performed and transformed into a continuous dataset. This is then split into multiple populations to perform statistical hypothesis test over two adjacent ones. They propose to use the Kolmogorov-Smirnov and Mann-Whitney U test of independence for univariate data, and the two sample Hotelling T^2 test for multivariate data. The work from [11] was later extended in [9] to analyze gradual drifts and multi-order dynamics. The logic remains the same, but now applying a sliding window that would compare not only adjacent populations, but also non-continuous populations provided a proper choice of a gap. Furthermore, they would increase the sizes of the windows so that they're able to identify *macro-level* changes which a smaller window size would not be able to identify. The Tsinghua Process Concept Drift Detection (TPCDD)[20] approach represents each trace into multiple relation features. The variation trend and partition of each relation is inspected to then combine all change points revealed and get a final result. It utilizes a relation matrix over which a minimum relation invariance distance (MRID) threshold is applied, which helps identify the candidate change points. DBSCAN is later used to create the clusters of candidate change points and identify the actual drifts. The Visual Drift Detection (VDD)[12] technique applies a set of behavioral constraints called "DECLARE" constraints. Such produce a continuous dataset of values that are clustered based on their trends of changes over time. A kernel cost function is applied over the matrix of each cluster, which throws he set of drifts identified per cluster. These drifts are compared and visually analyzed to define the global and local concept drifts. The Robust and Accurate Approach[37] extracts the frequencies of the directly-follows relations from the log. It will then build a contingency matrix with such information from a reference and test window of a fixed window size and perform a G-test of independence over it. After a certain number of consecutive rejected tests, it will signal a drift. The algorithm also processes the log forwards and backwards. Such procedure, authors argue, helps identify the deletion of events as drifts as well.

Online Analysis

Online analysis refer to the scenario where changes are discovered in near real time. It is appropriate for situations where organizations are interested in detecting a change at the moment when it happens. Such algorithms trigger alarms and enable organizations to take quick remedial actions and avoid repercussions over the process. These are best adequate for event streams, as the drifts are signaled while the system is generating data. The approach from [13] combines conformance checking from Process Mining with Local Outlier Factoring (LOF) from Data Mining. It first requires either a reference process model or an initially discovered one out of the event log. It identifies deviating traces in the stream, which are classified within different microclusters based on their LOF score. Drifts are then identified over each of these independent microclusters and aggregated. In [14] a statistical G-test of independence is performed over distributions of special behavioral relations between events, as observed in two adjacent windows of adaptive size. The frequency of this relations is stored in a contingency matrix. Eventually, after a certain amount of consecutive rejected tests, a concept drift is signaled. Similarly, [10] performs a Chi-squared test of independence over a contingency matrix which stores the frequencies of a special type of feature extracted called *runs*. These allow to store more details over the relations, like concurrency or loops. Again, a filter is applied so that a concept drift is detected only after a certain given number of successive rejected statistical tests. In [18] a framework is proposed which applies Principal Component Analysis (PCA) to project the original multidimensional data stream into a lower dimensional space. Different types of changes in the data variables, such as changes in mean and variance or the correlations between variables, can be observed over one or several Principal Components (PCs). They then apply a change-score to each of these PCs and determine a final change-score by aggregating all selected components.

2.3.3 StrProM

As stated in Subsection 2.3.2, this thesis is focused on developing a new unsupervised data stream learning technique for concept drift detection. While several techniques were briefly explained previously, the biggest contributor to this master thesis was a technique developed by Dr. Ing. Marwan Hassani et.al. called *StrProM*[1]. This novel approach builds prefix-trees to extract sequential patterns (directly-follows) of events from the stream. Such trees are later used as a process discovery tool to generate process models using the Heuristics Miner algorithm[38]. The approach steps will be explained next, as it's the basis of a concept drift technique which this master thesis attempted to extend and improve. More details on each component of the algorithm, data structures, and processing steps will be given in 4.

Authors of this paper argue that several approaches collect counts of events and case-related directly-follows relations by observing the event stream and use this data as input for the Heuristics Miner algorithm. These approaches require different data structures to store information regarding the activities, relations and cases. After a new event is seen, all of these data structures must be updated, which becomes a costly operation on an infinite-stream scenario. Their approach aims to reduce this process time by decreasing the effort of updating three structures and utilize one

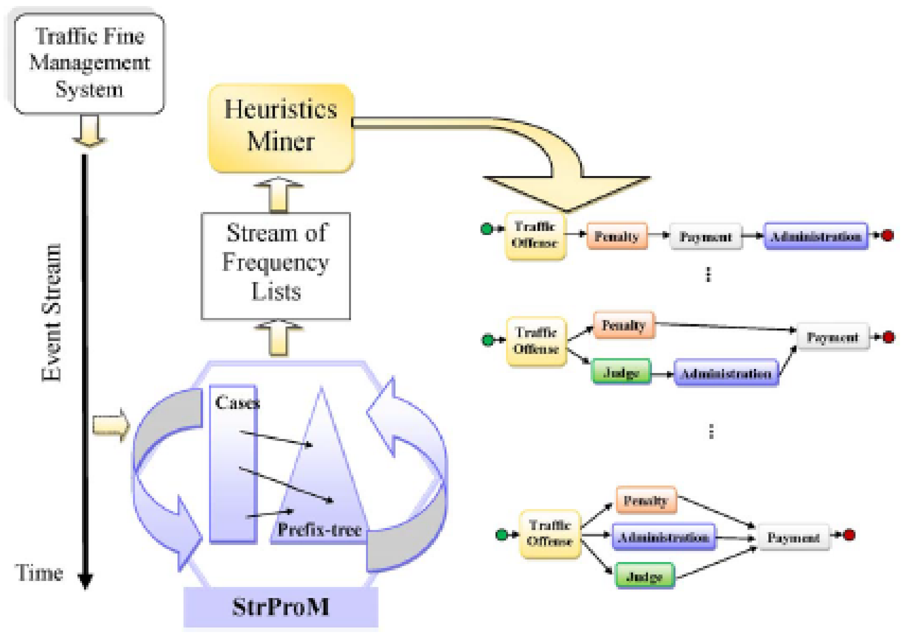


Figure 2.6: An overview of our StrProM algorithm using an example stream of events from a Traffic Fine Management System. The streaming process models (Right) present a drift in the models (causal nets) over the time with the activities: Traffic Offense, Penalty, Payment, Administration, Judge [1]

.

data structure instead, the prefix-trees. This would allow a much faster process time, but demand a more extensive processing of the structure to yield the frequencies of the tree entities. Because this information is not required as often while the stream progresses with new events, such step is delayed to decrease the average event processing time significantly.

After a certain period of time, the algorithm applies a pruning step to the tree's branches by eliminating branches with a support below a particular minimum support threshold. The purpose of this step is to store redundant information into a more compressed structure. Next, two frequency maps are used to store the information within the tree: one for the activities and another for the relations. For each node n with label a and its child m with label b, the value store in m is retrieved. This value is added to the relation frequency map for $a > b$ and to the activity a frequency map as well. Subsequently, once all the information from the tree has been extracted, this tree can be replaced to include only open cases. Each open case stores the last activity seen for that case. Once the tree is reset, a new node is attached for each open case to the root node with the last activity seen and a value of 0. This means that after the pruning step finishes, a tree of height 1 will be available to continue processing the event stream.

The algorithm also applies a decaying mechanism, where the information within a stream of events gets less importance as the data becomes older. In order to grant more importance to newer processes instances or cases, they "forget" older relations and activities. When the tree pruning happens and the new frequencies are added to both the activities and relations frequency maps, the old counts are modified using a decaying factor $\lambda \in [0, 1]$ with the follwing formula:

$$F_A^{new} = (1-\lambda)F_A^{old} + F_A^{extracted} \quad F_R^{new} = (1-\lambda)F_R^{old} + F_R^{extracted} \tag{2.3}$$

Where F_A refers to the frequency map of the activities and F_R refers to the frequency maps of the relations.

Another pruning step is performed whenever the amount of active cases being tracked exceeds the maximum number allowed C_{max}. When working with real-world datasets, the number of simultaneously open cases tends to grow very large, while the number of activities and relations remains smaller. Monitoring open cases they argue is the dominant factor considering memory consumption. Each open case stores the timestamp of the last event seen as well, which allows them to prune those open cases with the oldest timestamp ones C_{max} has been exceeded.

Eventually, once a certain amount of events have been processed, the frequency maps are sent as an input to the Heuristics Miner algorithm. Details on how this algorithm works can be found in Appendix A Algorithm 2. With this, a process model is continuously discovered and updated as the event stream continues. Experiments showed that StrProM achieved better results in the processing time per event and memory storage compared to its competitors, while still achieving similar results in fitness and precision of their models with respect to the event log. These continuous set of process models will serve as the basis for a concept drift detection technique described next.

2.3.4 StrProMCDD

After defining a novel process discovery algorithm, Dr. Ing. Marwan Hassani extended *StrProM* to the field of concept drift detection. The result was a new algorithm *StrProMCDD*[8], which utilizes the process models generated to perform a set of distance metrics over them and with this identify drifts within the event stream. Furthermore, it utilizes an adaptive window concept based on ADWIN[3], where the window keeps increasing as long as no change is detected and decreases once a new drift appears. Appendix A Algorithm 3 shows the detailed algorithm.

The algorithm works very similarly to *StrProM*, with the main difference being that the frequency maps generated after every pruning step are stored in a temporal ordered list. The old ones with be found at the beginning while the most recent ones are located at the end of the list. Once the maximum size of the ordered list (window) has been reached (W_{Max}), the algorithm follows the logic behind ADWIN and will perform multiplle cuts over the list of frequency maps. For each cut, the algorithm will create a mean aggregation of the lists of each sub-population and generate a process model out of these aggregated frequency maps. Figure 2.7 shows the adaptive window logic behind *StrProMCDD*. The idea behind is, as no change within the models has been identified, the window can continue to grow to process more events while the underlying distribution is stable. Once a drift has been identified the algorithm wants to be more susceptible to changes, therefore decreasing the size of the window to perform the detection procedure more frequently. More details on how ADWIN is implemented are explained in 4.

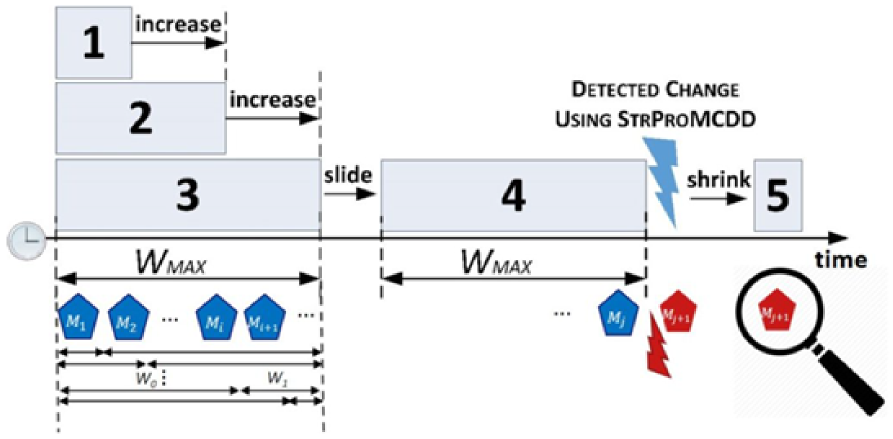


Figure 2.7: The applied adaptive window concept: the window keeps increasing as long as the underlying distribution does not change. Once it reaches its maximum allowed size, it starts sliding to catch new items by forgetting older ones. When StrProMCDD detects a concept drift, it shrinks its size to focus on the new trend [8]

.

Once the process models are built through every partition of the window, different distance functions are applied over both models and a threshold is defined to compare against. A drift is signaled if this threshold is exceeded. Such distances functions are: dependency distance, edge distance, activity frequency, relation frequency, routing distance, and relative importance. Results showed that certain distance functions performed better than others based on the dataset set that was selected. This work lead to different ideas over how to use *StrProM* for concept drift detection without the need of generating a process model. This set the ground for the work and problem definition of this master thesis.

2.3.5 Tree-Distance Metric

In order to apply a concept drift detection algorithm over a set of prefix-trees (with nodal attributes), a transformation of a discrete dataset into a continuous set is required. Such transformation has not only to use information stored within the nodes of the trees, but take advantage of the topology of the tree as well. The structure of the tree, in the context of a event stream, stores valuable information regarding the traces which becomes significant when working on detecting control-flow drifts.

In [2] a novel mathematical framework for statistical analysis of populations of tree-structured objects. Under the context of object oriented data analysis (OODA), finding a center point (e.g., mean or median) allows the use of Euclidean approaches, similar to the scenario we are presented with. They argue then that trees or graphs are seen important in medical image analysis, but because the data is strongly non-Euclidean, fundamental tools of standard vector space statistical analysis are not available. Motivation behind their framework was the recent developments in medical image analysis, in particular, analysis of a sample of blood vessel trees in a human brain. In this scenario, the data objects become curved and standard Euclidean approaches are very successful. This allows the use of methods like PCA, to analyze the variation regarding a reference point.

The theoretical contribution of the authors is the introduction of a new metric δ on tree space. Such metric consists of two parts: the integer part dI, which captures the topological aspects of the tree structure, and the fractional part f_δ, which captures characteristics of the nodal attributes. For any two topological binary trees s and t with nodal attributes, the Euclidean distance is defined as:

$$\delta(s,t) = d_I(s,t) + f_\delta^2(s,t), \tag{2.4}$$

where

$$d_I(s,t) = \sum_{k=1}^{\infty} 1\{k \in IND(s) \triangle IND(t)\} \tag{2.5}$$

and

$$\begin{aligned} f_\delta^2(s,t) = \sum_{k=1}^{\infty} &\alpha_k((x_{sk}-x_{tk})^2 + (y_{sk}-y_{tk})^2) \\ &\times 1\{k \in IND(s) \cap IND(t)\} \\ &+ \sum_{k=1}^{\infty} \alpha_k(x_{sk}^2 + y_{tk}^2) 1\{k \in IND(s) \setminus IND(t)\} \\ &+ \sum_{k=1}^{\infty} \alpha_k(x_{sk}^2 + y_{tk}^2) 1\{k \in IND(t) \setminus IND(s)\} \end{aligned} \tag{2.6}$$

Equation 2.5, as previously mentioned, captures the topological aspects of the tree structures. The symbol $\triangle$ is used to denote the symmetric set difference ($A \triangle B = (A \cap \overline{B}) \cup (\overline{A} \cap B)$, where $\overline{A}$ is the complement of A). $d_I(s,t)$ counts the total number of nodes which show up only in either s or t, but not both of them. It can also be seen as the smallest number of addition and deletion of nodes required to change tree s into t. Since d_I is always an integer, they referred to it as the *integer tree metric.*

Second part of the Eucledian distance is reflected in Equation 2.6. This captures, as mentioned before, the characteristics of the nodal attributes within the trees. Here, α_k is a non-negative weight series which allow intervention on the importance of various nodes. For the prefix-trees developed in this thesis project, all nodes have the same importance. Therefore, such weight is not utilized. The attributes contained in the node k of tree t are denoted by (x_{tk}, y_{tk}) (two node attributes). Therefore the first part of the equation takes the sum of square differences $(\sum_{k=1}^{\infty}(x_{sk} - x_{tk})^2 + (y_{sk} - y_{tk})^2)$ between attributes of the node intersection in both trees $(\times 1\{k \in IND(s) \cap IND(t)\})$. The last two summations in Equation 2.6 are used to avoid loss of information from those nodal attributes that are present in one tree and not the other. If performs the sum of the square of both attribute values in the nodes which are present in tree s and not in t and viceversa.

2.3.6 ADWIN

The unsupervised drift detection technique for multivariate data streams chosen for this thesis was ADWIN [3]. This nobel technique has been strongly used over concept drift detection for event streams in the literature due to its efficiency in time and memory requirements. Such algorithm has a simple idea: whenever two "large enough" subwindows of W exhibit "distinct enough" averages, once can conclude that the corresponding expected values are different, and the older portion of the window should be dropped. We touched on this point in Chapter 2 Section 2.2.3. ADWIN generates a threshold for a cut denoted as $\in_{cut}$. Such value is computed as follows: Let n_0 and n_1 be the length of W_0 and W_1 respectively. Let $\overline{\mu}_{W_0}$ and $\overline{\mu}_{W_1}$ be the averages of the values in W_0 and W_1, and μ_{W_0} and μ_{W_1} their expected values. The algorithm then defines:

$$m = \frac{1}{1/n_0 + 1/n_1} \tag{2.7}$$

as the harmonic mean of n_0 and n_1 and

$$\delta' = \frac{\delta}{n}, and \quad \in_{cut} = \sqrt{\frac{1}{2m} \times \ln \frac{4}{\delta'}} \tag{2.8}$$

The statistical test will then look at different distributions in W_0 and W_1 and check whether the observed average in both sub-populations differs my more than the threshold $\in_{cut}$. The distance metric computed previously then can be continuously fed to this technique which will signal if a drift is detected. Fortunately, authors of the technique also developed a set of libraries in Python as open-source and available to use under Scikit-Multiflow[4]. The package allows the creation of an *ADWIN* class object, which has a function *add_element* that takes as parameter the value to be analyzed. Later, another function called *detect_change* is used to detect concept drifts over this continuous set of values.

[4]Details about Scikit-Multiflow and full documentation available at `https://scikit-multiflow.readthedocs.io/`

Chapter 3

Problem Formulation

This chapter focuses on identifying areas of opportunity from the literature presented in Chapter 2. This will allow the definition of our research questions and justify the need of a new framework. In order to land these questions, a couple of preliminary knowledge is required.

We refer to the following string representation for event streams:

$$S = \langle e_1, e_2, e_3, ... \rangle \tag{3.1}$$

Let us note that an event stream is a potentially infinite sequence of events, where events are ordered by time and indexed. Furthermore, events of the same trace do not need to be consecutive in the event stream, i.e. traces can be "overlapping". We would then process each event with the following representation:

$$e = (c, a, t) \tag{3.2}$$

where c represents the case identifier, a refers to the activity label, and t to the time-stamp. Each event is unique, as the combination of the three attributes can never repeat. Though most stream generation tools provide a unique identifier for each event, such information is not needed for the purpose of this framework.

The prefix-trees, which will be covered in Chapter 4 are composed of nodes. Each node stores a set of attributes which will be represented as follows:

$$n = (i, a, p, pL, ch, f) \tag{3.3}$$

where i represents a random universal unique identifier for the node, a refers to the activity label, p represents the parent node or predecessor (referencing the directly-follows relation from 2.3.3, this would be equivalent to storing node a inside node b on the relation $(a > b)$), pL refers to the list of all parent nodes for this node (starting from the root node and up until the parent node. This same list could be represented as $pL = \langle p_0, p_1, p_1, ..., p_n \rangle$ where p_0 represents the root node and p_n the last parent node), ch represents a dictionary storing all the children nodes of this node (this dictionary can be represented as $ch = \{a : n\}$ where a refers to the activity of the children node and n to the node itself; each children node would be an entry in the dictionary), and f represents the frequency of this node (amount of times this sequence of events has been seen).

Cases are represented with the following notation:

$$c = (ci, n, fl) \tag{3.4}$$

where ci refers to the case identifier, n represents the last node seen for that case and fl a Boolean flag to note if the case is still active (open) or inactive (closed).

An extensive analysis was conducted to understand the capabilities and limitations of current concept drift detection techniques. Though there are several supervised-learning algorithms, such have a significant limitation in realistic settings: true label arrival. In order to train the model, a true label for event e_i at time t is required after predictions are made and before the next instance

2*Algorithm	Disc→Cont			Perspectives			Analysis		Window		Stream		Drift Type			
	Probability Function	Feature Extraction	Distance Function	Control-Flow	Data	Resource/Performance	Compare Distribution	Prediction	Dynamic	Fixed	Online	Offline	Sudden	Gradual	Incremental	Recurring
StrProMCDD [8]			✓	✓			✓		✓		✓		✓			
PCA-CD [18]	✓				∼		✓		✓		✓		✓			
DOA [13]			✓	✓			✓		✓		✓				✓	✓
ProM [11]		✓		✓			✓			✓		✓	✓	✓		
MOD [9]		✓		✓			✓		✓			✓	✓	✓		✓
UDD [14]			✓	✓			✓		✓		✓		✓			
ProDrift [10]		✓		✓			✓		✓		✓		✓	✓		
TPCDD [20]		✓		✓			∼		#			✓	✓			
VDD [12]		✓		✓			✓		#			✓	✓	✓	✓	✓
D3 [19]	✓				∼			✓		✓	✓		✓			
TC [15]		✓		✓	✓		∼			✓	✓		✓			
TBC [16]		✓				✓	∼			✓		✓	✓			
EMD [17]			✓	✓		✓	∼			✓		✓	✓			
Robust [37]		✓		✓			✓			✓		✓	✓	✓		

Table 3.1: State-of-the-art unsupervised concept drift detection techniques characteristics (# = not applicable, ∼ = Other).

e_{i+1} arrives. In most real-world scenarios, labels may arrive to late or not at all. Furthermore, in an online setting, the algorithms must identify the drifts in near real-time, i.e. the need for a label may then hinder the performance of the algorithms. Because of these reasons, an unsupervised-learning technique for detection changes seems a more adequate approach over event streams.

Table 3.1 shows a comparison of state-of-the-art unsupervised-learning concept drift detection techniques. Based on the previously mentioned reasons, it was decided to exclude supervised-learning techniques from the comparative analysis. Five main components were chosen to classify each of these techniques:

(i) *Disc → Cont*: Type of transformation technique used to convert the a discrete dataset into continuous. Some techniques use a a probability function or probability density function. Most of them focus on extracting the frequency of a set of features from the event log. A few others use a defined distance function over components of the event log (e.g., traces).

(ii) *Perspectives*: Aspects of the event log that the technique is designed to analyze. While most techniques focus on the control-flow because it allows to discover a process model out of the stream of events, others focus on data components stored in the events, the resources which execute the activities of the events, or the performance of the process itself.

(iii) *Analysis*: Type of statistical analysis done over the continuous dataset. Most techniques focus on comparing the distribution of two adjacent populations. Others rely on predictions and analyzing the error of their predictions to signal a drift. There are a few which apply other statistical techniques (e.g., clustering).

(iv) *Window*: Window approach used to observe the event stream. A fixed window size relies on some external knowledge of the log to define the correct size. Adaptive window approaches continuously update the size of the window based on the changes detected. Other approaches will process the event log completely in one run, i.e. using a single window of observation.

(v) *Stream*: Class of stream analysis approach. While offline techniques will provide the drift results after processing the whole log, online approaches signal drifts in near real-time while the log is being processed.

(vi) *Drift Type*: Type of drifts that the technique is focused on detecting. Algorithms may work with one ore more of these types, though very few argue that they're able to detect all types of drifts. Note that those techniques which did not specify the types of drifts that they could detect were assigned to "Sudden" drifts only.

These set of characteristics help us classify the algorithms to understand over which scenarios would each perform better. Though all of these techniques achieve the objective of identifying concept drifts over an event stream, they also show a list of limitations:

- Defining a set of feature vectors imply some a-priori knowledge of the possible nature of the drift. Therefore, these methods are not fully automated.
- Depending on the data structure utilized, many techniques dismiss of important information from the traces that can help identify drifts from the control-flow perspective.
- Many techniques perform efficiently with the logs used within their experimental evaluation, but performance decreases significantly with real-life event logs; even scenarios where they cannot process them at all.
- Some techniques rely on the creation of a process model before performing the drift detection. This step creates both a loss of information by transforming the data, as well as an extra time and memory requirement.

With this in mind, the need for a new unsupervised-learning concept drift detection framework appears. While developing the framework, a couple of questions aroused that guided the design of it:

1. What concept drift detection framework would be able to address the limitations aforementioned?
   (i) Which is the most optimal data structure for storing and processing a multivariate stream of events S?
   (ii) Which approach can transform the discrete data stored in prefix-trees into continuous?
   (iii) Which unsupervised drift detection technique, which utilizes an adaptive window, is the most efficient (time and memory) on identifying sudden and gradual drifts?
2. Can the proposed framework perform to the same level, if not outperform, the unsupervised state-of-the-art concept drift detection over event streams techniques?
3. Can the proposed framework provide insights regarding the process besides the known drifts (gold-standard) for real-world scenarios?

Chapter 4 will describe in detail the components of the framework which attempts to answer these questions.

Chapter 4

Method

This chapter presents a new framework for unsupervised-learning concept drift detection over an event stream. First, the procedure for initializing an event stream is shown 4.1. Then, the data processing and storage technique based on *StrProM* will be explained in detail in Section 4.2. Then, Section 4.3 introduces the method for converting the discrete data to continuous. Later, the concept drift detection technique implemented is described in Section 4.4. Finally, Section 4.5 presents the drift characterization component of the framework.

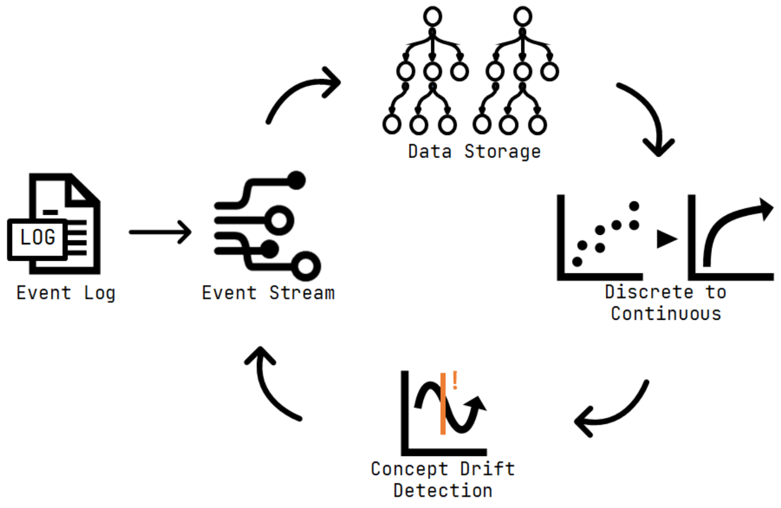


Figure 4.1: Prefix-Tree StrProMCDD framework.

Figure 4.1 shows a high-level representation of the framework which will be described in this chapter. Following are the steps which the framework takes in order to identify concept drifts out of an event stream:

1. An event log (or BPMN process model) is imported and metadata related to the end of each case is extracted,

2. An event stream is generated using one of the approaches presented in Chapter 2 Section 2.2,

3. The event stream will send continuously events into a listener until the complete log has been processed,

4. The algorithm will process each event and store it in a prefix-tree relative to the current window of observation,

5. Once the window is big enough, the data will be transformed into a continuous data set using a distance metric described in 2 Section 2.3.5,

6. A concept drift detection technique will process such continuous dataset and signal a drift if the distribution of the data has significantly changed within the window,

7. The process will continue repeat until the event stream stops the transmission of events.

The complete implementation of this framework can be found in the following Github repository.

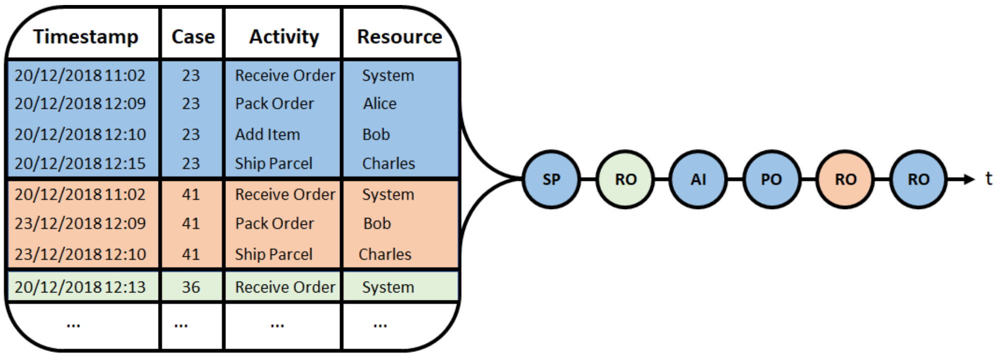
Algorithm 1: Log-Based Concept Drift Detection Algorithm over Event Streams

```
Input: log/model: input to generate event stream, W: adaptive window for drift
       detection, ADWIN: Concept drift detection technique
log ← $PM4PY.readEventLog(XESfile)$;
$eventStream = log.convertToStream()$;
OR        /* Depending if an XES log or BPMN model is used as input */;
$parameters = PLG2.streamParameters$;
model ← $PLG2.importModel()$;
$PLG2.startStream(model, parameters)$;
foreach event in eventStream do
    $insertByEvent(event)$;
    if W.cddFlag = True then              /* Concept Drift Detection activated */
        if len(W.PrefixTreeList) = W.WinSize then    /* Max window size reached */
            $W.ConceptDriftDetection(ADWIN)$;
            $W.WinSize = MIN(W.WinSize + 1, W.MaxWindowSize)$;
            if len(W.PrefixTreeList) = W.WinSize then       /* No drift detected */
                $W.PrefixTreeList = W.PrefixTreeList.dropLast()$;
end
```

Algorithm 1 shows the pseudo-code for the execution of the framework. Note that several functions used within the algorithm are shown in Appendix A.

4.1 Creating and Initializing an Event Stream

First step of the framework is the generation of an event stream out of a log. Important to note that this framework assumes that the input data is either: a process model, which can be used to randomly generate an event stream with PLG2 2.2.2, or an XES file, which PM4PY 2.2.1 would tranform into an event stream. Both scenarios will be explained in this section.



Timestamp	Case	Activity	Resource
20/12/2018 11:02	23	Receive Order	System
20/12/2018 12:09	23	Pack Order	Alice
20/12/2018 12:10	23	Add Item	Bob
20/12/2018 12:15	23	Ship Parcel	Charles
20/12/2018 11:02	41	Receive Order	System
23/12/2018 12:09	41	Pack Order	Bob
23/12/2018 12:10	41	Ship Parcel	Charles
20/12/2018 12:13	36	Receive Order	System
...	...	...	...

Figure 4.2: Log to stream conversion example.

4.1.1 PLG2 to Event Stream

In Subsection 2.2.2 a detailed description of PLG2 and its capabilities was given. One capability not previously mentioned was the ability to import business process models into the tool. This allows the users to generate an artificial log or event stream out of an existing process model. The tool allows two types of process model formats: BPMN[1] model and PLG file. Figure 4.2 shows an example end result of this step, were the log from Table 2.1 is converted into a stream of events ordered by time of arrival.

BPMN is an acronym for Business Process Model and Notation. It is a well-known graphical representation of a business process, which has been maintained by a set of organizations over the years. Very similar to a flowchart diagram, it provides a user-friendly notation which is also able to represent complex business semantics. There are several tools in the market which provide an interface to create BPMN models or import existing models in other formats, like Signavio[2]. An example BPMN model for an accounts receivable process can be seen in Figure 4.3.

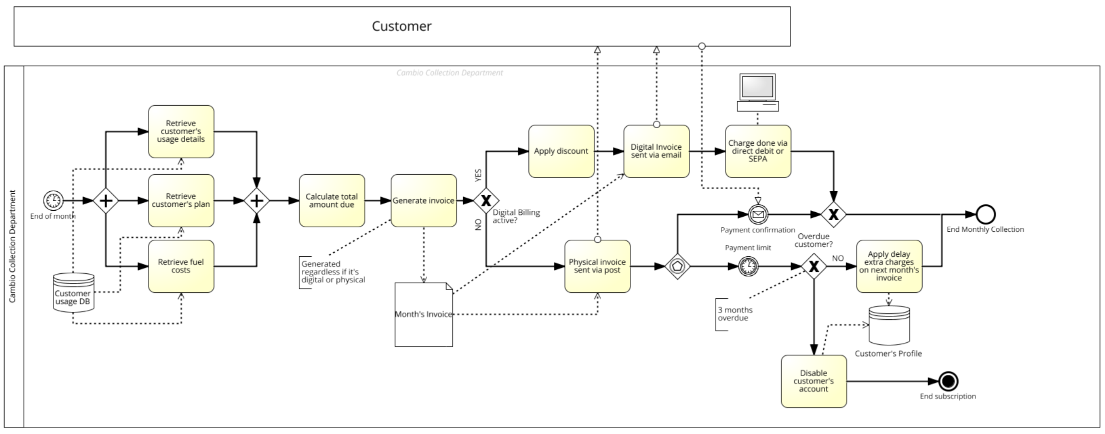

Customer
Cambio Collection Department
Cambio Collection Department
End of month
Retrieve customer's usage details
Retrieve customer's plan
Retrieve fuel costs
Customer usage DB
Calculate total amount due
Generate invoice
Generated regardless if it's digital or physical
Month's Invoice
YES
NO
Digital Billing active?
Apply discount
Digital Invoice sent via email
Charge done via direct debit or SEPA
Physical invoice sent via post
Payment confirmation
Payment limit
Overdue customer?
NO
Apply delay extra charges on next month's invoice
3 months overdue
Customer's Profile
Disable customer's account
End Monthly Collection
End subscription

Figure 4.3: Example BPMN model representing a car rental accounts receivable process.

PLG files is a file format generated by the PLG2 tool itself. It is a specific format which is not available in any other platform. Very similar to a basic BPMN model, it allows to add some semantics to describe the process. All the workflow control-flow patterns mentioned in Subsection 2.2.2 are present in the model.

Once a model is imported or generated within the tool, the next step is the stream generation. Such step was described already in Subsection 2.2.2. The most important parameters to take in consideration are the noise configuration, the network port defined, maximum number of parallel instances, and the time multiplier. Once all these are correctly set and the stream is started, the algorithm needs to connect to the local port number to "receive" the events which are being transmitted by the tool. In the case of this framework, a Telnet session was created locally to connect to the port defined (tests and experiments were done with port 8888 successfully). Need to make sure the chosen port is available and the application has the required permissions to "transmit" to it. We suggest looking at the example stream generated in the PLG documentation site: `https://plg.processmining.it/`.

4.1.2 XES to Event Stream

The other alternative available through this framework is to provide an event log in an XES format. Though there exists many other formats for representing and event log, PM4PY specifically deals with only XES and CSV formats. Furthermore, for the purpose of this thesis we

[1]More information regarding BPMN available at `https://www.bpmn.org/`
[2]More information regarding Signavio available at `https://www.signavio.com/`

decided to use only XES, as it is the most widely-used format in the literature reviewed. Using the *streaming.importer.xes*, *objects.log.importer.xes*, and *objects.conversion.log* libraries from PM4PY, the event log is first imported. Then, converted into an event stream using a sorting based on the timestamp key ('time:timestamp'). This results on an iterable object where the events are ordered by timestamp from oldest to newest. By iterating over such object one replicates an "online" stream.

4.1.3 Initializing the framework

Besides initializing the stream, from any of the two alternatives described, there are a set of objects which need to be initialized in order for the framework to proceed and processing the stream. Next are the steps followed to complete the setup:

1. *Create End-Events lookup dataset.* In a real-world scenario, an external entity would provide information regarding when a case is finished. Process owner and entities involved should be familiarized enough signal the possible paths a case can take and what are the end events of each. These are key for the algorithm to understand if a case is still active, and therefore, needs to store which was the last event seen to continue its variant. In the context of this framework, such events are extracted by pre-processing the event log (in the case of an XES log provided as input), or defined by looking at the process model (in the case of a process model provided as input).

2. *Initialize context variables.* A set of variables, which are used during the whole execution of the framework, are created and initialized. These include: frequency lists for activities and relations (these are the global lists which are continues), end events dictionary (created in the previous step), list of cases seen, ordered dictionary to store tracking cases, counters for number of events per model and number of events overall, tree (instance of PrefixTree class), current node (root node to start the first tree), and a window (described in 4.3).

3. *Initialize model building variables (optional).* The framework provides an option for building a petri net representation of the process for every window of observation done. Such brings high computational cost and time over the algorithms, so it is not used for the context of these experiments.

4.2 Prefix-Trees as a Data Processing and Storage Technique

As previously mentioned in 2.3.3, this framework relies on a data storage structure called "prefix-tree" or "trie". Tries are a well-established data structure for compactly storing ordered dictionaries. These data objects are mathematically represented as simple graphs. These have a unique path between every pair of nodes, where one node is designated as the root node and all other are children or parent nodes. Each single data value on the sequence is represented by a single node in a tree structure with arcs connecting the set of nodes. These structures offer advantages over other indexing structures (e.g., binary search trees or hash tables) according to [39]: they are more compact because overlapping prefixes are stored only once, and unless the set of keys are extremely small, lookup is faster. In fact, lookup in a trie in the worst-case is $O(m)$ time (where m is the length of the search sequence), whereas for the other indexing structures worst-case could reach $O(N)$, where N is the number of records. Furthermore, tries cannot have collisions of key storage, as each node in the tree will always have a single pointer (compared to a hash table where multiple keys can point to the same position in the table). Based on the work in [40], *StrProM* implemented a prefix-tree structure for storing and processing an event stream. Figure 4.4 shows an example prefix-tree generated after processing thirteen traces of an event stream. Each node shows its corresponding activity label and frequency. We'll explain in detail the logic behind such implementation and how it was adjusted to achieve the goals proposed.

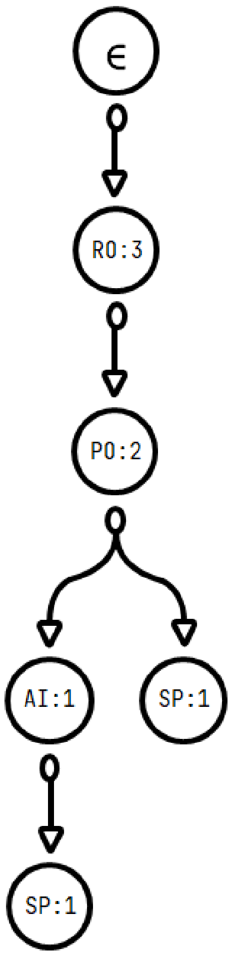


Figure 4.4: Example prefix-tree from event stream of Figure 4.2

4.2.1 Prefix-Tree Components

As mentioned previously, a prefix-tree is composed of nodes and edges (arcs). In the context of this thesis, each node n corresponds to an event and stores information arriving from the event stream. Refer to Equation 3.3 for details on such attributes stored per node. The edges of our prefix tree are represented by the children nodes ch of each node. Though some type of trees may store information within the edges, in this context they act as the relation between two nodes ($a > b$). A Python class was created to store all data related to the prefix-tree for a specific period of observation. A new tree instance is created once the number of events for the pruning step has been processed from the stream. The different components of our trees will be explained further in this section.

Trie Node

A trie node is a node from the prefix-tree which is not the root node. Every event process from the event stream must create a new node in the tree or increase the frequency of an existing node. A new node is created if the last event seen does not have this new event as a children node. Every node is unique, while there may be other nodes which have the same values for different attributes. Nevertheless, two nodes may never share the same activity label a and parent node list pL. Therefore, each node can only have a single parent node p. A Python class was created to store all node data. Every new event node creates a new instance of the object *TrieNode*.

Root Node

All prefix-trees start from a root node which we will denote as ϵ. A root node is a node itself where its activity label a is the string 'root' and an empty ($\perp$) parent node p. Furthermore, the list of parent nodes pL will always remain empty and the frequency f of such node remain as 0. Children nodes ch of the root nodes are identified as the first events of a trace. The root nodes are a representation of the starting point of the traces and connects all traces which have been seen during the observation period of that prefix tree. There are two scenarios where a root node gets a children assigned:

- An event from a new case (never seen before) is received and such event is not already a children event of the root node.
- A case remains open after a pruning step and therefore the last event seen of that case will now be a children event of the root node (incomplete cases).

Tree Branches

Tree branches are a key component for this framework. They are the representation of a trace variant within the process (a unique sequence of events for a case). The number of branches of a tree will act as the number of different trace variations which the prefix-tree has observed during the current period. A tree can have [1,∞] branches. The end node of a branch represents the end of a trace, while the start node will always be the root node. Sub-branches can also be created from intermediary nodes, which means that there are traces that have equal "prefixes" (two traces had the same sequence of events up to certain point). Two branches may never merge into a single branch. Because each branch relates to a different trace variation, each event seen in that trace variant is unique. This means that no node can have more than a single parent node p (i.e., a unique sequence of events prior to it).

4.2.2 Prefix-Tree Attributes

In 4.2.1 it is mentioned that a Python class was created to store the information regarding a prefix-tree. Each instance of the *PrefixTree* class has a set of attributes which reference important components of the tree. This section describes each of these attributes and what are they used for in the context of the framework. Initialization function for the class can be found in Appendix B Listing B.

User-defined parameters

Following are the set of parameters required to initialize a prefix-tree.

- **PruningSteps (Integer)**: Number of events that must be processed before pruning has to be performed over the tree. Fixed number that should be set based on some knowledge regarding the stream. (Default value: 1000)
- **TreePruneSteps (Integer)**: Number of events that must be processed before an intermediary pruning has to be performed over the tree. It is a fraction of the *PruningSteps* value. (Default value: PruningSteps / 4)
- **LambdaDecay (Double)**: Decaying factor that decreases the relevance of older counts of events and relation when pruning the tree. If value is 0, older events become as relevant as new ones (no decay); if set to 1, older events are not relevant at all (complete decay). $\lambda \in [0, 1]$. (Default value: 0.25)
- **Cmax (Integer)**: Maximum number of cases that the algorithm is able to track concurrently as open cases. Memory requirements increase the higher the value. (Default value: 6)

- **TPO (Double)**: Threshold used to filter nodes which have a support (frequency) lower than the defined value. Parameter is equivalent to the *TPO* threshold from the Heuristics Miner [38] algorithm. (Default value: 1)

Not user-defined attributes

Next are the set of attributes which are generated automatically when a new prefix-tree is created:

- **NodeFrequencies (Dictionary)**: Dictionary which stores the frequency per node. Updated after every *TreePruneSteps* and *PruningStep* with the decaying old values and newly seen values.

- **RelationFrequencies (Dictionary)**: Dictionary which stores the frequency per relation ($a > b$). Updated after every *TreePruneStep* and *PruningStep* with the decaying old values and newly seen values.

- **StartActFrequencies (Dictionary)**: Dictionary which stores the frequency of those nodes which are the start of a trace (children of the root node). Updated during the *TreePruneStep.*

- **EndActFrequencies (Dictionary)**: Dictionary which stores the frequency of those nodes which are the end of a trace (last node of a branch). Updated during the *TreePruneStep.*

- **TreeNodes (List)**: List which stores all nodes seen by the prefix-tree. Used to reset the frequencies of all nodes during *TreePruneStep* and *PruningStep.*

- **EventsSeen (Integer)**: Counter for the number of events the algorithm has process since the beginning (t = 0). Used to identify point in time when a concept drift was detected.

Node Attributes

Based on the notation described in Chapter 3, this subsection will describe the attributes the *Node* class stores for every instance of the object. Initialization function for the class can be found in Appendix B Listing B. Following is the list of attributes for the class:

- **NodeId (UUID)**: Randomly generated unique identifier for the node.

- **Activity (String)**: String with the activity name of the event. Multiple nodes can have the same activity name.

- **Parent (Node)**: Nested node instance which stores the parent node (previously seen node for this case). A node can only have one parent node.

- **ParentList (List)**: List of all previous parent nodes. Used to identify if this same node exists on another tree.

- **Children (Dictionary)**: Dictionary which stores all the children nodes of this node. Each key stores the *Activity* and the node itself in the value.

- **Frequency (Integer)**: Stores the frequency of this node. It is reset after every *TreePruneStep* and *PruningStep.*

- **BranchId (String)**: String which is composed of the *ParentList* of nodes and the *Activity* of this node. Used for aggregating the frequencies of nodes between trees.

Case Attributes

This subsection will describe the attributes the *Case* class stores for every instance of the object. Initialization function for the class can be found in Appendix B Listing B. Following is the list of attributes for the class:

- **CaseId (String)**: String storing the case identifier. Value is used to compare against the active cases in the queue or identify if the case of a new event has been seen before.
- **CaseNode (Node)**: Node object instance of the last event seen for that case. Used to continue growing the branch of that case if a new event is seen.
- **Active (Bool)**: Bool flag to signal if this is an active case (have not seen the end event for this case yet) or if is inactive.

4.2.3 Processing Events from the Event Stream

Once the stream is "sending" events in an online fashion and all variables are initialized, the framework is ready to process such events. A set of methods from both the *TrieNode* and *PrefixTree* classes are used to achieve this. The required result is the generation of a discrete dataset over which a transformation algorithm will be applied to convert it to a continuous dataset (described in 4.3). This section will explain how each of these methods work together to generate a discrete dataset representing the event stream. All pseudo codes and listings can be found in Appendix A and B respectively.

Insert-By-Event

This is the main method used for processing the events from the stream. Once a new event arrives, this method is called so a node is either created or updated within the current prefix-tree. Because of the size of this method, each component will be split into parts and described individually. First part focuses on the handling of the cases and actions related to each scenario. A complete algorithm breakdown can be seen in Appendix A Algorithm 4. Once the new event is received all its attributes are stored in a set of variables (*caseID, eventID, eventTimestamp*)(lines 1-3). Then, depending on if the case is a new case that has never been seen, or if it is an existing case that is not currently being tracked, or a tracked case, a set of tasks will be triggered.

For completely new cases, the *currentNode* of the tree becomes the *root* node and a new *Case* has to be created with the *caseID* and *currentNode* as the pointing node (lines 5 and 6). This newly crated case c must be added to the *Dcase* ordered dictionary so that it becomes one of the tracking cases (line 7). Afterwards, the algorithm must check if *Dcase* has reached its maximum number by comparing it with *Cmax* (line 8). If so, the last element (oldest) must be dropped (line 9). Later, if the eventID is not within the list of start activities for this tree (*StartActFrequencies*), a new record needs to be created in the dictionary and the frequency set to 1 (lines 10 and 11). If it has been registered before, then we just increase the frequency of that record in the dictionary (line 19). Finally, if the *eventID* is not one of the *currentNode.ch*, then a new node n is created with the corresponding *eventID* as activity (a) and the *currentNode* as the parent node (p) and assigned as a children of the *currentNode* (lines 12-14). If already a children, let the *currentNode* become that children (line 16).

For existing cases that are not currently tracked, we first run the *lookForNode* method to find a node (if there exists one) within the tree to be the "potential" node (line 22). This logic was chosen so that the algorithm does not create a new branch for existing cases that are not being tracked. Instead, it tries to find the first node that has the same *eventID* on the tree and takes the parent of it as the *findNode.* A complete algorithm breakdown can be seen in Appendix A Algorithm 5. If no node is found with such activity name, we create a new branch and assign the *root* node to the *currentNode* (lines 23 and 24). If there was a n found, then this *parentNode* is assigned as the *currentNode* (line 26). Then, a new case c is created with the existing *caseID* and the *currentNode* as pointer (line 28). This same must be added to the ordered dictionary

Dcase. Same as before, it needs to review if the maximum number of cases for tracking has been exceeded. Finally, as in the previous scenario, it checks if this *eventID* exists as a children of the *currentNode* so that it can create a new node or just increase the frequency of the existing one.

The last scenario assumes the case from the event received exists and it is being tracked (exist in the *Dcase* dicitonary). Therefore, the only step steps are to move this case to the from of the list (newest case) and repeat the steps for checking if the activity already exists as a children of the *currentNode* or not (lines 40-47). Figure 4.5 shows an example using the tree defined in 4.4. The red-highlighted figures represent the actions taken for each scenario. First scenario creates a new node and branch for a new case never seen before; second scenario creates a new node under an existing one from where an active case was pointing to; third scenario looks for an existing node that had that *eventId* on the tree. After not finding one it generates a new node and branch, adds this case to the tracked list and points it to the new node.

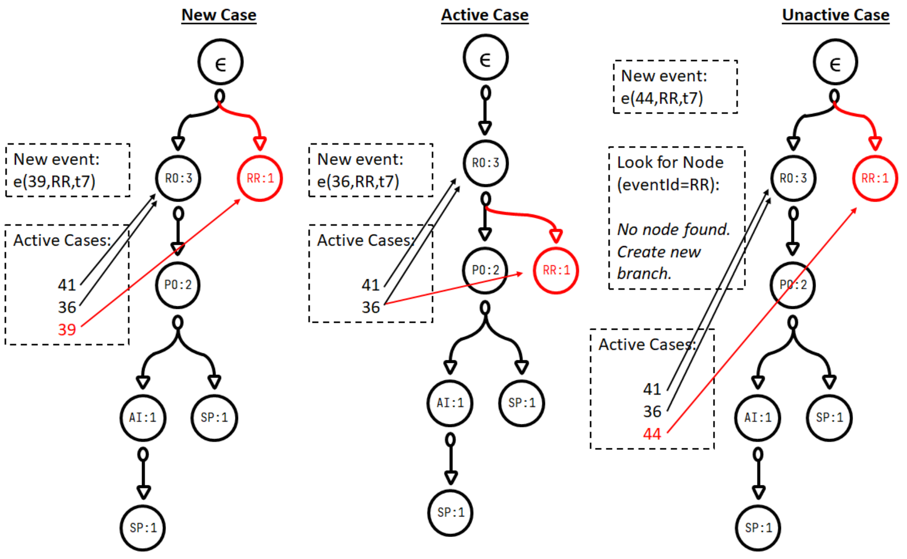


Figure 4.5: Example of the three different scenarios for handling an event depending on the case status

After handling the cases scenarios, the algorithm needs to verify if the case has ended or if it is still an active case. A complete algorithm breakdown can be seen in Appendix A Algorithm 6. First we add the *currentNode* to the list of *treeNodes*, assign the *currentNode* as the last node *n* seen for this case *c* and increase the *pruningCounter* (lines 1-3). Then, we need to verify if the *eventID* and *eventTimestamp* received are the same as the ones stored in the *EndEventsDic* for this *caseID* (line 4). If it is, set the flag *fl* of the case *c* as False (inactive) (line 5). Furthermore, if the *eventID* is not already registered as an end event in for the tree (*EndActFrequencies*), then create a new entry and assign frequency of 1. If existing, increase the frequency (lines 6-10).

The last step of the method reviews if enough events have been seen to trigger either a *TreePruneStep* or a *PruningStep*. Recall from Subsection 4.2.2 that one condition is dependent from the other and there will be a point in time were both of them are triggered. A complete algorithm breakdown can be seen in Appendix A Algorithm 7. It first checks if the remainder of the division between the *pruningCounter* and the prefix-tree *TreePruneSteps* is 0 (line 1). If such condition fulfills, then three functions are executed:

- *T.PruneTree*() (Algorithm 8)
- *T.DecayTree*() (Algorithm 10)
- *T.ResetFrequencies*() (Algorithm 11)

To ilustrate an example, lets define a stream S from the event log Table 2.1:

$$S = \Big\langle (23, RO, t0)(41, TO, t1)(23, PO, t3)(23, AI, t4)(36, RO, t5)(23, SP, t6) \Big\rangle$$

Knowing that case 23 ends with the activity "Ship Parcel" at a corresponding timestamp, that case would set its *c.active* flag to $False$. Therefore cases 41 and 36 would remain active and pointing to the last activity seen $c.caseNode$, which would be "Receive Order" for both of them. If a pruning step is performed at this point in time, the tree would prune the branches and save the frequencies of the activities, while leaving only a single node for each of the remaining active cases that are being tracked with a frequency of zero. Figure 4.6 shows the result of the prefix-tree after performing the pruning. The next events which are seen for cases 41 and 36 would create nodes under this new tree, until the next pruning step is performed.

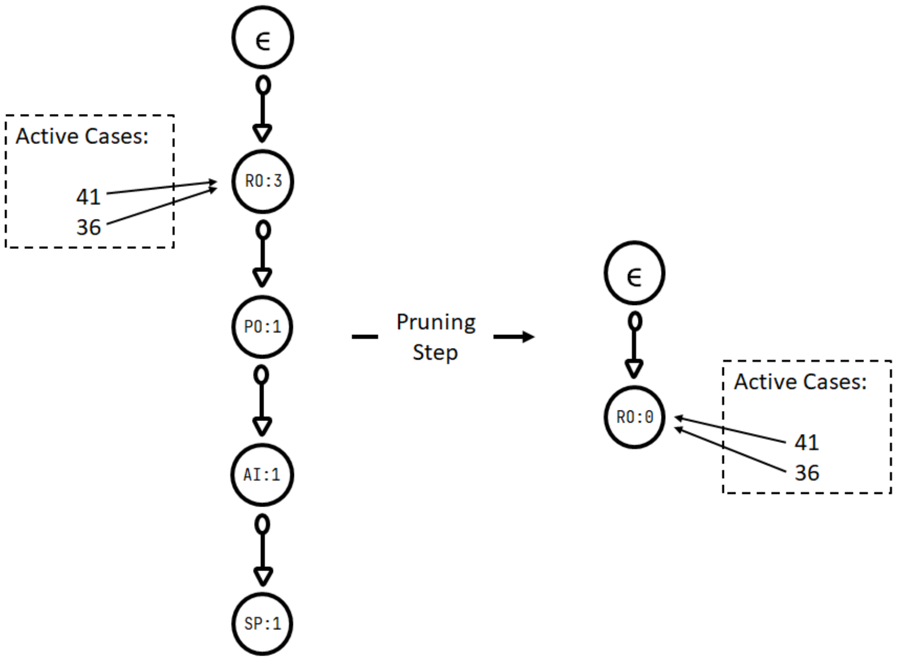


Figure 4.6: Example of a pruning step from log 2.1

After pruning the tree, decaying the global frequency lists and resetting the frequencies of the nodes within the tree, the *traceCounter* is increased and the *currentNode* will point back to the *root* of the tree (lines 2-6). The last part of the *InsertByEvent* method corresponds to Section 4.3.

The result of executing these steps as more events arrive from the event stream is the generation of multiple prefix-trees. Each tree will store the discrete information of the population (*PruningSteps* number of events) they were observing. Now that multiple trees are generated, such data must be converted into a continuous dataset, which will be used as an input for an concept drift detection technique.

4.3 Discrete-to-Continuous Conversion

The series of trees generated by the algorithm are a discrete representation of our data. A discrete value, as explained in [41], is a numeric representation of a categorical attribute. It is the counting of something which got a value assigned. On the other hand, continuous values are a measurement of the size or proportion of a specific attribute. Such value can be higher or lower depending on its characteristics. As explained in Section 2.2.3, concept drift detection techniques are applied over a series of successive chronological populations. This means that there is a need to transform our discrete dataset into a continuous one before applying any drift detection technique.

In order to do so, we have to look at what features of our dataset are we interested in and believe will provide meaningful data for a drift detection algorithm. In Sections 2.3.1 and 2.3.2 a set of supervised and unsupervised learning techniques used for concept drift detection were described. Each technique applies its own approach for extracting a set of features from the data stream and converting them into a continuous chronological dataset representing the populations' evolution. While techniques like [34] [18] use the probability distribution of the dataset, other techniques like [13] [11] [12] extract a set of features and then apply some measurement algorithm (LOF, J measure of disimilarity, and DECLARE constraints respectively) over them to create a continuous dataset. In our approach we believe that the Prefix-Trees store relevant information, besides the frequency of activities and relations, that can be useful for applying a drift detection technique. The structure of the tree is a representation of the variation of traces and can be significant when focusing on detecting drifts from the control-flow perspective. Therefore, a transformation technique which can take advantage of such characteristics of this data structure, has to be used.

The tree distance metric described in Subsection 2.3.5, based on PCA over trees, appears to be a good approach to generate a continuous value for a series of trees. Listing B show the script used to compute such metric. Important to note that in our context, we're using this approach with both nodes and relations. This means that we look at not only those nodes which are present or not in both trees, but also the relations between a pair of nodes. As the equations have been already explained, the algorithm has been excluded from the Appendix A. A Python class was created to store relevant information from this metric: *TreeDistance* class. The attributes of such class are the following:

- **win0NodesNotInWin1 (Dictionary)**: Stores the $key : value^2$ pairs of those nodes and relations present in W_0 but not in W_1.
- **win1NodesNotInWin0 (Dictionary)**: Stores the $key : value^2$ pairs of those nodes and relations present in W_1 but not in W_0.
- **interDict (Dictionary)**: Stores the $key : (v_0 - v_1)^2$ pairs of those nodes and relations present in both W_0 and W_1.
- **interSumOfFreq (Double)**: Sum of all values from *interDict*.
- **notInterDict (Dictionary)**: Stores the combination of both *win0NodesNotInWin1* and *win1NodesNotInWin0*.
- **notInterSumOfFreq (Double)**: Sum of all values from *notInterDict*.
- **treeDistanceMetric (Double)**: Euclidean distance between two trees metric. Equation 2.4

Figure 4.7 shows an example of two trees created from a set of sub-population of the stream. Nodes with the same coloring are identical nodes from topology perspective. These set of nodes would be part of the *interDict*. Those nodes which have different coloring between the trees, though some of them have the same activity name, are unique from a topology perspective. Therefore, they would be part of the *notInterDict*. These distinct nodes are the ones which influence the most over the metric. The formula would then also look at the frequencies stored on these nodes for the Equation 2.6.

Initialization function for the class can be found in Appendix B. Now that a method for converting our discrete dataset into a continuous series of datapoints was presented, a technique for iterating over such list of trees and analyzing these values has to be chosen.

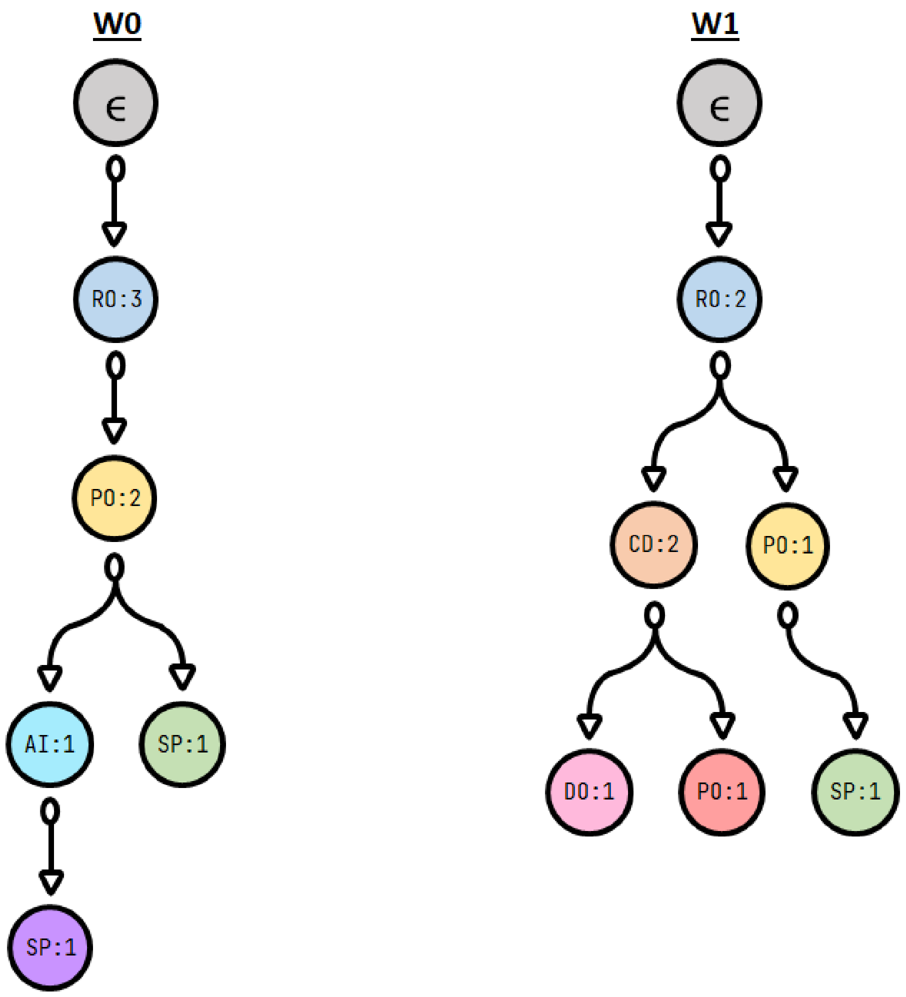


Figure 4.7: Example of two aggregated trees generated(W_0 and W_1) from sub-populations of window W

4.4 Concept Drift Detection Technique

As explained in previous sections, the algorithm will generate a series of Prefix-Trees out of the event stream and store them in a chronological way (increasing number of events seen) within the *PrefixTreeList* list from the *Window* class. Algorithm 13 in Appendix A shows a complete breakdown of the steps taken to store each during the pruning step. If the list has reached its maximum size *MaxWindowSize*, then the last item of the least (oldest) has to be deleted so that new trees, with more recent information, can be stored.

To store the data of each window, a Python class *Window* was created. The class has the following attributes:

- **maxWindowSize (Integer)**: Maximum number of prefix-trees that the window W is able to store. Such number is fixed and a parameter provided by the user.

- **winSize (Integer)**: Number of prefix-trees that should be stored in order to activate the drift detection procedure. It is used as a threshold to know when should the drift detection start. This value starts with same value as *maxWindowSize*, but decreases depending if a drift was detected recently or grows back up to *maxWindowSize* if no drift has been detected.

- **prefixTreeList (List)**: Stores the prefix-trees generated from the event stream. It has a maximum length dependent on the size of *maxWindowSize*.
- **driftsIdentified (List)**: Stores the *T.eventsSeen* value for of the last tree from W_0 for each of the drifts identified.
- **cddFlag (Bool)**: Flag used to trigger CDD once a new tree has been completely processed. Set to *True* after every new tree or drift identified; set to *False* if no drift was detected after CDD or the prefix-tree is has not achieved the *PruningSteps*.

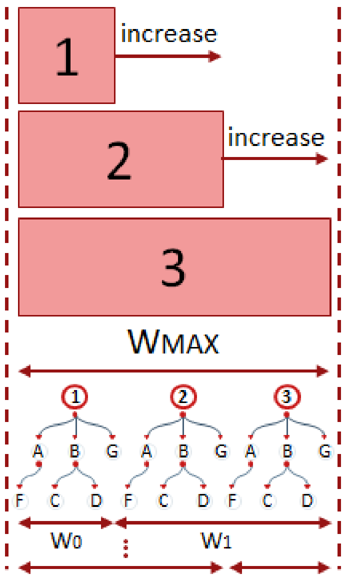


Figure 4.8: Adaptive Window approach storing Prefix-Trees

Figure 4.8 shows a diagram of the evolution of the window. These approach was adapted from [8] to now store Prefix-Trees rather than the frequency lists. It uses an adaptive window *W*, which focuses on short intervals for highly deviating periods or increase its width in case of uniform observations (no drift detected). Algorithm 14 in Appendix A shows a complete breakdown of the steps. The idea behind the adaptive window approach is to store your data structures up until the maximum size of the window is reached (line 1-2). At that point the concept drift detection process is triggered (line 3). For all partitions of the window into two sub-populations, an aggregation of the data structures will be done and a concept drift detection technique applied over those sub-populations data structures (Algorithm 15). Basically, the set of trees within each of the partitions are aggregated by the mean of frequency values of the nodes and relations, resulting in a single aggregated prefix-tree for population W_0 and W_1 (lines 11 - 14 from Algorithm 17. For this step, a sub-class of the *Window* class was created. The class has the following attributes:

- **winNodeFreq (List)**: Stores the node frequencies of each tree within the sub-population.
- **winRelFreq (List)**: Stores the relation frequencies of each tree within the sub-population.

Now that the data has been transformed into a continuous dataset by using a distance metric between the sub-populations of trees, such values need to be used as input for a drift detection technique.

Algorithm 16 shows a breakdown of how such packages were used in this implementation. First the distance metric computed between W_0 and W_1 is added to the ADWIN object (line 1). Then, the *detect_change* is used to verify if such element caused a drift (line 2). If a drift was detected, the elements from sub-population W_0 (older elements) are removed from the *prefixTreeList* (line

3). Afterwards, the *WinSize* is adjusted to include reduce the required number of trees to perform drift detection (line 4). Recalling how the adaptive window approach works, once a drift is detected we want to focus on shorter intervals where we assume there are highly deviating periods. This adjustment on *WinSize* allows the algorithm to perform drift detection more frequently. The new size will leave space so that one more tree can be added to the list before performing CDD again. The *indexSlider* is then reset to 1 so that drift detection can be done over all partitions of W and the *cddFlag* is activated (lines 5-6). If a drift was not detected the procedure continues and the *indexSlider* is increased and the flag remains inactivated (lines 8-9). The updated *indexSlider* is return so that the drift detection can continue if more partitions are available (line 11).

The *cddFlag* has a secondary use once a drift is detected. Within the 18, once the *driftDetectionADWIN* function executes, a condition is set to check if a drift was detected (line 1). If a drift was detected, information regarding the drift (number of events of both sub-populations, critical nodes which caused the drift, point in time when drift was located) is stored (line 2). In order to store the drift information, a class *Drift* was created. Once an object with the drift information is created, it is then stored in *driftsIdentified* list of *W*.

4.5 Drift Characterization

The last component from this framework provides further information regarding the drifts. This is, according to [42], understanding *what* has changed in the process behavior. They argue that detection and localization of a process drift does not provide enough insight to undertake a process improvement initiative. Some techniques focus on identifying behavior differences between two process models, but the accuracy of these highly rely on the quality of the discovered process models.

4.5.1 Critical Nodes

Our proposed framework attempts to provide such information using the prefix-trees which are related to the drift's sub-populations. Algorithm 19 shows the logic behind choosing which are the critical nodes that produced a drift. From the *treeDistance* object generated in 15, if a drift is detected, the framework retrieves all nodes and relations from *treeDistance.notInterDict* which have a *value*[2] bigger than a *criticalThreshold* defined. Knowing that the nodes and relations which differ between the two aggregated prefix-trees have the higher weight over the distance metric, we focus on identifying those with the highest value from such list. This threshold should have some context knowledge behind its value. If the value is too high, no node or relation will be considered critical. On the other hand, if it is too low, most of the them will be selected as critical. If no context known regarding the expected frequency of events, it is suggested to leave the default value.

The critical nodes represent each unique node from one aggregated tree which does not exist on the other. Remember from 4.2.2 that each node has a *branchId* attribute, which helps uniquely identify each node. The critical nodes store only the activity name of the last node of the branch. Therefore, there could be several critical nodes with the same name. This does not mean they are the same node, it but that they have the same activity as the last activity of that sequence. Similarly, there could be relations that have the same activity sequence, but that does not mean they are the same two nodes connected.

4.5.2 Drift Logs

To analyze further information about the drifts, the proposed framework provides the sub-logs of each drift identified. The objective here is to allow the users to perform their own drift analysis with the logs from the drifts that the framework has signaled. Algorithm 20 shows the logic behind the log splitting component. Important to note that this is built as a configuration of the tool. It is not necessary to execute drift detection, but it is available if the users wants to receive such sub-logs after the framework completes its execution. To achieve this, a configuration parameter should be set using the command line interface (CLI), or corresponding configuration file from the IDE used. An example CLI script to execute this configuration is as follows:

```
C:\Users\XXX> BPIC2015_CDD.py -c True
```

Listing 4.1: Config script execution for drift logs.

This will generate a new folder named 'results', which will contain two logs per drift, for *Window 0* and *Window 1* respectively. With these, users can do further analysis using external tools (like ProM or Disco) to understand in more detail what caused these drifts identified. We believe it is a very important functionality which has not been seen in the literature reviewed for this thesis project.

Chapter 5

Experimental Setup

This chapter aims to describe the experimental setup done during the evaluation of this framework over both synthetic and real-world datasets. First, a brief description of how the framework is evaluated in terms of drifts detected and performance is described 5.1. Then, an overview of the artificial datasets used in the experiments will be given 5.2 as well as for a real-world dataset 5.3. Finally, a few details are given about the hardware used for running the experiments 5.4. The complete implementation of this framework can be found in the following Github repository.

5.1 Objective Evaluation

The online detection of concept drifts provides an objective evaluation mechanism. Classical metrics used in data mining can be adopted to evaluate the effectiveness of the approach presented. Such metrics will be number of true positives (TP), false positives (FP), false negatives (FN), and the derived metrics from these (precision, recall, and F-1 score). To accurately define these metrics, a lag period l surrounding a detected or actual concept drift will be used. The interpretation is as follows [9]:

- *TP*: a concept drift is detected at time t and there is an actual change within $t \pm l$.
- *FP*: a concept drift is detected at time t but there is no actual change within $t \pm l$.
- *FN*: a concept drift is happened at time t but no change has been detected within $t \pm l$.

Precision measures the fraction of detected changes that are correct ($TP/(TP+FP)$), while *recall* measures the fraction of actual changes that have been detected ($TP/(TP+FN)$). *F1-score* is defined as the harmonic mean between precision and recall ($2 * precision * recall/(precision + recall)$). Techniques that are able to detected changes with high precision and recall (close to 1) are preferred.

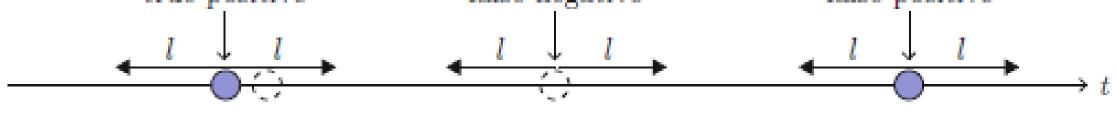


Figure 5.1: Evaluation of change detection techniques. Solid circle and dashed circle indicate a change detected and an actual change respectively [9].

Furthermore, a metric for assessing the ability of the algorithm to find drifts as early as possible in an event stream called *mean delay* will be used. It is measured as the number of events between the actual position of the drift and the end of the detection window. Finally, both execution times of the algorithm and the total memory consumption (without considering the setup steps) will be captured to evaluate the performance from a system's perspective.

5.2 Artificial Dataset

Artificial datasets provide a good environment for testing concept drift detection algorithms. They allow the manual introduction of drifts in relatively smaller event logs. This helps authors define the "gold-standard" for the drifts intended to be identified. Two different artificial datasets are used within the experiments.

5.2.1 Loan Application Dataset

One of the nobel methods described previously [10] generated a benchmark of 72 event logs[1] by taking an example of a business process for assessing loan applications. The base model is shown in Figure 5.2. It is composed of 15 activities, one start event and three end events. It also exhibits different control-flow structures including loops, parallel and alternative branches. In order to assess the ability of the algorithm to detect sudden drifts, authors systematically altered the base model by applying one out of twelve simple change patterns described in [43]. A more detailed description of the patterns and what change operations they capture is presented in [10]. Combinations of these patterns were used in a nested approach to result in a composite pattern. These were inserted in logs of different sizes (number of traces). The experiments performed in Chapter 6 will only consider the nested patterns (IOR, IRO, OIR, ORI, RIO, ROI) of 2,500 traces each. We believe this gives a better view on how the framework performs over drifts which have a combination of changes, which relates more to reality, than a single change in the model. The intention was to identify the 9 drifts located in each event log with an *inter-drift* (distance between two drifts) distance of approximately 12,000 events. Though this number is not exact, it was the result of the amount of events in average over 300 traces. Such analysis was done using Disco[2]. We defined a lag l of $2 \times PruningSteps$ events to use in the logic of identifying true positives, false positives, and false negatives. This number would represent two more trees to the right or to the left of the drift. We believe that's an accurate estimation, based on the relative position of the drifts.

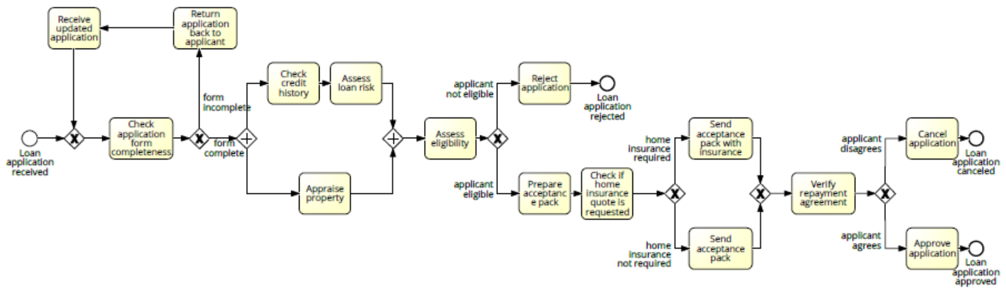

Receive updated application
Return application back to applicant
form incomplete
Loan application received
Check application form completeness
form complete
Check credit history
Assess loan risk
Appraise property
Assess eligibility
applicant not eligible
Reject application
Loan application rejected
applicant eligible
Prepare acceptance pack
Check if home insurance quote is requested
home insurance required
Send acceptance pack with insurance
home insurance not required
Send acceptance pack
Verify repayment agreement
applicant disagrees
Cancel application
Loan application canceled
applicant agrees
Approve application
Loan application approved

Figure 5.2: Base BPMN model of a loan application process [10].

5.2.2 PLG2 Artificial Stream

The second artificial log was generated through the PLG2 tool described in Chapter 2. A random process model was created with 21 activities and 12 gateways. Because of the size of the image, such is located in Appendix C Figure C.1. This process will serve as an input for the *Event Stream Generator* 2.2 to create an artificial event stream that will send to a local port. An evolution of this process model was generated to create an artificial drift. Such evolution contains 25 activities and 14 gateways. This introduction of new activities and decision points should be identified by the framework as a drift. At a certain point in time, within the stream generator parameters the new model will be selected. The tool will then send events from the new process model into the

[1]Base model and event logs available at Apromore ProDrift Logs Repository
[2]More information about Disco at `https://fluxicon.com/disco/`

local port. Depending on the amount of events set to be sent per second, the drift may take a bit of time to be detected. Such parameter, as well as the number of parallel processes instances running will be defined in the experiment results.

5.3 Real-World Dataset

5.3.1 BPI Challenge 2015

A useful real-life dataset used thoroughly in the literature ([8], [13], [15]) is the "Business Process Intelligence Challenge 2015" (BPIC2015)[3]. The data was provided by five Dutch municipalities containing all building permit applications over a period of approximately four years. It is a compilation of five log files, each one with a different number of cases c and events e.

Though the process should be relatively similar, each municipality performs variations to it. This generates a perfect scenario for concept drift detection, where the algorithms should identify a drift once they start processing the events from a new municipality, with a total of four drifts. Figures C.6 and C.5 show the process maps of the first two municipalities, where there is a clear distinction between both process flow. Therefore, the five logs must be merged to generate a single event stream for online concept drift detection. Such will be used in experiments shown on Chapter 6. Following are the details of each log:

1. Log 1: 1199 cases, 52217 events.
2. Log 2: 832 cases, 44354 events.
3. Log 3: 1409 cases, 59681 events.
4. Log 4: 1053 cases, 47293 events.
5. Log 5: 1156 cases, 59083 events.

5.3.2 BPI Challenge 2020

Another real-life dataset available for research is related to the "Business Process Intelligence Challenge 2020" (BPIC2020)[4]. The challenge is related to the reimbursement process at TU/e from the staff travels to conferences, customers, and project meetings. Such are are distinguished by the type of trip: domestic or international.

The data contains information from 2017 and 2018. In 2017, the process was implemented as a pilot over only two departments at TU/e. By 2018 the process was extended to all departments from TU/e. Furthermore, the process changed drastically from one year to the other. Some activities were not required anymore, while others were introduced. This change from the pilot process to the new one is a "gold-standard" drift which the algorithm should be able to detect. Though the amount of cases also highly increased during this transition, our model is not designed to identify this type of drift.

The event log contains 6,449 cases with a total of 72,151 events. Each event has further data stored related to the process (creation date, created by, effective date, organization id, etc.). This are not relevant for the purpose of this thesis, but may be useful for other concept drift detection approaches focused on the data perspective. Figures C.3 and C.4 from Appendix C shows process maps before and after the drift was present. Though the set of activities are not clearly seen, the main focus of these images is identifying a clear process change at a certain point in time. The process maps show a distinctive difference on the sequence of activities which each international declaration may follow.

[3]More information related to the BPIC2015, the process, and event logs can be found at `https://www.win.tue.nl/bpi/doku.php?id=2015:challenge`

[4]More information related to the BPIC2020, the process, and event logs can be found at `https://icpmconference.org/2020/bpi-challenge/`

5.3.3 Hospital & BPI Challenge 2012

The last real-world dataset used during the experimental phase of this thesis was the 'Hospital & BPI Challenge 2012'. It is composed of two distinct datasets and merged to create artificial drifts from real data.

The 'Hospital' dataset is composed of 1,143 cases and 150,291 events. The first event arrival date is on 03/January/2005 and goes until 20/March/2008. This dataset is shared by the "Business Process Intelligence Challenge 2011"[5] edition. Log was provided by a Dutch Academic Hospital and relates to activities that a patient from the Gynaecology department would go through. In some scenarios, the patients may go through different phases (overlapping), therefore some attributes are repeated. On the other hand, the "Business Process Intelligence Challenge 2012"[6] contains 13,807 cases and 262,00 events. These go from 01/October/2011 to 14/March/2012. The log represents an application process for a personal loan or overdraft within a global financial organization, provided by a Dutch Financial Institute. Furthermore, the challenge description states that the event log is a merger of three intertwined sub-processes itself. This would make a case for more intra-drifts, but because we don't have details of points in time where these merges happened, or if they are present since the beginning, we cannot include them over the "known-drifts". The dates of the cases and events from the BPI Challenge 2012 log were adjusted to be included within the time-frame of the 2011 dataset. With the merge of both logs, two artificial drifts were created. The first drift appears on the 01/October/2005, after around 16,000 events seen; the second appears on the 14/March/2006, after around 275,000 events seen. These two are a perfect scenario for drift detection, as both processes are distinct and should represent changes in the data distribution.

5.4 Hardware Details

Following are the details of the system used for performing the experiments on the context of this thesis project:

- CPU $\rightarrow$ Intel(R) Core(TM) i7-8750H @2.20GHz 2.21 GHz
- RAM $\rightarrow$ 16GB (64bit)
- OS $\rightarrow$ Windows 10
- Python $\rightarrow$ version 3.8 (64bit)

[5]More information related to the BPIC2011, the process, and event logs can be found at `https://www.win.tue.nl/bpi/doku.php?id=2011:challenge`

[6]More information related to the BPIC2012, the process, and event logs can be found at `https://www.win.tue.nl/bpi/doku.php?id=2012:challenge`

Chapter 6

Results

This chapter illustrates the results of the experiments performed to address the research questions mentioned in 3, using the framework presented in Chapter 4. The experiments will focus firstly on discovering the best set of parameters to use in Section 6.1. Then, a comparison will be shown against other state-of-the-art techniques from a quality of detection perspective in Section 6.2. After, a further analysis will be done now focusing on the speed of detection (mean delay) in Section 6.3. A final contrast regarding the running time of the algorithms will be demonstrated in Section 6.4. Another experiment was performed to show the effectiveness of the algorithm in a completely online scenario in Section 6.5. Lastly, Section 6.6 will analyze the results of the drifts characterization provided by the proposed framework.

6.1 What are the best parameter settings for the proposed framework?

In Chapter 4 several user and non-user defined attributes were covered, which are critical on defining the structure of the framework and how it will work. The two main parameters which will affect the drift detection process are *maxWindowSize* and *PruningSteps*. These two are components from the reference method *StrProM CDD*, covered in Subsection 2.3.4. They determine both the maximum amount of trees that the window of observation can hold and the size of the trees which will be generated from the event stream respectively. One could set them with the values the authors from the reference algorithm [8], or perform some data exploration tasks to find the best values for your specific dataset. The following experiments therefore attempt to explore the dataset and suggest the best values for both artificial and real-world datasets based on an experimental evaluation.

6.1.1 Which would be a good window size for storing Prefix-Trees?

The parameter *maxWindowSize* restricts how many pruning steps and trees should be done before the concept drift detection procedure starts. If a small number is defined, concept drift detection will be triggered very frequently over a small number of sub-populations, while if the number is very big, the drift detection will be delayed until several pruning steps have happened and bigger number of sub-populations are stored inside the window (W). This also means that if the number is small, drifts which take longer to be represented in the stream will not be discovered. On the other hand, if the window is too big, smaller drifts will probably be missed. The *PruningStep* used for this experiment was of 500 events based on its performance during the development phase.

Figure 6.1 shows a comparison of the experiments done to detect the best size for *maxWindowSize*. For each of the event logs, the framework attempted to discover the 9 drifts artificially set every 10% of the log (∼12,000 events per drift). It is clear that the framework performed better depending on the type of nested pattern. It appears that the framework has trouble identifying

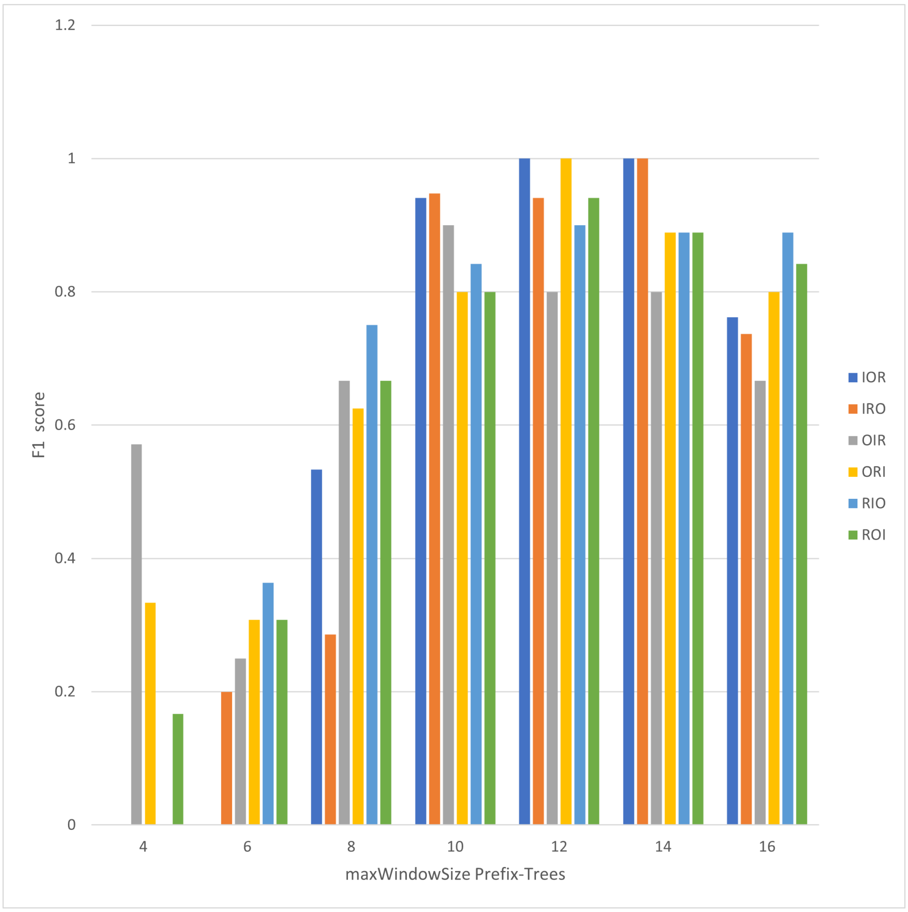


Figure 6.1: F1-score for the different *maxWindowSize* sizes over the nested pattern logs from [10].

those drifts where the change to the process is focused creating loops over existing activities. Most probably this is related to the structure of the trees, because the loops are represented as a branch which repeats its node sequence several times. Therefore, when comparing it against other trees, these nodes are unique unless the same amount of loops for the same sequence of events exist on another tree. Nevertheless, it performed seemingly well with the other patterns, which focused on introducing new activities or changing the sequence by adding parallelization or choices first.

Regarding the size of the window, it seems that the best performing size for all the event logs was 12 Prefix-Trees. One would think that because the trees are storing more data (information regarding 12,000 events), it is the best fit to identify drifts that are within 12,000 events each. Surprisingly, this size is able to identify almost all drift in most of the logs, while preserving a small amount of false positives. Smaller window sizes were not able to identify the 9 drifts, while also detecting a high amount of false positives. We believe that even if the several sub-populations of smaller sizes are combined to generate an aggregated tree, this aggregation causes the structure of the trees from both *W0* and *W1* be relatively similar. Thus, the distance metric is not affected greatly by the topology of the tree and the determining factor would be the nodal attributes. With smaller window sizes the frequency of the activities will be equally small. Therefore, the nodal attributes become less relevant in the process of calculating the distance between two trees. On the other hand, a bigger window of 16 trees shows how the performance starts to degrade, by not being able to identify most of the drifts. As the window can now cover two different drifts within the same observation period, the framework is not able to detect both of them accurately. This gives more support to selecting 12 Prefix-Trees as the best *maxWindowSize* to proceed for these set of logs.

Though 12 Prefix-Trees seems to be the best size with respect to the F1-score, this was not the same result when looking at the delay for detecting such drifts. Figure 6.2 shows a comparison of the mean delay taken by each *maxWindowSize* on identifying the drifts over the different nested pattern logs. Though it seems most of the algorithms perform equally in terms of mean delay over the logs, the highest values are seen with a higher number of prefix-trees in the window. This was expected, as literature [10] has shown that as the window size increases, some of the drifts are missed or require a longer period to be identified. Our hypothesis is that as more events are required to be seen to fill W and perform the drift detection procedure, the drifts which happened in the middle of this window require more events to be identified. One important finding is that the type of drifts does not seem to produce a high impact over the delay over which they are identified. We cannot see a clear pattern in the graph which suggests that the framework performs better over certain types of drifts than others, with respect to the mean delay. There is though an evident trade-off between the reliability of the framework and the time it takes to identify the drifts. It will be important to look at competitors and identify if they show a similar behavior with these event logs or if it is caused specifically by the data structure and technique used by this framework.

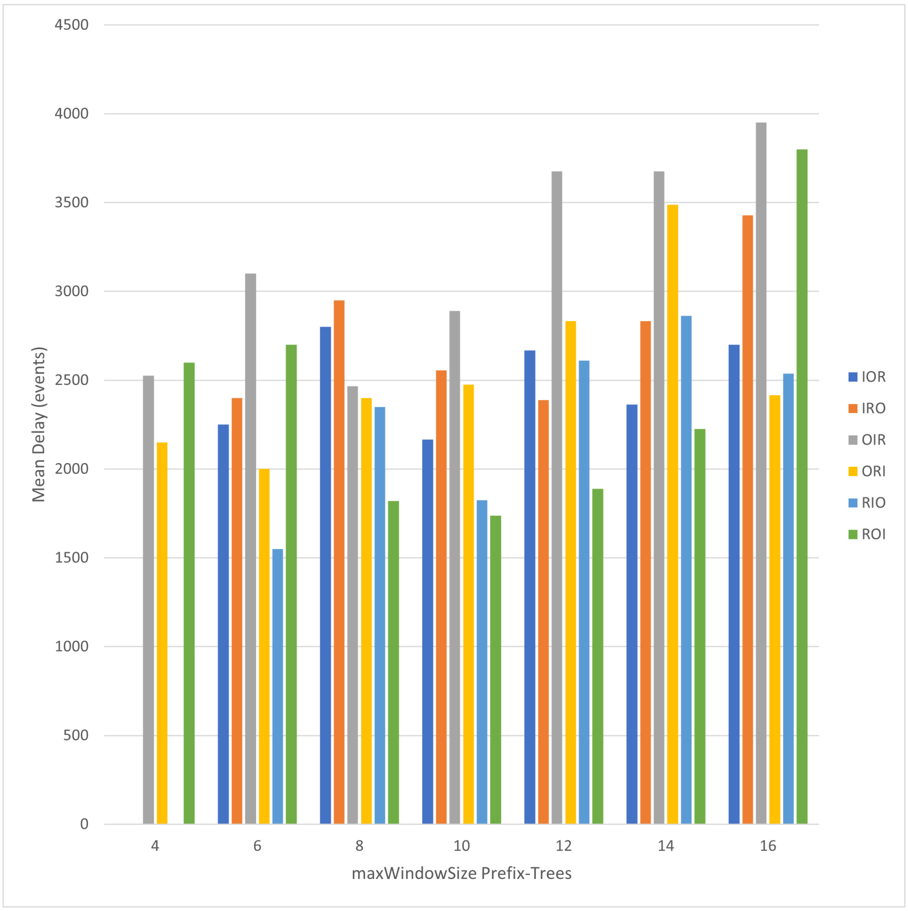


Figure 6.2: Mean delay for the different *maxWindowSize* sizes over the nested pattern logs from [10].

Besides evaluating such parameters over a synthetic log, we wanted verify if this value would also work over real-world datasets. To achieve this, the BPIC2015 dataset from 5.3.1 was used. For each of the six different *maxwindowSize*, the five logs, from each dutch district, were run with the variation of the window W sizes to analyze performance over the F1 score, recall, and the mean delay of detection. Window size of 4 trees was discarded from the analysis, as the results showed a very poor performance against the other sizes. This caused by the inability of detecting the known-drifts, while also detecting several false positives. The size of the trees selected was 1,000 events (*PruningSteps*). Because the logs have a higher inter-drift distance, we believe this was a better parameter to run than the one for the artificial dataset (500 events). Figure 6.3 shows the result of such experiment. The best performing parameter in terms of F1 score and recall was 10 prefix-trees for the maximum window size. We decided to include the recall to show the performance in terms of the known-drift detection, as the F1 score was strongly affected by the high amount of false positives detected. As the size of the window increased, so did the amount of false positives detected by the framework. The mean delay mimics that behavior, though a decrease is observed from 12 trees to 14.

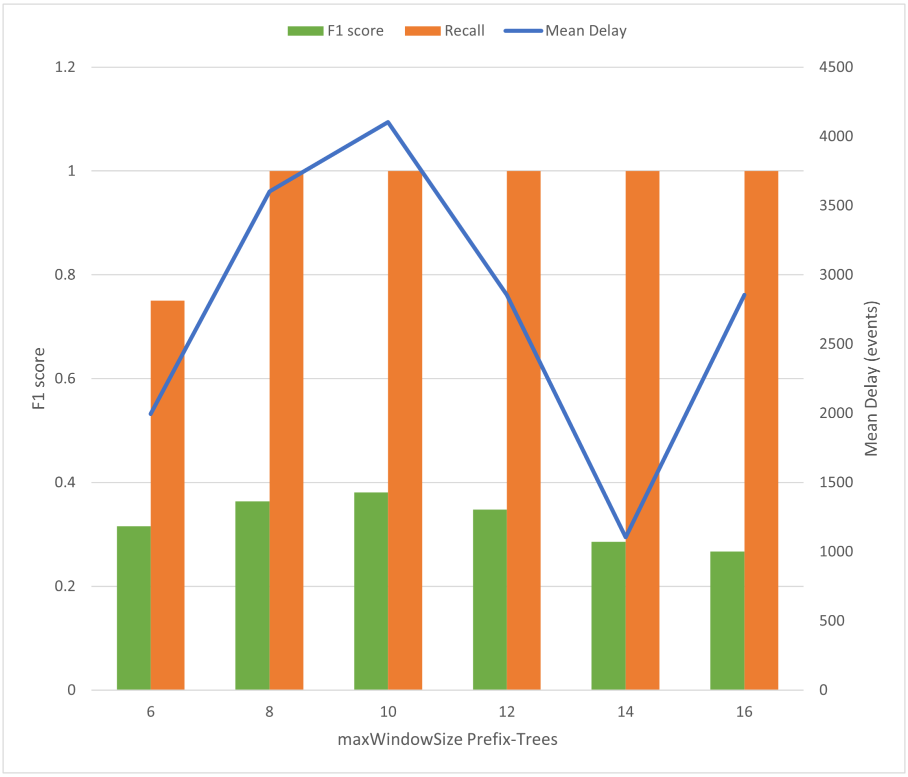


Figure 6.3: F1 score, recall, and mean delay for different *maxWindowSize* values with the BPIC 2015 dataset.

maxWindowSize (trees)	**Time of execution (minutes)**
6	4.158
8	6.64
10	9.4
12	13.88
14	14.89
16	14.41

Table 6.1: Times of execution in minutes for BPI 2015 event log using proposed framework.

Furthermore, there is a significant difference on the time of execution depending on the sizes of the window. Table 6.2 shows the execution times in minutes for the different *maxWindowSize* evaluated over the BPI 2015 event log. There is an evident increase in time as the window increases. We would then suggest to use a size which performs well in terms of quality of detection, but does not increase significantly the time of execution. The best result which fulfills these conditions would be 10 prefix-trees. We suggest the use of such value as the default for real-world datasets, as it shows a trade-off in performance over quality of detection and speed.

6.1.2 What should be the Prefix-Trees size?

The second parameter of importance is the amount of events each tree is allowed to hold (*PruningSteps*). This has the same effect as the *maxWindowSize*. The smaller/bigger the trees, the more/less susceptible it can become to different drifts. It is important to note that the framework is dealing with parallel running cases *c*. This means that the trees should be big enough so they can store the most amount of complete traces as possible, without disrupting the drift detection procedure. Not only that, but the size of the trees will also affect the frequency of the *TreePruneSteps* and *PruningSteps*. This will impact the topology of the tree, i.e. the way the log is represented within the data structure, as well as the importance of each trace after applying the decaying mechanism. Remember we want to capture as much behavior as possible in order for the trees to be an accurate representation of the window *W* of observation, and therefore, of the sub-log seen within that time period. In order to answer this question, we performed tests over the same set of logs, but now adjusting the size of *PruningSteps* attribute. Considering the inter-drift distance of 12,000 events, the sizes chosen were 200, 500, 800, and 1000 events per tree. We believe these are relevant sizes which are able to capture a significant amount of complete cases *c*. Furthermore, the reference algorithm used [1] used 800 event within their experimental evaluation. Thus, we want to confirm if such value would be the best fit for this framework over artificial datasets. For each of these sizes, we used the same value for the maximum size of the window based on the previous experiment for artificial datasets: 12 Prefix-Trees for *maxWindowSize*.

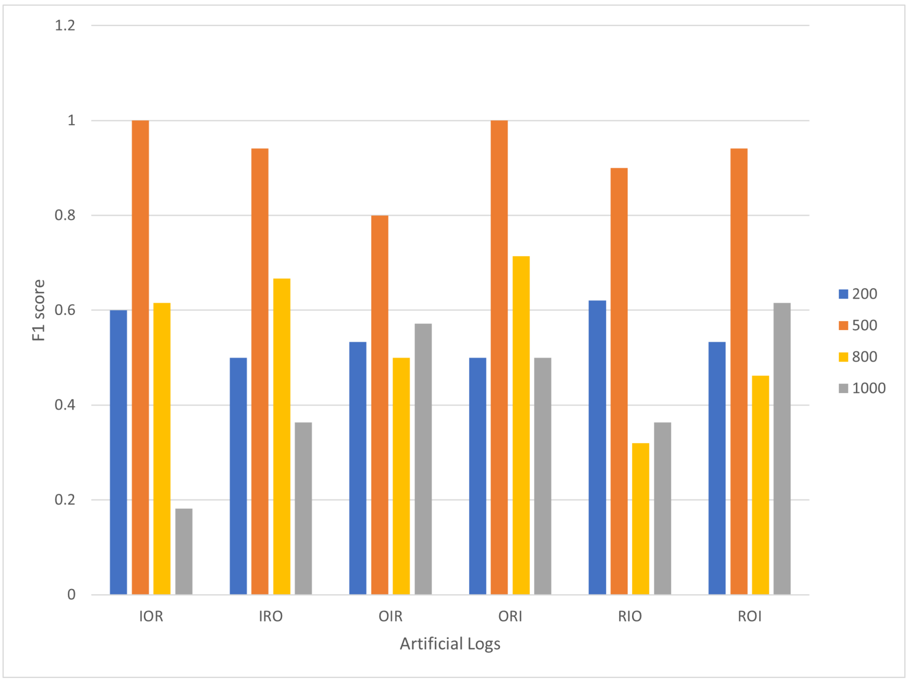


Figure 6.4: F1-score for the different *PruningSteps* sizes over the nested pattern logs from [10].

Figure 6.4 shows the results of such experiment. It shows the comparison of the F1-score from the three different tree sizes over each event log. Except for one of the logs, the results seem to behave similarly. It is clear that the prefix-trees with 500 events outperform those with smaller or bigger sizes. We believe this is correlated to the statement mentioned earlier, where a bigger size of the tree is not able to identify intra-tree drifts, while smaller sizes are very susceptible to false positives and do not take advantage of the topology when computing the distance metric. In fact, we see that trees of 200 events identify more false positives than known drifts, while trees of 1000 events are not able to identify even half of the known drifts in most cases. We believe though that the size of the cases c has a high weight over how big or small the trees should be. If a trace is very short (less than 10 events), a big tree will capture capture the process both before and after the drift. Therefore, some prior knowledge, though it is not required, can be considered when defining an accurate size for the trees. We would suggest regardless to use a *PruningSteps* of 500 events if there is no certainty on the size of the traces and their impact over the drifts over artificial datasets.

Similarly to the *maxWindowSize*, an evaluation over real-world dataset was performed to determine if such choice would also work for this type of dataset. Now because of the bigger size of the log and inter-drift distance, 200 events were excluded from the experiment and 2000 events was added to analyze the results with bigger trees. Again, the BPIC2015 dataset from 5.3.1 was chosen for the evaluation. Figure 6.5 shows the results of such experiment. We identify a similar behavior than with the previous parameters and experiment. There is an incremental trend in performance as the size of the trees increases, peaking at 1000 events per tree. Eventually, if a bigger size is used the performance in terms of mean delay of detection starts to decrease. Again, user has to analyze the trade-off between quality and speed of detection. From our perspective, we believe the default value for the tree size for real-world datasets should be 1000 events, if no data exploration tasks like these are performed. We believe these results prove that the selected parameters, in the context of this framework, perform better than the ones used by the referenced authors.

With these parameters clearly defined and proper values selected based on a data exploration procedure, the floor is set to perform a comparative analysis against unsupervised-learning online concept drift detection techniques. Same dataset used for the data exploration will be used in further experiments.

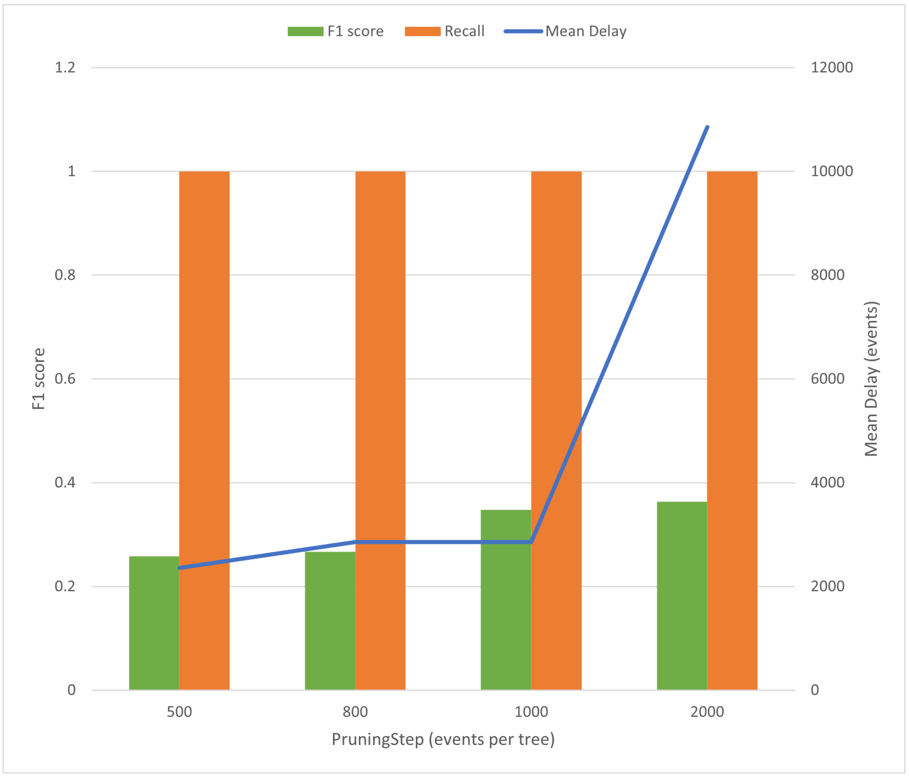


Figure 6.5: F1 score, recall, and mean delay for different *PrunigSteps* values with the BPIC 2015 dataset.

6.2 How does the proposed framework compare to competitors from a quality of detection perspective?

As stated previously, there is no relevance on introducing a new framework if it will not be able to perform, at least, to the level of the existing CDD techniques. To verify such statement, we selected three techniques reviewed in the literature from Subsection 2.3.2: *ProDrift* [10], *DOA* [13], and Robust drift detection [37].

ProDrift is a technique previously implemented as a plug-in within a well-known process mining tool *Apromore*[1]. A command line tool called ProDrift 2.5 was downloaded with its corresponding documentation for performing the experiments. As explained previously, this technique relies on defining a set of features (called runs), which are extracted from the event stream and built into a contingency matrix. Later, a Chi-squared test of independence is applied over such matrix and depending on the amount of consecutive rejected tests, it will signal a drift. For these set of experiments, the "events" drift detection method will be used, together with ADWIN as the adaptive window approach, sensitivity selected was *medium, high* and *veryhigh*, and the noise filter used was 0.
DOA relies on a different approach. It first generates (or if provided as an input) a process model using the *Heuristics Miner* [38] algorithm. While processing the event stream, it performs conformance checking over each trace against the base process model. It will then apply a local outlier factor (LOF) score over the non-conforming traces and create micro-clusters of models with these. Eventually, these models are evaluated over the non-conforming traces and therefore, signaling a drift whenever the traces are conforming with such. The tool was cloned from a github repository[2] and adjusted to perform the corresponding tests. All parameters were set as default.
Robust and Accurate drift detection [37] is an offline algorithm which works similarly to ProDrift. It performs a statistical test over a contingency matrix storing the frequency of a set of features. In contrast, the statistical test is a G-test of independence, similarly to [14], rather than a Chi-squared test. The algorithm extracts the directly-follows relations from the log and builds a contingency matrix with the corresponding data over a window of observation. This window size is fixed, and requires a $Windowsize \div 2$ number of consecutive rejected tests to signal a drift. The framework will also process the log backwards to detect, as argued by the authors, those deletion of relations that a forwards pass would not be able to. The implementation is available at their Github[3] repository. The tool uses a set of packages from ProDrift to run. The default parameters were used during all experiments, as well as a window size of 1000 events and 500 for the number of rejected tests.
Our framework "Prefix-Tree CDD", for this experiment, will use the best performing parameters for artificial datasets from previous tests: *maxWindowSize* of 12 prefix-trees and *PruningSteps* of 500 events. F1-score and recall will be used to evaluate the performance of the algorithms in terms of how well can they detect the 9 different drifts in each log.

Figure 6.6 shows the results of executing each of the four algorithms over the six different nested pattern logs. Our framework outperforms the other three algorithms in all logs. ProDrift and Robust drift detection were not able to identify the nine drifts within the logs, contrary to what they state in their corresponding experimental evaluations. The algorithms, for an unknown reason, detect drifts only up to around half of the log, while also detecting several false positives. For ProDrift, the sensitivity adjustment made no significant change over the detected drifts. The tool did detected nine drifts, but some of them were not the defined ones in the experimental setup (false positives). Though in their paper they argued an F1-score of almost 1 in all scenarios, we were unable of replicating such results. We believe that the "event" configuration of the tool does not work as expected. Tests showed that the "run" configuration did processed the complete log, though it signaled drifts after certain number of traces seen, which would not allow a comparison analysis with our approach which uses number of events. DOA showed the exact

[1]More information about Apromore at `https://apromore.org/`
[2]DOA project available at Github repository
[3]Robust and Accurate Drift Detection project available at Github repository

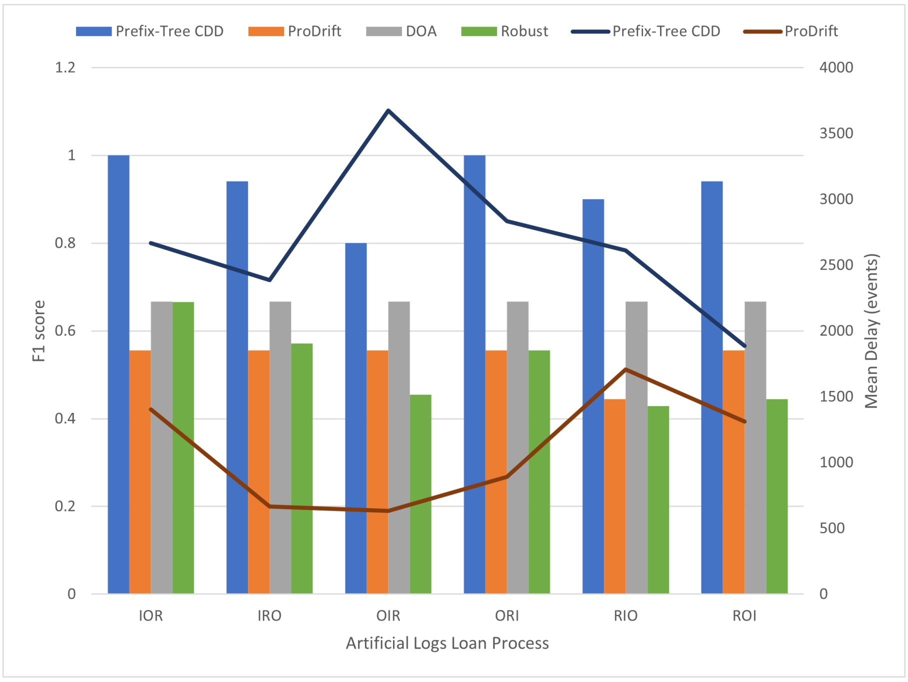


Figure 6.6: F1-score for the different techniques over the nested pattern logs from [10].

same results for each of the event logs (1500 conforming traces and 1500 non-conforming traces). One would expect that the logs, though ingesting the drifts around the same locations, would generate different models which affect the conformance checking of the event stream. Throwing the same results for six distinct logs do not give meaningful insights on how the tool is able to process different drifts. It appears that it depends completely on the location of such drifts rather than the types of drifts. Robust drift detection was able to detect some of the drifts, but struggled with all the different patterns similarly to ProDrift. We believe that the directly-follows, though it is a strong component to analyze the log structure, struggles identifying drifts related to changes in the sequence of activities (parallelization or choice) and also the introduction of loops. Furthermore, it appears that the backwards processing does not assist in identifying drifts which the forwards did not. In many cases the two passes identified the same exact drifts (true positives and false positives). Our framework, on the contrary, showed different results based on the pattern. It showed very promising results over all logs, with an F1-score equal to or above 0.8. We believe the reason behind the lower performance over the 'OIR' log is associated with the type of nested pattern that was ingested, which first adjusts the sequence of activities by creating loops. Regardless, we see a clear dominance in terms of drift detection by Prefix-Tree CDD compared to the other three nobel algorithms.

To give a complete analysis over different drift scenarios, we performed the same experiment over real-world datasets. The three datasets described in 5.3 were run over the proposed framework, ProDrift, Robust drift detection, and DOA. Figure 6.3 shows the results for the BPI Challenge 2015 dataset. The propsed framework best performed with 10 prefix-trees for the *maxWindowSize* and 1000 events for the *PruningSteps* based on the results from the previous analysis. ProDrift was not able to identify the four drifts, it identified one false positive at 19,999 events. Besides an initial window size of 10,000 events (as requested by the tool in order to run), all other parameters were set as default, with the "event" configuration and noise filter as 0. DOA on the other hand was not able to process the log completely. At around 2900 cases, while performing the conformance checking, the script failed with the error message "list index out of range" at the following script line: $global\_colors[len(global\_colors) - 1] = palette[found\_index]$. Finally, Robust drift detection threw a memory exception during the execution after around 6 hours of total run time.

Exception encountered was $Exceptioninthread"Thread - 4"java.lang.OutOfMemoryError : Javaheapspace$. Even after assigning the maximum amount of heap memory (16GB), it was not able to process the log. The command used to execute the script was: $java\ -Xmx16G\ -cp\ ".;./lib/*"\ Test.Main\ apply\ ../challenge2015\_complete.xes$ 1000 2000. Therefore, no results were given that allow further interpretation against other nobel techniques.

The BPI Challenge 2020 dataset was also utilized to perform drift detection. Figure 6.7 shows the results of this experiment using the proposed framework. Different results are shown compared to those of the BPI Challenge 2015. Here, the best performing size for the window in terms of the F1 score was of 10 prefix-trees as the *maxWindowSize*. As with the BPI 2015 dataset, a window size of 10 trees appears to detect the known-drift, while also not a high amount of false positives. Though sizes 8 and 12 were also able to detect the known-drift, each had a higher amount of false positives. Furthermore, size 6 and 8 were not able to identify the drift. We believe that for size 6, the smaller size of the window did not allow it to store a high amount of events between December 2017 and January 2018 to create trees which were significantly different. For the size 14, it appears that the high amount of events in a single window made a generalization of the process when building the aggregated trees for comparison of the distance metric.

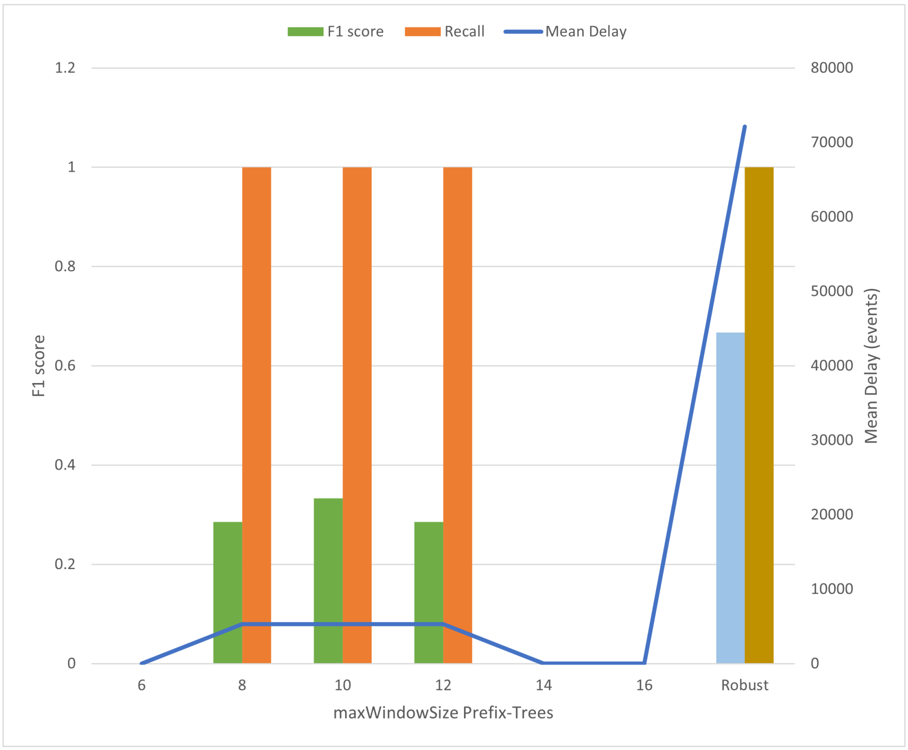


Figure 6.7: F1 score and mean delay for different *maxWindowSize* values with the BPIC 2020 dataset.

With respect to ProDrift, the tool was unable to run the script with the provided log. It threw a *NullPointerException* error from JAVA while trying to initialize the stream with the $driftdetector.ControlFlowDriftDetector\_EventStream.<init>$ function. Again, with exception of the initial window size, all parameters used were default ones. DOA once again was unable to process the complete event log, throwing the same error message as in the previous experiment, now at around 6,400 cases during the conformance checking. Robust drift detection was now not able to start processing the log. An exception thrown by the use of one of the ProDrift packages while attempting to convert the log into a stream cancelled the execution. Example of the error encountered is shown in Listing 6.2. The results seen in Figure 6.7 are based on the documentation provided by the authors of [37] within experimental results. They argue detecting two drifts, one at 12,603 events and the other at 12,426 events. Their justification for the second drift (Christmas holidays) would not be accurate against the description of the challenge. Furthermore, while analyzing the event log with Disco it was found that the drift should occur around 11,700 events, therefore signaling that this second drift should be located at the beginning of 2018 and not in 2017, as presented in their paper. Though they actually detect the single drift and just one false positive, it is important to remember that this is an offline algorithm. Therefore, the mean delay of detection corresponds to the size of the complete event log. With that in mind, we would still argue that our online detection algorithm would outperform it in terms of delay on detection, while achieving the same recall.

```
java -Xmx16G -cp ".;.\lib\*" Test.Main apply C:\\Users\\a0081\\Documents\\BDMA\\
    Thesis\\dataset\\BPI_Challenge_2020\\InternationalDeclarations.xes 1000 2000
Number of available CPU cores: 12
...\\InternationalDeclarations.xes
1000, ws: 2000
Exception in thread "main" java.lang.NullPointerException
        at org.apromore.prodrift.util.XLogManager.getEventType(XLogManager.java
    :687)
        at org.apromore.prodrift.util.XLogManager.isCompleteEvent(XLogManager.java
    :805)
        at org.apromore.prodrift.util.LogStreamer.logStreamer(LogStreamer.java:53)
        at Test.Main.main(Main.java:119)
```

Listing 6.1: Exception thrown by Robust drift detection when attempting to process the InternationalDeclarations.xes file from the BPI Challenge 2020

Finally, the merged log from the Hospital and BPI Challenge 2012 datasets was used for an experiment. The results of running such over the proposed framework are shown in Figure 6.8. Here, all max window sizes except 6 and 16 trees were able to identify the two known-drifts. Sizes 8, 10, and 12 show a slightly higher performance due to the lower amount of false positives identified. Though it is important to note that the "known-drifts" are those explained in Subsection 5.3.3, but the logs may contain several other drifts that are not previously known. A more in-depth analysis would needed to be done in order to confirm if these drifts found should be considered false positives, or if they are in fact concept drifts in the process. This task is out of the scope of this master thesis. Regarding ProDrift, the tool was able to identify only one of the two drifts, at 21,075 events. This resulted in a higher F1 score due to it not signaling any other false positives. Though this result seems better, we believe the probability of more drifts within the merged logs is quite high. It would seem very odd that a process remains constant for three years. Knowing that ProDrift couldn't identify the second known-drift, which should be very clear looking at the distribution of the runs or events, we would argue that the proposed framework performs better in terms of drift detection quality. DOA once again was unable to complete its execution with the same exception thrown at around 6000 traces. Robust drift detection again threw a *Exception in thread "Thread − 4" java.lang.OutOfMemoryError : Java heap space* exception after 4 hours of running time.

With these results, one could argue that the proposed framework outperforms ProDrift, DOA, and Robust drift detection in terms of quality of detection, in both artificial as well as in real-world datasets. It would be meaningful to perform further analysis on the real-world logs to identify other existing drifts outside of the already known. This would encourage and improve the results shown by the proposed framework.

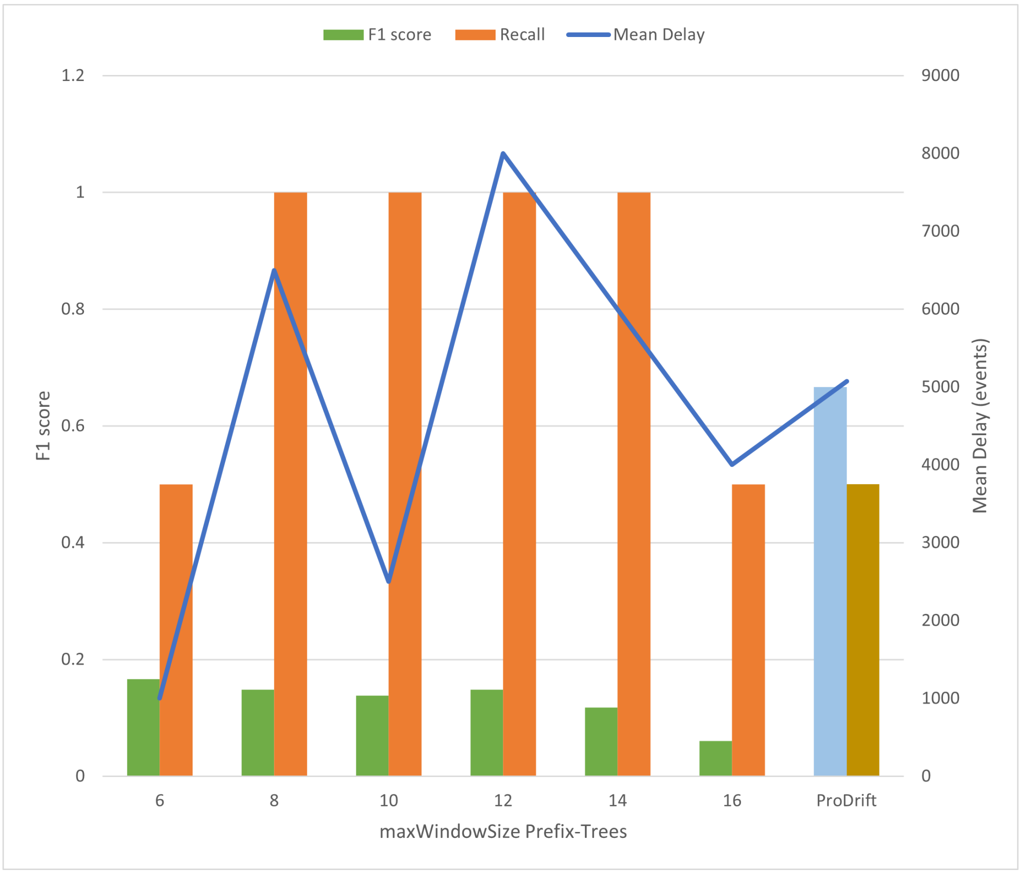


Figure 6.8: F1 score, recall, and mean delay for different *maxWindowSize* values with the Hospital & BPIC 2012 merged dataset.

6.3 How fast can the proposed framework detect drifts compared to other techniques?

With respect to the mean delay of detection the results are different. Figure 6.6 shows the results of the mean delay test performed between Prefix-Tree CDD and ProDrift over the six different event logs. It appears that ProDrift is able to detect the drifts faster in all logs. What is important to note is that the mean delay is calculated only over those drifts that were identified by the algorithms. This means that for ProDrift, the mean delay is calculated only over four or five drifts per log, as explained earlier, as seen with the F1-score results. This gives it an advantage over our framework because the average delay is compute over less drifts than with the proposed framework. With this in mind, we would still argue that Prefix-Tree CDD performs well in terms of mean delay, considering that the *maxWindowSize* used for this experiment was 12 prefix-trees. The *PruningSteps* size was also maintained at 500 events. We were not able to include DOA, nor Robust drift detection in this experiment. DOA does not output any metrics related to the delay of detection, though it requires to build a process model out of the whole event log, which could be considered as a complete log delay. Robust drift detection is an offline algorithm, i.e., the mean delay is defined as the complete event log. Therefore, any drifts identified would have the maximum mean delay of detection.

Over the real-world dataset, interesting findings regarding the mean delay of detection arose. BPI Challenge 2015 Figure 6.3 showed very similar results to the artificial datasets, where as the size of the *maxWindowSize* increase, so did the mean delay of detection. Same behavior could be seen in the BPI Challenge 2020 6.7, where the best performing sizes showed the same amount of delay. Surprisingly this was not the case for the Hospital & BPI Challenge 2012 dataset 6.8. Here, the best performing window size was 10 trees, which showed less delay than 8 and 12 trees, while detecting the two known-drifts and a similar amount of false positives. The only comparison that could be done with competitors was with ProDrift. The algorithm showed a very high delay for the single drift it was able to detect. This is another point which encourages the use of the proposed framework over competitors. Neither DOA nor Robust drift detection were able to process the complete log, with the exceptions explained in previous experiments thrown after several hours of execution.

6.4 Is the proposed framework more efficient in time of execution than other techniques?

The next spectrum to evaluate the performance of the framework against competitors was over the time of execution. Figure 6.9 shows the execution time in seconds of each technique over the six different artificial event logs. As expected, DOA has the highest times in all six logs. This probably related to the time invested in generating the base model by analyzing the whole log, and then applying a conformance check over each trace to identify the clusters and drifts. ProDrift and Robust drift detection outperform our proposed framework in time of execution over the artificial datasets. What's important to note is that our framework appears to take considerable longer times over logs where more drifts (regardless if true positives or false positives) are expected. This means that the part of the framework which appears to be more costly is the drift detection procedure, rather than the storage of the information on the prefix-trees. This supports our hypothesis of selecting the prefix-trees as a data storage because of its good performance in terms of time of execution. We believe therefore that our algorithm is able to compete with, and even outperform, most state-of-the-art techniques which rely similarly in extracting features and building some type of data structure. Furthermore, it will probably outperform those algorithms which require the discovery of a process model in order to perform the drift detection.

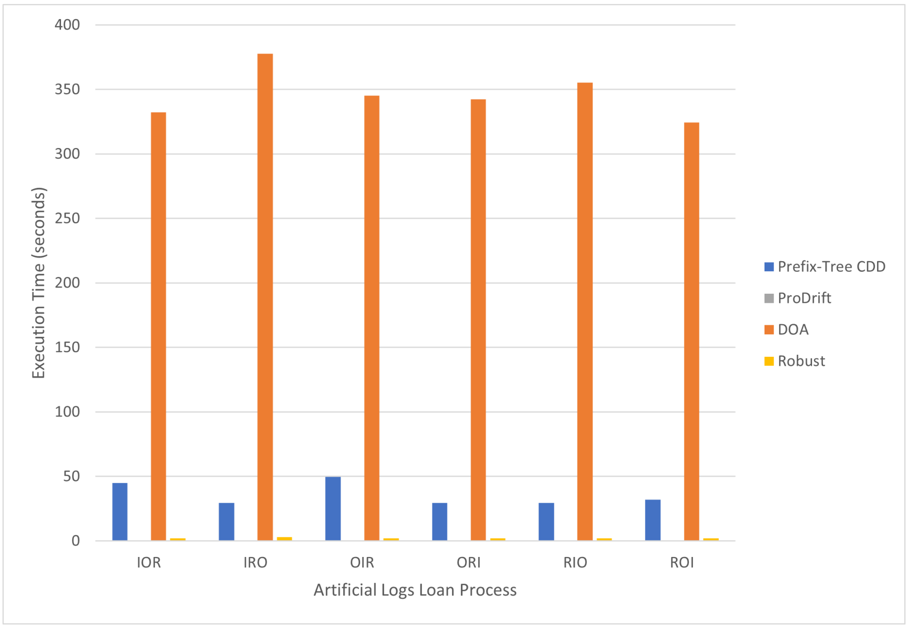


Figure 6.9: Execution time in seconds for the different techniques over the nested pattern logs from [10].

In terms of memory requirements neither ProDrift, DOA, or Robust drift detection provide information of the amount of memory used by such tools, this both in their corresponding papers as during the experiments performed. In fact, none of the techniques reviewed in Section 2.3 documented the amount of memory which their algorithms required to execute. Prefix-Tree CDD used around 387MB of memory over each of the artificial logs used. We believe though the number may be high, most concept drift detection techniques which generate data structures would require a similar amount. Furthermore, those techniques like DOA, which discover a process model and perform continues evaluations of the stream over such model, would require even more memory than the proposed framework.

With respect to the real-world datasets, due to the poor performance from the competitors in executing the code, a comparison regarding execution time and memory was not available. The proposed algorithm took around 11-20 seconds, 5-20 minutes, and 5-20 minutes, to execute for BPI Challenge 2020, 2015, and Hospital & BPI Challenge 2012 respectively, depending on the size of the *maxWindowSize* selected.

maxWindowSize (trees)	**BPIC 2015**	**BPIC 2011+2012**	**BPIC 2020**
6	4.158	4.65	0.183
8	6.64	6.41	0.216
10	9.4	8.63	0.233
12	13.88	11.43	0.225
14	14.89	13.06	0.283
16	14.41	19.35	0.25
ProDrift	∼60		

Table 6.2: Times of execution in minutes for BPI 2015, 2020, and 2011+2012 event logs using proposed framework.

In contrast, ProDrift took over 1 hour to process the BPI Challenge 2015 and the Hospital & BPI Challenge 2012 datasets each. DOA took more than 2 hours to process these same datasets before the script failed. Robust drift detection failed to process any of the logs after several hours of execution before signaling an error. Regarding memory no comparison could be done with the competitors. The proposed framework required ∼519MB of memory for the BPI Challenge 2020 logs. Due to the low performance when using the memory profiler, we excluded the experiment for the other two real-world datasets. We would still argue that due to the time of execution for both ProDrift and DOA, the would probably require more memory in order to execute. Robust drift detection threw errors in two different logs related to heap space assigned for the JAVA machine to run. This signals a high requirement of memory consumption for the algorithm to work, compare to the proposed framework.

6.5 Can the proposed framework work in a completely online scenario?

Another relevant experiment, in our opinion, is the performance of such algorithm over an actual completely online event stream. PLG2, as explained in Chapter 2 provides an adequate environment to perform such test. While executing the event stream from the process model shown in C.1, at a certain point in time the stream will be adjusted to start emitting events the evolution process model C.2. The algorithm should be able to capture a drift from this change in the process events. There is no actual way to know exactly when is the drift happening, as the stream does not provide such information. What it is known is the rate at which the tool is sending events to the port. In this case, such rate is 4500 events every 30 seconds. The idea is that around 1.5 - 2 minutes, the stream will be adjusted and hopefully a concept drift will be detected by the framework. Figure 2.3 shows the stream setup used at the start of this experiment.

Listing B shows the output of the proposed framework over the stream of events generated by PLG2.

```
ADWIN change detected at: 16000 events
Reference window size: 2000events
Test window size: 8000events
Tree distance metric: 573.2507667562994
```

Listing 6.2: Prefix-Tree CDD results for PLG2 experiment

It appears that the framework took a couple of seconds after the process change was done to detect it. Nevertheless, it was able to detect successfully the drift in a complete online setup by processing events received through a local port with a telnet connection. Such experiment is not replicable over the competitors, as many code adjustments would have to be done in order for them to process events from a telnet connection. Still, we think this is a strong point to support the use of this algorithm further in completely online scenarios.

6.6 What further information of the drifts can we obtain from the proposed framework?

As explained in Chapter 4, one component of the framework is intended to characterize the drifts detected. Basically, focusing on understanding *what* caused a drift in the underline process. To prove the usefulness of the tool in that context, the BPI Challenge 2020 dataset described in Section 5.3 was used for this experiment. The parameters used for this experiment were 10 prefix-trees and 1000 events for *maxWindowSize* and *PruningSteps* respectively.

The total execution time for processing the entire event log was 9.92 seconds. It required 519MB of memory for the whole script to run. Interestingly, the execution time and memory required to run the framework did not change regardless of the sizes of the window or the trees selected. It appears that the main factor to determine the time and memory requirements is focused solely on the size of the stream. The framework was able to detect the gold-standard drift. In total, three drifts were detected.

6.6.1 Critical Activities & Relations

The algorithm prints the most "critical activities and relations" in regards to each drift. This allows to get more details on which activities and relations had a higher impact over the drift identified. For the gold standard drift, the most critical activities and relations were (in order of importance, being 1 the most important):

1. Activity: Permit APPROVED by ADMINISTRATION
2. Relation: Permit SUBMITTED by EMPLOYEE → Permit APPROVED by ADMINISTRATION
3. Activity: Declaration SUBMITTED by EMPLOYEE
4. Relation: End trip → Declaration SUBMITTED by EMPLOYEE
5. Activity: Declaration FINAL_APPROVED by SUPERVISOR
6. Relation: Declaration SUBMITTED by EMPLOYEE → Declaration FINAL_APPROVED by SUPERVISOR
7. Activity: Request Payment
8. Relation: Declaration FINAL_APPROVED by SUPERVISOR → Request Payment
9. Activity: Request Payment

10. Relation: Declaration FINAL_APPROVED by SUPERVISOR → Request Payment
11. Activity: Declaration APPROVED by PRE_APPROVER
12. Relation: Declaration SUBMITTED by EMPLOYEE →
    Declaration APPROVED by PRE_APPROVER

Figure C.3 and C.4 in Appendix C shows an enlarged version of the process map before and after the drift discovered by the framework. These critical activities and relations make sense when looking more in detail on how the process changed after 2018. The activity "Permit APPROVED by ADMINISTRATION" was not being performed during the pilot. It was introduced after 2018 as a mandatory activity after the submission of the permit by the employee. This leads to the second most relevant node or relation: Permit SUBMITTED by EMPLOYEE → Permit APPROVED by ADMINISTRATION. Again, because such activity didn't exist before, this relation would also be newly seen in the business process. The third activity with high weight over the drift is "Declaration SUBMITTED by EMPLOYEE". This activity by itself is not very relevant in terms of the drift because it has existed since before and after the pilot. The reason why it is listed as the third most relevant is because there was a significant amount of cases c which had the trace "Start trip → End trip → Declaration SUBMITTED by EMPLOYEE". For simplicity purposes the framework will just print the last activity of the trace, but in reality the complete trace was seen several times in *W1* and none in *W0*, which means that there are several declarations after the drift which start with the trip and once the trip ends, it is immediately submitted by the employee. During the pilot there were no traces which followed such execution sequence. Similar scenarios are present with the other critical activities and relations.

For the remaining two drifts identified, further analysis of the results has to be done to determine if any of them can be considered real drifts or not. For the third drift identified for example, the sequence of "Declaration APPROVED by ADMINISTRATION → Declaration FINAL_APPROVED by SUPERVISOR → Request Payment" was seen several times *W1*, but not at all in *W0*. This means that during the observation period of *W1* there were several declarations that probably because of being incomplete, while processing them in parallel, followed that sequence of activities. Further investigation would be needed to understand if this hypothesis is correct or if there are actual declarations which follow this pattern. If the declaration approval by the administration is done in batches for example and takes a high amount of time between the submission from the employee and the approval to happen, sub-windows may consider them as new traces. Therefore, causing this drift when in reality the process has not changed, but the time invested between activities may have. This could be an area of opportunity for future work.

6.6.2 Drift Logs

The second part of the characterization exercise is to provide the logs for each drift, so that further drift analysis can be done by the users. As explained in Chapter4, the framework requires a configuration parameter to be set to perform this task. Such was done with the BPI Challenge 2020 dataset and generated 10 logs in total, two for each drift detected. A set of these logs were used to generate the process maps C.3 and C.4 with Disco. This is an example of the process discovery tasks that can be performed with the sub-logs provided, which allows a more visual analysis over the process before and after the drift.

Chapter 7

Conclusions

This master thesis introduced a new framework for unsupervised-learning onlin concept drift detection over event streams. Before describing the problem and methodology, an overview of how process mining is used on the field of stream processing was given. In particular, after presenting what concept drift detection is and the two approaches for stream learning, an in-depth description of current supervised and unsupervised learning techniques for concept drift detection was shown. These techniques display a list of limitations based on the data structure utilized, features extracted from the event log, and drift detection technique performed. Here, we introduced the main component of the framework, referring to the prefix-tree data structure introduced in a process discovery technique paper [1]. More specifically, we showed how this data structure can be used as within concept drift detection framework. First, we explained how an event stream can be stored and represented by a set of prefix-trees. Then, by applying a PCA technique over prefix-trees [2], we were able to convert our discrete information to continuous. This would then be fed to a well-known concept drift detection technique [3], which focuses on most up-to-date information and discards outdated items. The algorithm eventually signals a drift once there is a significant deviation in the distribution of the continuous data. Furthermore, the framework produces a set of drift characterizing components to give more context on what caused a drift. It also allows the users to perform their own process analysis with the sub-logs generated from the drifts identified. To demonstrate the relevance of the framework, we tested it over a set of artificial and real-world event logs, as well as an online event stream generator [21]. Results showed promising results on the use of such framework for both artificial and real-world scenarios over multivariate online event streams. Based on an data exploration exercise, the best parameters for identifying known-drifts in these set of logs were defined. The framework was able to detect all known-drifts for each of the datasets chosen. Moreover, we performed a comparison with three other nobel concept drift detection techniques applied over event streams. Analysis showed that the proposed framework outperforms the three approaches on artificial and real-world datasets in terms of quality and delay of detection, as well as execution times and memory requirements over real-world scenarios. Unfortunately, the competitors we were not able to replicate the exact results described by the authors of each technique in their corresponding papers. Nevertheless, the results obtained showed a clear support on the use of the proposed framework over such tools. In addition, the tool was able to identify drifts in a complete online scenario, while also providing detailed information about each of the drifts identified in one of the real-world event logs. In conclusion, the results of this work can be considered sufficient to (i) address the limitations presented by nobel concept drift detection techniques (ii) prove that the proposed framework outperforms a set of these nobel technqiues and (iii) provide insights on what is the cause behind the drifts detected in the process for real-world scenarios.

7.1 Future Directions

This work has raised some interesting questions which can lead to future research. The prefix-tree data structure is an specific type of graph, composed of a root node and unidirectional edges. Though the framework stored such data structure under dictionaries, this could be extended to graph databases, where graph algorithms could be potentially used as drift detection techniques. Examples of such could be centrality algorithms, community detection, similarity algorithms or path finding algorithms. Another important aspect would be to perform more comparative analysis with other drift detection techniques. Unfortunately, the selected set were not able to run the real-world datasets under reasonable memory and time constraints. A more in-depth comparison with a bigger array of techniques and additional logs would provide stronger evidence on the use of the framework. Finally, results shown over real-world scenarios were heavily affected by the amount of false positives identified on the logs. Supplementary process mining tasks, like process discovery and comparing, over the sub-logs of the identified drifts could aid on determining if such are in fact non-existing drifts or could they be considered actual drifts.

Bibliography

[1] Marwan Hassani, Sergio Siccha, Florian Richter, and Thomas Seidl. Efficient process discovery from event streams using sequential pattern mining. In *IEEE Symposium Series on Computational Intelligence, SSCI 2015, Cape Town, South Africa, December 7-10, 2015*, pages 1366–1373. IEEE, 2015. iii, ix, 14, 15, 47, 59, 66

[2] Haonan Wang and J. S. Marron. Object oriented data analysis: Sets of trees. *The Annals of Statistics*, 35(5), Oct 2007. iii, xiii, 17, 59, 74

[3] Albert Bifet and Ricard Gavaldà. Learning from time-changing data with adaptive windowing. In *Proceedings of the Seventh SIAM International Conference on Data Mining, April 26-28, 2007, Minneapolis, Minnesota, USA*, pages 443–448. SIAM, 2007. iii, 11, 16, 18, 59

[4] Wil van der Aalst and et al. Process mining manifesto. In *Business Process Management Workshops*, pages 169–194, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ix, 3, 4

[5] S.J. van Zelst. *Process mining with streaming data*. PhD thesis, Mathematics and Computer Science, March 2019. Proefschrift. ix, 6

[6] R. P. Jagadeesh Chandra Bose, Wil M. P. van der Aalst, Indrė Žliobaitė, and Mykola Pechenizkiy. Dealing with concept drifts in process mining. *IEEE Transactions on Neural Networks and Learning Systems*, 25(1):154–171, 2014. ix, 11

[7] Heitor Murilo Gomes, Albert Bifet, Jesse Read, Jean Paul Barddal, Fabrício Enembreck, Bernhard Pfahringer, Geoff Holmes, and Talel Abdessalem. Adaptive random forests for evolving data stream classification. *Machine Learning*, 106:1–27, 10 2017. ix, 12

[8] Marwan Hassani. Concept drift detection of event streams using an adaptive window. In Mauro Iacono, Francesco Palmieri, Marco Gribaudo, and Massimo Ficco, editors, *Proceedings of the 33rd International ECMS Conference on Modelling and Simulation, ECMS 2019 Caserta, Italy, June 11-14, 2019*, pages 230–239. European Council for Modeling and Simulation, 2019. ix, 1, 16, 20, 35, 41, 43, 65

[9] J. Martjushev, R. P. Jagadeesh Chandra Bose, and Wil M. P. van der Aalst. Change point detection and dealing with gradual and multi-order dynamics in process mining. In Raimundas Matulevicius and Marlon Dumas, editors, *Perspectives in Business Informatics Research - 14th International Conference, BIR 2015, Tartu, Estonia, August 26-28, 2015, Proceedings*, volume 229 of *Lecture Notes in Business Information Processing*, pages 161–178. Springer, 2015. ix, 1, 13, 20, 39

[10] Abderrahmane Maaradji, Marlon Dumas, Marcello La Rosa, and Alireza Ostovar. Detecting sudden and gradual drifts in business processes from execution traces. *IEEE Transactions on Knowledge and Data Engineering*, 29(10):2140–2154, 2017. ix, ix, ix, ix, x, x, 1, 2, 14, 20, 40, 44, 45, 47, 49, 50, 54

[11] I. Žliobaitė R. P. J. C. Bose, W. M. P. van der Aalst and M. Pechenizkiy. Dealing With Concept Drifts in Process Mining. 2014. 1, 13, 20, 33

[12] Anton Yeshchenko, Claudio Di Ciccio, Jan Mendling, and Artem Polyvyanyy. Visual drift detection for event sequence data of business processes. *CoRR*, abs/2011.09130, 2020. 1, 14, 20, 33

[13] Ludwig Zellner et.al. Concept drift detection on streaming data with dynamic outlier aggregation. In Sander J. J. Leemans and Henrik Leopold, editors, *Process Mining Workshops - ICPM 2020 International Workshops, Padua, Italy, October 5-8, 2020, Revised Selected Papers*, volume 406 of *Lecture Notes in Business Information Processing*, pages 206–217. Springer, 2020. 1, 14, 20, 33, 41, 49

[14] Alireza Ostovar, Abderrahmane Maaradji, Marcello La Rosa, Arthur H. M. ter Hofstede, and Boudewijn F. van Dongen. Detecting drift from event streams of unpredictable business processes. In Isabelle Comyn-Wattiau, Katsumi Tanaka, Il-Yeol Song, Shuichiro Yamamoto, and Motoshi Saeki, editors, *Conceptual Modeling - 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings*, volume 9974 of *Lecture Notes in Computer Science*, pages 330–346, 2016. 1, 14, 20, 49

[15] B.F.A. Hompes, J.C.A.M. Buijs, W.M.P. van der Aalst, P.M. Dixit, and J. Buurman. Detecting change in processes using comparative trace clustering. In P. Caravolo and S. Rinderle-Ma, editors, *Proceedings of the 5th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2015), Vienna, Austria, December 9-11, 2015*, CEUR Workshop Proceedings, pages 95–108. CEUR-WS.org, 2015. 5th International Symposium on Data-Driven Process Discovery and Analysis (SIMPDA 2015, SIMPDA 2015 ; Conference date: 09-12-2015 Through 11-12-2015. 1, 20, 41

[16] Rafael Accorsi and Thomas Stocker. Discovering workflow changes with time-based trace clustering. In Karl Aberer, Ernesto Damiani, and Tharam S. Dillon, editors, *Data-Driven Process Discovery and Analysis - First International Symposium, SIMPDA 2011, Campione d'Italia, Italy, June 29 - July 1, 2011, Revised Selected Papers*, volume 116 of *Lecture Notes in Business Information Processing*, pages 154–168. Springer, 2011. 1, 20

[17] Tobias Brockhoff, Merih Seran Uysal, and Wil M. P. van der Aalst. Time-aware concept drift detection using the earth mover's distance. In Boudewijn F. van Dongen, Marco Montali, and Moe Thandar Wynn, editors, *2nd International Conference on Process Mining, ICPM 2020, Padua, Italy, October 4-9, 2020*, pages 33–40. IEEE, 2020. 1, 20

[18] Abdulhakim A. Qahtan, Basma Alharbi, Suojin Wang, and Xiangliang Zhang. *A PCA-Based Change Detection Framework for Multidimensional Data Streams: Change Detection in Multidimensional Data Streams*, page 935–944. Association for Computing Machinery, New York, NY, USA, 2015. 1, 13, 14, 20, 33

[19] Ömer Gözüaçık, Alican Büyükçakır, Hamed Bonab, and Fazli Can. Unsupervised concept drift detection with a discriminative classifier. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, page 2365–2368, New York, NY, USA, 2019. Association for Computing Machinery. 1, 20

[20] Canbin Zheng, Lijie Wen, and Jianmin Wang. Detecting process concept drifts from event logs. In Hervé Panetto, Christophe Debruyne, Walid Gaaloul, Mike P. Papazoglou, Adrian Paschke, Claudio Agostino Ardagna, and Robert Meersman, editors, *On the Move to Meaningful Internet Systems. OTM 2017 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October 23-27, 2017, Proceedings, Part I*, volume 10573 of *Lecture Notes in Computer Science*, pages 524–542. Springer, 2017. 1, 13, 20

[21] Andrea Burattin. PLG2: multiperspective processes randomization and simulation for online and offline settings. *CoRR*, abs/1506.08415, 2015. 2, 7, 8, 59

[22] Dirk Fahland. Advanced process mining, September 2020. 3

[23] Charu C. Aggarwal, editor. *Data Streams - Models and Algorithms*, volume 31 of *Advances in Database Systems*. Springer, 2007. 5

[24] Andrea Burattin, Alessandro Sperduti, and Wil M. P. van der Aalst. Control-flow discovery from event streams. *2014 IEEE Congress on Evolutionary Computation (CEC)*, Jul 2014. 5

[25] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: A review. *SIGMOD Rec.*, 34(2):18–26, June 2005. 5

[26] J. Martjushev, R.P. Jagadeesh Chandra Bose, and W.M.P. van der Aalst. Change point detection and dealing with gradual and multi-order dynamics in process mining. In R. Matulevicius and M. Dumas, editors, *Perspectives in Business Informatics Research*, Lecture Notes in Business Information Processing, pages 161–178, Germany, 2015. Springer. 10

[27] E. S. Page. Continuous inspection schemes. *Biometrika*, 41(1/2):100–115, 1954. 11

[28] Vitor Cerqueira, Heitor Murilo Gomes, and Albert Bifet. Unsupervised concept drift detection using a student–teacher approach. In Annalisa Appice, Grigorios Tsoumakas, Yannis Manolopoulos, and Stan Matwin, editors, *Discovery Science*, pages 190–204, Cham, 2020. Springer International Publishing. 12

[29] João Gama, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. Learning with drift detection. In Ana L. C. Bazzan and Sofiane Labidi, editors, *Advances in Artificial Intelligence – SBIA 2004*, pages 286–295, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. 12

[30] Manuel Baena-García, José Campo-Ávila, Raúl Fidalgo-Merino, Albert Bifet, Ricard Gavald, and Rafael Morales-Bueno. Early drift detection method. 01 2006. 12

[31] Ali Pesaranghader and Herna L. Viktor. Fast hoeffding drift detection method for evolving data streams. In Paolo Frasconi, Niels Landwehr, Giuseppe Manco, and Jilles Vreeken, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 96–111, Cham, 2016. Springer International Publishing. 12

[32] Myuu Myuu Wai Yan. Accurate detecting concept drift in evolving data streams. *ICT Express*, 6(4):332–338, 2020. 12

[33] Albert Bifet and Ricard Gavaldà. Adaptive learning from evolving data streams. In Niall M. Adams, Céline Robardet, Arno Siebes, and Jean-François Boulicaut, editors, *Advances in Intelligent Data Analysis VIII*, pages 249–260, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. 13

[34] Sunanda Gamage and Upeka Premaratne. Detecting and adapting to concept drift in continually evolving stochastic processes. In *Proceedings of the International Conference on Big Data and Internet of Thing*, BDIOT2017, page 109–114, New York, NY, USA, 2017. Association for Computing Machinery. 13, 33

[35] Heitor M. Gomes, Albert Bifet, Jesse Read, Jean Paul Barddal, Fabrício Enembreck, Bernhard Pfharinger, Geoff Holmes, and Talel Abdessalem. Adaptive random forests for evolving data stream classification. *Mach. Learn.*, 106(9–10):1469–1495, October 2017. 13

[36] Viktor Losing, Barbara Hammer, Heiko Wersing, and Albert Bifet. Randomizing the self-adjusting memory for enhanced handling of concept drift. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2020. 13

[37] Yang Lu, Qifan Chen, and Simon Poon. A robust and accurate approach to detect process drifts from event streams, 2021. 14, 20, 49, 51

[38] A.J.M.M. Weijters, W.M.P. Aalst, van der, and A.K. Alves De Medeiros. *Process mining with the HeuristicsMiner algorithm.* BETA publicatie : working papers. Technische Universiteit Eindhoven, 2006. 14, 29, 49

[39] Ulrich Germann, Eric Joanis, and Samuel Larkin. Tightly packed tries: How to fit large models into memory, and make them load fast, too. 06 2009. 26

[40] Luiz F. Mendes, Bolin Ding, and Jiawei Han. Stream sequential pattern mining with precise error bounds. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008), December 15-19, 2008, Pisa, Italy*, pages 941–946. IEEE Computer Society, 2008. 26

[41] K. D. Joshi. *Foundations of Discrete Mathematics.* John Wiley Sons, Inc., USA, 1989. 32

[42] Alireza Ostovar, Abderrahmane Maaradji, Marcello La Rosa, and Arthur H. M. ter Hofstede. Characterizing drift from event streams of business processes. In Eric Dubois and Klaus Pohl, editors, *Advanced Information Systems Engineering - 29th International Conference, CAiSE 2017, Essen, Germany, June 12-16, 2017, Proceedings*, volume 10253 of *Lecture Notes in Computer Science*, pages 210–228. Springer, 2017. 36

[43] Barbara Weber, M.U. Reichert, and S.B. Rinderle. Change patterns and change support features - enhancing flexibility in process-aware information systems. *Data knowledge engineering*, 66(3):438–466, September 2008. 10.1016/j.datak.2008.05.001 ; null ; Conference date: 13-06-2007 Through 15-06-2007. 40

Appendix A

Algorithms

Algorithm 3: *StrProM* CD Detector [8]

```
Input: $S$: event stream, $W$: adaptive window, $\epsilon_{cut}$: cutting threshold
1  Initialize StrProM[T:indexed prefix-tree];
2  while $T \leftarrow$ observeStream() do
3      if pruning step then
4          $FrequencyList \leftarrow collectTreeData(T)$;
5          $pruneTree()$;
6          $W \leftarrow W + FrequencyList$;
7          if $W.size > maxWindowSize$ then
8              $w = W - W.first$;
9          while $W$ shrinks do
10             foreach $W_0W_1 \leftarrow W$ do
11                 if $\delta(W_0, W_1) > \epsilon_{cut}$ then
12                     $W = W - W.first$;
13             end
14         end
15         if size of $W$ decreased significantly then
16             $triggerCDAlarm()$;
17 end
```

Algorithm 2: The *StrProM* Algorithm[1]

```
   Input: $S$: event stream, $\lambda \in [0,1]$: decay factor, $p$: pruning period, $c_{max}$: case
          observation limit
1  Initialize the data structure $T_{traces}$, $D_{cases}$;
2  Initialize frequency maps $F_A$ and $F_R$;
3  while pruning period $p$ do
4      for $0, \dots, p$ do
5          $(c, a, t) \leftarrow observe(S)$;
6          Update $T_{traces}$ with observed event;
7          if $\exists c \in D_{cases}$ then
8              $(n, t) \leftarrow D_{cases}(c)$;
9              if $\exists T_{traces}(n.a)$ then
10                 $T_{traces}(n.a) \leftarrow T_{traces}(n.a) + 1$;
11             else
12                 $T_{traces}(n.a) \leftarrow 1$;
13             $D_{cases}(c) \leftarrow (n.a, t)$;
14         else
15             if $\exists T_{traces}(a)$ then
16                 $T_{traces}(a) \leftarrow T_{traces}(a) + 1$;
17             else
18                 $T_{traces}(a) \leftarrow 1$;
19             $D_{cases}(c) \leftarrow (a, t)$;
             /* remove least recent cases from observation */ if $D_{cases}$ reach size of
           $c_{max}$ then
20             delete oldest cases from observation;
                 /* after pruning period, collect data and prune tree */ forall
       $n.a, n.a.b \in T_{traces}$ do
21         ( /* collect pairs of nodes and their children first */) ( /* consider
            decay factor d here */) $F_A(b) \leftarrow (1-\lambda)F_A(b) + T_{traces}(n.a.b)$;
22         $F_R(a, b) \leftarrow (1-\lambda)F_R(a, b) + T_{traces}(n.a.b)$;
23     forall $a \in T_{traces}$ do
24         (                        /* children of root are still missing */)
            $F_A(a) \leftarrow (1-\lambda)F_A(a) + T_{traces}(a)$;
                 /* create new tree for next period */ initialize $T^{new}_{traces}$;
25     forall $c \in D_{cases}$ do
26         (           /* update open case pointers */) $(n.a, t) \leftarrow D_{cases}(c)$;
27         $T^{new}_{traces}(a) \leftarrow 0$;
28         $D_{cases}(c) = (a, t)$;
                 /* next period's tree has height 1 */ $T_{traces} \leftarrow T^{new}_{traces}$;
```

Algorithm 4: Method *insertByEvent* Part 1 (Handling of cases) from *PrefixTree* class

Input: *e*: event from the stream, *T*: prefix-tree, *EndDict*: end events dictionary, *W*: adaptive window, *cL*: case list, *Dcase*: ordered case dictionary

```
caseID ← e["case : concept : name"];
eventID ← e["concept : name"];
eventTimestamp ← e["time : timestamp"];
if caseID not in cL then                    /* new case that has not been seen before */
    currentNode ← T.root;
    c ← Case(caseID, currentNode);
    Dcase[caseID] ← c;
    if len(Dcase) > T.Cmax then                 /* queue of tracking cases is full */
        Dcase.pop(last = True);
    if eventID not in T.StartActFrequencies then  /* not already stored as a start activity */
        T.StartActFrequencies[eventID] = 1;
        if eventID not in currentNode.children then   /* activity not a children */
            n ← TrieNode(eventID, currentNode);
            currentNode.ch ← n;
        else
            currentNode ← currentNode.ch[eventID];
        end
    else
        T.StartActFrequencies[eventID] += 1;
    end
else if caseID not in Dcase then             /* case seen before, but not tracked */
    findNode ← T.root.loogForNode(eventID);
    if findNode = None then                  /* did not find a node with that activity */
        currentNode ← T.root;
    else
        currentNode ← findNode;
    end
    c ← Case(caseID, currentNode);
    Dcase[caseID] ← c;
    if len(Dcase) > T.Cmax then                 /* queue of tracking cases is full */
        Dcase.pop(last = True);
    if eventID not in currentNode.children then       /* activity not a children */
        n ← TrieNode(eventID, currentNode);
        currentNode.ch ← n;
    else
        currentNode ← currentNode.ch[eventID];
        currentNode.f += 1
    end
else                                                   /* case is being tracked */
    Dcase.move_to_end(caseID);
    if eventID not in currentNode.children then       /* activity not a children */
        n ← TrieNode(eventID, currentNode);
        currentNode.ch ← n;
    else
        currentNode ← currentNode.ch[eventID];
        currentNode.f += 1
    end
end
```

Algorithm 5: Method *lookForNode* from the *TrieNode* class

```
Input: eventID: activity to find, findNode: object where parent node will be stored
1 foreach $event, node$ in $node.children$ do
2     if $eventID = event$ then          /* the event is a children of this node */
3         $findNode \leftarrow node$;
4         return $findNode$;
5     else                               /* recursively call the same function */
6         return $node.lookForNode(eventID, findNode)$;
7     end
8 end
```

Algorithm 6: Method *insertByEvent* Part 2 (Ending Traces) from *PrefixTree* class

```
Input: currentNode: current node either created or existing, EndEventsDic: dictionary
       storing each case, its end activity, and timestamp
1 $T.treeNodes.append(current)$;
2 $Dcase[caseID].node \leftarrow currentNode$;
3 $pruningCounter+ = 1$;
4 if eventID and eventTimestamp for this case in the EndEventsDic are the same then
    /* this is the end event of the case */
5     $Dcase[caseID].fl = False$;
6     if eventID not in T.EndActFrequencies then  /* the activity is not registered
        as an end activity of the tree */
7         $T.EndActFrequencies[eventID] = 1$;
8     else                                 /* increase frequency of existing */
9         $T.EndActFrequencies[eventID]+ = 1$;
10    end
11 $traceCounter+ = 1$;
12 $currentNode \leftarrow T.root$;
```

Algorithm 7: Method *insertByEvent* Part 3 (Tree Prune Step) from *PrefixTree* class

```
Input: pruningCounter: number of events seen for this tree, FAold, FBold: global
       dictionaries defined in StrProm which store frequencies of activities and relations
1 if pruningCounter % T.TreePruneSteps = 0 then /* tree pruning step achieved */
2     $T.pruneTree()$;
3     $FAold, FBold = T.decayTree(FAold, FBold)$;
4     $T.resetFrequencies()$;
5 $traceCounter+ = 1$;
6 $currentNode \leftarrow T.root$;
```

Algorithm 8: Method *PruneTree* from *PrefixTree* class

```
Input: n: start from root node of the tree to prune
1 foreach $event, node$ in $n.children$ do
2     $currentNode = node$;
3     $current.pruneRoots(T)$;
4 end
```

Algorithm 9: Method *PruneRoots* from *TrieNode* class

```
Input: $n$: node of the tree to prune, $T$: prefix-tree structure to prune
1  $T.nodeFrequencies[n.branchId] += n.f$;
2  foreach $event, node$ in $n.children$ do
3      if $node$ exists then                              /* there are children nodes */
4          $T.RelationFrequencies[n.branchId, node.branchId] += node.f$;
5          $node.pruneRoots(T)$;
6  end
```

Algorithm 10: Method *DecayTree* from *PrefixTree* class

```
Input: FAold, FBold: global dictionaries defined in StrProm which store frequencies of
       activities and relations, $T$: prefix-tree structure to decay
1  $FAnew = DecayingFunction$ Equation 2.3;
2  $FBnew = DecayingFunction$ Equation 2.3;
3  foreach $key, value$ in $T.NodeFrequencies$ do
4      $FAnew[key] += value$;
5  end
6  foreach $key, value$ in $T.RelationFrequencies$ do
7      $FBnew[key] += value$;
8  end
9  return FAnew, FBnew;
```

Algorithm 11: Method *ResetFrequencies* from *PrefixTree* class

```
Input: $T$: prefix-tree structure to reset the frequencies
1  Initialize $T.NodeFrequencies$;
2  Initialize $T.RelationFrequencies$;
3  Initialize $T.StartActFrequencies$;
4  Initialize $T.EndActFrequencies$;
5  foreach $n$ in $T.TreeNodes$ do
6      $n.f = 0$;
7  end
```

Algorithm 12: Method *insertByEvent* Part 4 (Pruning Step) from *PrefixTree* class

```
Input: pruningCounter: number of events seen for this tree, FAold, FBold: global
       dictionaries defined in StrProm which store frequencies of activities and relations,
       $T$: prefix-tree structure to prune, Dcase: ordered dictionary of tracking cases
1  $T.EventsSeen \leftarrow pruningCounter$;
2  $T.NodeFrequencies \leftarrow FAold.copy()$;
3  $T.RelationFrequencies \leftarrow FBold.copy()$;
4  $Dcase = T.cleanCases(Dcase, W)$;
5  if $Dcase$ empty then                                  /* all cases are closed */
6      $currentNode = T.root$;
7  else                                                  /* active cases remaining */
8      $lastEventSeen = T.FirstCaseElement(Dcase)$;
9      $currentNode = T.root.ch[lastEventSeen]$;
10 end
```

Algorithm 13: Method *insertByEvent* Part 5 (Prefix-Tree added to Window) from *PrefixTree* class

```
Input: T: prefix-tree structure to store, W: adaptive window to store the prefix-tree
       (continuation of Algorithm 12)
1  if len(W.PrefixTreeList = W.MaxWindowSize) then   /* Max window size reached */
2      Delete last prefix-tree in W.PrefixTreeList;
3  $W.PrefixTreeList.append(T)$;
4  $W.cddFlag = True$;
5  Reset global variables;
6  $T.reset()$;
```

Algorithm 14: Function to perform CDD.

```
Input: W: adaptive window to storing the prefix-trees
1  if W.cddFlag = True) then                     /* Change detection flag is active */
2      if len(W.prefixTreeList = W.WinSize) then  /* Size of list reached maximum */
3          $W.conceptDriftDetection()$;
4          $W.WinSize = MIN(W.WinSize + 1, W.maxWindowSize)$;
5          if len(W.prefixTreeList = W.WinSize) then      /* No drift was detected */
6              Delete last prefix-tree in W.PrefixTreeList;
```

Algorithm 15: Method *conceptDriftDetection* from *Window* class (Part 1)

```
Input: W: adaptive window to storing the prefix-trees
1  $indexSlider = 1$ while indexSlider ¡ len(W.prefixTreeList) do       /* Through every
     partition of W */
2      $W_0 \leftarrow W.prefixTreeList[START : indexSlider]$;
3      $W_1 \leftarrow W.prefixTreeList[indexSlider : END]$;
4      $Window_0, Window_1 = W.buildContinMatrix(W_0, W_1)$;
5  end
6  $treeDistance = prefixTreeDistances(Window_0, Window_1)$;
7  $indexSlider = driftDetection(treeDistance, W, indexSlider)$;
```

Algorithm 16: Function *driftDetectionADWIN*

```
Input: W: adaptive window to storing the prefix-trees, a: ADWIN object, m: metric
       value (distance between two trees), indexSlider: slider to adjust the
       sub-population of W
1  $a.add\_element(m)$;
2  if a.detected_change() then                  /* ADWIN detected a concept drift */
3      Remove prefix-trees of W.prefixTreeList from sub-population $W_0$;
4      $W.WinSize = len(W.prefixTreeList) + 1$;
5      $indexSlider \leftarrow 1$;
6      $W.cddFlag \leftarrow True$;
7  else                                               /* active cases remaining */
8      $indexSlider += 1$;
9      $W.cddFlag \leftarrow False$;
10 end
11 return indexSlider;
```

Algorithm 17: Method *buildContinMatrix* from *Window* class

```
Input: $W_0$: list with set of prefix-trees from sub-population $W_0$, $W_1$: list with set of
       prefix-trees from sub-population $W_1$
1  $W_0PT, W_1PT = PrefixTree()$;
2  $W_0TreeLists, W_1TreeLists = W.subWindowTree()$;
3  foreach $tree$ in $W_0$ do                                /* For each tree in the $W_0$ */
4      $W_0TreeLists.winNodeFreq.append(tree.nodeFrequencies)$;
5      $W_0TreeLists.winRelFreq.append(tree.relationFrequencies)$;
6  end
7  foreach $tree$ in $W_1$ do                                /* For each tree in the $W_1$ */
8      $W_1TreeLists.winNodeFreq.append(tree.nodeFrequencies)$;
9      $W_1TreeLists.winRelFreq.append(tree.relationFrequencies)$;
10 end
11 $W_0PT.nodeFrequencies = W_0TreeLists.Freq.mean()$;
12 $W_0PT.relationFrequencies = W_0TreeLists.winRelFreq.mean()$;
13 $W_1PT.nodeFrequencies = W_1TreeLists.winNodeFreq.mean()$;
14 $W_1PT.relationFrequencies = W_1TreeLists.winRelFreq.mean()$;
15 $Window_0 \leftarrow W_0PT.nodeFrequencies \cup W_0PT.relationFrequencies$;
16 $Window_1 \leftarrow W_1PT.nodeFrequencies \cup W_1PT.relationFrequencies$;
17 return $Window_0, Window_1$
```

Algorithm 18: Method *conceptDriftDetection* from *Window* class (Part 2)

```
Input: $W$: adaptive window storing the prefix-trees
1  if $W.cddFlag = True$ then                              /* If a drift was detected */
2      Save information of the drift;
3      $drift = Drift(Drift\ information)$;
4      $W.driftsIdentified.append(drift)$;
```

Algorithm 19: Method *conceptDriftDetection* from *Window* class (Part 3)

```
Input: $criticalThreshold$: threshold set to define if a node is critical or not.
1  $treeDistance = prefixTreeDistances(Window0, Window1)$;
2  if $W.cddFlag = True$ then                              /* If a drift was detected */
3      $criticalNodes = node\ from\ treeDistance.notInterDict\ if\ value^2 >$
         $criticalThreshold$;
4      $print(criticalNodes)$;
```

Algorithm 20: Function *logProvider*

```
Input: $log$: complete log processed as stream, $W$: adaptive window storing the
       prefix-trees
1  foreach $drift$ in $W.driftsIdentified$ do
2      $W0Start = drift.eventsSeen - drift.refWinSize$;
3      $W0End, W1Start = drift.eventsSeen$;
4      $W1End = drift.eventsSeen + drift.testWinSize$;
5      $W0Sublog = log[W0Start : W0End]$;
6      $W1Sublog = log[W1Start : W1End]$;
7      $save(W0Sublog, W1Sublog)$;
8  end
```

Appendix B

Python Scripts

```python
def __init__(self):
    self.reset()
    self.lambdaDecay = 0.25
    self.pruningSteps = 1000
    self.treePruneSteps = self.pruningSteps / 4
    self.Cmax = 6 # Max number of cases to track concurrently
    self.TPO = 1
```

Listing B.1: INIT method for *PrefixTree* class

```python
def reset(self):
    self.root = TrieNode()
    self.nodeFrequencies = dict()
    self.relationFrequencies = dict()
    self.startActFrequencies = dict()
    self.endActFrequencies = dict()
    self.treeNodes = []  # List which holds all the nodes of the trees
```

Listing B.2: "Reset" method for *PrefixTree* class

```python
def resetFrequencies(self):
    self.root = TrieNode()
    self.nodeFrequencies = dict()
    self.relationFrequencies = dict()
    self.startActFrequencies = dict()
    self.endActFrequencies = dict()
    for node in self.treeNodes:
        node.frequency = 0
```

Listing B.3: "ResetFrequencies" method for *PrefixTree* class

```python
def pruneTree(self):
    for event, node in self.root.children.items():
        current = node # Current becomes the children
        current.pruneRoots(self) # Prune the roots of the tree
```

Listing B.4: "PruneTree" method for *PrefixTree* class

```python
def decayTree(self, FAold, FBold): # Decay the frequencies of the old nodes and add the new frequencies
    FAold2 = {k: v * (1 - self.lambdaDecay) for k, v in FAold.items()} # Decay for node list
    FBold2 = {k: v * (1 - self.lambdaDecay) for k, v in FBold.items()} # Decay for relations list
    for k, v in self.nodeFrequencies.items():
        FAold2.setdefault(k, 0)
        FAold2[k] += v # Add new frequencies for nodes
    for k, v in self.relationFrequencies.items():
        FBold2.setdefault(k, 0)
        FBold2[k] += v # Add new frequencies for relations
    return FAold2, FBold2 #, startActiv2, endActiv2
```

Listing B.5: "DecayTree" method for *PrefixTree* class

```python
def cleanCases(self, Dcase, window):
    current = self.root
    activeCases = OrderedDict({caseID: caseObject for caseID, caseObject in Dcase.items() if caseObject.active == True})  # If it is an active case add it to the new case list
    for case, casObject in activeCases.items():
        eventID = casObject.node.activity
        if eventID not in current.children.keys(): # If the last activity is not already a children of the root
            # current = current.createOrCheckNode(window, eventID)
            current.children[eventID] = TrieNode(eventID, current) # Leave the last activity as nodes in the tree
        else:
            current = current.children[eventID]
            casObject.node = current  # Point the case to the existing node of the tree

    return activeCases
```

Listing B.6: "CleanCases" method for *PrefixTree* class

```python
def firstCaseElement(self, Dcase):
    '''Return the first element from an ordered collection
       or an arbitrary element from an unordered collection.
       Raise StopIteration if the collection is empty.
    '''
    return next(iter(Dcase.values()))
```

Listing B.7: "FirstCaseElement" method for *PrefixTree* class

```python
def __init__(self, activity='root', parentNode=None):
    self.nodeId = uuid.uuid1()  # Random identifier for the node
    self.activity = activity  # Activity name for the event stored in the node
    self.parent = parentNode  # Parent node/events
    self.parentList = []  # Create list of all parent nodes for this node
    self.fillParentList()  # Fill the list with the previous parents and the new parent
    self.children = dict()  # Children nodes/events are stored in a dictionary
    self.frequency = int()  # Frequency of the event
    if self.parent:
        self.branchId = ','.join(self.parentList) + "," + activity
```

Listing B.8: INIT method for *TrieNode* class

```python
if self.parent:  # If the node has a parent node
    self.parentList = self.parent.parentList.copy()
    self.parentList.append(self.parent.activity)
```

Listing B.9: "FillParentList" method for *TrieNode* class

```python
tree.nodeFrequencies.setdefault(self.branchId, 0)
tree.nodeFrequencies[self.branchId] += self.frequency  # Add frequency to FA (nodes
    )
for event, node in self.children.items():
    if node:  # If there's children nodes and it's not the root node
        tree.relationFrequencies.setdefault((self.branchId, node.branchId), 0)
        tree.relationFrequencies[
            (self.branchId, node.branchId)] += node.frequency  # Add frequency to
    FB (relations)
        node.pruneRoots(tree)  # Repeat the pruning for the children
```

Listing B.10: ”PruneRoots” method for *TrieNode* class

```python
def lookForNode(self, eventID, findNode):
    for event, node in self.children.items():
        # if node: # If there's children nodes and it's not the root node
        if eventID == event: # If the event of the node is the same as the one we'
    re searching for
            findNode = self
            return findNode
        else:
            return node.lookForNode(eventID, findNode)

    return None
```

Listing B.11: ”LookForNode” method for *TrieNode* class

```python
def __init__(self, caseIdentifier, caseNode):
    self.caseId = caseIdentifier  # Case name
    self.node = caseNode # Node/Event to which the case is pointing to
    self.active = True # Case active/inactive flag
```

Listing B.12: INIT method for *Case* class

```python
def prefixTreeDistances(Window0, Window1):
    win0NotInWin1 = {k: v ** 2 for k, v in Window0.items() if k not in Window1.keys
    ()}
    win1NotInWin0 = {k: v ** 2 for k, v in Window1.items() if k not in Window0.keys
    ()}

    lenW0NotW1 = len(win0NotInWin1)
    lenW1NotW0 = len(win1NotInWin0)

    dI = lenW0NotW1 + lenW1NotW0

    interDict = {k: ((Window0[k] - Window1[k]) ** 2) for k in set(Window0) & set(
    Window1)}
    interSum = sum(interDict.values())

    notInterSum = sum(win0NotInWin1.values()) + sum(win1NotInWin0.values())

    notInterDict = {**win0NotInWin1, **win1NotInWin0}
    sortedNotInterDict = sorted(notInterDict.items(), key=lambda x: x[1], reverse=
    True)

    totalTreeDistance = dI + ((interSum + notInterSum) ** (1/2))
```

Listing B.13: Euclidean distance between two trees [2]

```python
def __init__(self, win0NotInWin1, win1NotInWin0, interDict, interSum, notInterDict,
     notInterSum, treeDistance):
    self.win0NodesNotInWin1 = win0NotInWin1
    self.win1NodesNotInWin0 = win1NotInWin0
    self.interDict = interDict
    self.interSumOfFreq = interSum
    self.notInterDict = notInterDict
    self.notInterSumOfFreq = notInterSum
    self.treeDistanceMetric = treeDistance
```

Listing B.14: INIT method for *TreeDistance* class

```python
def __init__(self):
    self.winNodeFreq = []
    self.winRelFreq = []
```

Listing B.15: INIT method for *subWindowTree* class

```python
def __init__(self, refWinSize, testWinSize, refTree, testTree, treeDistance,
    eventsSeen, criticalNodes):
    self.refWinSize = refWinSize
    self.testWinSize = testWinSize
    self.refTree = refTree
    self.testTree = testTree
    self.treeDistance = treeDistance
    self.eventsSeen = eventsSeen
    self.criticalNodes = criticalNodes
```

Listing B.16: INIT method for *Drift* class

Appendix C

Process Models

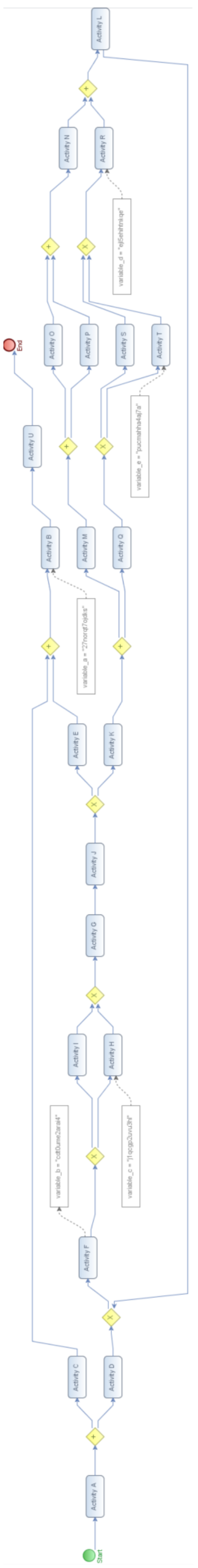


Figure C.1: PLG2 artificial process model. Containing 21 activities and 12 gateways.

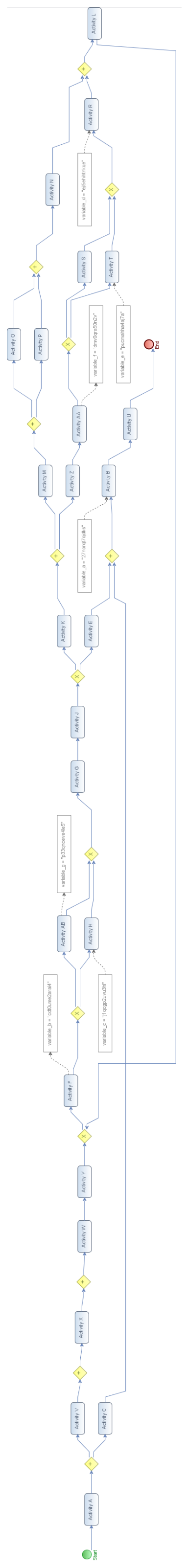


Figure C.2: PLG2 artificial process model evolved. Containing 25 activities and 14 gateways.

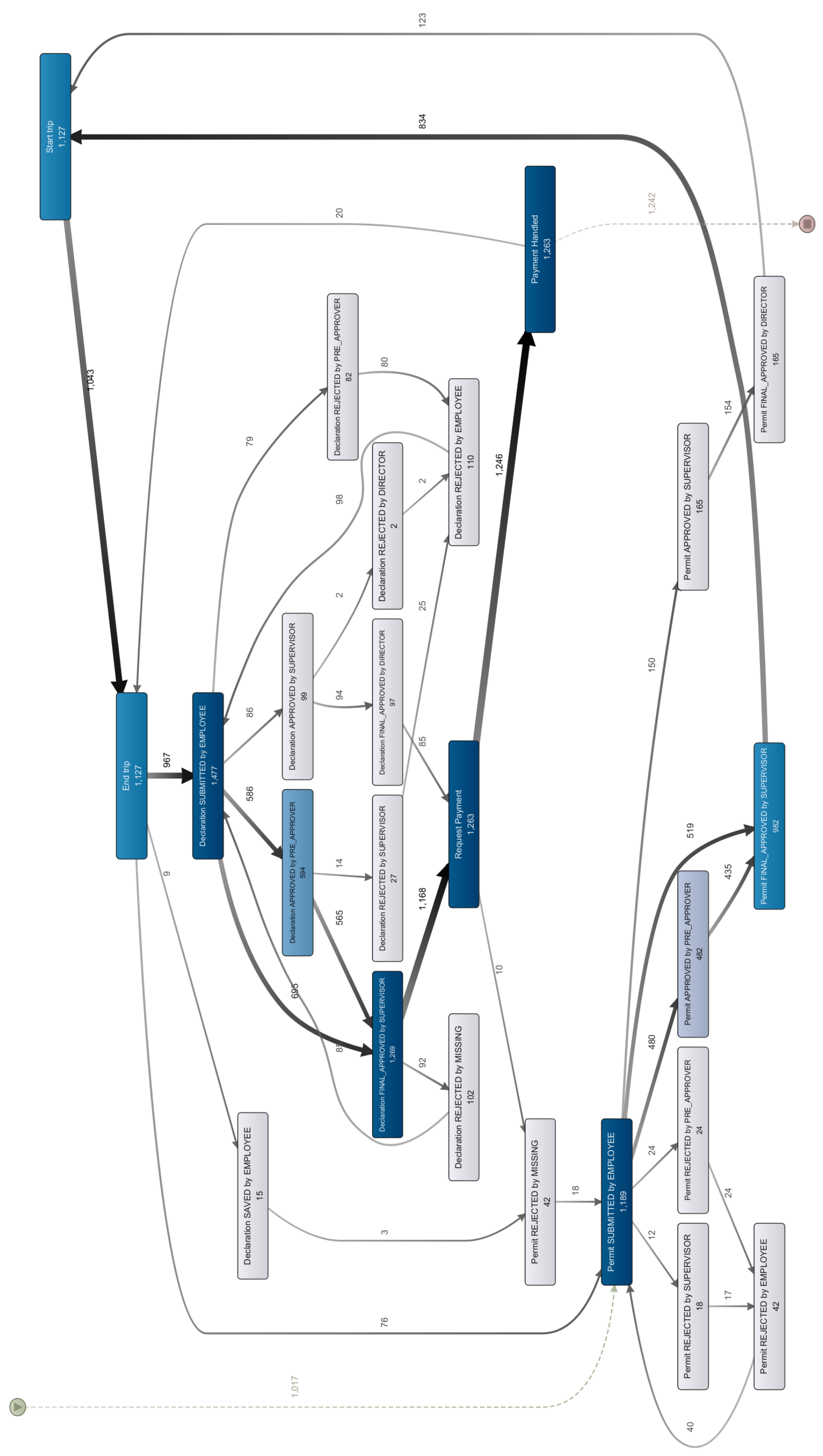

Start trip
1,127
End trip
1,127
Declaration SUBMITTED by EMPLOYEE
1,477
Declaration APPROVED by SUPERVISOR
99
Declaration APPROVED by PRE_APPROVER
594
Declaration REJECTED by PRE_APPROVER
82
Declaration REJECTED by DIRECTOR
2
Declaration FINAL_APPROVED by DIRECTOR
97
Declaration REJECTED by SUPERVISOR
27
Declaration FINAL_APPROVED by SUPERVISOR
1,269
Declaration REJECTED by EMPLOYEE
110
Declaration REJECTED by MISSING
102
Declaration SAVED by EMPLOYEE
15
Request Payment
1,263
Payment Handled
1,263
Permit REJECTED by MISSING
42
Permit SUBMITTED by EMPLOYEE
1,189
Permit APPROVED by SUPERVISOR
165
Permit FINAL_APPROVED by DIRECTOR
165
Permit APPROVED by PRE_APPROVER
482
Permit REJECTED by PRE_APPROVER
24
Permit FINAL_APPROVED by SUPERVISOR
982
Permit REJECTED by SUPERVISOR
18
Permit REJECTED by EMPLOYEE
42
123
834
1,043
20
1,242
967
79
80
98
2
86
586
94
2
25
85
14
565
695
87
92
1,168
1,246
10
9
3
76
18
150
154
519
435
480
24
12
24
17
40
1,017

Figure C.3: Process map from the International Declarations process before the known drift.

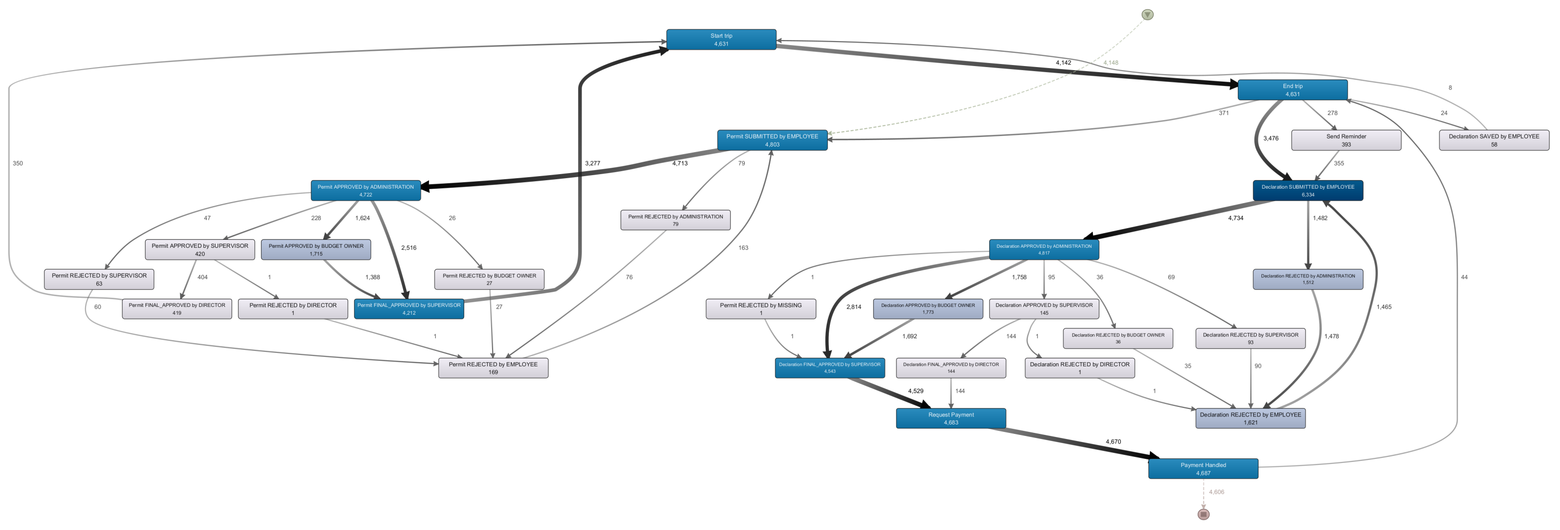

Start trip
4,631
End trip
4,631
Send Reminder
393
Declaration SAVED by EMPLOYEE
58
Permit SUBMITTED by EMPLOYEE
4,803
Declaration SUBMITTED by EMPLOYEE
6,334
Permit APPROVED by ADMINISTRATION
4,722
Permit REJECTED by ADMINISTRATION
79
Declaration APPROVED by ADMINISTRATION
4,617
Declaration REJECTED by ADMINISTRATION
1,813
Permit APPROVED by SUPERVISOR
420
Permit APPROVED by BUDGET OWNER
1,715
Permit REJECTED by SUPERVISOR
63
Permit REJECTED by BUDGET OWNER
27
Permit FINAL_APPROVED by DIRECTOR
419
Permit REJECTED by DIRECTOR
1
Permit FINAL_APPROVED by SUPERVISOR
4,212
Permit REJECTED by EMPLOYEE
169
Permit REJECTED by MISSING
1
Declaration APPROVED by BUDGET OWNER
1,773
Declaration APPROVED by SUPERVISOR
145
Declaration REJECTED by BUDGET OWNER
36
Declaration REJECTED by SUPERVISOR
93
Declaration FINAL_APPROVED by SUPERVISOR
4,543
Declaration FINAL_APPROVED by DIRECTOR
144
Declaration REJECTED by DIRECTOR
1
Declaration REJECTED by EMPLOYEE
1,621
Request Payment
4,683
Payment Handled
4,687
4,142
4,148
8
24
278
371
3,476
355
350
3,277
4,713
79
47
228
1,624
26
2,516
404
1
1,388
60
27
76
163
1
4,734
1,482
1,758
95
36
69
1,465
44
2,814
1,692
144
1
35
90
1,478
4,529
144
1
4,670
4,606

Figure C.4: Process map from the International Declarations process after the known drift.

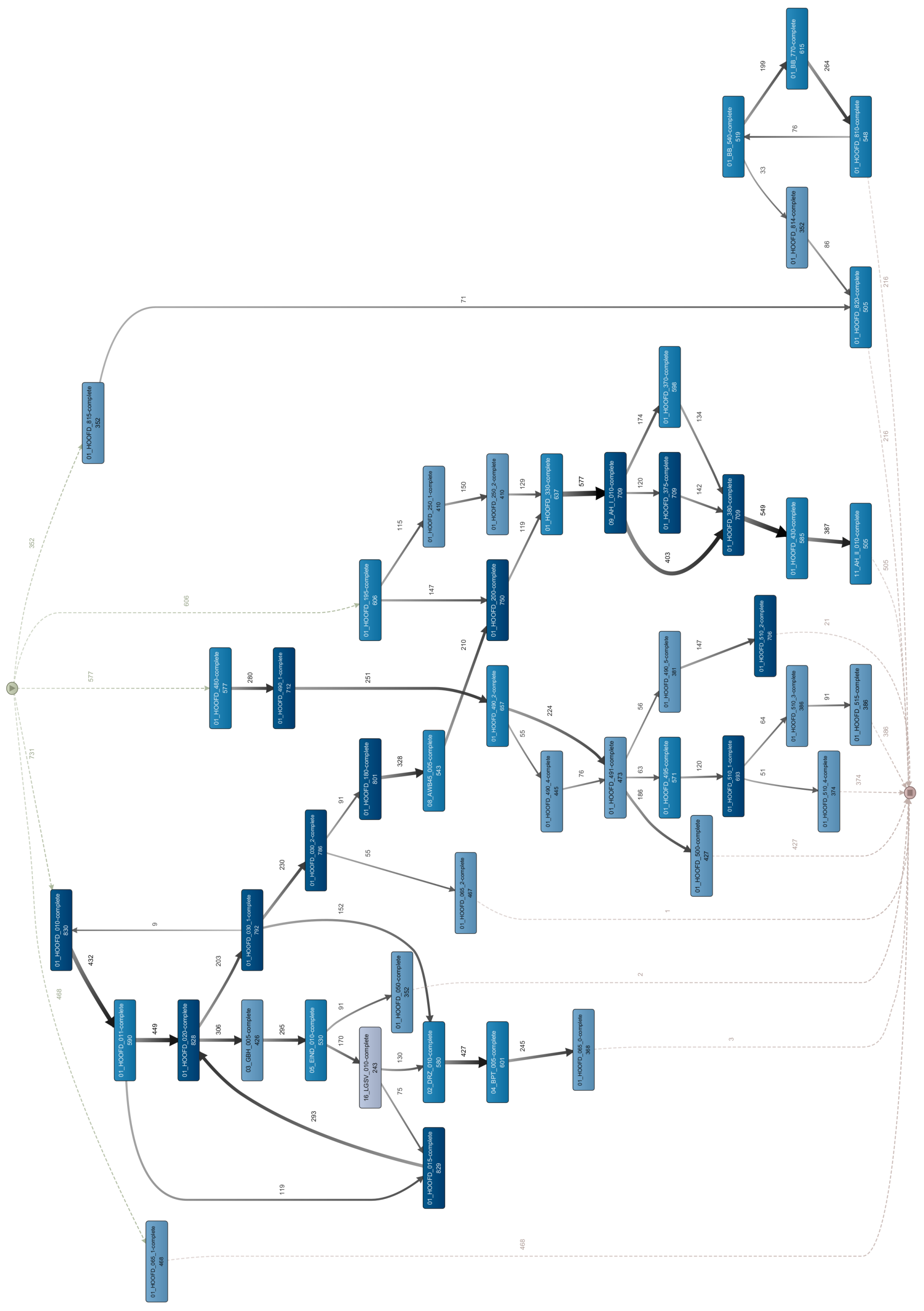


Figure C.5: Process map for the second Dutch municipality of the building permit application process (BPIC2015).

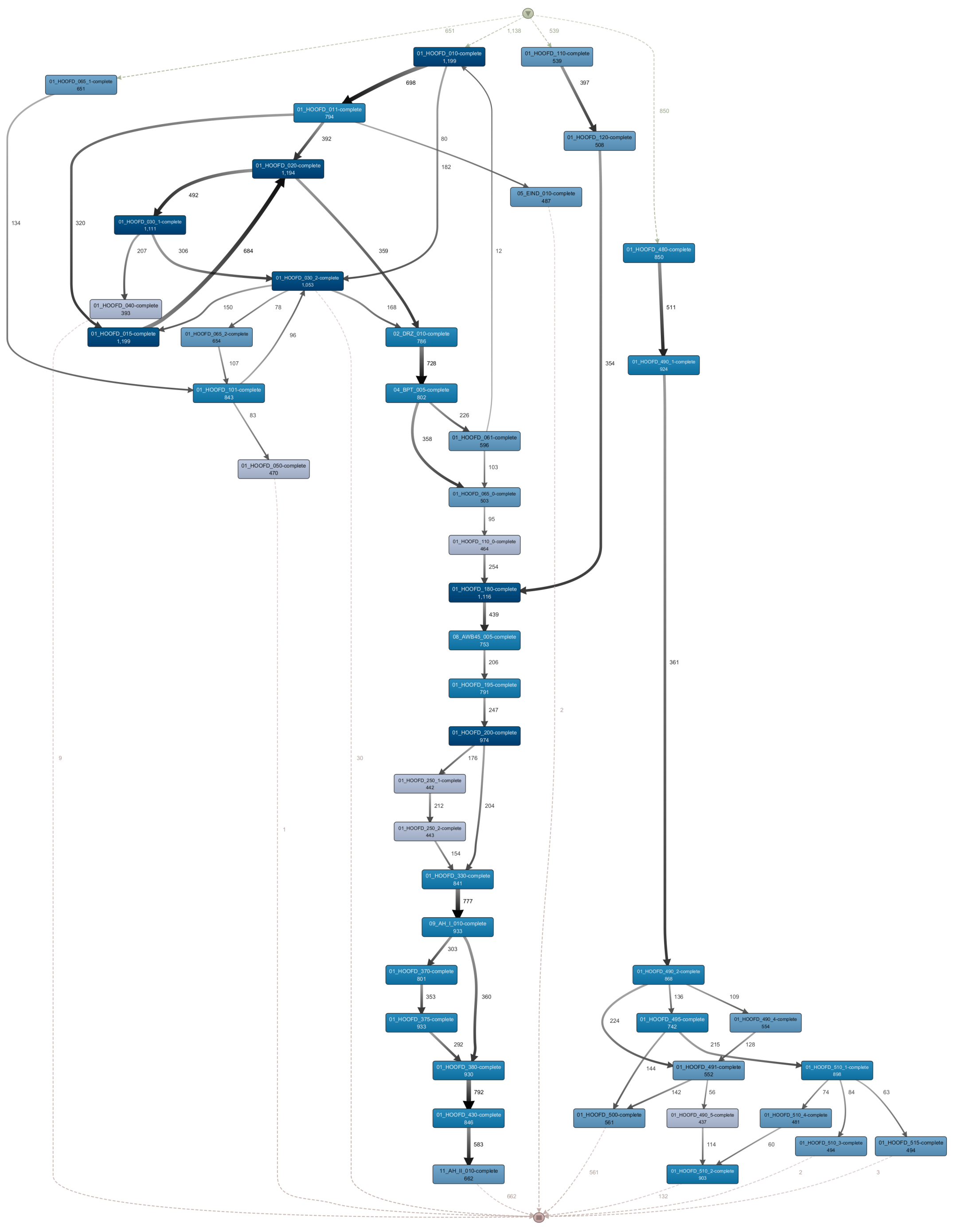


Figure C.6: Process map for the first Dutch municipality of the building permit application process (BPIC2015).