BACHELOR

Online Next Activity Prediction Using ADWIN Concept Drift Detection and Prefix Trees

Kosciuszek, Teddy L.

Award date:
2022

Online Next Activity Prediction Using ADWIN Concept Drift Detection and Prefix Trees

Thaddeus Kosciuszek

Eindhoven University of Technology, (TU/e)

Abstract. Existing research in process mining yields efficient means of detecting concept drift as well as making incremental updates to process models, thereby improving performance in predicting the next event in a business process. This study explores the improvements presented by combining existing state of the art methods of both incremental process model updates and concept drift detection in a realistic simulated streaming environment whereby the process model is updated on the fly as changes in the form of drifts are detected. Using metadata from the detection of such drifts, a framework is presented that performs on par with or better than existing methods in terms of accuracy when tested on the 10 datasets included in this analysis.

Keywords: process mining · incremental learning · event prediction · business process · concept drift · online processing

1 Introduction

Process mining is a field that involves using modeling techniques to quantify, characterize, predict, and make estimates regarding events that occur in a specific type of flow through a business process. Former research has looked further into the updating of these models when changes are detected in the underlying business process by using a method called incremental updating [1]. Other research has developed frameworks to build an efficient representation of processes using tree structures [2]. This paper seeks to combine these two methods listed in an on the fly manner as events are observed. This way, drifts can be detected and accounted for when updating a process model. Using tree structures to detect drifts combined with incremental updates to the model, the combination works to maintain optimal performance of the model. Once changes are detected new updates are be made. As variation in incoming data stabilizes after a concept drift, updates and training are considered to be less important for maintaining accuracy until new changes are identified and the model is again updated to reflect such changes.

1.1 Problem Formulation

While much of the existing research covers extensive applications in the field of process mining, no studies were found which combine the online detection of concept drifts with dynamic setting of training parameters. There is a significant amount of research that has detailed methods used to represent business processes efficiently alongside other analyses identifying changes in such processes [2, 3, 4, 5]. [2] includes online methods for analyzing event streams which we will discuss in Section 2.4. However, searching through previous work yielded no results for dynamically setting parameters or utilising the identification of concept drifts to update underlying next activity prediction models. Exploring the combination of the methods of incremental updating and online prediction may yield improved accuracy with optimised running times. This analysis is setup to explore the combination of these two methods to determine if results follow such expected improvements. Additional digging led to the discovery that Pauwels et al [1] only skimmed the surface of the importance of certain design choices in their processes and how such choices impact the evaluation results of such a process model.

1.2 Contribution

This report combines the existing methods presented by Pauwels et al. [1] and Huete Guzmán, J. S. [2]. In doing so, the main goal of this analysis is to identify factors that influence overall performance when retraining next activity prediction models in an online environment. The result is a framework that performs well in correcting for concept drift in datasets with known drifts. In the experiments of [1], calculation of drifts was carried out ahead of time and could not be considered to mirror the performance of online computation while identifying and correcting for drifts. The advantage of previous studies provided for instantaneous knowledge of when drifts occur after precalculating drifts across an entire dataset. Our results show improvements upon those results alongside drifts which have been corrected for in "real-time" as they occur naturally in an event stream which is introduced in Seciton 2.4.

1.3 Research Questions

This analysis is setup to determine if there are any means by which concept drifts and associated metadata of the identified drifts can be utilised to improve performance of a next activity prediction model. The developed methods are created to dynamically alter input parameters to an algorithm which makes incremental updates to some prediction model. Dynamic updating methods are used to determine whether the presented generalised framework can optimize the running time across many datasets while maintaining or improving upon accuracy. Below, the specific research questions guiding this explorative analysis are introduced.

1. How do the inputs to a concept drift detection algorithm affect its performance with respect to number of drifts detected and the amount of time needed to detect such drifts [2]?
2. How does the frequency with which we update a process model impact its performance [1]?
3. How does the training size with which we update the model impact its performance [1]?

4. Are we able to combine update frequency and training size into a dynamically updating, generalised framework that performs well across datasets?
5. How does such a framework perform compared to baselines immediately after identifying a concept drift?

We first introduce the context of process mining within which this framework is presented. The literature and research context of this analysis are covered in Section 2. The Preliminaries Section begins with some key terms such as events, activities, and traces which are key to understanding the very core building blocks upon which process models operate. We then build upon this foundation with streaming processes and concept drift. Tying this all together, we cover the state of the art in process mining. The most recent literature found covers prefix tree concept drift detection, incremental updates, and the SDL model selected for use in this analysis.

Next we move on to the development of the Methods of the proposed framework in Section 3. Development is shown using 8 datasets for exploration and ideation. Deeper exploration into research questions 1, 2, and 3 are detailed in the Methods (Section 3) where the relevant findings are shown. We also present here a link to survival and reliability analysis with the use of the Weibull distribution (Section 2.10).

In the Evaluation (found in Section 4), we layout the findings and answers to research questions 4 and 5. To test the performance of our developed solution, we test across all 8 datasets used in the development phase. We additionally provide results for a test against a newly created merged dataset made up of data from development. Lastly we use one external, untested dataset to validate on unseen data.

2 Preliminaries and Literature Review

2.1 Process Mining:

There are typically a few main reasons to utilise process modeling techniques to mine an underlying process from some dataset. Common objectives include predicting what the next activity is, or how long until the next activity takes place [6].

Approximations of the next event can in the case of an unknown process be used to estimate the ordering and flow of the underlying process. This predictive process provides value in the sense that businesses or other entities are able to streamline their operations with a better understanding of what the next step in some process might be. As such these entities are better able to allocate resources to accommodate expected requirements [7]. By estimating these values, businesses and other entities can make better predictions of expected timing, flow, and events which will occur within their business processes.

Another example is from Spenrath Y. et. al's work regarding online prediction of consumer behavior [8]. When a business is able to keep up with changing consumer behavior and preferences about wider ranges of consumers, it can adapt to new trends and avoid stagnating. One final example where process mining provides helpful insight is that of locating bottlenecks in online event streams [9]. Identifying bottlenecks where flows through a business process stop at some point can help to identify inefficiencies and pain points in the customer experience. Next, common vocabulary in the field of process mining is introduced.

2.2 Event

An event e is the individual row-level record that holds m attributes $(a_1, a_2, ..., a_m)$ which are captured at the time of the event occurrence. Hassani et al [3] provides a concise mathematical representation of an event as:

$$e = (c, r, a, t) \tag{1}$$

The event is described here as a tuple which contains multiple attributes. For the purpose of this analysis, events have been abridged to only include the case c, the role of the department completing the event r, activity or event type a, and the time at which the event occurred t. We also note the total space of all events as E. The overall set of events is stored in a multiset which is referred to as a *process log* [3]. Continuing to build upon the definitions provided by Hassani, we will refer to functions which project each of the attributes as follows: $c(e) = c$, $r(e) = r$, $a(e) = a$, $t(e) = t$.

2.3 Trace

Moving forward from the foundation of an event, we define a trace (sometimes also referred to as a case) as a set of events which is temporally ordered by t, all sharing the same case identifier c [3]. Formalizing the definition of a trace, we present the character < meaning "directly follows". When two events $e_1 = (c_1, r, a, t_1)$ and $e_2 = (c_2, r, b, t_2)$ are correctly ordered according to their timestamps $t(e)$: $t_1 < t_2$ where $c_1 = c_2$ (corresponding to the same case/being of the same trace) and no e_3 exists where $t_1 < t_3 < t_2$.

2.4 Process Log and Streaming Processes

A process log is the overall record containing all of the events (and as such traces) in process data. A log file not only consists of many events but also many traces. The log is ordered by time and can be considered a mapping $S : \mathbb{N} \rightarrow \mathbb{E}$ defining an *event stream*. When examining any two events e_1 and e_2 where $S(m) = e_1$ and $S(n) = e_2$, the mapping holds if $t(e_1) < t(e_2)$ for $m < n$ [3]. This can be simplified through the following representation of an *eventstream*:

$$S =< e_1, e_2, ... > \tag{2}$$

Streaming process logs provide the ability to be analysed in real time as the data is acquired as opposed to previously recorded data whereby all the available data is known at the time of analysis.

2.5 Concept Drift:

Concept drift is the phenomenon of the constant changing of the underlying process behind a model. "Processes can change with respect to three main process perspectives: control-flow, data, and resource" [2]. [4] presents two points needed to successfully deal with concept drifts: capturing the characteristics of those traces as well as identifying when those characteristics change. Drifts create difficulties in classifying future information as the accuracy in predictions output by the model lose their relevance and performance as time progresses. This occurs as the current observed time $t(e_i)$ grows farther from the time the model was last updated $t(e_{update})$ There are four different classifications of drifts according to [5]: sudden drift, gradual drift, recurring drift, and incremental drift. We give brief explanations of these types of drifts below:

1. *Sudden drift*: corresponds to substituting an existing process with a new process as an instantaneous change
2. *Gradual drift*: is when the current process is replaced by a new process. Both processes can coexist at the same time while the first is gradually discontinued.
3. *Recurring drift*: occurs when processes reappear and disappear, oscillating back and forth. This can demonstrate seasonal influence that may or may not be periodic.
4. *Incremental drift*: is a substitution between two processes via small incremental changes, normally undergoing sequences of quality improvement.

One topic of concern when working with process models is that of *catastrophic forgetting* [10]. The newly trained data corrects for concept drift over time as the performance of a process model $\hat{p}$ decreases. However, when asked to recall the same predictions seen far before the newly trained time, the accuracy suffers and it is unable to provide the same performance as seen before. This is considered to be the downfall of "forgetting" previously trained data.

Due to the heavy resources invested in acquiring data, training a model, and predicting results, catastrophic forgetting is undesirable. To identify previously seen trends such as in the case of a *recurring drift*, models must be retrained if catastrophic forgetting occurs, leading overall to more resources being used to maintain performance. Catastrophic forgetting is a phenomenon process models generally hope to avoid. However, catastrophic forgetting must be taken into consideration when building and checking the performance of such a model.

2.6 Prefix Trees

Background literature reveals that extensive work has gone into determining the most efficient ways to represent a process. Intuitively, the more optimised the storage method of the process, the faster an algorithm can parse through and analyze the underlying process. In the case of big data, it can become unreasonable to store all historical data to build a representation of a process due to storage sizes, running times, and efficiency of comparison to other processes. Recent literature details the representation of such prefixes (or former events) in processes with many different forms. Some have tried abstract domains to represent prefixes as multidimensional polygons in the case of the apron library [11, 12, 13]. Others have used a tree structure as seen in [2]. Huete Guzmán, J. S. developed such a tree comparison structure for use in identifying concept drifts. In collaboration with the TU/e, he named this method *PrefixTreeCDD* [2]. This tree method is considered to be the state of the art in representing a process model in an efficient way, yet still having detailed metadata regarding the underlying process. We detail the specifics of using prefix trees for concept drift detection below.

A prefix tree can be used to represent the possible outcomes that some modeled process $\hat{p}$ can follow. In applying this back to the idea of concept drift, two trees are maintained with respect to some time t in the event stream, with the number of leaves (in this case referred to as nodes with no children) signifying the number of possible outcomes which a trace can follow. Two examples of different prefix trees are shown in Figure 1. To construct such trees, the algorithm uses what is known as an adjustable window, also known commonly as ADWIN methods. The adjustable window is used to define which subsections of events to examine when comparing to subsets of an event stream. This is discussed further below in section 2.7. [2] describes a formal representation of the prefix tree as composed of nodes n holding the following attributes:

$$n = (i, a, p, pL, ch, f) \tag{3}$$

Whereby n is the prefix tree, i is a random unique identifier for the node, a is the activity label, p is the parent node or predecessor, pL is the list of parents up until the root node, ch is a dictionary of all children nodes of this node, and f is the frequency of this node (being the amount of times this event has been seen) [2].

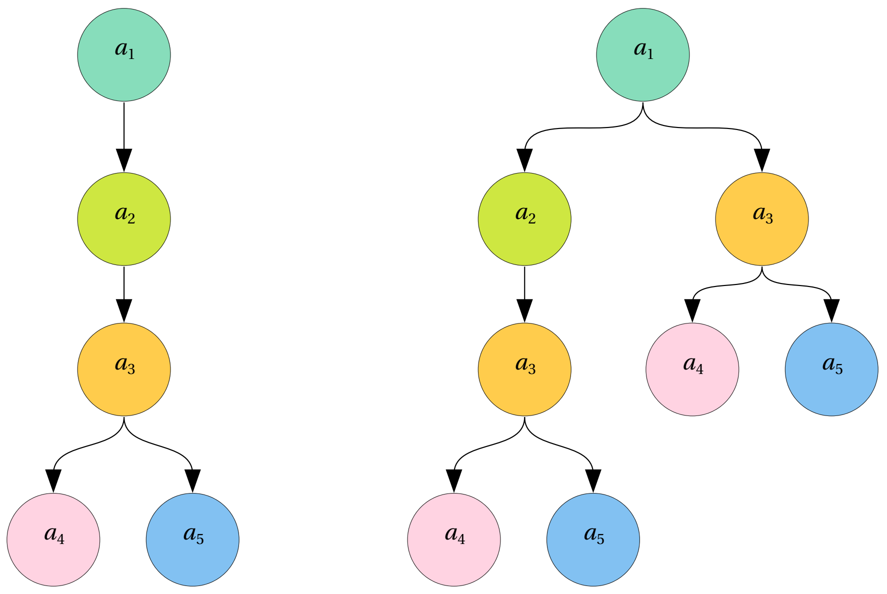


Fig. 1: Two Examples of Prefix Trees (Adapted from [1])

2.7 Adjustable Windows & Concept Drift Detection

Adjustable windows, otherwise known as ADWIN methods are an additional tool which allow for comparisons within two subsets of events known as windows. The method compares the two windows, denoted as: W_{i-1}, W_i within the same stream over time. By joining an adjustable window with the methods of a prefix tree (from Section 2.6), a "distance metric" is created through which a concept drift can be identified. The value of this distance metric determines how significant a change from the previously modeled process has occurred, with larger distances signifying a greater change in the process.

The function of an adjustable window operates in a way such that two windows are compared to each other with a prefix tree representing the process observed from the events contained in each window. If no differences are identified between the two windows when using statistical tests for comparison, the test window grows until it hits the maximum size of the test window parameter. This parameter is set by the user in the *PrefixTreeCDD* algorithm. When no changes are identified and the window size of the test window has already reached the maximal set parameter, the test window begins sliding forward in time. The sliding action drops events from the beginning of the test window which expire as they leave its context.

Once a change is identified at any point in this process, a new reference window is created at the point of the change and the testing window moves beyond the identified change and is set back to the minimum window size. The reasoning is that the longer it takes for a change to be detected, the more gradual the change is expected to be (although not necessarily). On the contrary, once a change is detected, another may occur very quickly and be missed if the test window is too large. A visualization of this is displayed in Figure 2.

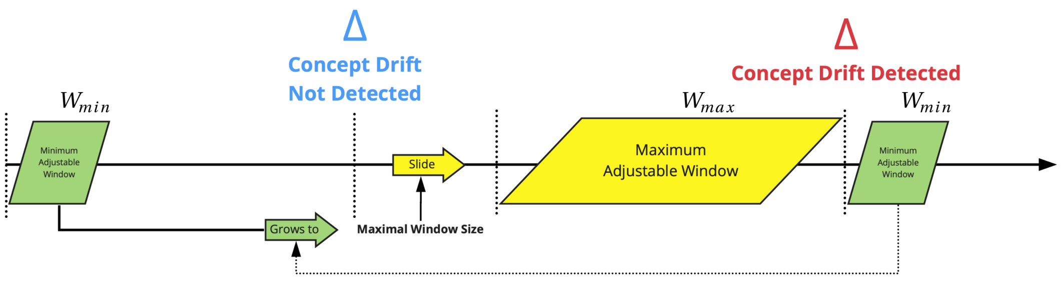

Δ
Concept Drift Not Detected
Δ
Concept Drift Detected
W_{min}
Minimum Adjustable Window
Grows to
Slide
Maximal Window Size
W_{max}
Maximum Adjustable Window
W_{min}
Minimum Adjustable Window

Fig. 2: Adjustable Window Procedure (Figure adapted from [3])

Methods to detect concept drift are well studied with many different approaches to detecting and dealing with drifts present in the literature. *PrefixTreeCDD*'s simple yet efficient operation in representing and comparing traces in process mining has led us to select this method for detecting drifts for the purpose of dynamic model updates. Huete Guzmán found that although *PrefixTreeCDD* was not able to outperform the *ProDrift* algorithm in terms of identification delay when a drift occurs, it far surpasses *ProDrift* and many others in terms of accurate and efficient identification of drifts.

While the underpinnings of how Guzman's framework was developed is not the immediate concern of this paper, we do quickly give a brief overview of its operation. Using the f value found in the node attributes of the prefix trees covered in Section 2.6. Huete Guzmán, J. S. was able to identify drifts by looking for differences between the prefix trees created in two windows. To test for drifts a statistical test is then run to determine if the observed values in the first and second windows differ more than some threshold value. We utilise the resulting distance metric that is the output of the statistical test when comparing the frequency f and ordering of nodes in two trees.

2.8 Incremental Updates

Incremental updating is the process of fitting new data to a previously trained model without having to retrain the model again. This process involves using data which is coming from a test set in the case of a static model or from the actual incoming data for which there is now a known outcome for. When retraining a model, all of the previous computation is lost when it is overwritten by a new model. On the contrary, when performing incremental updates, there is the opportunity to incorporate new data into the seen context of the model by updating the weights which have been learned by the previously trained model. Not only does this take significantly less time than full retraining, but it also means that the most recently fitted data has become the most relevant (leading to more accurate predictions). [1] tests the performance of both incremental updates and retraining with respect to accuracy and running time. The findings suggest that incremental updates provide a boost in performance in both metrics. One key point missing from Pauwels et al.'s work however, was the optimization and dynamic setting of these parameters. Their work predetermined the drift points on a static dataset and presented the improvements of updating the model exactly at those drift points. We propose a solution to address these shortcomings below in Section 3 through combining Pauwels' methods with those of Huete Guzmán, J. S.'s *PrefixTreeCDD*.

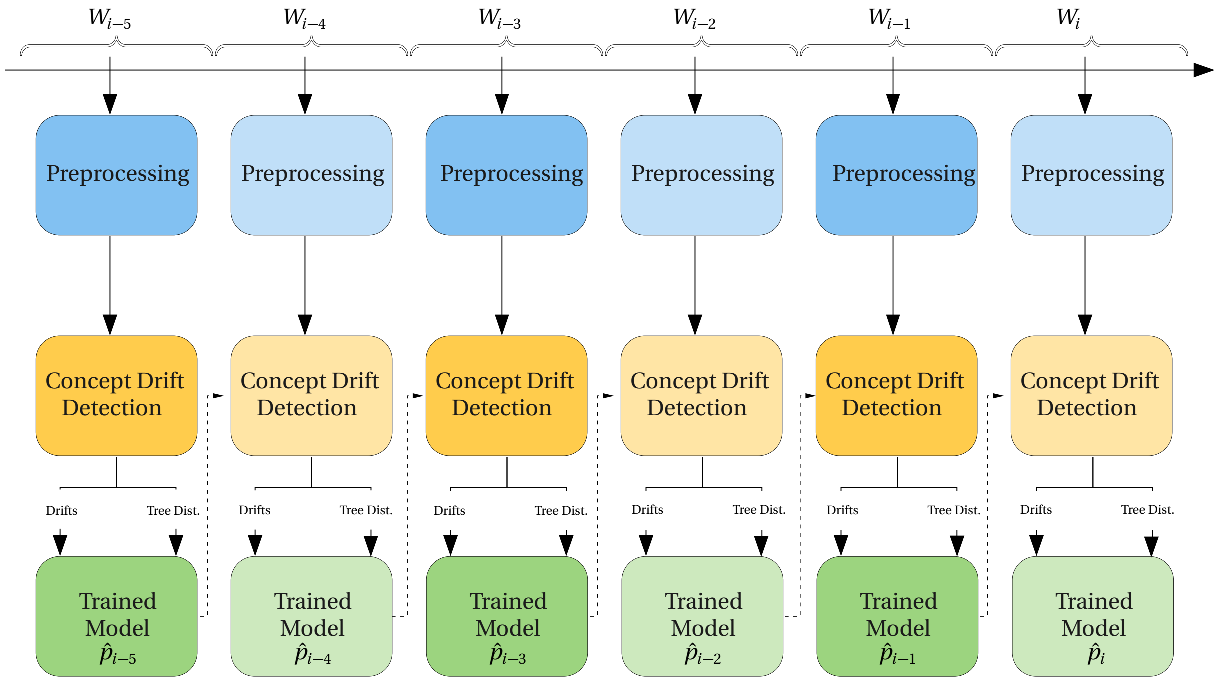

W_{i-5}
W_{i-4}
W_{i-3}
W_{i-2}
W_{i-1}
W_i
Preprocessing
Concept Drift Detection
Drifts
Tree Dist.
Trained Model $\hat{p}_{i-5}$
Trained Model $\hat{p}_{i-4}$
Trained Model $\hat{p}_{i-3}$
Trained Model $\hat{p}_{i-2}$
Trained Model $\hat{p}_{i-1}$
Trained Model $\hat{p}_i$

Fig. 3: Incremental Updating of Concept Drift - Ideal Conditions

2.9 Single Dense Layer Network

The Single Dense Layer (henceforth referred to as "SDL") is a neural network developed by Pauwels et al. specifically for process models [1]. Pauwels et al. specifically developed these SDL methods for the purpose of providing a light and thus fast framework to facilitate an accurate process model. SDL performs on par with other similar methods such as long-short term memory (LSTM), DBN, and CNN which lag in comparison by means of either lower accuracy or higher runtimes. As such Pauwels et al. [1] concluded that SDL was all around the state of the art for process mining and prediction based on its optimised runtimes and accuracy in prediction. We now present a brief explanation of the workings of the SDL neural network model. The SDL is made up of a number of input layers which are fed into a concatenation layer, then a dense layer (hence the name), and finally a softmax layer. The softmax layer predicts the probability of each of the categorical outputs in the layer. In this case the categorical outputs are the number of unique activities $\#a(e)$ which an event can assume in the data. The event with the highest output probability is then selected as the predicted event in the process. An illustration of this concept can be seen in Figure 4. Pauwels' work uses an input with a $k = 10$ concept prefix. The concept prefix is set such that inputs to the model include the events and other meta-data such as role, and time of event for the current event and $k = 10$ previous events. As such, each input to the SDL model is given the current attributes for: activity, role, and time alongside the last $k = 10$ for each of these attributes within each trace.

2.10 Survival, Reliability, & Weibull Function

Here we introduce some of the foundations of survival and reliability analysis. While initially its relevance may seem obscure, we will clear up the connection to the field in the Methods found in Section 3. A brief introduction into survival and reliability analysis yields us a series of tools to use to develop a formula in parallel with what we expect from the data. In 1958 Edward L. Kaplan and Paul Meier published a report on using incomplete observations to develop estimates of probabilities of population survival [14, 15]. These estimates came to be known as the Kaplan Meier survival curve and are still widely used today to examine differences between two or more populations when analyzing individual factors relating to survival of different groups. One function very commonly used in survival and reliability analysis is the Weibull distribution due to its versatility. Within its distribution it is able to take on the form of the exponential function while also offering the flexibility to accommodate a number of other curve shapes. The two most common forms of this function are shown below in Formulas 4 and 5 [16]. Additionally Table 1 has been provided to describe the definitions of different parameter values within the equations

$$F(t) = 1 - \exp(-(\frac{t-\tau}{\alpha})^{\beta}) \tag{4}$$

$$F(t) = 1 - \exp(\lambda(t-\tau)) \tag{5}$$

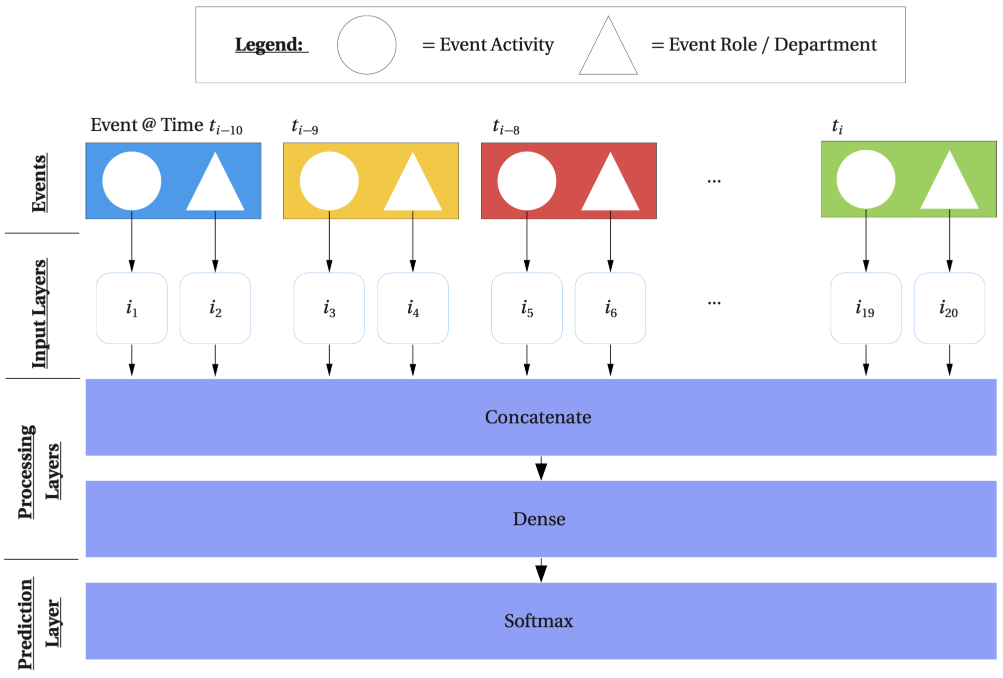


Fig. 4: Single Dense Layer Network Diagram (Adapted from [1])

For some more background information on the Weibull distribution, we introduce some former applications whereby it has been used effectively. In Chen et. al. the Weibull function was used to determine the survival curves of Lysteria monocytogenes at different hydrostatic pressure levels [17]. The survival curve models the amount of time that has passed before some percentile of the observed population has died. A more formal definition states the following; The survival curve can be created assuming various situations. It involves computing of probabilities of occurrence of event at a certain point of time and multiplying these successive probabilities by any earlier computed probabilities to get the final estimate. [18].

Other applications of the Weibull distribution are closer to the field of process mining. For instance, Velu uses the Weibull distribution in modeling the amount of

Table 1: Parameter Definitions for Weibull Formulas

Parameter	Definition
α	The scale parameter
β	The shape parameter
τ	The location parameter
λ	A combination scale and shape parameter $\lambda = \alpha^{-\beta}$
t	The value for time (x axis)

time a firm survives given a specified amount of business model innovation [19]. The message we can take away is that such innovations in a firm could also be considered as changes to the underlying processes that occur at such a firm. With this in mind, we move on to setup the foundation for the framework by which we dynamically set the input parameters for updating of $\hat{p}$.

3 Methods

Here we layout the procedures used to develop our framework, which we name *DynaTrainCDD* before moving on to Section 4 whereby we run the experiments and test its performance. In the following subsections, we introduce the different parameters used for optimization of incremental update accuracies and timings. The framework developed is a combination of the drift detection methods developed by Guzman et al. in [2] as well as the incremental updating process utilizing an SDL model developed by Pauwels et al. [1]. In tuning the drift detection parameters we look at the effect of changing the window size, tree size, and lambda. For altering the SDL models, we examine the effects of changing the update frequency and training size which is fed into each update.

Table 2: Development Datasets: Statistic Overview

Dataset	**Total**			**Train**			**Test**		
Parameter:	*#cases*	*#events*	*#activities*	*#cases*	*#events*	*#activities*	*#cases*	*#events*	*#activities*
Helpdesk	4,580	21,348	14	2,286	10,674	14	2,295	10,674	13
BPIC11	1,143	150,291	624	593	75,146	507	549	75,145	467
BPIC12	13,087	190,827	23	6,604	9,414	23	6,484	9,413	23
BPIC15.1	1,199	52,217	398	701	26,108	289	579	26,109	327
BPIC15.2	832	44,354	410	490	22,177	315	432	22,177	324
BPIC15.3	1,409	59,681	383	778	29,840	304	675	29,841	286
BPIC15.4	1,053	47,293	356	589	23,646	296	581	23,647	244
BPIC15.5	1,156	59,083	389	641	29,542	288	587	29,541	302
Duration	*#days*	*#weeks*	*#months*	*#days*	*#weeks*	*#months*	*#days*	*#weeks*	*#months*
Helpdesk	1,451	207	47	673	96	22	778	111	25
BPIC11	1,172	167	38	623	89	20	549	78	18
BPIC12	155	22	5	86	12	2	79	11	2
BPIC15.1	1,761	251	57	776	110	25	985	140	32
BPIC15.2	1,709	244	56	895	127	29	814	116	26
BPIC15.3	1,889	269	62	1,075	153	35	814	116	26
BPIC15.4	1,933	276	63	1,188	169	39	745	106	24
BPIC15.5	1,926	275	63	1,088	155	35	838	119	27

3.1 The Function of Drift Detection Parameters

Digging deeper into the function of the parameter inputs for *PrefixTreeCDD* we look to answer our first research question. Namely, it would be helpful to know how the inputs to *PrefixTreeCDD* affect its ability to detect drifts and overall running time. Table 4 shows the results of the changes to parameters used in the algorithm to detect concept drifts developed by [2]. Notably, locking all other parameters except for the window size, reveals that a larger window size leads to an increase in run time. Doing the same for tree size reveals that a larger tree size has the opposite effect, with lower run times experienced for runs with larger trees. Looking towards the lambda values, we again notice a decrease in run time for the otherwise same settings due to fewer historical events becoming relevant in determining location of drifts. To determine the effectiveness of the changes in identifying drifts, Table 3 shows how many drifts were identified in each variation. One stand out finding is that on all runs, *Helpdesk* experienced very few detected drifts with at most one. On the other hand the *BPIC11* dataset has the most cumulative identified drifts.

In analyzing the Table the same way as used for the timings, it appears that for window size, there is a peak in detection around window size 10 while increasing prefix tree sizes decreases the number of drifts detected. As for lambda, there there is not one stand out correlation that appears across the datasets with mixed results as the values of lambda are adjusted. In choosing the desired parameters for the drift detection algorithm, we first analysed the impact of changing the window size, tree size, and lambda. The results from this show us the varying drifts detected under each setting and how these settings affected the running times. In his thesis, Huete Guzmán, J. S. describes the default recommended parameter settings [2]. He notes that the chosen value of window size is 1000, number of prefix trees as 10, and a value of lambda of 0.25. These parameters were determined as those which maximised the observed F1 and Recall scores within Huete Guzmán, J. S.'s experiments, while minimizing the running times. We accept these default settings as the preferred settings of his *PrefixTreeCDD* algorithm and use them as a baseline for building our framework. While these settings help with efficiency, it is important to note that Huete Guzmán, J. S. also found that increasing the max Window prefix tree size led to fewer events before a detected drift: a shorter delay between when a drift occurs and when it is detected. However, by increasing the window prefix tree size, it led to lower accuracy in the form of false positive drifts being detected and longer run times [2].

3.2 Effect of Update Frequency & Training Size

Research questions 2 and 3 seek to provide insights into the adjustment of parameters of Pauwels' algorithm using incremental updates. Looking at the results shown in Table 6, the relationship between update frequency, train size, and total runtime is established. Specifically: increases in the update frequency (in number of events) correspond to decreases in overall runtime. Train size increases correspond to increases in runtime naturally as higher values cause more data to be fitted in each update to the model.

Table 3: Identified Number of Drifts by Parameter Combination

Window Size	PrefixTreeSize	Lambda	Helpdesk	BPIC11	BPIC12	BPIC15_1	BPIC15_2	BPIC15_3	BPIC15_4	BPIC15_5
8	500	0	1	16	11	7	3	5	3	6
8	500	0.15	0	11	4	4	6	5	5	6
8	500	0.25	1	20	15	9	6	5	4	2
8	800	0	0	7	8	2	3	1	3	3
8	800	0.15	0	5	4	2	3	5	3	3
8	800	0.25	0	12	12	2	4	4	3	3
8	1000	0	1	11	6	2	1	2	2	2
8	1000	0.15	0	6	1	1	1	2	2	2
8	1000	0.25	1	12	5	1	2	2	2	2
10	500	0	1	21	17	6	5	5	5	7
10	500	0.15	0	12	7	6	7	6	4	6
10	500	0.25	1	25	16	8	7	6	4	5
10	800	0	0	11	9	3	4	3	4	3
10	800	0.15	0	4	4	2	4	1	4	3
10	800	0.25	0	12	10	3	4	3	3	3
10	1000	0	0	17	6	1	2	0	2	6
10	1000	0.15	0	5	1	1	1	2	2	5
10	1000	0.25	0	17	5	1	0	4	2	4
12	500	0	1	21	14	7	8	6	5	6
12	500	0.15	0	10	4	8	9	7	5	8
12	500	0.25	1	22	14	9	9	10	6	6
12	800	0	0	12	9	1	2	4	3	4
12	800	0.15	0	7	3	2	4	4	4	4
12	800	0.25	0	13	14	2	5	4	5	4
12	1000	0	0	15	8	2	3	2	3	4
12	1000	0.15	0	4	4	2	1	2	3	3
12	1000	0.25	0	16	10	1	3	2	3	2

In relating this Table back to the work of [1], the experiments run in the framework introduced by [1] vary based on the dataset depending on the number of recorded events in each month of every dataset respectively (Table 6). They used batch sizes of one month and window sizes of both 1 and 5 (unit in months). One month of events ranges from between 1500 to around 2200 events for the subsets of the *BPIC15* dataset. The analysis presented in this paper allows for much more granular level of fine-tuning with a discretely defined event number as opposed to months. When using months as a unit, the number of considered events can change between each dataset. However, months does provide a nice reference point as identifiable business changes are likely implemented over time. If observing the data is too granular, there may be unnecessary processing occurring when drifts are not.

Table 4: Running Time of Drift Detection by Dataset (Total Time in seconds, **Best** in bold, Second Best underlined, *Worst* italicised)

Update Freq	Train Size	Helpdesk	BPIC11	BPIC12	BPIC15_1	BPIC15_2	BPIC15_3	BPIC15_4	BPIC15_5
200	200	110.001	3362.824	985.806	578.738	500.505	616.747	439.771	641.358
200	500	*125.212*	7129.185	*1135.369*	1048.712	882.474	1070.823	758.654	1123.648
500	200	46.083	1345.023	396.146	234.850	197.522	254.000	186.809	262.323
500	500	51.182	2875.557	452.463	425.418	358.726	443.971	309.589	453.777
1000	200	22.186	692.263	199.579	117.661	101.422	126.166	102.037	141.091
1000	500	25.388	1467.718	233.292	215.698	184.171	218.945	165.997	242.348
1000	1000	30.644	2703.788	276.130	375.692	318.639	379.232	271.027	400.889
1000	5000	57.890	*12243.376*	661.383	*1535.580*	*1269.520*	*1537.219*	*1047.840*	*1599.840*
5000	200	6.362	158.929	44.837	26.551	21.831	29.782	32.131	41.605
5000	500	6.515	325.700	51.410	49.506	39.505	51.752	41.675	66.440
5000	1000	8.061	598.865	60.588	83.787	67.599	85.052	65.104	108.757
5000	5000	17.910	2768.874	147.393	365.866	305.194	361.286	273.975	447.839
7500	200	**3.829**	164.780	31.009	17.801	13.515	21.557	18.464	32.585
7500	500	4.358	327.110	37.229	31.980	24.385	37.521	30.690	50.746
7500	1000	5.397	590.841	41.886	57.203	43.056	63.256	44.240	81.743
7500	5000	12.349	2546.206	98.579	254.665	192.335	266.446	190.796	348.508
10000	200	3.833	**138.316**	**29.001**	**12.867**	**13.447**	**18.349**	**16.991**	**29.692**
10000	500	4.305	271.732	33.085	24.430	24.592	30.846	22.805	44.954
10000	1000	5.443	487.628	37.457	43.190	42.818	51.214	35.806	72.113
10000	5000	11.882	2060.252	89.896	195.796	187.550	220.187	151.610	295.527

With the information presented in Table 6 there are two obvious optimizations that could be implemented with respect to the evaluation metrics. One of these optimizations is seeking to maximize accuracy at the price of the runtime. The other optimization is to approach with the goal of minimizing runtime while sacrificing the least amount of accuracy as a result. This framework seeks to do the former by maximizing the resulting accuracy with slightly higher running times expected. The detected drifts including their index and severity can be utilised alongside the current parameters of update size and training size to estimate the most desirable update size and training size parameters for the next batch.

When a detected drift is caught it triggers a drift recovery curve run. This drift recovery curve in turn seeks to correct for the drift and return accuracy to expected levels. We seek to achieve a higher accuracy than the accuracy observed by Pauwels' work [1]. Doing so implies that for at least some number of updates performed under the framework, we need to have a window size lower than the window size used in each batch update than each dataset had in [1]'s methods. In all of Pauwels' experiments, the update window was the same as training size and the dataset with the lowest values for each of these parameters was Helpdesk with roughly 1000 events. To determine the optimal settings for the new algorithm, we run a series of static window size tests across the datasets to observe which parameters return the best accuracies and running times. The results of these implementations are shown in Tables 5 and 6 respectively. The associated figures corresponding to these tables are located in the Appendices in Figures 11, 12, 13, and 14.

Table 5: Static Updating Varied Parameters Average Accuracy (**Best** in bold, Second Best underlined, *Worst* italicised)

Update Freq	Train Size	Helpdesk	BPIC11	BPIC12	BPIC15_1	BPIC15_2	BPIC15_3	BPIC15_4	BPIC15_5
200	200	0.842	0.555	0.770	0.774	0.769	0.776	0.805	0.778
200	500	0.839	0.556	0.767	**0.777**	**0.773**	**0.787**	**0.814**	**0.788**
500	200	0.841	0.537	0.759	0.734	0.733	0.740	0.759	0.723
500	500	**0.842**	0.564	0.770	0.770	0.767	0.781	0.797	0.777
1000	200	0.833	0.513	0.744	0.690	0.674	0.707	0.688	0.661
1000	500	0.837	0.551	0.762	0.741	0.723	0.756	0.763	0.728
1000	1000	0.838	0.573	0.774	0.762	0.746	0.779	0.791	0.761
1000	5000	0.838	**0.574**	0.770	0.762	0.748	0.784	0.797	0.765
5000	200	0.808	0.445	0.710	0.520	0.549	0.531	0.481	0.502
5000	500	0.816	0.506	0.742	0.567	0.603	0.612	0.591	0.602
5000	1000	0.819	0.535	0.757	0.606	0.649	0.646	0.651	0.638
5000	5000	0.818	0.572	**0.778**	0.641	0.657	0.697	0.709	0.676
7500	200	0.806	0.127	0.631	0.517	0.451	0.495	0.426	0.449
7500	500	0.810	0.160	0.697	0.561	0.503	0.587	0.551	0.526
7500	1000	0.800	0.191	0.718	0.597	0.570	0.618	0.606	0.572
7500	5000	0.804	0.220	0.753	0.625	0.600	0.669	0.680	0.621
10000	200	*0.773*	*0.121*	*0.597*	*0.458*	*0.4270*	*0.470*	*0.370*	*0.403*
10000	500	0.775	0.157	0.685	0.488	0.486	0.574	0.500	0.504
10000	1000	0.776	0.186	0.713	0.5380	0.510	0.596	0.568	0.534
10000	5000	0.774	0.220	0.748	0.589	0.547	0.650	0.643	0.579

From the perspective of maximizing accuracy with no regard for required computational resources: altering the update frequency to a lower, and thus, more frequent value and increasing the training size could yield impressive results as seen in Figures 11, 12, 13, and 14 (Located in the Appendices). However, it is important to note that in the runs these were tested on, experimental settings never tested values above 500 for train size when the frequency was lower than 1000 due to the extreme computational times expected to achieve an output result. Including higher training sizes may lead to even higher results, however it is unknown at this time what that outcome would be or whether the trade off for significantly more com-

Table 6: Static Update Varied Parameters: Total Train Time (secs, **Best** in bold, Second Best underlined, *Worst* italicised)

Update Freq	Train Size	Helpdesk	BPIC11	BPIC12	BPIC15_1	BPIC15_2	BPIC15_3	BPIC15_4	BPIC15_5
200	200	108.211	3375.452	978.335	578.738	500.505	616.747	439.771	641.358
200	500	*122.992*	7190.742	*1130.794*	1048.712	882.474	1070.823	758.654	1123.648
500	200	43.643	1374.482	399.310	234.850	197.522	254.000	186.809	262.323
500	500	49.774	2870.678	453.163	425.418	358.726	443.971	309.589	453.777
1000	200	21.297	682.182	199.922	117.661	101.422	126.166	102.037	141.091
1000	500	25.228	1469.858	231.592	215.698	184.171	218.945	165.997	242.348
1000	1000	29.784	2703.074	283.938	375.692	318.639	379.232	271.027	400.889
1000	5000	60.082	*12367.273*	681.508	*1535.580*	*1269.520*	*1537.219*	*1047.840*	*1599.840*
5000	200	6.104	189.536	44.569	26.551	21.831	29.782	32.131	41.605
5000	500	7.080	387.883	52.061	49.506	39.505	51.752	41.675	66.440
5000	1000	7.729	706.860	63.362	83.787	67.599	85.052	65.104	108.757
5000	5000	17.796	3056.633	151.810	365.866	305.194	361.286	273.975	447.839
7500	200	**3.829**	164.780	31.009	17.801	13.515	21.557	18.464	32.585
7500	500	4.358	327.110	37.229	31.980	24.385	37.521	30.690	50.746
7500	1000	5.397	590.841	41.886	57.203	43.056	63.256	44.240	81.743
7500	5000	12.349	2546.206	98.579	254.665	192.335	266.446	190.796	348.508
10000	200	3.833	**138.316**	**29.001**	**12.867**	**13.447**	**18.349**	**16.991**	**29.692**
10000	500	4.305	271.732	33.085	24.430	24.592	30.846	22.805	44.954
10000	1000	5.443	487.628	37.457	43.190	42.818	51.214	35.806	72.113
10000	5000	11.882	2060.252	89.896	195.796	187.550	220.187	151.610	295.527

putational resources would be worth the potentially higher accuracy. Additionally important is that while the results presented in Figures 11, 12, 13, and 14 (See Appendices) reveal a general best performance from an update frequency of 200 and training size of 500, the *BPIC11* dataset which has significantly more drifts than any of the other datasets does not experience the same performance with parameters of 500 and 500 respectively beating out the others. This could be due to the higher number of possible activities in this dataset, leading Pauwels' SDL framework to struggle to classify the predicted next activity.

3.3 Developing the Dynamic Framework

With the insights provided in Sections 3.1 and 3.2, we now progress to develop the Dynamic Framework which we intend to use to answer research questions 4 and 5 introduced in Section 1.3. In making a determination as to how severe a drift is, we propose the development of two heuristics; one heuristic each to set the dynamic value for the update frequency as well as the training size, both introduced above in Section 3.2.

Initial research into the creation of these heuristics led us to explore the connections we could draw to distributions available from the field of survival and reliability analysis. At surface level, survival and reliability may not appear to overlap significantly with process mining, however, we present the view that the the metadata regarding drifts can be utilised in a way that determines the "reliability of the information" based on the number of events since the last drift. When a drift has not been observed for some time, intuition is that less frequent updates and smaller update sizes are needed to maintain the performance of $\hat{p}$. With less frequent drifts, we can place more trust in the trained model and hold the expectation that we can increase the timing between updates (update frequency) and decrease the amount of data used in each of those updates (train size).

Taking our learnings from the exploration of update frequency and training size from Section 3.2, we introduce a Weibull function used to set each of the stated parameters dynamically. This Weibull function can be considered to track the reliability of the model as it experiences drift over time. The intuition in developing this model is as follows: the more time since the last drift, the more lenient the values of update frequency and train size can be. Parallels can be made to the adjustable window approach whereby the window expands until its maximum size and then continues sliding until another drift is found which we covered in the Preliminaries in Section 2.7 [1, 2, 3]. In this case, however, the Weibull function is utilised to help dynamically loosen the "reigns" on our algorithm parameters as increasing amounts of data are trained and the model is considered to be more trustworthy due to a lack of drifts.

This framework itself will not be allowed to remain unchecked, however. Within this framework safeguards are built to make incremental updates to the model with the parameters u_{min}, t_{min},u_{max}, and t_{max}. Through the implementation of these ranges, the minimum and maximum number of events between updates as well as the minimum and maximum number of events that must be included in the training of each update to $\hat{p}$ are explicitly set. More information on these parameters and other calculated parameters for the algorithm can be seen in Table 7.

3.4 Drift Recovery Formula & Setting of Algorithm Parameters

Moving back to the concepts introduced by the Weibull distribution in Section 2.10, we present a drift recovery formula through which values can be chosen for both the update frequency and training size. Using the parameters shown in Table 7, two arrays of integers are created seen in Equations 6 & 7. Each array contains all integers between and including minimal and maximal values respectively for both update frequency and train size. Important to note is that the training size is in reversed order from maximal to minimal. This reversal is significant in that the parameters in the leading part of both arrays represent values for when drift is present and models are considered less trustworthy while the trailing half contains values used when the process model is considered more trustworthy; at this point the model no longer requires such frequent and large updates.

$$updateArray =< u_{min}, u_{min}+1, u_{min}+2, ..., u_{max}-1, u_{max} > \tag{6}$$

Table 7: Algorithm Parameter Definitions

Parameter	Definition
u_{min}	The minimum number of events as an integer between model updates
u_{max}	The maximal number of events as an integer between model updates
t_{min}	The minimal number of events as an integer to include in training data used in model updates
t_{max}	The maximal number of events as an integer to include in training data used in model updates
d	The number of events we set to hold the retraining values above the threshold parameter
k	The threshold parameter whereby we define what value we would like the algorithm to remain above before relaxing the drift recovery formula to our u_{max} and t_{min} respectively.

$$trainArray = < t_{max}, t_{max}+1, t_{max}+2, ..., t_{min}-1, t_{min} > \tag{7}$$

We now define the *dynammic drift recovery* formula using the second Weibull function found in Formula 5. In creating this formula, we want an equation which begins at a value of 1 indicating the need for maximal correction after a drift, then trailing off to our minimal value of 0 indicating the ability to relax our parameters. The output is a float in the range of $[0,1]$ which is then used to determine the parameter to use by multiplying this value by the length of the *updateArray* and *trainArray* from Equations 6 & 7. By multiplying these two, we identify the index of the parameter chosen as input in updating $\hat{p}$ at time t, with t being the number of events after the most recent drift is identified. As we want all of our values to start from time $t = 0$ after a drift occurs, there is no need to use the location parameter τ. We can set τ to zero and remove it from Formula 5. We are now left with the equation seen in Formula 8 whereby the only remaining parameter we need is λ as t is set automatically by the number of events which have passed since the last drift divided by u_{max} such that the value for t is never greater than 1.

$$Recov_{drift} = 1 - \exp(\lambda t) \tag{8}$$

In choosing a value for λ we consider first what we know regarding [2]'s work whereby there is an expected delay in reporting of an identified concept drift. [2]'s research indicates that we should expect around a 3500 event delay with the default parameters for the *PrefixTreeCDD* algorithm. With this in mind, we know that once a drift has been identified, it likely holds some level of history which can be used to update $\hat{p}$ the process model.

To create the final formula we will use as the drift recovery curve in our framework we must find the value of λ. To do so, we take the chosen value for d (which must be lower than our u_{max}) parameter and formulate the equation by filling in t with our $\frac{d}{u_{max}}$. This essentially sets d as a number of events to maintain the integrity of retraining of the model above the parameter k. In this case we have chosen k as a value of 80% or 0.8 as we feel that after holding the drift recovery curve output at 80% until d events have passed, this should yield favorable results with respect to accuracy.

To set the value of our λ we layout the equation filled in as follows:

$$1 - \exp \lambda\left(\frac{d}{u_{max}} - 1\right) = k \tag{9}$$

Then solving for λ we can calculate lambda such that we have an equation that has outputs which start at 1 and end at 0 going through the point $\frac{d}{u_{max}}$ at the value k.

$$\ln(1-k) = \lambda\left(\frac{d}{u_{max}} - 1\right) \tag{10}$$

$$\lambda = \frac{\ln(1-k)}{\left(\frac{d}{u_{max}} - 1\right)} \tag{11}$$

Algorithm 1 Drift Recovery Calculation

```
1: $drifts \leftarrow$ list of drift objects containing drift location, severity
2: for every $event$ in dataset do
3:     $lastDrift \leftarrow 0$
4:     if CDDAlgo detects $drift$ at $event$ then
5:         $drifts \leftarrow drifts + drift$
6:         $lastDrift \leftarrow 0$
7:         if length of drifts < 5 then
8:             $DriftRecCurveInput \leftarrow 0$
9:         else
10:            $DriftRecCurveInput \leftarrow quantile(drift, driftSeverities)$
11:    if $lastDrift > u_{max}$ then
12:        $DriftRecCurveInput \leftarrow 0.75$
```

Now we have our final equation which we have determined by making measured decisions which will serve as the foundation for dynamically changing u_{freq} and t_{size}. When integrating this together with the values we have for u_{freq} and t_{size} as well as our arrays $updateArray$ and $trainArray$ we use the input index parameters $\in [0,1]$ to retrieve values form these arrays. To calculate the input to send to the *drift recovery function*, we utilize the tree distance metric from *PrefixTreeCDD* [2]. In the case that we have identified fewer than 5 drifts in a given dataset, the input to the *drift recovery function* is set to 1 and considered to be the most severe. After 5 drifts have been detected, the sixth and all subsequent drifts are compared against the existing stored library of drifts and their tree distance metrics and a percentile is calculated of where it falls within the distribution of those drifts to serve as the input to the *drift recovery function*. In the case that a drift is not detected yet the value for u_{max} events since the last training event has been reached, we run an update anyway with the input to the drift detection formula set at a value of 0.5 which can be viewed as half severity. The algorithm for this can be seen in Algorithm 1.

Ultimately the values chosen for u_{min}, u_{max} and t_{min}, t_{max} were as follows:

Table 8: Parameter Choices

u_{min}	u_{max}	t_{min}	t_{max}
200	500	500	750

The above listed parameters were chosen due to accuracies observed across datasets in Table 5. Maintaining update and training sizes between 200 - 500 and 500 - 750 respectively appeared to provide the best results while not leading to excessive running times which were presented in Table 6. As we have 500 for the value of u_{max}, the value of 490 is chosen for the parameter d as holding this value as close to 500 as possible will maximize accuracy based on the formulation of the $driftrecovery$ formula.

In Appendix B, Figures 15 & 16 show the idealised and actual plots for the *Drift Recovery Formula*. We find that the selected parameters are the ones which are expected to perform the best on the data given what is known about the datasets from the Tables 5 & 6 as well as their associated plots in Appendix A.

The output of the Drift recovery function is a value $\in [0,1]$ which is then multiply by the number of elements in the arrays $updateArray$ and $trainArray$ before rounding to a whole integer to determine the index in the respective array of the u_{freq} and t_{size} values for the current batch. These batches are then updated as would normally be done during Pauwels' *Incremental Update* methods [1].

3.5 Summary Of Methods

Figure 5 shows the progression through the overall methods and how the presented research ties together in this analysis. Next, the Experimental Evaluation in Section 4 details the conditions formulated to run the experiments and lists the observed results.

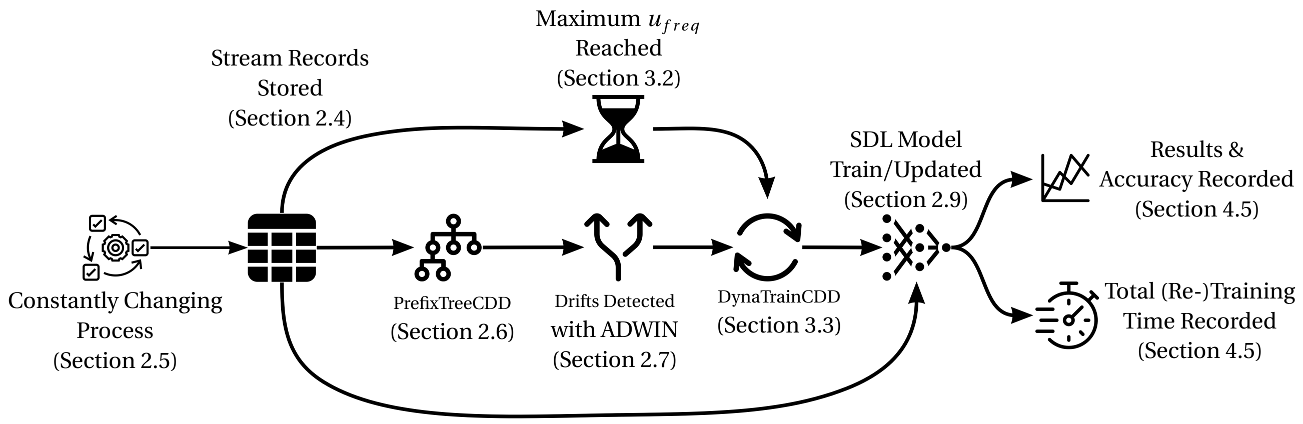

Maximum u_{freq} Reached (Section 3.2)
Stream Records Stored (Section 2.4)
SDL Model Train/Updated (Section 2.9)
Results & Accuracy Recorded (Section 4.5)
Constantly Changing Process (Section 2.5)
PrefixTreeCDD (Section 2.6)
Drifts Detected with ADWIN (Section 2.7)
DynaTrainCDD (Section 3.3)
Total (Re-)Training Time Recorded (Section 4.5)

Fig. 5: Framework Structure and Workflow

4 Experimental Evaluation

We now layout the experimental conditions and results in the following subsections. First we review the datasets used in our experiments in Subsection 4.1. Then the preparation and processing used on these datasets is laid out in Subsection 4.2. In the Experimental settings in Section 4.3 we discuss the conditions under which we run the experiments. Evaluation metrics are reviewed in Section 4.4 whereby we discuss the means of measuring the performance of the developed methods. Our results and discussion surrounding overall performance are presented in Section 4.5 where Research Questions 4 and 5 from Section 1.3 are explored. We then conclude the findings of our experiments in Section 4.6.

4.1 Datasets

The data used in this analysis are from sets commonly used in process mining research as a benchmark. The *BPIC15* dataset is one made of loan applications with 5 separate files, each from different Municipalities within The Netherlands. The information included within the datasets is comprised of event logs with a timestamp, event name identifiers (also called activities), trace identifiers (also referred to as a cases), as well as other fields relating to what administrative departments have processed the events and whether or not the event has been completed among other recorded datapoints. Additionally utilised in the experiments are the datasets *BPIC11* [20] which includes process event information relating to a Dutch academic hospital, *BPIC12* [21] recording another loan application process, and *Helpdesk* [22] which details the ticketing system for an Italian software company. For validation, we include two other datasets. Namely: *BPIC17* [23] and *BPIC15* (merged) [24] are included for the purpose of validating our developed results. For the merged *BPIC15* all datasets 1-5 have been appended to one another with the dates shifted such that the start date of the next set occurs on the end date of the previous one. In this way we are able to create one coherent dataset with the existing drifts included in each subset. Additionally, the changes experienced between each subset are considered to be *sudden drifts* which we have covered in Section 2.5. This inclusion allows us to validate the performance of *PrefixTreeCDD* [2] when faced with the challenge of identifying the different types of drifts.

4.2 Processing

To create a functional experimental environment, all of the .xes files initially obtained from [20, 21, 22, 24] were converted into CSV files. Much of the preprocessing noted in [1] is replicated except for alterations which allowed for the incremental updating processes to be executed from a CSV file such that the *PrefixTreeCDD* framework from [2] could be utilised in tandem. New events are read into the pseudo-streaming environment line by line from the csv. Additionally, after parsing from csv, the event stream was sorted based on event timestamp such that they are viewed by the algorithm in the order that they occur, and not only in chronological order aggregated by trace as output from the .xes converter which was utilised for the task [25].

Table 9: Validation Datasets Statistic Overview

Dataset	Total			Train			Test		
Unique records:	*#cases*	*#events*	*#activities*	*#cases*	*#events*	*#activities*	*#cases*	*#events*	*#activities*
	42,995	1,202,267	66	746	30,057	55	30,764	1,172,210	66
Duration:	*#days*	*#weeks*	*#months*	*#days*	*#weeks*	*#months*	*#days*	*#weeks*	*#months*
	397	56.71	13	19	2.71	0.63	379	54.14	12.63

One notable finding following the methods laid out in [1] involves changing all of the unique names of events or activities in the data into distinct integer values. While it is generally advisable to expand these columns out into one-hot encoded columns using binary indications as to whether or not a specific record is identified as one type of event or activity, doing so caused significant problems when trying to integrate the methods of one hot encoding into the existing framework, thus this preprocessing was left as it was presented in [1]. With one-hot encoding, inputs to the model would have been a sparse matrix with each event having 0 values for events, activities, and roles that they are not, while having values of 1 for any event that they are considered to be. Given the high numbers of possible activities as seen in Table 2 & 9, attempting to utilise one-hot encoding would have made the input space significantly larger and required an entire reworking of the entire SDL framework which was developed by Pauwels [1].

One final change made from the methods used in Pauwels' work is that his *Incremental Update* and *Last Drift* methods were reformulated for use under our developed *DynaTrainCDD* framework. Since Pauwels' work operated based on a monthly aggregated time to determine batch size, it was not compatible with our batch sizes determined by a unique number of events. To account for this, we calculated the batch sizes that Pauwels used by averaging the number of events per month over each individual dataset by dividing the total events in each dataset by the number of months which we have presented in Tables 2 & 9. The resulting average serves as the window size $w = 1$ which is comparable to the methods adopted by Pauwels [1]. The calculated averages are displayed in Table 10.

Table 10: Average Monthly Events by Dataset

	Helpdesk	**BPIC11**	**BPIC12**	**BPIC15_1**	**BPIC15_2**	**BPIC15_3**	**BPIC15_4**	**BPIC15_5**	**BPIC15_Merged**	**BPIC17**
Avg Monthly Events	1000	7150	87400	2000	1500	2000	1500	2200	1000	92500

4.3 Experimental Settings

Hardware Specifications The details of the hardware used to obtain the results detailed in this report are listed below

Operating System: Mac OS 12.4
Model A1989; MacBookPro15,2; MacBook Pro (13-inch, 2019, 4 ports)
CPU 2.4 GHz Quad-Core Intel Core i5
Memory 8 GB 2133 MHz LPDDR3
Graphics Intel Iris Plus Graphics 655 1536 MB

The methods to be compared are as follows:

1. *DynaTrainCDD*
2. *IncrementalUpdate*$(w = 1)$ [1]
3. *IncrementalUpdate*$(w = LastDrift)$ [1]
4. No Retrain

The developed *DynaTrainCDD* method is to be compared against Pauwels' methods of *IncrementalUpdate*$(w = 1)$, *IncrementalUpdate*$(w = LastDrift)$, as well as against no retraining after the initial training of a model. In the *IncrementalUpdate* methods, the value of w determines the window size of both the update frequency and the training size. For $w = 1$ this means a value of 1 month of aggregated data while for $w = LastDrift$ this is all data between the current drift and the last observed drift [1]. The only difference with $w = LastDrift$ is that the update frequency remains every month, however the training size can extend all the way to the value of the last drift (or beginning of time if no drift has been observed).

All datasets except for *BPIC17* are trained into an SDL model initially (which was introduced in Section 2.9) with 0.5 of the data as training data and the remainder of the data is left to the respective methods to evaluate their performance. For the validation set of *BPIC15* (merged), initial training size is set to be equal roughly the same number of events as an individual *BPIC15* dataset with a training size of 0.1 of the total merged data. This way we can see how the method performs when experiencing sudden drifts with new, never before seen data.

4.4 Evaluation Metrics

To evaluate the efficacy of the proposed solution, a combination of running time and accuracy are considered. Each method is run on each dataset to determine overall performance in each individual application. Specifically, to calculate the accuracy, the output of predicted next activity from the SDL model will be compared against the actual next activity. Then these boolean values of True or False correct predictions are run through a window averaging function to determine the accuracy at some given point within the output set such that they can be plotted over time. A total average is also reported in tabular format. The running time is determined by summing the total cumulative retraining duration for the SDL model. This recorded time is non-inclusive of the intial train or concept drift detection formula on each of the datasets as the performance of the concept drift detection and initial train are considered out of scope in this analysis. We purely seek to evaluate performance on retraining and updating of existing models with information we receive from the *PrefixTreeCDD* methods [2].

4.5 Results and Discussion

We now present the results of the *DynaTrainCDD* framework we have proposed in the Methods found in Section 3 compared with the above listed methods from Pauwels' *Incremental Updates* [1]. We will discuss the results first from the viewpoint of accuracy before moving on to examine how the developed methods affect the running time of the models. Lastly, we present the results using an external, unseen dataset (*BPIC17*) for the purpose of validating our methods before comparing to the aggregated *BPIC15* (merged) dataset which is also used for validation.

Development Data Results Looking at Table 11, we present the results of *DynaTrainCDD* alongside the methods used by [1]. Our method is denoted by *DynaTrainCDD*. We see that for every dataset except for *BPIC11*, *BPIC12*, and *Helpdesk*, *DynaTrainCDD* was able to outperform the others. Particularly with the *BPIC15* data, we see significant improvements of around 1-2% more accuracy on average. For *BPIC11*, *BPIC12*, and *Helpdesk* we note some interesting findings. On *Helpdesk*, *DynaTrainCDD* only performed 0.0049 worse than the *IncrementalUpdate* ($w =$

1) method. On *BPIC11*, *DynaTrainCDD* performed 0.173 worse than No Retrain whatsoever. Overall all methods on *BPIC11* suffered with respect to accuracy. This can be attributed due to the significantly higher number of unique activities which each next event can take on (seen in Table 2). With more activities, the expected performance decreases as the baseline of the random selection in these cases becomes $\frac{1}{TotalNumberofActivities}$. As such the probability of correctly randomly selecting one activity is much lower than in datasets with fewer activities. On *BPIC12*, *DynaTrainCDD* performs the worst, however all methods perform worse than No Retrain. This result suggests that this dataset may experience some *recurring drifts* in a way such that the baseline trained model is able to effectively identify the possible outcomes (see Section 2.5 on drift types).

Table 11: Average Accuracy of Developed Experimental Methods (**Best** in bold, Second Best underlined, *Worst* italicised)

Method	*Helpdesk*	*BPIC11*	*BPIC12*	*BPIC15_1*	*BPIC15_2*	*BPIC15_3*	*BPIC15_4*	*BPIC15_5*
DynaTrainCDD	0.8377	0.2473	*0.7473*	**0.7615**	**0.7538**	**0.7816**	**0.7888**	**0.7671**
IncrementalUpdate $(w = 1)$	**0.8426**	0.2282	0.7502	0.7447	0.7445	0.7613	0.7773	0.7406
IncrementalUpdate $(w = LastDrift)$ [1]	0.8323	*0.2280*	0.7474	0.6743	0.6572	0.7127	0.7161	0.6977
No Retrain	*0.7642*	**0.2646**	**0.7930**	*0.2425*	*0.2843*	*0.2981*	*0.2592*	*0.2428*

Figures 6 & 7 show plots of the accuracy over each development dataset. We observe that as expected based on Table 11 that for all datasets except for *Helpdesk*, *BPIC11*, and *BPIC12*, *DynaTrainCDD* performs better overall with smaller and more frequent, dynamically set updates. We bring to attention two notable points. The first point being that on the *BPIC15* data, we observe faster recovery from drifts which occurred in the training. This is likely attributable to the more frequent updates able to accommodate for faster recovery times than those used by [1].

When looking at where our methods did not perform the best, we turn to *BPIC12* for some very interesting findings. In *BPIC12* the best performance is observed when the model is not retrained at all after initial training on half of the overall data. The difference is very minimal (see Table 11), however still noticeable. This is very quickly followed by Pauwels' methods and then *DynaTrainCDD* which performs the worst in comparison (although still generally very well). We believe the reasoning behind the results observed with *BPIC12* are caused by the reasons listed above, mainly relating to the recurring type of drifts that we expect are present in this dataset. This is what we believe causes the variation between no retraining and *DynaTrainCDD* presented in this report.

In an online setting we are unsure when or if drifts are occurring as the event stream is constantly incoming. Running a model with the assumption that drifts will not occur, assuming that a business process will never change, is unrealistic. While never retraining would be ideal in terms of running time, assuming we need to spend no time in training updates leaves a model highly vulnerable to unexpected drifts. Any drift encountered by a model which is not updated (dynamically or not) could cause the accuracy to unexpectedly plummet. Thus utilising the method whereby no retraining occurs could be risky in the case a future drift is encountered, leading to a lower expected accuracy.

Table 12: Total Training Time (seconds, **Best** in bold, <u>Second Best</u> underlined, *Worst* italicised)

Method	*Helpdesk*	*BPIC11*	*BPIC12*	*BPIC15_1*	*BPIC15_2*	*BPIC15_3*	*BPIC15_4*	*BPIC15_5*
DynaTrainCDD	<u>44.439</u>	**2830.526**	*444.673*	<u>407.256</u>	<u>348.296</u>	<u>426.515</u>	<u>299.566</u>	<u>438.440</u>
IncrementalUpdate ($w = 1$) [1]	**29.784**	*3680.473*	<u>167.456</u>	**360.305**	**293.987**	**353.182**	**262.477**	**397.981**
Last Drift [1]	*120.137*	<u>3535.270</u>	**120.895**	*2430.327*	*2122.920*	*1567.416*	*1376.079*	*1474.330*
No Retrain [1]	-	-	-	-	-	-	-	-

Next, we examine the running times. In Table 12. We see that while the *Incremental Update* methods run faster on the *Helpdesk*, *BPIC12* as well as all *BPIC15* datasets, *DynaTrainCDD* outperforms other methods on *BPIC11*. Specifically, we note only about 1.5 times running times of the fastest training times on the *BPIC15* dataset. Not only did accuracy increase on the *BPIC15* dataset, however we also observe minimal increases in running times.

Validation Data Results Next we present the results observed on an external validation set. This dataset: *BPIC17* is formatted in the same way as those used to develop our methods, with events, traces, timestamps, and resource attributes. Similarly to some of the other BPIC datasets, it concerns loan applications again. The makeup of the data is detailed in Table 9. We now discuss our observations when running our methods on the validation dataset. Another validation dataset we have chosen to include is *BPIC15* (merged) which is a dataset comprised of all the other BPIC15 subsets appended into one long dataset with sudden drifts occurring between each of the subsets.

When looking at Figure 8 and Table 13, we notice that *No Retrain* performs the best. The second best performance is *DynaTrainCDD* followed by Pauwels' *IncrementalDrift* methods respecitvely [1]. This is also reflected in Table 13 where all the results for accuracy and running time are displayed for our two validation sets. However, this doesn't tell the full story. As noted earlier when discussing the results for *BPIC12*, we suggested that any unexpected drift would cause the baseline model to become unreliable and lose accuracy. Around the 450k event mark, we notice exactly that whereby the accuracy starts falling from 0.85 to 0.7 on the *No Retrain* methods and *IncrementalUpdate* ($w = 1$).

Interestingly, both *DynaTrainCDD* and the *IncrementalUpdate*($w = LastDrift$) remain above 0.8. This leads to a large difference in accuracy that is extremely undesirable when highest accuracy is preferred. We believe that overall, the performance shown in Figure 8 reflects very few drifts until around 450k events. At the

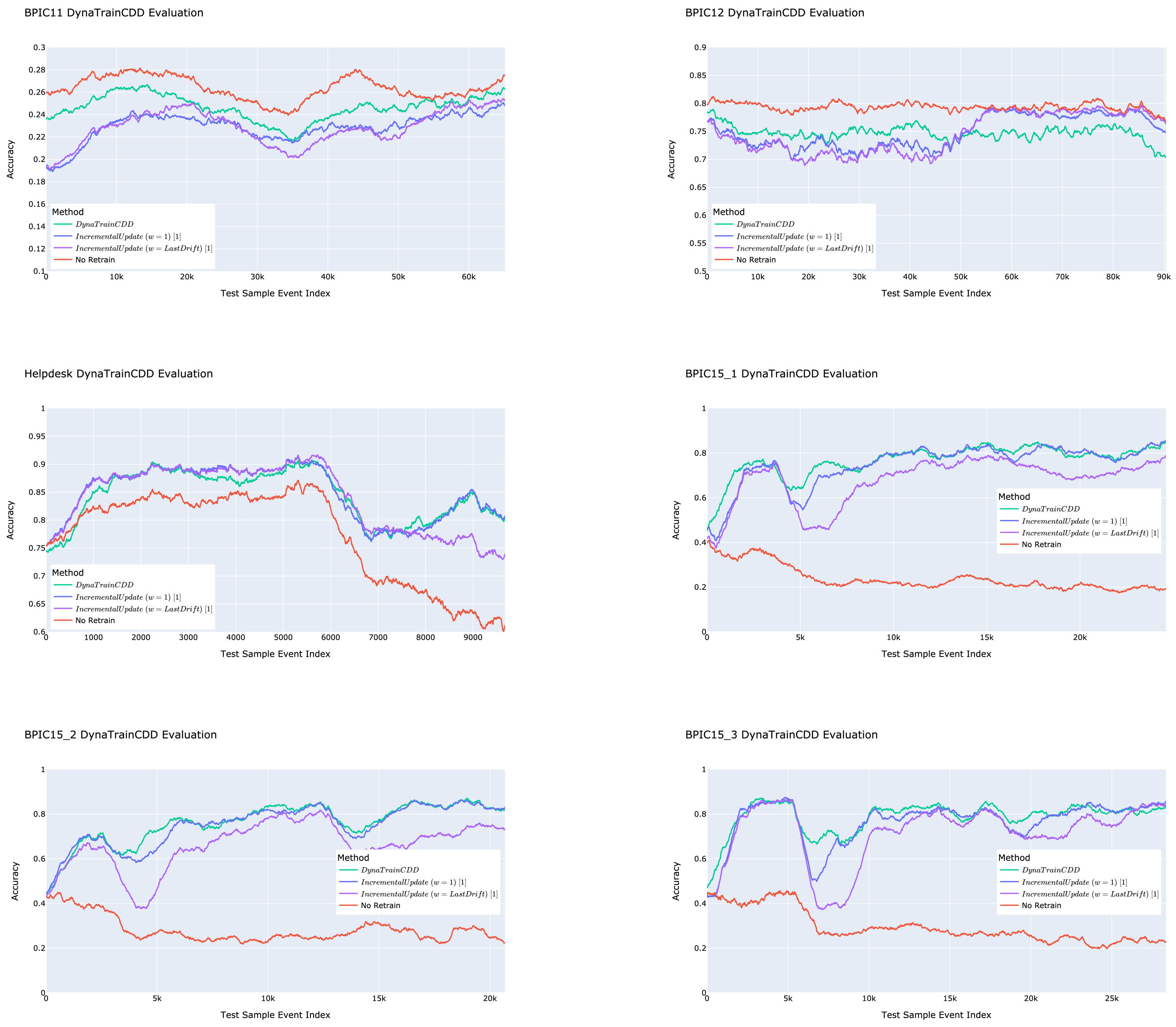
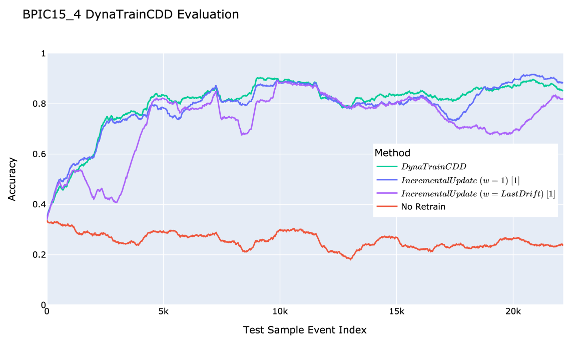


Fig. 6: Results on Development Datasets (1 of 2)

point when a drift does occur, both *DynaTrainCDD* and static incremental updates show their strength in maintaining accuracy.

In looking towards the *BPIC15* (Merged) Dataset we see that *DynaTrainCDD* performs the best with respect to accuracy around 4% higher than its nearest com-



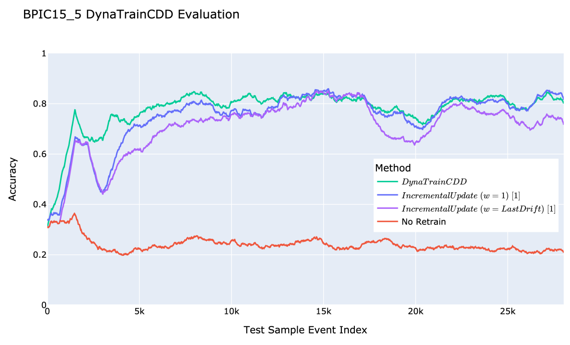


Fig. 7: Results on Development Datasets (2 of 2)

petitor (being *IncrementalUpdate* $(w = 1)$ with 0.191). Overall the accuracies experienced across the *BPIC15* (merged) dataset are significantly lower than any other accuracies we have seen in our results. We attribute this to the extremely high number of activities to predict, similar to the results seen on the *BPIC11* dataset in Table 11. The number of unique activities is the sum of all unique activities present in the *BPIC15* datasets combined. As previously mentioned, this makes the probability that the SDL model is able to determine the correct next activity significantly lower. Since the training set for *BPIC15* only included data from *BPIC15_1*, the results we see in Figure 9 make sense as each time a new dataset with its respective drift point is encountered in the testing data, accuracy drops while the model struggles to keep up until it has been retrained a few times.

While accuracy provides one part of the overall picture, we must also consider running time. Looking back to Table 13 we note that *DynaTrainCDD* performs better than *IncrementalUpdate*$(w = LastDrift)$ *BPIC15* (merged) but not on *BPIC17*. The higher running time of *DynaTrainCDD* is justified in accuracy for *BPIC17* however, as it performs better overall for everything except for the *No Retrain* method. As we previously mentioned, it is not reasonable to leave a dataset without updates as the chance of a drift occurring leaves the model open to large risks for accuracy if a drift does occur. Thus we still believe that *DynaTrainCDD* presents the best solution in this case. In *BPIC15* (merged), *DynaTrainCDD* displays significantly better accuracy than any other method, thus we also conclude that although it has a higher running time than *IncrementalUpdate*$(w = 1)$, it still remains the best choice overall.

Lastly, in Figure 10 and associated Table 14 we examine the results of the *BPIC15* (merged) dataset and answer research question 5 in looking to compare the developed methods to a baseline. We note that *DynaTrainCDD* is the best performer when observing the overall results in the period of 20 thousand events following each sudden drift. Surprisingly, not retraining at all is the second best performing method. The results in Table 14 further show the strength of *DynaTrainCDD* follow-

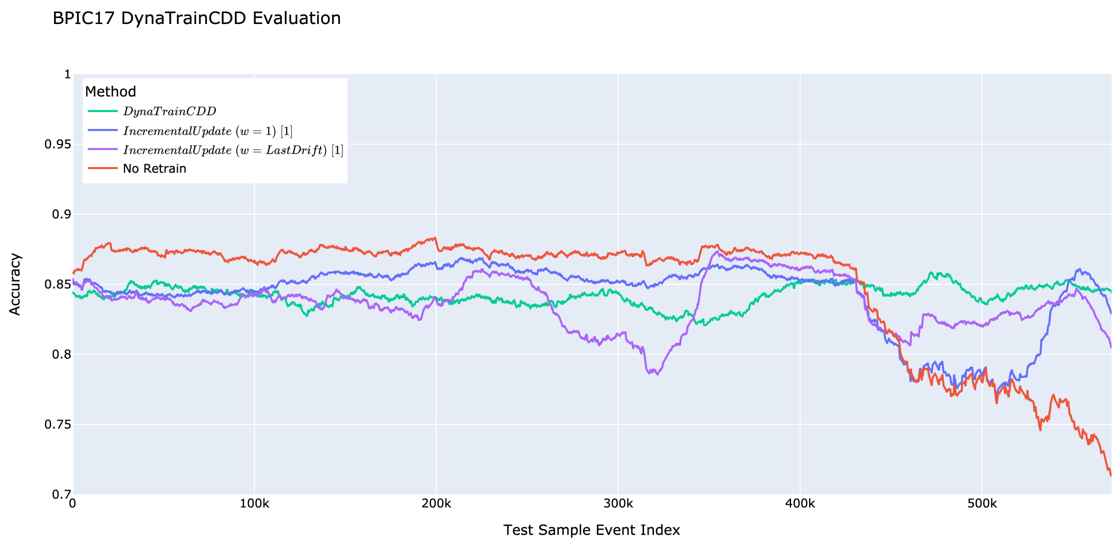


Fig. 8: External Data Validation (1 of 3)

Table 13: Validation Results (**Best** in bold, <u>Second Best</u> underlined, *Worst* italicised)

	BPIC15 Merged		BPIC17	
Method	Accuracy	Total Update Train Time	Accuracy	Total Update Train Time
DynaTrainCDD	**0.234**	<u>3460.941</u>	<u>0.843</u>	*3335.099*
IncrementalUpdate $(w = 1)$ [1]	<u>0.191</u>	**2701.935**	0.842	<u>2656.926</u>
IncrementalUpdate $(w = LastDrift)$ [1]	0.165	*10240.607*	*0.836*	**1539.348**
No Retrain	*0.133*	-	**0.845**	-

Table 14: BPIC15 Merged Post-Drift Performance (**Best** in bold, <u>Second Best</u> underlined, *Worst* italicised)

Method	**Drift 1**	**Drift 2**	**Drift 3**	**Drift 4**
DynaTrainCDD	**0.192**	**0.260**	**0.220**	**0.248**
IncrementalUpdate (w=1)	0.130	0.203	0.152	0.197
Last Drift	*0.124*	*0.197*	*0.144*	*0.175*
No Retrain	<u>0.184</u>	<u>0.255</u>	<u>0.182</u>	<u>0.217</u>

ing a (known) sudden drift as those which have been created in merging all subsets of the *BPIC15* dataset.

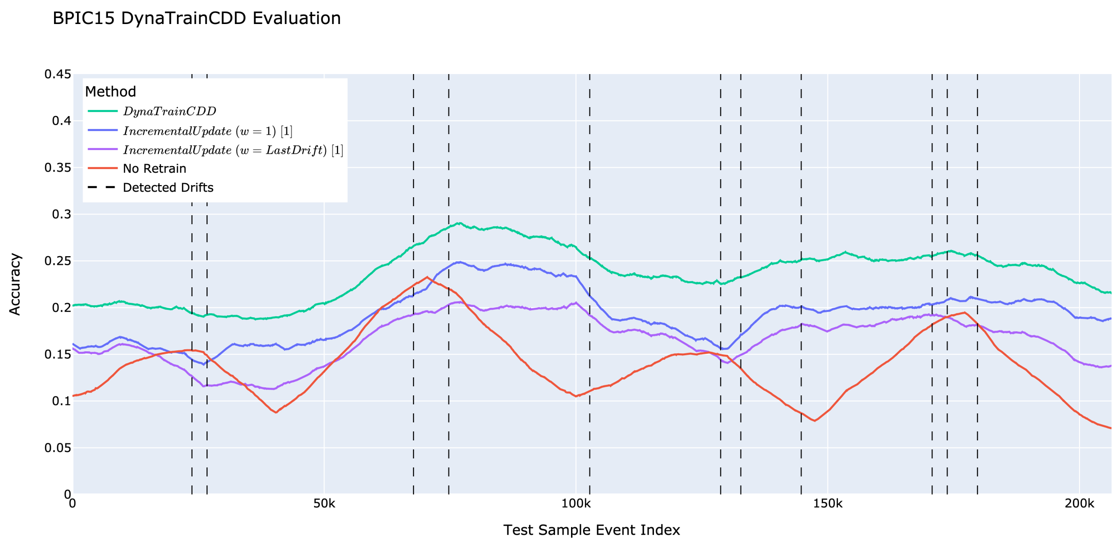


Fig. 9: External Data Validation (2 of 3)

4.6 Experimental Conclusion

Overall the results presented show that our *DynaTrainCDD* methods provide great value to the existing solutions in maximizing accuracy of process models. Additionally, the presented results suggest that even increasing the frequency with which updates are made could yield an easy way to increase accuracy of models while not requiring much time to develop such a solution. However, where our presented framework provides significant value is in higher accuracy without massive increases in running times when concept drift detection is necessary in streaming process event streams. With the developed *DynaTrainCDD* methods, we are able to take into account drifts as they occur and prioritize when more computation is a valid means to increase accuracy.

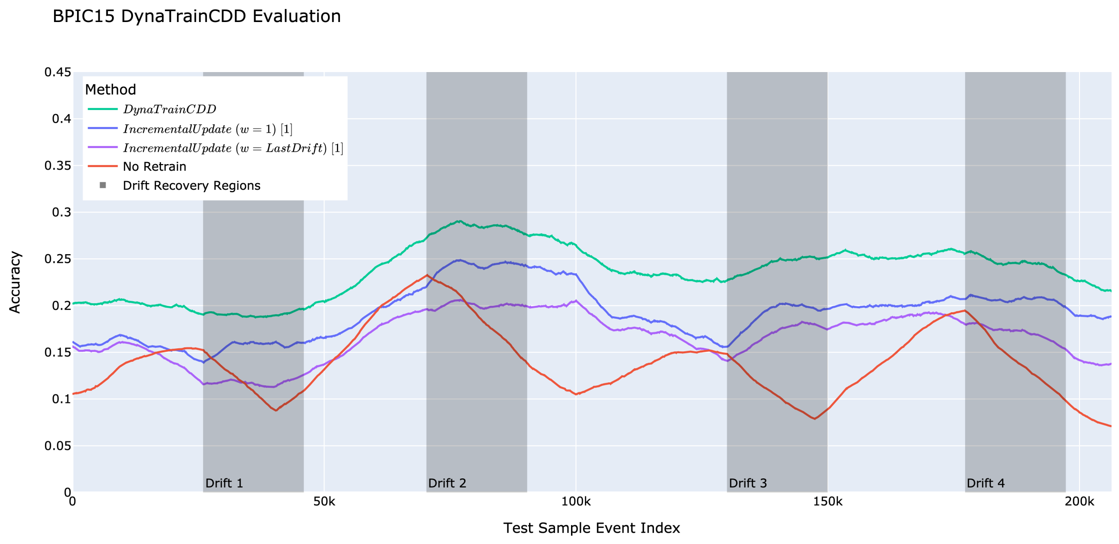


Fig. 10: External Data Validation (3 of 3)

5 Limitations

In the development of the procedures to combine the incremental updates as proposed by Pauwels [1] and the Prefix Trees as introduced by Huete Guzmán, J. S. et al. [2] certain assumptions needed to be made based on how those methods themselves were implemented.

It must be assumed that the it is possible to determine from a business process perspective when a trace has been marked as complete. This primarily means that it is possible to determine when there are no more events expected in the specific trace. In the case of the *BPIC15* data used in this analysis, this entails going through and finding the last seen event in each trace and marking that as the end of each trace. This is necessary for the purpose of notifying the prefix trees used for concept drift detection as to when they should remove the events only seen in older traces from the prefix trees once their relevance is outdated or obsolete.

Additionally an assumption is made regarding the processing of events ahead of time. Namely, this entails knowing the last 10 events in each trace that is currently being tracked due to the inputs needed for the modeling methods used by the Single Dense Layer (SDL) to generate predictions.

We also assume that one-hot encoding adds no significant improvement or detriment to results over changing categories to arbitrary integers as initially implemented in [1]. This implementation uses ordinal encoding as opposed to one-hot encoding which is found to perform better when processing input data to neural networks [26]. However, other alternatives do exist as described in [26] which also provide even better accuracy than one-hot encoding and this could lead to better results in future work.

It is assumed assumed that there is access to real-time performance on accuracy for all predictions previously from $\hat{p}_0$ until time $\hat{p}_t$ for the purpose of determining if some undetected drift has occurred.

The drift recovery curve which has been developed is assumed to be the best way to change the parameters of retraining the underlying predictive model. This proposed solution may not be the best solution and other more appropriate solutions may exist that were unknown at the time of this experimentation.

6 Conclusions

We have examined the context within process mining and relevant applications for incremental or dynamic retraining of process models. After laying out the steps for developing a dynamic framework we have shown the effectiveness and efficiency of retraining partial amounts of data. From our findings, we conclude that the use of such partial updates leads to comparable or better accuracy as well as acceptable running time performance on all datasets we have tested. Further research is needed on relevant fields such as whether there exists a better drift recovery formula as presented in this paper. Additionally, it would be helpful to examine more closely which types of drifts the drift detection algorithm chosen for this analysis (*PrefixTreeCDD*) is able to identify and determine if other identification methods may be better suited for this framework.

References

[1] S. Pauwels and T. Calders, "Incremental predictive process monitoring: The next activity case," 2021.

[2] J. S. Huete Guzmán, "Log-based concept drift detection over event streams," Master's thesis, Eindhoven University of Technology (TU/e) - The Netherlands, 2022.

[3] M. Hassani, "Concept drift detection of event streams using an adaptive window," vol. 33, pp. 230–239, European Council for Modelling and Simulation, 6 2019.

[4] J. Martjushev, R. Jagadeesh Chandra Bose, and W. van der Aalst, "Change point detection and dealing with gradual and multi-order dynamics in process mining," in *Perspectives in Business Informatics Research* (R. Matulevicius and M. Dumas, eds.), Lecture Notes in Business Information Processing, (Germany), pp. 161–178, Springer, 2015.

[5] R. Jagadeesh Chandra Bose, W. Aalst, van der, I. Zliobaite, and M. Pechenizkiy, "Dealing with concept drifts in process mining," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 1, pp. 154–171, 2014.

[6] W. M. P. van der Aalst, *Process Mining: A 360 Degree Overview*, pp. 3–34. Cham: Springer International Publishing, 2022.

[7] M. Hassani and S. Habets, "Predicting next touch point in A customer journey: A use case in telecommunication," in *Proceedings of the 35th International ECMS International Conference on Modelling and Simulation, ECMS 2021, Virtual Event, UK, May 31 - June 2, 2021*, pp. 48–54, European Council for Modeling and Simulation, 2021.

[8] Y. Spenrath, M. Hassani, and B. F. van Dongen, "Online prediction of aggregated retailer consumer behaviour," in *Process Mining Workshops* (J. Munoz-Gama and X. Lu, eds.), (Cham), pp. 211–223, Springer International Publishing, 2022.

[9] Y. Spenrath and M. Hassani, "Predicting business process bottlenecks in online events streams under concept drifts," in *Proceedings of the 34th International ECMS Conference on Modelling and Simulation, ECMS 2020, Wildau, Germany, June 9-12, 2020* (M. Steglich, C. Mueller, G. Neumann, and M. Walther, eds.), pp. 190–196, European Council for Modeling and Simulation, 2020.

[10] J. Serra, D. Suris, M. Miron, and A. Karatzoglou, "Overcoming catastrophic forgetting with hard attention to the task," in *Proceedings of the 35th International Conference on Machine Learning* (J. Dy and A. Krause, eds.), vol. 80 of *Proceedings of Machine Learning Research*, pp. 4548–4557, PMLR, 10–15 Jul 2018.

[11] B. Jeannet and A. Miné, "Apron: A library of numerical abstract domains for static analysis."

[12] A. Miné, "The apron numerical abstract domain library." https://github.com/antoinemine/apron, 2021.

[13] r. pyapron, "pyapron: A python api for apron." https://github.com/pyapron/pyapron, 2018.

[14] M. Jason T. Rich, M. J. Gail Neely, M. Randal C. Paniello, M. D. Courtney C. J. Voelker, M. Brian Nussenbaum, and M. Eric W. Wang, "A practical guide to

understanding kaplan-meier curves," *Otolaryngology–Head and Neck Surgery*, vol. 143, no. 3, pp. 331–336, 2010. PMID: 20723767.

[15] E. L. Kaplan and P. Meier, "Nonparametric estimation from incomplete observations," *Journal of the American Statistical Association*, vol. 53, no. 282, pp. 457–481, 1958.

[16] C.-D. Lai, D. Murthy, and M. Xie, *Weibull Distributions and Their Applications*, pp. 63–78. London: Springer London, 2006.

[17] H. Chen and D. G. Hoover, "Use of weibull model to describe and predict pressure inactivation of listeria monocytogenes scott a in whole milk," *Innovative Food Science & Emerging Technologies*, vol. 5, no. 3, pp. 269–276, 2004.

[18] M. K. Goel, P. Khanna, and J. Kishore, "Understanding survival analysis: Kaplan-Meier estimate," *Int J Ayurveda Res*, vol. 1, pp. 274–278, Oct. 2010.

[19] C. Velu, "Business model innovation and third-party alliance on the survival of new firms," *Technovation*, vol. 35, pp. 1–11, 2015.

[20] B. van Dongen, "Real-life event logs - Hospital log," 3 2011.

[21] B. van Dongen, "BPI Challenge 2012," 4 2012.

[22] I. Verenich, "Helpdesk dataset." https://data.mendeley.com/datasets/39bp3vv62t/1, 2016.

[23] B. van Dongen, "BPI Challenge 2017," 2 2017.

[24] B. B. van Dongen, "Bpi challenge 2015," May 2015.

[25] F. I. for Applied Information Technology (FIT), "Pm4py: Process mining for python." https://github.com/pm4py/pm4py-core, 2021.

[26] K. Potdar, T. S. Pardawala, and C. D. Pai, "A comparative study of categorical variable encoding techniques for neural network classifiers," *International Journal of Computer Applications*, vol. 175, pp. 7–9, 2017.

Appendix A Methods Plots

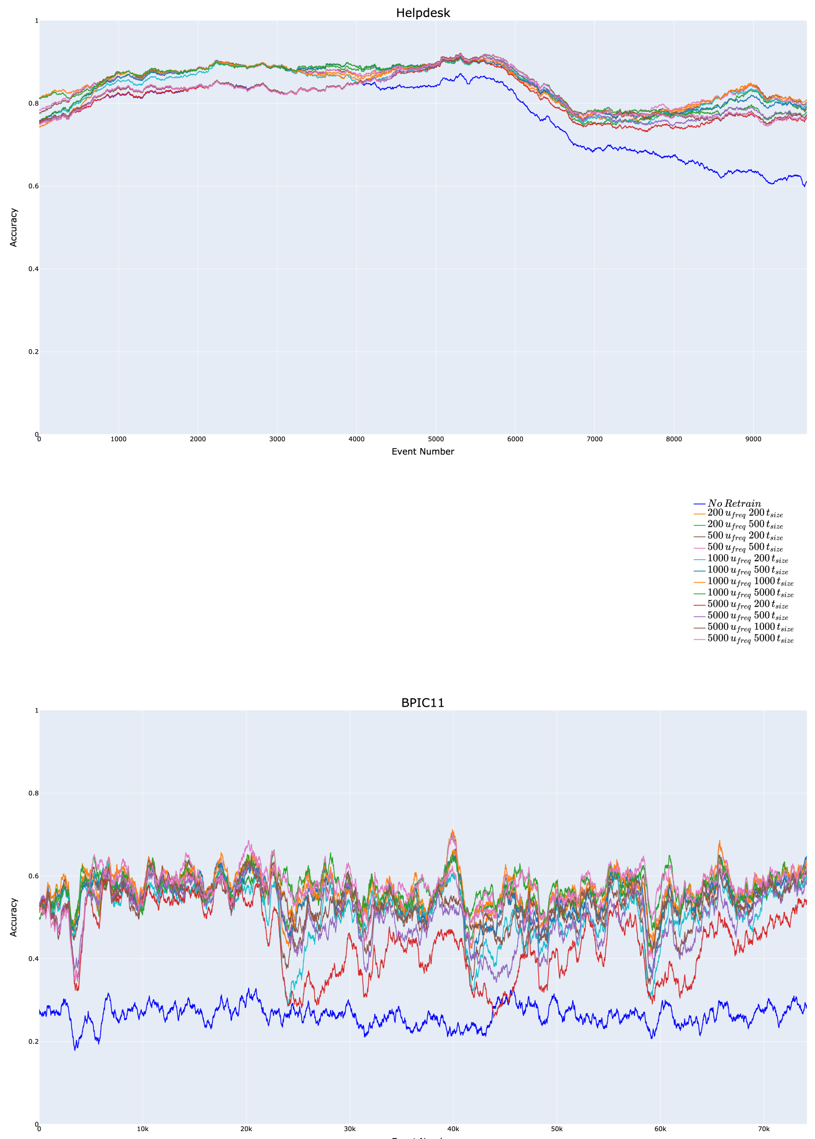


Fig. 11: Average Accuracy Results By Dataset (1 of 4)

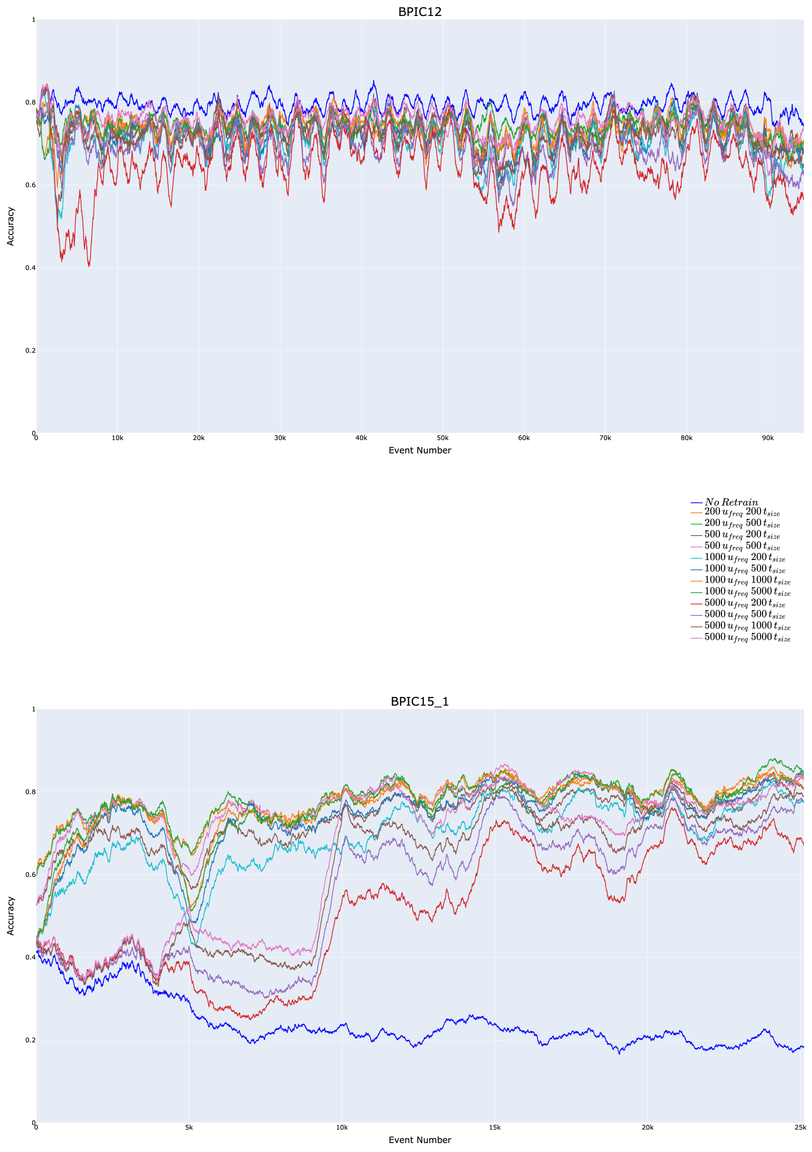


Fig. 12: Average Accuracy Results By Dataset (2 of 4)

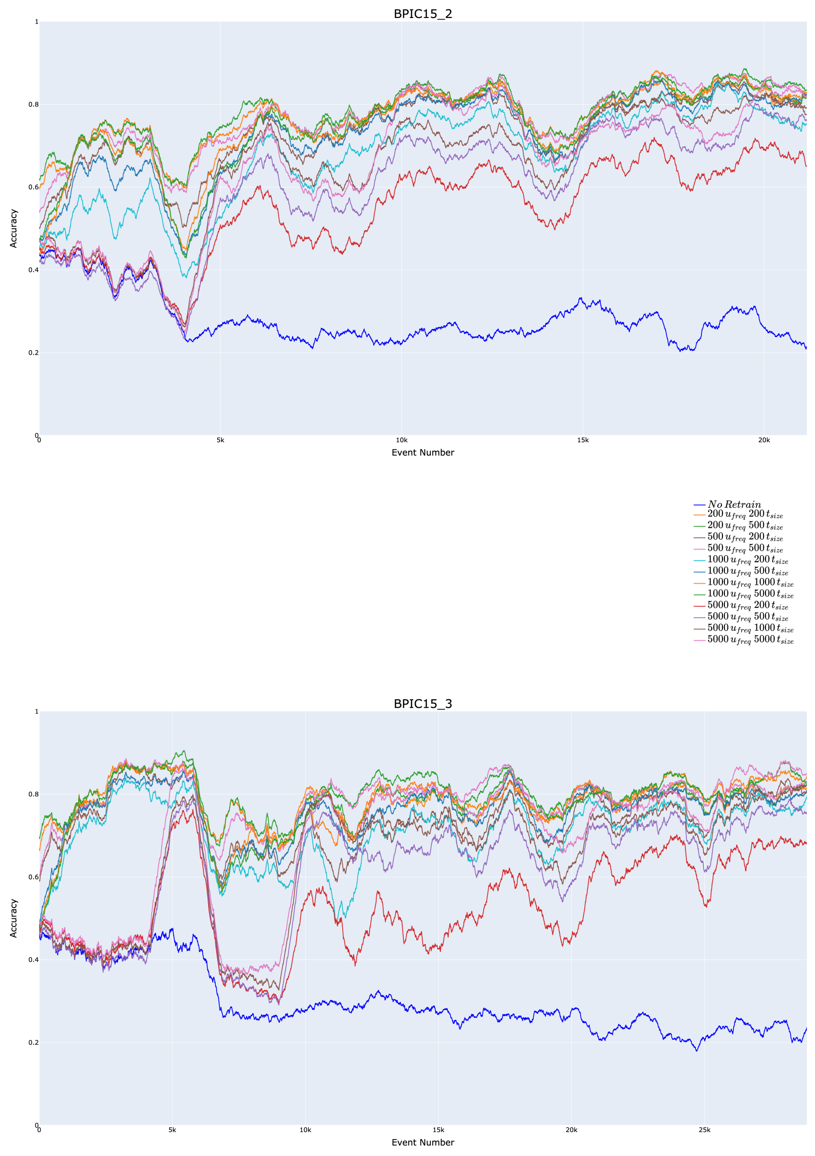


Fig. 13: Average Accuracy Results By Dataset (3 of 4)

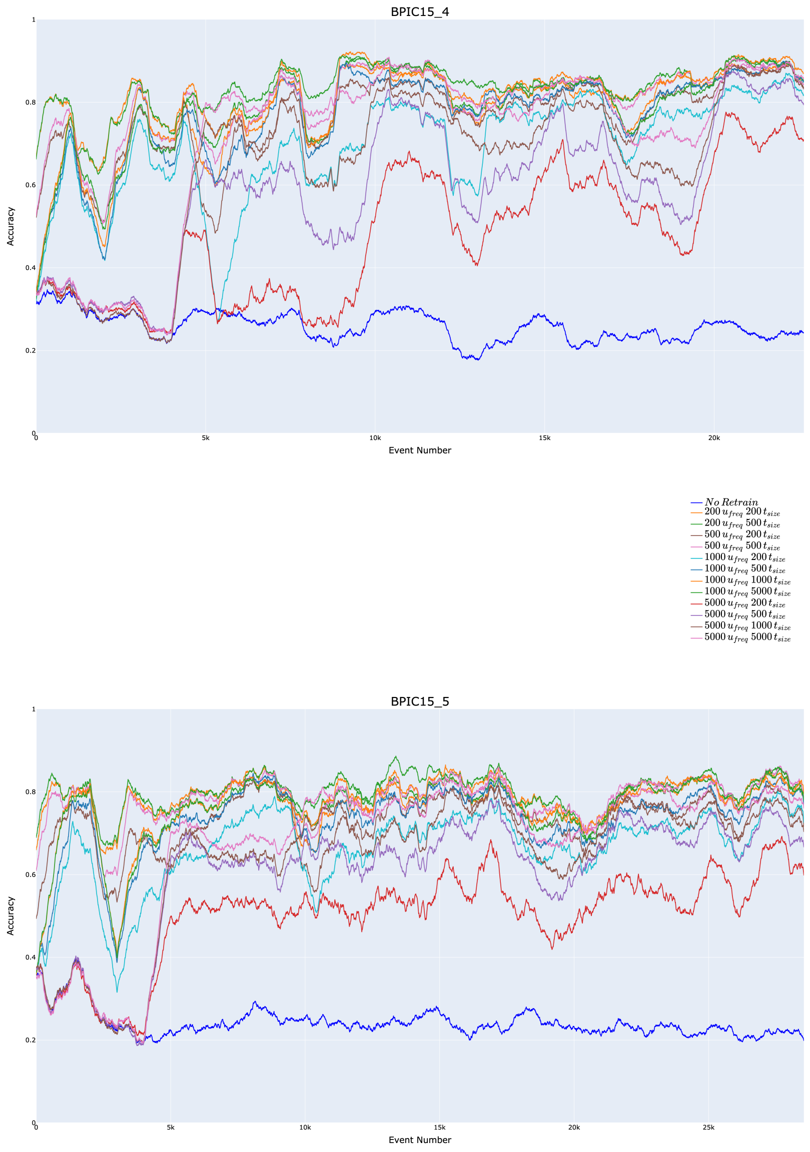


Fig. 14: Average Accuracy Results By Dataset (4 of 4)

Appendix B Drift Recovery Plots

Fig. 15: Idealised Drift Recovery Formula Plot

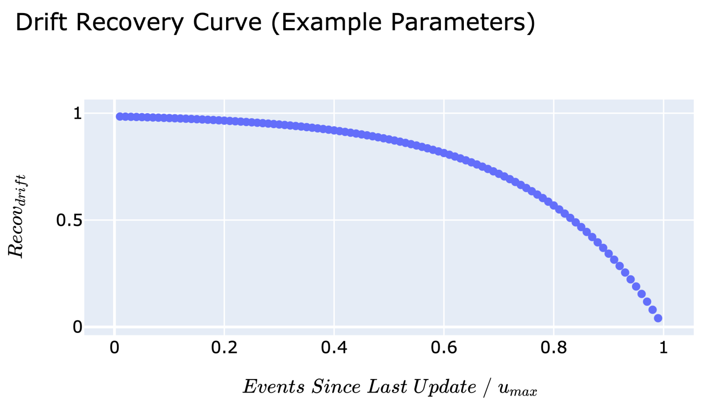
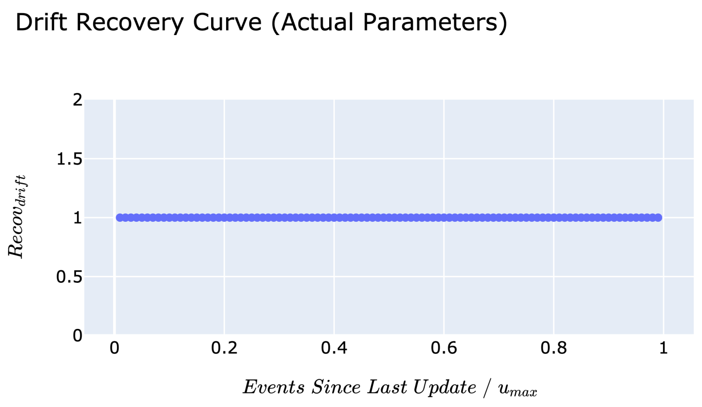


Fig. 16: Chosen Parameter Drift Recovery Formula Plot


Drift Recovery Curve (Actual Parameters)
$Recov_{drift}$
$Events\ Since\ Last\ Update\ /\ u_{max}$