Robust Stream Classification using Time Neutralising Decision Trees

Brandon Gower-Winter
Utrecht University
8 Hiedelberglaan, Utrecht, 3584 CS, NL
b.gower-winter@uu.nl

Georg Krempl
Utrecht University
8 Hiedelberglaan, Utrecht, 3584 CS, NL
g.m.krempl@uu.nl

Abstract

Data Streams are characterised as a potentially infinite source of data subject to non-stationarity. This non-stationarity, also called Concept Drift, often results in deployed predictive models having to be perpetually retrained as the underlying data generating process of the stream changes over time. Drift Detectors are a class of methods used to identify these changes, triggering these retraining procedures. Historically, Stream Learning has operated under the assumption that instances are only seen once, and cannot be stored for later use. Contemporary work relaxes this assumption, yet drift detection and simple resetting procedures are still ubiquitous. Building on related work which has shown that traditional Batch Learning algorithms often outperform Stream Learning algorithms when instances can be stored, we introduce Time Neutralising Trees (TNT), a Decision Tree architecture that enables robust stream classification in settings with Concept Drift. During training, TNT will filter out ("neutralise") old training samples that are no longer relevant given the current context of the stream. If those instances become relevant again, TNT will then re-include them. We evaluate TNT, and related algorithms on 24 real and semi-real classification data streams from the USP DS repository. Results show strong evidence that TNT achieves best-in-class performance on a wide-range of data streams with varying concept drift types.

1 Introduction

In an increasingly digitised world, data streams are at the heart of many online Machine Learning (ML) applications. These streams are often unbounded in size with samples arriving with extreme frequency. These streams also exhibit evolutionary behaviour in that they are often non-stationary. This non-stationarity is known as Concept Drift [17, 29, 36, 44]. In stream prediction settings such as stream classification, concept drift can have severe effects on a deployed model's predictive performance. In some cases, models may need to be perpetually retrained and old instances need to be forgotten if they conflict the current context of the stream. Historically, Change or Drift Detectors are used to identify when a model needs to be reset or updated [33, 27, 38]. This class of methods is particularly useful in the traditional stream learning setting where instances are only ever seen once and cannot be stored for any period of time.

In recent years, the restriction on storing instances has been criticised [44], with researchers noting that compute and memory resources are nowhere near as scarce as they used to be when this field was first founded decades ago. In fact, it is almost entirely impossible to effectively use modern ML methods such as Deep Neural Networks or many modern drift detectors [37, 38] without relaxing this restriction. Recent work has also shown that in settings where instances can be stored for a period of time, simple retraining or resetting regimes for batch ML algorithms often offers better predictive performance over the duration of a data stream when compared to traditional incremental stream learning methods [21]. Furthermore, the use of drift detectors to determine when a model should be

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

reset has been criticised, this is because most drift detectors only tell "When" drift has occurred, but often do not tell "Where" in feature space the drift occurred [36].

In this work, we aim to demonstrate that the somewhat arbitrary sample storage restriction, and over-reliance on drift detectors as reset mechanisms limits one's ability to develop robust stream classification algorithms. We do this by introducing a new method called the Time Neutralising Tree (TNT). In short, at every split decision node during training, TNT will first look to see if splitting on time (the arrival time indices of the instances) is the most informative split decision. If so, TNT will filter out (neutralise) older instances that do not match the label distributions of recent instances. In doing so, TNT is able to contextually determine which instances are relevant at any given point in the data stream, allowing it to adapt to concept drift phenomena without an explicit drift detection method. The contributions of our paper are three-fold:

1. We refine the definition of concept drift by conditioning its occurrence on sub-regions of the feature space, rather than the global perspective taken in most literature. Through identifying regions in the feature space that have drifted, we are able to provide formal definitions, using Information Theory, for when retraining or resetting are appropriate actions to take.
2. Using these theoretical findings, we introduce Time Neutralising Trees (TNT) which dynamically filter out instances that are irrelevant to learn on given the current context of the stream.
3. We evaluate TNT and other relevant methods on 24 real and semi-real data stream classification tasks taken from the USP DS repository [49]. Results show that TNT performs better than most other methods. We hope this inspires further investigation into stream learning methods that move away from the assumption that data storage is not allowed in stream learning. To the best of our knowledge, this paper provides the the most comprehensive evaluation of decision tree algorithms for data stream classification tasks.

The paper proceeds as follows: Section 2 provides a summary of related work, Section 3 describes TNT, Section 4 outlines our experimental design, Section 5 reports the results of our experiments and the paper concludes with a discussion and outline of future work (Section 6). All source code for this work is made available at: `https://github.com/BrandonGower-Winter/TNT`.

2 Background and Related Work

A data stream may be defined as a mode of access to a potentially infinite sequence of instances generated by some concept and delivered to a learning algorithm by an Instance-Delivery-Process (IDP) [44]. In a traditional supervised machine learning task, it is assumed that the underlying data generating process remains constant throughout time $(x_0, y_0), (x_1, y_2), \dots (x_n, y_n) \sim P(X, Y)$ [27], where x_i and y_i are the i-th instance's feature and label, sampled from stationary concept $P(X, Y)$.

In stream learning, the underlying data generating process may change over time such that each instance is potentially sampled from a different time-dependent concept $(x_0, y_0), (x_1, y_2), \dots (x_i, y_i) \dots \sim P_t(X, Y)$ with timestamp (or instance index) t. Formally, concept drift is then defined as a change in concept from one timestep to another $P_t(X, Y) \neq P_{t+1}(X, Y)$, although some have argued that it would be more appropriate to model concept drift as a distribution process [27]. Concept drift is said to typically occur in either the posterior distribution $P(Y|X)$ or in the covariates $P(X)$ with posterior drift typically requiring retraining or resetting while the covariate shift is only correlated with retraining or resetting [44]. There also exist several patterns that describe concept drift. For example, drift occurring over a short period of time is called abrupt drift. Drift occurring over a longer period of time is called incremental or gradual drift (the distinction between the two is not relevant for this work, see Lu et al. [36]). Drift occurring in a cyclical pattern (e.g., public transportation usage on weekdays vs. weekends), is known as recurring drift.

2.1 Drift Detectors

Formally, a drift detector can be defined as a binary classification task, where given a set of descriptors (descriptions of the observed data), a drift detector outputs a binary "Yes"/"No" depending on whether drift has been detected or not: $\mathcal{M} : W \mapsto \{0, 1\}$ where $\mathcal{M}$ is the drift detector and W is a window that contains the descriptions of the data. Drift detection typically consists of 4 stages [36, 27]. Stage

1 selects the instances to send to the drift detector (window-based methods are most common). Stage 2 uses the selected instances to build descriptors of the data. Stage 3 computes the dissimilarity of the descriptors, and Stage 4 calibrates the dissimilarity which identifies drift in the selected instances.

Drift detectors are typically categorised as either supervised or unsupervised. Supervised detectors (e.g., DDM [16], ADWIN [3], STUDD [7], PUDD [37]) typically use class labels and the predictions made by a deployed model to detect drift while unsupervised detectors (e.g., D3 [22], IBDD [48]) only require access to instance features. Both categories of detectors are limited in the types of drift they can detect [58] with supervised detectors only reliable when labelled data is present and if changes in the performance of the deployed predictor reliably indicate drift [32, 26, 44]. Conversely, unsupervised detectors can only detect changes in the covariates $P(X)$ and not in the posterior distribution $P(Y|X)$. Recently, drift detection has received criticism for being limited in the information they provide, as well as being subject to the Window Dilemma [21]. Most detectors only tell you "When" drift has occurred, but not "Where" in the feature space it has occurred [36]. As we will show, feature space conditioning is imperative in settings with concept drift, as it informs a deployed classifier on how it should adapt.

2.2 Incremental Decision Trees

Incremental Decision Trees and their derivates are the most commonly-used methods in stream learning. The most famous of which is the Hoeffding Tree (HT) [28] which uses the Hoeffding Bound to determine tree splits. Since its inception newer incremental decision tree such as the Extremely Fast Decision Tree (EFDT) [39] and PLASTIC [25] were developed which are less conservative when making split decisions. Even state-of-the-art stream methods (mostly ensembles) such as Adaptive Random Forests [19], Aggregated Mondrian Forests [41], Streaming Random Patches [20] and Streaming Gradient Boosted Trees [23] use incremental decision trees as their base models. Given that incremental decision trees are not drift-aware, they are typically combined with a drift detector (commonly DDM [16] or ADWIN [3]). A drift-aware incremental decision tree will wait for a drift-*warning* signal from its drift detector to start training new replacement tree in the background, and upon a drift-*reset* signal will replace the old with the background tree. This procedure represents a global reset mechanism by which the entire tree is replaced. There are tree variants such as the Hoeffding Adaptive Tree Drift (HAT) [4] and Locally Adaptive Streaming Tree (LAST) [1] which place drift detectors nodes in the tree structure. HAT uses a detector to reset subtrees when a drift-signal is received while LAST uses drift signals to determine when to split. These methods represent a more conservative approach where only drifted regions of the tree are replaced.

In the standard stream learning setting, when a drift event replaces or resets some number of components in a learning algorithm, the learned information contained in those components is typically lost forever. This is clearly undesirable in settings with recurring contexts where forgotten information may once again be useful for making accurate predictions. Typical solutions to this recurring contexts problem utilise archival techniques such as those FLORA [55], and COPL [30]. In literature however, decision tree architectures rarely explore archival strategies.

Nevertheless, it has been argued that much of the success of these methods is due to the Naive Bayes models that occupy the leaf nodes of these decision trees, rather that the tree algorithms themselves. This is because incremental decision trees are typically quite conservative when it comes to making decision splits [44]. Furthermore, recent work has shown that in settings where instances can be stored for some period of time, even simple retraining or resetting regimes using batch decision trees can outperform their stream learning counterparts [21]. In this work, we take this one step further by showing that in settings where instances can be stored for some period of time, decision trees architectures can explicitly determine which instances are relevant to train on given the current context of the data stream, thus enabling them to be drift adaptive.

3 Methodology

In this section, we introduce Time Neutralising Trees (TNT), a concept drift-robust batch learning method for stream settings where instances can be stored for period of time. Section 3.1 introduces the theory behind TNT, and Section 3.2 describes the method.

3.1 Motivation

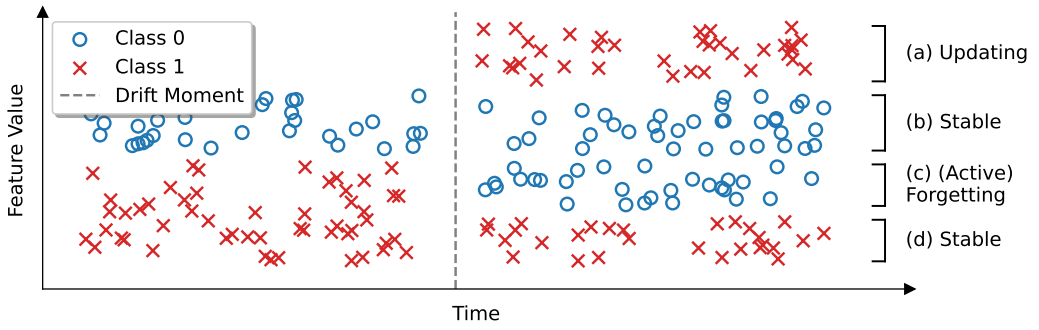


Figure 1: illustrates why feature subspaces must be taken into account when learning in settings with concept drift. Given a univariate stream with a single drift moment (dashed line) and assuming a model was trained on data prior to the drift, different actions need to be taken to update the model on the post-drift data, depending on the location in the feature space. Subspace (a) is entirely new, so the optimal action is to learn in that new region. Subspaces (b) and (d) are stable (i.e., no drift), so no action needs to be taken (the model still classifies instances in those regions correctly). Subspace (c) however, requires that the model actively forgets instances from the past before learning on instances from the current time context (i.e., the pre-drift context conflicts with the post-drift context).

We first discuss the theory behind TNT. Figure 1 illustrates a synthetic univariate stream with a concept drift at time some arbitrary time point t. Given a model θ trained on instances prior to the drift event, we want to decide how θ should be updated to adapt to the drift. As shown, updating θ depends on where in the feature space we observe drift. In subspaces (b) and (d), no drift has occurred and θ does not need to be updated. Subspace (a) is entirely new (i.e., there are no past instances in this subspace) so the optimal action is to simply update θ on those new instances. However, subspace (c) represents a region where past instances are no longer relevant in the current context of the stream (i.e., they degrade prediction quality). Here the optimal action is to actively forget those old instances and update θ on the new data. When viewed holistically, subspaces (a, b, d) require no further action beyond retraining, which given a sufficient set of instances X and labels Y, any suitable batch learner can adapt to. We will now focus on scenarios like subspace (c) where active forgetting is useful.

Consider a non-stationary data stream for a supervised learning task $S = \{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n), \ldots\}$ where $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$. In non-stationary streams, an instance $(x_i, y_i) \sim \mathcal{D}_{t=i}$ where $\mathcal{D}_t$ is a distribution map of time to a concept $\mathcal{D} : t \to P_t(X, Y)$. If a drift occurs at timestep t_{drift}, $\mathcal{D}_{t<t_{drift}} \neq \mathcal{D}_{t \geq t_{drift}}$.

In a stationary stream, expected Mutual Information (MI) $I(X;Y) = H(Y) - H(Y|X)$ remains constant throughout the lifetime of a stream. In a non-stationary stream, the MI of the entire stream is dependent on the global distribution which is a weighted mixture of the $K = |\mathcal{D}|$ *distinct* concepts that have occurred in the stream's lifetime:

$$\bar{P}(X, Y) = \sum_{k=1}^{K} w_k P_k(X, Y) \tag{1}$$

where w_k is the normalised duration of concept $P_k(X, Y)$. If the number of distinct concepts $K \to \infty$, then the MI of the stream tends to 0 (see Appendix A.2):

$$\lim_{K \to \infty} I_{\bar{P}}(X;Y) = 0 \tag{2}$$

Note that we are particular in our usage of the word *distinct*. Two streams constructed using concepts $A \to B \to C$ and $A \to B \to A \to C$ resp. have the same number of *distinct* concepts. We have included empirical simulations that support Eq. 2 in the Appendix (A.2). Given that MI is convex with respect to the conditionals $P(Y|X)$ and fixed distribution $P(X)$ [9], Jensen's Inequality states

that the expected MI of the entire stream must be less than or equal to the expected MI of the *distinct* concepts in that stream:

$$I_{\bar{P}}(X;Y) \leq \sum_{k=1}^{K} w_k I_k(X;Y) \tag{3}$$

Using Eq. 3, we now have a method for solving the active forgetting problem in subspace of Fig 1. Given a set of timesteps $T = \{0, 1, \ldots, n, \ldots\}$, our goal is find some time interval $\mathcal{T}^* \subseteq T$ where:

$$\mathcal{T}^* = \underset{\mathcal{T} \in \mathcal{P}(T)}{\operatorname{argmax}} \; I(X;Y|\mathcal{T}) - I_{\bar{P}}(X;Y) \tag{4}$$

In practice, we would want $\mathcal{T}^*$ to include the most recent timestep $n \in \mathcal{T}^*$. That way we ensure the time interval we find is relevant to the current stream context. Note that, fundamentally, supervised drift detectors already approximate this process, but without the ability to recover instances once they are forgotten due to the inability of most stream learning algorithms to store past data. In a modern setting where we do have the ability to store data, we can approximate $\mathcal{T}^*$ more accurately. Decision Trees are naturally equipped to do this as they maximize information gain through partitioning the feature space. This is the primary motivation for TNT, by treating time T as an additional feature, we are able to find subspaces where active forgetting is useful, thus allowing for a drift robust batch learner to be trained without an explicit drift detection mechanism.

3.2 Time Neutralising Trees

In this Section, we describe the Time Neutralising Tree (TNT). Like traditional decision trees, given a set of instances X, and labels Y, TNT partitions the feature space by impurity reduction (TNT uses Information Gain, see Alg. 1). This process continues until some stopping condition is reached for all possible decision paths (TNT uses minimum samples N_{min}). Rather than maintaining an incremental decision tree, TNT aims to create a new decision tree optimised to the current time point. To this end, from all currently available data TNT filters out (neutralises) the irrelevant data during the training process. Thus, unlike traditional decision trees, *during training* TNT also accepts the time indices of the instances T. These could be the actual clock times of each instance's arrival or a simple index denoting the instance's position in the stream. At every decision node, TNT will also calculate the best possible split on T. Then, using the following rule:

$$\text{BestTimeSplit}(T, Y) > \text{BestFeatureSplit}(X, Y) + \lambda \tag{5}$$

TNT determines if time is a more informative feature than the best feature split. If that is the case, TNT begins the filtering process. $\lambda \in [0, 1)$ controls the reactivity of TNT. If λ is close to 0.0, it will filter out instances as soon as time is the most relevant feature. As λ increases, TNT becomes more reluctant to filter historical instances unless the IG from such a split is sufficiently large. We include λ as a tuneable parameter because streams can exhibit various phenomena such as label autocorrelation and temporal dependencies [49] which necessitate different degrees of reactivity.

When the filtering process is initiated on a decision node (Alg. 2), TNT will create a subtree that splits only on time until the node purity, minimum samples (N_{min}) or some minimum level of information gain τ are reached. Then, using the right most leaf node (i.e., node that most accurately represents the current stream context for that subspace) of the time tree as reference, a Mann-Whitney U test is performed ($p = \alpha_{mwu}$) on each other leaf node's label distribution. We use the Mann-Whitney U test because it carries the fewest assumptions about the label distribution, but it could be replaced with another relevant statistical test. If the U test fails (i.e., $p > \alpha_{mwu}$), that leaf node is merged with the reference leaf node as it contains data that is relevant in the current stream context (i.e., a node with a label distribution similar to the reference node's label distribution contains information that is relevant in the current stream context). All leaf nodes in the time subtree that are not merged are discarded, and all merged nodes are used to create a set of filtered instances that are returned to the main tree (See Fig. 2 for visual aid.). Note: we cannot only perform a single split on time because relevant instances may exist in disjoint subsets of time (e.g. reoccurring concepts as in Fig. 2b), so we must continually split during the filtering process until the stopping conditions are met. It is this

filtering process that allows TNT to "neutralise" the effects of concept drift in areas where active forgetting is useful.

Once a filtering step is performed, the filtered instances, their labels, and time indices are returned (lines 13-14 in Alg. 1). TNT will then recalculate the best feature split for the filtered instances and continue the training process. Note: the time indices T are only used during the training process, not during inference. That is because any splits on time immediately become irrelevant as new instances arrive at inference time. Time is only used to filter out training instances.

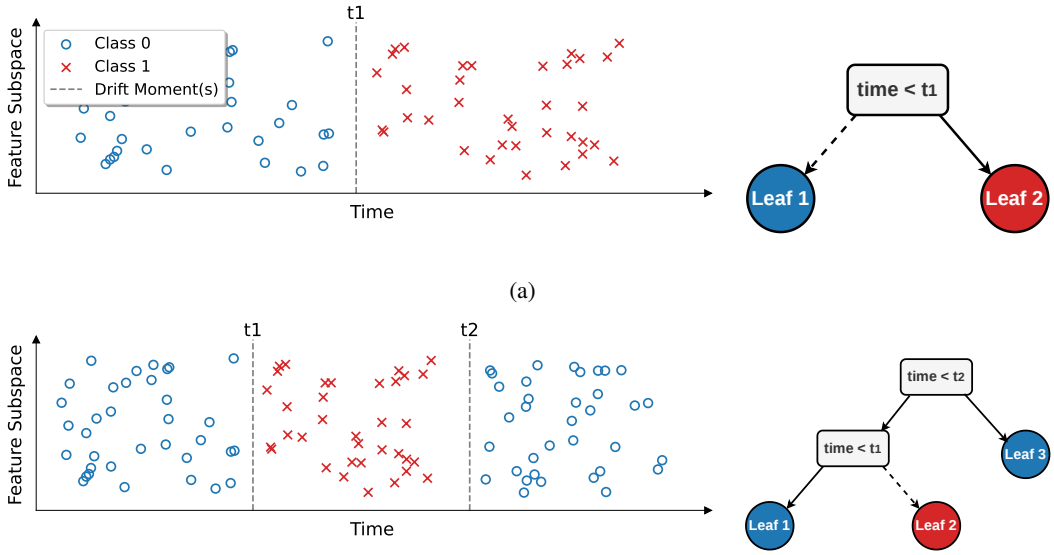


Figure 2: illustrates the time filtering process used by TNT. Here we see an arbitrary subspace of a univariate stream at two different moments in time. Figure (a) shows the stream after one drift has occurred at time $t1$. The corresponding time filtering tree is shown on the right whereby the old instances (class 0, blue) are filtered out (dashed line) and no longer considered by TNT when determining additional split nodes during training. Figure (b) shows the same subspace at a later moment in time where a reoccurring drift is seen at $t2$. The corresponding time filtering tree now filters out the previously selected instances (class 1, red, dashed line) and keeps the class 0 instances for further training.

4 Experimental Design

For our evaluation, we make use of 24 real and semi-real data streams taken from the USP DS repository [49]. The streams are binary and multi-class classification tasks that cover a wide-range of application domains such as Urban Development, Medicine, and Epidemiology. These stream vary in the number of features they possess (between 2 and 3600) and sample size (between 503 and 581,012). A summary of these streams are included in the Appendix (A.3).

As is standard practice in stream learning, we make use of the test-then-train paradigm for evaluation. Thereby, deployed models first predict on instances before they train on them. For baselines, we make use of Majority Class (MC) and Last Class (LC) classifiers. These are also sometimes called Prior and NoChange classifiers which predict the majority and last label seen, resp. These two baselines reveal class imbalance and autocorrelation in the labels which are common in data streams. We evaluate the following stream learning algorithms: Hoeffding Tree (HT) [28], Extremely Fast Decision Tree (EFDT) [39], PLASTIC (PLS) [25], Hoeffding Adaptive Tree (HAT) [4] and Locally Adaptive Streaming Tree (LAST) [1]. HAT and LAST are the only drift-aware models. Given this, we also make use of two supervised drift detectors: ADWIN [3] and DDM [16] and one unsupervised drift detector: D3 [22]. All three detectors are used to reset a Hoeffding Tree when they detect drift. In our notation, HT_{DD} thus refers to a Hoeffding Tree that is reset completely when the drift

detector (DD, which is either ADWIN, DDM or D3) signals that drift has been detected. We do not evaluate more unsupervised detectors because their comparison is somewhat unfair. Unsupervised detectors typically require that you predefine their detection behaviour. Here we use D3 to reset a model, but unsupervised detectors can also be used as monitoring algorithms to determine when new instance labels should be acquired, which is beyond the scope of this work. For all stream learning methods, once they make a prediction on a previously unseen instance, they immediately train on it in accordance with test-then-train procedure.

For the batch learners, we make use of standard batch Decision Trees (DT) and sliding windows. Given that we cannot train batch DTs incrementally, they are retrained from scratch every 50 instances. The DTs we use are: DT_{50}, DT_{100}, DT_{250}, DT_{500}, and DT_{∞}. That is, a decision tree DT_n is trained on the last n instances of the stream. This gives us an broad overview of the types of performance one can achieve when data can stored a various sizes. For DT_{∞}, it is trained on all previously seen instances in the stream. For TNT, it follows the same training procedure as DT_{∞}. However, we acknowledge that storing all data over an entire stream is infeasible in certain settings so we also evaluate TNT_{500} which follows the same training procedure as DT_{500} (i.e., every 50 instances, the model is retrained on the last 500 instances). With this selection, we believe our evaluation covers a broad range of learning strategies over varying time scales and data storage demands.

During evaluation, we track each model's Cohen's Kappa Score [8], recording it every 50 instances over the last 50 instances. The exception to this are the first 50 instances for which no metrics are tracked as this period is used as a warm start for all methods. We use kappa score because accuracy is not always an effective metric in stream settings [18]. For all stochastic methods, 10 replicates per stream are run using a pseudo-random number generator and provided seeds to ensure reproducibility. Fractional ranking is used to determine the ordering of methods per stream (i.e., if the kappa for some methods are [85.0, 80.0, 80.0, 60.0] their corresponding ranks are [1, 2.5, 2.5, 4]). Each method is ranked by mean kappa score, and a Mann-Whitney U test ($p = 0.05$) with Holm-Bonferroni correction is used to determine if two methods are statistically different. If they are not, they are given the same rank. Once ranking is complete, a critical difference diagram ($p = 0.05$) is constructed [11, 2] to determine statistical significant rankings across the evaluated methods. Additional experimental setup details are included in the Appendix (A.3).

4.1 Hyper-parameter Tuning and Sensitivity Analysis

Before evaluating TNT, we sought to understand how its 5 tunable parameters affect its predictive performance. They are: filter factor λ, minimum split node samples N_{min}, minimum time node entropy split τ, confidence threshold for determining time node merges p_{mwu}, and the maximum number of threshold splits to consider per feature thresholds_{max}. To avoid data leakage and bias [49], we parameter tune TNT on 5 synthetic streams which cover a variety of drift types. They are: `Hyperplane` [28], `RandomRBFDrift` [40], `SEA` [51], `STAGGER` [46] and a custom stream `RotatingMoons` (A.4). After some initial exploration of the hyper-parameters, we used Quasi-Monte Carlo Latin Hypercube Sampling [42] to generate 100 parameter configurations which we explored across all synthetic streams, recording the final accuracy of the model per configuration per stream. For each stream, we then train a Random Forest [6] regressor using the parameter configurations as input and the final accuracy as a target. From each trained Forest, we extract feature importances (A.5.1). We also plot each parameter and obtained accuracy per stream using LOWESS regression to analyse per feature level dependencies (A.5.2).

Results indicate that N_{min} and thresholds_{max} are generally the most important hyper-parameters across the synthetic streams. This is understandable as they both control the model's ability to generalise. For `Hyperplane`, a low filter factor λ is the most significant parameter. This is because active forgetting is pivotal in this stream where old instances rapidly become outdated. For the other streams, there is a evidence to suggest that too low of a filter factor (which increases instance filtering) is detrimental to the TNT's performance, confirming our intuition that there needs some balance between reactivity and preservation in the time filtering process. Interestingly, λ was not as pivotal in determining model performance as we expected. We discuss this further in the postmortem (A.7.1).

For the final step, we performed an ablation on TNT whereby we replaced impure leaf nodes with either a Logistic Regression (LR) or Naive Bayes (NB) model as is done in incremental decision trees. This is in contrast to the majority class prediction strategy used by most batch decision trees. Results (A.5.3) indicated that using LR on non-pure leaf nodes gave a slight performance improvement as

Table 1: Hyper-parameters and their respective values used in our evaluation.

Name	Description	Value
λ	Controls the time filtering sensitivity.	0.20
N_{min}	Minimum number of samples needed to make a spit decision.	20
τ	Stopping threshold for when time splits are not informative.	0.05
α_{mwu}	Threshold used to merge time leaf nodes using Mann-Whitney U Test.	0.05
thresholds$_{max}$	Number of split candidates generated per feature.	15
leaf model	Predictor that replaces non-pure leaf nodes.	LR

it mitigates the effects of poorly chosen split node minimums N_{min} and thresholds$_{max}$ which, as noted, greatly affect TNT's generalisability. A complete description of the parameter tuning process is included in Appendix (A.5), and Table 1 lists the final hyper-parameters used in our evaluation.

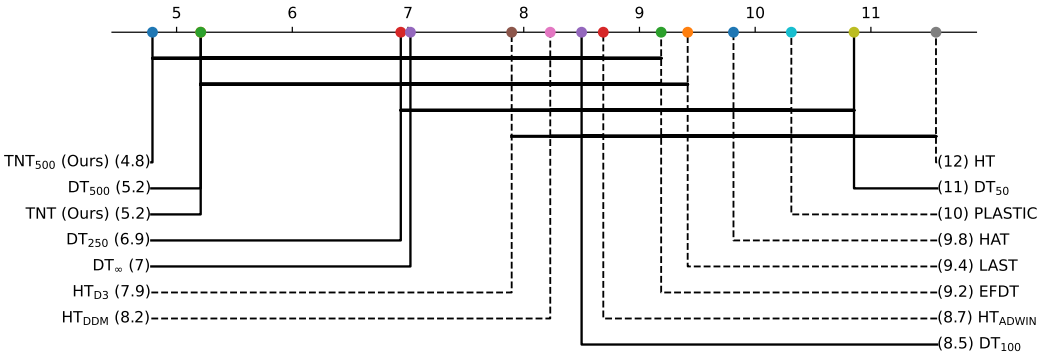


Figure 3: Critical Difference Diagram ($p = 0.05$) of all methods across all streams investigated. A lower rank is better. Dashed lines represent Stream Learning methods while solid lines represent Batch Learning methods. Solid horizontal bars represent cliques for which no statistical significance could be found. Baselines MC and LC are omitted for clarity.

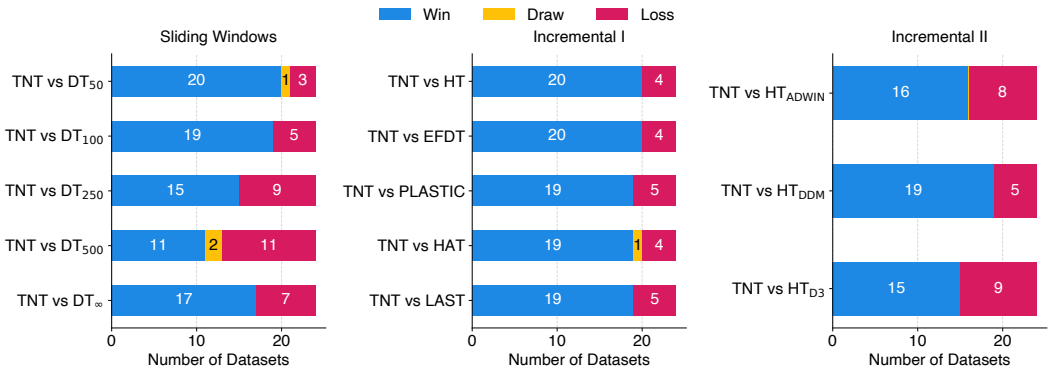
5 Results

Table 7 reports on the average Kappa score for all models across all streams investigated, and Fig. 3 summarises the results with a CD diagram. Overall, results show a clear separation in the performance between batch learning DTs and the incremental DTs. All incremental DTs rank noticeably lower than the best performing batch DTs (TNT$_{500}$, TNT and DT$_{500}$ resp.). These results further echo the findings in Gower-Winter et al. [21], if training instances can be stored for any period of time, it is often better to simply retrain a model on those instances than use an incremental DT. Our results indicate as few as 250 samples is enough when evaluating on the USP DS repository.

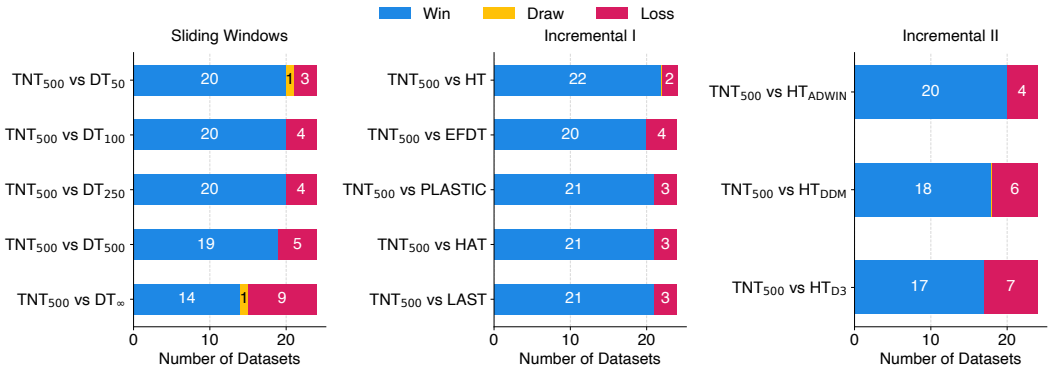
TNT performed better than expected with both variants (TNT$_{500}$, TNT) achieving mean ranks (4.8, 5.2 resp.) higher than their traditional batch DT equivalents (DT$_{500}$, DT$_{\infty}$, ranks 5.2, 7 resp.). This shows that TNT adapts to the current context of the stream effectively (A.6) . This is particularly noticeable on the `GasSensorArray` (GS), `StartLightCurves` (SL) and `Yoga` (YG) streams. TNT$_{500}$ on average ranked higher than TNT implying that passive forgetting may still be useful even when using active forgetting strategies like the instance filtering used by TNT. This is further supported by the higher rank of DT$_{500}$ relative to DT$_{\infty}$. One possible reason for this is that fewer, but still sufficient, samples prevent the batch DTs from overfitting on historical data. For TNT, passive forgetting using a sliding window also counteracts some loss in predictive performance that can occur when λ is tuned too conservatively. Regardless, Fig. 4 shows that both variants of TNT explored in this work outperform both the traditional batch and incremental decision trees in the vast majority of cases.

For the incremental DTs, there are several streams where they noticeably outperform the batch DTs. They are Electricity (EL), Forest Cover Type (FC), and Outdoor Objects (OD). This is discussed in more detail in the postmortem (A.7.2), but these also happen to be the data streams with high

autocorrelation in the labels and/or streams that have been extensively studied throughout the history of stream learning. For every stream that the incremental learners excelled at, there are equally as many streams with high feature counts or class imbalances for which they are unable to generalise at all (e.g., Cancer (CA), Heartbeats (HB), and Ozone (OZ)). The results for the incremental DTs are also inflated because they are able to train immediately after making a prediction, ensuring they are always as up-to-date. In contrast, the Batch DTs must make 50 predictions before they are allowed to update themselves. Related work [21] has shown that the predictive performance of incremental learners decreases (sometimes significantly) when they are subject to verification latency [31]. Nevertheless, our results are clear. When historical instances can be stored for a period of time, batch learning methods typically outperform their incremental learning counterparts. With this new framing, TNT stands as a best-in-class, drift-robust stream classifier.



(a)



(b)

Figure 4: the win-loss ratio of TNT (a) and TNT_{500} (b) on the USP DS repository benchmark against the other methods investigated in this work. A win means that the method on the left had a significantly greater Kappa score the method on the right. The first column reports on the batch (sliding window) learners and columns two and three report on the incremental learners.

5.1 TNT as a Preprocessing Step

The results from our primary investigation indicate that TNT performs best-in-class for single decision trees. However, state-of-the-art ensembles, such as Adaptive Random Forests [19] and Streaming Random Patches [20] out perform single decision trees in most, memory unconstrained, settings. During the development of TNT we found that simply creating an ensemble of TNT trees was not an effective strategy as it results in many repeated computations across the TNT base models. Thus, we explicitly leave the development of a TNT ensemble for future work. Nevertheless, this section demonstrates that TNT can be used as a preprocessing step to enable more complex batch learning architectures such as Random Forests [6] and Gradient Boosting models [14, 15] to be drift-adaptive.

Table 2: Average Kappa Scores of batch learning algorithms (left) with and without using TNT as a preprocessing step to filter out irrelevant samples before training, compared to state-of-the-art stream learning ensembles (right). Kappa scores are averaged over 10 replicates per method on the Hyperplane data stream.

Method	Kappa without TNT	Kappa with TNT
TNT	0.72.	-
DT	0.38	0.73
RF	0.39	0.76
GB	0.38	0.75
MLP	0.39	0.74
DNN	0.39	0.75

Method	Kappa
AMF	0.52
ARF	0.72
SRP	0.76

Table 2 reports the results of comparing traditional Decision Tree (DT), Random Forest (RF), Gradient Boosting (GB), Multi-Layer Perceptron (MLP), and Deep Neural Network (DNN) architectures with and without the use of TNT as a preprocessing step. When TNT is used for preprocessing, all samples that TNT keeps (i.e., does not filter away) are used to train the primary architecture. We also compare the results of this preprocessing to state-of-the-art streaming ensembles ARF, SRP and Aggregated Mondrian Forests (AMF) [41]. We use the `Hyperplane` stream evaluated under the same conditions as the main experiments. We use `Hyperplane` because it simulates pure posterior drift (change in P(Y|X)), enabling us to isolate the effectiveness of the TNT preprocessing step at filtering away irrelevant training samples. Results show that TNT enables models, with no drift adaptation capabilities to match the match the performance of state-of-the-art stream learning algorithms.

6 Conclusions and Future Work

This paper introduces Time Neutralising Trees (TNT), a batch learning decision tree architecture that is robust to concept drift. TNT uses the time indices of instances as a feature during training, whereby training samples that are no longer relevant to the current context of the stream are filtered ("neutralised") out. We evaluated TNT on 24 real and semi-real data streams from the USP DS repository [49] which showed that TNT achieves best-in-class performance when compared to other decision tree architectures. The main takeaway from this paper is that when historical data can be kept for some period of time, batch learning strategies outperform their stream learning counterparts. In such settings, TNT stands as viable candidate for creating drift robust stream classifiers.

Future work will address the this work's limitations. Namely, by extending the time-filtering procedure of TNT to ensemble architectures such as Random Forests [6], so that TNT's filtering methodology can be compared to state-of-the-art stream learning algorithms such as Adaptive Random and Aggregated Mondrian Forests [19, 41], Streaming Random Patches [20] and Streaming Gradient Boosted Trees [23]. We also seek to reduce the sensitivity of TNT on the time filtering threshold λ by making it context sensitive. Intuition says that a Hoeffding-like bound may be useful as is the case in incremental decision trees [28].

References

[1] D. N. Assis, J. P. Barddal, and F. Enembreck. Just change on change: Adaptive splitting time for decision trees in data stream classification. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, pages 351–357, 2024.

[2] A. Benavoli, G. Corani, and F. Mangili. Should we really use post-hoc tests based on mean-ranks? *The Journal of Machine Learning Research*, 17(1):152–161, 2016.

[3] A. Bifet and R. Gavalda. Learning from time-changing data with adaptive windowing. In *Proc. of the 2007 SIAM Int. Conf. on Data Mining*, pages 443–448. SIAM, 2007.

[4] A. Bifet and R. Gavalda. Adaptive learning from evolving data streams. In *Int. Symposium on Intelligent Data Analysis*, pages 249–260. Springer, 2009.

[5] J. A. Blackard and D. J. Dean. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and Electronics in Agriculture*, 24(3):131–151, 1999.

[6] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[7] V. Cerqueira, H. M. Gomes, A. Bifet, and L. Torgo. STUDD: A student–teacher method for unsupervised concept drift detection. *Machine Learning*, 112(11):4351–4378, 2023.

[8] J. Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[9] T. M. Cover and J. A. Thomas. Elements of information theory. 2006.

[10] H. A. Dau, E. Keogh, K. Kamgar, C.-C. M. Yeh, Y. Zhu, S. Gharghabi, C. A. Ratanamahatana, Yanping, B. Hu, N. Begum, A. Bagnall, A. Mueen, G. Batista, and Hexagon-ML. The UCR Time Series Classification Archive, 2018.

[11] J. Demšar. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine learning research*, 7(1):1–30, 2006.

[12] D. Dheeru and E. K. Taniskidou. UCI Machine Learning Repository. 2017.

[13] G. Ditzler and R. Polikar. Incremental learning of concept drift from streaming imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 25(10):2283–2301, 2012.

[14] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[15] J. H. Friedman. Stochastic gradient boosting. *Computational statistics & data analysis*, 38(4):367–378, 2002.

[16] J. Gama, P. Medas, G. Castillo, and P. Rodrigues. Learning with drift detection. In *Brazilian Symposium on Artificial Intelligence*, pages 286–295. Springer, 2004.

[17] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A survey on concept drift adaptation. *ACM Computing Surveys (CSUR)*, 46(4):1–37, 2014.

[18] H. M. Gomes, J. P. Barddal, F. Enembreck, and A. Bifet. A survey on ensemble learning for data stream classification. *ACM Computing Surveys (CSUR)*, 50(2):1–36, 2017.

[19] H. M. Gomes, A. Bifet, J. Read, J. P. Barddal, F. Enembreck, B. Pfharinger, G. Holmes, and T. Abdessalem. Adaptive random forests for evolving data stream classification. *Machine Learning*, 106(9):1469–1495, 2017.

[20] H. M. Gomes, J. Read, and A. Bifet. Streaming random patches for evolving data stream classification. In *2019 IEEE international conference on data mining (ICDM)*, pages 240–249. IEEE, 2019.

[21] B. Gower-Winter, M. Groen, and G. Krempl. The window dilemma: Why concept drift detection is ill-posed. In *International Symposium on Intelligent Data Analysis*, pages 371–384. Springer, 2026.

[22] Ö. Gözüaçık, A. Büyükçakır, H. Bonab, and F. Can. Unsupervised concept drift detection with a discriminative classifier. In *Proc. of the 28th ACM Int. Conf. on Information and Knowledge Management*, pages 2365–2368, 2019.

[23] N. Gunasekara, B. Pfahringer, H. Gomes, and A. Bifet. Gradient boosted trees for evolving data streams. *Machine Learning*, 113(5):3325–3352, 2024.

[24] M. Harries. Splice-2 comparative evaluation: Electricity pricing. 1999.

[25] M. Heyden, H. M. Gomes, E. Fouché, B. Pfahringer, and K. Böhm. Leveraging plasticity in incremental decision trees. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 38–54. Springer, 2024.

[26] F. Hinder, V. Vaquet, J. Brinkrolf, and B. Hammer. On the change of decision boundary and loss in learning with concept drift. In *Int. Symposium on Intelligent Data Analysis*, pages 182–194. Springer, 2023.

[27] F. Hinder, V. Vaquet, and B. Hammer. One or two things we know about concept drift—a survey on monitoring in evolving environments. part a: detecting concept drift. *Frontiers in Artificial Intelligence*, 7:1330257, 2024.

[28] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proc. of the seventh ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 97–106, 2001.

[29] I. Khamassi, M. Sayed-Mouchaweh, M. Hammami, and K. Ghédira. Discussion and review on evolving data streams and concept drift adapting. *Evolving systems*, 9(1):1–23, 2018.

[30] I. Koychev. Tracking changing user interests through prior-learning of context. In *International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*, pages 223–232. Springer, 2002.

[31] G. Krempl, I. Žliobaite, D. Brzeziński, E. Hüllermeier, M. Last, V. Lemaire, T. Noack, A. Shaker, S. Sievi, M. Spiliopoulou, et al. Open challenges for data stream mining research. *ACM SIGKDD explorations newsletter*, 16(1):1–10, 2014.

[32] L. I. Kuncheva and I. Žliobaitė. On the window size for classification in changing environments. *Intelligent Data Analysis*, 13(6):861–872, 2009.

[33] A. Liu, J. Lu, F. Liu, and G. Zhang. Accumulating regional density dissimilarity for concept drift detection in data streams. *Pattern Recognition*, 76:256–272, 2018.

[34] V. Losing, B. Hammer, and H. Wersing. Interactive online learning for obstacle classification on a mobile robot. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2015.

[35] V. Losing, B. Hammer, and H. Wersing. Knn classifier with self adjusting memory for heterogeneous concept drift. In *IEEE 16th Int. Conf. on Data Mining (ICDM)*, pages 291–300. IEEE, 2016.

[36] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering*, 31(12):2346–2363, 2018.

[37] P. Lu, J. Lu, A. Liu, and G. Zhang. Early concept drift detection via prediction uncertainty. In *Proc. of the AAAI Conf. on Artificial Intelligence*, volume 39, pages 19124–19132, 2025.

[38] D. Lukats, O. Zielinski, A. Hahn, and F. Stahl. A benchmark and survey of fully unsupervised concept drift detectors on real-world data streams. *Int. Journal of Data Science and Analytics*, 19(1):1–31, 2025.

[39] C. Manapragada, G. I. Webb, and M. Salehi. Extremely fast decision tree. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1953–1962, 2018.

[40] J. Montiel, M. Halford, S. M. Mastelini, G. Bolmier, R. Sourty, R. Vaysse, A. Zouitine, H. M. Gomes, J. Read, T. Abdessalem, et al. River: machine learning for streaming data in python. 2021.

[41] J. Mourtada, S. Gaïffas, and E. Scornet. Amf: Aggregated mondrian forests for online learning. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 83(3):505–533, 2021.

[42] H. Niederreiter. *Random number generation and quasi-Monte Carlo methods*. SIAM, 1992.

[43] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[44] J. Read and I. Žliobaitė. Supervised learning from data streams: An overview and update. *ACM Computing Surveys*, 2025.

[45] D. M. D. Reis, P. Flach, S. Matwin, and G. Batista. Fast unsupervised online drift detection using incremental kolmogorov-smirnov test. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1545–1554, 2016.

[46] J. C. Schlimmer and R. H. Granger Jr. Incremental learning from noisy data. *Machine learning*, 1(3):317–354, 1986.

[47] V. M. Souza. Asphalt pavement classification using smartphone accelerometer and complexity invariant distance. *Engineering Applications of Artificial Intelligence*, 74:198–211, 2018.

[48] V. M. Souza, A. R. Parmezan, F. A. Chowdhury, and A. Mueen. Efficient unsupervised drift detector for fast and high-dimensional data streams. *Knowledge and Information Systems*, 63(6):1497–1527, 2021.

[49] V. M. A. Souza, D. M. d. Reis, A. G. Maletzke, and G. E. A. P. A. Batista. Challenges in benchmarking stream learning algorithms with real-world data. *Data Mining and Knowledge Discovery*, 34:1805–1858, 2020.

[50] V. M. A. Souza, D. F. Silva, J. Gama, and G. E. A. P. A. Batista. Data stream classification guided by clustering on nonstationary environments and extreme verification latency. pages 873–881, 2015.

[51] W. N. Street and Y. Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 377–382, 2001.

[52] L. Ulanova, N. Begum, M. Shokoohi-Yekta, and E. Keogh. Clustering in the face of fast changing streams. In *Proceedings of the 2016 SIAM international conference on data mining*, pages 1–9. SIAM, 2016.

[53] A. Vergara, S. Vembu, T. Ayhan, M. A. Ryan, M. L. Homer, and R. Huerta. Chemical gas sensor drift compensation using classifier ensembles. *Sensors and Actuators B: Chemical*, 166:320–329, 2012.

[54] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[55] G. Widmer and M. Kubat. Learning in the presence of concept drift and hidden contexts. *Machine learning*, 23(1):69–101, 1996.

[56] X. Zhu. Stream data mining repository. *URL:http://www.cse.fau.edu/xqzhu/stream.html*, 2010.

[57] I. Žliobaitė. Combining similarity in time and space for training set formation under concept drift. *Intelligent Data Analysis*, 15(4):589–611, 2011.

[58] I. Zliobaitė. Change with delayed labeling: When is it detectable? In *IEEE int. conf. on Data Mining Workshops (ICDMW 2010)*, page 843 – 850, 2010.

A Appendix

The Appendix contains additional details that were not included in the main paper for reasons such as page limits or because they reduced the readability of the main body of text. The Appendix contains the following: The pseudocode for the most critical components of TNT (Section A.1), an empirical evaluation that shows that the Mutual Information of the entire stream decreases as

the number of distinct drift events increases (Section A.2), additional experimental details that ensure paper reproducibility (Section A.3), a description of the `RotatingMoons` artificial data stream used to parameter tune TNT (Section A.4), additional details and results from the parameter tuning process (Section A.5), additional results (Section A.6), and a postmortem that reflects on the main contributions of this paper (Section A.7).

A.1 TNT Pseudocode

This section includes pseudocode for the most critical components of TNT. They are, the impurity reduction (learning) procedure (Alg. 1) and the time filtering (Alg. 2) processes.

Algorithm 1 Time Neutralising Decision Tree Construction

Require: Features X, Labels y, Time indices t
Require: Hyperparameters: Min samples N_{min}, Min info gain τ, Time penalty λ, MWU alpha α_{mwu}

function BUILDTREE(X, y, t)
  $N \leftarrow |y|$
  if $N < N_{min}$ **or** IsPure(y) **then**
    return LeafNode(mode(y))
  end if
  $\Delta_s, f^*, v^* \leftarrow$ BESTSPATIALSPLIT(X, y)
  $\Delta_t, t_{split} \leftarrow$ BESTTEMPORALSPLIT(t, y)
  if $\Delta_t > (\Delta_s + \lambda)$ **then**
    $X_f, y_f, t_f \leftarrow$ TEMPORALFILTER($X, y, t, \alpha_{mwu}, N_{min}, \tau$) ▷ See Alg. 2
    if $|y_f| < N_{min}$ **or** IsPure(y_f) **then**
      return LeafNode(mode(y_f))
    end if
    $\Delta_s, f^*, v^* \leftarrow$ BESTSPATIALSPLIT(X_f, y_f) ▷ Re-evaluate spatial split
    $X, y, t \leftarrow X_f, y_f, t_f$ ▷ Update reference data for downstream nodes
  end if
  Node $\leftarrow$ SplitNode(f^*, v^*) ▷ Create new split decision node
  Mask$_L \leftarrow X[:, f^*] \leq v^*$
  Node.Left $\leftarrow$ BUILDTREE(X[Mask$_L$], y[Mask$_L$], t[Mask$_L$])
  Node.Right $\leftarrow$ BUILDTREE($X[\neg$Mask$_L$], $y[\neg$Mask$_L$], $t[\neg$Mask$_L$])
  return Node
end function

A.2 Mutual Information decreases as number of distinct drift events increases.

In Section 3.1 we noted that the number of *distinct* concept drift events in some feature subspace $|\mathcal{D}| \to \infty$, the global Mutual Information of that subspace tend to 0.0:

$$\lim_{|\mathcal{D}|\to\infty} I_{\bar{P}}(X;Y) = 0.0 \tag{6}$$

Intuitively, this should make sense. If a *distinct* concept is defined as a concept that is not the same as any previously encountered concept, then as $|\mathcal{D}| \to \infty$, the feature space is reduced to random noise when viewing the entire subspace throughout the entire lifetime of the stream. Random noise by definition contains no Mutual Information. This sections provides empirical evidence in support of this claim. We do this by constructing a 100 synthetic univariate streams for a binary classification task where each class is sampled from a Gaussian distribution with means μ_1 μ_2 resp. with unit variance. Every 50 instances we randomly alter the means of these Gaussians simulating an abrupt drift. We do this for 25 drift events recording the global Mutual Information of the entire stream as well as the average Mutual Information of the each concept after every drift event. Fig. 5 shows the results of this process whereby we can see that while the average Mutual Information of each Concept remains high (blue line), the global Mutual Information of the entire stream tends to 0.0. It is this phenomena that inspired the creation of TNT. By using time as a feature, we can extract

Algorithm 2 TNT Temporal Filtering. For the leaf node constructors, we use python-like notation where t, y min and max refer keyword arguments that set properties of the same name.

Require: Features X, Labels y, Time indices t
Require: Hyperparameters: MWU alpha α_{mwu}, Min samples N_{min}, Min Info Gain τ

function GETTIMELEAVES(t, y)
  if $|y| < N_{min}$ **or** IsPure(y) **then**
    return LeafNode($t = t, y = y, \min = \min(t), \max = \max(t)$)
  end if
  $\Delta, t_{split} \leftarrow$ BESTTEMPORALSPLIT(t, y)
  if $\Delta \leq \tau$ **then** ▷ Create leaf if split is not informative enough.
    return LeafNode($t = t, y = y, \min = \min(t), \max = \max(t)$)
  end if
  $\text{Mask}_L \leftarrow t \leq t_{split}$
  $Leaves_L \leftarrow$ GETTIMELEAVES($t[\text{Mask}_L], y[\text{Mask}_L]$)
  $Leaves_R \leftarrow$ GETTIMELEAVES($t[\neg\text{Mask}_L], y[\neg\text{Mask}_L]$)
  return $Leaves_L \cup Leaves_R$
end function

function TEMPORALFILTER($X, y, t, \alpha_{mwu}, N_{min}, \tau$)
  $Leaves \leftarrow$ GETTIMELEAVES(t, y)
  SORT($Leaves$) ▷ Sort leaves in ascending time order
  $L_{recent} \leftarrow Leaves[-1]$ ▷ Get leaf that represents most recent window of data
  $y_{recent} \leftarrow L_{recent}.y$
  $Mask_{keep} \leftarrow (t \geq L_{recent}.\min)$
  for L_{hist} **in** $Leaves[0, 1, \ldots, -2]$ **do** ▷ -2 refers to the second last item in the list
    $p \leftarrow$ MANNWHITNEYU($y_{recent}, L_{hist}.y$)
    if $p > \alpha_{mwu}$ **then** ▷ Distributions are similar so we can merge the leaves.
      $Mask_{keep} \leftarrow Mask_{keep} \vee (t \geq L_{hist}.\min$ **and** $t \leq L_{hist}.\max)$
    end if
  end for
  return $X[Mask_{keep}], y[Mask_{keep}], t[Mask_{keep}]$
end function

the individual concepts and filter out old ones, increasing the Mutual Information of previously uninformative subspaces. The code to generate Fig. 5 has been included with the TNT source code.

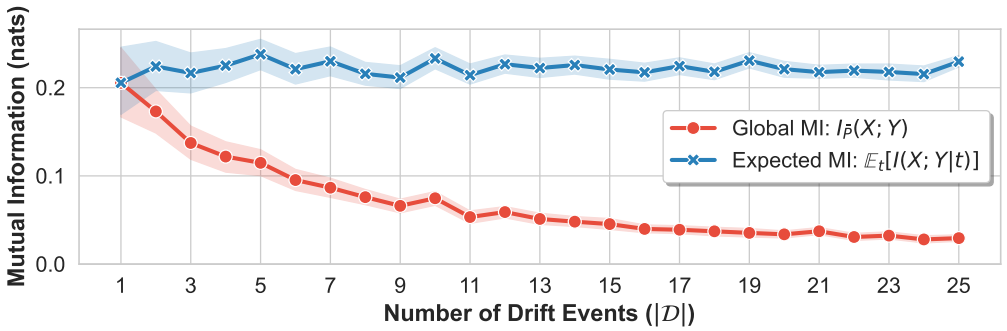


Figure 5: shows how global Mutual Information (MI) (red) in a univariate stream collapses as the number of distinct drift events $|D|$ (K in the main paper) $\to \infty$ while the expected MI of all distinct concepts (blue) remains high. This Fig. shows why active forgetting is useful. By only looking at the most recent concept in a subspace of the stream, that subspace remains informative (i.e. predictive).

A.3 Experimental Details

This Section provides additional experimental details. Firstly, Table 3 includes a summary of the number features, class, and instances for every data stream used in this work. For detailed descriptions on each data stream, we refer the reader to the accompanying citations. As noted in Section 4, we acquired these data streams from the USP DS repository[1] [49]. All streams are stored as CSV files. During preprocessing, we found that the `Asfault`, `Posture` and `SmartMeter` streams were malformed. Further investigation revealed that this was due to a formatting error whereby two samples were stored as a single entry. These malformed entries were simply deleted. Furthermore, the feature 'at3' in the `Luxembourg` stream was revealed to be overly informative (i.e., it perfectly predicts the class label). We did not include it when running experiments. Before running experiments all numerical features in all streams are standardised. Although this step is unnecessary as we only investigated Decision Tree architectures. We do also want to note that the listed features, and instance counts of the downloaded CSVs do not always match the values listed on the USP DS repository website[1]. We do not know why this is the case but Table 3 includes our updated counts. Furthermore, undocumented behaviour in the Kappa statistic calculated by river results in a value 0.0 being returned if only one class exists in both the labels and predictions. This can occur when there is only one class present over some window in a autocorrelated or imbalanced stream. We fixed this post-hoc but setting the Kappa statistic to 1.0 for all windows where an algorithm had a 0.0 Kappa score, but perfect Balanced Accuracy and Accuracy score. This does not meaningfully change the results in any way, but is a more accurate reflection of the performance of the algorithms evaluated.

All code and experiments were run using `Python 3` with the Batch Decision Trees sourced from `scikit-learn` [43] and Stream Learning Decision Trees from `river` [40]. The exception to this is PLASTIC [25] which does not have an implementation in `river` so instead we used the `CapyMOA`. Note that we chose not to use `CapyMOA` for the other Stream Learning Decision Trees because the package is still very much under active development at the time of writing. TNT is implemented in `Python 3`, with time consuming operations written in `Cython`. TNT is `scikit-learn` compatible. All experiments were run using Google Colab or a 2020 MacBook Pro with 2 GHz Quad-Core Intel i5 and 16GB RAM. The average run-time for an experiment is highly algorithm and stream-dependent. For example, TNT run on the `Forest Cover Type` stream took $\sim$50 hours, but it takes less than a minute to run TNT on the `Keystroke` stream. We have included the run time complexity of TNT when trained on the entirety of each stream over 10 replicates in Figure 6.

For parameter selection, we opted to use default parameters for most models as they are informed by decades of research. For the Batch Decision Trees, we set `min_samples_leaf = 10` as the parameter to balance under/overfitting. The logic being that it is half the N_{min} used by TNT as so they approximately equivalent when taking implementation specifics into account. For the Stream Learners, we used the default parameters for all `river` models. For PLASTIC, we set `split_confidence` to 1^{-3} and unbounded the `max_depth` parameter as in the original implementation[2]. We also set the `leaf_prediction` strategy to Majority Class (MC) as the other options did not appear to be working correctly and would frequently return `None` even after PLASTIC trained on several hundred instances.

All experiments run in this work are stored as CSV files, and all figures are generated from these files. The source code contains a summary description of the structure of these CSVs. For statistical testing and Quasi-Monte Carlo Latin Hypercube sampling, we make use of `scipy` [54], and use $p = 0.05$ for all statistical tests performed. For constructing the Critical Difference diagram (Fig. 3 in the main paper), we use `scikit-posthocs`[3]. As noted before, we made use of pseudo-random number generators to ensure reproducibility. In general, a default seed value of 42 was used. For the 10 replicates in the main evaluation, we used the following seeds that were initially themselves randomly generated: [1341, 1967, 2414, 3258, 3485, 3676, 5588, 5616, 7079, 8172].

For some of the larger streams, compute limitations dictated that we downsample the training set for DT_∞ and TNT. They are: `Forest Cover Type` and `Posture` for both models and `Heartbeats` for just DT_∞. When the total stream size became larger than 10 000, we performed stratified sampling

[1] `https://sites.google.com/view/uspdsrepository`

[2] `https://github.com/heymarco/CapyMOA-PLASTIC/blob/plastic/src/capymoa/learner/classifier/plastic.py`

[3] `https://scikit-posthocs.readthedocs.io/en/latest/index.html`

(of 10 000 instances) to preserve class label distributions. This process was also psuedo-random ensuring that it was equivalent across both methods.

Table 3: The USP DS streams [49] evaluated in the work.

Name	Features	Classes	Instances
Asfault (AF) [47]	64	5	8,565
Cancer (CA) [48]	2,352	7	9,256
Chess (CH) [57]	8	3	503
Electricity (EL) [24]	8	2	45,312
Forest Cover Type (FC) [5]	54	7	581,012
Gas Sensor Array (GS) [53]	128	6	13,910
Heartbeats (HB) [52]	280	8	77,904
Insects-Abrupt (IA) [49]	33	6	52,848
Insects-Gradual (IG) [49]	33	6	24,150
Insects-Incremental (II) [49]	33	6	57,018
Insects-Reoccurring (IR) [49]	33	6	79,986
Keystroke (KS) [50]	10	4	1,600
Luxembourg (LX) [57]	30	2	1,901
MIRS (MS) [10]	3,600	2	4,260
NOAA Weather (NW) [13]	8	2	18,159
Outdoor Objects (OD) [34]	21	40	4000
Ozone (OZ) [12]	72	2	2,534
Posture (PO) [45]	3	11	164,859
Powersupply (PS) [56]	2	24	29,928
Rialto (RT) [35]	23	10	82,250
SmartMeter (SM) [48]	96	10	22,949
StarLightCurves (SL) [48]	1024	3	9,236
UWave (UV) [48]	315	8	4,478
Yoga (YG) [10]	426	2	3,300

A.4 Rotating Moons Synthetic Data Stream

In this work we also introduce a new synthetic data stream called `RotatingMoons`. It uses the `make_moons` generator provided by `scikit-learn`[4] which is a two-dimensional non-linear binary classification task. `RotatingMoons` works by accepting a rotation R, and samples per cycle n parameters. By generating n samples from `make_moons`, a rotation is applied to the data:

$$X_\theta = \begin{bmatrix} x_1^1 & x_2^1 \\ x_1^2 & x_2^2 \\ \dots & \dots \\ x_1^n & x_2^n \end{bmatrix} \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \tag{7}$$

where θ is the rotation angle in radians determined by ever increasing multiples of R until a full revolution is complete. To simulate reoccurring concepts, this process can be repeated infinitely. Fig. 7 visualises the `RotatingMoons` stream at different rotations and the code for the generator is included with the Supplementary Materials.

A.5 Parameter Tuning

Recall from Section 4.1 that TNT has 5 tunable parameters: filter factor λ, minimum split node samples N_{min}, minimum time node entropy split τ, confidence threshold for determining time node merges p_{mwu}, and the maximum number of threshold splits to consider per feature thresholds$_{max}$. To avoid data leakage and bias [49], we parameter tune TNT on 5 synthetic streams which cover a variety of drift types. They are: `Hyperplane` [28], `RandomRBFDrift` [40], `SEA` [51], `STAGGER` [46] and a custom stream `RotatingMoons` (A.4). After some initial exploration of the hyper-parameters, we

[4] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html

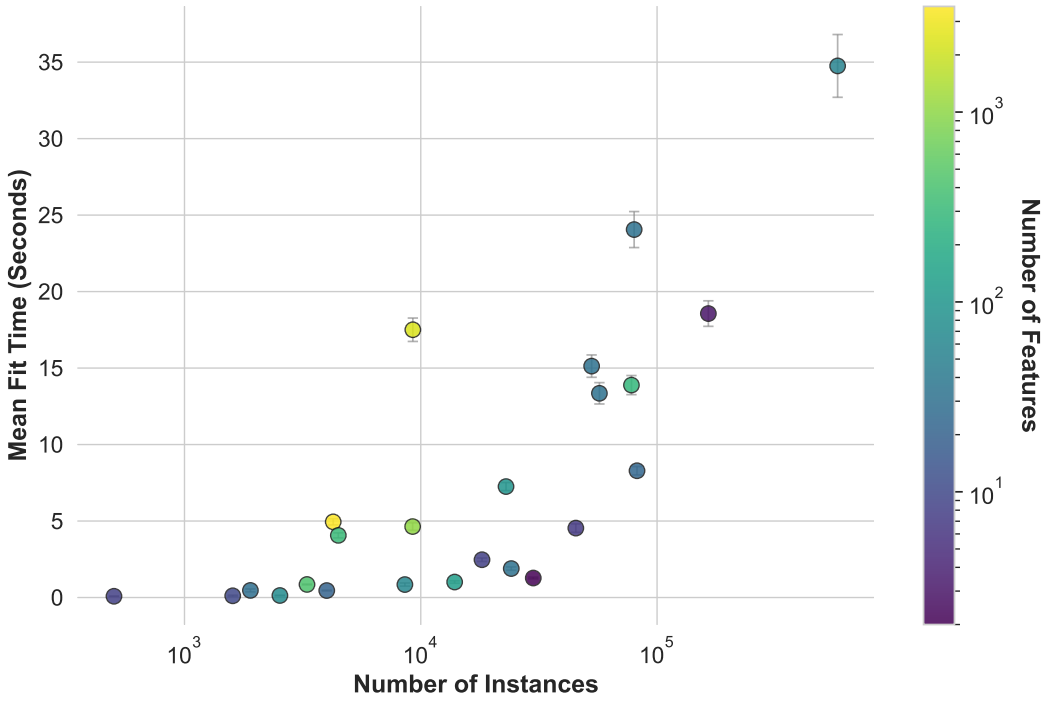


Figure 6: shows the compute time (in seconds) taken for TNT to fit each stream (Table 3) evaluated. Each stream (point) is plotted by number of instances (x-axis) and number of features (colour bar). Error bars indicate the standard deviation over 10 replicates. Note that this plot shows the time it takes to fit the entire stream, and is not indicative of the time taken to evaluate TNT as described in Section 4 where TNT is retrained every 50 instances on an ever growing stream. This plot only serves to show the training compute trends as both the number of instances and features increase.

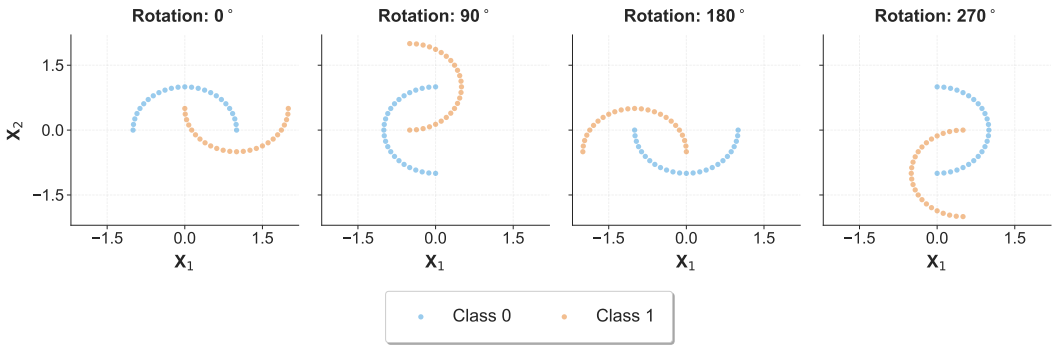


Figure 7: shows a visualisation of the synthetic RotatingMoons data stream using a rotation angle of 90° for one cycle. To simulating reoccurring concepts, the cycle is repeated left → right.

used Quasi-Monte Carlo Latin Hypercube Sampling [42] to generate 100 parameter configurations which we explored across all synthetic streams, recording the final accuracy of the model per configuration per stream. For each stream, we then train a Random Forest [6] regressor using the parameter configurations as input and the final accuracy as a target. From each trained Forest, we extract feature importances (Section A.5.1). We also plot each parameter and obtained accuracy per stream using LOWESS regression to analyse per feature level dependencies (Section A.5.2). Lastly, we also ran a leaf node ablation (Section A.5.3). Table 4 notes the range explored for each parameter configuration, and Table 5 summarises the parameter values used to generate each artificial stream.

Table 4: Hyper-parameters, the range of values explored, and their respective final values used in our evaluation.

Name	Range Explored	Final Value
λ	[0.01, 0.5]	0.20
N_{min}	[10, 50]	20
τ	[0.01, 0.1]	0.05
α_{mwu}	[0.0001, 0.1]	0.05
thresholds$_{max}$	[5, 25]	15
leaf model	{MC, LR, NB}	LR

Table 5: Synthetic generators used in this work, their parameters, and total number of instances (N). Random seed 42 is used for all streams.

Name	Range Explored	Total (N)
Hyperplane [28]	mag_change = 0.2	5000
RandomRBFDrift [40]	change_speed = 1.0	1000
RotatingMoons (A.4)	sample_per_phase = 100 rotation_angle = 90 cycles = 4	1600
SEA [51]	sample_per_variant = 200	800
STAGGER [46]	sample_per_variant = 150	450

A.5.1 Feature Importance

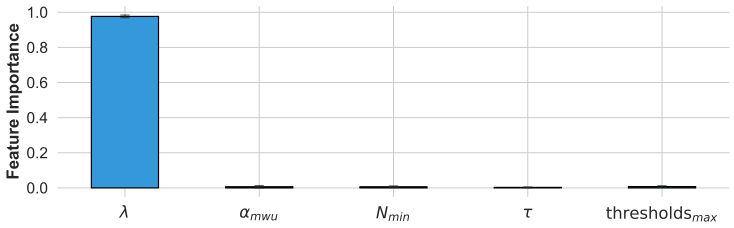
Figure 8 shows the result of the feature importance analysis in the form of mean impurity reduction. A Random Forest regressor is trained using parameter configurations generated from the Latin Hypercube Sampling using the parameter ranges in Table 4 as input features X with the target y being the accuracy that TNT achieved with that parameter configuration. This is done per synthetic stream. Results show that across most synthetic streams investigated, the minimum node split size N_{min} and number of spatial splits considered thresholds$_{max}$ are the most important features. This makes sense as they control how effectively TNT is able to generalise in a given domain. Just like in regular batch decision trees, too much splitting causes a tree to overfit while not enough splitting causes a tree to underfit. Interestingly, `Hyperplane` is the only synthetic stream where the forgetting factor λ is a significant predictor of accuracy. In `Hyperplane`, the decision boundary $P(Y|X)$ is constantly moving while the covariates $P(X)$ remain stable. This is precisely the setting where active forgetting is useful (Section 3.1), and we see that setting a forgetting factor λ as low as possible allows TNT to effectively filter out old irrelevant instances thus adapting to concept drift.

A.5.2 Per Feature Sensitivity Analysis

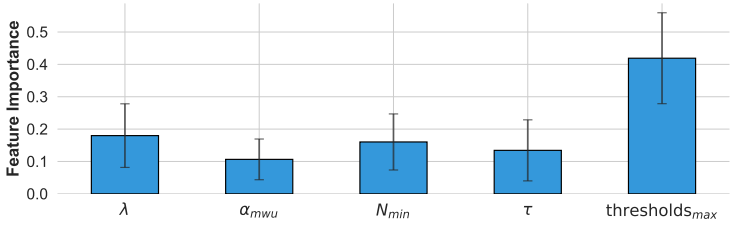
While feature importance extraction allows one to understand the complex interactions between hyper-parameters, we also performed sensitivity analysis on a per-feature level per stream using LOWESS regression. Figure 9 shows the results which reinforces the findings of the feature importance analysis. N_{min} is strongly correlated with the accuracy of TNT on the `SEA` and `STAGGER` streams while thresholds$_{max}$ strongly correlates with accuracy on the `RotatingMoons` stream. Interestingly, this per-feature perspective on the `RandomRBFDrift` stream implies that no single feature overwhelmingly predicts TNT's accuracy. This is echoed in Fig. 8b where all features carry some predictive power. Filter factor λ negatively correlates with accuracy for `Hyperplane` stream for reasons described in the previous Section.

A.5.3 Leaf Node Ablation

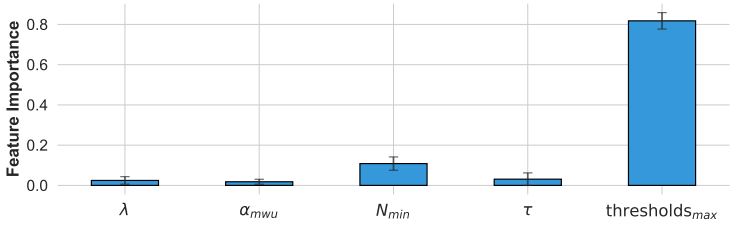
Incremental Decision Trees, such as Hoeffding Trees [28], often place Naive Bayes classifiers on their leaf nodes. They do this to mitigate underfitting while the Hoeffding Tree conservatively learns split decisions over the lifetime of a stream. We investigated if this type of design (replacing leaf impure leaf nodes with lightweight classifiers) would benefit TNT and make it more robust to underfitting.



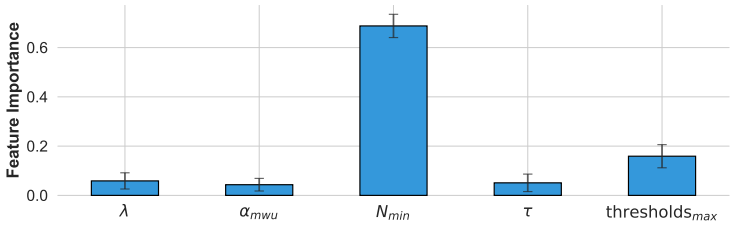
(a) Hyperplane



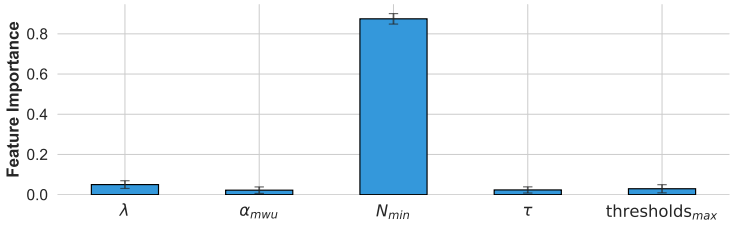
(b) RandomRBFDrift



(c) RotatingMoons



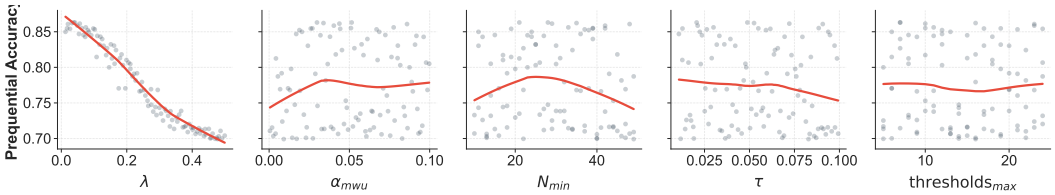
(d) SEA



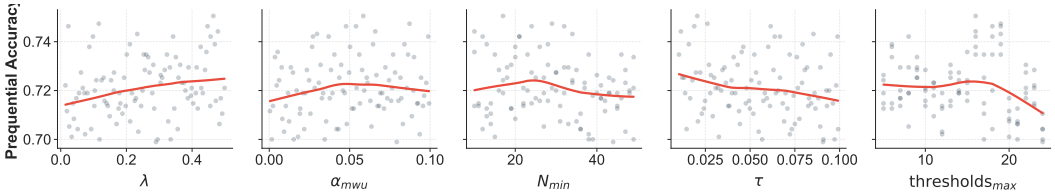
(e) STAGGER

Figure 8: Feature Importances (Mean Decrease in Impurity) of a Random Forest Regressor trained to predict the accuracy of various parameter configurations of TNT per synthetic stream investigated.

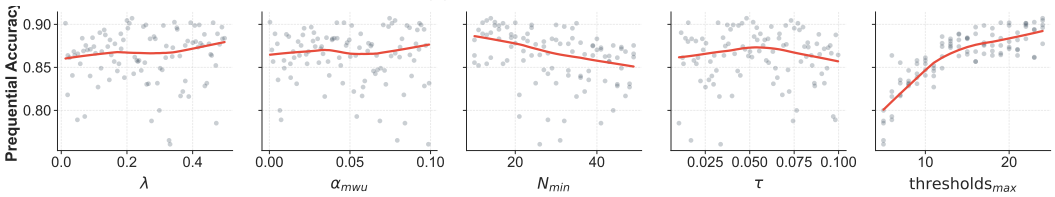
To do this, we performed an ablation study using the same evaluation procedure detailed in Section 4. To avoid data leakage, we evaluated these ablations on the synthetic streams (Table 5) and not on the USP DS repository. We investigated three versions of TNT: (1) a baseline where TNT uses Majority class prediction on its leaf nodes, (2) TNT where impure leaf nodes are replaced by a Logistic Regression (LR) model and (3) TNT where impure leaf nodes are replaced by a Naive Bayes (NB) model. For LR we set a maximum of 200 iterations to reach convergence and a random seed of 42 for every new instantiation to ensure reproducibility. In practice, it may be worth increasing the



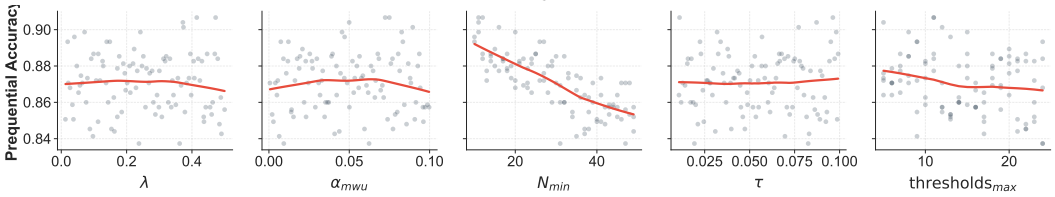
(a) Hyperplane



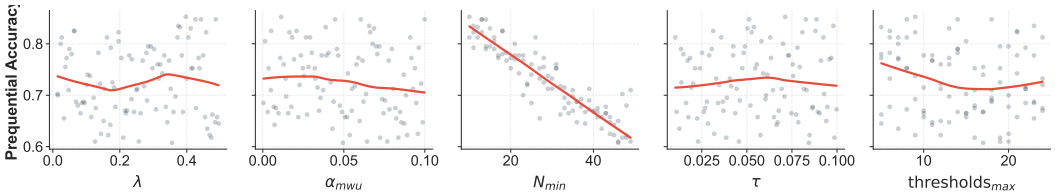
(b) RandomRBFDrift



(c) RotatingMoons



(d) SEA



(e) STAGGER

Figure 9: Per Feature Sensitivity Analysis of each of TNT's hyper-parameters per synthetic stream investigated. Trend lines (red) are determined by LOWESS regression.

maximum iterations and exploring different random states. Results of this evaluation are summarised in Table 6 where there is clear evidence that using a lightweight classifier on impure leaf nodes is beneficial. Overall, there wasn't a significant difference when comparing LR to NB. However, given that average Kappa score of the LR variant was greatest (by a small margin) amongst all synthetic streams investigated, we chose to use it for our main evaluation.

A.6 Additional Results

This section includes additional results that did not make it into the main paper. Fig. 3 shows the Critical Difference diagram of the methods evaluated across all streams and Table 8 lists the ranking per stream for all methods investigated in this work. We used fractional ranking whereby similarly performing models were given the same rank. For example, if the Kappa scores for 4 arbitrary models on some arbitrary stream were: [85, 80, 80, 60], they would be given the ranks [1, 2.5, 2.5, 4] resp.

Table 6: Mean Kappa scores for different variants of TNT. MC uses Majority Class to make predictions at leaf nodes, while LR and NB replace leaf nodes with Logistic Regression and Naive Bayes models resp. Bold values indicate highest average Kappa score.

	MC	LR	NB
Hyperplane	**0.61**	**0.61**	**0.61**
RandomRBFDrift	0.40	**0.44**	**0.44**
RotatingMoons	0.78	0.81	**0.82**
SEA	0.78	**0.81**	0.80
STAGGER	0.44	**0.57**	0.53
Average	0.60	**0.65**	0.64

Table 8 does not present any meaningful results beyond the main table (Table 7), but we have included it for completeness.

We also wanted to show how the forgetting works on an actual data stream, but this very difficult to visualise, and confirm, in high-dimensional spaces. This is because instances that look like they should be excluded when viewing one feature, may be perfectly separable in another feature. Nevertheless, Fig. 10 showcases the instance filtering of TNT on the same synthetic stream we used to motivate the problem in Section 3.1. In Fig. 10, TNT is trained on the entire stream and then we track which instances are excluded during the training process. As shown, TNT excluded the instances in the subspace $X_1 = [25, 50]$ where instances prior to the drift event (dashed line) conflict with new instances post drift event. This confirms that TNT does what we designed it to do: filter out irrelevant instances during training based on the stream's current context, thus enabling TNT to adapt to concept drift.

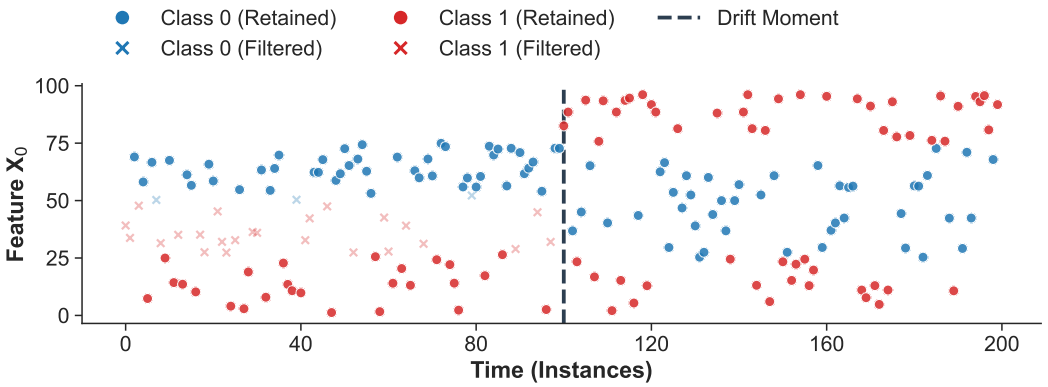


Figure 10: shows how TNT filters instances that are useful to forget. This image was generated by training TNT on a synthetic stream that matches the stream shown in Fig. 1 which we used to motivate the design of TNT. TNT filters out instances (crosses) that it deems no longer relevant to the stream after retraining. This Figure matches the original concept in Fig. 1 with instances in the $X_1 = [25, 50]$ subspace (pre-drift event) being filtered out as they are no longer relevant for making predictions once the drift event (dashed line) has occurred.

A.7 Postmortem

In this final Section, we provide a *postmortem* of the research conducted in this work. It is somewhat informal in nature, and contains insights that we either did not have time to validate, or did not include in the main paper because they were not, in our opinion, significant enough. This Section proceeds as follows: Section A.7.1 discusses the consequences of the parameter tuning process in this work, Section A.7.2 discusses some of the nuances regarding the deployment of Incremental Stream Learners (specifically Decision Trees) and Section A.7.3 reflects on Time Neutralising Trees (TNT) and provides insight into potential future research directions.

Table 7: Cohen's Kappa for each method (column) and data stream (row). Bolded values indicate the statistically best performing method. Underlined values are used to indicate top performing methods when no single method is best. Mean rank per method is reported with a bold value indicating lowest (best) rank.

	Baselines		Sliding Window					Incremental			Drift Detectors					Ours	
	LC	MC	DT_{50}	DT_{100}	DT_{250}	DT_{500}	DT_{∞}	HT	EFDT	PLS	HAT	LAST	HT_{ADWIN}	HT_{DDM}	HT_{D3}	TNT	TNT_{500}
AF	0.19	0	0.51	0.61	0.65	0.68	0.66	0.52	0.59	0.57	0.52	0.67	0.62	0.58	0.41	0.65	0.69
CA	-0.01	0.01	0.07	0.11	0.14	0.17	0.20	0.03	0.03	0.02	0.03	0.09	0.04	0.03	0.02	0.18	0.17
CH*	0.14	0.09	0.55	0.55	0.55	0.57	0.57	0.38	0.48	0.38	0.36	0.26	0.38	0.38	0.44	0.54	0.54
EL	0.63	0	0.55	0.54	0.51	0.51	0.41	0.48	0.54	0.46	0.58	0.51	0.58	0.61	0.61	0.46	0.52
FC	**0.84**	0.13	0.18	0.18	0.28	0.53	0.27	0.42	0.48	0.48	0.45	0.55	0.40	0.57	0.48	0.27	0.56
GS	0.18	0.07	0.66	0.67	0.67	0.65	0.69	0.35	0.63	0.56	0.35	0.54	0.64	0.69	**0.76**	0.75	0.69
HB	0.75	0.75	0.82	0.83	0.85	0.86	**0.88**	0.53	0.70	0.86	0.74	0.84	0.81	0.82	0.80	0.85	0.82
IA	0.04	-0.01	0.42	0.49	0.54	0.55	0.42	0.37	0.42	0.46	0.47	0.39	0.51	0.45	0.46	0.48	**0.58**
IG	0.11	-0.01	0.47	0.53	0.57	0.59	0.42	0.37	0.39	0.41	0.39	0.37	0.42	0.42	0.48	0.55	**0.61**
II	-0.02	-0.06	0.25	0.36	0.42	0.45	0.41	0.41	0.42	0.47	0.44	0.43	0.47	0.45	0.46	0.45	0.46
IR	0.09	0	0.45	0.51	0.55	0.56	0.28	0.30	0.36	0.38	0.38	0.33	0.47	0.39	0.43	0.43	**0.60**
KS	-0.25	-0.24	0.74	0.79	0.82	0.83	0.83	0.61	0.66	0.35	0.61	0.59	0.61	0.65	0.79	0.86	**0.88**
LX*	-0.05	0	0.11	0.17	0.20	0.24	0.22	0.12	0.13	0.13	0.13	0.12	0.12	0.12	0.13	0.22	0.23
MS	-0.03	0	0.42	0.46	0.49	0.50	0.53	0.23	0.43	0.33	0.31	0.33	0.26	0.22	0.32	0.56	0.54
NW	0.19	0	0.18	0.21	0.26	0.33	**0.36**	0.28	0.30	0.23	0.29	0.27	0.15	0.26	0.19	0.35	0.32
OD	**0.88**	0	0.05	0.08	0.16	0.26	0.42	0.57	0.55	0.12	0.57	0.57	0.71	0.64	0.80	0.47	0.29
OZ	0.39	0.33	0.31	0.28	0.25	0.26	0.30	0.33	0.33	0.33	0.32	0.33	0.34	0.36	0.34	0.25	0.21
PO	-0.02	0	0.15	0.22	0.30	0.35	**0.41**	0.33	0.40	0.30	0.35	0.35	0.26	0.33	0.06	0.40	0.38
PS	-0.04	-0.01	0.06	0.12	0.16	0.16	0.09	0.13	0.13	0.10	0.12	0.13	0.12	0.13	0.09	0.14	0.16
RT	-0.11	0	0.29	0.62	0.61	0.57	0.71	0.26	0.51	0.44	0.28	0.51	0.47	0.37	0.56	**0.75**	0.65
SM	-0.10	-0.10	0.20	0.42	0.50	0.53	0.54	0.46	0.38	0.31	0.46	0.30	0.54	0.48	**0.61**	0.55	0.57
SL	-0.01	0	0.61	0.68	0.73	0.76	0.82	0.56	0.71	0.67	0.59	0.50	0.66	0.64	0.67	**0.84**	0.80
UW	-0.02	-0.02	0.22	0.34	0.40	0.43	0.35	0.21	0.31	0.10	0.21	0.34	0.45	0.47	**0.49**	0.41	0.47
YG	-0.03	0	0.12	0.26	0.36	0.47	0.49	0.07	0.07	0.01	0.06	0.12	0.09	0.07	0.07	**0.59**	0.55
Rank	13.67	15.71	10.85	8.5	6.93	5.21	7.02	11.56	9.19	10.31	9.81	9.42	8.69	8.22	7.90	5.21	**4.79**

* Dataset not large enough to statistically distinguish performance over entire stream.

Table 8: Ranks for each classifier (column) and data stream investigated (row). Ranks were determined using fractional ranking where methods that are not significantly different according to a Mann-Whitney U test ($p = 0.05$) are given the same rank.

	Baselines		Sliding Window					Incremental			Drift Detectors					Ours	
	LC	MC	DT_{50}	DT_{100}	DT_{250}	DT_{500}	DT_{∞}	HT	EFDT	PLS	HAT	LAST	HT_{ADWIN}	HT_{DDM}	HT_{D3}	TNT	TNT_{500}
AF	16	17	14	10.5	4	4	4	10.5	10.5	10.5	10.5	4	4	10.5	15	4	4
CA	17	16	7	7	5	3.5	1.5	12	12	12	12	7	12	12	12	1.5	3.5
CH	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
EL	2	17	5	7.5	10.5	10.5	16	14	7.5	14	5	10.5	5	2	2	14	10.5
FC	1	17	15	16	12	4.5	14	10	7.5	6	9	4.5	11	2	7.5	13	3
GS	15.5	15.5	7.5	7.5	7.5	7.5	3	15.5	10.5	12.5	15.5	12.5	10.5	5	1	2	4
HB	13	13	7	7	3	3	1	17	16	3	13	7	13	7	13	7	10
IA	16	17	11	5	3	2	13	15	12	8	8	14	4	10	8	6	1
IG	16.5	16.5	11	5	3	2	11	11	11	11	11	11	11	11	6	4	1
II	16	17	15	14	11	6	12	13	9.5	1.5	8	9.5	1.5	6	3	6	4
IR	16	17	6	4	3	2	15	14	12	10	10	13	5	10	7.5	7.5	1
KS	16.5	16.5	8	6.5	4	4	4	12.5	9.5	15	12.5	12.5	12.5	9.5	6.5	2	1
LX	16.5	16.5	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
MS	16.5	16.5	7.5	3.5	3.5	3.5	3.5	14.5	7.5	10.5	10.5	10.5	13	14.5	10.5	3.5	3.5
NW	14	17	14	11.5	8.5	3	1	8.5	5	11.5	5	8.5	16	8.5	14	2	5
OD	1	17	16	15	13.5	12	10	6	8	13.5	6	6	3	4	2	9	11
OZ	8	8	8	8	16.5	8	8	8	8	8	8	8	8	8	8	8	16.5
PO	17	16	14	13	11	6	1	8	3	10	6	6	12	9	15	2	4
PS	17	16	15	7.5	2	2	12.5	7.5	7.5	12.5	7.5	7.5	11	7.5	14	4	2
RT	17	16	13	4	5	6	2	15	8.5	11	14	8.5	10	12	7	1	3
SM	17	16	15	11	7	6	4	9.5	12	13	9.5	14	5	8	1	3	2
SL	16.5	16.5	12	8	5	4	2	14.5	6	8	13	14.5	10.5	10.5	8	1	3
UW	16.5	16.5	12	9.5	6.5	5	9.5	14	9.5	14	14	9.5	3	3	1	6.5	3
YG	16.5	16.5	10.5	6	5	3.5	3.5	10.5	10.5	15	10.5	10.5	10.5	10.5	10.5	1	2

A.7.1 Synthetic Streams Parameter Tuning and Active Forgetting

In this work, we used synthetic data streams to parameter tune TNT to avoid data leakage and bias which are known problems in Stream Learning research [49]. The streams we used are discussed in Section A.5. We note that using synthetic streams has received criticism recently [49] for being unrealistic, which this section reflects on. Based on the results of parameter tuning TNT using the synthetic streams, we settled on a series of hyper-parameters that were "desirable". However, it is worth considering if those values were "desirable" on the USP DS repository. Our intuition tells use that $\lambda = 0.2$ is likely too conservative for streams where active forgetting is necessary. This is quite clear on the `Hyperplane` stream (Fig. 9a) where the optimal λ was 0.0. That begs the question, when is active forgetting useful? An informal investigation reveals that on many of the data streams we evaluated, active forgetting was only contextually, and typically transiently useful. For example, immediately after an abrupt drift, active forgetting is useful, but, as the stream progressed, it was less useful as TNT (and the other Batch DTs) were simply able to separate the change in distributions. This is evident by the high mean ranking of DT_∞ which, to us, implies that simply training on all of the data is a very viable strategy in many stream settings. More specifically, that may imply that the "concept drift" in many streams does not result in conflicting feature subspaces along time (i.e., where active forgetting is useful).

Conversely, it is clear from our results that passive forgetting (i.e., sliding windows) is useful. This is shown by the fact that TNT_{500}, DT_{500}, and DT_{250} ranked 1st, (tied) 2nd, and 4th resp. It is also a core assumption of data stream learning that recent data is more relevant to future predictions than past data. That begs another question, why? Or more specifically, why was the time filtering active forgetting of TNT not enough? We have already mentioned that λ may have been tuned too conservatively, but another theory is that near-optimal active forgetting is not achievable through observations of the training instances alone. In other words, simply looking at historical instances of a stream is not enough to determine which of those instances is useful to train on. Instead, understanding how the streams drifts / evolves over time would solve this problem. Any further speculation is beyond the scope of this paper, and is fundamentally antithetical to the reactive nature of Stream Learning. In fact, it is more inline with the proactive approaches proposed in Drift Modelling. Nevertheless, we believe this line of thought may be useful in future work.

A.7.2 Incremental Stream Learners

This Section reflects on the use of traditional Incremental Stream Learners (specifically Hoeffding Trees and their variants) (HTs) in the modern age. Our results, combined with the results of Gower-Winter et al. [21], paint a very convincing picture that incremental decisions trees are insufficient in the modern era where both compute and memory is more readily available. We are not saying that these methods are useless, but rather that their usage should be reserved for the settings upon which they were developed (e.g., low resource Edge Computing). Futhermore, we also want to take note of the settings where HTs perfromed well. These were the `Electricity`, `Forest Cover Type` and `Outdoor Objects` streams. Interestingly, these also happen to be the streams with the highest label autocorrelation (evident by the Kappa score of the Last Class (LC) classifier). Furthermore, `Electricity`, `Forest Cover Type` and `Outdoor Objects` also happen to be some of the oldest data streams, and we can't help but think that data leakage (or at least some bias) is present in the design of these HT architectures. It is also worth noting: In this work, HT, and its variants, train immediately after they make a prediction on an instance. This is obviously unrealistic in many settings as labels do not arrive instantly, but it is standard practice. It is almost certain that if we restricted the training periods of the incremental DTs to follow that of the Batch DTs (training only every 50 instances), their performance would drop, further exacerbating the divide between the Batch and Stream learners [21].

To prove this point, we ran additional experiments on the aforementioned streams: `Electricity` (EL), `Forest Cover Type` (FC) and `Outdoor Objects` (OD) using the best supervised detector-based stream learning approach from our main evaluation: DDM. In these experiments, DDM was only able to update every 50 instances training on the last 50 instances in the stream, thus matching the updating cadence of the batch DTs. Table 9 reports the results which clearly shows the benefit of updating after every instance[5]. In fact, the performance drop is so extreme in all cases, that DDM would have ranked worse than most of the batch learning methods on all three streams evaluated.

[5]These are the non-posthoc adjusted Kappa Scores caused by the river bug which affected the main results in Table 7.

These results further support our claim that incremental Decision Trees are simply inferior to their Batch Learning counterparts. While we cannot say for certain, this performance drop is likely for all other stream learning methods investigated [21].

Table 9: Average Kappa Score of Hoeffding Tree (with DDM) trained incrementally (test-then-train) and trained in a batch fashion (which simulates label delay, and mimics the training regime of the batch Decision Trees investigated in this work). Results show clear the benefit of training immediately after making a prediction on an instance (i.e., with no label delay).

	EL	**FC**	**OD**
DDM	0.61	0.42	0.63
DDM (Batched)	0.45	0.07	0.19

A.7.3 Time Neutralising Trees (TNT)

This Section reflects on our implementation of TNT. Overall, we are happy with the results of TNT on the USP DS repository. Both variants of TNT ranked well overall: 1st for TNT_{500} and 2nd (tied) for the base variant (TNT). We believe our approach stands as the new best-in-class for single Decision Trees in stream learning settings with concept drift. That being said, we must acknowledge that the current state-of-the-art stream learners are all tree ensemble approaches such as ARF [19], AMF [41], SRP [20] and SGBT [23]. This presents us with a natural next step in the development of concept drift robust batch learners (i.e., An ensemble approach for TNT). We did explore this idea initially. However, ensemble stream learning approaches (e.g., ARF) adopt a dynamic approach in the way the partition the feature space for each base tree. More specifically, there is kind of evolutionary survival of the fittest game being played implicitly whereby poorly performing base trees are replaced until a stable variant emerges. Achieving a batch learning equivalent of this dynamic is far beyond the scope of this paper, but is something that we will explicitly tackle in the future.

Lastly, we do want to acknowledge that the current implementation of TNT has a potential fail case. This has to do with the way TNT merges leaf nodes during the filtering process. In this version of TNT, we perform a Mann-Whitney U test on the label distributions of each leaf node using the right most leaf node (i.e., the most recent samples in the stream) as reference. If they are sufficiently similar, then the nodes are "merged" (i.e., instances from the older leaf node are not filtered out). This approach worked well in this work, but in a setting where a subspace being evaluated for filtering consists of instances with the same label distribution P(Y), but different posterior distribution P(Y|X), leaf nodes will still be merged even if they shouldn't be. A simple example of this is a binary "class flip" whereby, for some feature subspace: P(X) and P(Y) remain the same, but P(Y|X) flips meaning that instances that were "positive" are now "negative" and vice-versa. A more robust solution to this limitation would be to train a lightweight model on the instances in the right most leaf node of the filter subtree, and only merge other leaf nodes if the model achieves a metric score > some threshold. This would, however, come with additional compute cost. Nevertheless, we reiterate that this version of TNT is capable of achieving state-of-art results.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Our main research goals were: (1) Stream learning should move towards batch style training regimes and (2) our proposed method TNT is a state-of-the-art stream classification method. With our extensive experimentation, we believe the contents of the paper accurately reflect the claims made in the abstract and introduction.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We are quite honest that the parameter tuning process resulted in hyper-parameters that were likely too conservative. We acknowledge a potential failure case (Section A.7.3) for TNT, and repeatedly state that our results are only relevant in settings where storing instances for a period of time is viable. Moreover, we also clearly state that TNT is a state-of-the-art single decision method, and that more work needs to be done in order to transfer the time filtering process to ensemble architectures.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best

judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

   Justification: From our perspective, Section 3.1 provides theoretical results, but they are largely based on extensive theoretical work already done in Information Theory. We simply cite the relevant sources. We do also include empirical evidence that the theoretical results are in fact valid (Section A.2).

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Besides the fact that we have included the source code, Sections 4 and A.3 cover, in great detail, the information needed to reproduce this work. Section 3.2 also describes our method and we include pseudocode for the relevant functions.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.

(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We have included the source code as supplementary material, and cite all relevant datasets.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Not only do we state the hyper-parameters selected (Section 4.1), we also include the range of parameters explored for TNT (Section A.5), and state the hyper-parameters for all other methods used in our evaluation (Section A.3).

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: We performed statistical tests for comparing all methods in our evaluation (Section 4). Error bars are included but also context dependent. For example, our main results (Table 7) do not include standard deviations because it would make the table unreadable, so we instead show significance through bolding the statistically best method(s). Other results include error bars where the add to the interpretability of said results (e.g., Fig. 5).

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Section A.3 lists the compute resources used. We do provide the compute times for TNT in the form of a performance analysis, but we did not keep track of the overall runtime of the entire experimental process. Primarily due to the fact that most experiments were run on a laptop that was also being used for day-to-day tasks, resulting in fluctuating compute resources that would have made any compute time reporting inconsistent and unreflective of the compute times had they been done on a dedicated machine.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Justification for this section is unclear because it is entirely personal opinion. We read the Code of Ethics, we believe we have conformed to them.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: It can be argued that every new Machine Learning algorithm carries with it the potential to do harm. Beyond this, TNT does not explicitly provide any positive and/or negative societal impacts.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: See previous question. We also note that all real-world data streams investigated come from a repository that is publicly accessible to everyone [49].

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.

- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All relevant software packages, algorithms and datasets are explicitly cited and acknowledged (Section A.3). Any version relevant dependencies can be seen in provided source code.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: Code comments and usage instructions for TNT are provided with the source code.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: No human subjects in this work.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.

- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: See previous question.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: We wish to be transparent that Gemini 3.1 Pro was used to create style sheets for the plots included in this work (i.e., Matplotlib configurations / setups that looked research paper-quality), to optimise boiler plate code that was slowing down evaluations and to convert some sketch equations into LaTeXcompliant equations. There was also implicit use of code autocomplete provided by the IDE used (PyCharm). Any and all output produced by LLMs was reviewed by a human. No data, core functional code, or any written text in this paper was generated using LLMs

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.