# Forgetful Forests: high performance learning data structures for streaming data under concept drift

Zhehu Yuan
New York University
New York, New York
zy2262@nyu.edu

Yinqi Sun
New York University
New York, New York
ys3540@nyu.edu

Dennis Shasha
New York University
New York, New York
shasha@cims.nyu.edu

## ABSTRACT

Database research can help machine learning performance in many ways. One way is to design better data structures. This paper combines the use of incremental computation and sequential and probabilistic filtering to enable "forgetful" tree-based learning algorithms to cope with concept drift data (i.e., data whose function from input to classification changes over time).

The forgetful algorithms described in this paper achieve high time performance while maintaining high quality predictions on streaming data. Specifically, the algorithms are up to 24 times faster than state-of-the-art incremental algorithms with at most a 2% loss of accuracy, or at least twice faster without any loss of accuracy. This makes such structures suitable for high volume streaming applications.

**PVLDB Reference Format:**
Zhehu Yuan, Yinqi Sun, and Dennis Shasha. Forgetful Forests: high performance learning data structures for streaming data under concept drift. PVLDB, 16(X): XXX-XXX, 2023.
doi:XX.XX/XXX.XX

## 1 INTRODUCTION

Supervised machine learning[3] tasks start with a set of labeled data. Researchers partition that data into training data and test data. They train their favorite algorithms on the training data and then derive accuracy results on the test data. The hope is that these results will hold on to yet-to-be-seen data because the mapping between input data and output label (for classification tasks) doesn't change.

This paradigm works well for applications such as medical research. In such settings, if a given set of lab results $L$ indicate a certain diagnosis $d$ at time $t$, then that same set of input measurements $L$ will suggest diagnosis $d$ at a new time $t'$.

However, there are many applications where the function between input and output label changes: inflation rates, variants of epidemics, and market forecasting. In such applications, the mapping from input to labeled output changes over time. This requires more work, but also presents a time performance opportunity because a learning system can judiciously "forget" (i.e. discard) old data and learn a new input-output function on only the relevant data and do so quickly. In addition to discarding data cleverly, such a system can take advantage of the properties of the data structures to speed up their maintenance.

These intuitions form the basic strategy of the forgetful data structures we describe here.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. X ISSN 2150-8097.
doi:XX.XX/XXX.XX

## 2 BACKGROUND

The training process of many machine learning models take a set of training samples of the form $(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)$ where in each training sample $(\boldsymbol{x}_i, y_i) \in (\boldsymbol{X}_{train}, \boldsymbol{Y}_{train})$, $\boldsymbol{x}_i$ is a vector of feature-values and $y_i$ is a class label[12, 19]. The goal is to learn a function from the $X$ values to the $y$ values. In the case when the mapping between $X$ and $y$ can change, an incremental algorithm will update the mapping as data arrives. Specifically, after receiving the k-th batch of training data, the **parameters** of the model $f$ **changes** to reflect that batch. At the end of the n-th training batch, the model $f_n$ can give a prediction of the following data point such that $\hat{y_{n+1}} = f_n(x_{n+1})$. This method of continuously updating the model on top of the previous model is called incremental learning.[18] [4]

Conventional decision tree methods, like CART [13], are not incremental. Instead, they learn a tree from an initial set of training data. Under the assumption that data is independent and identically distributed (i.i.d.), conventional decision tree methods form the tree once and for all. A naive incremental approach (needed when the data is not i.i.d.) would be to rebuild the tree from scratch periodically. But rebuilding the decision tree can be expensive. Alternative methods such as VFDT[21] or iSOUP-Tree [16] incrementally update the decision tree with the primary goal of reducing memory consumption.

### 2.1 Hoeffding Tree

In the Hoeffding Tree (or VFDT)[5], each node considers only a fixed subset of the training set, designated by a parameter $n$, and uses that data to choose the splitting attribute and value of that node. In this way, once a node has been fitted on $n$ data points, it won't be updated anymore. The number of data points $n$ considered by each node is calculated using the Hoeffding bound [8], $n = \frac{R^2 ln(1/\delta)}{2\epsilon^2}$, where $R$ is the range of the variable, $\delta$ is the confidence fraction which is set by user, and $\epsilon = \hat{G}(x_1) - \hat{G}(x_2)$ is the distance between the best splitter $x_1$ and the second best splitter $x_2$ based on the $\hat{G}$ function. For example, after a node has received $n$ training data points, we might have $\hat{G}(x_1) = 0.2$ and $\hat{G}(x_2) = 0.1$, where $\hat{G}()$ (e.g., information gain) is the heuristic measure used to choose spitting attributes, and $x_1$ and $x_2$ are the best and second best splitting

attributes for the $n$ training data points. Then for some fraction $\delta$, there will be a $1-\delta$ chance that $G(x_1) - G(x_2) > 0.1 - \epsilon$.

## 2.2 Adaptive Hoeffding Tree

The Adaptive Hoeffding Tree [1] will hold a variable-length window $W$ of recently seen data. We will have $1-\delta$ confidence that the splitting attribute has changed if the two windows are "large enough", and their heuristic measurement are "distinct enough". To define "large enough" and "distinct enough", the Adaptive Hoeffing tree uses the Hoeffding bound:, when $|\hat{G}(W) - \hat{G}(x_1)|$ is larger than $2 * \epsilon$. In scikit-multiflow's implementation, a bootstrap sampling strategy is applied to improve the algorithm's time performance.

## 2.3 iSOUP-Tree

In contrast to the Hoeffding Tree, the iSOUP-Tree [16] uses the FIMT-DD method [10]. That works as follows. There are two learners at each leaf to make predictions. One learner is a linear function $y = wx + b$ used to predict the result, where $w$ and $b$ are variables trained with the data and the results that have already arrived at this leaf, $y$ is the prediction result, and $X$ is the input data. The other learner computes the average value of the $y$ from the training data seen so far. The learner with the lower absolute error will be used to make predictions. Different leaves in the same tree may choose different learners.

## 2.4 Adaptive Random Forest

The Adaptive Random Forest [7] is a random forest, each of whose trees is a Hoeffding Tree without early pruning. Whenever a new node is created, a random subset of features with a certain size is selected. Split attempts are limited to these features for the given node. The number of features in each subset is a hyperparameter that was set before training and will not change any more. The data is randomly selected with replacement from the training set, while the chance for each data item to be selected is determined by a binomial distribution. To detect concept drift, it uses the the method of the Adaptive Hoeffding Tree as described above. Further, each tree has two confidence interval levels to reflect its poor performance in the face of concept drift. When the lower confidence level of a tree is reached (meaning the tree has not been performing well), the Random Forest will create a new background tree that trains like any other tree in the forest, but the background tree will not contribute to the prediction. If the tree already has one background tree, the older one will be replaced. When the higher confidence level of a tree is reached (meaning the tree has been performing very badly), the Random Forest will delete that tree and replace it with the corresponding background tree.

# 3 FORGETFUL DATA STRUCTURES

This paper introduces both a Forgetful Decision Tree and a Forgetful Random Forest. They sequentially and probabilistically forget old data and combine the retained old data with new data to track datasets that may undergo concept drift. In the process, the values of several hyperparameters are adjusted depending on the relative accuracy of the data structure at hand (whether a decision tree or a random forest). We present the overall algorithms in subsections 4.4 and 4.5 but first we present the subroutines, each of which takes several input parameters.

The **accuracy** in this paper is measured based on the confusion matrix: the proportion of true positive plus true negative instances relative to all test samples:

$$\frac{|TruePositive\ in\ testset| + |TrueNegative\ in\ testset|}{Size(testset)}$$

## 3.1 Forgetful decision trees

When new data is acquired from the data stream, function $UpdateSubTree(.)$ (Algorithm 2) will be called on the root node to incrementally and recursively update the entire decision tree.

- The stopping criteria $E(\cdot)$ may comprise maximum tree height, minimum samples to split, or minimum impurity decrease. If it returns $True$, then the node is not further split. Also, if all data in some node have the same label, that node will not be further split.

**Algorithm 1:** BuildSubTree

**input :** This is called when a subtree must be rebuilt from scratch.
$\boldsymbol{X}, \boldsymbol{Y}$, incoming training data batch for current node
$E(\cdot)$, the stopping criteria, returns boolean
$G(\cdot)$, a function to score the fitness of a feature for splitting.

**global :** $current.X, current.Y$, the data retained
$current.gRecord$, the sorted data retained
$current.ranges$, the ranges for splitting retained
$current.label$, the prediction label

```
1  begin
2      current.X, current.Y ← X, Y;
3      if E(current, current.X, current.Y) is True then
4          current.label ← the class having the greatest
             probability;
5      else
6          current.gRecord, ranges ← minG(X, Y);
7          // find the best value ranges
8          Xs, Ys ← Split (current.X, current.Y) at rangess;
9          i=0;
10         for each x, y in Xs, Ys do
11             child ←
                 BuildSubTree(x, y, current.children[i], E(.), G(.));
12             i++;
13         end
14         current.ranges ← ranges;
15     end
16 end
```

- The evaluation function $G(\cdot)$ evaluates the score of each feature and each range for splitting. It will typically be a Gini Impurity[6] score or an entropy reduction coefficient. As we discuss below, the functions *minG* and *minGInc* find

split points that minimize the weighted sum of scores of each subset of the data after splitting. Thus, the score must be evaluated on many split points (e.g., if the input attribute is *age*, then possible splitting criteria could be $age > 30$, $age > 32$, ...) to find the optimal ranges. Those evaluations run much faster if the data is sorted for each input attribute.

- $currentParams.rSize$ governs the size of $current.X + X$ and $current.Y + Y$ to be retained, when new data for $X$ and $Y$ arrives. For example, suppose $currentParams.rSize = 100$. Then $size(X) + size(current.X) - 100$ of the oldest of $current.X$ and $current.Y$ (the data present before the incoming batch) will be discarded. The algorithm then appends $X$ and $Y$ to what remains of $current.X$ and $current.Y$. All nodes in the subtrees will discard the same data items as the root node. In this way, the tree is trained with only the newest $currentParams.rSize$ of data in the tree. Discarding old data will help overcome concept drift, because the newer data better reflects the mapping from $X$ to $Y$ after concept drift. $currentParams.rSize$ should never be less than $size(X)$, because we don't want to forget any new incoming data. $currentParams.rSize$ is computed in algorithm 3 below and will will be explained in subsection 3.2.

After the retained old data is concatenated with the incoming batch data, the decision tree is updated in a top-down fashion using the $UpdateSubTree(.)$ function based on $G(.)$.

In the $BuildSubTree(.)$ function, $minG(\cdot)$ will sort the data at the current node by all its features and store the order in the current node. By contrast, in the Incremental Update ($UpdateSubTree(.)$ function), $minGInc(\cdot)$ will sort only the incoming data (from $X$ and $Y$) and merge it with the previous data that is already sorted. Merging is of course faster than sorting, which conveys a time advantage to $UpdateSubTree(.)$ compared with $BuildSubTree(.)$.

At every interior node, function $minGInc(.)$ calculates a score for every feature by evaluating function $G$ on the data allocated to the current node. This calculation leads to the identification of the best feature and best value (or potentially values) to split on, while the splitting gives rise to two or more ranges for a feature. The data discarded in line 2 of (Algorithm 2) will not be considered by $minGInc(.)$. If, at some node, the best splitting value (or values) is different from the choice before the arrival of the new data, the algorithm rebuilds the subtree with the data retained in *current* as well as the new data allocated to this node (the $BuildSubTree(.)$ function described in (Algorithm 1). Otherwise, if the new best value range is the same as the range from the old tree, the algorithm splits only the incoming data among the children and then recursively calls the $UpdateSubTree(.)$ function on these children nodes.

In summary, the forgetting strategy ensures that the model is trained only on the newest $currentParams.rSize$ data. The rebuilding strategy determines whether a split point can be retained in which case tree reconstruction is vastly accelerated. Even if not, the calculation of the split point based on $G(.)$ (e.g. Gini score) is somewhat accelerated because the relevant data is already nearly sorted.

## 3.2 Adaptive Calculation of Retain Size and Max Tree Height

Retaining more historical data (larger $currentParams.rSize$) will result in higher accuracy when there is no concept drift, because the old information is useful. When concept drift occurs, $currentParams.rSize$ should be small, because old information won't reflect the new concept (which is some new mapping from input to label). Of course, a smaller $currentParams.rSize$ will result in increased speed. Thus, changing $currentParams.rSize$ can improve accuracy and reduce time. We use the following rules:

- When accuracy increases (i.e., the more recent predictions have been more accurate than previous ones) a lot, the model can make good use of more data, we want $currentParams.rSize$ to increase with the effect that we discard little or no data. When the accuracy increase is mild, the model has perhaps achieved an accuracy plateau, so we increase $currentParams.rSize$, but only slightly.
- When accuracy decreases, we want to decrease $currentParams.rSize$ to forget old data, because this indicates that concept drift has happened. When accuracy decreases a lot, the new data may follow completely different rules from the old data, so we want to forget most of the old data, suggesting $currentParams.rSize$ should be very small. When concept drift is mild and accuracy decreases only a little, we want to retain more old data, so $currentParams.rSize$ should decrease only a little.
- When accuracy changes little or not at all, we allow $currentParams.rSize$ to slowly increase.

To achieve the above requirements, we use the $AdaptParameters(.)$ function (Algorithm 3). That function performs adaptively changes $currentParams.rSize$, $maxHeight$, and $currentParams.iRate$ based on the change in accuracy. It is called when new data is acquired from the data stream and before function $UpdateSubTree(.)$ [Algorithm 2] is called. The $currentParams.rSize$ and $maxHeight$ will be applied to the parameters when calling $UpdateSubTree(.)$. The $currentParams.iRate$ and $currentParams.rSize$ will also be inputs to the next call to the $AdaptParameters(.)$ function on this tree.

The $AdaptParameters(.)$ function will first test the accuracy of the model on new incoming data yielding $newAcc$. The function then recalls the accuracy that was tested last time as $lastAcc$. Next, because we want $newAcc$ and $lastAcc$ to improve upon random guessing, we subtract the accuracy of random guessing from $newAcc$ and from $lastAcc$ (the $guessAcc$ was already subtracted from $currentParams.lastAcc$ in the last update). We take the accuracy of random guessing ($guessAcc$) to be $1/nClasses$.[1] The intuitive reason to subtract $guessAcc$ is that a $lastAcc$ that is no greater than $guessAcc$ suggests that the model is no better than guessing just based on the number of classes. That in turn suggests that concept drift has likely occurred and so old data should be discarded.

[1] We have tried to set $guessAcc$ to the accuracy of guessing the most frequent class always. That would be a good random strategy for skewed data. But after testing, we found $guessAcc = 1/nClasses$ has a higher accuracy when the batch size is small and the dataset is very skewed towards one class. Otherwise, the two values of $guessAcc$ have similar accuracy.

**Algorithm 2:** UpdateSubTree

**input :** $\boldsymbol{X}$, $\boldsymbol{Y}$, incoming batch of training data for current node
$E(\cdot)$, the stopping criteria
$G(\cdot)$, the function to score the fitness of a feature for splitting
**global :** $current.X, current.Y$, the data retained of previous update
$current.gRecord$, the sorted data retained of previous update
$current.ranges$, the ranges for splitting retained of previous update
$current.label$, the prediction label

1 **begin**
2 Discard the oldest $(size(X) + size(current.X) - currentParams.rSize)$ rows in $current.X$ and $current.Y$;
3 $current.X, current.Y \leftarrow current.X + X, current.Y + Y$;
4 // insert the incoming batch, $currentParams.rSize$ rows will be in $current$ after insertion.
5 **if** E($current$, $current.X$, $current.Y$) is $True$ **then**
6 // don't split subtree more
7 $current.label \leftarrow$ the class having the greatest probability;
8 **else**
9 $current.gRecord, ranges \leftarrow minGInc(X, Y, G, current.gRecord)$;
10 // find the best value ranges
11 **for** each $range$ in $ranges$ **do**
12 $child \leftarrow current.children[i]$;
13 **if** $range$ not in $current.ranges$ **then**
14 // we need to rebuild subtree
15 $x, y \leftarrow$ Split ($current.X$, $current.Y$) at $range$;
16 $child \leftarrow BuildSubTree(x, y, child, E(.), G(.))$;
17 **else**
18 $x, y \leftarrow$ Split ($X$, $Y$) at $range$;
19 $child \leftarrow UpdateSubTree$(x, y, child, E(.), G(.), child.(X, Y, gRecord, ranges, label),currentParams.rSize);
20 **end**
21 $i++$;
22 **end**
23 $current.ranges \leftarrow ranges$;
24 **end**
25 **end**

Following that, $AdaptParameters(.)$ will calculate the rate of change ($rChange$) of $currentParams.rSize$ by the equation on line 23 of Algorithm 3:

- When $newAcc/lastAcc >= 1$, the $max$ in the exponent will ensure that $rChange$ will be $(newAcc/lastAcc)^2$. In this way, the $rChange$ curves slightly upward when $newAcc$ is

**Algorithm 3:** AdaptParameters

**input :** $\boldsymbol{X}$, $\boldsymbol{Y}$, incoming training data
**output :** $maxHeight$, the maximum height of the tree.
**global :** $currentParams.iRate$, the increase rate of $currentParams.rSize$ when the accuracy is stable
$currentParams.rSize$, the retain size of data
$currentParams.warmSize$, minimum size of the data for the decision tree to be considered ready to test
$currentParams.coldStartup$, $True$ if the decision tree is in Cold Startup mode.
$currentParams.(lastAcc, lastSize)$, the accuracy and data size of testing of previous update

1 **begin**
2 $guessAcc \leftarrow 1/nClasses$;
3 // $nClasses$ is the number of classes in $Y$
4 $newAcc \leftarrow evaluate(\boldsymbol{X}, \boldsymbol{Y}) - guessAcc$;
5 $lastAcc \leftarrow currentParams.lastAcc$;
6 // accuracy on latest batch
7 **if** $currentParams.coldStartup == True$ **then**
8 **while** $currentParams.rSize + size(X)$>=currentParams.warmSize **do**
9 $currentParams.warmSize \leftarrow 2 * currentParams.warmSize$;
10 **if** on the last 50% of the data, accuracy is better than current $guessAcc$ **then**
11 $currentParams.coldStartup \leftarrow False$;
12 // no longer in startup mode
$currentParams.(lastAcc, lastSize) \leftarrow newAcc - 1/nClasses, size(\boldsymbol{X})$;
13 continue;
14 **end**
15 **end**
16 $currentParams.rSize \leftarrow currentParams.rSize + size(X)$;
17 **else**
18 **if** $newAcc <= 0$ **then**
19 // we have left startup mode but our accuracy is bad, indicating concept drift
$currentParams.rSize \leftarrow size(\boldsymbol{X})$
20 **else if** $oldAcc <= 0$ **then**
21 $currentParams.rSize \leftarrow currentParams.rSize + size(X)$;
22 **else**
23 $currentParams.iRate \leftarrow currentParams.iRate * lastAcc/newAcc$;
24 $rChange \leftarrow (newAcc/lastAcc)^{(max(2,(3-newAcc/lastAcc)))}$;
25 $currentParams.rSize \leftarrow min(currentParams.rSize * rChange + currentParams.iRate * size(\boldsymbol{X}), currentParams.rSize + size(\boldsymbol{X}))$;
26 $currentParams.(lastAcc, lastSize) \leftarrow newAcc, size(\boldsymbol{X})$;
27 **end**
28 **end**
29 **if** $currentParams.rSize < size(X)$ **then**
30 $currentParams.rSize \leftarrow size(X)$;
31 **end**
32 $maxHeight \leftarrow log2(currentParams.rSize)$;
33 **end**

equal to, or slightly higher than *lastAcc*, but curves steeply upward when *newAcc* is much larger than *lastAcc*.
- When $newAcc/lastAcc < 1$, *rChange* is equal to $(newAcc/lastAcc)^{3-newAcc/lastAcc}$. In this way, *rChange* is flat or curves slightly downward when *newAcc* is slightly lower than *lastAcc* but curves very steeply downwards when *newAcc* is much lower than *lastAcc*.

Other functions to set *rChange* are possible, but this one has the following properties that we expected: (i) it is continuous regardless of the values of newAcc and lastAcc; (ii) *rChange* is close to 1 when *newAcc* is close to *lastAcc*; (iii) when *newAcc* differs from *lastAcc* significantly in either direction, *rChange* reacts strongly.

Finally, we will calculate and update the new $currentParams.rSize$ by multiplying the old $currentParams.rSize$ by *rChange*. To effect a slow increase in $currentParams.rSize$ when $newAcc \approx lastAcc$ and $rChange \approx 1$, we increase $currentParams.rSize$ by $currentParams.iRate * size(X)$ in additional to $currentParams.rSize(old) * rChange$, where $currentParams.iRate$ (the increase rate) is a number that is maintained from one call to $AdaptParameters(.)$ to another. The *min* in line 25 reflects the fact that the size of the incoming data is $size(X)$, so $currentParams.rSize$ cannot increase by more than $size(X)$. Also, we do not allow $currentParams.rSize$ to be less than $size(X)$, because we do not want to forget any new incoming data. When $AdaptParameters(.)$ is called the first time, we will set $currentParams.rSize = size(X) + size(current.X)$.

The two special cases of lines 18 through 21 happen when $newAcc <= 0$ or $lastAcc <= 0$. When $newAcc <= 0$, the prediction of the model is no better than random guessing. In that case, we infer that the old data cannot help in predicting new data, so we will forget all of the old data by setting $currentParams.rSize = size(X)$. When $lastAcc <= 0$ but $newAcc > 0$, then all the old data may be useful. So we set $currentParams.rSize = currentParams.rSize + size(X)$.

The above adaptation strategy requires a dampening parameter $currentParams.iRate$ to limit the increase rate of $currentParams.rSize$. When the accuracy is large, the model may be close to its maximum possible accuracy, so we may want a smaller $currentParams.iRate$ and in turn to increase $currentParams.rSize$ slower. After a drastic concept drift event, when the accuracy has been significantly decreased, we want to increase $currentParams.iRate$ to retain more new data after forgetting most of the old data. This will shorten the duration of the cold start after the concept drift. To achieve this, we will adaptively change it as follows: $currentParams.iRate(new)$ equal to $currentParams.iRate(old) * lastAcc/newAcc$ before each time $currentParams.rSize$ is updated, as in line 23 of (algorithm 3). $currentParams.iRate$ will not be changed if either $lastAcc <= 0$ or $newAcc <= 0$.

Upon initialization, if the first increment is small, then *newAcc* may not exceed $1/nClasses$, and the model will forget all of the old data every time. To avoid such poor performance at cold start, the Forgetful Decision Tree will be initialized in Cold Startup mode. In that Cold Startup mode, the Forgetful Decision Tree will not forget any data (line 16 in algorithm 3). When $currentParams.rSize$ reaches $currentParams.warmSize$, the Forgetful Decision Tree will leave Cold Startup mode if *newAcc* is better than *guessAcc* since the last 50% of data arrived. Otherwise, $currentParams.warmSize$ will be doubled. The above process will be repeated until leaving the Cold Startup mode.

Max tree height is closely related to the size of data retained in the tree. We want each leaf node to have about one data item on average when the tree is perfectly balanced, so we always set $maxHeight = log2(currentParams.rSize)$.

In summary, algorithm 3 provide a method to adjust $currentParams.rSize$, $maxHeight$, and $currentParams.iRate$. We still need an initial value of $currentParams.iRate$. We show experimentally how to find that in section 4.1.

## 3.3 Forgetful random forests.

When new data is acquired from the data stream, the $UpdateForest(.)$ function (Algorithm 4) will incrementally update the entire random forest.

The Forgetful Random Forest (Algorithm 4) is based on the Forgetful Decision Tree described above in section 3.1. Each Random Forest contains *nTree* decision trees. The $UpdateSubTreeRF(.)$ and $BuildSubTreeRF(.)$ functions for each decision tree in the Random Forest are the same as those from the decision tree algorithms described in section 3.1 except:

- Only a limited number of features are considered at each split, increasing the chance of invoking $UpdateSubTree(.)$ during recursion, and thus saving time by avoiding the need to rebuild subtrees from scratch. The number of features considered by each Random Forest tree is uniformly and randomly chosen within range $(\lfloor\sqrt{nFeatures}\rfloor+1, nFeatures]$, where *nFeatures* is the number of features in the dataset. The features considered by each ensemble tree are randomly and uniformly selected without replacement when created and will not change in subsequent updates. Further, every node inside the same ensemble tree considers the same features.
- To increase the diversity in the forest, $UpdateSubTreeRF(.)$ function will randomly and uniformly discard old data without replacement, instead of discarding data based on time of insertion.
- To decrease the correlation between trees and increase the diversity in the forest, we give the user the option to choose the Leveraging Bagging [2] strategy to the data arriving at each Random Forest tree. The size of the data after bagging is $W$ times the size of original data, where $W$ is a random number with a expected value of 6, generated by a $Poisson(\lambda = 6)$ distribution. To avoid the performance hit resulting from too many copies of the data, we never allow $W$ to be larger than 10. Each data item in the expanded data is randomly and uniformly selected from the original data with replacement. We apply bagging to each decision tree inside the random forest.

## 3.4 Discard Poorly Performing Trees

To discard the trees with features that perform poorly after concept drift, we will call the $Discard(.)$ function (Algorithm 5) when *newAcc* is significantly less than *lastAcc*. As in algorithm 3, we will subtract the accuracy of random guessing *guessAcc* from *newAcc*

and *oldAcc* to show the improvement of the model with respect to random guessing. Significance is based on a $p-value$ test: the accuracy of the forest has changed with a $p-value < tThresh$ based on a 2 sample t-test. The variable $tThresh$ is a hyper-parameter that will be tuned in section 4.2.

To detect slight but continuous decreases in accuracy, we will update *lastAcc* and *lastSize* by averaging them with *newAcc* and $size(X)$ when we observe an insignificant change in accuracy. In other cases, when the change in accuracy is significant, we will replace *lastAcc* and *lastSize* with *newAcc* and $size(X)$ after the $Discard(.)$ function is called.

The $Discard(.)$ function (Algorithm 5) removes $((newAcc - lastAcc)/lastAcc) * nTree$ Random Forest trees having the least accuracy when evaluated on the new data. The discarded trees are replaced with new decision trees. Each new tree will take all the data from the tree it replaced, but the tree will be rebuilt, the *currentParams* for that tree will be re-initialized, and the considered features will be re-selected for that tree. After building the new tree, the algorithm will test the tree on the latest data to calculate $nTree.lastAcc$ and $newTree.lastSize$. In this way, new trees adapt their $currentParams.rSize$ and $currentParams.iRate$ with the newly arriving data.

## 4 TUNING THE VALUES OF THE HYPERPARAMETERS

Decision trees and random forests have six hyperparameters to set, which are $G(\cdot)$, $E(\cdot)$, $currentParams.warmSize$, $tThresh$, $nTree$, and $currentParams.iRate$. To find the best values for these hyperparameters, we generated 18 datasets with different intensity of concept drifts, number of concept drifts, and Gaussian noise, using the generator inspired by Harvard Dataverse [14].

After testing on the 18 new datasets, we find that some hyperparameters have optimal values (with respect to accuracy) that apply to all datasets. Others have values that vary depending on the dataset but can be learned.

Here are the ones whose optimal values can be fixed in advance.

- The evaluation function $G(\cdot)$ can be a Gini Impurity[6] score or an entropy reduction coefficient. Which one is chosen doesn't make a material difference, so we set $G(\cdot)$ to entropy reduction coefficient for all datasets.
- The maximum tree height (*maxHeight*) is adaptively set based on the methods in section 3.2 to log base 2 of $currentParams.rSize$. Applying other stopping criteria does not materially affect the accuracy. For that reason, we ignore other stopping criteria.
- To mitigate the inaccuracies of a cold startup, the model will not discard any data in Cold Startup mode. To leave Cold Startup mode, the accuracy should be better than random guessing on the last 50% of data, when $currentParams.rSize$ is at least $currentParams.warmSize$. $currentParams.warmSize$ adapts if it is too small (line 9 in algorithm 3), so we will set its initial value to 64 data items.

The parameters that must be learned through training are $currentParams.iRate$, $tThresh$, and $nTree$. These are described in section 3

**Algorithm 4:** UpdateForest

**input :** $(\boldsymbol{X}, \boldsymbol{Y})$, incoming training data for current node
$nTree$, the number of decision trees that are actively updated
$firstCall$, whether UpdateForest is called for the first time
$tThresh$, the threshold for discarding trees
$E(.)$, the stopping criteria, returns boolean
$G(.)$, a function to score the fitness of a feature for splitting

**global :** $current.trees$ the list of decision trees that are actively updated
$current.(lastAcc, lastSize)$ the accuracy and data size of testing of of previous update
$current.allParams$, the parameters that will be updated for each tree in $current.trees$

```
1  begin
2      newAcc ← fractionCorrect(current, X, Y) − 1/nClasses;
3      // nClasses is the number of classes in Y
4      lastAcc ← current.lastAcc;
5      if 2Sample_t_test(newAcc, lastAcc, size(X),
         current.lastSize)>tThresh then
6          if newAcc < lastAcc then
7              current.trees, current.allParams ←
                 Discard(((newAcc-lastAcc)/lastAcc)*nTree, X,
                 Y, current.trees, current.allParams, E(.), G(.));
8          end
9          current.lastAcc ← newAcc;
10         current.lastSize ← size(X);
11     else
12         current.lastAcc ←
             (lastAcc*current.lastSize+newAcc*size(X))
             / (current.lastSize+size(X));
13         // lastAcc is updated based on a weighted average,
             which is weighted by the size of the new data
             compared to the previous data.
14         current.lastSize ← current.lastSize + size(X);
15     end
16     if firstCall then
17         for each tree in current.trees do
18             maxHeight ← log2(size(X));
19             BuildSubTreeRF(Bagging(X, Y), tree,
                 E(maxHeight), G(.));
20         end
21     else
22         for each tree in current.trees do
23             baggedX, baggedY ← Bagging(X, Y);
24             maxHeight, current.allParams[tree] ←
                 AdaptParameters(baggedX, baggedY,
                 current.allParams[tree]);
25             tree ← UpdateSubTreeRF(baggedX, baggedY,
                 tree, E(maxHeight), G(.), tree.(X, Y, gRecord,
                 ranges, label), current.allParams[tree].rSize);
26         end
27     end
28 end
```

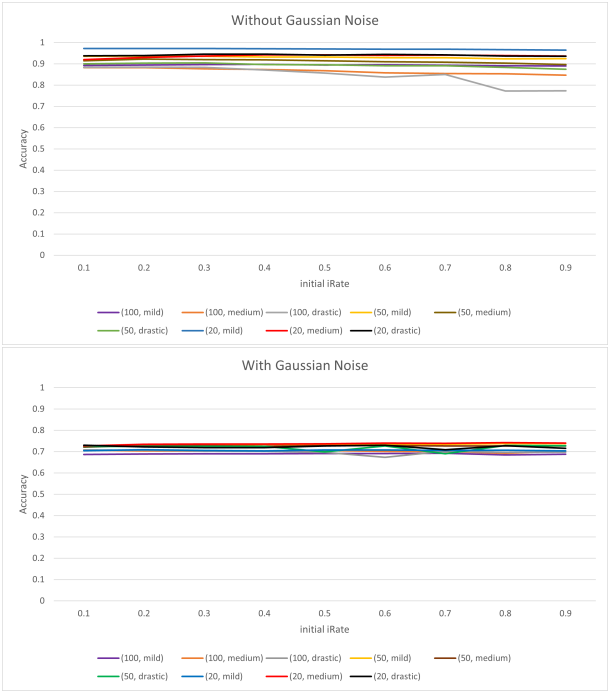
**Algorithm 5:** Discard

**input :** $nDiscard$, the number of trees that will be discarded
$(\boldsymbol{X}, \boldsymbol{Y})$, incoming training data for current node
$E(\cdot)$, the stopping criteria, returns boolean
$G(\cdot)$, a function to score the fitness of a feature for splitting

**global :** $current.trees$, the list of decision trees that are actively updated
$current.allParams$, the parameters that will be updated for each tree in $current.trees$

```
1  begin
2      discardTree ← nDiscard trees in current.trees having
         the smallest accuracy (fractionCorrect);
3      for each tree in discardTree do
4          treeNew ← create a new Tree;
5          treeNew ←
             BuildSubTreeRF(tree.X, tree.Y, treeNew, E(.), G(, );
6          Initialize current.allParams[treeNew];
7          current.allParams[treeNew] ←
             evaluate(treeNew, tree.X, tree.Y);
8          replace tree with treeNew in current.trees;
9      end
10 end
```

**Figure 1: $currentParams.iRate$ against Accuracy: $currentParams.iRate = 0.3$ usually results in a high accuracy**




## 4.1 Tuning Increase Rate

The adaptation strategy in section 3.2 need an initial value for parameter $currentParams.iRate$ which influences the increase rate of $currentParams.rSize$. Too much data will be retained if the initial $currentParams.iRate$ is large, but cold start will last too long if $currentParams.iRate$ is too small.

To find the best initial $currentParams.iRate$, we created 18 simulated datasets. Each dataset contains 50,000 data items labeled with 0, 1 without noise. Each item is characterized by 10 binary features. Each dataset is labeled with $(C, I)$, where $C$ means that it has $(C-1)$ uniformly distributed concept drifts, and $I$ is the intensity of the concept drift, while a mild concept drift will drift one feature, a medium concept drift will drift 3 features, and a drastic concept drift will drift 5 features. Also, for each dataset, we have one version without Gaussian Noise, and the other version with Gaussian Noise ($\lambda = 0$, $std = 1$, unit is the number of features). We tested the Forgetful Decision Tree with different initial $currentParams.iRate$ values on these synthetic datasets, while other hyperparameter values were fixed in advance as in section 4.

From Figure 1, we observe that the Foregetful Decision Tree does well when $currentParams.iRate = 0.3$ or $currentParams.iRate = 0.4$ initially. We will use $currentParams.iRate = 0.3$ as our initial setting and use it in the experiments of section 5 for all our algorithms, because most simulated datasets have higher accuracy at $currentParams.iRate = 0.3$ than at $currentParams.iRate = 0.4$.

Also, we observe a significant increase of $currentParams.iRate$ after each concept drift. This implies that new data accumulates after concept drift. However, the hyperparameter $currentParams.iRate$ will decrease to a low level after the accuracy starts to increase, thus avoiding the retention of too much data.

## 4.2 Tuning Discard Threshold

To find the best $tThresh$, we tested the Forgetful Random Forest with different $tThresh$ values on the synthetic datasets of section 4.1. Other hyperparameter values were fixed in advance as in section 4. To measure statistical stability in the face of the noise caused by the random setting of the initial seeds, we tested the random forests six times with different seeds and record the average values. From Figure 2 and Figure 3, we observe that all datasets enjoy a good accuracy when $tThresh = 0.05$, both with bagging and without bagging.

## 4.3 Tuning Number of Trees

To find the best $nTree$ value, we again use the simulated datasets of section 4.1. We tested Forgetful Random Forest with different $nTree$ values on these synthetic datasets, while the other hyperparameter values were fixed in advance as in section 4. To measure statistical stability in the face of the noise caused by randomization caused by the setting of the initial seeds, we tested the random forests six times with different seeds and recorded the average values. From Figure 4 and Figure 5, we observe that the accuracy of all datasets stops growing after $nTree > 20$ for both with bagging and without bagging, so we will set $nTree = 20$.

**Figure 2: $tThresh$ against Accuracy without Bagging: $tThresh$ = 0.05 usually results in a high accuracy**

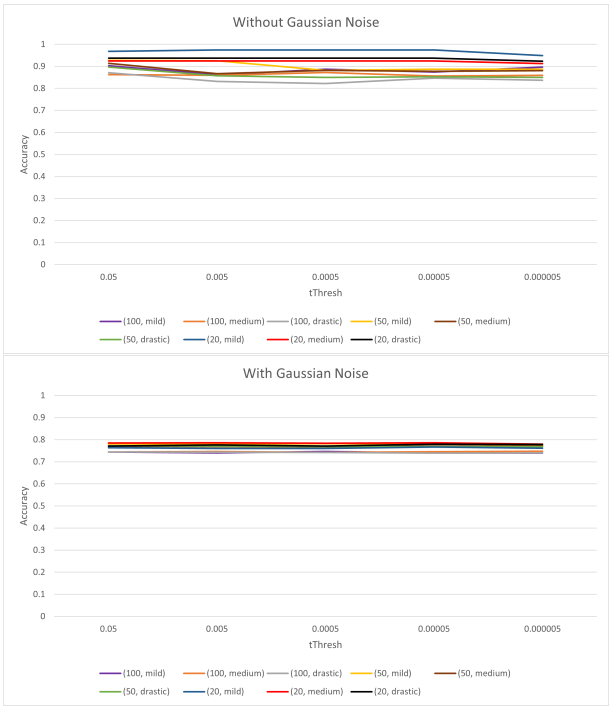



**Figure 3: $tThresh$ against Accuracy with Bagging: $tThresh$ = 0.05 usually results in a high accuracy**

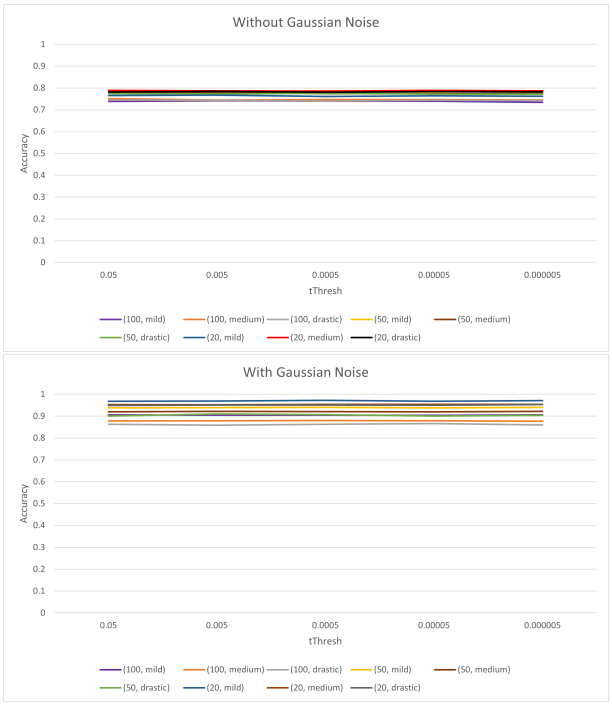



**Figure 4: $nTree$ against Accuracy without bagging: The accuracy stops increasing after $nTree$ >= 20**

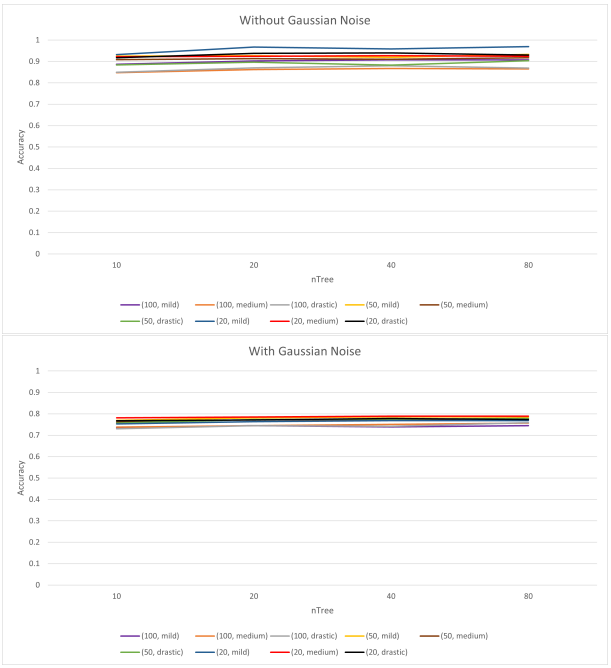
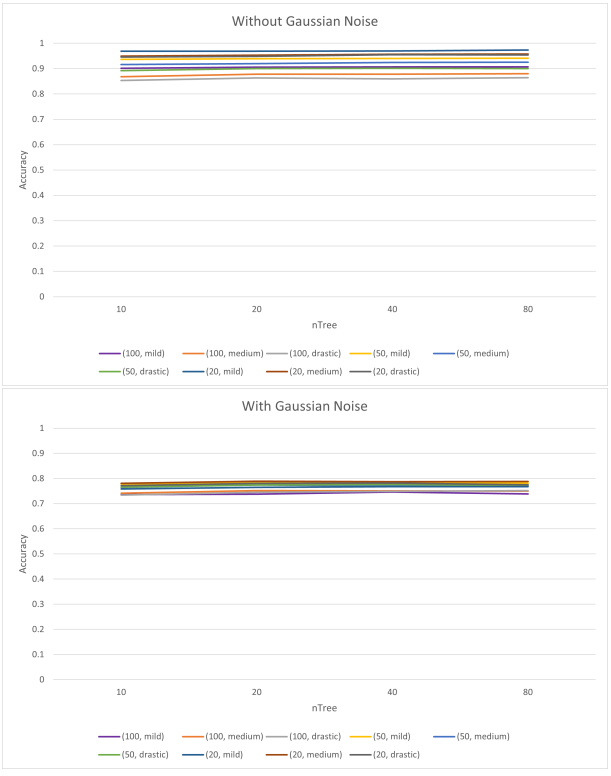



**Figure 5: $nTree$ against Accuracy with bagging: The accuracy stops increasing after $nTree$ >= 20**



### 4.4 Main Decision Tree Algorithm

The Forgetful Decision Tree main routine (algorithm 6) is called initially and then each time a new batch (an incremental batch) of data is received. The routine will make predictions with the tree before the batch and then update the batch. The algorithm 6 describes only the update part. The prediction part is what would be done by any Decision Tree. Accuracy results are recorded only after the accuracy flattens out, which means the accuracy changes 10% or less between the last 500 data items and the previous 500 data items.

**Algorithm 6:** ForgetfulDecisionTree

**input :**
*dataStream*, the stream of data

**begin**
  $E(.) \leftarrow MaximumTreeHeight(.)$ ;
  $G(.) \leftarrow Entropy(.)$;
  $currentParams.iRate \leftarrow 0.3$;
  $currentParams.coldStartup \leftarrow True$;
  $currentParams.warmSize \leftarrow 64$ data items;
  $currentParams.(lastAcc, lastSize) \leftarrow (0, 0)$;
  $root \leftarrow$ new decision tree root node;
  $initialFlag \leftarrow True$;
  **while** receiving new batch $X, Y$ from $dataStream$ **do**
    **if** $initialFlag$ **then**
      $initialFlag \leftarrow False$;
      $currentParams.rSize \leftarrow size(X)$;
      $maxHeight \leftarrow log2(currentParams.rSize)$;
      $root \leftarrow BuildSubTree(X, Y, root, E(maxHeight), G(.))$;
    **else**
      $maxHeight, currentParams \leftarrow AdaptParameters$(X, Y, currentParams.(iRate, rSize, warmSize, coldStartup, lastAcc, lastSize);
      $root \leftarrow UpdateSubTree$(X, Y, root, E(maxHeight), G(.), root.(X, Y, gRecord, ranges, label), currentParams.rSize))
    **end**
  **end**
**end**

### 4.5 Main Random Forest Algorithm

The Forgetful Random Forest main routine (algorithm 7) is called initially and then each time an incremental batch of data is received. The routine will make predictions with the Random Forest before the batch and then update the Random Forest. The algorithm 7 describes only the update part. The prediction part is what would be done by any Random Forest. Accuracy results will apply after the accuracy flattens out, which means the accuracy changes 10% or less between the last 500 data items and the previous 500 data items.

## 5 EXPERIMENTS

This section compares the following algorithms: Forgetful Decision Tree, Forgetful Random Forest with bagging, Forgetful Random Forest without bagging, Hoeffding Tree [21] [9], Hoeffding Adaptive Tree[1], iSOUP Tree[16], train once, and Adaptive Random Forest[7]. The forgetful algorithms from section 3 use the hyperparameter settings from section 4 on both real and simulated datasets produced by others.

We measure time consumption, the accuracy and, where appropriate, the F1-score.

Because we have tuned the forgetful data structures, we have also tuned the state-of-the-art algorithms on the same generated data sets of section 4.The following settings yield the best accuracy for the state-of-the-art algorithms

**Algorithm 7:** ForgetfulRandomForest

**input :** $features$, all features in data
*dataStream*, the stream of data for training

**begin**
  $E(.) \leftarrow MaximumTreeHeight(.)$ ;
  $G(.) \leftarrow Entropy(.)$;
  $current.lastAcc \leftarrow 0$;
  $current.lastSize \leftarrow 0$;
  $current.trees \leftarrow nTree$ new decision trees;
  $nTree \leftarrow 20$;
  $tThresh \leftarrow 0.05$ **for** each $tree$ in $current.trees$ **do**
    //Initialize *consider* of each *tree*
    $nConsider \leftarrow$ single value from uniform distribution random variable between $(\lfloor\sqrt{size(features)}\rfloor + 1$ and $size(features)]$ ;
    $allConsider[tree] \leftarrow$ uniformly and randomly select $nConsider$ features from $features$ without replacement;
    $current.allParams[tree].iRate \leftarrow 0.3$;
    $current.allParams[tree].coldStartup \leftarrow True$;
    $current.allParams[tree].warmSize \leftarrow 64$ data items;
    $current.allParams[tree].(lastAcc, lastSize) \leftarrow (0, 0)$
  **end**
  $firstCall \leftarrow True$;
  **while** receiving new batch $X, Y$ from $dataStream$ **do**
    $firstCall \leftarrow False$;
    $current \leftarrow UpdateForest$(X, Y, nTree, firstCall, tThresh, current.(lastAcc, lastSize, trees), allConsider,current.allParams, E(.), G(.));
  **end**
**end**

- Previous papers ([21] and [9]) provide two different configurations for the Hoeffding Tree. The configuration from [21] usually has the highest accuracy, so we will use it in the following experiment: $split\_confidence = 10^{-7}, grace\_period =$

200, and $tie\_threshold = 0.05$. Because the traditional Hoeffding Tree cannot deal with concept drift, so we set $leaf\_prediction = NaiveBayesAdaptive$ to allow the model to adapt when concept drift happens.

- The designer of Hoeffding Adaptive Tree suggests six versions of configuration of the Hoeffding Adaptive Tree, which are HAT-INC, HATEWMA, HAT-ADWIN, HAT-INC NB, HATEWMA NB, and HAT-ADWIN NB. HAT-ADWIN NB has the best accuracy, and we will use it in the following experiment. The configuration is $leaf\_prediction = NaiveBayes$, $split\_confidence = 0.0001$, $grace\_period = 200$, and $tie\_threshold = 0.05$.
- The designer provides only one configuration for iSOUP-Tree [16], so we will use it in the following experiment. The configuration is $leaf\_prediction = adaptive$, $split\_confidence = 0.0001$, $grace\_period = 200$, and $tie\_threshold = 0.05$.
- For the train-once model, we will train the model only once with all of the arrived data before starting to apply accuracy result, and the model will never be updated again. In this case, we will use non-incremental decision tree, which is CART algorithm [13], to fit the model. We use the setting with the best accuracy, which is $criterion = gini$, and no other restrictions.
- The designer of Adaptive Random Forest provided six variant configurations of Adaptive Random Forest [7]: the variants $ARF_{moderate}, ARF_{fast}, ARF_{PHT}, ARF_{noBkg}, ARF_{stdRF}$, and $ARF_{maj}$. $ARF_{fast}$ has the highest accuracy in most cases that we tested, so we will use that configuration: $\delta_w = 0.01$, $\delta_d = 0.001$, and $learners = 100$.

## 5.1 Hyperparameter Settings for Forgetful Data Structures

The previous sections gave us the following hyperparameters settings:

- $\dot{G}()$ is entropy reduction for all of the Forgetful Random Forests and the Forgetful Decision Tree.
- $currentParams.warmSize$ should be initially small, because it can be adaptively increased. We initially set it to 64 data items.
- Following the tuning of section 4.1, we will initialize $currentParams.iRate$ to 0.3.
- $tThresh$ defines the threshold for discarding trees inside a Random Forest. Following the tuning of section 4.2, we will set $tThresh = 0.05$.
- $nTree$ is the number of trees inside the Forgetful Random Forest. Following the tuning of section 4.3, we set it to 20 for both versions of the Forgetful Random Forests.

## 5.2 Metrics

Beside **accuracy**, we also use **precision**, **recall**, and **F1-score** to evaluate our methods. Precision and recall are appropriate to problems where there is a class of interest and the question is which percentage of predictions of that class are correct (precision) and how many instances of that class are predicted (recall). This is appropriate for the phishing application where the question is whether the website is a phishing website. Accuracy is more appropriate in all other applications. For example, in the electricity datasets, price up and price down are both classes of interest. Therefore, we present precision, recall, and the F1-score for Phishing only. We use the following formula based on the confusion matrix for the following tests.

- $accuracy = \frac{|TruePositive|+|TrueNegative|}{Size(test-set)}$
- $precision = \frac{|TruePositive|}{|TruePositive|+|FalsePositive|}$
- $recall = \frac{|TruePositive|}{|TruePositive|+|FalseNegative|}$
- $F1-score = \frac{|2*TruePositive|}{2*|TruePositive|+|FalseNegative|+|FalsePositive|}$

In contrast to most static labeled machine learning tasks, we don't partition the data into a training set and a test set. Instead, when each batch of data arrives, we measure the accuracy and F1-score of the predictions on that incremental set, before we use it to update the models. We start measuring accuracy and F1-score after the accuracy of Forgetful Decision Tree flattens out, in order to avoid the inaccuracies during start-up. That point is different for each dataset as described in section 5.3, but all algorithms will start measuring the accuracy and F1-score at same point for the same dataset.

## 5.3 Datasets

We use four real datasets and two synthetic datasets to test the performance of our forgetful methods against the state-of-the-art incremental algorithms.

- Forest Cover Type (ForestCover) [11] dataset captures images of forests for each 30 x 30 meter cell determined from the US Forest Service (USFS) Region 2 Resource Information System (RIS) data. Each increment consists of 400 image observations. The task is to infer the forest type. This dataset suffers from concept drift because later increments have different mappings from input image to forest type than earlier ones. For this dataset, the accuracy first increases and then flattens out after the first 24,000 data items have been observed, out of 581,102 data items.
- The Electricity [15] dataset describes the price and demand of electricity. The task is to forecast the price trend in the next 30 minutes. Each increment consists of data from one day. This data suffers from concept drift because of market and other external influences. For this dataset, the accuracy never stabilizes, so we start measuring accuracy after the first increment, which is after the first 49 data items have arrived, out of 36,407 data items.
- Phishing [20] contains 11,055 web pages accessed over time, some of which are malicious. The task is to predict which pages are malicious. Each increment consists of 100 pages. The tactics of Phishing purveyors get more sophisticated over time, so this dataset suffers from concept drift. For this dataset, the accuracy flattens out after the first 500 data items have arrived.
- Power Supply [22] contains three years of power supply records of an Italian electrical utility, comprising 29,928 data items. Each data item contains two features, which are the amount of power supplied from the main grid and

the amount of power transformed from other grids. Each data item is labeled with the hour of the day when it was collected (from 0 to 23). The task is to predict the label from the power measurements. Concept drifts arise because of season, weather, and the differences between working days and weekends. Each increment consists of 100 data items, and the accuracy flattens out after 1,000 data items have arrived.

- Two Synthetic datasets from [14]. Both are time-ordered and are designed to suffer from concept drift over time. One, called *Gradual*, has 41,000 data points. Gradual is characterized by complete label changes that happen gradually over 1,000 data points at three single points, and 10,000 data items between each concept drift. Another dataset, called *Abrupt*, has 40,000 data points. It undergoes complete label changes at three single points, with 10,000 data items between each concept drift. Each increment consists of 100 data points. Unlike the datasets that were used in section 4, these datasets contains only four features, two of them are binary classes without noise, and the other two are sparse values generated by $sin(x)$ and $sin^{-1}(y)$, where $x$ and $y$ are the uniformly generated random numbers. For both datasets, the accuracy flattens out after 1,000 data items have arrived.

Because the real datasets all contain categorical variables and our methods don't handle those directly, we modify the categorical variables into their one-hot encodings using the OneHotEncoder of sklearn [17]. For example, a categorical variable $color = \{R, G, B\}$ will be transfered to three binary variables $isR = \{True, False\}$, $isG = \{True, False\}$, and $isB = \{True, False\}$. Also, all of the forgetful methods in the following tests use only binary splits at each node.

To measure statistical stability in the face of the noise caused by the randomized setting of the initial seeds, we test all decision trees and random forests six times with different seeds and record the average values with a 95% confidence interval for time consumption, accuracy, and F1-score.

The following experiments are performed on an Intel Xeon Platinum 8268 24C 205W 2.9GHz Processor with 200 gigabytes of memory.

## 5.4 Quality and Time Performance of Forgetful Decision Tree

Figure 6 compares the time consumption of different incremental decision trees. For all datasets, the Forgetful Decision Tree takes at least three times less time than the other incremental methods.

Figure 7 compares the accuracy of different incremental decision trees and train-once model. For all datasets, the Forgetful Decision Tree is as accurate or more accurate than other incremental methods.

Figure 8 compares the precision, recall, and F1-score of different incremental decision trees. Because these metrics are not appropriate for other datasets, we use them only on the phishing dataset. The precision and recalls vary. For example, the Hoeffding Adaptive Tree has a better precision but a worse recall than the Forgetful Decision Tree, while the iSOUP tree has a better recall but a worse precision. Overall, the Forgetful Decision Tree has a similar F1-score to the other incremental methods.

**Figure 6: Time Consumption of Decision Trees: Based on this logarithmic scale, the Forgetful Decision Tree is at least three times faster than the state-of-the-art incremental Decision Trees.**

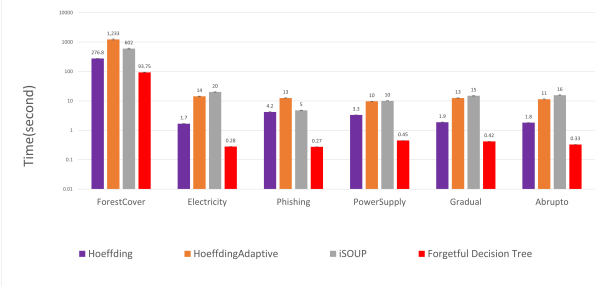



**Figure 7: Accuracy of Decision Trees: The Forgetful Decision Tree is at least as accurate as the state-of-the-art incremental Decision Trees(iSOUP-tree) and at most 9% more accurate.**

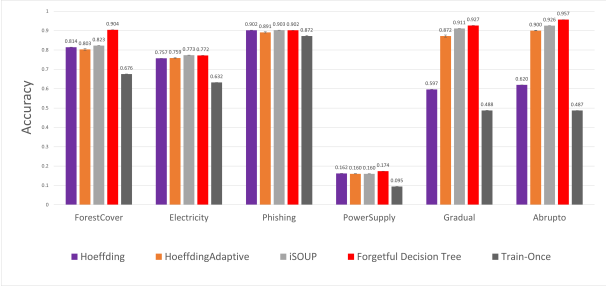



**Figure 8: Precision, Recall, and F1-score of Decision Trees: While precision and recall results vary, the Forgetful Decision Tree has a similar F1-score to the other incremental Decision Trees for the Phishing dataset (the only one where F1-score is appropriate).**

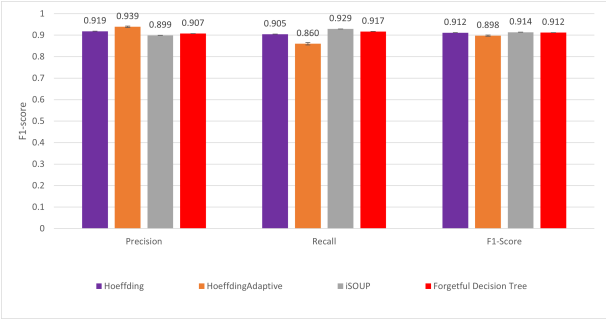


**Figure 9: Time Consumption of Random Forests: As can be seen on this logarithmic scale, the Forgetful Random Forest without bagging is at least 24 times faster than the Adaptive Random Forest. The Forgetful Random Forest with bagging is at least 2.5 times faster than Adaptive Random Forest.**

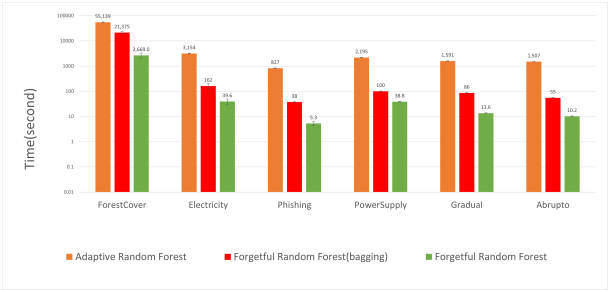



**Figure 10: Accuracy of Random Forests: Without bagging, the Forgetful Random Forest is slightly less accurate (at most 2%) than the Adaptive Random Forest. With bagging, the Forgetful Random Forest has a similar accuracy to the Adaptive Random Forest**

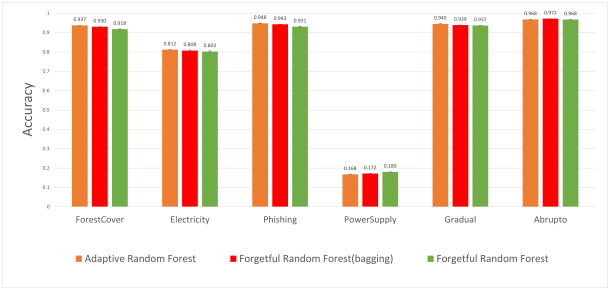
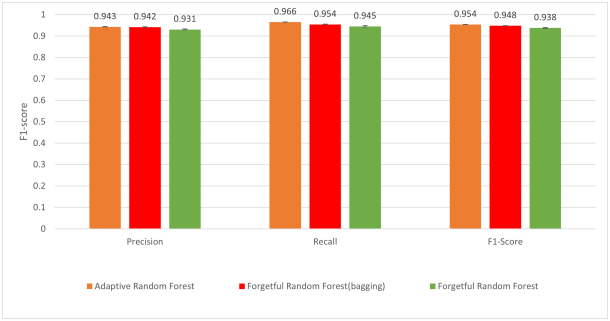



**Figure 11: F1-score of Random Forests: the Forgetful Random Forest without bagging has a lower precision, recall, and F1-score (by 0.02) compared to the Adaptive Random Forest. The Forgetful Random Forest with bagging also has a F1-score (by 0.01) but a similar precision to the Adaptive Random Forest.**




## 5.5 Quality and Time Performance of Forgetful Random Forest

Figure 9 compares the time performance of maintaining different random forests. From this figure, we observe that the Forgetful Random Forest without bagging is the fastest algorithm. In particular, it is at least 24 times faster than the Adaptive Random Forest. The Forgetful Random Forest with bagging is about 10 times slower than without bagging, but it's still 2.5 times faster than the Adaptive Random Forest.

Figure 10 compares the accuracy of different random forests. From these figures, we observe that the Forgetful Random Forest without bagging is slightly less accurate than the Adaptive Random Forest (by at most 2%). By contrast, the Forgetful Random Forest with bagging has a similar accuracy compared to the Adaptive Random Forest. Sometimes the loss of accuracy might be worth it if the streaming data enters at a high enough rate.

Figure 11 compares the precision, recall, and F1-score of training different random forests when these evaluations are appropriate. From these figures, we observe that the Forgetful Random Forest without bagging has a lower precision, recall, and F1-score than the Adaptive Random Forest (by at most 0.02). By contrast, the Forgetful Random Forest with bagging has a similar precision but a lower recall and F1-score (by at most 0.01) compared to the Adaptive Random Forest.

# 6 CONCLUSION

Forgetful Decision Trees and Forgetful Random Forests constitute simple new fast and accurate incremental data structure algorithms. We have found that

- The Forgetful Decision Tree is at least three times faster and at least as accurate as state-of-the-art incremental Decision Tree algorithms for a variety of concept drift datasets. When the precision, recall, and F1-score are appropriate, the Forgetful Decision Tree has a similar F1-score as state-of-the-art incremental Decision Tree algorithms.
- The Forgetful Random Forest without bagging is at least 24 times faster than state-of-the-art incremental Random Forest algorithms, but is less accurate by 2% or less.
- By contrast, the Forgetful Random Forest with bagging has a similar accuracy to and is 2.5 times as fast as the Adaptive Random Forest.
- At a conceptual level, our experiments show that it helps to set parameter values based on changes in accuracy. We do this for *currentParams.rSize* (retained data), *maxHeight* (of decision trees), *currentParams.iRate* (increase rate of *currentParams.rSize*), and the number of features to consider at each decision tree in the Forgetful Random Forests.

In summary, forgetful data structures speed up traditional decision trees and random forests and help them adapt to concept drift. Further, we have observed that bagging can increase accuracy but at a substantial cost in time performance. The most pressing question for future work is whether some other method can be combined with forgetfulness to increase accuracy at less cost in a streaming concept drift setting.

# REFERENCES


[1] Albert Bifet and Ricard Gavaldà. 2009. Adaptive Learning from Evolving Data Streams. In Advances in Intelligent Data Analysis VIII, Niall M. Adams, Céline Robardet, Arno Siebes, and Jean-François Boulicaut (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 249–260.

[2] Albert Bifet, Geoff Holmes, and Bernhard Pfahringer. 2010. Leveraging Bagging for Evolving Data Streams. In Machine Learning and Knowledge Discovery in Databases, José Luis Balcázar, Francesco Bonchi, Aristides Gionis, and Michèle Sebag (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 135–150.

[3] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. 2008. Supervised Learning. Springer Berlin Heidelberg, Berlin, Heidelberg, 21–49. https://doi.org/10.1007/978-3-540-75171-7_2

[4] Chris Diehl and G. Cauwenberghs. 2003. SVM incremental learning, adaptation and optimization. Proceedings of the International Joint Conference on Neural Networks, 2003. 4 (2003), 2685–2690 vol.4.

[5] Pedro Domingos and Geoff Hulten. 2000. Mining High-Speed Data Streams. In Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (Boston, Massachusetts, USA) (KDD '00). Association for Computing Machinery, New York, NY, USA, 71–80. https://doi.org/10.1145/347090.347107

[6] C. Gini. 1997. Concentration and dependency ratios. (1997), 769–789.

[7] Heitor M. Gomes, Albert Bifet, Jesse Read, Jean Paul Barddal, Fabrício Enembreck, Bernhard Pfharinger, Geoff Holmes, and Talel Abdessalem. 2017. Adaptive Random Forests for Evolving Data Stream Classification. Mach. Learn. 106, 9–10, 1469–1495. https://doi.org/10.1007/s10994-017-5642-8

[8] Wassily Hoeffding. 1963. Probability Inequalities for Sums of Bounded Random Variables. J. Amer. Statist. Assoc. 58, 301 (1963), 13–30. https://doi.org/10.1080/01621459.1963.10500830 arXiv:https://www.tandfonline.com/doi/pdf/10.1080/01621459.1963.10500830

[9] Geoff Hulten, Laurie Spencer, and Pedro M. Domingos. 2001. Mining time-changing data streams. In Knowledge Discovery and Data Mining.

[10] Elena Ikonomovska, Joao Gama, and Saso Dzeroski. 2011. Learning model trees from evolving data streams. Data Mining and Knowledge Discovery 23, 128–168. https://doi.org/10.1007/s10618-010-0201-y

[11] Dr. Charles W. Anderson Jock A. Blackard, Dr. Denis J. Dean. 1998. Covertype Data Set. https://archive.ics.uci.edu/ml/datasets/Covertype

[12] Yann LeCun, Y. Bengio, and Geoffrey Hinton. 2015. Deep Learning. Nature 521 (05 2015), 436–44. https://doi.org/10.1038/nature14539

[13] Wei-Yin Loh. 2011. Classification and Regression Trees. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 1 (01 2011), 14 – 23. https://doi.org/10.1002/widm.8

[14] Jesús López Lobo. 2020. Synthetic datasets for concept drift detection purposes. Harvard Dataverse. https://doi.org/10.7910/DVN/5OWRGB

[15] A. Bifet M. Harries, J. Gama. 2009. https://www.openml.org/d/151

[16] Aljaz Osojnik, P. Panov, and Saso Dzeroski. 2017. Tree-based methods for online multi-target regression. Journal of Intelligent Information Systems 50 (2017), 315–339.

[17] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research 12, 85 (2011), 2825–2830. http://jmlr.org/papers/v12/pedregosa11a.html

[18] Robi Polikar, L. Upda, S.S. Upda, and Vasant Honavar. 2001. Learn++: An incremental learning algorithm for supervised neural networks. Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on 31 (12 2001), 497 – 508. https://doi.org/10.1109/5326.983933

[19] Stuart Russell and Peter Norvig. 2010. Artificial Intelligence: A Modern Approach (3 ed.). Prentice Hall.

[20] Tegjyot Singh Sethi and Mehmed Kantardzic. 2017. On the Reliable Detection of Concept Drift from Streaming Unlabeled Data. arXiv. https://doi.org/10.48550/ARXIV.1704.00023

[21] Jian Sun, Hongyu Jia, Bo Hu, Xiao Huang, Hao Zhang, Hai Wan, and Xibin Zhao. 2020. Speeding up Very Fast Decision Tree with Low Computational Cost. (7 2020), 1272–1278. https://doi.org/10.24963/ijcai.2020/177 Main track.

[22] X. Zhu. 2010. Stream Data Mining Repository. http://www.cse.fau.edu/~xqzhu/stream.html