TECHNICAL UNIVERSITY OF CRETE SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Online Ensemble Classification Algorithms of Big Data Streams at Apache Flink

Vasileios Vittis

Thesis Committee:

Professor Antonios Deligiannakis (ECE)

Professor Minos Garofalakis (ECE)

Associate Professor Vasilis Samoladas (ECE)

September 2021

Abstract

The growing need to make high-precision real-time decisions from dynamic data creates, the need to create modern systems capable of coping with diverse problems. Thus, the demands generated by the 4 Vs (volume, variety, velocity, and veracity) make the classical systems inefficient, thus creating space for systems that process data only once, without the need to store them. Ensemble Systems consist of individual subsystems with different characteristics, participating in the voting process in order to make the final decision. These subsystems are implemented by the state-of-the-art decision tree algorithm, Hoeffding Tree, due to its simple construction and the fewer assumptions it makes. It is important that such models take advantage of the available distributed environments in order to effectively speed up the learning process. In this dissertation, we create a distributed ensemble learning system for binary classification, consisting of Hoeffding Trees, creating a Random Forest. After observations about the response time and development space of the specific system, we implemented techniques that purposefully solve such problems. The results of the experimental process confirm the proposed methodology, when compared with corresponding techniques in the literature.

Περίληψη

Η αυξανόμενη ανάγκη λήψης αποφάσεων με υψηλή ακρίβεια σε πραγματικό χρόνο από δυναμικά δεδομένα, δημιουργεί την ανάγκη δημιουργίας σύγχρονων συστημάτων, ικανά να ανταπεξέλθουν σε όλων των ειδών προβλημάτων. Έτσι, οι απαιτήσεις που παράγονται από τον όγκο και τον ρυθμό και την αλλαγή των δεδομένων καθιστούν τα κλασσικά συστήματα μη αποδοτικά, με αποτέλεσμα να δημιουργείται χώρος για συστήματα που επεξεργάζονται τα δεδομένα μόνο μια φορά, χωρίς την ανάγκη αποθήκευσης τους. Τα συλλογικά συστήματα εκμάθησης (Ensemble Systems), αποτελούνται από επιμέρους υποσυστήματα με διαφορετικά χαρακτηριστικά, συμμετέχοντας στην διαδικασία ψηφοφορίας με σκοπό την λήψη της τελικής απόφασης. Η κορωνίδα αυτών των υποσυστημάτων είναι ο state-of-the-art αλγόριθμος δένδρων αποφάσεων, Hoeffding Tree, λόγω της απλής κατασκευής τους και των λιγότερων υποθέσεων που κάνουν. Σημαντικό είναι τέτοιου είδους μοντέλα να εκμεταλλεύονται τα διαθέσιμα κατανεμημένα περιβάλλοντα, έτσι ώστε να επιταχυνθεί αποτελεσματικά η διαδικασία εκμάθησης. Στη συγκεκριμένη διπλωματική εργασίας, δημιουργούμε ένα κατανεμημένο συλλογικό σύστημα δυαδικών αποφάσεων, αποτελούμενο από Hoeffding Trees, δημιουργώντας ένα Random Forest. Ύστερα παρατηρήσεων σχετικά με τον χρόνο απόκρισής και χώρο ανάπτυξης του συγκεκριμένου συστήματος, υλοποιήθηκαν τεχνικές που στοχευμένα λύνουν τέτοιου είδους προβλήματα. Τα αποτελέσματα της πειραματικής διαδικασίας επιβεβαιώνουν την προτεινόμενη μεθοδολογία, όταν συγκρίνονται με αντίστοιχες τεχνικές της βιβλιογραφίας.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my thesis supervisor Prof. Antonios Deligiannakis for advising my wisely and supporting me on various problems that I faced. I also would like to express my admiration to my two committee members Prof. Minos Garofalakis and Vasilis Samoladas for being a great motivation throughout my studies.

Moreover, I would like to thank my family for their continuous support during all the ups and downs. I also like to thank my friends from Chania for the experiences we lived together and especially I want to thank Nikos Tzimos for all the endless nights of study and Giannis Roditis for being there for me at any time I needed help.

This thesis is a product of

Vasileios Vittis

Table of Contents

List of Tables

List of Figures

Chapter 1:

Introduction

1.1 Thesis Contribution

The following thesis proposes a streaming ensemble system composed of Hoeffding Trees (VFDT), forming a Random Forest, implemented in a distributed environment at Apache Flink. The thesis is build based on three axes, real time training, anytime output and exact-one state consistency. The proposed implementation considers scenarios where the Concept Drift is present, proposing an alternative version of the most well-known Drift Detection Method (DDM), as well as testing the impact of different combination functions. Also, three optimizations were implemented, including the Gaussian Approximation, which is a well-established efficient way of handling numeric attributes in a streaming setting, a stricter and memory-efficient version of Hoeffding Trees (SVFDT) which guarantees almost the same predictive performance as VFDT, resulting higher scalability and lower computational and memory costs and a proposition which tackles the problem of building deep Hoeffding Trees in periods of data stagnation during Big Data streams. To the best of our knowledge, the proposed combination of algorithms which consists of the aforementioned ensemble system, makes the existing thesis unique.

1.2 Thesis Outline

In Chapter 2, we present the main concept of Ensemble Learning. We organize our analysis into three main components, starting from the base learner, where we explain the basic concept of decision trees and we describe the state-of-the art Hoeffding Tree algorithm as long as all its extensions. In addition, we formulate the notion of concept drift and all the existing concept drift detectors. Moreover, we present the most used combinations functions, as well as we analyze the problem of diversity in an ensemble system. In Chapter 3, we introduce the main contribution of this thesis. In Chapter 4, we provide a brief overview of all principles that rule Apache Flink framework as long with the implementation details of our solution. In Chapter 5, we discuss the results obtained, by our systemic analysis of every component of the proposed system, including evaluations of accuracy, run-time and memory consumption. Finally, in Chapter 6 we introduce our conclusion and propose the future work.

Chapter 2:

Background and Related Work

2.1 Data Stream Ensemble Learning

An ensemble learner can be described as a combination of multiple (weak) learners which form one (strong) learner with expected higher predictive performance. This statement, which may now sound logical to someone, was not always the case. In 2001, Nick Street et al [1], who proposed a new streaming ensemble algorithm for large-scale classification, concluded that their algorithm's accuracy appeared to be about the same as a single classifier, despite the fact that there was room for significant improvement in the future. Ensemble learning can be categorized either on the supervised or unsupervised learning. The key difference, is the assumption of the a priori knowledge of the true label of each instance. An unbounded data stream S, generated by *source* S_i^t, is a sequence of examples $z^t = (x^t, y^t)$ for $t = 1,2, \dots, T$ where x^t is a multi-dimensional instance observed at time t and $y^t \in \{-1, 1\}$ denotes the corresponding label, in case of binary classification. We consider a set of K (distributed) learners, $K = \{1, \dots, k\}$ where each learner observes a different sequence of instances. We denote as $s_i^t \in \{-1,1\}$ the local prediction of learner k at time t, resulting to the ensemble local prediction vector $s^t \triangleq (s_1^t, \dots, s_k^t)$. For each learner k we also maintain a weight vector $w^t \triangleq (w_1^t, \dots, w_k^t)$ which is combined linearly with the local predictions.

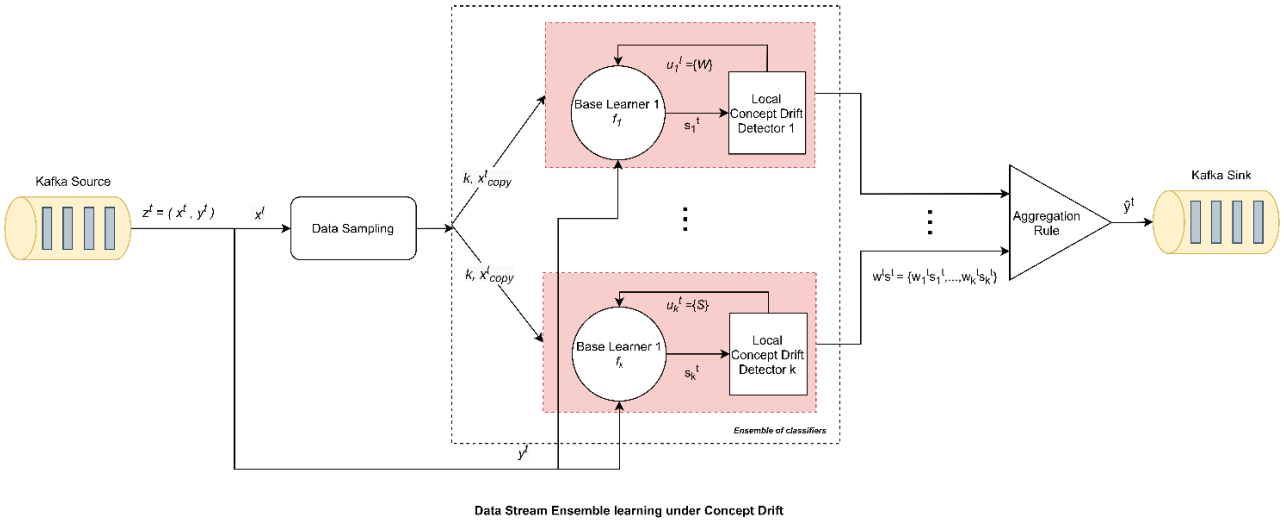


Figure 1. Ensemble Learning: x_i^t= multi-dimensional instance, s_i^t = local prediction of i classifier, s = local predictions vector, $\hat{y}^t$= global prediction, y^t= true label, w_i^t= local updated weights vector, u_i^t= a local indicator {W: warning, S: signal, F: false alarm}

The main motivation for using an ensemble classifier is the *no free lunch theorem* which is formulated by Wolpert [2]. According to it, there is no single classifier that is appropriate for all given tasks, thus we are looking to create a pool of diverse and complementary individual classifiers in order to compete at any given problem. In addition, based on Cha Zhang [3] who mentions that if in fact there was an expert, whose predictions were always true, we would never need any other decision maker. Alas, no such expert exists; every decision maker has an imperfect past record. The main challenges of every ensemble system are to control five metrics: bias, variance, accuracy, precision and diversity over different training sets, which in many cases are competing one with each other. Such a challenge is difficult to handle and the goal of ensemble systems is to create several classifiers with relatively fixed bias and by combining their outputs to reduce the variance.

In Figure 1.1, the latter forementioned author gives a vivid representation of the notion of an ensemble, by depicting three different models with their respective decision-making rules based on a two-feature dataset.

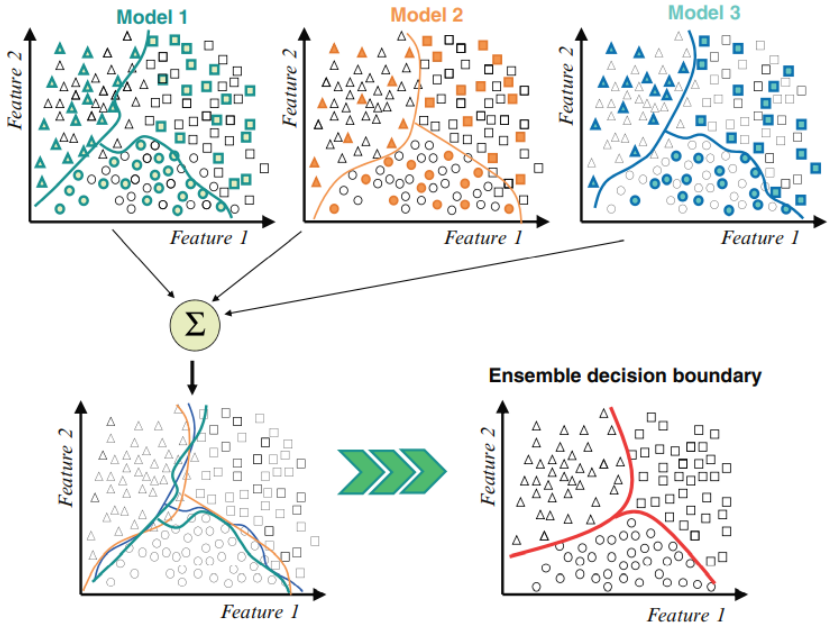


Fig. 1.1 Variability reduction using ensemble systems

Figure 2. Variability reduction using ensemble learning

Achieving such a state, requires the development of a complete ensemble learner following a holistic approach. A complete ensemble classifier has to be designed based on optimizing the following axes: Architecture and Voting combination, high Diversity, appropriate selection of Base Learner, Concept Drift adaptation and Distributed Implementation. Each one of these are thoroughly presented throughout this thesis.

2.2 Ensemble Architecture

The ensemble architecture defines how the classifiers interact with each other. There are several ways in which an ensemble can be formed but the main three ensemble arrangements are the following: Parallel, meta-learner and hierarchical. Briefly, a hierarchical ensemble imposes a treelike structure or a strict order (cascading) over its members, whereas in a meta-learning structure, the combiner (meta-learner) is trained on meta-data, adding an extra layer of base leaner. In the parallel arrangement, the most widely used architecture is the Flat Architecture. It is the simplest method, which makes the least assumptions about individual classifiers. An ensemble can be composed from heterogenous or homogeneous base learners. As S.Kotsiantis [4] mentions, the independence of classifier outputs is generally considered to be an advantage for obtaining better multiple classifier systems. Therefore, either all classifiers will be different, with or without forming coalitions, or it can be the same classifier with different settings. A common approach is to generate N classifiers using the same generation algorithm, all with different parameter settings, different subsets of attributes and training data. This structural architecture can be easily translated to parallel system using a learning algorithm per machine (see Section 2.3). Each classifier's output is aggregated by a combiner f, which can be a simple linear function such as Weighted Voting. Figure 3 shows the abstract Flat Architecture; with the light blue rectangles being the classifiers and with the dark blue their respective local prediction. The f function is responsible for collecting all partial predictions and produce the final one. In the next section we will discuss the base learner component. Selecting an appropriate base learner according to the classification problem is an important step for obtaining an accurate ensemble.

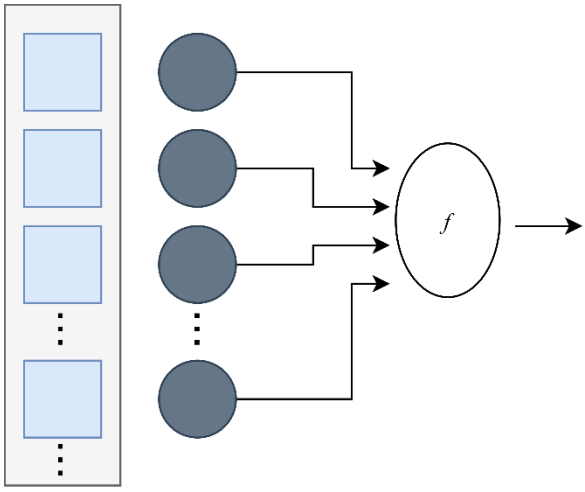


Figure 3. Flat Ensemble Structural Architecture

2.2.1 Base Learner

A base learner can be explicitly constructed for solving many different problems. When a base learner is trained for classifying a new instance with a finite discrete number of outcomes, it is called classification base learner. Given such a learner, there are two more subcategories where a base learner lies between, the first one is when the classification problem is between two possible outcomes, so it is further characterized as binary classification base learner, while when there are more than two possible discrete outcomes the base learner is called multiclass classification base learner. In addition, if the number of possible outcomes ranges between a spectrum of possible outcomes (continuous), then we are dealing with a regression problem, therefore the base learner becomes a regression base learner. Regardless of the type of its output, a base learner can also be further categorized into two categories which are directly associated with the type of data that it handles. The first one is when the type of data has numerical or continuous behavior, taking values from a subset of real numbers, while the second case is when data follows a categorical pattern which take values from a finite number of possible values.

There are many different base learners for dealing with all kind of problems. Based on the holistic research [5] the most popular classifiers within an ensemble (also dealing with concept drift) were the Decision Tree, SVM and Naïve Bayes classifier, collecting 23%, 15% and 14% respectively among the classifiers used. The SVM classifier, which is the acronym of support-vector machine, is a linear model which creates a line or a hyperplane which separates the data into classes. Also, the Naïve Bayes classifier belongs to the family of simple "probabilistic classifiers" based on applying the Bayes theorem. Decision trees are the most common base learner for ensemble learning in a streaming setting for the reason that it is very easy to interpret and visualize them. In a decision tree, each internal node corresponds to an attribute that splits into a branch for each attribute value, and leaves correspond to classification predictors, usually majority class classifiers.

2.2.1.1 Classification Decision Tree

The main goal of a classification decision tree is to produce a function $y = f(x)$ such that it maps the set of all possible examples into a predefined set of class labels. *Given a training set S with input attributes set* $A = \{a_1, a_2, \dots, a_n\}$ *and a target attribute* y *from an unknown fixed distribution D over the labels instance space, the goal is to induce an optimal classifier with minimum generalization error.*[6] A decision tree is built from the root to the leaves based on some splitting and stopping criteria. The decision tree consists of nodes that form a rooted tree, meaning that it is a directed tree with a node called root that has no incoming edges. All other nodes have examples from only one incoming edge. A node with outgoing edges is called internal node and its purpose it to redirect

incoming instances to the next level. All other nodes are called leaves or decision nodes, where new instances are classified by navigating themselves from the root to a leaf, according to the outcome of the tests along the path. In the case of decision trees which handles only discrete attributes, the number of possible outgoing edges of an internal (-test) node are equal to the number of possible outcomes of that specific attribute. On the other hand, decision trees, which also handle numerical attributes, are searching for only one point which splits a given data space into two subspaces. Considering the latter case, there are two different approaches; the first one considers that at every next level of a certain path of the decision tree the set of attributes is reduced by one, removing the attribute which was used at the previous level, while the second one, uses the same attributes over and over again, regardless of the level. Therefore, considering a set of attributes $X = \{x_1, x_2, \dots, x_n\}$ and an attribute x_i used as the splitting attribute at some node of level l, the former approach considers a $X_{new} = X - \{x_i\}$ and the latter $X_{new} = X$. In practice, the first limits the number of possible splits but controls the depth of the tree, while the second makes better refinement of the data space, risking of making deep trees. (See Section 2.2.1.5)

At this point, we can showcase a simple but comprehensive example in order to establish the basic concept of a Classification Decision Tree. In Figure 3, we observe a time snapshot of the training phase of a supervised base learner, where incoming instances are traversed through the tree, resulting to the current state of the tree. The blue node is the root and all the orange ones are the internal (-test) nodes, while the green ones are the leaves which hold the final outcome.

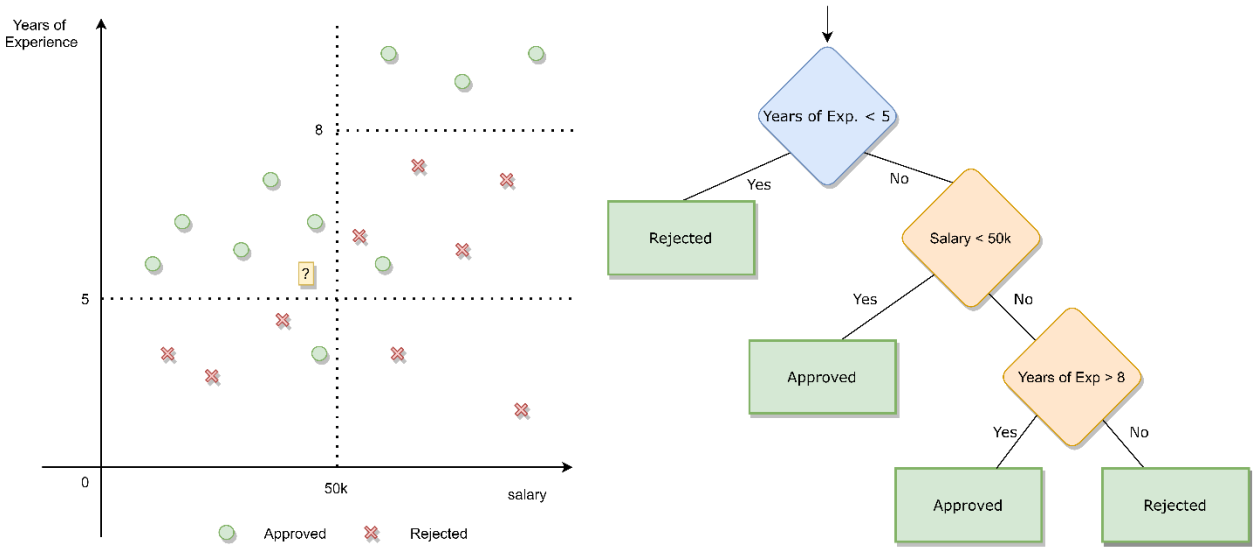


Figure 3. Basic notion of Classification Decision Tree

The use case is as follows: there a couple of employees in a big tech company where some of them requested a promotion. As we can see based on the latest promotions, a decision tree has been built by categorizing the approved and rejected request based on two numerical attributes: years

of experience and current salary. It is obvious that the base learner has been trained quite well as it splits the data space at almost in perfect cut-offs. We can observe that there are cases where the base learner has classifiers incorrectly two positive outcomes as negative. The main question posed is: *What will base learner answer in the case of a new unknown instance?* (Yellow question mark). The new instance will traverse through the tree and will answer based on its respective values of attributes on the questions of each internal node. As we can see the unknown instances has a more than five years of experience and gets below 50k as its main salary, therefore its request will be approved. In the Section 3.5 we will discuss how we can keep track the correct and incorrect answers of our model in order to do structural changes.

ID3 and C4.5 Algorithms

The ID3 algorithm [7] by Quinlan, is the base algorithm of building a decision tree using discrete attributes. Based on ID3, an extension algorithm was constructed, called C4.5 [8] by the same author. C4.5 builds decision trees from a set of training data in the same way as ID3, using the concept of information gain. Therefore, in the case of continuous attributes, the split at an internal node will be binary composed of the following tests $x_i < \theta \; and \; x_i > \theta$. At each node of the tree, C4.5 chooses the attribute of the data that most effectively splits its set of samples into subsets enriched in one class or the other. The splitting criterion is the normalized information gain. (See Section 2.2.1.1.5) The attribute with the highest normalized information gain is chosen to make the decision.

Algorithm 1: C4.5 Algorithm

Input: An attribute-valued dataset D

Output: C4.5 Tree

1. **if** D is "pure" or other stopping condition met **then**
   stop
   end if
2. **for all** a attribute a $\in D$ **do**
   Compute information-theoretic condition if we split on a
3. **end for**
4. a_{best} = Best attribute according to above computed condition
5. Tree = Create a decision node that test a_{best} in the root
6. D_v = Induce sub-datasets from D based on a_{best}
7. **for all** D_v **do**
   $Tree_v$ = C4.5 (D_v)
   Attach $Tree_v$ to the corresponding branch of Tree
8. **end for**
9. **return** Tree

2.2.1.2 Estimating Split Criteria

Entropy and Information Gain

Choosing the correct statistical measure in the process of repeatedly splitting on attributes is equivalent to partitioning the initial training set into smaller training sets until the entropy of each of these subsets is zero and therefore pure. The attribute which minimizes its value at any given point should be selected as the splitting attribute. Entropy is an information-theoretic measure of the 'uncertainty' contained in a training set, due to the presence of more than one possible classification.[9] The entropy of the training set is denoted by E. It is measured in 'bits' of information and is defined by the formula:

$$E = -\sum_{i=1}^{K} p_i \log_2 p_i$$

An important property of entropy is that $E > 0, since\ p_i \in [0,1]$ and that the range of $E = \log_2 |C|$, where C is the number of distinct classes.

In case of a binary classification problems, p_i can be defined as the probability that an instance belongs to the respective class $\{p_1, p_2\}$. So, the general formula can be fitted as follows:

$$E = -\sum_{i=1}^{K=2} p_i \log_2 p_i = -p_1 \log_2 p_1 - p_2 \log_2 p_2$$

Entropy at its own cannot answer the initial question of which attribute is the more appropriate for splitting the original set into two subsets. Here comes Information Gain as a complement of Entropy, that gives the answer. This measure was made popular after the C4.5 decision tree algorithm. Information Gain implies the amount of information gained of a random variable by observing another random variable. One commonly used method is to select the attribute that minimizes the value of produced entropy, thus maximizing the information gain. The Information Gain of an attribute $x_i \in X$ at a node l is the difference between the entropy of the class before and after splitting by the attribute.

$$IG(l, x_i) = E_{start} - E_{end}$$

The entropy of S after splitting on attribute x_i is:

$$E_{end} = E(S, x_i) = \sum_{i} \frac{E(S_\alpha)\ |S_\alpha|}{|S|}$$

Where S_i is the subset of S where x_i has value α. The chosen attribute and value α are the threshold that maximizes the value the Information Gain.

2.2.1.3 Streaming Decision Trees

Streaming Decision Trees, which are also called Incremental Decision Trees are design in order to meet the streaming standards. Unlabeled instances arrive one at a time and need to be rapidly classified into one out of a predefined set of labels. The stream is considered infinite and therefore mining algorithms cannot store many instances into the main memory and consequently can process them only once. After classification, the true label of the sample is considered to be available with which the system's performance can be calculated. As it is mentioned in [10] the main differences between data streams and conventional static datasets include:

- data items in the stream appear sequentially over time and the arrival rate is very rapid (relatively high with respect to the processing power of the system)
- there is no control over the order of incoming items and the processing system should be ready to react at any time
- the size of data may be unbounded
- only one scan of items from a data stream is possible
- data streams are susceptible to change (data distributions generating examples may change on the fly).

In stream mining, the state-of-the art decision tree classifier is the *Hoeffding tree,* introduced by Domingos and Hulten [11]. Traditional decision trees scan the entire dataset to discover the best attribute to form the initial split of the data. Once this is found, data is split by the value of the chosen attribute, and the algorithm is applied recursively to the resulting sub data, to build subtrees. On the other hand, Hoeffding tree is based on the idea that, instead of looking at previous (stored) instances to decide what splits to do in the trees, we can wait to receive enough instances and make split decisions when they can be made confidently.

Hoeffding Tree and VFDT Algorithm

The biggest problem in extending decision trees to data streams is that the measures of attribute importance used to determine the best choice of attributes requires counts or probabilities computed over all of the training data. Clearly, this is not possible when the data is a stream. One solution, proposed by Domingos and Hulten, is to use the Hoeffding bound to estimate when the number of records accumulated at a node is "enough" for a robust decision.

The Hoeffding bound [12] states that, given a random variable r in the range L, and n independent observations of r having mean value $\bar{r}$, the true mean of r is at least $\bar{r} - \epsilon$, where

$$\epsilon = \sqrt{\frac{L^2 \ln (1/\delta)}{2n}}$$

with probability $1-\delta$, where δ is a user-defined threshold probability. The Hoeffding Bound ensures that no split is made unless there is a confidence of $1-\delta$ that a particular attribute is the best attribute for splitting the current node at a specific value. Based on Domingos and Hulten, we assume that we have a gain function G that represents the attribute's importance when splitting a specific leaf node. When the Hoeffding bound is satisfied, the G function is calculated for all attributes and both the best and second-best attributes are chosen to calculate, as follows: $\Delta G = G_{highest} - G_{second\_highest} \geq 0$ .The variable ΔG is recalculated at every satisfaction of the splitting constraints. Authors also add another dimension to the existing problem by saying that if a leaf has processed n training examples and if $\Delta G > \epsilon$, then Hoeffding Bound guarantees with confidence $1-\delta$ that the attribute with the highest information gain is the correct choice since ΔG differs from its true value by ϵ.

The pseudocode of the Hoeffding Tree is shown below and it is based on the Hoeffding's bound.

Algorithm 2: Hoeffding Tree Algorithm

Input: S	is a sequence of examples
$\boldsymbol{X}$	is a set of discrete attributes
$G(.)$	is a split evaluation function
Δ	is one minus the desired probability of choosing the correct attribute at any given node
Output: HT	is a decision tree

Procedure HoeffdingTree (*S, X, G, δ)*

1. Let *HT* be a tree with a single leaf l1 (the root)
2. Let $\mathbf{X_1} = \mathbf{X} \cup \{X_\emptyset\}$
3. Let $\bar{G}_1(X_\emptyset)$ be the $\bar{G}$ obtained by predicting the most frequent class in *S*.
4. **For** each class y_k
5. **For** each value x_{ij} of each attribute $X_i \in \mathbf{X}$
6. Let $n_{ijk}(l_1) = 0$
7. **For** each example $(\mathbf{x}, y_k)$ in S
8. Sort (x, y)into a leaf l using *HT*
9. **For** each x_{ij} in x such that $X_i \in \mathbf{X}$
10. Increment $n_{ijk}(l)$
11. Label l with the majority class among the examples seen so far at l.
12. **If** the examples seen so far at l are not all of the same class **then**
13. Compute $\bar{G}_l(X_i)$ for each attribute $X_i \in \mathbf{X} - \{X_\emptyset\}$ using the counts $n_{ijk}(l)$
14. Let X_a be the attribute with highest $\bar{G}_l$
15. Let X_b be the attribute with second-highest $\bar{G}_l$
16. Compute $\epsilon = \sqrt{\frac{R^2 \ln 1/\delta}{2n}}$
17. **If** $\bar{G}_l(X_a) - \bar{G}_l(X_b) > \epsilon$ *and* $X_a \neq X_\emptyset$ **then**
18. Replace l by an internal node that splits on X_a
19. **For** each branch of the split
20. Add a new leaf l_m , and let $\boldsymbol{X_m} = \boldsymbol{X} - \{X_a\}$
21. Let $\bar{G}_m(X_\emptyset)$ be the $\bar{G}$ obtained by predicting the most frequent class at l_m
22. **For** each class y_k and each value x_{ij} of each attribute $X_i \in \boldsymbol{X_m} - \{X_\emptyset\}$
23. Let $n_{ijk}(l_m) = 0$
24. Return *HT*.

Moreover, the Hoeffding Tree algorithm maintains in each node the statistics needed for splitting attributes. For discrete attributes, a count n_{ijk} which represent the case where the attribute x_i takes the value j and has the label k. The memory needed depends on the number of leaves of the tree and not on the length of the data stream. A theoretically appealing feature of the Hoeffding Tree not shared by other incremental decision tree learners is that it has sound guarantees of performance. It was shown in [11] that its output is asymptotically nearly identical to that of a non-incremental learner using infinitely many examples, in the following sense.

2.2.1.4 Hoeffding Tree Extensions

CVFDT

VFDT does not consider the concept drift problem. An improvement for VFDT was proposed by Hulten et al. [13] called the Concept-adapting Very Fast Decision Tree (CVFDT) algorithm, which addresses concept drift while maintaining similar efficiency and speed to VFDT. The main idea behind CVFDT is to grow alternative subtrees for internal nodes. Whenever there is an internal subtree that poorly reflects the current concept, CVFDT replaces this node with the alternative subtree that has better performance. Depending on the available memory, CVFDT defines a maximum limit on the total number of alternative subtrees that can exist at the same time. If there are alternative subtrees that are not making any progress, CVFDT prunes them to save memory. So, the main idea behind CVFDT is to grow alternative subtrees for internal nodes.

SVFDT

SVFDT (Strict Very Fast Decision Tree) by V.Costa [14], is an extension of the original VFDT algorithm which focuses mostly on the memory consumption aspect of the algorithm. It tries to create more shallow trees reducing traversal times. The main observation of the paper is that although the VFDT has been widely used in data stream mining, in the last years, several authors have suggested modifications to increase its performance, putting aside memory concerns by proposing memory-costly solutions. The main drawback that SVFDT tries to compensate is its accuracy. Its experiments showed that the proposed algorithm obtained similar predictive performance and, in some cases, slight less than its competitors, around the ± 1%, while its size reduced by -65%. Authors also mentioned that according to Krawczyk et al. [10] data stream researches are shifting their focus to ensemble-based solutions. Ensembles can use only weak learners as long as their correlation is low. *However, the use of several base-learners increases memory costs, limiting the use of ensembles.* They also emphasize that in order to deal with memory cost restrictions and managing to keep the same predictive performance, they proposed a new base learner.

The main concept of SVFDT is that despite the fact that a leaf can satisfy the VFDT split conditions (according to the Hoeffding Bound and tiebreak value), can still remain a leaf if SVFDT considers this split unnecessary. When leaves satisfy the VFDT split condition, statistics corresponding to

it are marked with an underscored *satisfyVFDT*. SVFDT has two versions: SVFDT-I and SVFDT-II, with each one of them aiming to a different aspect in the process of a Hoeffding Tree. SVFDT-II is a less strict set of rules that act like a skipping mechanism to speed-up growing. Essentially, what SVFDT does it to keep some historical statistics about entropy, information gain and the proposed by Domingos variable of nmin. For each version of the algorithm, there is a different evaluation function.

For *SVFDT-I*, the function used is implemented by the underlying concept of the 3-σ rule.

$$\varphi(x, X) = \begin{cases} True, & if\ x \geq \bar{X} - \sigma(X) \\ False, & otherwise \end{cases} \qquad (1)$$

where x is the current variable that we are testing and X is the history table of statistics. Based on this (1) the follow constraints are employed every time there is a split attempt.

1. $\varphi(H_l, \{H_{l0}, H_{l1}, \dots, H_{lL}\})$, where the former parameter is the current entropy of l and the latter is a set of all entropies of all current leaves L in the tree, including l (Statement 1)
2. $\varphi\big(H_l, \{H_{satisfyVFDT_0}, H_{satisfyVFDT_1}, \dots, H_{satisfyVFDT_S}\}\big)$, where the latter parameter corresponds to the entropies computed at all S times that any leaf satisfied the VFDT split conditions. (Statement 2)
3. $\varphi\big(IG_l, \{IG_{satisfyVFDT_0}, IG_{satisfyVFDT_1}, \dots, IG_{satisfyVFDT_S}\}\big)$, where IG_l it the IG of the best split feature at l and the latter parameter is a set of the IGs computed all S times that a leaf satisfied the VFDT split conditions (Statement 3);
4. $n_l \geq \overline{\{n_{satisfyVFDT_0}, n_{satisfyVFDT_0}, \dots, n_{satisfyVFDT_0}\}}$, where the former parameter corresponds to the number of elements seen at l and the latter to the average number of elements observed at all S times a leaf satisfied the VFDT split conditions (Statement 4)

For *SVFDT-II*, the evaluation function is:

$$\omega(x, X) = \begin{cases} True, & if\ x \geq \bar{X} + \sigma(X) \\ False, & otherwise \end{cases} \qquad (2)$$

The main difference is that in this case the evaluation function does add as opposed to subtraction. At a split attempt, there is a splitting condition, which when it hold true that the *SVFDT-I* does not need to be updated.

$$\omega\big(H_l, \{H_{split_0}, H_{split_1}, \dots, H_{split_s}\}\big)\ OR\ \omega\big(IG_l, \{IG_{split_0}, IG_{split_1}, \dots, IG_{split_s}\}\big)$$

A note that is posed to SVFDT is that all the forementioned statistics need memory space. *What are the extra memory costs?* The memory costs added to VFDT to compute the constraints 2,3,4 is $O(1)$. Complementary, the memory cost of constraint 1 is $O(L_{max})$, where L_{max} is the maximum number of leaves observed during the tree induction. Also, the time complexity of the first statement of also $O(L_{max})$, while the other have $O(1)$ complexity. *SVFDT-II*, we have an additional time cost of $O(t_{satisfiedVFDT})$, where $t_{satisfiedVFDT}$ is the number of times a leaf satisfied the VFDT split conditions.

The main flowchart of SVFDT algorithm is shown in the follow figure.

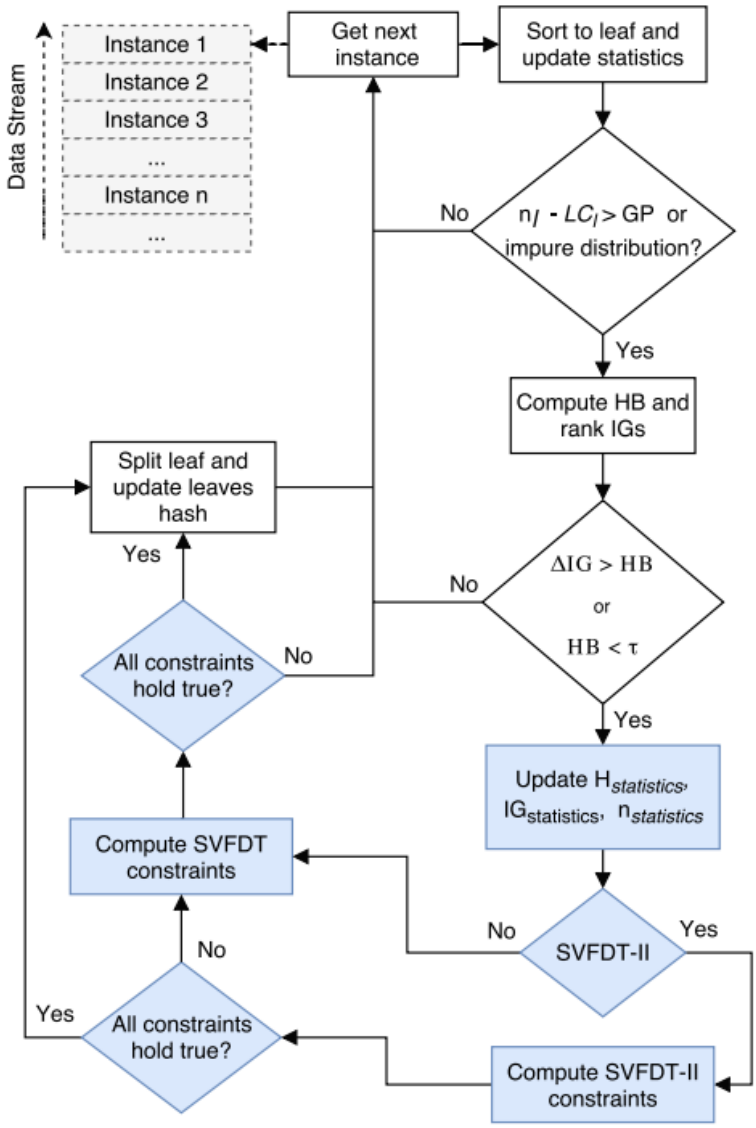


Figure 4 SVFDT diagram. Parts colored in blue denote modifications in the traditional VFDT algorithm

2.2.1.5 Handling Numeric Attributes

Gaussian Approximation

Handling numeric attributes in a data stream classifier, is much more difficult than in a non-streaming setting. Also, continuing in the same context as SVFDT, we are trying to embed smart modifications that have the same functionality as the original VFDT algorithm but in a more efficient manner. For that reason, we implemented Gaussian Approximation for efficiently storing splitting statistics as a further optimization.

This method, presented in [15] by B. Pfahringer, approximates a numeric distribution in small constant space, using a Gaussian (commonly known as normal) distribution. Such a distribution can be incrementally maintained by storing only four numbers in memory, and is completely insensitive to data order. A Gaussian distribution is essentially defined by its mean value, which is the center of the distribution, its standard deviation or variance, which is the spread of the distribution, as long its min and max values for quicker calculation of cumulative density function. The shape of the distribution is a classic bell-shaped curve that is known by scientists and statisticians to be a good representation of certain types of data. For each numeric attribute the numeric approximation procedure maintains a separate Gaussian distribution per class label.

The main thing that we should not forget is that we are under a streaming context and we are not able to do multiple passes over data. Therefore, we are searching for an incremental way that can guarantee at any time the correct statistics of the stream.

Algorithm 3: Numerically robust incremental Gaussian

Input: Value — is the value of an attribute of an incoming instance
Weight — is the value that defines the number of insertions

Output: Variance

1. $weightSum = weight_{first}$
2. $mean = value_{first}$
3. $varianceSum = 0$
4. ***for*** *all data points* $(value, weight)$ *after first*
5. ***do***
6. $\quad weightSum = weightSum + weight$
7. $\quad lastMean = mean$
8. $\quad mean = mean + \frac{value - lastMean}{weightSum}$
9. $\quad varianceSum = varianceSum + (value - lastMean) \cdot (value - mean)$
10. ***end for***

anytime output:
return *mean*
return $variance = \frac{varianceSum}{weightSum - 1}$

The method is similar to this which was described by Gama et al. in the UFFT system[16]. The part of the UFFT system that handles numeric attributes has small differences because it uses the quadratic discriminant which splits the X-axis into three intervals $(-\infty, d_1), (d_1, d_2), (d_2, \infty)$, where d_1 and d_2 are the possible roots of the equation $p(-)\varphi\{(\bar{x}_-, \sigma_-)\} = p(+)\varphi\{(\bar{x}_+, \sigma_+)\}$ where $p(-)$ denotes the estimated probability that an example belongs to class $(-)$ and the $\varphi\{(\bar{x}_-, \sigma_-)\}$ function is the normal distribution of class $(-)$. UFFT selects the d_i that it is closer to the sample means of both classes. The problem of UFFT is that it does not consider that there is a high chance a given attribute will not follow a normal distribution or the means of normal distributions of both classes are so close that there is no root. So, B. Pfahringer extends their approach by searching a set of points spread equally across the range between the minimum and maximum values observed, are evaluated as potential split points. The number of points is determined by a parameter, so the search for split points is parametric. For each candidate point the weight of values to either side of the split can be approximated for each class, using their respective Gaussian curves, and the information gain is computed from these weights.

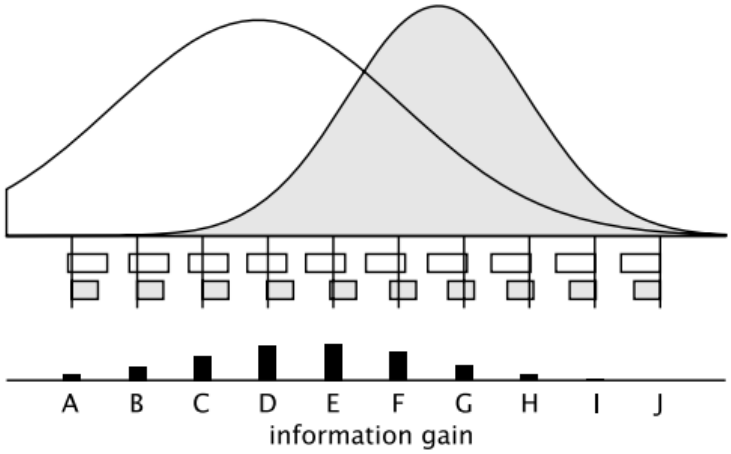


Figure 5. Gaussian Approximation of 2 classes

For example, the class shown to the left has a lower mean, higher variance and higher example weight (larger area under the curve) than the other class. Below the curves the range of values has been divided into ten split points, labeled A to J. The horizontal bars show the proportion of values that are estimated to lie on either side of each split, and the vertical bar at the bottom displays the relative amount of information gain calculated for each split. For the two-class example (the left figure), the split point that would be chosen as the best is point E, which according to the evaluation has the highest information gain.

2.2.1.6 Leaf Estimator

Continuing our analysis, our next stop is the leaf estimator, which plays an important role to the whole system. Here, there two already established classification methods. Majority Voting and Naïve Bayes classifier. In order to classify an unlabeled example, it must traverse through the tree from the root to a leaf. It follows the path based on the answer it gives to every test node at the appropriate attribute and value combination. The simplest classification method is the Majority Voting, where the example is classified with the most representative class of training examples that have reached the leaf. The second classification method uses a naïve-Bayes classifier. The use of the naïve-Bayes classifier at the tree leaves does not enter any overhead in the training phase, because at each leaf we already maintain sufficient statistics to compute the information gain.

2.2.2 Combination Function

A research area, which has not seen a tantamount attention is the *Combination Function* (CF). It constitutes a key component in the construction of a complete Ensemble Learner, as a least thoughtful combination function can have serious repercussions on the performance. With the term of the Combination Function, we essentially mean how outputs from ensemble members are used during prediction. There are many different forms of Combination Functions and each of them has to be selected based on the given problem. The first two categories, which are not widely used because either they have a too specific use case or they provide little performance improvement are the *Rank* and *Relational* combination functions. Rank is used when the base learner can produce a list of predictions at the same time for more than one class label with the respective probability or count. On the other hand, Relational system allows a group of learners to indirectly predict the class label of hard to classify instances by translating base learners' predictions in order to reflect the class label that they most likely represent.

Moreover, another category of combination functions which has the most straight-forward logic among all, is the *Majority Voting*. Despite its admittable simplicity, Majority Voting is a very common strategy [1,17], used as the main combination function by many important algorithms in the ensemble area [18], and often its error rate, Majority Voting Error (MVE), has been set as a performance bases for the assessment of different performance measures when testing multiple algorithms [19].

Based on D.Ruta [19,20], we formulate the MVE and we make it clear that throughout our analysis we refer to a simple combination method on top of binary inputs by assigning 1 to correct classified instance and 0 to an incorrect classified instance.

Given a system of M classifiers: $D = \{D_1, \dots, D_M\}$, let $y_j(x_i)$, where $i = 1, \dots, N$ and $j = 1, \dots, M$ be the binary output of the j^{th} classifier for the i^{th} multidimensional input sample x_i. Let the error rate of such an ensemble member be:

$$e_j = \frac{1}{N} \sum_{i=1}^{N} y_j(x_i) \qquad (1)$$

We can extend the (1) by introducing the weighted error-rate which punishes each classifiers incorrect answer by a factor of β.

$$e_j = \frac{1}{N} \sum_{i=1}^{N} \beta \cdot y_j(x_i) \qquad (2)$$

Also, we can express the ensemble's Mean Error rate (ME) as following:

$$\bar{e} = \frac{1}{M} \sum_{i=1}^{M} e_j \qquad (3)$$

This measure (2) takes the average from individual classifier error rates within the ensemble.

So, given the binary outputs from M classifiers for a single input sample, the final decision extracted from the Majority Voting system is defined as y_i which can be obtained according to the following formula:

$$y_i = \begin{cases} 0, & if \sum_{j=1}^{M} y_j(x_i) \leq \left\lfloor \frac{M}{2} \right\rfloor \\ 1, & if \sum_{j=1}^{M} y_j(x_i) > \left\lfloor \frac{M}{2} \right\rfloor \end{cases} \qquad (4)$$

The MVE can then be formulated as:

$$MVE = \frac{1}{N} \sum_{i=1}^{N} y_i \qquad (5)$$

An important issue is when $\sum_{j=1}^{M} y_i(x_i) = \frac{M}{2}$ and the M is an even number. However, in this work we are not consider it as a problem and without any loss of generality, we assume an odd number for the number of total classifiers. The above analysis serves the purpose of establishing a base threshold with which our implemented Weighted Majority Voting will be compared error wise.

The last two categories which complete the spectrum of the most well-known combination functions are the *Weighted Majority* and the *Classifier Selection* systems.

Weighted Majority is a sound concept of a combination function, which weights classifiers' prediction based on some given criteria. There are many evaluation functions with which weights can be fluctuated during a data streaming context and they are mainly affected based on the given problem. Such a category suits best with evolving data characteristics where the combination function has to react to a number of changes. A basic criterion, used in Weighted Majority, which has a wide range of application in ensemble learning, is based on base learner's latest performance either on the most recent chunk of data in case of batch processing or on the latest instances in case of streaming processing. Here, we have to point out two assumptions that are taking place by us in such situations, firstly, we consider that recency is strongly corelated with relevancy and that we are dealing in a supervised environment. The former is a result of only checking the relative performance based on the latest received instances and the latter due to the prior knowledge of the class or label of the testing instance. In order to judge the base learner's correctness for a given instance, we need to know immediately its true class.

The performance of a base learner, can be interpreted in many different ways and has strong correlation with the quantification of ensemble's pairwise diversity. (See Section 2.2.3). Despite all the proposed metrics which belong to a more sophisticated aspect of a classifier's performance measurement, Error Rate, as formulated in (1), is usually preferred. As we previously discussed, it solely depends on the ratio between the number of classified incorrectly instances and the total instances seen. Some more worth mentioning techniques are about updating the classifier's weight based on whether or not its prediction is agreed with the potential correct prediction of the Majority Vote. Some other techniques are using diversity as a measure in order to remove classifiers having repeatedly the same answers from the ensemble.

Across the last few decades, many papers have been proposed new ways and perspectives for Weighted Majority. One of the firsts were both Littlestone and Warmuth in 1994 who created the well know setting of Weighted Majority (MW) Algorithm. To construct their compound algorithm, a positive weight is given to each of the algorithms (ensemble members) in the pool. The compound algorithm then collects weighted votes from all the algorithms in the pool, and gives the prediction that has a higher vote. Each algorithm begins with a weight of one and if the compound algorithm makes a mistake, the algorithms in the pool that contributed to the wrong predicting will be discounted by a certain ratio β where $0 < \beta < 1$. They have proved that there is an upper bound on the number of mistakes made in a given sequence of predictions from a pool of algorithms $\boldsymbol{A}$ is $O(\log|A| + m)$ if one algorithm makes at most m mistakes. There are many variations of the weighted majority algorithm to handle different situations, like shifting targets, infinite pools, or randomized predictions. The core mechanism remains similar, with the final performances of the compound algorithm bounded by a function of the performance of the

specialist (best performing algorithm) in the pool. Winnow is a similar algorithm, but also increases the weights of experts that predict correctly (Littlestone, 1991). Other equally established algorithms are Online Accuracy Update Ensemble (OAUE) [Brzezinski and Stefanowski 2014] and Adaptive Classifiers-Ensemble (ACE) [Nishida et al. 2005].

There are a number of strategies that can be used to combine different classifiers. The simplest strategy is to select the best performing one. Although the simplest approach may seem like a good idea, it does not guarantee the optimal performance of a given classifier. Instead, it should be designed to select the optimal subset of classifiers and combine them if necessary. Classifier selection techniques are split into two categories, the *static classifier selection (SCS)* and *dynamic classifier selection (DCS).* The former finds the optimal selection combination during the validation set and then uses it during the whole training and testing phases, while the latter selects them online, during classification based on training performance of the unlabeled instances. After the first steps of Woods [5] who proposed the forementioned simplest approach of selecting the best performance member, Giacinto and Roli incorporated classifiers outputs produced during classifications. For reasons of completeness, we have to mention two issues that we have to take into consideration when we select to implement such an algorithm. Firstly, the fact that a hypothetical selection algorithm is able to select the optimal combination and improve the overall performance, does not guarantee that it will remain as it is for future unseen concepts. Secondly, another issue that many faces is that the algorithm's complexity can be exponentially increased especially when choosing complex evaluation criteria and on top of that, if those require checking all the possible subsets of classifiers, then the problem can be more cost-effective than helpful performance wise.

After considering all the above, in our effort to formally present the Classifier Selection issue, we should consult Dymitr Ruta [19] who has properly formulated the given problem. Again, same as our respective analysis for the Majority Voting we consider a system of M classifiers where $D = \{D_1, \dots, D_M\}$ and assigning as: $y_i = [y_{i1}, \dots, y_{iM}\}$ the joint output of a system for the i^{th} multidimensional input sample x_i. Assuming that the binary output is available, we denote as $y_{ij} = 0$ the correct prediction of the j^{th} classifier for a given x_i and respectively $\mathrm{y_{ij}} = 1$ in case of error.

2.2.3 Diversity

Diversity is often identified as one of the building blocks of ensemble-based classifiers. The motivation for the importance of diversity can be intuitively explained using the anthropomorphic example of a group of individuals, such that their opinions are always homogeneous. This group can safely be replaced by any of its members if its only purpose is decision making [21]. But unfortunately, it is not as simple as "augment diversity measure d and the overall accuracy will improve proportionally". In an ensemble system there are two main ways with which diversity can be induced; the first one called input manipulation and second one is called output manipulation. In case of input manipulation, there also two different strategies that boost the system's diversity. The first one is to train different classifiers with different chunks of data (horizontal partitioning) or with different subsets of features (vertical partitioning). The vertical partitioning is more a base learner's structural way to increase diversity while the horizontal partitioning can be achieved through external methodologies, such that we will discuss below.

Decision trees are one of the most suitable classifiers and they are famous because they score higher diversity than all other classifiers. As Breiman [40] mentioned: "Injecting the right kind of randomness, makes RF more accurate". More precisely, the benefits of using randomly selection a subset of the input features is (1) useful when dealing with high dimensional inputs because it reduces the computational cost of finding the best-split feature at each node on every decision tree of Random Forest. On the other hand, this called right randomness also produce some problems that lurk in a first read of the problem. A striking problem is that during the random selection of features among all base learners of an ensemble there is a small probability of a feature not to be selected, while it is unknown whether or not this particular attribute could have the best splitting ability.

Online Bagging

In data stream learning it is infeasible to perform multiple passes over input data and the entire stream cannot be stored. Thus, an adaptation of an ensemble system to streaming data depends on an appropriate online bootstrap aggregating process. In order to implement this application on a streaming context, first we have to understand how bagging works in non-streaming. In non-streaming bagging [13], each of the n base models is trained in a bootstrap sample of size Z created by drawing random samples with replacement from the training set. Each bootstrapped sample contains an original training instance K times, where P(K = k) follows a binomial distribution. For large values of Z this binomial distribution adheres to a Poisson (λ = 1) distribution. Based on that, authors in Oza (2005) [14] proposed the online bagging algorithm,

which approximates the original random sampling with replacement by weighting instances according to a Poisson ($\lambda = 1$) distribution.

$$P(x = k)\binom{n}{k} p^k (1-p)^{n-k} = \binom{n}{k} \frac{1^k}{n} \left(1 - \frac{1}{n}\right)^{n-k}$$

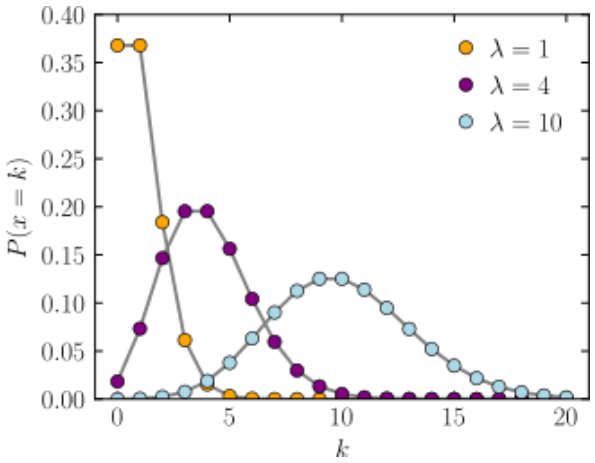


Figure 6. Poisson Distribution for different values of λ

As we can see from the above figure there is approximately 36% of a sample not to be selected, a fact that is not always desirable. Therefore, there are many modern models that can handle more samples and they use Poisson ($\lambda = 6$). The original algorithm of online bagging has the practical effect of increasing the probability of assigning higher weights to instances while training the base models, thus we managed to increase the diversity of the weights and modify the input space of the classifiers inside the Random Forest. However, the optimal value of λ may be different for each dataset.

Algorithm 4: Online Bagging (Stream, M)

Input:	(x, y)	is the Stream of pairs (x, y) where x is a multidimensional feature vector and y the true label.
	M	is the ensemble size
Output:	$\hat{y}$	a stream of predictions for each x

1. initialize base models h_m for all $m \in \{1,2, \dots, M\}$
2. **for** each example (x, y) in Stream
3. **do** predict $\hat{y} \leftarrow argmax_{y \in Y} \sum_{t=1}^{T} I(h_t(x) = y)$
4. **for** $m = 1,2, \dots, M$
5. **do** $w \leftarrow Poisson(1)$
6. update h_m with example (x, y) and weight w

We can see that even the online bagging algorithm follows a test-then-train concept. As we can see the new example will enter the system and will have as weight the result of the respective λ value.

Again, same as the vertical partitioning, there are couple of problems that they are not obvious at first. The problem hears to the name of Out-Of-Bag Error. With bagging, some instances may be sampled several times for some given predictor, while others may not be sampled at all. When used a Poisson (λ = 1) distribution, this means that only about 63% of the training instances are sampled on average for each predictor. The remaining 37% of the training instances that are not sampled are called out-of-bag (OOB) instances. Note that they are not the same 37% for all predictors. This problem should not scare systems which deal with unbounded data streams but make the system lose some momentum when its sole purpose is to be as adaptive and ready as possible.

All in all, either vertical or horizontal partitioning, both provide higher diversity which can only help the system in higher variance of ensemble's member predictions and lower bias as it minimizes the risk of identical base learners. Also, at the same page we can find the use of a concept drift detector, as it always tries to be ahead of the incoming stream and be ready to react to any unexpected changes

2.2.4 Concept Drift Detection in Data Streams

Our last key component in the road of making a complete ensemble-based system is its concept drift detection. From the concept drift component is the ability to learn new concepts (*plasticity*) while retaining previous learned knowledge (*stability*), which is also referred as the stability-plasticity dilemma. Before formulating the problem of concept drifting, it is very important to mention the main assumption that a system makes during its effort for adaptation. Most stream classifiers assume that *recency* is analogous to *relevance.* It is supposed that old instances are associated with previously outdated concepts, while new instances are committed to the most current concept. In addition, another assumption that a system has to make is that the change happens unexpectedly and is unpredictable, although in some particular real-world situations the change can be known ahead of time in correlation the occurrence of particular environmental events. But solutions for the general case of drift entail the solutions for the particular case. In order to formally define the concept drift notion we will use the analysis of J. Gama et al. [23].

Formally *concept drift* between a time point t_0 and a time point t_1 can be define as

$$\exists\, X : p_{t_0}(X, y) \neq p_{t_1}(X, y)$$

where p_{t_0} denotes the joint probability distribution at time t_0 between the set of input variables X and the target variable y. Changes in data can be characterized as changes in the relation of those two components.

Therefore, system's change is produced after identifying one or more of the following changes:

1. the prior probabilities of classes $p(y)$ may change,
2. the class conditional probabilities $p(X|y)$ may change
3. the posterior probabilities of classes $p(y|X)$ may change affecting the prediction.

We are interest to know two implication of these changes (i) whether the data distribution $p(y|X)$ changes and affects the predictive decision and (ii) whether the changes are visible from the data distribution without knowing the true labels (change of $p(X)$).

In order to complete our analysis on what exactly is concept drift and how we can formally identify it, we have to give a visual representation. For that purpose, we can use the work of I.Katakis [24]. In the follow figure we will give an example of an extreme change in order to fully understand the notion of concept drift.

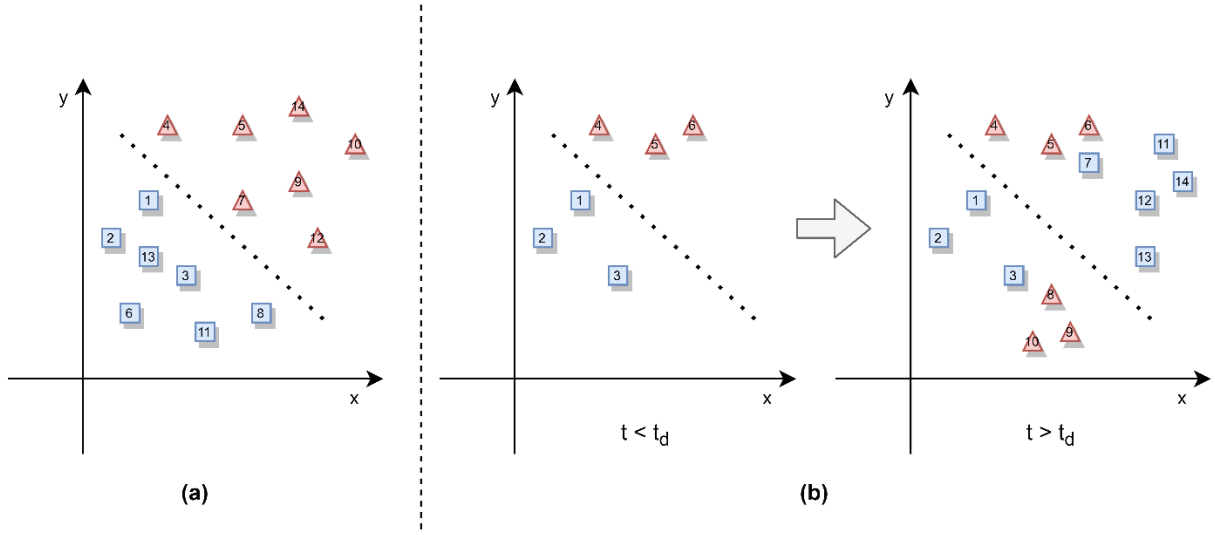


Figure 7. (a) Streaming classification problem without concept-drift. (b) Streaming clasification problem with concept-drift.

In Figure 7(a), we can observe a streaming classification problem where the blue squares are of the same class and the red rectangles are of the opposite class. We can see that even with a simple linear classifier data are separated into two perfect regions. This is not the case with Figure 7(b) which represents a stream classification problem with concept drift in two successive time periods. In the first case before drift time point (t_d) items can easily be separated. However, after time point the concepts of class blue square and red triangle change making difficult for the existing classifier which was trained from previous data to classify incoming instances.

In the above figure we show an extreme paradigm of concept drift. This example lies in the real concept drift subcategory. In general, there are two general types of drift: *real drift* and *virtual drift.* The first one is associated with the change of $p(y|X)$, which essentially means that not only the new data moved in new regions of data space but also their respective classed explored new areas different from the already defined class territories. On the other hand, virtual drift is happening

when only the $p(X)$ while the $p(y|X)$ remains the same. The latter case means that new data explored new data regions respecting the already established class regions. Figure 8 show a visual representation of real (c) and virtual drift (b).

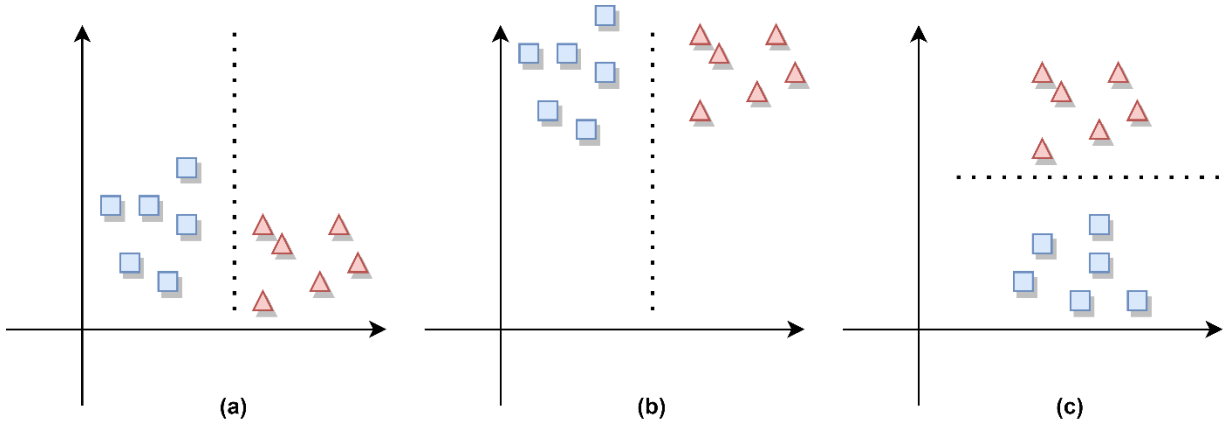


Figure 8. (a) Original Data (b) Virtual Drift (c) Real Drift

After, establishing the concept of data drift during a stream, we now are able to examine the distinction among different types of drift that an ensemble system has to be able to identify and for that reason we have to introduce the extra dimension of time between two different concepts. We will approach the problem similarly as A. S. Iwashita [5] et al. who presented the different types in a compact and representative way. Let C_1 and C_2 be two different concepts generated by two different data sources, and $I = \{i_1, i_2, \dots, i_d, \dots, i_n\}$ a sequence of instance. Instances prior of i_d have draw from a stable source creating C_1 which does not change. After Δx instances, the concept stabilized once more, but in another target concept C_2. The concept among instances $i_d + 1$ and $i_d + \Delta_x$ is drifting from C_1 to C_2. According to Δ_x length, the drift can be called gradual or abrupt. In gradual drift, the two concepts slowly swap; whereas in abrupt drift occurs suddenly.

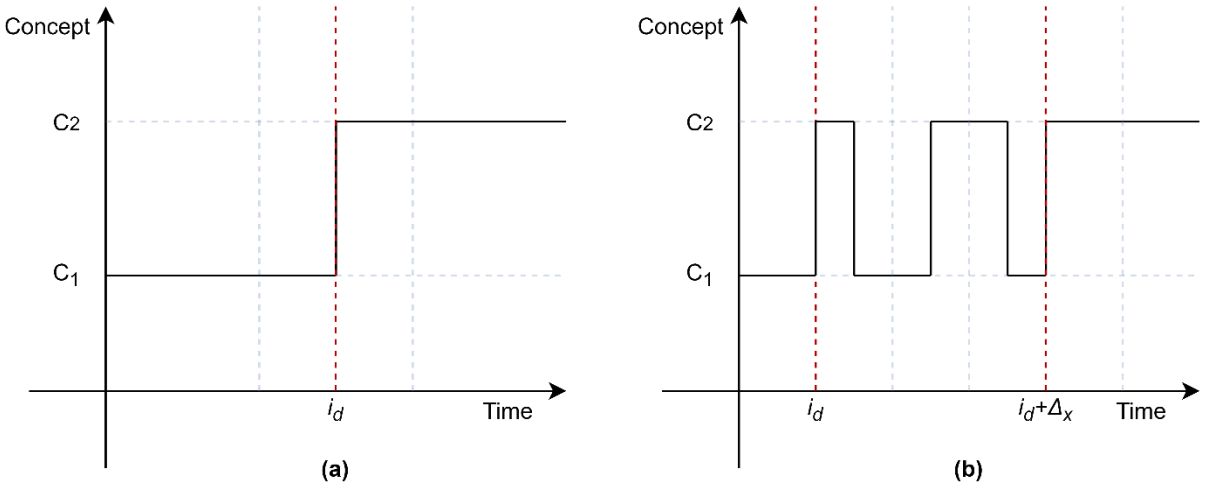


Figure 9. Abrupt and Gradual Concept Drift

2.2.4.1 Detecting changes with Drift Detectors

There are several methods in machine learning to deal with changing concepts. In machine learning drifting concepts are often handled by time windows or weighted examples according to their age or utility. In general, approaches to cope with concept drift can be classified into two categories:

- approaches that adapt a learner at regular intervals without considering whether changes have really occurred;
- approaches that first detect concept changes, and next, learner is adapted to these changes. Examples of the former approaches are weighted examples and time windows of fixed size

Therefore, the main concept of detecting changes during a stream is that after receiving an instance, the online approach updates the classifier; whereas the batch approach waits to receive plenty of instances to start learning. Regardless of the way the system will receive data, incremental learning behaves like online learning with the model update as instances arrive; whereas non-incremental reuses data on learning phase. At the end, active drift detection observes the stream to search for changes and determine whether and when a drift occurs, it warns the learner to take the correct action, while passive drift detection considers drift may occur constantly or occasionally, therefore continually updates the learner as data arrive. [17]

Based on the vast categories of concept drift detectors, in this thesis we will do a special mention to the active ones which means that a system uses an external component while it does not do occasionally structural changes. More precise, when the system's base learner is a decision tree which has adaptive techniques on its own, then the combination of such classifier with a powerful concept drift detector promises both high accuracy but also quick adaptability to changes. So, we are looking for such a detector that is well-established with the minimum distance of detection from the real drift and the higher detection accuracy. The majority of concept drift detectors use the performance of the base learner as an indicator of change. The main assumption is that if a classifier is trained with data from one source, then asymptotically reaches an ideal classifier. But when the source of data changes over time then it has a hard time to cope with the next concept and at some point, is loses its predictive power. Such methods have to be very careful in order to be able to distinguish the change produces by noisy data and the real drift of data. Also, most of those methods use a mechanism of warning and drift signals informing the base learner in order to take the appropriate actions.

The basic idea of drift detection based on tracking classifier's error rate is shown in Figure 10. So, we can see that at first due to the fact that the base learner is immature and it has not given enough time to train the error-rate skyrockets to nearly the 1.0 mark, which means that the classifier is wrong around all the time, but as time passes and the learner manages to adapt to the data the

error rate has a downside trajectory which almost reaches the perfect classifier. From a point and then which is denoted with the term *real drift point* the concept changes and the learner finds it hard to keep up with the new instance which means that it wrongly classifies new data. All this period of time, the drift detector component monitors the base learner's error-rate and when some error-rate value satisfies its evaluation function send a warning signal. It is base learner's business to take advantage to this info and unravel its strategy for such scenarios. After a while, when the error-rate still rises the detector confirms the undergoing drift and again sends a signal.

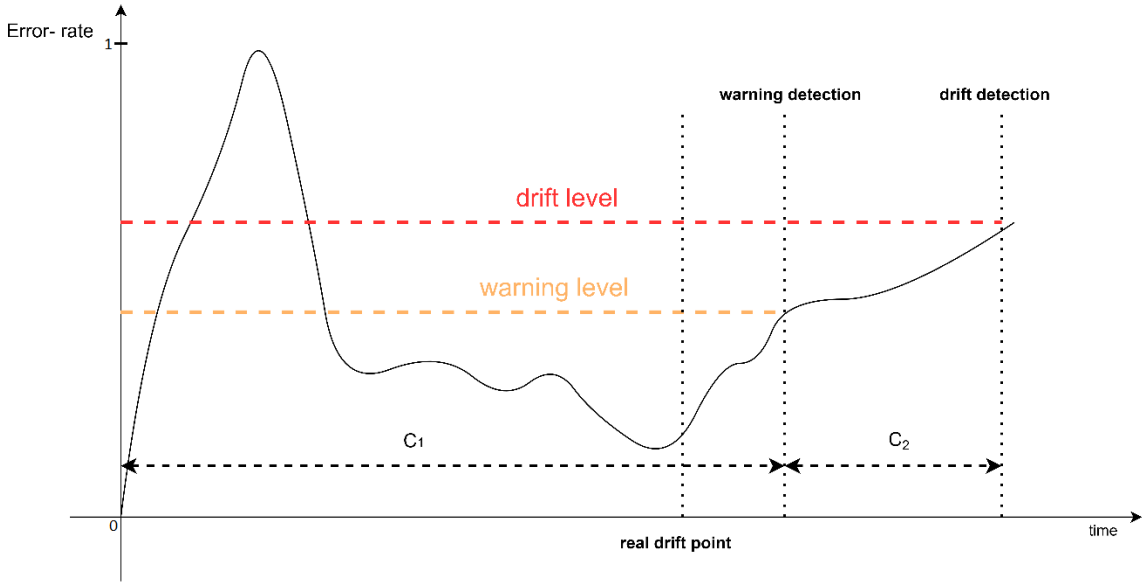


Figure 10. Error fluctuation under concept drift

DDM

DDM (Drift Detection Method) [25] is the most well-known representative of that strategy. It estimates classifier error (and its standard deviation), which (assuming the convergence of the classifier training method) has to decrease as more training examples are received. If the classifier error is increasing with the number of training examples, then this suggests a concept drift, and the current model should be rebuilt. More technically, DDM generates a warning signal if the estimated error plus twice its deviation reaches a warning level. If the warning level is reached, new incoming examples are remembered in a special window. If afterwards the error falls below the warning threshold, this warning is treated as a false alarm and this special window is dropped. However, it the error increases with time and reaches the drift level, the current classifier is discarded and a new one is learned from the recent labeled examples stored in the window. For both levels we keep track the error-rate p_i and its standard deviation $s_i = \sqrt{\frac{p_i(1-p_i)}{i}}$

Its warning level and signal criteria are:

Warning Level:

$$p_i + s_i \geq p_{min} + 2 \cdot s_{min}$$

Drift Level:

$$p_i + s_i \geq p_{min} + 3 \cdot s_{min}$$

The values of constants multiplied with s_{min} in the warning and drift level respectively are not randomly selecting. The denote the level of drift, if the new concept satisfies the drift constraints means that by 99.73% is different from the previous concept.

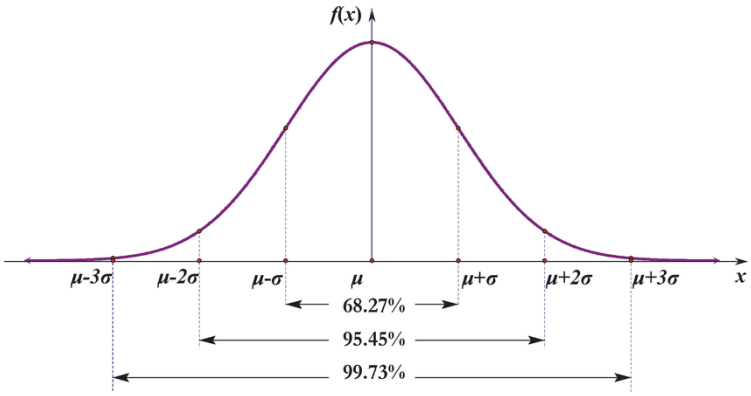


Figure 11. Level of Drift in DDM

The expected result from employing DDM as a concept drift detector in the system is the following:

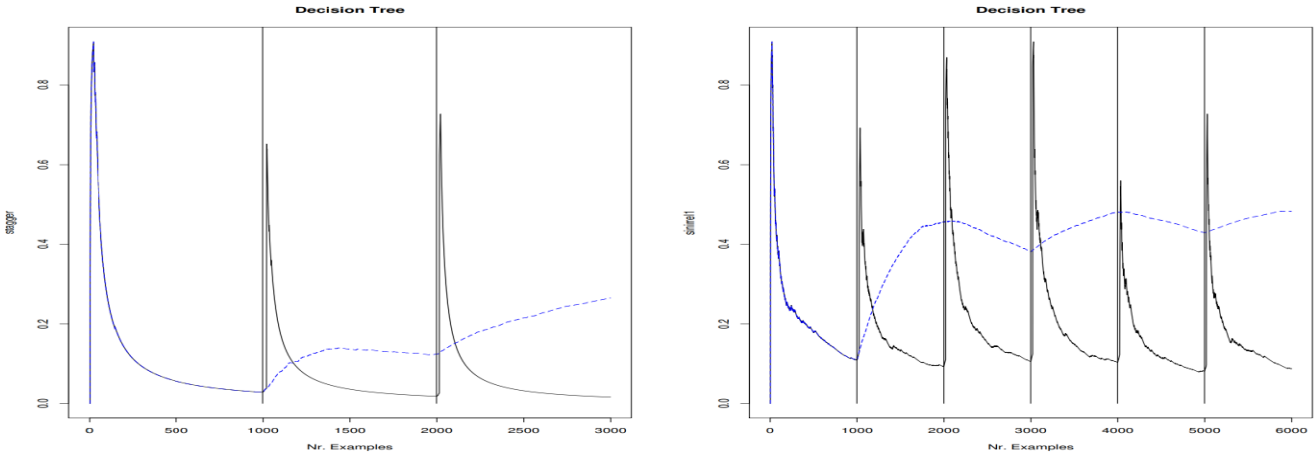


Figure 12. Error rate in Stagger and Sinire1 using DDM.

In Fig.12 we can see two different experiments using DDM in the main system. In both subfigures we see a solid black, a blue dashed line and horizontal lines, which represent the error-rate behavior with and without using DDM and the time points of real drifts respectively. We can

immediately conclude that the error-rate system without DDM always increases as the time passes by. Someone could say that is almost linearly getting worse, whereas when using DDM the error-rate manages to stay relatively low independently of the time. The main problem that this thesis pays more attention is the spikes of error-rate immediately after each real drift. These spikes represent an underlying problem concerning that the adaptive mechanism is not mature enough to handle the new concept. The base algorithm takes into consideration this case by setting an extra constraint which says that the monitoring will start again after $n > 30$ examples. This number takes an arbitrary value which is not supported by somehow, because does not include the type of drift. For abrupt changes 30 examples maybe enough but for gradual drifts may be very little.

DDM was introduces in 2004 and showed the path to other researchers of how to deal efficiently a concept drift scenario. A problem with the most research papers of that era, is that almost none of them test the behavior of their algorithm with big data. So, for a while, it was unknown if their good predictive performance of detecting drifts would remain the same with more data.

RDDM

RDDM which was introduced by R.Barros in 2017 [26], had as its original proposal to overcome deficiencies and thus improve the detections and accuracy results of DDM. This includes their motivation and heuristic assumptions. The main idea behind RDDM is to periodically shorten the number of instances of very long stable concepts to tackle a known performance loss problem of DDM. It is assumed that such a drop is caused by decreased sensitivity to concepts drifts as many thousand instances, it takes a fairly error rate and trigger the drifts. Another symptom of the same problem that the authors noticed is that DDM tends to stay at the warning level for a very large number of instances of the base learner running in parallel, this behavior might also make DDM fail to detect some of the existing gradual drifts, as the base learner is slowly adapting itself to the new concept without a drift detection.

RDDM is essentially a better version of DDM, having less distance between real drift and detecting drift, greater sensitivity (true positive rate) and greater false negatives, but it has much worse false positives. False positives indicate that it detects drifts that they are not happening. Focusing on the biggest in size datasets on their experimental evaluation we can observe that RDDM has on average 17.6 ($\pm$10.8) times more false alarms than DDM. So, in systems that is expensive to create a new strategy every time that a new concept is detected, RDDM is not a viable choice.

2.3 Distributed Streaming Decision Trees and Random Forest

In the Section 2.2, we presented an ensemble system from different perspectives in a streaming context. In this Section, we add another dimension, the distributed computing. In 2019 A. Bifet et al. [27] mentioned that when dealing with large quantities of data, an important **trend** will be how to do online learning using distributed streaming engines, as Apache Spark, Apache Flink, Apache Storm and others. Algorithms have to be distributed in an efficient way, so that the performance of the distributed algorithms does not suffer from the network cost of distributing the data. One of the most used and well-known platforms is SAMOA [28] which provides a collection of distributed streaming algorithms for the most common data mining and machine learning tasks such as classification, clustering, and regression, as well as programming abstractions to develop new algorithms that run on top of distributed stream processing engines (DSPEs). On top of SAMOA is there is the MOA [29] which is the most popular open source framework for data stream mining, with a very active growing community. Both SAMOA and MOA provide solutions at a higher level in relation to the purpose of this thesis and despite the fact there is a selection of open-source code, is not helping at any way thesis's desire to solve at a lower-level problems of already existing algorithms in Apache Flink.

Here, we have to bring one important difference in order to avoid unnecessary misinterpretations. There is a massive difference of a distributed computation and a computation in a distributed environment. A computation in a distributed environment is closer to parallel implementation rather than in distributed computation. This work implements an improved version of Hoeffding Tree under concept drift in a distributed environment. This is totally different with propositions that implement a Hoeffding Tree in a distributed way. Some examples of a distributed implementation of Hoeffding Tree are [30] and [31]. The first one distributes the decision tree's nodes in a distributed environment and manages the information of the best and second-best attribute's information gain using Geometric Method for monitoring a fragmented continuous skyline over distributed streams. [32]

It must also be mentioned that parallel and distributed computing is very important for Machine Learning (ML) practitioners because taking advantage of a parallel or a distributed execution a ML system may: (i) increase its speed; (ii) increase the range of applications where it can be used (because it can process more data, for example). As [33] mentions, Random Forests is a very powerful ensemble method combining a set of decision trees; the Random Forest usually outperforms the single best classifier in the ensemble. More interestingly, the Random Forest classifier outperforms both, confirming that it is an extremely good method for classifying data streams.

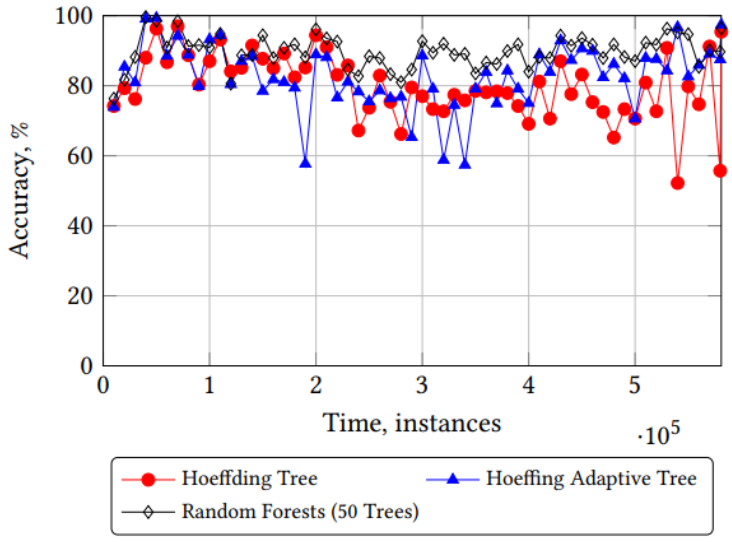


Figure 13. Accuracy comparing the Hoeffding Tree, Hoeffding Adaptive Tree and Random Forest based on the Forest Covertype dataset

Adaptive Random Forest (ARF)

The motivation of ARF by H.Gomes et al. [34] was that there was no random forest algorithms that could be considered state-of-the art in comparison to bagging and boosting based algorithms in the challenging context of evolving data streams. The advantage of ARF is that through their tests they showed that they use feasible number of computational resources while maintain high predictive ability. ARF includes an effective resampling method and an adaptive operator that can cope with different types of concept drift. Their drift adaptation strategy that does not simply reset base models whenever a drift is detected. In fact, it starts training a background tree after a warning has been detected and only replace the primary model if the drift occurs. This strategy can be adapted to other ensembles as it is not dependent on the base model. One of the main points that they emphasize is their parallel version (ARF[M]) of implementation. As they explicitly mention *"Anticipating the results presented in the experiments section, the parallel version is around 3 times faster than the serial version and since we are simply paralleling independent operations there is no loss in classification performance"*. In addition, authors drew the following conclusion *"Since ARF[M] distributes the training and drift detection among several threads it is unsurprisingly the most efficient in terms of CPU time and memory used."* All in all, one small problem with ARF is that their system was not tested with big data, while the maximum number of instances used, were below the 1M mark. But, on their future work proposal they mention that there is a possibility is to implement a big data stream version of ARF.

Chapter 3:

Proposed Solution

Our Approach

3.1 Project Architecture

In this thesis we are considering that we have immediately knowledge of the corresponding class of each input of our system, therefore our ensemble learner lies under the supervised learning framework. Those assumptions make our system vulnerable to real world problems, but implementing more algorithms for dealing with all kind of possible problems will result in a project out of the context of a thesis. What exactly are we implementing?

Firstly, we will have a fixed size Ensemble of homogeneous incremental learning classifiers using the main principles of Hoeffding Tree and Hoeffding Bound. This ensemble is essentially a *Random Forest* and is implemented in parallel in a distributed environment, at *Apache Flink*. We further implement two improvements on top of the base model. The first one is associated with handling numerical attributes using *Gaussian Approximation* of maintaining the necessary statistics for finding the best splitting attribute-value pair. So, instead of keeping instances, which can be proved fatal and definitely is unfeasible in a streaming context with unbounded data, we are making a Gaussian approximation for each attribute. The second optimization serves the purpose of handling the potential limited resources. In a distributed environment where decision trees are implemented in parallel there is a high chance that more than one base learner will end up in the same machine sharing both computational and memory resources. This is inevitable, when the number of models is greater than the cores and machines of the available cluster.

Therefore, here comes a challenge that thesis has considered; situations were multiple models are deployed with lower parallelism. For that purpose, we also implemented the second version of *Strict Very Fast Decision Tree (SVFDT-II)* in order to reduce the memory consumption of our system.

3.2 Resampling

Secondly, this thesis has noticed a problem that is produced by the parallel execution of models. We have to mention here that in this thesis we are using the original methodology of *Online Bagging*. As we have already mentioned only 63% of the original data, has a returned probability different to zero. Suppose the following scenario; in a parallel implementation, the same incoming

instance has to be distributed to all available classifiers for either training or testing. This process of distribution in Apache Flink is performed using hashing, which generally is a very expensive function. So, instead of sending the same instance to all base learners and each one of those deploy the online bagging locally, we have moved the online bagging outside of the internal of every classifier into a separate module which repeatedly calls the Poisson distribution based on the ensemble's size. Hence, we have saved a good amount of unnecessary data transactions.

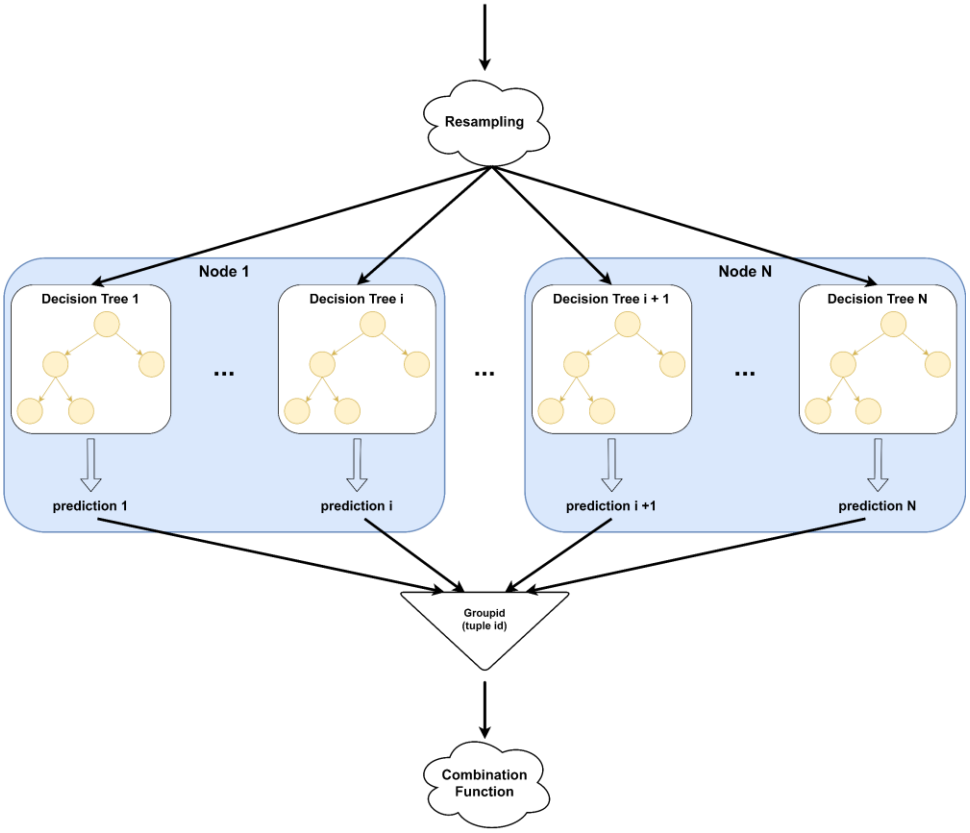


Figure 14. Distributed Random Forest Abstraction

3.3 Base Learner

Thirdly, in our background analysis we mentioned a problem that the majority of proposed algorithm ignore. Big data are usually referred to as data that is so large or complex that it's difficult to process using traditional methods. The last couple of years we are working in an era that every proposed system has to consider big data scenarios. Most of the algorithms had tested their system from around 100k to 3M instances. We further noticed that it is unfeasible to have an incremental base learner whose size increases proportionally to the size of incoming data. Despite the fact that we implemented SVFDT-II and we managed to reduce the size and rate of base learner's growth, we did not succeed to disengage its growth from the amount of incoming data. If some of the proposed implementations tested their systems with data more than 50M (relatively big data), they will also conclude to the fact that while the base learner does not become

better, its size increases almost linearly. So, we propose a new variation of the base learner for periods of *"data drought"*. Data drought indicates that data fed to the system don't "carry" enough information for the base learner to get better and increase its predictive ability. The main point of our proposition is that we periodically check the accuracy of the base learner. If we notice that the base learner is not getting better or worse, we create a window; if the window's size is greater than a user-defined threshold, then we stop the splitting process while we continue to update the internal nodes of the base learner by inserting new instances. Once the accuracy changes behavior, the window is dropped. We can conclude that the base learner's performance remains stationary when its accuracy fluctuates around its mean value for a long window size.

In order to formulate our proposition, we need to consider a vector $I = \{i_1, i_2, \dots, i_n\}$ of incoming instances. Because we are implementing the test-then-train methodology, there is continuous update of its classification performance. As each new instance is fed to the system, it contributes to the cumulative weight. We define as an answering vector $o = \{o_1, o_2, \dots, o_n\}$ where $o \in \{0,1\}$. Then the weight of a base learner at a time point t_{now} is

$$w_{t_{now}} = \frac{1}{t_{now}} \sum_{t=0}^{t_{now}} o_t$$

which denotes the ratio of correctly classified instances to the total number of instances seen. We denote as w_t the produced weight of the base learner at timepoint t after seen the instance i_t

We also keep the approximate mean value ($\overline{w}$) and the standard deviation $\sigma(w)$ of the weight vector with time and memory complexity of $O(1)$. Therefore, the constraint used is:

$$\overline{w} - \sigma(w) \leq w_t \leq \overline{w} + \sigma(w)$$

This constraint is loose enough not to trap the base learner into a continuous pipeline, but too strong to guarantee that if its weight is fluctuating inside area defined by the upper and lower bounds then there is no reason to consider any splitting process. So, what exactly are we expecting to happen with our proposition? Firstly, we need to understand the power of data. All, modern machine learning models are data driven, which means that the structural process is adapted on the behavior of data. Although, a base learner's goal is to create the best possible splitting criteria there are numerous occasions which it is impossible to do so, because data are distributed in the data space in such a way that there is always room for improvement. This is exactly the problem that we are trying to tackle; scenarios where a decision tree is trying to find the ideal criteria by continually splitting its nodes (deeper tree) when at the same time data are not able to be further separated. Such a scenario can be evident with numerous ways, we selected to monitor the base learner's performance because it was the most simple and straightforward. Consider the following scenario, which is depicted in the following figure. At first the error-rate is relatively high because the base learner is not mature enough as it has not been trained with enough data. After some relatively little time the error-rate drops in some acceptable level. Due to the kind of data, the learner does not manage to get better. If the length of its incapability is proportionally

great to length from the start of its training, then system detects that there is time to start a window where the splitting process has to stop. We have to mention that the training is still active and new instances are still traversing the existing tree and update the necessary statistics.

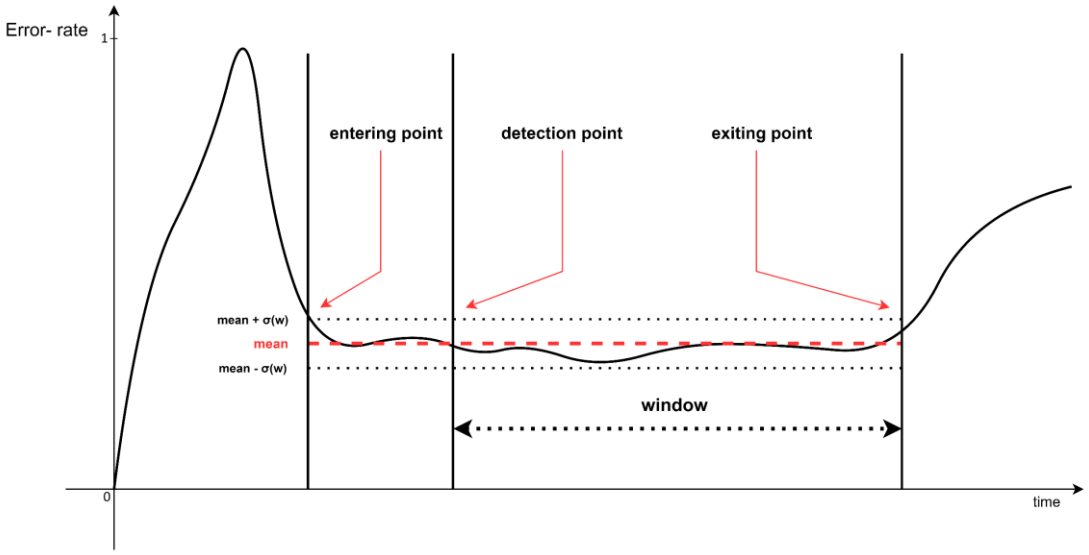


Figure 15. Base Learner proposition

We can support our proposition through some extensive experiments performed in a single base learner based on the most well-known datasets. (See Section 5.4.2)

3.4 Combination Function

In addition, we will keep tracking the classifiers performance by assigning them a weight. This weight we be calculated based on each learner's error rate during both the training and testing phase. (Error rate = 1 – Accuracy). We will conduct *Weighted Voting over Selected Classifiers*. We are selecting only the *top-k classifiers*, where again k is a user defined parameter.

The reason we have extensively explained the different categories of Combination Function in Section 2.2.2 has to do with the fact that we are implementing an approach coming from the union of Weighted Voting and Classifier Selection category. Also, we use the basic Majority Voting error rate as a comparison base of our algorithm.

3.5 Concept Drift Detector

Last but not least, we will consider that our data distribution is changing during the stream, hence, in order to manage that, we are implementing a Concept Drift Detection system. Our system consists of *one well-established concept drift detector (DDM)*. DDM provides a twostep detection points. The first one is the warning signal and the second one is the concept drift signal. With the former, the system will create a *Background Tree* and with the latter it will replace the existing one, such as ARF. (See Section 2.3) In case of a false alarm, which happens when the warning signal cannot hold its value, the Background Tree is deleted. The main thing that we are noticing is that there is no actual drift phase. If we assume that we are in a stable phase and a warning signal comes, then we are entering in a warning phase and subsequently when a drift signal comes, we are back at a stable phase, this time newer's one.

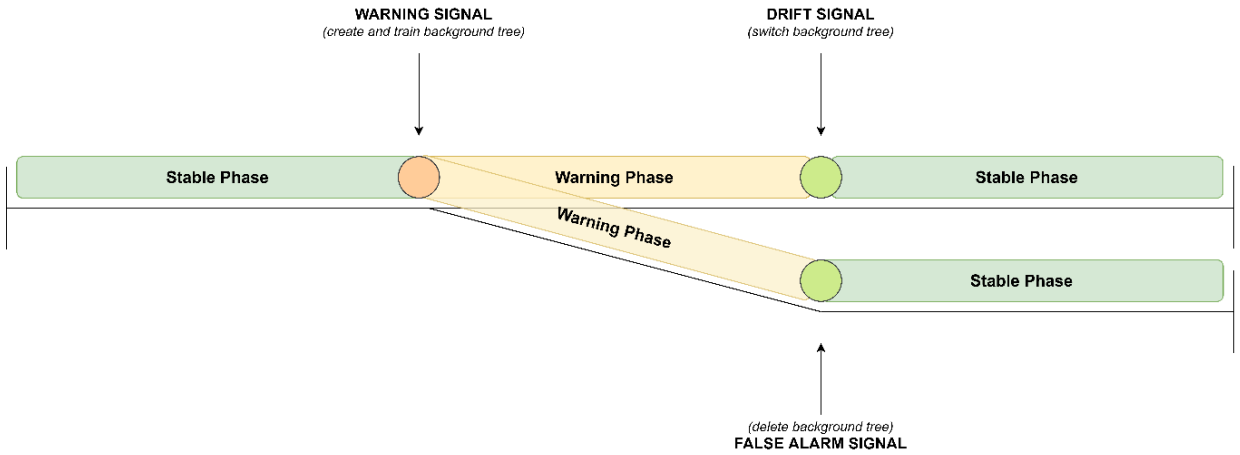


Figure 16. DDM basic signals concept

Recall in Section 2.2.4.1 that we made a special reference in the DDM's error-rate "spikes". For that reason, we are introducing a drift phase. During the drift phase we will replace the existing decision tree with the background one only when the latter has greater accuracy (or lower error-rate), otherwise we will keep the same base learner. As we have already discussed, DDM has an arbitrary number of 30 examples to deal with such a problem which does not take into account the length and magnitude of change.

Therefore, the proposed concept is:

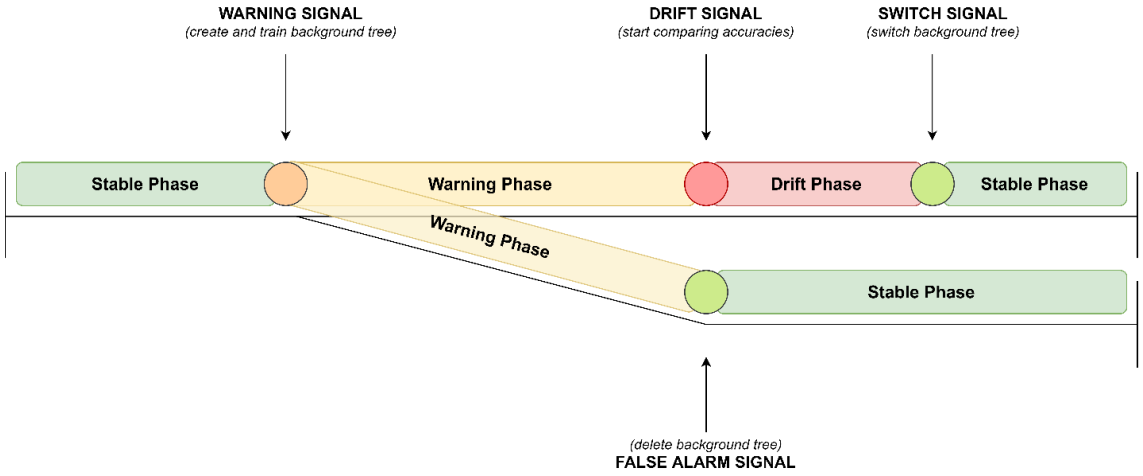


Figure 17. Concept Drift proposition

Opposed to DDM signaling system, in our case when we encounter a drift signal, we start comparing the performance of the rival trees. When the performance of the background tree is greater than the already existing one then we generate a switch signal which means that the replacement method is now ready to be executed. With such a proposition we tackle the problem that DDM can be replaced with a random generator in its transitional phases. We, as a system observer, care only for the overall system's performance and we want to be ensured that our any time testing will have a guaranteed high subsistence. Please, consult Section 5.4.3 where we analyze the results of our proposition.

Chapter 4:

Implementation

4.1 Apache Flink Overview

Apache Flink [35] is a framework and distributed processing engine for stateful computations over *unbounded and bounded* data streams. Flink has been designed to run in *all common cluster environments*, perform computations at *in-memory speed* and at *any scale*. The Flink runtime consists of two types of processes: a *JobManager* and one or more *TaskManagers* (also called *workers*). The *Client* (our implementation) is not part of the runtime and program execution, but is used to prepare and send a dataflow to the JobManager.

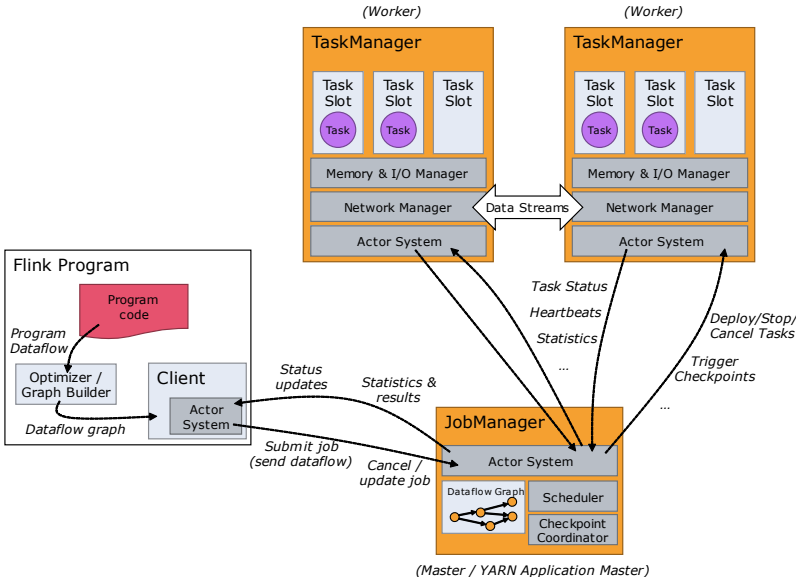


Figure 18. Apache Flink Architecture

For distributed execution, Flink *chains* operator subtasks together into *tasks*. Each task is executed by one thread. Chaining operators together into tasks is a useful optimization: it reduces the overhead of thread-to-thread handover and buffering, and increases overall throughput while decreasing latency

State

The most powerful concept of Apache Flink is its state. Every non-trivial streaming application is stateful. At a high level of abstraction, we can consider state as a snapshot of an application (operator) at any particular time which remembers information about past inputs/events, which

can affect the future output. A system using state will know everything to what has happened in the application till a particular point of time.

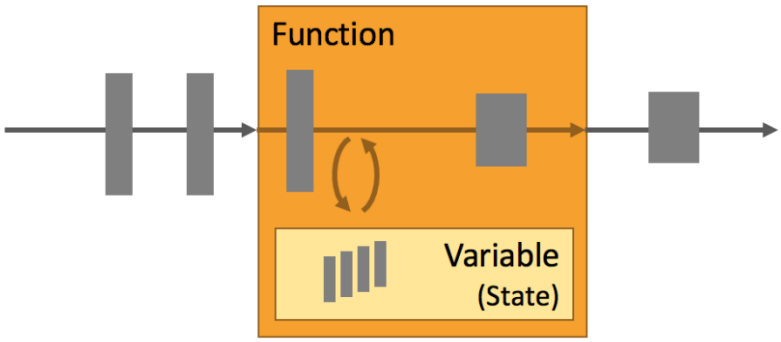


Figure 19. Apache Flink State Abstraction

A saved state can be used in any of the following ways.

- To search for certain event patterns happened so far.
- To train a Machine Learning model over a stream of data points. In this case state will hold the current version of model parameters.
- To achieve fault-tolerance through checkpointing. A state helps in restarting the system from the failure point. In case of any failures, if our system is fault tolerant and if we have saved a state of that application, the we can restart processing exactly from the same checkpoint where the system got corrupted.
- To rescale the jobs and to increase parallelism in a job.

Kafka Connector

Apache Kafka [36] is an open-source distributed event streaming platform.

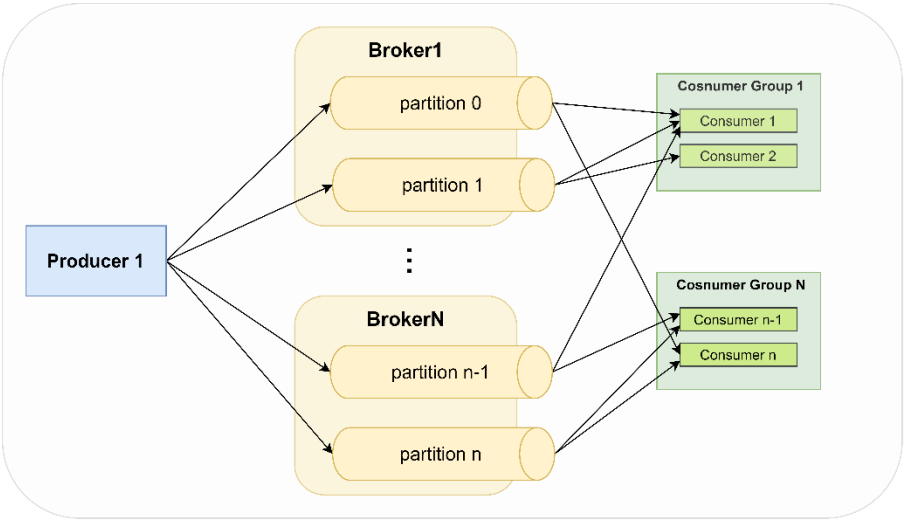


Figure 20. Apache Kafka Architecture

Apache Kafka provides:

- Hight Throughput: Deliver messages at network limited throughput using a cluster of machines with latencies as low as 2ms.
- Scalability: Scale production clusters up to a thousand brokers, trillions of messages per day, petabytes of data, hundreds of thousands of partitions. Elastically expand and contract storage and processing.
- Permanent Storage: Store streams of data safely in a distributed, durable, fault-tolerant cluster.
- High availability: Stretch clusters efficiently over availability zones or connect separate clusters across geographic regions.

4.2 Proposed Implementation

4.2.1 Project Architecture

Our implementation consists of an Apache Kafka Source and Sink as the basic components to read and write data. There is a separate component of Data Sampling which implements the Online Bagging. Data are distributed based on the Hoeffding Tree id and end up to the each responding machine and therefore to each pair of learner and concept drift detector. Each learner is trained or tested by incoming data while on the same time the Concept Drift Detector monitors its performance. Testing data are distributed to every learner component and their prediction are aggregated by a given rule in order extract the final prediction. In the Appendix Section you can further see our system's flowchart (parts of it, will be used across our presentation of the proposed implementation.

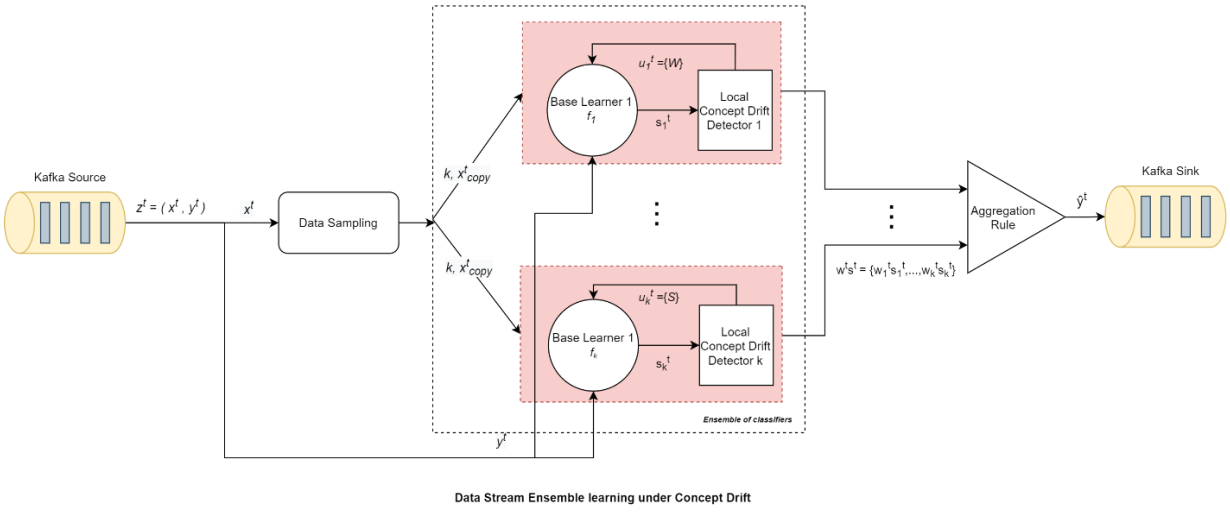


Figure 21. System's Architecture

Source and Sampling Component

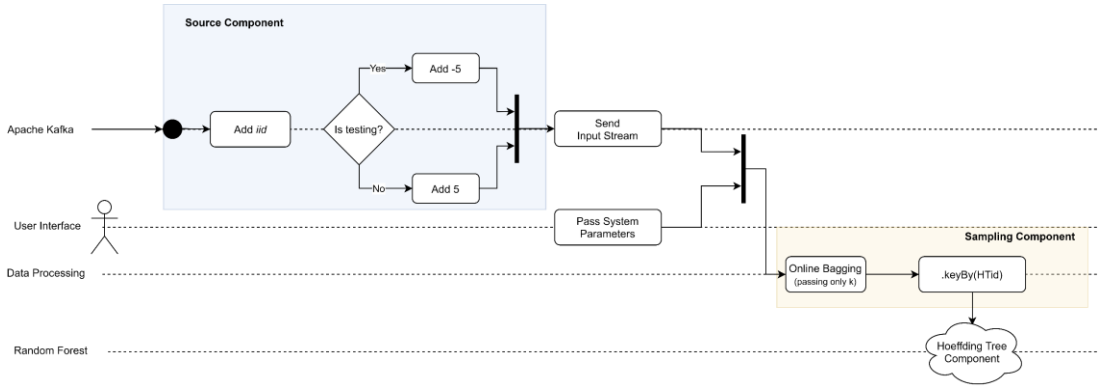


Figure 22. Source & Sampling Component Abstraction

Source Component

Source Component is responsible for preprocessing data by changing its format and making them ready for consumption from our system. In this implementation we can handle only numerical data with either distinct or continuous data. Each tuple of data has to be comma separated with the class label at the end. We do not restrict data for their number of feature or the type of representing their class label. A sample data can be of the following structure.

$$< feature_1, feature_2, \dots, feature_n, label_{true} >$$

Considering that our system is running continually, we have implemented such as system that receives only training data and some periodical tests from the user. More precisely, we consider that the user quires our system an unknown number of times at arbitrarily time intervals. In order to create such a scenario, we distinguish incoming data by assigning one unique identical number. In the current implementation we read from input files but the source can be of any kind (real-time events). If an instance is considered a training one, we assign to it the number 5 otherwise -5. Also, in order to simulate user's queries, we randomly select along our input data random instances that have as their purpose to test our system. We are implementing an ensemble system, so we need an aggregate id in order to perform our combination function. We achieve that by assigning to each incoming instance an incremental id number ($iid - instance\ id$)

Training instance:

$$< feature_1, feature_2, \dots, feature_n, label_{true}, 5, iid >$$

Testing instance:

$$< feature_1, feature_2, \dots, feature_n, label_{true}, -5, iid >$$

Finally, the Source component is responsible to a unique internal Kafka Topic from which the main implementation will receive data.

Sampling Component

The Sampling Component is responsible for identifying the input data and implementing the algorithm of Online Bagging. We can observe that in line 2 if the purpose of the incoming instance is testing then we distribute it to all available member of the ensemble as we need the opinion of each one of them. In addition, we see in line 5 that if an incoming instance is a training, then we iterate through all member by extracting a sample from the Poisson distribution. The Sampling Component distributes training data to a subset of the ensemble based on whether or not its w value is greater than 0. Based on the m value each instance is distribute to the corresponding Base Learner Component. We take advantage the property of the *keyBy* operator that the Apache Flink provides us. *KeyBy* operate guarantees that every instance grouped by the same key will enter to a *KeyedStream* that has its own computation.

Algorithm 5: Sampling Component

Input:	stream	String type stream of pairs (x, y)
	M	ensemble size
Output:	<stream, m, w>	a stream of predictions for each x

```
1.  Extract purpose id from stream
2.  if purpose id equals -5 then
3.      for every member m in M do
4.          output stream ← <stream, m, 0> //0 denotes to No-Weight
5.  if purpose id equals 5 then
6.      for every member m in M do
7.          w ← Poisson(1)
8.          if w > 0 then
9.           output stream ← <stream, m, w> // w for Gaussian Approx.
10. return output stream
```

Base Learner Component

The Base Learner Component is essentially our machine learning model[1] and is incubated by a StatefulMap function, or otherwise by a state. This component is responsible for creating, updating (training) and testing our base learner as long as its background base learner (in case of concept drift). Initially, the state and therefore the Base Learner Component is empty. In this case we create the main Hoeffding Tree (HT). Moreover, we have adopted the notion of "age of maturity". It is a user defined value and it serves the purpose of giving the HT some time in order

[1] Please note that the Hoeffding Tree component is basically a separate model in a unique machine, while the ensemble is distributed implemented.

to enter to a stable phase. In this phase we only accept training instances as we assume that we are not ready to answer to testing queries. Our Hoeffding Tree creating follows the exact same parameters as the original algorithm. In case the period of "age of maturity" is completed, we enter to the next phase. In this phase we are accepting both training and testing instances. We also have adopted the Test-then-Train methodology which firstly traverses the tree until the target leaf, asks the majority class of this particular node and updates the HT's weight based on the correctness of the answer. Secondly, it takes again the same path and updates all the necessary statistics of the target node.[2] The only time we collect the output is in case we have received a testing instance. Algorithm 6 showcases the above analysis.

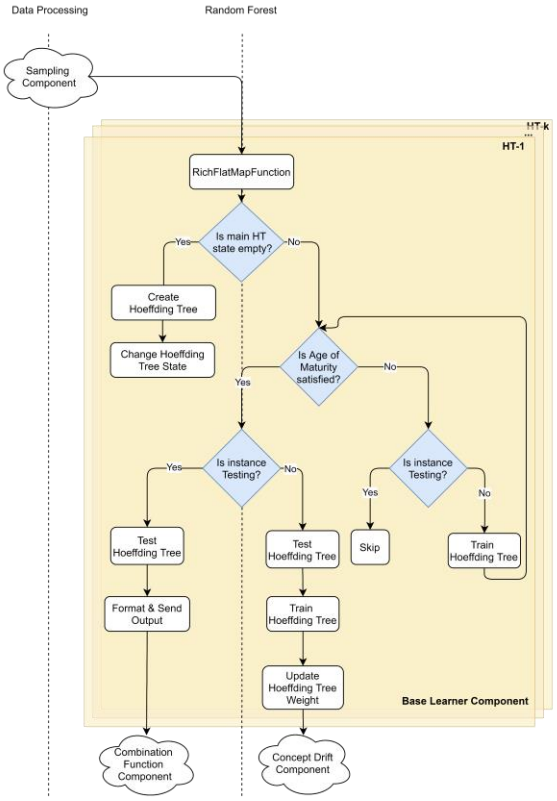


Figure 23. Base Learner Component

Algorithm 6: Hoeffding Tree Component

Input: (x, y, id, w) a tuple of features, true label, HT id and its weight

grace_period is the user defined parameter for age of maturity

Output: output stream a tuple of instance id with the corresponding prediction, the HT id and its weight

```
1.  if state is empty then
2.      Create Hoeffding Tree()
3.  else:
4.      if age of maturity < grace_period then
5.          if instance is Training then
6.              Test-then-Train Hoeffding Tree()        // Update HT weight
7.          if instance is Testing then
8.              Skip instance
9.      else:
10.         if instance is Training then
11.             Test-then-Train Hoeffding Tree()
12.         if instance is Testing then
13.             Test Hoeffding Tree()
14.             output stream ← <instance id, prediction, HT id, weight>
15. return output stream
```

[2] In the first traversal, beyond the returned prediction (majority class), we return the target Node in order to save an extra second traversal.

The output needed from the Combination Function Component must contain the instance id with which we will aggregate our predictions, the actual prediction a long side Hoeffding Tree's weight and id. Now, as we discussed in Section 3.3, we have made some modifications in the basic VFDT algorithm. Therefore, we will present both Create, Training and Testing pseudocodes.

Create Hoeffding Tree

During the creation of the Hoeffding Tree we make some assumptions correlated to the problem we solve. Firstly, we have prior knowledge of the number of attributes with a result to create as many HashMaps as the number of attributes. Secondly, we know that we are dealing only with a binary problem so in both cases of maintaining the required statistics and the label counts we need only two. (0: Class0, 1: Class1). Thirdly, in order to select the required random features, we use the algorithm of Reservoir Sampling, instead of using some specific strategy.

Algorithm 7: Create Hoeffding Tree

Input:	Max:	is the number of how many features we have to select from
	m_features:	is the number of the size of the random subset of Max
	max_examples_seen:	is the number of examples between checks for split
	delta:	one minus the desired probability of choosing the correct feature at any given node
	tie_threshold:	is the number between splitting values of selected feature for split
Output:	root of Hoeffding Tree	

1. **for** *each attribute in m_features* **then**
2. Create a HashMap for statistics
3. Create a HashMap for label counts
4. instances_seen ← 0
5. correctly classified ← 0
6. weight ← 0
7. InitializeRoot (m_features, max_examples_seen, delta, tie_threshold)
8. Reservoir Sampling (m_features, Max)

As we have already mentioned we have implemented the Gaussian Approximation for keeping in track with the feature's changes. The HashMap structure used is the following: for each feature and for each class we maintain four values (sum of weights, mean, sum of the variance, min and max value)

Test Hoeffding Tree

A Hoeffding Tree has to keep track with its progress and have to broadcast its information to the State. In order to do that it needs to contain the number of correctly classified tuples, the weight and the number of seen instances. For defining at each Hoeffding Tree (h) the weight (w_h) we have to take the ratio between the correctly classified tuples (c_h) and the total tuples seen (n_h), $w_h = \frac{c_h}{n_h}, where\ c_h < n_h$.

Algorithm 8: Test Hoeffding Tree

Input:	node:	is the root of the Hoeffding Tree
	input_sample:	is an array of values of the corresponding attribute
	purpose_id:	is the id for identifying the different tuples
Output:	predicted_value	is the prediction from the Hoeffding Tree
	targetNode	is the node after the traversal. Used by the training algorithm

```
1.  filtered_input ←Filter input_sample based on m_features
2.  if purpose_id is a testing then
3.      predicted_value, targetNode ← TestHT(node, filtered_input)
4.  else:
5.      instances_seen ← instances_seen + 1
6.      predicted label, targetNode ← TestHT (node, filtered_input)
1.      if predicted label is equal to true label then
2.          correctly_classified ← correctly_classified + 1
3.      UpdateWeight(correctly_classified, instances_seen)
4.  return predicted_value, targetNode
```

As we have discussed previously, we need to use the instance id for distinction between testing and training ones. In Line 3, TestHT function traverses through the HT by comparing its corresponding splitting attribute, value pair until in sorts itself to a leaf node. Then it returns the label of the target node. In Line 10, UpdateWeight just update the existing HT's tree with the new values.

Train Hoeffding Tree

While we train the Hoeffding Tree we deploy both the SVFDT-II constraints and ours. We have already mentioned that there is a case that Hoeffding Tree's constraints are satisfied by the SVFDT-II ones not. Essentially, with both implementations we have enclose the basic VFDT constraints. In the Appendix Section there is a flowchart that represents the following pseudocode.

Algorithm 9: Train Hoeffding Tree

Input:	node (l):	is the target Node of the Hoeffding Tree from Testing
	input_sample:	is an array of values of the corresponding attribute
	w	is the instance weight from Online Bagging
	DS	is the value that indicates data stagnation
Output:	rootNode	is the HT's root

1. filtered_input ←Filter input_sample respectively to m_features
2. weight ← get Hoeffding Tree's weight
3. **if** $\overline{weight} - \sigma(weight) < weight < \overline{weight} + \sigma(weight)$ **then**
4. window_size ← window_size + 1
5. **else:**
6. window_size ← 0
7. **if** *window_size / instances_seen* > *DS* **then**
8. InsertNewSample = { Update Gaussian Approximation metrics using w; Update label counts; Update *nmin* }
9. **else:**
10. $n_l \leftarrow$ number of elements in node l
11. $n_{l\ last\ check} \leftarrow$ number of elements in node l from the last check
12. **if** *class at* l *is impure and* $n_l - n_{l\ last\ check} >$ *max examples seen* **then**
13. $IG_{best}, IG_{second\ best} \leftarrow$ the highest and second highest $IG(.)$
14. **if** $(IG_{best} - IG_{second\ best}) > HB\ or\ HB < \tau$ **then**
15. $\overline{H}$ and $\sigma(H)$ using $H_{statistics}$
16. $\overline{IG}$ and $\sigma(IG)$ using $IG_{statistics}$
17. $svfdt\_ii\_constaints \leftarrow H_l \geq \overline{H} + \sigma(H)\ or\ IG_{best} \geq \overline{IG} + \sigma(IG)$
18. **if** $(svfdt\_ii\_constaints)$ **then**
19. Split node l into l_{left} and l_{right}
20. InsertNewSample in node the correct child node
21. **else:**
22. InsertNewSample in node l
23. **else:**
24. InsertNewSample in node l
25. **else:**
26. InsertNewSample in node l
27. **return** rootNode

We can see in Line 3 that if the weight fluctuates around its mean value then we create a window. If at any point this condition is not satisfied then the window is dropped. In Line 17 and 18 we see the exact same functionality presented in the SVFDT-II proposition. It evident enough that at any case we insert the new sample in the target node in order to keep up with the stream.

Combination Function Component

Combination Function Component is responsible for grouping incoming predictions bases on the instance id and perform the aggregation function. We have implemented three different methods; Majority Voting, Weighted Voting using only the top k predictions and Weighted Voting using a cut-off threshold which accepts predictions which have its respective weight above that threshold. After our Experimental Evaluation, we concluded that using Weighted Voting using top-k was the better choice. Therefore, the following pseudocode is based on this. Here, we want to mention that we took advantage of Apache Flink's *countWindow* operator. This operator creates a window based on a user defined value. It waits until the current instances received matches the predefined value. In practice, we waiting until we have collected as many predictions as the number of Hoeffding Trees. The following code shows the functionality after the forementioned process.

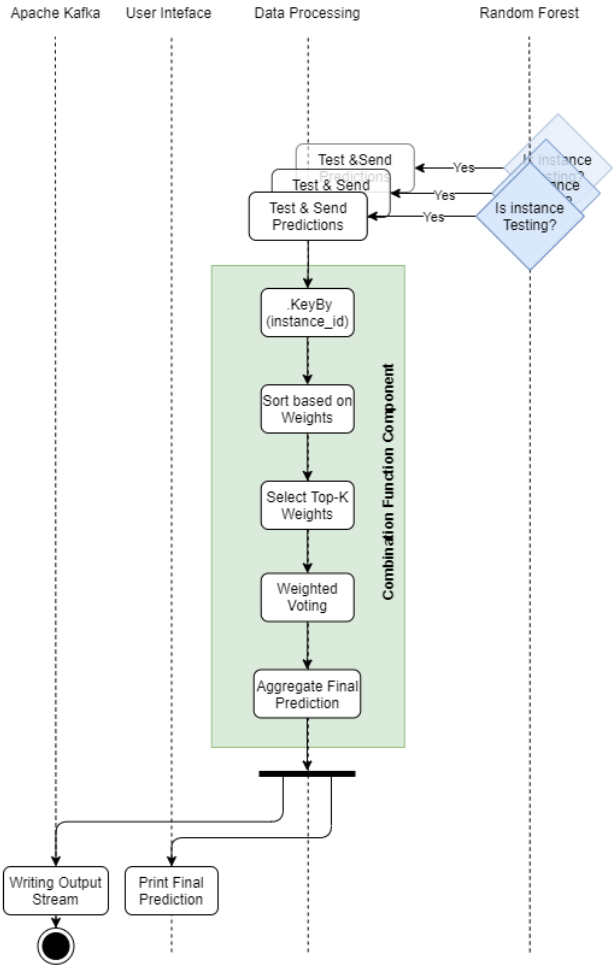


Figure 24. Combination Function Component

Algorithm 10: Combination Function

Input:	$list_{predictions}$	is a list of each Hoeffding Tree's predictions
	$list_{weights}$	is a list with the corresponding weights
	k	is the value for selecting the top-k weights
Output:	final prediction	is the system's prediction

```
1.  sorted ←Sort weights in an ascending order
2.  for all positions until k do
3.      if weight equals to 0 then
4.          class0_weight ← class0_weight + weight
5.      else:
6.          class1_weight ← class1_weight + weight
7.  if class0_weight > class1_weight then
8.      final prediction ← 0
9.  else:
10.     final prediction ← 1
11. return final prediction
```

Drift Detector Component

The Drift Detector Component is responsible for monitoring the Base Learner Component's performance and more precisely its error-rate. This component considers only training instances and is connected with the Base Learner Component after the latter updates its weight, as shown in Figure 25. The component implements the basic algorithm of DDM alongside our proposition. After the train process has been completed, we extract the corresponding accuracy metric. In case that the DDM warning constraints hold true then we enter in a warning phase. Our goal is to create a Hoeffding Tree in case there is not already one or test and then train it in case it has been initialized. So, in this case we have two parallel decision trees that are trained by the same input data using different characteristics. The main idea is that recency is relevancy and the new background tree will be trained only from the newest instances and consequently it will perform better. Being in a warning phase, there are two possibilities, either we go back to stable phase given a false alarm signal, or we enter in a drift phase given a drift signal. By entering in the drift phase, we wait until the background tree's accuracy is greater that the already existing one. If this happens, then we perform the switch process which concludes the removal of the existing tree and its replacement from the background tree.

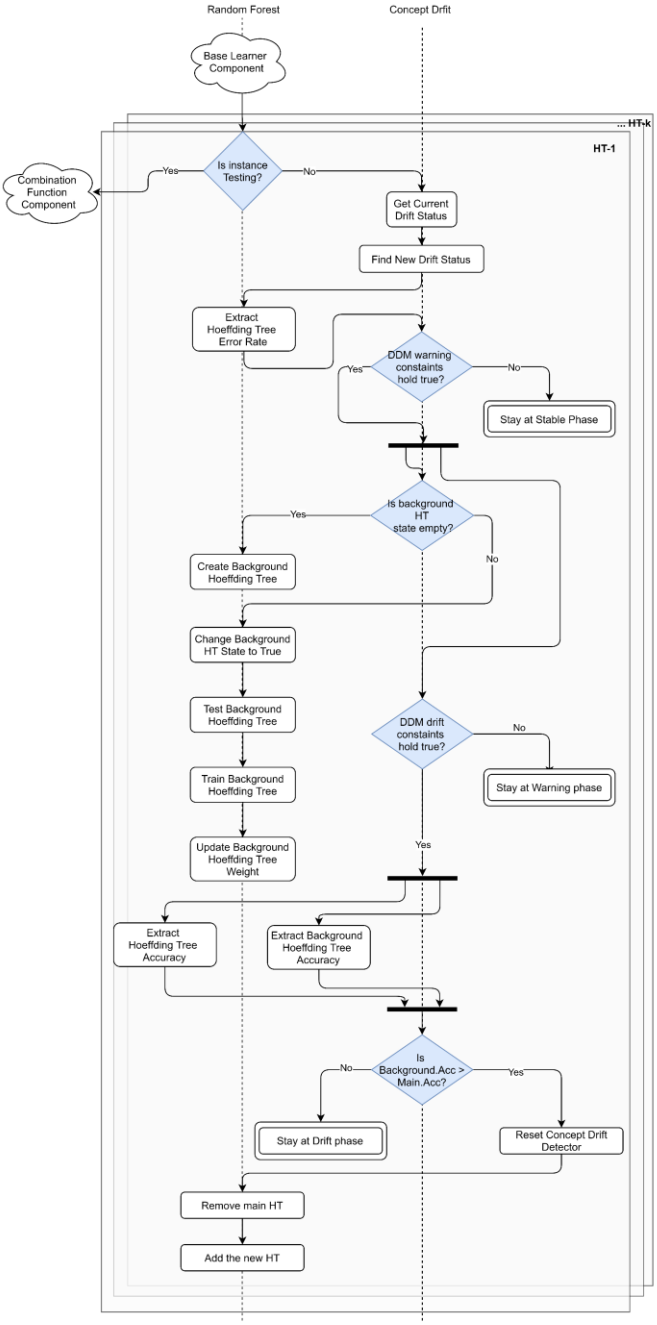


Figure 25. Concept Drift Detector Component

Algorithm 11: Concept Drift Detection

Input: error rate is the misclassification performance of the Hoeffding Tree

Output: signal can take three values: W for warning, D for drift and F for false alarm

```
Find New Drift Status using error rate
warning phase ← DDM warning constraints
if warning phase holds true then
        signal ← warning
        state ← get current background Ht's state
        if state is empty then
                Create Hoeffding Tree()                          // Background Tree
        else:
                state ← not empty
                Test-then-Train Background Hoeffding Tree()      // Update HT weight
        drift phase ← DDM drift constraints
        if drift signal holds true then
                signal ← drift
                acc_backgroundHT ← get Background Tree's accuracy
                acc_HT ← get Tree's accuracy
                if acc_backgroundHT > acc_HT then
                        Reset Concept Drift Detector
                        Switch positions of HT and Background HT
                else:
                        Stay at Drift Phase
        else:
                Stay at Warning Phase
else if signal equals to warning and warning phase does not hold true the
        signal ← false alarm

else:
        signal ←stable phase
        Stay at Stable Phase
return signal
```

Chapter 5:

Experimental Evaluation

The experimental analysis is designed based on two different tracks. The first one is to prove the functionality of our propositions. The second track is to prove how our system scales and how it copes with different number of decision trees and big data. Hence, we need a variety of experiments with different characteristics and lengths.

5.1 Testing Setup

The experiments associated with the scalability of this thesis presented were performed on the SoftNet Cluster of the SoftNet lab [37] with twelve Quad Core Xeon X3323 2.5GHz, 8GB. The Apache Flink version is 1.10.0 with Scala 2.11 and the Apache Flink Kafka Connector's version is 1.9.3. The experiments associated with the functionality of all the base learner and concept drift detector were performed in a local machine with i7-4720HQ CPU 2.60GHz, 16GB due to cluster unavailability at the time of these experiments.

5.2 Datasets

One purpose of our experimental evaluation is to generate evolving data streams that possess diverse concept changes (abrupt or gradual) with different drift duration. The included datasets are divided into two categories: artificial and real-world. In most cases, it is considered to test a proposed algorithm with only a few thousand instances, such as DDM and RDDM, where the evaluation was conducted based on a range between 50K and 3M instances while real-world datasets (ELEC, Airlines) are products of a real-world situations and their length cannot be changed. For both cases, we have selected the most well-known datasets in order to achieve an all-round evaluation. All the artificial datasets were previously used in the area and they are already included in the MOA framework [38]. MOA gives the choice to select, among others, the underling stream, the concept drift (the dataset which will replace the initial one after the drift), the position and duration of the drift, as long as the number of drifts and the choice of balanced dataset, in order to cover different scenarios. In our setup, we will produce streams with multiple concept drifts at regular intervals. Having such a powerful tool in our disposal, the evaluation process becomes more systematic.

An issue that rises in the case of artificial datasets is how we can quantify the different types of concept drifts. What makes a drift abrupt or gradual? Does the dataset's length influence the significance of the drift? One way to answer some of these questions, is to conduct multiple experiments and evaluate their results. Another way is to use the most common practices that are already performed in the same context. In [4] and [5], a drift length of 50 in 100.000 instances is considered as an abrupt drift type, whereas 500 as gradual. On the other hand, in [6] they do not distinguish concepts drifts in abrupt or gradual but they try to keep proportional the drift width and the total instances. Finally, in a comparative study on concept drift detectors [6] they define as fast gradual concept drift, the case when the width is 200 in 4000 instances and a gradual one when it is 1000.

Sine: Sine dataset consists from two numeric attributes with two additional (optional) irrelevant attributes. The attributes are uniformly distributed between [0,1] and follow the alternation of the $\boldsymbol{y = \sin(x)}$.

RandomTree

This generator, which was introduced in [19], makes a decision tree by choosing the attributes at random to split, and assigns a random class label to each leaf. After the tree is built, new examples are generated by assigning uniformly distributed random values to the attributes and the class label is determined via the tree. It has predefined parameters to control the number of classes, attributes and depth of a tree. Concept drift is created by changing the tree Random parameter:

Agrawal:

This generator, which was introduced in [39], consists of six numeric attributes and three categorical attributes to describe the hypothetical loan applications. For the numeric attributes, there is a perturbation factor that makes to shift the true value by adding an offset. It can produce ten different functions to determine whether the loan should be approved or not. The concept drift happens by changing the functions. For our experiment, we use the six functions referring as function 2 to 7 with 5% perturbation noise

SEA:

The SEA dataset [1] produces data streams with three continuous attributes (f_1, f_2, f_3). The range of values that each attribute can assume is between 0 and 10. Only the first two attributes (f_1, f_2) are relevant, i.e. f_3 does not influence the class value determination.

Airlines:

The Airlines dataset contains 539,383 records with 7 attributes (3 numeric and 4 nominal) and the goal is to predict whether or not a given flight will be delayed given information on the scheduled departure.

Electricity:

The Electricity Market dataset was used by Gama [25]. This data was collected from the Australian New South Wales Electricity Market. In this market, the prices are not fixed and are affected by demand and supply of the market. The prices in this market are set every five minutes. The ELEC2 dataset contains 45,312 instances. The class label identifies the change of the price related to a moving average of the last 24 hours. The class level only reflects deviations of the price on a one-day average and removes the impact of longer-term price trends

The detail numerical information of drift sizes and dataset used are described in Table 1.

Table 1
Description of datasets

Dataset	Total instances	No. of attributes	No. of classes	No. of drifts	Drift width
Sine-100k	100.000	4	2	4	100
Sine-3M	3.000.000	4	2	4	500
RBF-100k	100.000	10	2	5	100
RBF-5M	5.000.000	10	2	5	500
Agrawal-100k	100.000	9	2	4	100
Agrawal-3M	3.000.000	9	2	5	100
Agrawal-24M	24.000.000	9	2	10	1.000
SEA-100k	100.000	3	2	4	100
SEA-3M	3.000.000	3	2	5	1.000
Airlines	539,383	7	2	-	-
Electricity	45,312	8	2	-	-

5.3 Performance measures

5.3.1 Random Forest Evaluation

Confusion Matrix

Several indices were employed to monitor our classification method. Apart from the accuracy score, which is calculated as the ratio of correctly classified samples by the total number of samples, showing the overall accuracy of the method, other metrics of classification performance were also used for the evaluation of the algorithm. In order to calculate these metrics, the number of True Positive (TP), False Positive (FP), False Negative (FN) and True Negative (TN) samples were computed.

Table 2
Confusion Matrix

		Actual Value	
		Positive (1)	Negative (0)
Predicted Value	Positive (1)	*True Positive*	*False Positive*
	Negative (0)	*False Negative*	*True Negative*

Based on the Confusion Matrix

- Recall (or Sensitivity) of a tuple is its ability to determine a positive instance as such $TPR = \frac{TP}{TP+FN}$. As a result, the False Negative Rate $FPR = 1 - TPR$
- Specificity is the True Negative Rate $TNR = \frac{TN}{TN+FP}$ and is the model's ability to determine a negative case as such
- Precision (positive predictive value) is calculated by: $PPV = \frac{TP}{TP+FP}$. Precision can be interpreted as the ability of the classifier not to label as positive a sample that is negative.
- Last but not least, is the F1 score which is defined by the combination of Sensitivity and Precision as it is calculated using those two metrics

$$F1 = \frac{PPV \cdot TPR}{PPV + TPR} = \frac{2TP}{2TP + FP + FN}$$

5.3.2 Scalability Performance

In order to test the scalability of a system we track the execution time according to the size of data for different values of parallelism (throughput). In an ideal system, we have to witness the following result; as the parallelism is increasing, throughput has to have the exact opposite behavior. If we double the parallelism then the throughput has to decrease in half.

5.4 Experimental Results

5.4.1 SVFDT-II Results

The following table compares the SVFDT-II impact a long side to the original algorithm of VFDT. In this case it is not necessary to have concept drift datasets, therefore the tests were based on 3 artificial datasets of 1M instances (Sine, RBF, Agrawal) and the Electricity (ELEC) real world dataset. In order to see how significant is the integration of SVFDT-II is our system, we will keep track not only the number of nodes but also the accuracy and time elapsed. The following results are the mean value of 30 outputs. (See the Appendix Table for the Extensive Table)

Table 3.
SVFDT-II Improvement

Datasets	*without SVFDT-II*			*with SVFDT-II*			*Performance Improvement*		
Dataset Name	Size (nodes)	Acc.	Time elapsed (sec)	Size (nodes)	Acc.	Time elapsed (sec)	Size (nodes)	Acc.	Time elapsed (sec)
Sine	558.00	0.993	11.512	254.00	0.983	9.871	**-54.48%**	**-1.01%**	**-14.25%**
RBF	3.155	0.907	16.991	1.080	0.894	13.132	**-65.77%**	**-1.43%**	**-22.71%**
Agrawal	2.566	0.949	19.402	946.00	0.949	19.981	**-63.13%**	**0.00%**	**2.98%**
ELEC	221.00	0.77	7.249	127.00	0.777	6.179	**-42.53%**	**0.91%**	**-14.76%**

We can see that the size of the Hoeffding Tree is reduced on average by 56.4% while its downside is that the predictive ability (translated by its accuracy) has dropped 0.38%. We can strongly support that this trade-off is more than acceptable and proves our right call to include it.

5.4.2 Base Learner Proposition Results

In order to test our proposition, we have to create such scenarios that prove our concept. In the following figures we have three different datasets under concept drift of different lengths. So, we created on purpose such periods that we called "data drought". We will monitor the number of nodes as well as the accuracy. We will compare the original VFDT with the SVFDT-II improvement using the original algorithm of DDM, with the same setup plus our proposition. In both cases the tree parameters are exactly the same. In order to completely understand its functionality, we will present our results both visually and numerically.

Sine-100k

In Fig.23 we can observe that our proposition does not interfere in short and frequent concept drifts. As Method1 we denote the VFDT plus SVFDT-II. In this particular experiment our proposition is not deployed.

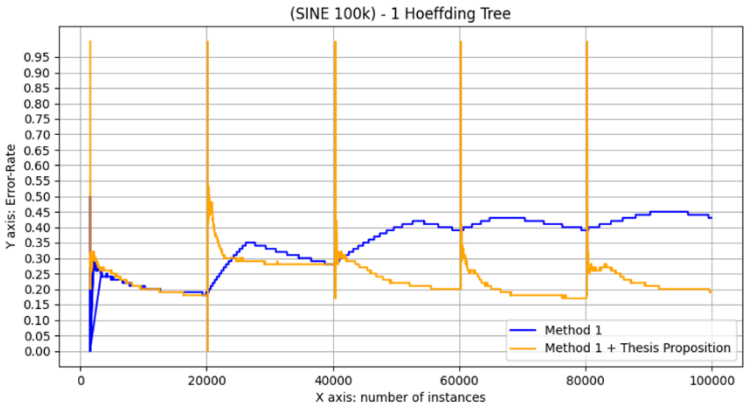


Figure 26. Sine 100k Base Learner Proposition Behavior

RBF-5M

On the other hand, in Fig.24 we see that we full potential of our proposition. In this experiment it is shown the combination of VFDT, SVFDT-II and our proposition with and without any concept drift detector. The two vertical red lines denote the two observation points. In first checkpoint, we start for both implementation from different points, having our proposition with the respective opposite around 1067 and 1605 nodes. In the second checkpoint we have 1067 and 4977 and for the implementation with and without our proposition respectively.

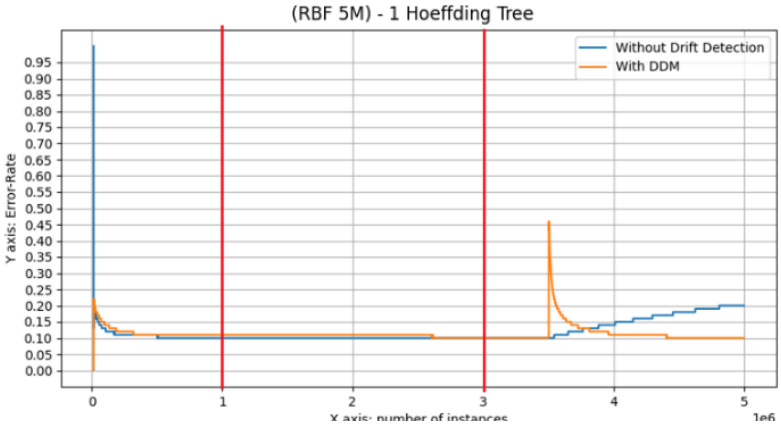


Figure 27. RBF-5M Base Learner Proposition Behavior

We additional tested in more two datasets Sine and Agrawal having both 3M data. Their Figures are in the Appendix Section. We can conclude that our proposition responds to scenarios where is the so called "data drought" as it reduces drastically the size and achieves to disengage the growth of the base learner according to number of inputs. The price it pays is the reduced accuracy which in some cases is almost 10% and in others is even.

Table 4.
Base Learner Proposition Results (Size and Accuracy)

		Size (Nodes)		**Error Rate**		**Performance Difference**	
Datasets	**Detection Points**	VFDT + SVDT-II	VFDT + SVDT-II + Proposition	VFDT + SVDT-II	VFDT + SVDT-II + Proposition	**Size (Nodes)**	**Error Rate**
Sine-3M	#1 (500k)	167	47	0.024	0.039	-71.86%	1.5%
	#2 (1.9M)	461	47	0.01	0.039	-89.80%	2.9%
	#3 (2.9M)	427	81	0.041	0.041	-81.03%	0%
RBF-5M	#1 (1M)	1605	1067	0.192	0.207	-33.52%	1.5%
	#2 (5M)	4977	1067	0.184	0.198	-78.56%	1.4%
Agrawal-3M	#1 (200k)	205	183	0.057	0.136	-10.73%	7.9%
	#2 (400k)	395	183	0.052	0.138	-53.67%	8.6%
	#3 (750k)	469	197	0.129	0.129	-58.00%	0%
	#4 (1M)	883	197	0.113	0.076	-77.69%	-3.7%
	#5 (1.5M)	301	101	0.073	0.082	-66.45%	0.9%
	#6 (2.5M)	1149	101	0.055	0.082	-91.21%	2.7%

5.4.3 Concept Drift Proposition Results

In this part of our experimental evaluatio we test our proposition for the Concept Drift Detector. In the following two figures we see two different scenarios of frequent and not frequent concept drifts. In the following figures with the blue line we have denoted the system using the DDM cocnept drift detector while with the green line the system with our proposed drift detector. The red dots show the error rate that we save by implementing our system. The interval of the changes are every 20k instances but we assume that we have not such prior knowledge of them. At each transition point the error rate skyrockets (red dots) which results that our system could have been replaced by a random classifier. The duration of such inability is around the one tenth of the total dataset.

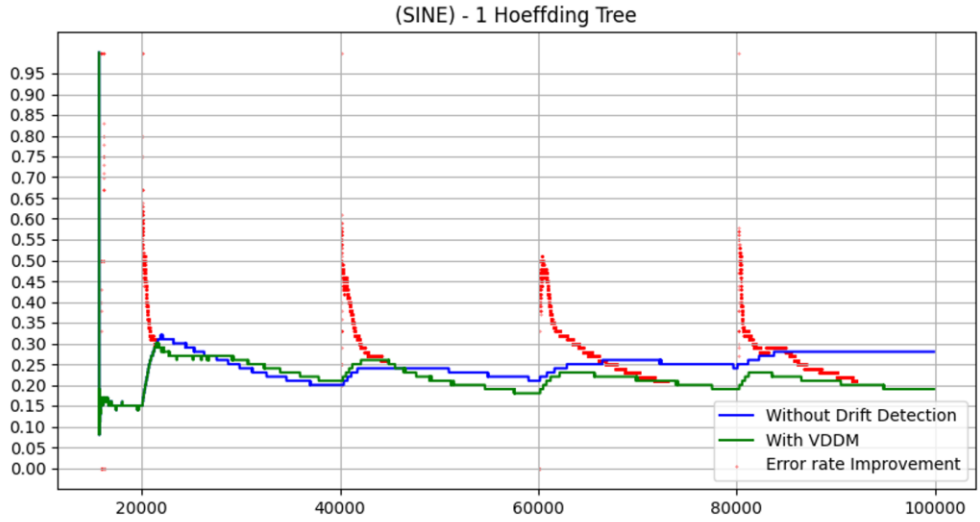


Figure 28. Sine dataset using our concept drift proposition

```
Background Tree 20061 Just Created
Pending for the Switch:  1 => 20100 Background Tree taking over 0.8416050686378036 => 0.4
Do the Switch:  1 => 21473 Background Tree taking over 0.7018817204301075 => 0.70193401592719
0.70193401592719 0.29806598407281004
Background Tree 40134 Just Created
Pending for the Switch:  1 => 40265 Background Tree taking over 0.7856201039323664 => 0.5352112676056338
Do the Switch:  1 => 44855 Background Tree taking over 0.7429779713438112 => 0.7430626927029804
0.7430626927029804 0.2569373072970196
Background Tree 60140 Just Created
Pending for the Switch:  1 => 60275 Background Tree taking over 0.8124266374520698 => 0.5205479452054794
Do the Switch:  1 => 73006 Background Tree taking over 0.7912377304285372 => 0.7912878787878788
0.7912878787878788 0.2087121212121124
Background Tree 80210 Just Created
Pending for the Switch:  1 => 80398 Background Tree taking over 0.8010321556173085 => 0.48623853211009177
57  0.81472815957489    End of Stream message 1
Hoeffding Tree 1 has ended its stream
End of Stream from all Hoeffding Trees
9.022
```

Table 5. Sine dataset. Numerical Representation of HT Switches

We can easily come to two conclusions. Firstly, our proposition is always better or equal better than the base learner using DDM concept drift detector. Secondly, we wait until our background tree becomes better than the existing one.

In the following figure we see the full functionality of our proposition while dealing with both vast periods between changes. The total instances of improvement are 5% of the total stream, which means that our proposition gives better answers during this 5% than the original DDM.

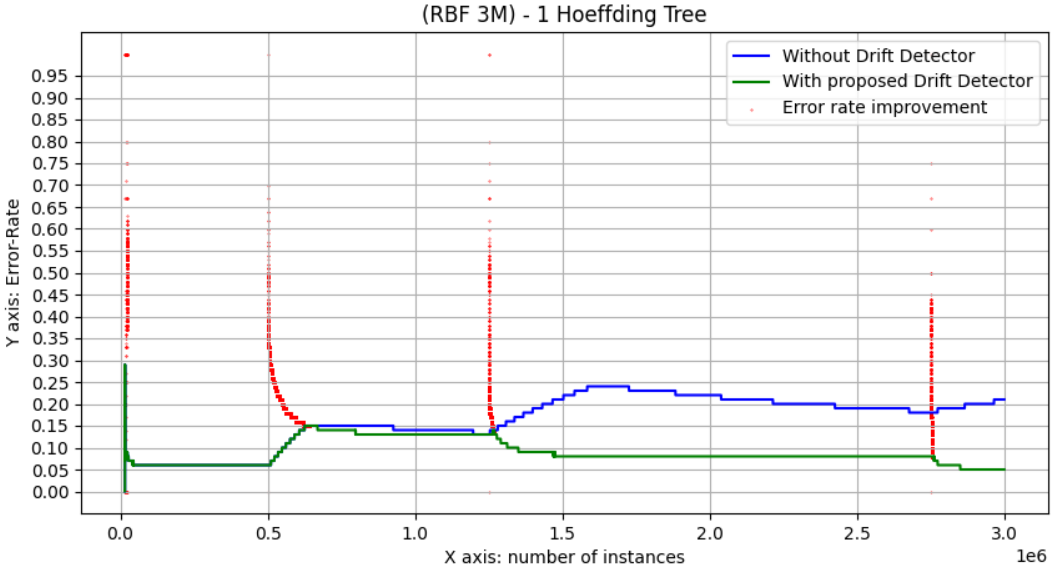


Figure 29. RBF 3M dataset using our concept drift proposition

```
Background Tree 500395 Just Created
Pending for the Switch:  1 => 500933 Background Tree taking over 0.9436625672309623 => 0.6134969325153374
Do the Switch:  1 => 641577 Background Tree taking over 0.850555697061249 => 0.8505557980992041
Background Tree 1250774 Just Created
Stable phase after a false alarm
Background Tree 1250779 Just Created
Pending for the Switch:  1 => 1251557 Background Tree taking over 0.8723657816860263 => 0.771255060728745
Do the Switch:  1 => 1266841 Background Tree taking over 0.8625667841309538 => 0.8625745283940964
Background Tree 2750882 Just Created
Pending for the Switch:  1 => 2751847 Background Tree taking over 0.9214541051356971 => 0.785831960461285
Do the Switch:  1 => 2757292 Background Tree taking over 0.9199404533976498 => 0.9199510403916769
41  0.9458388291396104  End of Stream message 1
```

Table 6. RBF 3M dataset. Numerical Representation of HT Switches

In Table 6, we can see that with the previous implementation of DDM we will have a huge drop in the performance from 0.94 to 0.61 by switching earlier that the ideal. While by our modification, we postpone this switch by 150k instances with a result of a smoother transition.

5.4.4 Ensemble Learning Results

In this part of the experimental evaluation, we treat our thesis as a complete machine learning model. We want to test its predictive ability in different scenarios. Therefore, we will use dataset with and without concept drift. The system has as its base learner the VFDT with the two Gaussian Approximation, SVFDT-II alongside our base learner proposition and our concept drift proposition. We have guaranteed that the tests are equally distributed in the dataset. The following datasets are also balanced.

Table 7.
Confusion Matrix Ensemble Results

Datasets	TP	FP	FN	TN	Recall	Accuracy	Specificity	Precision	F1 score
Sine-100k	658	167	187	677	0.790	0.778	0.802	0.797	0.788
RBF-1M	1611	249	219	1601	0.872	0.880	0.865	0.866	0.873
Agrawal-1M	912	14	98	1003	0.944	0.902	0.986	0.984	0.942
Electricity	296	80	190	396	0.719	0.609	0.831	0.787	0.686
Titanic	5	3	3	5	0.625	0.625	0.625	0.625	0.625

Scalability Results

Other than approaching this thesis from a mathematical point of view and treating it as machine learning model, we have to consider its scalability and evaluate its behavior in a distributed environment. In order to fully depict how efficient our proposition scales, we have to make two different tests.

The first has to be a scenario where we have a given dataset and a constant number of Hoeffding Trees, while changing the parallelism. Hence, we have to decide which dataset should be using and how many HTs are enough. We have to use such a dataset that its computational time is significantly larger than cluster's deployment time. We have decided to use Agrawal with 3M instances, which has 9 features. As for the number of HTs, we have decided to go with 32 of them. Don't forget that at the worst case we will have simultaneously deployed 64 of them. (32 main and 32 background) which are more than enough to showcase the scalability capability of our system. A more typical number of trees used in a random forest is perhaps 50 to 200 trees (according to Breiman's experiments [40]). Therefore, we are using only 2 out of 9 features in each of the 32 base learners in order to fully unravel the functionality of a Random Forest.

Secondly, it is necessary to do better than many others and consider a test with Big Data. In this approach we have tested our system with 50 Hoeffding Trees and parallelism of 32 using 24M instances. (24M instances of 9 numerical features is more than 3Gb)

Table 8.
Scalability Results

No. of Hoeffding Trees	Kafka Brokers	Kafka Partitions	Parallelism	Run 1		Run 2		Run 3		Run 4		Mean Exec. Time	Throughput
				Min.	Sec	Min.	Sec	Min.	Sec	Min.	Sec	Sec	**Instances / sec**
32	4	2	1	>3h	-	-	-	-	-	-	-	-	-
32	4	2	2	2h 2m	7320	1h48m	6480	2h 27m	8820	-	-	7320	**409.836**
32	4	8	4	28m 40s	1720	29m 2s	1742	32m 2s	1922	29m 4s	1744	1782	**1683.501**
32	4	8	8	17m 52s	1072	11m 8s	668	15m 6s	906	20m	1200	961.5	**3120.124**
32	4	8	12	13m 34s	814	18m52	1132	21m34s	1294	17m53s	1073	1078.25	**2782.2861**
32	4	8	16	12m 35s	755	9m 40s	580	8m 47s	527	10m 12s	612	618.5	**4850.444**
32	4	8	32	5m 21 s	321	4m 14s	254	5m 03s	303	4m 54s	294	293	**10238.907**

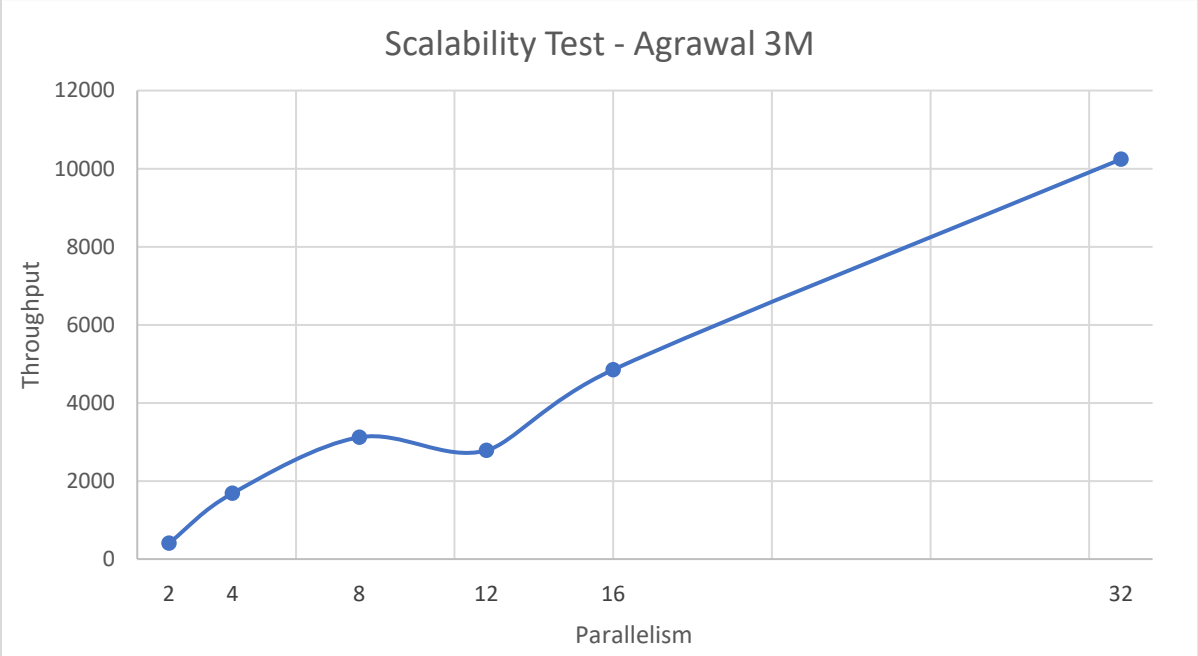


Figure 30. System's Scalability

Figure 30 can be considered as an expected result. Given the 32 Hoeffding Trees, in case of the small values of parallelism, the number of HTs that we are trying to fit in a tight space, seems to be a proper challenge, in which our system underperforms. For example, in parallelism of 2, Apache Flink tries to distribute the trees equally, 16 at each subtask, which means that the total number of instances coming at the same node, are destined for 16 different trees. Whereas, in the case of parallelism of 32 with 32 trees, each node receives instances for only one tree, a fact that increases the throughput.

Table 9.
System with 24M

No. of Hoeffding Trees	Kafka Brokers	Kafka Partitions	Parallelism	Run 1		Run 2		Run 3		Mean Exec. Time	Throughput
				Min.	Sec	Min.	Sec	Min.	Sec	Sec	**Instances / sec**
50	4	32	**32**	21m 41s	1301	14m 16s	856	17m 43s	1063	1073	**22,367.19**

Based on the above table, we can clearly see that our system is competent to handle a significant amount of Big Data. As we have already mentioned, the purpose of our proposition is to be any time query system that can handle an unbounded stream of data, deployed for such a scenario that there is no need of stopping it. Therefore, this experiment serves the purpose of showcasing that the system can handle rapid incoming instances. The rate with which we tested our system, may be a slightly extreme version of a use case.

Chapter 6:

Conclusions – Future Work

6.1 Conclusions

In this thesis, we proposed a distributed implementation of Random Forest at Apache Flink that detects concept drifts in evolving data streams. We implemented such optimizations, both on the base learner and the concept drift detectors in order to tackle problems that emerge in the Big Data world. We observed that we efficiently maintained the necessary statistics under the Gaussian Approximation and we managed to reduce each decision tree's size in return of not much significant loss of accuracy using the SVDFT-II algorithm alongside our proposition. At the same time, we observed that the proposed solution based on the Drift Detection Method achieved better overall accuracy throughout the input stream.

6.2 Future Work

The current work can be extended in many different ways. First of all, we would like in the future to develop a system that tracks all member's prediction and use some disagreement metric in order to find how many and which groups are formed. With such a useful information we can easily delete members of the ensemble that have the same recent history and add a new one only if we guarantee that its recent answers are a lot different from all other member. Therefore, we want to transition our fixed-size ensemble to a dynamic one by boosting its diversity. Diversity in general is not much tested in the ensemble systems and we think that it is a good opportunity. Finally, we could examine if the proposed solutions can have the same behavior in different datasets.

Appendix

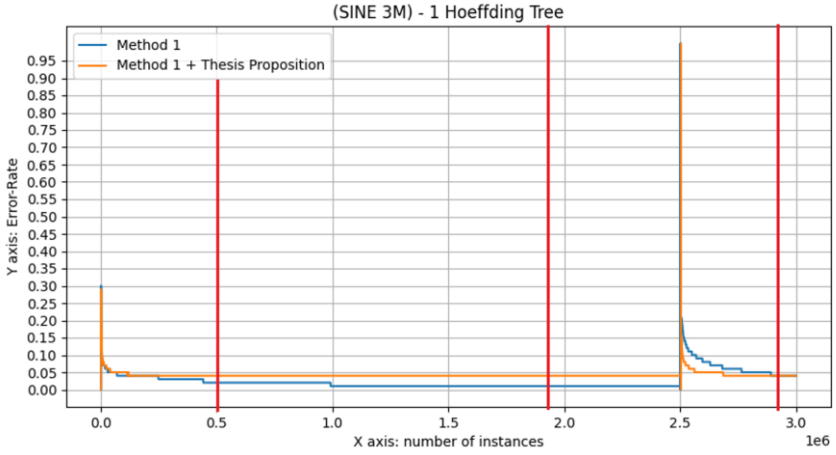


Figure used in Table 4 in Base Learner Proposition Results Section

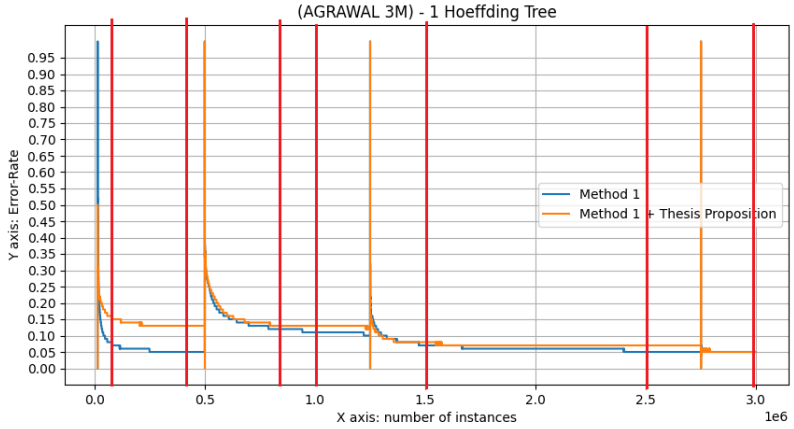
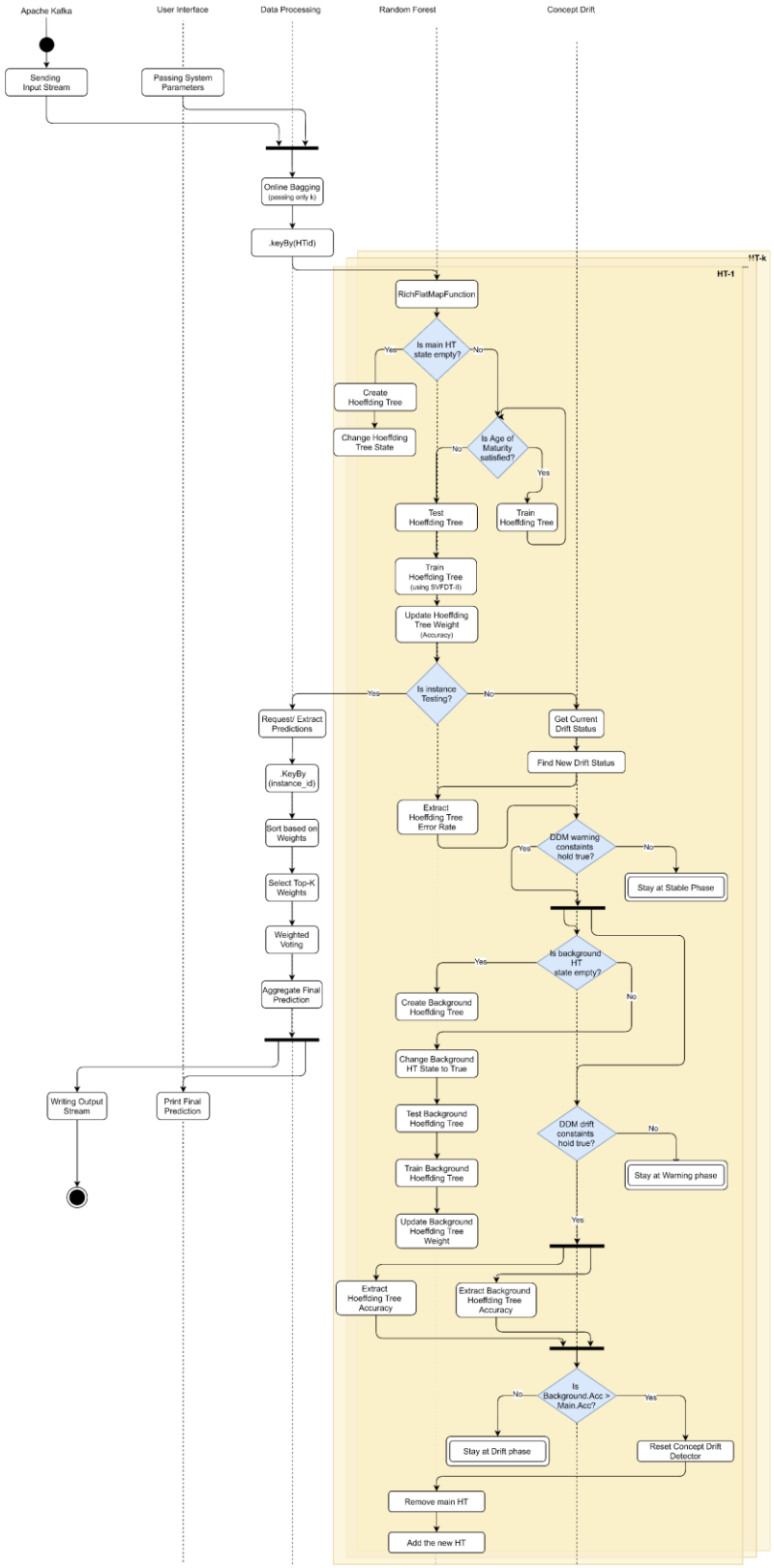
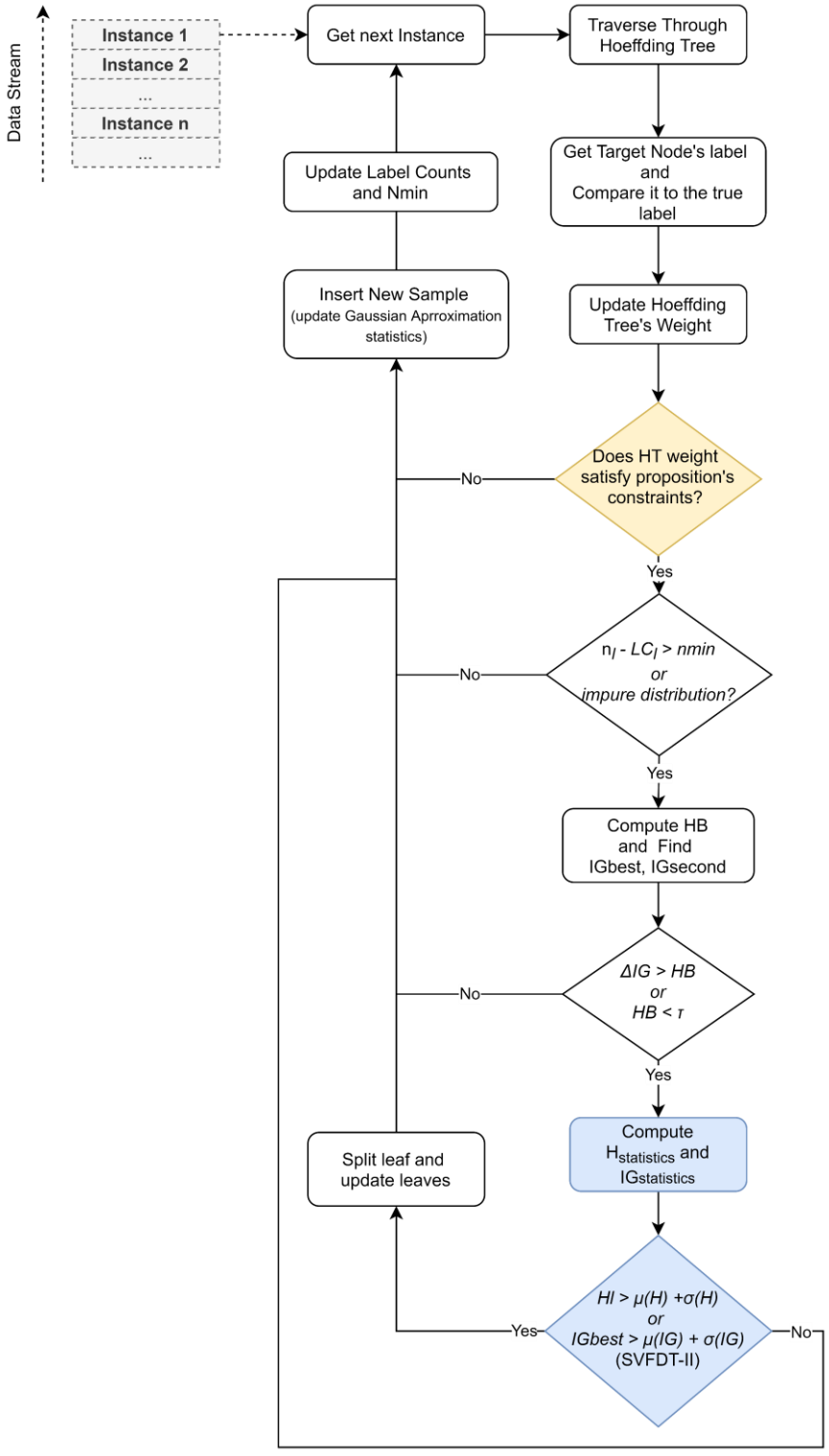


Figure used in Table 4 in Base Learner Proposition Results Section



Flowchart of System's Architecture used in Proposed Implementation Section



Flowchart of Training Method used in Train Hoeffding Tree Section

Bibliography

[1] Street W N and Kim Y 2001 A streaming ensemble algorithm (SEA) for large-scale classification *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '01* the seventh ACM SIGKDD international conference (San Francisco, California: ACM Press) pp 377–82

[2] Wolpert D H 1992 Stacked generalization *Neural Networks* **5** 241–59

[3] Zhang C and Ma Y 2012 *Ensemble Machine Learning* (Boston, MA: Springer US)

[4] Kotsiantis S, Patriarcheas K and Xenos M 2010 A combinational incremental ensemble of classifiers as a technique for predicting students' performance in distance education *Knowledge-Based Systems* **23** 529–35

[5] Iwashita A S and Papa J P 2019 An Overview on Concept Drift Learning *IEEE Access* **7** 1532–47

[6] Rokach L and Maimon O 2005 Top-Down Induction of Decision Trees Classifiers—A Survey *IEEE Trans. Syst., Man, Cybern. C* **35** 476–87

[7] Quinlan J R 1986 Induction of decision trees *Mach Learn* **1** 81–106

[8] Salzberg S L 1994 C4.5: Programs for Machine Learning by J. Ross Quinlan. Morgan Kaufmann Publishers, Inc., 1993 *Mach Learn* **16** 235–40

[9] Bramer M 2013 *Principles of Data Mining* (London: Springer London)

[10] Krawczyk B, Minku L L, Gama J, Stefanowski J and Woźniak M 2017 Ensemble learning for data stream analysis: A survey *Information Fusion* **37** 132–56

[11] Domingos P and Hulten G 2000 Mining high-speed data streams *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '00* the sixth ACM SIGKDD international conference (Boston, Massachusetts, United States: ACM Press) pp 71–80

[12] Hoeffding W 1963 Probability inequalities for sums of bounded random variables. *Journal of American Statistical Association* 58(1):13-30

[13] Hulten G, Spencer L and Domingos P 2001 Mining time-changing data streams *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '01* the seventh ACM SIGKDD international conference (San Francisco, California: ACM Press) pp 97–106

[14] da Costa V G T, de Carvalho A C P de L F and Junior S B 2018 Strict Very Fast Decision Tree: a memory conservative algorithm for data stream mining *arXiv:1805.06368 [cs]*

[15] Pfahringer B, Holmes G and Kirkby R 2008 Handling Numeric Attributes in Hoeffding Trees *Advances in Knowledge Discovery and Data Mining* Lecture Notes in Computer Science vol 5012, ed T Washio, E Suzuki, K M Ting and A Inokuchi (Berlin, Heidelberg: Springer Berlin Heidelberg) pp 296–307

[16] Gama J, Medas P and Rocha R 2004 Forest trees for on-line data *Proceedings of the 2004 ACM symposium on Applied computing - SAC '04* the 2004 ACM symposium (Nicosia, Cyprus: ACM Press) p 632

[17] Ditzler G and Polikar R 2013 Incremental Learning of Concept Drift from Streaming Imbalanced Data *IEEE Trans. Knowl. Data Eng.* **25** 2283–301

[18] Oza N C Online Bagging and Boosting 7

[19] Ruta D and Gabrys B 2005 Classifier selection for majority voting *Information Fusion* **6** 63–81

[20] Ruta D and Gabrys B Analysis of the correlation between majority voting error and

the diversity measures in multiple classifier systems 7

[21] Kolter J Z and Maloof M A Dynamic Weighted Majority: An Ensemble Method for Drifting Concepts 36

[22] Gomes H M, Barddal J P, Enembreck F and Bifet A 2017 A Survey on Ensemble Learning for Data Stream Classification *ACM Comput. Surv.* **50** 1–36

[23] Gama J, Žliobaitė I, Bifet A, Pechenizkiy M and Bouchachia A 2014 A survey on concept drift adaptation *ACM Comput. Surv.* **46** 1–37

[24] Katakis I, Tsoumakas G and Vlahavas I 2010 Tracking recurring contexts using ensemble classifiers: an application to email filtering *Knowl Inf Syst* **22** 371–91

[25] Gama J, Medas P, Castillo G and Rodrigues P 2004 Learning with Drift Detection *Advances in Artificial Intelligence – SBIA 2004* Lecture Notes in Computer Science vol 3171, ed A L C Bazzan and S Labidi (Berlin, Heidelberg: Springer Berlin Heidelberg) pp 286–95

[26] Barros R S M, Cabral D R L, Gonçalves P M and Santos S G T C 2017 RDDM: Reactive drift detection method *Expert Systems with Applications* **90** 344–55

[27] Bifet A, Hammer B and Schleif F-M 2019 Recent trends in streaming data analysis, concept drift and analysis of dynamic data sets *Computational Intelligence* 10

[28] https://incubator.apache.org/projects/samoa.html SAMOA Incubation Status Page - Apache Incubator

[29] https://moa.cms.waikato.ac.nz/ MOA

[30] Moumoulidou Z 2018 Dynamic Decision Trees in a Distributed Environment 73

[31] Ziakas C Implementation of decision trees for data streams in the Spark Streaming platform 52

[32] Papapetrou O and Garofalakis M 2014 Continuous fragmented skylines over distributed streams *2014 IEEE 30th International Conference on Data Engineering* 2014 IEEE 30th International Conference on Data Engineering (ICDE) (Chicago, IL, USA: IEEE) pp 124–35

[33] Bifet A, Zhang J, Fan W, He C, Zhang J, Qian J, Holmes G and Pfahringer B 2017 Extremely Fast Decision Tree Mining for Evolving Data Streams *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* KDD '17: The 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (Halifax NS Canada: ACM) pp 1733–42

[34] Gomes H M, Bifet A, Read J, Barddal J P, Enembreck F, Pfharinger B, Holmes G and Abdessalem T 2017 Adaptive random forests for evolving data stream classification *Mach Learn* **106** 1469–95

[35] https://flink.apache.org/ Apache Flink: Stateful Computations over Data Streams

[36] https://kafka.apache.org/ Apache Kafka

[37] https://www.softnet.tuc.gr/en/ Software Technology and Network Applications Laboratory | SoftNet

[38] https://www.cs.waikato.ac.nz/~abifet/MOA/API/namespacemoa_1_1streams_1_1generators.html MOA: Package moa.streams.generators

[39] Agrawal R, Imielinski T and Swami A 1993 Database mining: a performance perspective *IEEE Trans. Knowl. Data Eng.* **5** 914–25

[40] Breiman, L. Random Forests. *Machine Learning* **45,** 5–32 (2001). https://doi.org/10.1023/A:1010933404324