Fast Unsupervised Online Drift Detection Using Incremental Kolmogorov-Smirnov Test

Denis dos Reis
ICMC-USP
São Carlos, SP, Brazil
denismr@usp.br

Peter Flach
University of Bristol
Bristol, United Kingdom
Peter.Flach@bristol.ac.uk

Stan Matwin
Halifax, NS, Canada
Institute of Computer Science,
Polish Academy of Sciences,
Warsaw
stan@cs.dal.ca

Gustavo Batista
ICMC-USP
São Carlos, SP, Brazil
gbatista@icmc.usp.br

ABSTRACT

Data stream research has grown rapidly over the last decade. Two major features distinguish data stream from batch learning: stream data are generated on the fly, possibly in a fast and variable rate; and the underlying data distribution can be non-stationary, leading to a phenomenon known as concept drift. Therefore, most of the research on data stream classification focuses on proposing efficient models that can adapt to concept drifts and maintain a stable performance over time. However, specifically for the classification task, the majority of such methods rely on the instantaneous availability of true labels for **all** already classified instances. This is a strong assumption that is rarely fulfilled in practical applications. Hence there is a clear need for efficient methods that can detect concept drifts in an unsupervised way. One possibility is the well-known Kolmogorov-Smirnov test, a statistical hypothesis test that checks whether two samples differ. This work has two main contributions. The first one is the Incremental Kolmogorov-Smirnov algorithm that allows performing the Kolmogorov-Smirnov hypothesis test instantly using two samples that change over time, where the change is an insertion and/or removal of an observation. Our algorithm employs a randomized tree and is able to perform the insertion and removal operations in $O(\log N)$ with high probability and calculate the Kolmogorov-Smirnov test in $O(1)$, where N is the number of sample observations. This is a significant speed-up compared to the $O(N \log N)$ cost of the non-incremental implementation. The second contribution is the use of the Incremental Kolmogorov-Smirnov test to detect concept drifts without true labels. Classification algorithms adapted to use the test rely on a limited portion of those labels just to update the classification model after a concept drift is detected.

KDD '16, August 13-17, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-4232-2/16/08...$15.00
DOI: http://dx.doi.org/10.1145/2939672.2939836

Keywords

Kolmogorov-Smirnov; Data Stream; Concept Drift; Cartesian Tree; Treap; Lazy Propagation

1. INTRODUCTION

In the last decade, there was a tremendous increase of interest in algorithms that can learn from data streams. Data streams are fast and potentially infinite sequences of data records in which the underlying distribution can change over time. Therefore, data stream mining requires efficient algorithms in terms of memory and processing time to detect and adapt to concept changes, also known as concept drifts.

Such an increase of interest has lead to the proposal of a large number of learning algorithms for diverse tasks such as classification, clustering and anomaly detection [14, 15, 18]. In particular, for classification, these proposals have been mostly evaluated and compared assuming that true labels are readily available as soon as predictions are issued [14, 7, 24, 5, 6]. The availability of all true labels seems unfeasible for most applications, since it may involve annotating data by expensive means in terms of costs and labor time, such as a hired domain expert. Therefore, minimizing the amount of necessary true labels and delaying their requisition are desirable perks for practical solutions.

The first main contribution of this work is the Incremental Kolmogorov-Smirnov (IKS) algorithm. Kolmogorov-Smirnov (KS) is a non-parametric hypothesis test that is used to check whether two samples originate from the same distribution. Applying KS on a pair of samples takes $O(N \log N)$ time, where N is the total number of sample observations. This performance poses a limitation on data stream algorithms that require the test to be executed repeatedly with always-growing samples or with sliding windows. IKS, on the other hand, produces the exact same results as KS while enabling the samples to change over time at a cost of $O(\log N)$ for insertion/removal of observations and $O(1)$ for computing the p-value for the test. This contribution may be useful across a wide variety of applications yet to come in data stream research and practice.

The second main contribution is a direct application of IKS in data stream classification. We use IKS to identify detectable concept drifts online, using only data of the feature space. Thus, the detection is affected neither by delayed true labeling nor by label scarcity. Additionally, it fits any base classifier. We propose actions that could be taken to update the classification model once a change is detected, using only a limited amount of data. We show that the overall performance, in terms of accuracy, is only slightly affected, while the number of true labels that are required to keep the model updated decreases considerably.

This paper is organized as follows. Section 2 presents an overview of the literature; Section 3 details the IKS algorithm; Section 4 presents our proposals regarding drift detection; Section 5 explains our experimental setup; Section 6 shows our experimental results; finally, Section 7 presents our conclusions and directions for future work.

2. RELATED WORK

Verification latency, or delay, is the period between the availability of an unlabeled (test) instance and the availability of its true label. Such period of time depends on the application domain. For instance, in applications for predicting the tendency of a stock price or electrical demand, the verification latency is the forecasting window, i.e., the amount of time ahead of the prediction. In this case, the verification latency is fixed for all predictions. In other applications, verification latency can be variable, as in the case of sensors that classify events originated by external conditions such as the environment [4].

The Null-latency scenarios are rare in practice, although they are common in research benchmark evaluation. They are rare because most of the real-world applications require a certain amount of time between prediction and actual event occurrence. Such time is used for taking the required actions for the prediction. For instance, in the electrical demand application, a reasonable period of time, say 30 minutes, is necessary for executing a plan to adapt the energy generation to the predicted demand.

Different papers address the issue of availability of label information by considering different settings. One setting is known as extreme verification latency and considers that no true labels are available after a fixed time of the data stream, while all true labels previous to this point are known [26, 12]. Another setting assumes that only a portion of true labels becomes available along the time. In particular, for semi-supervised learning, it is common to assume that a small part of the labels becomes available with null verification latency, while the majority has infinite (extreme) verification latency [22].

Similarly to the semi-supervised setting, active learning also assumes that just part of the data is labeled. The critical difference between these setting is that in the active learning setting, the algorithm can choose which examples will be labeled by an oracle. Our proposal is, in essence, an active learning algorithm. The labels are required to learn the classification model, as expected for a supervised task. The stream is monitored in an unsupervised way and no additional labels are required if no concept drift is detected. In the case of a concept drift, the algorithm asks for a set of labeled data so the model can be updated.

In the literature, the most related work to ours was recently proposed by Zliobaite [28]. She showed that some types of concept drifts become undetectable when no true labels are provided. She proposed a method to identify detectable drifts without label information. Two consecutive, non-intersecting, sliding windows of equal size slide through the stream. The windows can contain either information from the feature space, or information from the classifier's output. At each time, *i.e.* one instance, a hypothesis test is performed to check whether the sliding windows have observations generated by the same distribution. If this is not the case, a drift is detected.

In the case of using information from the feature space, Zliobaite suggests using entire examples as multivariate random variables, or mapping them to univariate variables. However, no experiments were performed on either setting. In the case of use of information from the classifier's output, Zliobaite suggests using the estimated probabilities of the labeled instances, if the classifier is capable of offering such information, or only the labels, otherwise.

Three hypothesis tests were applied in the experimental setup. Among them, the use of the standard Kolmogorov-Smirnov test [21] was advocated since it is non-parametric and computationally efficient – when compared to the other options. The non-parametric tests suit better the data stream mixture of distributions due to existence of different classes and also regions of concept transition. While [28] offers results regarding drift detection, no results are reported regarding the impact in classification in terms of accuracy or in number of required labels.

Haque et al. [17] proposed an algorithm that does not use labels to identify the optimal size of a dynamically sized sliding window that is supposed to contain instances belonging to the same concept. Once a drift is detected and the window is shrank in order to keep only the instances of the newest concept, the algorithm assumes the instantaneous availability of 99.7% of the true labels.

Masud et al. [22] presented an algorithm that is both capable of dealing with delayed labeling and identifying novelty, *i.e.*, new classes during the stream. However, apart from the specific assumptions regarding the feature space, it also assumes that, once an instance is observed, the classifier puts it aside for a prolonged period of time before needing to provide a classification, creating a slightly different scenario for the problem of delayed labeling.

Although Kuncheva et al. [20] and Amir and Toshniwal [2] introduced methods to directly tackle delayed labels, they target problems with stationary distributions, *i.e.*, problems without concept drift. Kuncheva proposes different variations of the Nearest Neighbor Classifier (NNC) for online learning. In all of them, the reference set grows incrementally over time, with unlabeled instances being added when they are classified with low confidence. In this case, the predicted labels are employed as if they were correct, until they are revised later, when the true labels become available. Amir's proposal is a follow-up of Kuncheva's work. The extension is the use of emerging patterns in order to select which instances should be added to the NNC's reference set.

Other papers have dealt with the problem of extreme verification latency, *i.e.*, no labels are available during the classification phase of the data stream. Extreme verification latency methods require only an initial amount of labeled data, that must be located at the beginning of the stream and, then, no labeled data is requested anymore.

Dyer et al. [12] and Souza et al. [26] aim at addressing exclusively the infinite delay problem. Both of them make assumptions regarding the shape of the data in the feature space and assume an incremental nature of the drifts. Dyer's proposal applies semi-supervised classification on batches of unlabeled data, considering the predicted labels of the previously classified instances as correct. Souza's approach clusters the data from time to time and checks for spatial similarity between the clusters to assume a movement of the data.

The literature has also dealt with the problem of label scarcity and delayed labeling using semi-supervised learning.

Pozzolo [8] directly tackled verification delay in the credit card fraud detection task. The proposal assumes that, while a greater portion of the data is affected by delay, the true labels are instantly available for a much smaller portion. The results suggest that combining a model built on the instantaneously labeled data and a different model built on labeled data that was affected by delay provides better results than using only one model induced upon all the data.

Masud et al. [23] addressed a problem where only a portion of the true labels is delivered. The labeled instances, among a supposed larger number of unlabeled instances, are used in a semi-supervised approach to classify future unlabeled data. The main difference of this approach is that, although only few true labels become available, they are provided without delay.

Wu et al. [27] introduced another semi-supervised approach to deal with scarce labeled data in streams. It is based on a decision tree that grows incrementally and holds clusters in its leaves to detect concept drifts.

3. INCREMENTAL KS

In this section, we first briefly review the standard Kolmogorov-Smirnov test [21] and how to apply it. Later, we introduce the required operations for the incremental version of the test.

3.1 Kolmogorov-Smirnov Test

Suppose we have two samples A and B containing univariate observations. We would like to know, with a significance the level of α, whether we can reject the null hypothesis that the observations in A and B originate from the same probability distribution.

If no information is available regarding the data distribution, but it is safe to assume that the drawn observations are i.i.d., we can use the rank-based Kolmogorov-Smirnov (KS) test to verify the proposed hypothesis. According to it, we can reject the null hypothesis at level α if the following inequality is satisfied:

$$D \overset{?}{>} c(\alpha)\sqrt{\frac{n+m}{nm}}$$

where the value of $c(\alpha)$ can be retrieved from a known table, n is the number of observations in A and m is the number of observations in B. The right side of the inequality is the *target* p-value. D is the Kolmogorov-Smirnov statistic, *i.e.*, the *obtained* p-value, and is defined as follows:

$$D = \sup_x |F_A(x) - F_B(x)|$$

where

$$F_C(x) = \frac{1}{|C|} \sum_{c \in C, c \leq x} 1$$

We note that D can actually be computed as follows:

$$D = \max_{x \in A \cup B} |F_A(x) - F_B(x)|$$

The incremental variant assumes that A and B can change over time. Precisely, we assume an abstract data type (ADT) with the following operations: a) insert new observation into A; b) insert new observation into B; c) remove observation from A; d) remove observation from B; e) apply KS test.

In the next section we introduce an algorithm that allows us to perform the first four operations in logarithmic time with high probability according to the total number of observations, and the last operation in constant time.

3.2 Incremental Variant

In this section, we introduce our proposal of an algorithm to fast compute the operations of the incremental version of the KS test. Before that, we need to clarify two points regarding our approach:

1. We created a distinction between $|A|$ and n and, equivalently, $|B|$ and m. $|A|$ and $|B|$ are the numbers of observations of A and B, respectively, inserted into a data structure and n and m are the actual numbers of elements in these samples. In principle, $|A|$ and n and $|B|$ and m should have the same values. However, we can manipulate the number of observations inserted into the data structure to circumvent a constraint of our implementation explained next;

2. We will assume that we are interested in effectively computing D when $|A| = r|B|$, where $r \in \mathbb{R}$ is a parameter that remains constant during the entire stream. At a first glance, this seems to be a very restrictive constraint. However, we notice that a common use of the incremental KS test in a data stream setting is to compare the content of two sliding windows of fixed and equal sizes, so that $r = 1$. We can handle other cases by replicating the elements inserted into the data structure, as Section 3.4 discusses in detail.

With the constraint that $|A| = r|B|$, we can rewrite D as

$$D = \frac{1}{|A|} \max_{x \in A \cup B} |F'_A(x) - F'_B(x)|$$

where F'_A is defined as a sum of ones

$$F'_A(x) = \sum_{a \in A, a \leq x} 1$$

and F'_B is now defined as a sum of r's

$$F'_B(x) = \sum_{b \in B, b \leq x} r$$

Moreover, if we define $G(x) = F'_A(x) - F'_B(x)$, then we have that

$$D = \frac{1}{|A|} \max\left\{\left(\max_{x \in A \cup B} G(x)\right), -\left(\min_{x \in A \cup B} G(x)\right)\right\} \quad (1)$$

Let us assume the existence of an array in which we have all the observations $o_i \in A \cup B$ sorted so that $o_i \leq o_{i+1}$ and,

Table 1: $G(x)$ for each observation $x \in A \cup B$. The table represents a sorted array so that $o_i \leq o_{i+1}$

Index	1	2	...	$\|A\| + \|B\| - 1$	$\|A\| + \|B\|$
o_i	o_1	o_2	...	$o_{\|A\|+\|B\|-1}$	$o_{\|A\|+\|B\|}$
$G(o_i)$	g_1	g_2	...	$g_{\|A\|+\|B\|-1}$	$g_{\|A\|+\|B\|}$

for each observation, we also have a corresponding value $g_i = G(o_i)$. Table 1 illustrates such a data structure.

Let us also assume that, when inserting a new observation o_j into the structure, we obey the following order $o_{j-1} < o_j \leq o_{j+1}$. In other words, all older observations with the same or higher value are kept on the right side of the new observation, and consequently have higher indexes. As a result, we note that $g_j = g_{j-1} + v$, where $v = 1$ if $o_j \in A$ or $v = -r$ if $o_j \in B$. In addition, after the insertion, all g_i with $i < j$ remain the same, while all g_i with $i > j$ are increased by v . Similarly, the removal of such an observation decreases all g_i with $i > j$ by v.

We can insert/remove a new observation o_j into/from the structure, virtually add/subtract a constant value to/from all g_i with $i > j$ in $O(\log |A| + |B|)$ with high probability. We can also compute the maximum and the minimum values of g_i in $O(1)$ using a randomized tree called *Treap* (or *Cartesian Tree*) [3, 25] with *bulk operation and lazy propagation*. Therefore, it is possible to exactly compute D whenever $|A| = r|B|$ in $O(1)$ and incrementally modify $|A|$ or $|B|$ with expected complexity of $O(\log |A| + |B|)$.

Appendix A is dedicated to explain the *Treap* with *lazy propagation*. The next section explains how it is employed in the Incremental Kolmogorov-Smirnov algorithm. A full implementation of the Incremental Kolmogorov-Smirnov is freely available as supplementary material [11].

3.3 IKS Algorithm Description

We implement the structure presented in Table 1 as a Treap with bulk operations (minimum, maximum and increase by) and lazy propagation. The values are the g_i, the BST keys are the observations from a feature in the data stream and the priorities are random values. The final solution follows from Equation 1. The following algorithm describes the implementation of the five operations that we need in the Incremental Kolmogorov Smirnov: inserting observation into A, inserting observation into B, removing observation from A, removing observation from B and applying the KS test. The last operation requires $|A| = r|B|$.

```
function AddObservationFromA(Treap, Observation)
    Left, Right ← Split(Treap, Observation)
    Left, Temp ← SplitLast(Left)
    if Temp is NIL then
        InitialValue ← 0
    else
        InitialValue ← Temp.Value
    end if
    Left ← Merge(Left, Temp)
    Priority ← DrawRandomValue()
    NewNode ← NewTreeNode(Priority, Observation, InitialValue)
    Right ← Merge(NewNode, Right)
    IncreaseBy(Right, 1)
    return Merge(Left, Right)
end function

function AddObservationFromB(Treap, Observation)
                        ▷ Same initial steps as AddObservationFromA
    IncreaseBy(Right, $-r$)
    return Merge(Left, Right)
end function

function RemoveObservationFromA(Treap, Observation)
    Left, Right ← Split(Treap, Observation)
    Ignore, Right ← SplitFirst(Right)
    IncreaseBy(Right, $-1$)
    return Merge(Left, Right)
end function

function RemoveObservationFromB(Treap, Observation)
                        ▷ Same initial steps as RemoveObservationFromA
    IncreaseBy(Right, $r$)
    return Merge(Left, Right)
end function

function KolmogorovSmirnovTest(Treap, $|A|$, $|B|$, n, m, $\alpha$)
    $D$ ← max{Treap.Max, $-$Treap.Min}/$|A|$
    $p$-value ← $c(\alpha)\sqrt{\frac{n+m}{n*m}}$
    if $D > p$-value then
        Reject null-hypothesis
    else
        Do not reject null-hypothesis
    end if
end function
```

3.4 Limitations and Workarounds

The only constraint in the Incremental Kolmogorov-Smirnov test over its original version is that $|A| = r|B|$, where r is a parameter that remains constant across the stream. Although it seems to be a restrictive limitation, in practice it is not.

If both samples have fixed size and the changes are only due to replacement of observations, *i.e.*, pairs of insertions and removals, the constraint is fulfilled regardless the size of the samples. In addition, if both samples always have the same size, $r = 1$ and the constraint is fulfilled no matter the size of the samples.

Otherwise, we suggest that r should be chosen as 1 by default and, in case of disparity among $|A|$ and $|B|$, resampling should be performed. In such case, the m and n should be used to compute the *target* p-value.

We note that, for instance, if $m = 2n$ and we insert all observations from A twice, so that $|A| = |B|$, both D and target p-value are exactly the same as the obtained by performing the standard version of Kolmogorov-Smirnov. In fact, if A has fixed size while B grows indefinitely, it is possible to keep exact equivalence between the Incremental KS and the Standard KS every time that $m = kn, k \in \mathbb{N}$. Differences in the Incremental KS and the Standard KS between consecutive values of k also become increasingly smaller.

4. DRIFT DETECTION AND ADAPTATION

This section explains how we applied the IKS algorithm in order to detect concept drift without the true label information and adapt the classification models.

For each feature of the feature space, we keep two samples of the same size W. The first of them, called *reference set*, stores the attribute values of the instances that were used to induce the classification model. Therefore, this sample is fixed. The second sample, called *current set*, is a sliding window that stores the attribute values of the last W instances of the stream. After each instance is classified, for each feature, the current set is updated and a KS test is performed between the current set and the reference set. If the KS rejects the null hypothesis from both samples coming from the same distribution, a drift is detected.

When a classifier outputs probability estimates, we can also perform the test as described with some minor changes.

The current set contains probability estimates for the classification labels that were output by the classifier (acting as confidence levels). The reference set contains probability estimates for the labels obtained in a *leave one out* procedure with the training data.

Our proposal differs from [28] in two ways. First, we opted to keep one of the windows fixed, rather than maintaining two consecutive sliding windows. The rationale behind this decision is straightforward: the need of detecting drifts is acknowledging that the classification model is outdated. Thus, it is more practical to simply use the data that are in the training set of the classification model rather than assuming it is outdated indirectly.

The second difference is that, when we use the attribute space to detect drift, we apply the test for each attribute individually rather than using a multivariate test or a map function to transform the examples into a single univariate value. Both options would require larger samples and, as we note, a change in a single attribute already is a concept drift. The downside of our decision is that if a concept drift in a specific attribute is undetectable [28], it could possibly be detected when applying a multivariate test.

We propose three reactions for the system to perform once a drift is detected. They follow:

- **Model replacement (MR):** once a drift is detected, the system requests the true labels of the instances in the current set, trains a new classifier with these data and updates the reference set accordingly, no matter what attribute was responsible for the drift to happen;

- α/β **transformation (AB):** for the attribute that caused the drift, the system translates and stretches the reference set values so their mean and standard deviation meet the ones in the current set. If the transformed reference set and the current set come from the same distribution, according to the KS test, then the classification model is retrained with the transformed reference set data. Notice that transforming the reference set does not require additional true labels. If the KS test fails, we perform the MR;

- **Adaptree (AT):** it is a modified decision tree. Once a drift is detected, the system requests only the true labels of the instances that reach the nodes of the tree with a decision based on the drifted attribute. The reference set for that attribute is updated to keep data of the current set, while the reference sets for attributes of ancestor/prime nodes are kept unchanged. This means that, in practice, we keep multiple reference sets: one for each attribute.

The next section explains how we evaluated our approaches.

5. EXPERIMENTAL SETUP

Our experimental setup has two parts. In the first one, we evaluate the Incremental Kolmogorov-Smirnov's performance in seconds against the successive application of the standard KS and a slightly optimized version for data streams. This particular version, called Optimized KS, computes the KS statistic in linear time for consecutive samples that differ due the increment or replacement of one observation, by performing linear sorted insertion/remotion of observations and not re-sorting the sample later. For this purpose, we ran two experiments. First, we tested the performance for always growing samples. The second test evaluates the time necessary to perform the KS test using two fixed size sliding windows. All the experiments were run in an i7 4790k @3.6Ghz, 16GB DDR3 RAM @1600Mhz.

In the second part, we evaluate our proposals for drift detection and adaptation in a classification context. We measure the accuracy and number of required true labels on different datasets. Since the classes are balanced, the accuracy is a good indicator of prediction quality.

We compare the results with two baselines and a topline. The first baseline (BL1) is a classifier that never adapts to drift. The second (BL2) is a classifier that randomly adapts to drift by retraining the classification model at random moments. The number of adaptations of BL2 is the same as of MR. Due to its random nature, we averaged the results of BL2 in 100 runs. The topline is a classifier always retrained upon the last W instances, where W is the size of the sliding window. The initial training size is also W, for all methods. Additionally, we compare our proposal with a classifier that detects drift using one of the methods proposed by Zliobaite [28] – specifically, two consecutive sliding windows with probability estimates. From now on, we will refer to Model Replacement with this method of detection as **Consecutive Sliding Windows Detection (CD)**.

The classifiers in our experiments are Nearest Neighbor, Decision Tree and Naive Bayes. Baselines 1 and 2 (BL1 and BL2), Topline (TL) and Model Replacement (MR) results are computed for all of them. Particularly, we apply MR with Naïve Bayes for probability estimates (MRP) and feature space (MR), since Naïve Bayes is the only accessed classifier that directly provides probability estimates. 1NN is also tested with α/β transformation, DT is also tested with Adaptree and CD is also tested with Naïve Bayes. For the methods that use decision trees, we only detect drifts of features that are decision nodes inside the tree.

All pairwise comparisons of the second part are based on Wilcoxon Signed Rank Test for statistical validity with a significance level of 0.05. The following section lists the datasets that were used in our experiments.

5.1 Datasets

We used 6 real datasets in our experiments. Three of them have artificially introduced drift. We explain each of the datasets below.

(A) **Arabic** [16] contains audio features of 88 people pronouncing Arabic digits between 0 and 9. 44 are females and 44 are males. The task is to predict which digit was pronounced. The dataset is originally i.i.d. To artificially introduce drift, we separated the data stream in 4 parts of equal size. These parts alternate between male and female voices. The stream has 8800 instances;

(B) **Posture** [19] contains data from a sensor that is carried by 5 different people. The task is to predict which movement is performed, among 11 possibilities. This is the only dataset that is not balanced across all classes, thus we note that the proportion of the majority class is 33%. There are 164860 instances. The dataset is originally i.i.d. To artificially introduce drift, the stream has segments of data that were produced by the same person;

(C) **Bike** [13] contains hourly count of rental bikes between years 2011 and 2012 in a bikeshare system with the corresponding weather and seasonal information. The task is to predict whether there is a *high* or *low* demand. We expect concept drift due to seasonality. It contains 17379 instances;

(D) **Keystroke** [26] contains features from five people typing the same password many times along the stream. The task is to predict who is typing the password. We expect concept drift since a person may type a password faster once they get used to it; there are 1600 instances;

(E) **Insects** [9] contains features from a laser sensor. The task is to identify the specimen of flying insect that is passing through the laser in a controlled environment, among 5 possibilities. Preliminary analysis showed that there is no drift in the feature space, however the prior distribution of the classes changes *gradually* over time. There are 5325 instances;

(F) **Abrupt Insects** is a modified version of the Insects dataset. We shuffled the data to eliminate prior distribution changes. After, we split the stream into 3 segments. In the middle one, we shuffled all the features to introduce abrupt drift in the feature space, without inserting additional artifacts in the data.

The exact datasets that were used in our experiments are freely available to the community [11].

All algorithms apply KS with a significance level of 0.001 to detect concept drift. The reasoning behind this decision is that, with a higher significance, we can avoid detecting drifts during periods when the sliding window is still transitioning between two concepts. The size of the sliding windows is 100 instances for all datasets but Arabic and Posture. As they are bigger and have a greater number of classes, we opted for sliding windows of 500 instances for them. Next section presents our results and analysis.

6. EXPERIMENTAL RESULTS

We have split the experimental results into two parts. We first tackle only the time efficiency of the Incremental Kolmogorov-Smirnov. Later, we show our results for the data stream classification.

6.1 IKS Performance

In the first experiment, we evaluate the performance of IKS test for always growing samples. The two samples, initially with 1 observation, increase their size by one at each time step and then the KS test is performed. We increased the size of the samples up to 5000 observations. The reported results are the average of 30 repetitions and the values are generated at random.

Figure 1 shows the accumulated time by the samples' size throughout the complete experiment. Although the confidence intervals were plotted, they are so tight that they have became indistinguishable from their corresponding curves. For instance, for a sample that grows up to 5000 observations, Incremental KS took an average of 0.0491 seconds to complete the experiment. Optimized KS took 1.0297 seconds and Standard KS took 12.7645 seconds on average. In summary, IKS was one order of magnitude faster than OKS and two orders of magnitude faster than standard KS.

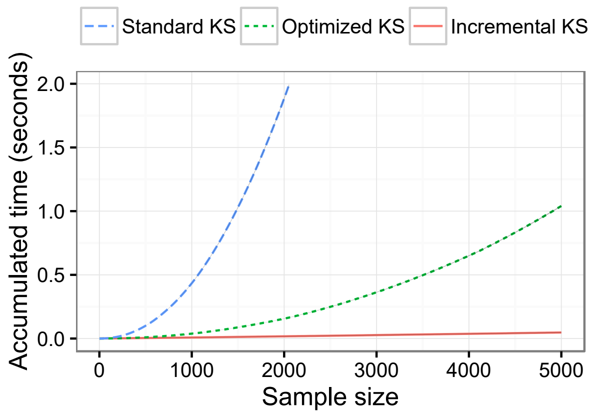

Figure 1: Averaged accumulated time for always growing sample.

The second test evaluates the time necessary to perform the KS test using two fixed size sliding windows through two parallel streams of observations: at each time step, one observation is removed from each sample and one is included. The stream has 10000 random observations and we vary the size of the window from 100 to 1000, increasing 100 at a time.

The stream experiment is more realistic and meaningful for the purpose of this work, since it meets our proposals for drift detection and classification. Figure 2 shows the time necessary to scan the whole streams by the size of the sliding windows that were in use. Again, confidence intervals were plotted, but they are too tight to be distinguishable from their correspondent curves. For instance, for a sliding window of size 1000 observations, Incremental KS took an average over 30 repetitions of 0.0410 seconds. Optimized KS took 0.9605 seconds and Standard KS took 5.326 seconds. Again, IKS was one order of magnitude faster than OKS and two orders of magnitude faster than standard KS.

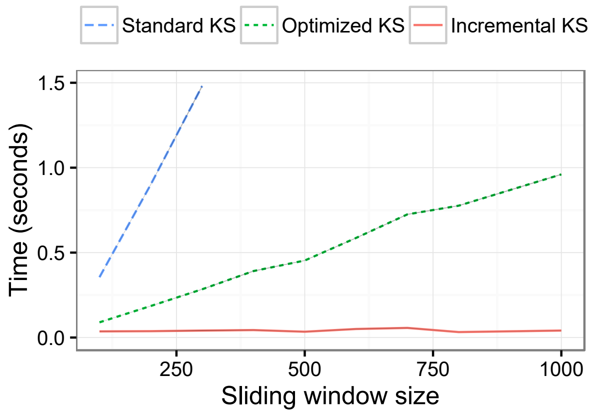

Figure 2: Averaged time for scanning a stream with different sliding window sizes.

6.2 Data Stream Classification

Tables 2 and 3 summarize our experimental results for accuracy and percentage of requested true labels, respectively. For all classifiers, Model Replacement statistically produced greater accuracy than the correspondent baselines and requested less true labels than the corresponding topline. Although the accuracy of the Model Replacement was statistically smaller than the accuracy of the topline for all classifiers, Model Replacement lost by a very small margin. On average, the Model Replacement accuracy was 99.39%, 94.31% and 93.69% of the Topline accuracy, for the Nearest Neighbor, Naïve Bayes and Decision Tree classifiers, respectively. Conversely, Model Replacement requested, on average, 47.59% for Nearest Neighbor and Naïve Bayes classifiers and 42.17% for Decision Tree classifier of the true labels.

All the remaining methods led to significantly smaller number of requested labels. However, the classification performance is usually reduced as well. A notable exception is Adaptree that obtained performance similar to Model Replacement for decision trees, but using significantly fewer labels (42.17% for MR and 35.62% for Adaptree). Unfortunately, decision tree classifiers did not perform as well as the Nearest Neighbor classifiers.

The α/β Transformation statistically reduced the portion of true labels that were required if compared to Model Replacement (16.89%, on average, for α/β). However, this particular transformation is sensitive to the nature of the drift since it assumes a monotonic drift [1]. If the assumption does not hold, even if the drift is detectable, it may cause lower accuracy. In our tests, the accuracy produced by this method was statistically smaller than Model Replacement alone. On average, α/β obtained 92.75% of the Topline accuracy.

Finally, using a fixed sample (reference set) was consistently more accurate than Consecutive Sliding Windows Detection (82.15% of Topline accuracy for CD), while the latter consistently consumed a smaller number of true labels (16.37% on average). We also note that detecting drifts based on the probability estimates led to more similar accuracy rates than the ones obtained by detecting drifts based on the feature space, yet consistently requiring fewer labels. Thus, using probability estimates is a promising approach for high dimensional data, if the classifier at hand is compatible. Additionally, mixing values from the feature space, probability estimates and outputted labels together is a possible alternative approach.

7. CONCLUSIONS

This work presented the Incremental Kolmogorov-Smirnov algorithm, a much faster method of recomputing the KS statistic for samples that change gradually over time. We evaluated the proposed test in data stream classification scenario, supporting a fast mechanism of concept drift detection. The classifiers were able to identify concept drifts without true label information and request only a portion of the true labels to adapt the models to new concepts. The usefulness of the Incremental Kolmogorov-Smirnov algorithm is not limited to this case study. We hope it to be widely applied in the future to empower different approaches for streaming problems, where a fast non-parametric hypothesis test may be needed.

As future work, we intend to develop new and more robust approaches to deal with detected concept drifts, such as the use of ensembles of previous models for recurrent concepts.

8. ACKNOWLEDGMENTS

The first author's research was supported by São Paulo Research Foundation (FAPESP) grants 2014/12333-7 and 2015/01701-8. The third author's research was supported by the Natural Sciences and Engineering Research Council of Canada, by the CALDO Programme, and by the National Research Centre of Poland (NCN) grant DEC-2013/09/B/-ST6/01549. The fourth author's research was supported by CNPq grants 03083/2013-1 and 446330/2014-0 and FAPESP grant 2013/50379-6.

9. REFERENCES

[1] R. Al-Otaibi, R. B. Prudêncio, M. Kull, and P. Flach. Versatile decision trees for learning over multiple contexts. In *ECML/PKDD*, pages 184–199. 2015.

[2] M. Amir and D. Toshniwal. Instance-based classification of streaming data using emerging patterns. In *ICT*, pages 228–236, 2010.

[3] C. R. Aragon and R. G. Seidel. Randomized search trees. In *FOCS*, pages 540–545. IEEE, 1989.

[4] G. E. A. P. A. Batista, E. J. Keogh, A. M. Neto, and E. Rowton. Sensors and software to allow computational entomology, an emerging application of data mining. In *KDD*, pages 761–764, 2011.

[5] A. Bifet and R. Gavalda. Learning from time-changing data with adaptive windowing. In *SDM*, pages 443–448, 2007.

[6] A. Bifet and R. Gavaldà. Adaptive learning from evolving data streams. In *IDA*, pages 249–260. 2009.

[7] A. Bifet, G. Holmes, and B. Pfahringer. Leveraging bagging for evolving data streams. In *ECML/PKDD*, pages 135–150. 2010.

[8] A. Dal Pozzolo, G. Boracchi, O. Caelen, C. Alippi, and G. Bontempi. Credit card fraud detection and concept-drift adaptation with delayed supervised information. In *IJCNN*, 2015.

[9] V. M. De Souza, D. F. Silva, G. E. Batista, et al. Classification of data streams applied to insect recognition: Initial results. In *BRACIS*, pages 76–81. IEEE, 2013.

[10] L. Devroye. A note on the height of binary search trees. *JACM*, 33(3):489–498, 1986.

[11] D. dos Reis. Fast unsupervised online drift detection using incremental kolmogorov-smirnov test, online supplementary material. https://github.com/denismr/incremental-ks, 2016.

[12] K. B. Dyer, R. Capo, and R. Polikar. Compose: A semisupervised learning framework for initially labeled nonstationary streaming data. *TNNLS*, 2013.

[13] H. Fanaee-T and J. a. Gama. Event labeling combining ensemble detectors and background knowledge. *Progress in Artificial Intelligence*, pages 113–127, 2013.

[14] J. Gama, P. Medas, G. Castillo, and P. Rodrigues. Learning with drift detection. In *SBIA*, pages 286–295. 2004.

[15] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan.

Table 2: Accuracy for each classifier / dataset (%).

Data	Nearest Neighbor					Naive Bayes						Decision Tree				
	BL1	BL2	MR	AB	TL	BL1	BL2	MR	MRP	CD	TL	BL1	BL2	MR	AT	TL
A	29.39	29.59	37.18	29.31	38.24	17.28	17.98	20.11	17.28	18.15	22.08	19.95	18.98	22.83	20.85	23.47
B	40.84	50.25	53.95	47.52	53.89	41.78	44.96	46.17	45.62	43.06	45.74	38.73	45.60	47.61	48.02	48.65
C	49.36	67.04	75.40	75.87	75.41	48.33	70.71	70.60	70.93	69.75	74.76	58.21	64.83	65.42	65.90	72.88
D	59.75	77.11	84.56	84.13	84.88	48.31	71.22	74.31	71.81	48.31	79.94	39.75	70.79	74.56	76.25	82.94
E	58.14	71.67	87.81	82.01	88.13	54.01	70.43	77.99	76.53	68.38	86.61	49.93	65.29	71.06	71.81	79.91
F	57.86	69.75	79.87	79.17	80.06	58.97	69.23	76.54	77.22	66.63	79.42	52.17	59.61	65.31	65.30	66.35

Table 3: Requested portion of true labels for each classifier / dataset (%).

Data	Nearest Neighbor					Naive Bayes						Decision Tree				
	BL1	BL2	MR	AB	TL	BL1	BL2	MR	MRP	CD	TL	BL1	BL2	MR	AT	TL
A	5.68	25.40	28.41	5.68	100.00	5.68	25.67	28.41	5.67	11.36	100.00	5.68	25.51	28.41	14.91	100.00
B	0.30	5.91	6.07	0.91	100.00	0.30	5.90	6.07	2.73	0.61	100.00	0.30	5.90	6.07	5.91	100.00
C	0.58	62.25	98.39	32.22	100.00	0.58	62.54	98.39	74.80	44.31	100.00	0.58	62.22	98.39	92.13	100.00
D	6.25	12.15	43.75	43.75	100.00	6.25	12.05	43.75	25.00	6.25	100.00	6.25	12.36	37.50	24.13	100.00
E	1.88	57.56	84.51	9.39	100.00	1.88	57.40	84.51	39.44	30.05	100.00	1.88	57.64	69.48	60.90	100.00
F	1.88	21.84	24.41	9.39	100.00	1.88	21.93	24.41	9.39	5.63	100.00	1.88	22.11	13.15	15.74	100.00

Clustering data streams. In *FOCS*, pages 359–366, 2000.

[16] N. Hammami and M. Bedda. Improved tree model for arabic speech recognition. In *ICCSIT*, volume 5, pages 521–526, 2010.

[17] A. Haque, L. Khan, and M. Baron. Semi supervised adaptive framework for classifying evolving data stream. In *PAKDD*, pages 383–394. 2015.

[18] D. J. Hill and B. S. Minsker. Anomaly detection in streaming environmental sensor data: A data-driven modeling approach. *Environmental Modelling & Software*, 25(9):1014–1022, 2010.

[19] B. Kaluža, V. Mirchevska, E. Dovgan, M. Luštrek, and M. Gams. An agent-based approach to care in independent living. In *AmI*, pages 177–186. 2010.

[20] L. I. Kuncheva et al. Nearest neighbour classifiers for streaming data with delayed labelling. In *ICDM*, pages 869–874. IEEE, 2008.

[21] R. H. Lopes. Kolmogorov-smirnov test. In *International Encyclopedia of Statistical Science*, pages 718–720. Springer, 2011.

[22] M. M. Masud, J. Gao, L. Khan, J. Han, and B. Thuraisingham. Classification and novel class detection in concept-drifting data streams under time constraints. *TKDE*, 23(6):859–874, 2011.

[23] M. M. Masud, C. Woolam, J. Gao, L. Khan, J. Han, K. W. Hamlen, and N. C. Oza. Facing the reality of data stream classification: coping with scarcity of labeled data. *KAIS*, 33(1):213–244, 2012.

[24] N. C. Oza. Online bagging and boosting. In *SMC*, volume 3, pages 2340–2345. IEEE, 2005.

[25] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4-5):464–497, 1996.

[26] V. M. Souza, D. F. Silva, J. a. Gama, and G. E. Batista. Data stream classification guided by clustering on nonstationary environments and extreme verification latency. In *SDM*, pages 873–881. SIAM, 2015.

[27] X. Wu, P. Li, and X. Hu. Learning from concept drifting data streams with unlabeled data. *Neurocomputing*, 92:145–155, 2012.

[28] I. Zliobaite. Change with delayed labeling: when is it detectable? In *ICDMW*, pages 843–850. IEEE, 2010.

APPENDIX

A. TREAP WITH LAZY PROPAGATION

A Cartesian Tree is defined as binary tree which has the following properties:

1. It has the heap ordering property, *i.e.* a non-leaf node has higher priority than its children;
2. It is built upon a sequence of numbers and the in-order traverse of the tree results in the same original sequence of numbers. In other words, the elements in the left subtree of a node are values that appeared earlier in the sequence than the root. The values in the right subtree appeared later than the root.

We can define a merge operation between two already existing Cartesian Trees. One of them is called the *left* tree and the other the *right* tree. Such merge can be used to build a Cartesian Tree from scratch, simply considering that the right tree is a single node with the next observation from the sequence. After the merging, the following constraints must hold: the resultant tree still has the heap ordering property and an in-order traverse in it is equivalent to an in-order traverse in the left tree, followed by an in-order traverse in the right tree.

Such merge operation is achieved recursively. Let L be the left tree and R be the right tree. If L's root has a greater priority than R's root, then L's root is the resultant tree's root Z. Z's left subtree is L's left subtree, preserving the in-order traverse to the left and the heap priority property. Z's right subtree is a recursive merge between L's right subtree – as left tree – and R – as right tree. If the priority of R's root is greater than the priority of L's root, the solution is analogous. The process is summarized by the following algorithm.

```
function MERGE(NIL, NIL)
    return NIL
end function

function MERGE(LeftTree, NIL)
    return LeftTree
end function

function MERGE(NIL, RightTree)
    return RightTree
end function

function MERGE(LeftTree, RightTree)
```

```
    if LeftTree.RootPriority > RightTree.RootPriority then
        LeftTree.RightSubtree ← Merge(LeftTree.RightSubtree, RightTree)
        return LeftTree
    else
        RightTree.LeftSubtree ← Merge(LeftTree,
RightTree.LeftSubtree)
        return RightTree
    end if
end function
```

A Treap is a special case of Cartesian Tree where each node has, apart from the heap priority value, an additional key. From the perspective of the priority, the tree is a heap, but from the perspective of the additional key (BST key), it is a binary search tree. A tree that meets both criteria is possible for any set of priority values and BST keys. The node with highest priority will be the root of the tree. The root of the left subtree will be composed by nodes with BST key smaller than the root's BST key, and the right subtree with BST keys greater than or equal to the root's BST key. The same idea recursively applies to the remainder of the tree.

The Treap is easily achieved by construction, using the merge algorithm defined for Cartesian Trees. In a merge operation, if the left and the right trees are both Treaps and all BST keys from the left tree are less or equal than the BST keys from the right tree, then the resultant tree is also a Treap. Figure 3 illustrates the merge operation of two Treaps.

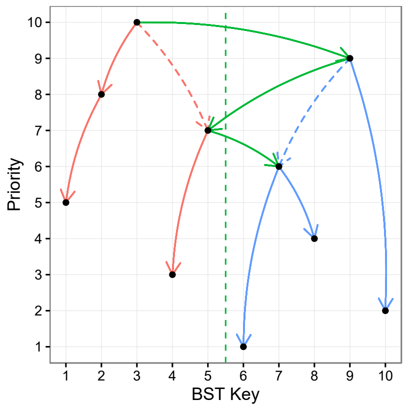

Figure 3: Merge operation between two treaps. The left tree is red and on the left side of the dashed vertical line. The right tree is blue and on the right side of the dashed vertical line. The arrows correspond to father–children relationships. The result tree's new relationships are green and removed relationships are dashed.

Inserting a new pair (priority value, BST key) into an already existing Treap may require a split operation. The following algorithm describes how this operation can be achieved. The input is a Treap and a BST key k. Two Treaps are the output. The first (left tree) contains all the elements from the original Treap that have lower BST keys than k. The second (right tree) contains the remaining elements, including the ones which have the same BST keys as the one that was applied in the query.

```
function SPLIT(NIL, ?)
    return NIL, NIL
end function

function SPLIT(Tree, Key)
    if Key ≤ Tree.BSTKey then
        Left, Tree.RightSubtree ← Split(Tree.LeftSubtree, Key)
        Right ← Tree
    else
        Tree.RightSubtree, Right ← Split(Tree.RightSubtree, Key)
        Left ← Tree
    end if
    return Left, Right
end function
```

Finally, with both *Merge* and *Split* operations, we can insert new elements to an already existing Treap by splitting it and re-merging it adequately. The following algorithm describes this process.

```
function INSERT(Tree, Priority, Key)
    TrivialTree ← NewTreeNode(Priority, Key)
    Left, Right ← Split(Tree, Key)
    return Merge(Merge(Left, TrivialTree), Right)
end function
```

The time complexity for both *Merge* and *Split* depends linearly on the depth of the Treap. For a list of pairs of unique keys and unique priorities, there is only one possible Treap, as the in-order traverse is locked by keys and the balance – *i.e.* the father-children relationships – is locked by the ranking of the priorities. In other words, the resulting Treap after the insertion of all the elements of such list, in any order, is always the same. The probability distribution of the priorities is not a matter of concern, provided that they are unique, since the Treap only uses their ranks. Specifically, the ranking of the priorities is analogous to the order of the elements when constructing a basic binary search tree at once. Figure 3 illustrates this fact by plotting a Cartesian Tree on a Cartesian Plane, where the priority corresponds to the y-axis and the key corresponds to the x-axis. Each node is father of both the topmost node to its left and the topmost node to its right. If we insert all the keys from this figure using only basic binary search tree insertion, from top to bottom, we obtain exactly the same tree. Therefore, if the priorities are chosen at random, the Treap inherits a common property of randomized binary search trees that are constructed at once: the height of the tree grows logarithmic with high probability [10]. Consequently, *Merging* and *Splitting* a Treap and inserting and removing elements, logarithmic in time, according to the total number of elements. Figure 4 empirically shows the logarithmic growth of the Treap.

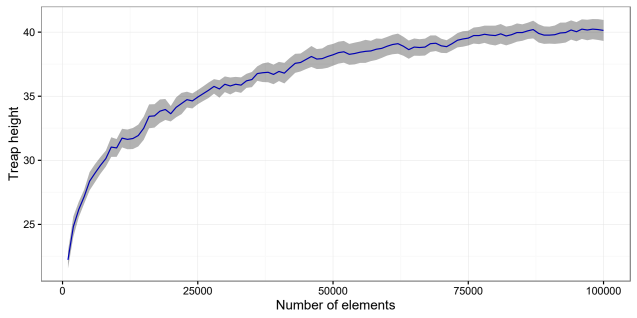

Figure 4: Height growth of a treap. The curve is an average of the heights of 30 cartesian trees and the shaded band indicates 95% confidence intervals.

As the priorities are fixed, if a node is predecessor of another one, it will never become the other way around. This property can be exploited so that we can easily put summary information about a subtree in its root and keep this information up to date, without interfering in the complexity of the operations that were presented so far. As an example, let us say that now each node of the tree has an additional and arbitrary value. The following algorithm makes changes in the previous operations in order to also efficiently compute the minimum and maximum values for each complete subtree.

```
function Update(Tree)
    Tree.Max ← max{ Tree.Value, Tree.LeftSubtree.Max, RightSubtree.Max}
    Tree.Min ← min{ Tree.Value, Tree.LeftSubtree.Min, RightSubtree.Min}
end function

function Merge(LeftTree, RightTree)
    if LeftTree.RootPriority > RightTree.RootPriority then
        LeftTree.RightSubtree ← Merge(LeftTree.RightSubtree, RightTree)
        Tree ← LeftTree
    else
        RightTree.LeftSubtree ← Merge(LeftTree, RightTree.LeftSubtree)
        Tree ← RightTree
    end if
    Update(Tree)
    return Tree
end function

function Split(Tree, Key)
    if Key ≤ Tree.BSTKey then
        Left, Tree.RightSubtree ← Split(Tree.LeftSubtree, key)
        Right ← Tree
    else
        Tree.RightSubtree, Right ← Split(Tree.RightSubtree, key)
        Left ← Tree
    end if
    Update(Left)
    Update(Right)
    return Left, Right
end function
```

Similarly, we can also efficiently perform bulk operations that change all values in a tree, if these operations are *suitable*. An operation is suitable if its resulting summaries for the whole tree can be computed using only the already existing summaries and **without** accessing the subtrees. The procedure of propagating a bulk operation from a root to its subtress only when it is strictly necessary is known as lazy propagation. It should be performed in the same time complexity as the complexity of applying the bulk operation in the root, usually $O(1)$. The following algorithm presents the needed modifications to obtain a new operation to our tree, *IncreaseBy*, which adds a constant to all values of the tree. This new operation is $O(1)$ and it does not change the complexity of the other operations.

```
function IncreaseBy(Tree, Constant)
    Tree.Value ← Tree.Value + Constant
    Tree.Max ← Tree.Max + Constant
    Tree.Min ← Tree.Min + Constant
    Tree.Lazy ← Tree.Lazy + Constant
                                        ▷ Tree.Lazy is initially 0
end function

function Unlazy(Tree)
    IncreaseBy(Tree.LeftSubtree, Tree.Lazy)
    IncreaseBy(Tree.RightSubtree, Tree.Lazy)
    Tree.Lazy ← 0
end function

function Update(Tree)
    Unlazy(Tree)
    Tree.Max ← max{ Tree.Value, Tree.LeftSubtree.Max, RightSubtree.Max}
    Tree.Min ← min{ Tree.Value, Tree.LeftSubtree.Min, RightSubtree.Min}
end function

function Merge(LeftTree, RightTree)
    if LeftTree.RootPriority > RightTree.RootPriority then
        Unlazy(LeftTree)
        LeftTree.RightSubtree ← Merge(LeftTree.RightSubtree, RightTree)
        Tree ← LeftTree
    else
        Unlazy(RightTree)
        RightTree.LeftSubtree ← Merge(LeftTree, RightTree.LeftSubtree)
        Tree ← RightTree
    end if
    Update(Tree)
    return Tree
end function

function Split(Tree, Key)
    Unlazy(Tree)
    if Key ≤ Tree.BSTKey then
        Left, Tree.RightSubtree ← Split(Tree.LeftSubtree, key)
        Right ← Tree
    else
        Tree.RightSubtree, Right ← Split(Tree.RightSubtree, key)
        Left ← Tree
    end if
    Update(Left)
    Update(Right)
    return Left, Right
end function
```

A new version of the Split operation is needed in order to remove an element (given a key) from the Treap. We call it SplitFirst. After applying this operation, we obtain a left tree with one element that contains the smallest key in the original tree, and a right tree with the remaining elements. The following algorithm describes it. Additionally, it describes the SplitLast method, which produces a right tree with the element that has the greatest key and a left tree with the remaining elements.

```
function SplitFirst(Tree)
    if Tree is NIL then return NIL, NIL
    end if
    Unlazy(Tree)
    if Tree.LeftSubtree is not NIL then
        Left, Tree.LeftSubtree ← SplitFirst(Tree.LeftSubtree)
        Right ← Tree
    else
        Right ← Tree.RightSubtree
        Tree.RightSubtree ← NIL
        Left ← Tree
    end if
    Update(Left)
    Update(Right)
    return Left, Right
end function

function SplitLast(Tree)
    if Tree is NIL then return NIL, NIL
    end if
    Unlazy(Tree)
    if Tree.RightSubtree is not NIL then
        Tree.RightSubtree, Right ← SplitLast(Tree.LeftSubtree)
        Left ← Tree
    else
        Left ← Tree.LeftSubtree
        Tree.LeftSubtree ← NIL
        Right ← Tree
    end if
    Update(Left), Update(Right)
    return Left, Right
end function
```