Self-Adaptive Induction of Regression Trees

Raúl Fidalgo-Merino and Marlon Núñez

Abstract—A new algorithm for incremental construction of binary regression trees is presented. This algorithm, called SAIRT, adapts the induced model when facing data streams involving unknown dynamics, like gradual and abrupt function drift, changes in certain regions of the function, noise, and virtual drift. It also handles both symbolic and numeric attributes. The proposed algorithm can automatically adapt its internal parameters and model structure to obtain new patterns, depending on the current dynamics of the data stream. SAIRT can monitor the usefulness of nodes and can forget examples from selected regions, storing the remaining ones in local windows associated to the leaves of the tree. On these conditions, current regression methods need a careful configuration depending on the dynamics of the problem. Experimentation suggests that the proposed algorithm obtains better results than current algorithms when dealing with data streams that involve changes with different speeds, noise levels, sampling distribution of examples, and partial or complete changes of the underlying function.

Index Terms—Machine learning, mining methods and algorithms, knowledge acquisition, heuristics design.

1 Introduction

Huge and potentially infinite volumes of data are often continuously generated by real-time systems like management communications networks, online transactions in the financial market or real industry, scientific and engineering experiments, surveillance systems, and other dynamic environments, producing *data streams*. In order to achieve better productivity, learning algorithms are able to construct models from these data containing the hidden underlying relations (or an approximation of them) between the independent features and the dependent one. If the dependent attribute is symbolic, then learning algorithms build *classification* models to approximate an underlying concept; if the dependent attribute is numerical, then the learning algorithms build *regression* models to approximate an underlying function.

Traditional learning algorithms can be executed in batch mode in order to obtain models representing relations in data. Incremental algorithms learn their models online, updating them as new data are available. A real problem arises when there are configuration changes in the system or changes in the nature of data. For instance, solar activity of the sun changes gradually from years with almost no solar activity to years with violent activity. People also change their interests (i.e., reading topics, clothes, and friend profiles) over time, and these changes could be abrupt, gradual, partial, or complete. If a data-stream-driven learning algorithm is expected to learn from complex systems like the ones mentioned above, they should adapt automatically to any kind of change in the underlying dynamic of the observed problem.

• *The authors are with the Departamento de Lenguajes y Ciencias de la Computacion, University of Malaga, Complejo Tecnologico, Campus de Teatinos, 29071 Malaga, Spain. E-mail: rfm@uma.es, mnunez@lcc.uma.es.*

Manuscript received 17 Feb. 2010; revised 10 Sept. 2010; accepted 2 Jan. 2011; published online 18 Jan. 2011.
Recommended for acceptance by K. Murphy.
For information on obtaining reprints of this article, please send e-mail to: tpami@computer.org, and reference IEEECS Log Number TPAMI-2010-02-0105.
Digital Object Identifier no. 10.1109/TPAMI.2011.19.

Currently, as far as we know, there is no regression algorithm that can learn incrementally in changing environments without an a priori configuration. All current algorithms need the user to provide an initial setup based on a conscientious study of the possible dynamics the data stream will have. As this is a complex, subjective, and (probably) incomplete process, there is no guarantee that these algorithms would be able to adapt to different conditions.

When dealing with dynamic data streams derived from complex systems, the underlying relations between variables may change over time. These changes will not always occur with the same speed: Sometimes they are faster, when the function shifts roughly from its original position (that is, the function changes abruptly), and others are slower, when changes are produced by slight movements of the function (that is, the function changes gradually). On the other hand, changes may also appear partially or totally. The underlying function may only drift for some values of the attributes (i.e., regions or parts of the function) that compose a tuple. Furthermore, different regions of the function can change at different speeds, as previous studies in the field of information retrieval [1] and data mining [2], [3] have found.

This can be illustrated by making use of a simple univariate function as the underlying relation to be learned. Suppose that the initial function in the data stream is a straight line (solid line in Fig. 1a). After some time, the function moves up by incrementing its interception in some small Δ (dotted lines). This process is repeated several times until the function reaches the final position (dashed line). This succession of small changes represents gradual changes in the whole function.

Changes in regions of the function can be illustrated by using the previous example. Fig. 1b illustrates the case of a function that changes at different speeds. Suppose initially that the function to be learned is a straight line (solid line). After some time, regions near the center suffer gradual changes, while abrupt changes appear at the

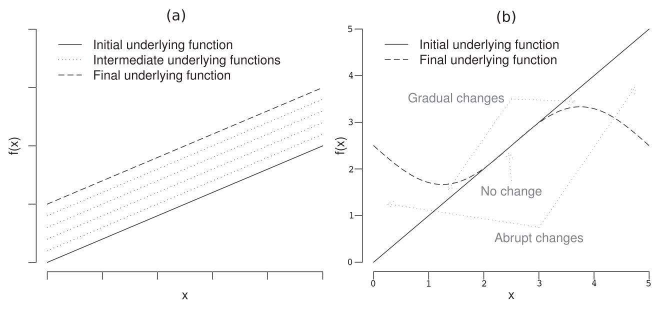

Fig. 1. Illustrative examples of changes in the underlying function: (a) Complete and gradual change. (b) Partial and abrupt/gradual change.

extreme regions. The center of the function does not change at any time.

A well-known technique to learn models in these conditions is to forget past examples and take into account new data, updating the model accordingly. With regard to forgetting mechanism, learning algorithms may use global or local windows, with fixed or adaptive size. Windows contain information about recent data from the data stream. The strategy of managing one global fixed window is not able to track the dynamics of a complex problem. If a small global window is used, then the learning algorithm will adapt its model quickly to abrupt changes, but might not work well when there are gradual changes or no changes in the function. On the other hand, if the global fixed window is large, the learner will adapt its model well when gradual or no changes are made, but will probably manage inconsistent information when fast changes are present. By using a global adaptive window, changes in the whole function can be addressed, but when these changes are produced in some regions, this strategy will produce poor results. So, it would seem that a strategy oriented to devoting an adaptive window to each part of the function (which we called *local windows*) covers the case of changing regions with different speeds.

Since our proposal is an incremental regression tree, each branch from the root of the tree to a leaf represents a region of the function. The purpose of the presented method, called Self-Adaptive Induction of Regression Trees (SAIRT), is to deal with tasks involving unknown dynamics. This method incrementally learns a regression tree in which each leaf maintains a local window, used to forget examples when a change in a region of the underlying function has been detected. SAIRT also adapts the regression tree structure and adjusts the local parameters in order to fulfill the current underlying function by improving local performance, optimizing the number of stored examples, and reducing processing time. The method may also face tasks with both numerical and symbolic attributes, a changing level of noise, and/or in the sampling distribution of the data (virtual drift).

This paper is organized as follows: Section 2 summarizes the related work, in Section 3, the proposed algorithm is explained, and Section 4 covers the experiments performed as well as comparisons with other algorithms. Finally, Section 5 describes future work and Section 6 presents conclusions.

2 Related Work

Although, in the traditional task of stationary regression learning (i.e., approximation of a function from a static database), several algorithms have been provided, construction of regression models from a stream of data in which different unknown dynamics may occur has received less attention from the community.

On the other hand, the field of concept learning on data streams has been an effervescent area in the recent past, even when changes are present. The works by Domingos and Hulten [4] and Núñez et al. [3] are notable when dealing with dynamic data streams for classification. The following paragraphs try to summarize the current state of the art in the field of regression induction from dynamic data streams by remarking on the most important characteristics (from our point of view) of the algorithms that face these problems: memory management and adaptation of internal parameters.

Regarding memory management, Maloof and Michalski [5] suggested that learning algorithms follow one of these three possibilities for the memory model when dealing with past training examples: *No instance memory*, in which the incremental learner retains no examples in memory but instead statistics about them (e.g., FIMT and FIRT-DD algorithms [6], [7], Online-RD/RA algorithms [8], and neural networks build this kind regression models), *Full instance memory*, in which the method retains all past training examples (e.g., M5 [9], CART [10], and IBk [11] are regression algorithms following this approach), and *Partial instance memory*, which is a strategy mainly oriented to dealing with changes in underlying patterns, and learning algorithms retain some of the past training examples that are within a window. The size of this window may be fixed or adaptive. AddExp method [12] and ER algorithm [13] inherently use adaptive windows of examples by creating and destroying regression models. Another way of making use of this strategy is to learn, from a window of examples, a regression model using one of the previously commented algorithms.

Another interesting aspect is the management of the internal parameters for dealing with real applications. This allows the algorithm to learn from different conditions without user action. When facing problems in which patterns change over time, almost all learning systems (e.g., FLORA [14], FRANN [15], or CVFDT [16] among others in the classification case, and FIRT-DD, ER, and AddExp in the regression case) require a carefully a priori establishment of a series of parameters in order to face a particular dynamic of the problem (e.g., noise level, speed of change, and/or temporal distribution of the examples). The main drawback is that the values set for their parameters may produce not useful models in the future because of the dynamics present in the stream [3], [17], [18], [19], for example, in those problems where the patterns obey certain subjectivity (user interests). In their research on classification tasks, Klinkenberg and Joachims [17] and Núñez et al. [3] proposed nonparameterized approaches for handling a global time window and several local time windows, respectively.

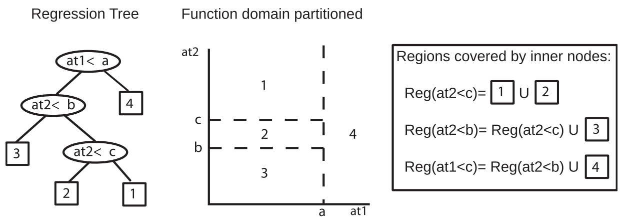
Regression Tree
Function domain partitioned
at1< a
at2< b
4
3
at2< c
2
1
at2
c
b
1
2
3
4
a
at1
Regions covered by inner nodes:
Reg(at2<c)= 1 U 2
Reg(at2<b)= Reg(at2<c) U 3
Reg(at1<c)= Reg(at2<b) U 4

Fig. 2. Example of regression tree and the partition of the function domain it produces.

3 DESCRIPTION

In this paper, a new method for inducing binary regression trees is presented. It takes advantage of its partition capability to easily monitor the dynamics in regions of the function. Binary regression tree algorithms divide the function domain using an iterative and data-driven process. Starting from the whole data set (that represents the function domain), it is split in two smaller subsets by choosing the *independent attribute-value* pair (e.g., $D(color = red) = \{true, false\}$) that maximize the reduction of deviation of the dependent value between the original set and the resulting subsets [9]. For continuous attributes, some authors choose a cut point as *value* (e.g., $D(age < 40) = \{true, false\}$) using the k-means algorithm [20] (with $\mathrm{k} = 2$, which takes linear complexity for unidimensional clustering [21]). This is repeated for each resulting subset whenever it is not accurate and simply represented (e.g., by a constant). The aim is to produce subsets easier to induce than their predecessors at each step. Fig. 2 gives an example of a binary regression tree and how its leaves and inner nodes partition the function domain.

SAIRT is an algorithm for the incremental induction of binary regression trees which also supports adaptability to gradual and abrupt function drift, the handling of symbolic and numeric attributes, and robustness to noise and virtual drift in data. Depending on the dynamics of the problem, this method expands or prunes subtrees and adjusts its internal parameters to improve the local performance measure in each node. SAIRT develops a partial memory management: It selectively forgets examples and stores the remaining ones in local windows present in the leaves of the tree. This method is designed to be used under unknown dynamics, and thus it is able to automatically adapt its parameters for each problem.

We consider data streams as sequences of examples labeled with a time stamp. This means that the method is capable of dealing with streams of examples in continuous (real) time. A consecutive index can also be used as a time label. As previously commented, an example is described by attribute-value pairs (independent variables) and a numeric-dependent value as label.

In essence, when a new example from the data stream must be processed, actions to be performed by SAIRT on the induced model can be grouped in three stages: testing the coherence of nodes in the path of the example (a node is coherent whenever its split contributes with the induction of the underlying function), treatment of a leaf or a noncoherent node, and updating statistics of visited nodes.

In the process of learning, SAIRT maintains some information in the model, explained in Section 3.1. Section 3.2 describes the adaptive mechanisms used to face data streams with unknown dynamics. Section 3.3 presents the algorithm and details its stages. Finally, in Section 3.4, a complexity analysis of SAIRT is provided.

3.1 Information Stored in the Model

SAIRT stores some information in each node of the regression tree in order to apply the adaptive mechanisms to fit the model to the dynamics of the data streams. These measures are:

- *Performance*. Every node in the model (whether inner node or leaf) calculates a quality measure referred to as *performance*. It summarizes how the node fulfills its related part of the function and, in the case of leaves, this measure is used to make decisions about forgetting examples and expansions. *Performance* is calculated via the *instantaneous error rate* (*ier*), defined in a leaf by the following equation:

$$ier_{leaf} = \frac{\sum_{e \in examples_{leaf}} (y_e - \hat{y})^2}{|examples_{leaf}| \cdot (y'_{max} - y'_{min})^2},$$

where $examples_{leaf}$ is the set of examples maintained in the leaf, y_e is the dependent value for example e, $\hat{y}$ is the dependent value estimated by the model, and y'_{max} and y'_{min} are the maximum and minimum regression value in the set of examples without considering outliers (by applying the interquartile range method by Tukey [22]). At inner nodes, it is calculated as $ier_{node} = \sum_{i \in children_{node}} \frac{|examples_i|}{|examples_{node}|} \cdot ier_i$.

This instantaneous measurement varies greatly under certain conditions, mainly when dealing with noise and after forgetting of several examples. Due to this high variability, instantaneous error rate alone is inappropriate to make decisions.

For this reason, we have used a smoothing formula in order to make robust decisions based on *ier*. Exponential smoothing is commonly used by control systems (e.g., to discard or retransmit messages in the nodes of a network, controlling the congestion in them). This method needs a parameter to control the level of smoothness (α), that is, the importance of past history. We have decided to fix as default the same value used by Nagle [23] and other authors [24], [25] ($\alpha = \frac{7}{8}$), and that has been widely studied in different conditions and has also been demonstrated in our experimentation to be valid for several problem configurations. This allows us to have a robust measure in order to make decisions and to quickly respond to changes in parts of the function.

So, the performance in each node is calculated by

$$perf(t) = \frac{7}{8} perf(t-1) + \frac{1}{8}(1 - ier),$$

where $perf(t)$ and $perf(t-1)$ are the current and previous performance measure of the node, respectively, and *ier* is the instantaneous error rate of that node.

A change in the tendency of *performance* determines the *state* of the node.

- *State*. Using the mentioned performance measure, SAIRT employs a state diagram in each leaf and inner node to find out if it is in one of the following states:
    - *Degradation state*. It means that the associated region of the function may have change. A node has this state whenever its performance worsens. In the case of leaves, they must react accordingly, usually forgetting examples (see Section 3.2.1).
    - *Improvement state*. This means that its related region of the underlying function is being learned adequately. A node is in this state whenever its performance improves. Leaves in this state may also carry out actions to forget examples in order to optimize the model (see Section 3.2.1).
- *Distribution of dependent values*. Inner nodes and leaves store the distribution of dependent values of the examples that traversed that node and are still maintained in the model.
- *Examples*. Each leaf stores a subset of examples from the data stream. These examples belong to its associated region of the function, and are managed using the adaptive mechanism presented in Section 3.2.1.

3.2 Adaptive Features

The aim of the SAIRT algorithm is to adapt its model to the current dynamics of the data streams, even to some parts of the underlying function to learn. This section describes the mechanisms that used to reach this objective: management of local adaptive windows present in leaves and structural adaptation of the model (expanding and/or pruning subtrees).

3.2.1 Local Window Management

As said previously, each leaf of the regression tree stores a window of examples. It is referred to as a *local (adaptive) window*. By managing these windows, the model accumulates or forgets examples from different regions of the underlying function depending on the dynamics present in the data stream.

Pseudocode 1 shows how the management of the local window is carried out. The current state of a leaf (explained in Section 3.1) leads the adaptive mechanisms of its local window size.

Pseudocode 1: AdaptLocalWindow

```
AdaptLocalWindow(leaf)
  IF InImprovementState(leaf) THEN
    IF $examples_{leaf} > examples_{brother}/perf_{leaf}$ THEN
      ForgetExamplesUntil(Time(OldestExample(leaf))+1, leaf)
    ENDIF
  ELSE – leaf in Degradation State –
    dropPerf=$perf_{leaf}(t) - perf_{leaf}(anTime)$
    dwff=$min\{pers_{leaf}(t) \cdot dropPerf, 1\}$
    olimit= $oldestTime_{leaf} + (anTime - oldestTime_{leaf}) \cdot dwff$
    ForgetExamplesUntil(olimit,leaf)
  ENDIF
```

- *Local window size in improvement state*. When a leaf is in the improvement state, its local window usually grows accumulating examples to better induce the underlying patterns.

  However, in this state, the leaf may also lose examples. This is because a well-constructed leaf with a high performance should discard older examples on receiving new ones to avoid excessive accumulation of examples. The heuristic used is based on comparisons of the leaf with its brother node (whether leaf or inner node). If the performance of the leaf is low, the upper boundary for the local window size is large; otherwise, the number of examples in the brother node is taken into account to control the number of examples in the leaf. Fig. 3a shows how the performance of the leaf increases, accumulating examples. When it is high enough (t_1), the local window slides until t_2, preserving its size and forgetting older examples.

  When the tree is functioning well (i.e., every node has high performance), the number of examples beneath the two branches of any node should be balanced. Any imbalance results in an older example being forgotten by a leaf.
- *Local window size in degradation state*. As stated before, when a leaf is in degradation state, a change is supposed to occur in its associated region of the function. With the aim of adapting the leaf to the new conditions, the algorithm may forget past examples, reducing its local window to improve its local performance.

  Fig. 3b shows how the performance of a leaf decreases when there is a change in its associated region. When the performance starts to decline, the leaf enters into *Degradation State* and that time is noted as *Anomaly time* (*anTime*). Then, a local window of examples (*Delayed Window*) is composed of the examples between the time of the oldest example in the leaf and *anTime*. These examples are presumed to belong to the previous dynamic.

  As shown in Fig. 3 and Pseudocode 1, deterioration in the performance of a leaf reduces the delayed window by a fraction of time. If the deterioration persists with following examples, a change in function is more probable and a greater fraction of the window needs to be discarded. Thus, the fraction of the delayed window to forget (*dwff*) is proportional

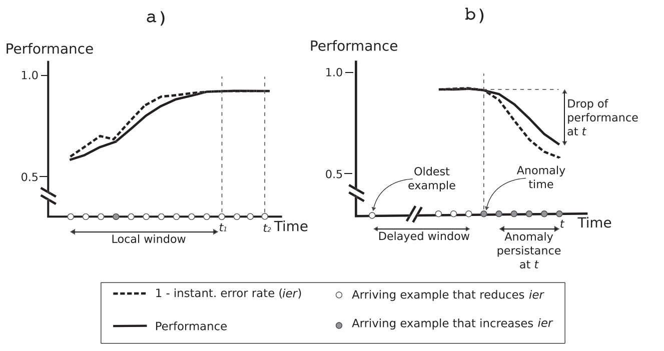

Fig. 3. Illustration of local window management in (a) the improvement state and (b) the degradation state.

to the drop of performance (*dropPerf*) from *anTime* and the current persistence ($pers_{leaf}$).

Then, the oldest limit for the delayed window (*olimit*) is calculated and examples that arrived to the leaf before that time are forgotten. Examples that are after the anomaly time are presumed to belong to the current function and should be maintained.

3.2.2 Structural Adaptation

In addition to the management of local windows, other mechanisms are needed to adapt the model structure to the current dynamics. They are detection of inconsistent inner nodes and expansion/labeling of leaves.

- *Coherence of inner nodes*. SAIRT performs tests at inner nodes to know if their splitting attributes are still valid in the induction of the underlying function. SAIRT uses techniques based on test of hypothesis for continuous distributions, equivalent to the ones used in symbolic domains (widely used and studied for prepruning decision trees [26], [27]). Specifically, we employ the *Kolmogorov-Smirnov* hypothesis test (with a significance level of 0.05) using as the null hypothesis the fact that the distributions of dependent values in the parent node (ddv_{node}) and in its descendant nodes ($ddv_{children}$) are similar. This test compares the distributions and rejects the null hypothesis when the split brings something statistically significant to the induction of the function (i.e., the splitting attribute is valid). If these distributions are similar, the null hypothesis is not rejected (i.e., the split is not useful) and the node must be adjusted. This process guarantees that any node (even the root of the tree) can be dropped if the underlying patterns drift.
- *Local expansion/labeling criterion*. Most supervised learning methods use some parameters to decide when to stop learning or when to make a decision. Regression tree induction needs an expanding/labeling criterion to prevent an unneccessary growing of the model (e.g., when noise is present). For example, some algorithms decide if a leaf is expanded or a linear model is calculated as its label [9]. Recent advances in the field of regression induction over time-changing data streams [7] use Hoeffding bounds as expansion criteria [28] (heuristic with several critical parameters that should be fixed a priori). Both approximations are not appropriate for our model as the time complexity should be optimized and the user should not change the parameter settings depending on the current dynamics of the function.

  As far as we know, there is no theoretically supported criterion to decide when to stop growing a regression tree, and which is capable of functioning without parameters. Therefore, we design a heuristic that prevent the leaf of being expanded when it contains few examples or has low error rate. It is formalized as

$$examples_{leaf} \cdot (1 - ier) > e^{examples_{leaf} \cdot ier}.$$

  This heuristic compares an estimation of the examples producing low error rate in the leaf with an estimation of the examples with high error rate in exponential factor. The experimentation, conducted using different problems of varying dynamics, led us to the conclusion that the exponential factor should be near to e.

3.3 The SAIRT Algorithm

Once presented the metrics and adjustable local parameters contained in the leaves of the model, this section provides the detailed explanation of the SAIRT algorithm (see Pseudocode 2).

Pseudocode 2: SAIRT algorithm

```
SAIRT (tree, example)

  node = Root(tree)                                              ⎫
  visited_nodes = {node}                                         ⎪
  WHILE !IsLeaf(node) AND IsCoherent(node) DO                    ⎬ Stage 1
    node = SelectNextChild(node, example)                        ⎪
    visited_nodes = visited_nodes ∪ {node}                       ⎪
  ENDWHILE                                                       ⎭

  IF IsLeaf(node) THEN        – node is called leaf –            ⎫
    Store(example, leaf)                                         ⎪
    UpdateStatistics({leaf})                                     ⎪
    node = TryExpansion(leaf)                                    ⎪
    IF IsLeaf(node) THEN                                         ⎪
      AdaptLocalWindow({leaf})                                   ⎪
    ENDIF                                                        ⎬ Stage 2
  ELSE      –node is not coherent–                               ⎪
    Drop(example, node)                                          ⎪
    AdaptLocalWindow(DegradedLeaves(node))                       ⎪
    leaf = Prune(node)                                           ⎪
    UpdateStatistics({leaf})                                     ⎪
    TryExpansion(leaf)                                           ⎪
  ENDIF                                                          ⎭

  UpdateStatistics(visited_nodes)                                } Stage 3
```

Throughout this section, Fig. 1b will be used to illustrate the stages that make up the algorithm. Thus, suppose that after inducing a model from an original function (solid line), enough examples from the new function (dashed line) to produce reactions arrives from the data stream. These reactions will be commented in the different stages of the algorithm.

3.3.1 Stage 1: Test Coherence of Nodes in Path

With each example from the data stream, SAIRT checks the coherence of the inner nodes (see Section 3.2.2) it traverses to find out if the splitting attribute is useful or not in the induction of its associated region of the function.

Whenever the node is coherent to the current function, the next node to visit is obtained by directing the example along the appropriate branch, using the value in the example of the splitting attribute in the current node.

In the example of Fig. 1, only the nodes associated with the gradual changing region (e.g., $x \in [1 \dots 2]$) will become noncoherent because the distributions of the dependent values of these nodes and their children are similar (e.g., $y \in [1.7 \dots 1.8]$).

When SAIRT finds a noncoherent node or arrives at a leaf, this stage finishes and the next one begins.

3.3.2 Stage 2: Treatment of a Leaf or a Noncoherent Node

Depending on the result of the previous phase, different actions would be carried out to adapt the model.

- *Stage 2a. Leaf treatment*. Once SAIRT arrives at a leaf, the algorithm process the information contained in it with the aim of a better adjustment of the model whether trying an expansion of the leaf or forgetting past examples (if needed).

  Specifically, SAIRT initially stores the example in the leaf and updates the local performance, state, and local distribution of dependent values. In order to take advantage of all available memory, SAIRT does not increment the number of stored examples when the physical memory limit is reached. That is, if the memory limit is reached, it forgets the oldest example of the local window whenever a new one arrives to the leaf. SAIRT also computes the median value of this distribution as the label to be returned when a prediction is required from the leaf.

  At this point, the leaf may need some adjustments for better adaptation to the associated region of the underlying function. First, SAIRT checks the expansion/labeling parameter in the leaf before expand it one level (applying the reduction on deviation criterion saw in Section 3). If the expansion is allowed, the new node must pass a coherence test before substitute the leaf. If, after these attempts, the leaf is not expanded, no structural improvements can be done with its data and an adjustment of its local window is done.

  This situation is seen in the abrupt changes regions of Fig. 1b (e.g., $x \in [0 \dots 1]$). They remain coherent, but the leaf must either be expanded or relabeled after forgetting of examples from the previous function. In the case of regions without changes, expansions are not allowed by the expansion/labeling parameter and the adjustment of the local window in improvement state is done. Early steps after the change produces forgetting of past examples in the gradual regions.
- *Stage 2b. Noncoherent node treatment*. If SAIRT detects a nonuseful splitting attribute (i.e., stops on a inner node), the algorithm substitutes that node with a new structure built depending on the current dynamics of its associated region.

  Specifically, at the beginning of this phase the example is dropped down the tree (without checking nodes) until its corresponding leaf, where it is stored and the statistics of the leaf are updated. As a change is supposed to occur in the associated region of the noncoherent node, delayed windows of degraded leaves below that node are adjusted as described in Section 3.2.1. Then, the surviving examples are collected to build a new leaf that substitutes the noncoherent node (i.e., the node is pruned). Finally, an attempt to reconstruct the new leaf is performed (as described previously).

  This phenomenon can be observed in the gradual changes regions of Fig. 1b. After forgetting of examples of the previous function, the distributions of dependent values of inner nodes and their children in these regions become similar (e.g., $x \in [1 \dots 2]$), producing adaptation of delayed windows, pruning, and reconstruction of those parts of the model involved in these changes.

3.3.3 Stage 3: Upward Updating of Statistics

Once stage 2 has been completed, SAIRT updates the local performance and state of each visited node, as seen in Section 3.1. Also, the values of the dependent attribute in the examples forgotten in the previous stage, if any, are removed from the local distribution of the dependent attribute. This process starts at the last visited node and finishes at the root of the tree. The aim is to have an up-to-date model at the end of this stage.

3.4 Complexity Issues

The complexity analysis of the proposed algorithm done in this section covers two opposite situations usually found when learning with SAIRT: New examples do not produce changes in the induced model and new examples produce selective forgetting or a restructuring of the model.

In the case of a static function and once a tree has been constructed that does not change over time, the complexity in time for each example is calculated as the cost of the example moving down the tree revising internal nodes (i.e., $O(n) \cdot O(log_2(t))$, where t is the number of nodes in the tree and n is the number of examples present in the leaves below the current node), storing it in the leaf (i.e., $O(1)$) and updating statistics of visited nodes (i.e., $O(n) \cdot O(log_2(t))$). In this case, we have $O((2n \cdot log_2(t)) + 1) \equiv O(n \cdot log_2(t))$.

If the new example provokes an adjustment of a subtree, the cost is higher. As before, the example moves down the tree revising internal nodes until it meets the leaf or the badly adjusted node. In any case, the example is placed on its corresponding leaf (i.e., $O(n \cdot log_2(t))$). Subsequently, it orders each leaf below that node to perform selective forgetting (i.e., $O(f \cdot t')$, where f is the number of examples to forget in each leaf and t' represents the number of nodes below the current node) and the tree is pruned by collecting the remaining examples of its hanging leaves (i.e., $O(n)$). Then, it tries to reconstruct the new node using the reduction of deviation criteria (i.e., $O(n \cdot a \cdot v)$, where n is the number of examples, a is the number of attributes of the problem, and v is the number of values in each attribute). Finally, it updates the statistics of the visited nodes (i.e., $O(n \cdot log_2(t))$). In total, we have $O(2n \cdot log_2(t) + f \cdot t + n + (n \cdot a \cdot v)) \equiv O(n \cdot a \cdot v)$. As can be seen, the highest complexity is from the calculation of the selection criteria when SAIRT needs to expand a leaf.

As in other tree-based methods, when an example is used to query a prediction to the model, the example is directed to the appropriate branch, depending on its values for each splitting attribute in the nodes it traverses (i.e., $O(log_2(t))$). When a leaf is reached, the median value of the dependent values in the set of examples stored is retrieved (i.e., $O(1)$).

As a whole, the complexity of our method is equivalent to that of other methods based on regression tree induction; however, SAIRT can also deal with changes in function and other dynamics (noise, virtual drift, etc.).

4 Experimentation

This section presents the results of the proposed algorithm when facing different kinds of problems from data streams where the underlying knowledge changes to classic (stationary) data sets.

To compare results, we have chosen several well-known algorithms based on different techniques on machine learning and data mining: *IBk* is a nearest-neighbor algorithm used for both classification and prediction tasks, *MLP* is a multilayer perceptron-based algorithm, *M5′* [29] is a regression tree-based learning algorithm that extends M5 and is used when facing problems without changes in the function to induce, and finally, *LinearRegression* tries to create a linear model to approximate the underlying function. We used a suite for data mining called Weka [29] to obtain results with the algorithms described above. In addition, the algorithm *FIRT-DD*, a regression tree-based algorithm able to deal with time-changing data streams, was also included for comparisons in this section. Otherwise stated, default parameters for all methods are used on all of the experiments. To provide better comparisons over time-changing data streams, the IB1 algorithm was used with two kind of windows: global fixed and global adaptive (referenced as AGW). In the latter case, the window management mechanism presented in Section 3.2.1 was sorted out to work with a global window, making use of a parameter to control the excessive accumulation of examples.

For each experiment, we will collect measurements of error rate (specifically, Root-Mean-Squared Error, RMSE), memory consumption, and time devoted to processing one example. To know when SAIRT reacts to changes in functions when there are none (i.e., false alarms), an additional metric, called *estimated rate of function change* (*erfc*), is collected for each experiment. It is calculated with the arrival of each example, and defined by

$$erfc_{tree} = \frac{\sum_{l \in L_{deg}} \frac{|E_l|}{|E_{tree}|} wf_l}{|L_{deg}|},$$

where L_{deg} is the set of degraded leaves in the tree, E_{node} is the set of examples stored below a node, and wf_l is the window fraction forgotten in leaf l. When this value is greater than a specified threshold (e.g., we use 10^{-4}), we suppose that a change in function is detected. This metric is not part of the proposed algorithm and will be used for evaluation purposes.

Throughout this section, low error levels and fast reaction to function drift will show the relevance of using a strategy based on local windows and self-adjustment of parameters when dealing with problems with unknown dynamics. Experiments suggest a high level of adaptability of our algorithm when facing unforeseen changes.

This section has been divided into two parts: Section 4.1 reports results on synthetic and real data streams involving changes in function and Section 4.2 shows how SAIRT and other algorithms behave on classic stationary experiments (without function change).

The following experiments have been carried out on a computer with a four multithreading processor of 3 GHz and 8 GBytes of memory, running GNU/Linux.

4.1 Experiments in Problems with Changes in Function

This section presents data streams involving: partial/total and gradual/abrupt changes in function, different noise level in data over time, numerical/symbolic attributes included, and virtual drift.

The rest of this section will be organized as follows: Section 4.1.1 will illustrate how SAIRT works on a simple synthetic data stream, Section 4.1.2 presents the performance of some algorithms when facing a large and complex synthetic data stream involving unknown conditions, and finally, Sections 4.1.3 and 4.1.4 evaluate the performance of the algorithms when dealing with real problems where unknown dynamics may occur.

4.1.1 The Incidence of Noise and Virtual Drift within the Function Change

The domain of this task is an n-dimensional numerical space. The examples are points in this space and are labeled depending on an aggregated sine function. Following a number of examples, the function drifts and thus the labels of the examples change. The aim is to illustrate how SAIRT deal with different dynamics.

This data set can be formally described as follows: x_i, $i \in \{1, 2, 3, 4, 5\}$, being variables with real domain ($x_i \in [0, 1]$), an example is composed of the values of these variables and is labeled according to the function: $y = \sum_j (2 \cdot \pi sin x_j + \; phase)$, where $j = [1, 2]$. Shifts in function take place, changing the value of $phase$ ($phase \in \{\frac{\pi}{4}, \frac{2\pi}{4}, \frac{3\pi}{4}, \frac{4\pi}{4}\}$). Examples are affected by noise at a level of 10 percent (i.e., each example has a probability of 10 percent of being labeled randomly within the domain of y).

Each training example occurs in a step of time. The data set contains 80,000 time steps and three equally spaced changes in the underlying function. An independent set of 10,000 test examples is used to evaluate the models every 2,000 time steps. Examples in the test set are noise-free and are labeled according to the function being evaluated. For each experiment based on this data set, 30 runs are randomly generated, and the statistics reported are the average RMSE, number of leaves, and examples stored in the model at each test point (see Fig. 4, the horizontal axis represents time, discontinued vertical lines show where the function changes).

A particular shape is found in the error rate curves when facing tasks with changes in patterns. These curves show sudden rises in the error rate proportional to the region involved in the change. This happens because models are evaluated with examples from the new function when they have not yet been adapted. Thus, not only the RMSE achieved at the end of each function is important but also its slope after each change as a symptom of its reaction capability.

Regarding results in the described experiment, SAIRT starts it accumulating examples and expanding the tree until the first change in the underlying function. After processing a few examples, it detects the changed regions and forgets selected examples from them. In addition, SAIRT expands the tree to better adapt to the new patterns. This process can be observed after each change in function for this experiment. In this experiment, the algorithm shows fast reaction to changes, as well as low error levels before

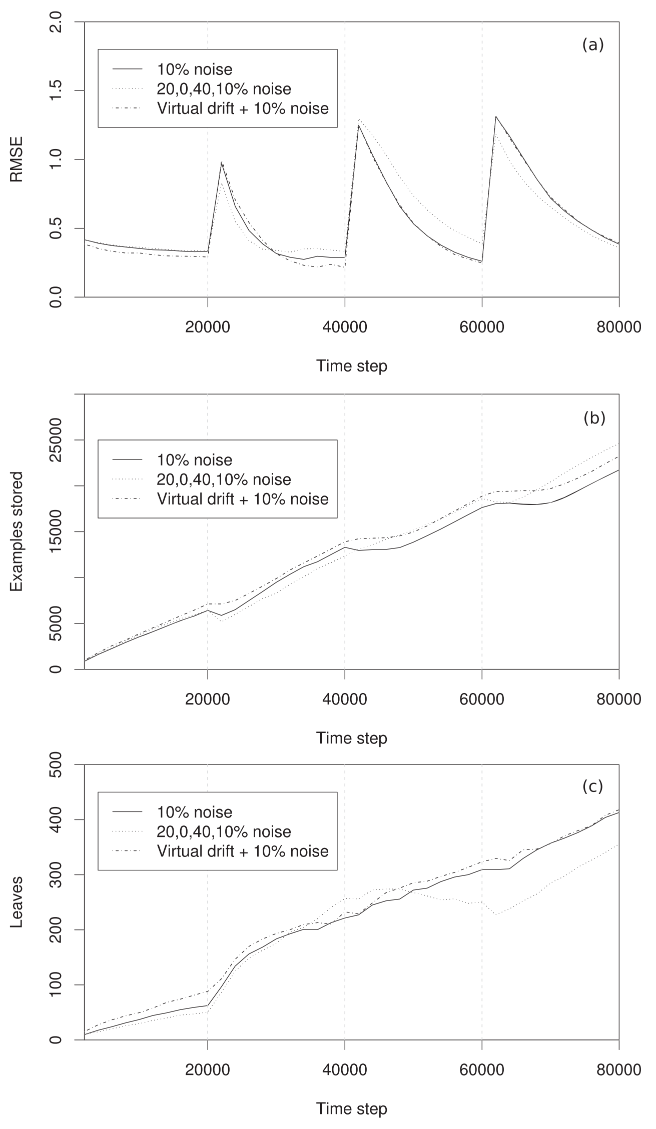

Fig. 4. (a) Root-mean-squared error, (b) examples stored, and (c) number of leaves in model on experiments based on the sine data set.

each change (RMSE are 0.33, 0.29, 0.26, and 0.38 at the end of each function, respectively).

Two different versions of the previous experiment have been designed to evaluate the behavior of the proposed algorithm when the noise level changes over time (percentages at each function are: 20 percent, no noise, 40 and 10 percent, respectively), and when virtual drift is present. The latter is done by sampling examples using a Normal distribution, and changing its mean parameter for each function configuration ($\mu \in \{0.25, 0.5, 0.75\}$) while its variance is fixed to 0.5. Noise level is 10 percent.

In the case of changing levels of noise (dotted line in Fig. 4), the algorithm is expected to perform worse than the previous study when noise level is above 10 percent. However, the RMSE curve shows similar results in both cases, except for very high levels of noise (40 percent).

Results on the experiment with virtual drift (i.e., dashed-dotted line in Fig. 4) do not show big differences with respect to the original experiment, although the sampling rate varies throughout this data stream.

These results suggest that the proposed algorithm is robust when facing data streams that involve changes in function, noise at different levels, and virtual drift.

4.1.2 The Incidence of Level of Change within the Function Change

In this section, we will evaluate the performance of different algorithms over a synthetic data stream which involves unknown dynamics. Throughout this section, the underlying function to approximate will suffer arbitrary changes in different regions and speeds, noise level, and sampling distribution (virtual drift).

For this experiment, we consider a 10-dimensional space ($x_i \in [0, 1], i \in \{1, \dots, 10\}$). Examples are points of this space. The underlying function is made up of three regions that may change over time, gradually or abruptly. Each region is enclosed by a threshold value that also varies over time. Initially, the function goes through a gradual changes phase by shifting the functions present at each region. Then, a phase of abrupt changes is produced by swamping the functions of each region. Formally,

- *Gradual changes phase*. In this phase, examples are labeled using the following piecewise function:

$$y = \begin{cases} \sum_{i=1}^{3} \frac{x_i}{b_1} + sign(p_1)(-1 + 2rc(p_i)), & \text{if } x_i \leq b_1, \\ \sum_{i=1}^{3} 5\sin\left(1 + rc(p_2)\right) \cdot 2\pi \frac{x_i}{b_2}, & \text{if } x_i \leq b_2, \\ \sum_{i=1}^{3} sign(p_3)(-2rc(p_3) + 1) \cdot x_i, & \text{otherwise}, \end{cases} \tag{1}$$

where b_1 and b_2 are the boundary between regions; $sign(a)$ marks the direction of the drift and is equal to -5 if $\frac{fc}{a/2}\%2 = 0$, where fc is the number of changes performed; otherwise $sign$ is equal to $+5$; $rc(a) = \frac{fc\%(a/2)}{a/2}$ gives the ratio of change depending on a; and $p_1 = 20, p_2 = 16, p_3 = 10$ are the periods for each region.

Boundaries between regions may also change. Their equations are defined by

$$b_1 = \sqrt[3]{\frac{1}{4}} + |\sin(2\pi \cdot rc(20))| \cdot \left(\sqrt[3]{\frac{1}{3}} - \sqrt[3]{\frac{1}{4}}\right),$$

$$b_2 = \sqrt[3]{\frac{2}{3}} + |\cos(2\pi \cdot rc(20))| \cdot \left(\sqrt[3]{\frac{3}{4}} - \sqrt[3]{\frac{2}{3}}\right).$$

The noise level in each time step can be calculated as $nl = \frac{fc}{24} \cdot 50$. Virtual drifts are produced by sampling examples from the i-dimensional space using a Normal distribution in which its mean change over time while the variance remains fixed to 0.5. This phase is composed of 25 different configurations for (1). The function length (in time steps) for each configuration can be computed by $fl = (7{,}000 \cdot \frac{b_1}{b_2}) + (100 \cdot nl)$.
- *Abrupt changes phase*. This phase starts using the last configuration of (1) from the previous phase, but the

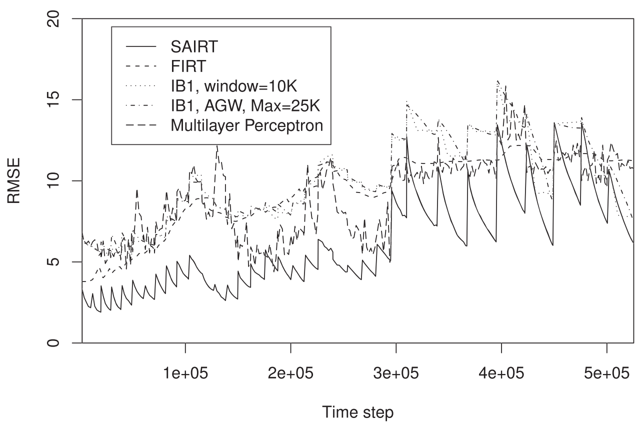

Fig. 5. Root-mean-squared error on a data stream with unknown dynamics.

three functions are exchanged. The function present in region 1 is now in the second region, the function in region 2 goes to region 3, and the function existing in region 3 is moved to region 1. There are eight function changes in this phase, each of them swap the labeled regions from the previous function as explained before, and reduces the noise by 5 percent. Examples are uniformly distributed in the i-dimensional space. The function length is now calculated by: $fl = 30{,}000 - (5{,}000 - (100 \cdot nl))$.

In this experiment, we obtain models using SAIRT, FIRT-DD, MLP,[1] and the IB1 algorithm with a global fixed window of 10,000 examples and an adaptive global window with maximum size of 25,000 examples. Metrics about RMSE and memory and time consumption as been collected throughout this data stream. To do so, an independent set of 10,000 examples is used. These examples are noise-free, have the same sampling distribution of training examples, and are labeled depending on the function present in each moment. Models are evaluated each 2,000 time steps and before each change. This experiment consists of 10 runs of this data set. On each test point a Wilcoxon hypothesis test (with significance level at 0.05) has been used to compare results.

Analysis of gradual changes phase. Fig. 5 summarizes the error rates obtained by the algorithms evaluated. Throughout this phase (from time step 1 to about 300,000) function changes gradually, being easier to induce in some intervals and harder in others (i.e., this phase combines different rates of gradual changes). Looking at Fig. 5, the proposed algorithm starts with the (significantly) lowest error rate among the algorithms evaluated. Through this phase, the error curve of SAIRT follows a saw teeth shape. Sudden rises show partial changes in the function, while recovery in error rate is due to fast model adaptation. Other interesting behavior is that SAIRT proves to be robust when a high level of noise (above 30 percent) and virtual drift are present. Regarding false alarms, SAIRT erroneously detects changes in function at the beginning of the experiment (due to lack of examples) and when noise level is high (above 40 percent).

1. Parameter settings for dealing with data streams are: N (number of epochs) set to 1 and V (number of examples to validate) set to 0.

Analyzing the performance of the other algorithms evaluated, MLP on this data stream produces an erratic error curve. Obviously, this reduces the reliability in its predictions. Both versions of the IB1 algorithm shows stable curves, but their error rates are high and significantly worse than the one obtained by the proposed algorithm. IB1 with AGW is slightly better than the fixed window version. FIRT results show a performance similar to IB1 with global window.

Analysis of abrupt change phase. This phase starts around time step 300,000. As abrupt changes come later than in the previous phase, the algorithms have more time to adapt their structures better. At the beginning of this phase, noise level is very high, but SAIRT distinguishes changes in function and forgets a high level of examples to track with abrupt changes. By doing so, the algorithm reacts more quickly to abrupt function changes than other methods evaluated, and can control the selective forgetting of examples to reach high performance. Regarding the error rate before each function change, SAIRT obtains (significantly) the best results. False alarms are only detected when conditions of massive noise are present (around 50 percent noise level).

Regarding other algorithms, MLP does not perform well as it seems unable to detect abrupt changes in the function, especially when there are high noise levels in the data stream. FIRT-DD follows the performance reached by MLP throughout this phase. Finally, results of both versions of IB1 algorithm show that they are able to track abrupt function changes, but their error rates stay higher than that obtained by SAIRT. In the case of using IB1 with AGW, the results are slightly better than those obtained by the fixed window version.

Overall measurements. Analyzing memory consumption of the evaluated algorithms on this experiment, MLP stores statistics for each neuron in its layers to maintain the model. IB1 with global fixed window and AGW stores as the model to update 10,000 examples and 23,000 examples on average, respectively. FIRT-DD uses a model with about 200 nodes and 10 alternate trees. Finally, SAIRT model has approximately 400 leaves and stores an average of 40,000 examples along this data set.

In this problem, FIRT, MLP, SAIRT, and IB1 with global fixed and AGW processes about 60,000, 4,000, 5, 250, and 115 examples per second, respectively. Although FIRT-DD is the fastest algorithm, its performance is significantly worse than the one obtained by SAIRT throughout this experiment. On the other hand, the main problem of MLP is that it does not induce the function accordingly and thus this algorithm is not adequate to deal with this data stream. For both versions of the IB1 algorithm, the error rate before each function change is poor in general. Finally, SAIRT has the lowest processing speed of the algorithms evaluated, but its results in this data stream (about half a million examples, 10 continuous attributes) are significantly better than the algorithms evaluated when dealing with different and unknown conditions such as different speed and rate of change, virtual drift, and different noise levels.

4.1.3 Case Study: The Electricity Market Data Set

This section evaluates some learning algorithms in a real data stream which involves unknown dynamics. The data, collected from the Australian New South Wales Electricity Market, were first described in [30]. This data stream, called

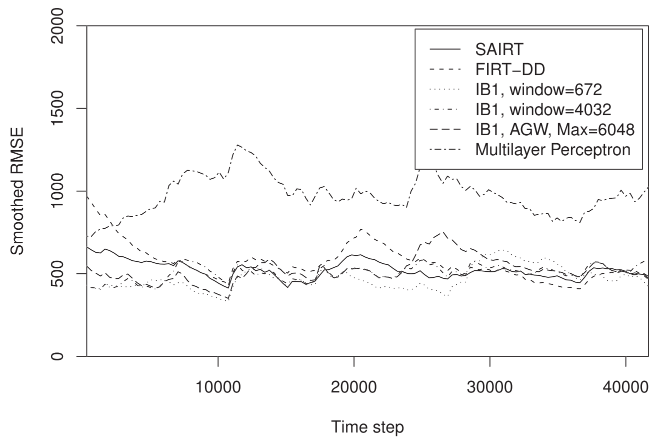

Fig. 6. Results on the electricity market data set.

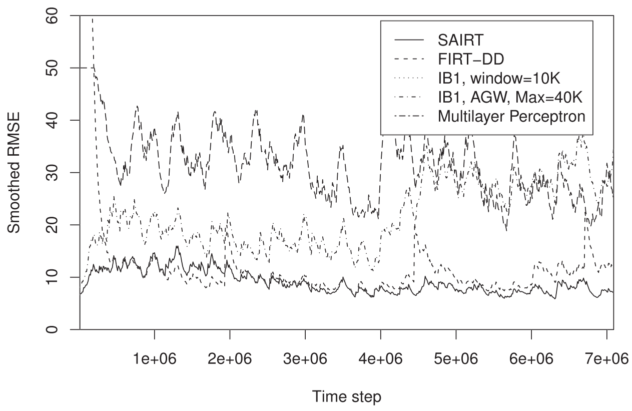

Fig. 7. Results on the departure delay data set.

Elec2, is presumed to have function changes, noise, and virtual drift.

The *Elec2* data stream contains 45,312 instances. Each example of the data set refers to a period of 30 minutes and they have five fields: the day of the week (symbolic value), the time stamp, the NSW electricity demand, the Vic electricity demand, the scheduled electricity transfer between states, and the change of the price related to a moving average in the last 24 hours (symbolic value). The value to be predicted is the NSW electricity demand. This data set is interesting as it is a real-world data set which involves unknown dynamics, and it has been dealt with by other authors [3], [31].

Root-mean-squared error is obtained by testing models induced by learning algorithms each week (i.e., each 336 examples). The test set is composed of examples from the following week; having more examples can result in an evaluation with examples from different dynamics. As the error measurement has high variations (because of small test sets), Fig. 6 shows its exponential smoothed version (with $\alpha = \frac{7}{8}$), which results in a clear graph. As supposed, the learning process is not affected. Algorithms used to compare with are FIRT-DD, MLP (with parameterization for online treatment), and IB1 with AGW and the two better configurations of fixed windows for tracking abrupt (672 examples) and gradual (4,032 examples) changes, chosen after exhaustive tests.

Results on root-mean-squared error are presented in Fig. 6. The three curves corresponding to IB1 start with low error rates, but after processing around 8,000 examples their performances are similar to that obtained by SAIRT. After that, these algorithms behave differently depending on their windows sizes. Fixed windows fit abruptly and gradual changes differently. For its part, results obtained with IB1 with AWG are near the best ones of the fixed window versions. The proposed algorithm tracks function changes, behaving well in the different gradual phases present in the experiment. After the initial 800 time steps, FIRT-DD closely follows the error curves of IB1 and SAIRT algorithms. Regarding MLP, the neural network-based algorithm has the worst results of those evaluated.

In this experiment, SAIRT, FIRT-DD, MLP, and IB1 with fixed global window (672 and 4,032 examples) and AGW are able to process 60, 44,000, 45,000, 50, 35, and 42 examples per second, respectively. Regarding memory usage, SAIRT model has 100 leaves and stores 5,000 examples on average, while the IB1 algorithms with fixed window store as many examples as window size, and its AGW version maintains 4,500 examples on average. The model induced by FIRT-DD contains about 100 nodes. MLP stores only a few statistics about the examples processed in each neuron.

In summary, IB1 seems to be a good option, but the main drawback is to set the right fixed global window size and this does not guarantee that the algorithm will work accordingly in the future. The use of the adaptive heuristics presented in Section 3.2.1 on a global window may solve this issue, but, when partial changes are present, this strategy is not valid. That is why, attending to the relationship between time, quality (in terms of error rate), memory management, model complexity, and algorithm parameterization, SAIRT obtains the best results for the algorithms evaluated in this data stream.

4.1.4 Case Study: The Departure Delay Data Set

This data set was obtained from the Data Expo competition of 2009. It consists on a large data stream in which each example represents a domestic flight in the USA from January 1987 to December 2008. Each registry is described by 29 attributes (e.g., depart date/time, air time, origin, destination, distance, depart delay, etc). As in [7], we extract a data stream that contains flights departing Chicago O'Hare International airport (the world's busiest airport until 2004) from January 1987 to December 2000, which results in about seven million examples after cleaning and sorting the data by scheduled departure time. For this task, the objective is to predict the departure delay of a flight depending on its characterization. Algorithms used to compare results with are FIRT-DD, MLP, IB1 with a fixed global window of 10,000 examples (about one month), and IB1 with an adaptive window size up to 40,000 examples.

For each of 10,000 examples (about one week), the algorithms obtain statistics about RMSE, memory consumption, and the learning time of their models using the following 10,000 examples. Fig. 7 shows the smoothed RMSE of the different methods.

In this experiment, SAIRT obtains lower RMSE than the FIRT-DD algorithm, except in time steps from 1.1 million to about 2 million. The nearest-neighbor algorithms obtain worse results than the previous ones, probably because of

TABLE 1
Results on Stationary Regression Tasks

	SAIRT	IB1	LR	M5	MLP
2d planes	1.012 ± 0.008 (2)	1.29555 ± 0.005 (4)	2.38425 ± 0.00506 (5)	1.00033 ± 0.00402 (1)	1.18559 ± 0.09464 (3)
Abalone	2.44 ± 0.12 (3)	2.893 ± 0.075 (5)	2.22809 ± 0.02498 (1)	2.32168 ± 0.05325 (2)	2.88519 ± 0.41949 (4)
Ailerons	0.0002 ± 0 (3)	0.0002 ± 0 (3)	0.0002 ± 0 (3)	0.0002 ± 0 (3)	0.0002 ± 0 (3)
Bank8FM	0.048 ± 0.0013 (3)	0.12127 ± 0.00161 (5)	0.03877 ± 0.00047 (1)	0.043 ± 0.00049 (2)	0.05014 ± 0.01191 (4)
Bank32nh	0.097 ± 0.0047 (3)	0.14823 ± 0.00288 (5)	0.0837 ± 0.00125 (1)	0.09338 ± 0.00220 (2)	0.10605 ± 0.02677 (4)
California Housing	74489.0 ± 2111 (3)	78404.8 ± 766.202 (4)	69628.7 ± 836.393 (2)	62747.6 ± 1111.23 (1)	85199.2 ± 19631.4 (5)
Computer Activity	3.73 ± 0.27 (2)	3.65235 ± 0.15107 (1)	9.67632 ± 0.14566 (5)	4.51202 ± 0.33514 (3)	5.71465 ± 1.25805 (4)
Computer Activity-s	3.99 ± 0.28 (2)	3.94476 ± 0.06736 (1)	9.98848 ± 0.45619 (5)	4.5909 ± 0.28317 (3)	5.40437 ± 0.78089 (4)
Delta Elevators	0.0017 ± 7.7E-05 (4)	0.002 ± 0 (5)	0.00145 ± 5.3E-05 (1)	0.0015 ± 0 (2)	0.00163 ± 0.00016 (3)
Elevators	0.0044 ± 0.00018 (4)	0.00455 ± 5.3E-05 (5)	0.0029 ± 6.7E-05 (1)	0.00376 ± 7.0E-05 (3)	0.00358 ± 0.00130 (2)
Friedman Example	2.26 ± 0.038 (2)	2.89438 ± 0.01499 (5)	2.63163 ± 0.01252 (4)	1.92968 ± 0.01005 (1)	2.35608 ± 0.71229 (3)
House8L	39134 ± 2194 (2)	44326.5 ± 733.845 (5)	41619.3 ± 768.917 (3)	33837.4 ± 972.846 (1)	44187.6 ± 5652.11 (4)
House16H	45638 ± 2509 (2)	49424.9 ± 913.629(4)	45704.3 ± 484.719 (3)	38824.9 ± 614.774 (1)	51845.4 ± 2925.69 (5)
Kinematics	0.2078 ± 0.0066 (4)	0.17466 ± 0.00116 (1)	0.20223 ± 0.00256 (3)	0.19162 ± 0.00249 (2)	0.21561 ± 0.03200 (5)
MvExample	0.505 ± 0.021 (1)	2.95248 ± 0.02096 (4)	4.49102 ± 0.02530 (5)	0.56354 ± 0.01320 (2)	0.84211 ± 0.12381 (3)
PoleTelecomm	22.42 ± 10.48 (3)	10.0400 ± 0.23698 (1)	30.5218 ± 0.12762 (4)	10.7223 ± 0.27942 (2)	38.4732 ± 7.64229 (5)
Pumadyn8nh	3.39 ± 0.14 (2)	5.1722 ± 0.06912 (5)	4.46175 ± 0.03539 (3)	3.36618 ± 0.03740 (1)	4.71839 ± 0.84028 (4)
Pumadyn32nm	0.026 ± 0.0017 (2)	0.03708 ± 0.00033 (5)	0.02681 ± 0.00022 (3)	0.00932 ± 0.00013 (1)	0.02921 ± 0.00198 (4)
			Summary		
Ranking	2.61	3.77	2.94	1.83	3.83

high dimensionality data, being as bad as MLP in the second half of this experiment, probably because of noise. With respect to executing time, FIRT-DD is faster (14,000 examples per second) than the proposed algorithm (10 examples per second), IB1 with fixed and adaptive window process 20 and 7 examples per second, respectively, and MLP learns 2,000 examples per second. Regarding memory consumption, our method induces a model with 3,000 leaves and stores about 150,000 examples on average; the model induced by FIRT-DD contains 500 nodes; MLP maintains few statistics about the learning data and IB1 with fixed window stores 10,000 examples while its adaptive window version contains about 30,000 examples.

4.2 Experimentation with Stationary Tasks

In this section, SAIRT is evaluated with real problems that do not change over time. Eighteen regression data sets were obtained from different repositories: the UCI Machine Learning Repository [32], the DELVE repository [33], and the StatLib repository [34]. These data sets have different characteristics: number of examples from 4,177 to 40,768, numerical and/or symbolic attributes, noise, virtual drift, etc. The performance of the analyzed algorithms on these data sets is documented in terms of root-mean-squared error, execution time, and examples stored. For SAIRT, we obtained the number of false alarms triggered. This time, the algorithms to which the results are compared are: IB1 (without global window), linear regression (LR), M5′ without linear models in leaves, and MLP.

The experiment was designed based on that proposed in [35]. It is described as follows: For each problem, 1) the examples are randomly distributed, 2) a two-fold cross validation is carried out with each learning algorithm, and 3) the resulting values for each statistic in each fold are noted. These steps are repeated five times (producing a 5×2 cross validation), providing 10 measurements for each algorithm and data set. These results were analyzed using the method recently proposed in [36], which focuses on comparing algorithms over multiple data sets.

The latter methodology assigns a rank to each algorithm on a data set, using its reported result based on the returned statistics (e.g., error rates). The average rank of each algorithm is then calculated. These average values are used to decide, with an F-test, if there is a significant difference between the algorithms. If so, the critical difference (CD) measurement is calculated. Any difference larger than this CD when analyzing a pair of algorithms confirms that they are significantly different.

In Table 1, results on average RMSE and standard deviation for each algorithm and data set are shown, as well as their ranks. At the bottom of Table 1, the average rank for each algorithm is provided.

The F-test, with a significance level of 0.05, reveals that there are differences between algorithms. Thus, the CD value for this experiment is 1.4378. Fig. 8 shows the critical difference diagram.

These results show that, for the data sets evaluated, there are two groups of algorithms that are not significantly different. The M5′ learning algorithm (without linear models in leaves) is the best one when dealing with the presented stationary task, being significantly different from IB1 and MLP. The rest of the algorithms are not significantly different from each other.

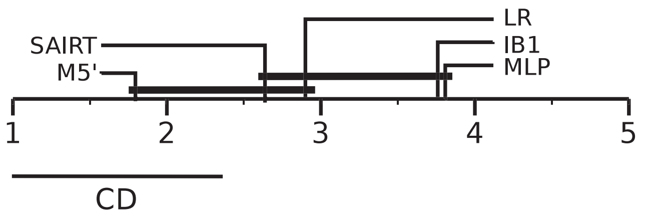

Fig. 8. Critical difference diagram on stationary regression tasks.

On the other hand, regarding execution time, MLP is the fastest algorithm, followed by SAIRT and linear regression learning. Algorithms that need significantly more time are M5′ and IB1.

With respect to the measurement *ercf*, which estimates changes in function detected in our model, SAIRT detects changes at the beginning of each data set when there are obviously none. The reason is the lack of examples, and this is corrected when more learning examples are processed.

In this experiment, the performance of our algorithm is similar to the ones obtained by those algorithms used for comparisons. However, it is important to note that error rates achieved by SAIRT were obtained from trees storing a subset of examples from the whole data set, and it is also several times faster than other algorithms. Moreover, despite the fact that our algorithm creates binary regression trees, we had observed that its models are smaller (they contain fewer leaves) than the M5′ algorithm without linear models in leaves, producing more readable trees and providing fast response to user needs.

5 Future Work

As previously mentioned, SAIRT is an algorithm with partial memory management, that is, it stores a subset of the training examples processed. Therefore, its consumption of resources limits the algorithm to problems where a medium level processing capacity is available. For example, for a similar data stream such as that used in Section 4.1.2, SAIRT stores about 40,000 examples and takes about 5 examples per second, while FIRT-DD processes about 60,000 examples per second storing 200 nodes, and IB1 with global fixed window of 10,000 length takes 250 examples per second. If the problem requires processing with few resources (e.g., embedded systems or mobile phones) and high-rate arrival of examples, this algorithm does not prove adequate. However, computers will be faster and have more and more memory, making partial memory learning more suitable, which does not discount this as a future line of investigation to optimize the algorithm presented with a no-memory management.

To get better models from the point of view of readability, SAIRT should build n-aries trees. The main problem is to obtain an algorithm with low temporal complexity able to split numerical attributes in several intervals. Another interesting feature is to take into account taxonomies to obtain composed attributes and include them in the regression tree.

With the aim of improving the precision of this algorithm, linear models can be incorporated into the leaves of the tree, as suggested in [9]. Drawbacks of this alternative are difficulty in reading the induced tree and an increment in time per example processed; making this sacrifice should result in an improvement in error rate.

6 Conclusions

An incremental regression tree learning method has been presented which is able to induce functions that change at different rates and speeds in data streams that include presence of noise and virtual drift in examples for problems with unknown conditions. As far as we know, there is no other algorithm able to construct and adapt regression trees to data streams whose underlying function changes over time as well as having other dynamics present (changes in noise, virtual drift, different speeds of changes in function, and partial or complete changes) without an a priori parameterization.

SAIRT uses an original strategy to deal with this kind of regression task consisting of using local adaptive windows, which forgets examples as a result of the leaf reducing its window size when the local performance decreases. The windows are located at the leaves of the tree, taking advantage of the "divide and conquer" strategy that naturally exists in the induction of regression trees.

Experimentation shows that SAIRT is more robust to noise than current methods, as can be seen in the evaluation of complex data streams presented in Section 4.1, even if the noise level changes. The proposed algorithm provides fast reaction to function changes as well as a smaller error rate than other methods studied in tasks with gradual and abrupt function changes and noise. This algorithm has also presented a low sensibility to change in the distribution of examples of data (virtual drift), as can be observed in Sections 4.1.1 and 4.1.2.

Regarding the results obtained when dealing with stationary data sets (see Section 4.2), SAIRT proves to be as valid as other well-known algorithms in the machine learning community like M5′, IBk, or MLP.

By adapting its internal parameters, SAIRT is suited to facing problems with unknown conditions. Current learning methods from data streams require the careful selection of the values of its user-defined parameters when faced with certain dynamics of the data stream. If these dynamics change over time (e.g., changes in distribution or noise level), the performance of the algorithms which cannot adapt their parameters may be very poor. For example, a conscientious study of the task to deal with should be carried out to set the size of a fixed window, and even this does not guarantee that the algorithm will work accordingly in the future because of the changing conditions. The heuristics presented in Section 3.2.1 can be sorted out to work with a global window, avoiding user intervention, but in this case, partial changes could not be addressed. On the other hand, we think that algorithms like FIRT-DD, MLP, and IB1 with fixed (and small) global window might be adequate for problems with a known dynamic and a data stream of hundreds or thousands of examples per second.

By using SAIRT, the user does not need to know the details of the algorithm and does not have to carry out any parameter configuration. In addition, the requirements about processing speed and memory consumption of the proposed algorithm seem to fulfill most applications running on the average personal computer. In order to take benefit of all available memory, SAIRT does not increment the number of stored examples when the physical memory limit is reached.

In order to face a wide spectrum of tasks, systems, and users, we think that learning algorithms should be as simple to use as possible and that the models generated should be understandable. This has been an important motivation in the design and evaluation of the SAIRT algorithm. The way in which the knowledge is represented helps to understand both the patterns discovered in each moment and the problem itself. We consider the adaptive capacity of the internal parameters to be fundamental because it will allow them to deal with real-world tasks where there is no guarantee that the dynamics of the problem will not change over time.

Acknowledgments

The authors wish to thank Elena Ikonomovska and João Gama for providing them with the implementation of the FIRT-DD algorithm. The authors appreciate the reviewer's suggestions, which helped them to improve the quality of this paper. This work has been partially supported by the project P07-TIC-02861 of the Junta de Andalucía—Consejería de Innovación, Ciencia y Empresa, Spain.

References

[1] D.H. Widyantoro, T.R. Ioerger, and J. Yen, "An Adaptive Algorithm for Learning Changes in User Interests," *Proc. Eighth Int'l Conf. Information and Knowledge Management,* pp. 405-412, 1999.

[2] W. Fan, "Systematic Data Selection to Mine Concept-Drifting Data Streams," *Proc. ACM SIGKDD,* pp. 128-137, 2004.

[3] M. Núñez, R. Fidalgo, and R. Morales, "Learning in Environments with Unknown Dynamics: Towards More Robust Concept Learners," *J. Machine Learning Research,* vol. 8, pp. 2595-2628, 2007.

[4] P. Domingos and G. Hulten, "Mining High-Speed Data Streams," *Proc. Sixth ACM SIGKDD,* pp. 71-80, 2000.

[5] M.A. Maloof and R.S. Michalski, "Selecting Examples for Partial Memory Learning," *Machine Learning,* vol. 41, no. 1, pp. 27-52, 2000.

[6] E. Ikonomovska and J. Gama, "Learning Model Trees from Data Streams," *Proc. Int'l Conf. Discovery Science,* J.F. Boulicaut, M.R. Berthold, and T. Horváth, eds., pp. 52-63, 2008.

[7] E. Ikonomovska, J. Gama, R. SebastiÃo, and D. Gjorgjevik, "Regression Trees from Data Streams with Drift Detection," *Proc. Int'l Conf. Discovery Science,* pp. 121-135, 2009.

[8] D. Potts and C. Sammut, "Incremental Learning of Linear Model Trees," *Machine Learning,* vol. 61, pp. 5-48, 2005.

[9] J.R. Quinlan, "Learning with Continuous Classes," *Proc. Fifth Australian Joint Conf. Artificial Intelligence,* pp. 307-310, 1992.

[10] L. Breiman, J. Friedman, C.J. Stone, and R. Olsen, *Classification and Regression Trees.* Wadsworth Publishing Company, 1984.

[11] D. Aha, D. Kibler, and M. Albert, "Instance-Based Learning Algorithms," *Machine Learning,* vol. 6, pp. 37-66, 1991.

[12] J.Z. Kolter and M.A. Maloof, "Using Additive Expert Ensembles to Cope with Concept Drift," *Proc. 22nd Int'l Conf. Machine Learning,* pp. 449-456, 2005.

[13] F. Rosenthal, P.B. Volk, M. Hahmann, D. Habich, and W. Lehner, "Drift-Aware Ensemble Regression," *Proc. Int'l Conf. Machine Learning and Data Mining,* pp. 221-235, 2009.

[14] G. Widmer and M. Kubat, "Learning in the Presence of Concept Drift and Hidden Contexts," *Machine Learning,* vol. 23, pp. 69-101, 1996.

[15] M. Kubat and G. Widmer, "Adapting to Drift in Continuous Domains (Extended Abstract)," *Proc. Eighth European Conf. Machine Learning,* pp. 307-310, 1995.

[16] G. Hulten, L. Spencer, and P. Domingos, "Mining Time-Changing Data Streams," *Proc. Seventh ACM SIGKDD,* pp. 97-106, 2001.

[17] R. Klinkenberg and T. Joachims, "Detecting Concept Drift with Support Vector Machines," *Proc. 17th Int'l Conf. Machine Learning,* pp. 487-494, 2000.

[18] R. Klinkenberg, "Learning Drifting Concepts: Example Selection versus Example Weighting," *Intelligent Data Analysis,* vol. 8, no. 3, pp. 281-300, 2004.

[19] M. Núñez, R. Fidalgo, and R. Morales, "On-Line Learning of Decision Trees in Problems with Unknown Dynamics," *Proc. Fourth Mexican Int'l Conf. Artificial Intelligence,* pp. 443-453, 2005.

[20] J.B. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations," *Proc. Fifth Berkeley Symp. Math. Statistics and Probability,* vol. 1, pp. 281-297, 1967.

[21] J. Dougherty, R. Kohavi, and M. Sahami, "Supervised and Unsupervised Discretization of Continuous Features," *Proc. 20th Int'l Conf. Machine Learning,* pp. 194-202, 1995.

[22] J.W. Tukey, *Exploratory Data Analysis.* Addison-Wesley, 1977.

[23] J. Nagle, "On Packets Switches with Infinite Storage," *IEEE Trans. Comm.,* vol. 35, no. 4, pp. 435-438, Apr. 1987.

[24] J. Postel, "RFC-793: TCP Specification," ARPANET WGRC, DDN Network Information Center, SRI Int'l, 1981.

[25] V. Paxson and M. Allman, "RFC-2988: Computing TCPs Transmission Timer," Network WGRC, 2000.

[26] F. Esposito, D. Malerba, and G. Semeraro, "A Comparative Analysis of Methods for Pruning Decision Trees," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 19, no. 5, pp. 476-491, May 1997.

[27] J.R. Quinlan, "Induction of Decision Trees," *Machine Learning,* vol. 1, pp. 81-106, 1986.

[28] W. Hoeffding, "Probability Inequalities for Sums of Bounded Random Variables," *J. Am. Statistical Assoc.,* vol. 58, pp. 13-30, 1963.

[29] I.H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques,* second ed. Morgan Kaufmann, 2005.

[30] M. Harries, C. Sammut, and K. Horn, "Extracting Hidden Context," *Machine Learning,* vol. 32, no. 2, pp. 101-126, 1998.

[31] J. Gama, P. Medas, and P. Rodrigues, "Learning in Dynamic Environments: Decision Trees for Data Streams," *Proc. Fourth Int'l Workshop Pattern Recognition in Information Systems,* pp. 149-158, 2004.

[32] A. Asuncion and D.J. Newman, "UCI Machine Learning Repository," http://www.ics.uci.edu/~mlearn/MLRepository.html, 2007.

[33] C.E. Rasmussen, R.M. Neal, and G. Hinton, "DELVE Dataset Repository," Univ. of Toronto, http://www.cs.toronto.edu/delve/data/datasets.html, 2003.

[34] P. Vlachos, "StatLib Repository," Carnegie Mellon Univ., Dept. of Statistics, http://lib.stat.cmu.edu/datasets/, 2007.

[35] T.G. Dietterich, "Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms," *Neural Computation,* vol. 10, no. 7, pp. 1895-1923, 1998.

[36] J. Demšar, "Statistical Comparisons of Classifiers over Multiple Data Sets," *J. Machine Learning Research,* vol. 7, pp. 1-30, 2006.

Raúl Fidalgo-Merino received the MSc degree in computer science from the Universidad de Málaga, Spain, in 2003 and the PhD degree in software engineering and artificial intelligence from the Universidad de Málaga in 2008. His research interests include mining time-changing data streams for classification and regression, event prediction, and knowledge discovery in databases, among others. He has been a research assistant at the Universidad de Málaga since 2004, and he is a member of the (IA)2 research group in the same university, which is oriented to provide or improve solutions in real problems by using artificial intelligence techniques.

Marlon Núñez received the degree in electronic engineering from the Javeriana University, Colombia, the master's and PhD degrees from the Polytechnic University of Madrid, and the diploma of advanced studies in physics and mathematics from the University of Málaga. In the field of machine learning, he has worked on constructing cost-sensitive decision trees, mining time-changing data streams for classification and regression, and event prediction models. He has also worked on causal model reasoning for processing telecommunication network events and causal model discovery for predicting space weather events. He is a full professor in the Department of Lenguajes y Ciencias de la Computación at the Universidad de Málaga, Spain.