Online Nonlinear Modeling via Self-Organizing Trees

Nuri Denizcan Vanli[a,*] and Suleyman Serdar Kozat[**]
[*] Massachusetts Institute of Technology, Laboratory for Information and Decision Systems, Cambrdige, MA 02139, USA
[**] Bilkent University, Department of Electrical and Electronics Engineering, Ankara 06800, Turkey
[a] Corresponding: denizcan@mit.edu

Contents

Abstract

We study online supervised learning and introduce regression and classification algorithms based on self-organizing trees (SOTs), which adaptively partition the feature space into small regions and combine simple local learners defined in these regions. The proposed algorithms sequentially minimize the cumulative loss by learning both the partitioning the feature space and the parameters of the local learners defined in each region. The output of the algorithm at each time instance is constructed by combining the outputs of a doubly exponential number (in the depth of the SOT) of different predictors defined on this tree with reduced computational and storage complexity. The introduced methods are generic such that they can incorporate different tree construction methods than the ones presented in this chapter. We present a comprehensive experimental study under stationary and non-stationary environments using benchmark datasets and illustrate remarkable performance improvements with respect to the state-of-the-art methods in the literature.

Keywords: Self-organizing trees, nonlinear learning, online learning, classification and regression trees, adaptive nonlinear filtering, nonlinear modeling, supervised learning

Chapter points

- We present a nonlinear modeling method for online supervised learning problems.
- Nonlinear modeling is introduced via SOTs, which adaptively partitions the feature space to minimize the loss of the algorithm.
- Experimental validation shows huge empirical performance improvements with respect to the state-of-the-art methods.

1. Introduction

Nonlinear adaptive learning is extensively investigated in the signal processing [1, 2, 3, 4] and machine learning literatures [5, 6, 7], especially for applications where linear modeling is inadequate, hence, does not provide satisfactory results due to the structural constraint on linearity. Although nonlinear approaches can be more powerful than linear methods in modeling, they usually suffer from overfitting, stability and convergence issues [8], which considerably limit their application to signal processing and machine learning problems. These issues are especially exacerbated in adaptive filtering due to the presence of feedback, which is even hard to control for linear models [9]. Furthermore, for applications involving big data, which require to process input vectors with considerably large dimensions, nonlinear models are usually avoided due to unmanageable computational complexity increase [10]. To overcome these difficulties, tree based nonlinear adaptive filters or regressors are introduced as elegant alternatives to linear models since these highly efficient methods retain the breadth of nonlinear models while mitigating the overfitting and convergence issues [11, 12, 13].

In its most basic form, a tree defines a hierarchical or nested partitioning of the feature space [12]. As an example, consider the binary tree in Figure 0.1, which partitions a two dimensional feature space. On this tree, each node is constructed by a bisection of the feature space (where we use hyperplanes for separation), which results in a complete nested and disjoint partitioning of the feature space. After the partitions are defined, the local learners in each region can be chosen as desired. As an example, to solve a regression problem, one can train a linear regressor in each region, which yields an overall piecewise linear regressor. In this sense, tree based modeling is a natural nonlinear extension of linear models via a tractable nested structure.

Although nonlinear modeling using trees is a powerful and efficient method, there exist several algorithmic parameters and design choices that affect their performance in many applications [11]. Tuning these parameters is a difficult task for applications involving nonstationary data exhibiting saturation effects, threshold phenomena or chaotic behavior [14]. In particular, the performance of tree based models heavily depends on a *careful partitioning* of the feature space. Selection of a good partition is

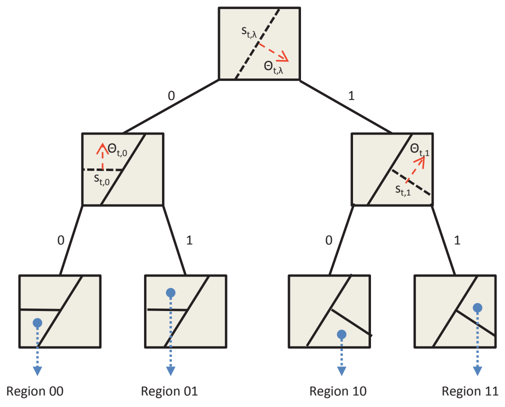

Figure 0.1 ***Feature space partitioning using a binary tree.*** The partitioning of a two dimensional feature space using a complete tree of depth-2 with hyperplanes for separation. The feature space is first bisected by $s_{t,\lambda}$, which is defined by the hyperplane $\boldsymbol{\phi}_{t,\lambda}$, where the region on the direction of $\boldsymbol{\phi}_{t,\lambda}$ vector corresponds to the child with “1” label. We then continue to bisect children regions using $s_{t,0}$ and $s_{t,1}$, defined by $\boldsymbol{\phi}_{t,0}$ and $\boldsymbol{\phi}_{t,1}$, respectively.

essential to balance the bias and variance of the regressor [12]. As an example, even for a uniform binary tree, while increasing the depth of the tree improves the modeling power, such an increase usually results in overfitting [15]. To address this issue, there exists nonlinear modeling algorithms that avoid such a direct commitment to a particular partition but instead construct a weighted average of all possible partitions (or equivalently, piecewise models) defined on a tree [6, 7, 16, 17]. Note that a full binary tree of depth-d defines a doubly exponential number of different partitions of the feature space [18] (For an example, see Figure 0.2). Each of these partitions can be represented by a certain collection of the nodes of the tree, where each node represents a particular region of the feature space. Any of these partitions can be used to construct a nonlinear model, e.g., by training a linear model in each region, we can obtain a piecewise linear model. Instead of selecting one of these partitions and fixing it as the nonlinear model, one can run all partitions in parallel and combine their outputs using a mixture of experts approach. Such methods are shown to mitigate the bias-variance tradeoff in a deterministic framework [6, 7, 16, 19]. However, these methods are naturally constrained to work on a fixed partitioning structure, i.e., the partitions are fixed and cannot be adapted to data.

Although there exist numerous methods to partition the feature space, many of these split criteria are typically chosen a priori and fixed such as the dyadic partitioning [20] and a specific loss (e.g., the gini index [21]) is minimized separately for each node. For instance, multivariate trees are extended to allow the simultaneous use of functional inner and leaf nodes to draw a decision in [13]. Similarly, the node spe-

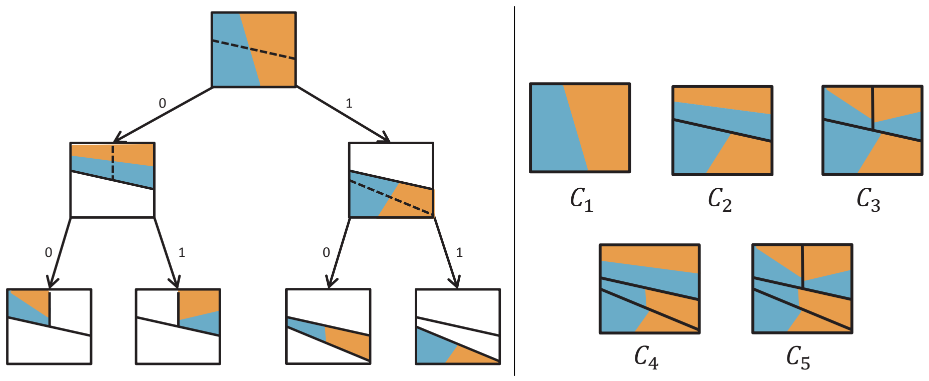

Figure 0.2 ***Example partitioning for a binary classification problem.*** The left figure shows an example partitioning of a 2-dimensional feature space using a depth-2 tree. The active region corresponding to a node is shown colored, where the dashed line represents the separating hyperplane at that node and the two different colored subregions in a node represents the local classifier trained in that region. The right figure shows all different partitions (and consequently classifiers) defined by the tree on the left.

cific individual decisions are combined in [22] via the context tree weighting method [23] and a piecewise linear model for sequential classification is obtained. Since the partitions in these methods are fixed and chosen even before the processing starts, the nonlinear modeling capability of such methods is very limited and significantly deteriorates in case of high dimensionality [24].

To resolve this issue, we introduce SOTs that jointly learns the optimal feature space partitioning to minimize the loss of the algorithm. In particular, we consider a binary tree, where a separator (e.g., a hyperplane) is used to bisect the feature space in a nested manner, and an online linear predictor is assigned to each node. The sequential losses of these node predictors are combined (with their corresponding weights that are sequentially learned) into a global loss that is parameterized via the separator functions and the parameters of the node predictors. We minimize this global loss using online gradient descent, i.e., by updating the complete set of SOT parameters, i.e., the separators, the node predictors, and combination weights, at each time instance. The resulting predictor is a highly dynamical SOT structure that jointly (and in an online manner) learns the region classifiers and the optimal feature space partitioning. In this respect, the proposed method is remarkably robust to drifting source statistics, i.e., non-stationarity. Since our approach is essentially based on a finite combination of linear models, it generalizes well and does not overfit or limitedly overfits (as also shown by our extensive set of experiments).

2. Self-Organizing Trees for Regression Problems

In this section, we consider the sequential nonlinear regression problem, where we observe a desired signal $\{d_t\}_{t\geq 1}$, $d_t \in \mathbb{R}$, and regression vectors $\{\boldsymbol{x}_t\}_{t\geq 1}$, $\boldsymbol{x}_t \in \mathbb{R}^p$, such that we sequentially estimate d_t by

$$\hat{d}_t = f_t(\boldsymbol{x}_t),$$

where $f_t(\cdot)$ is the adaptive nonlinear regression function defined by the SOT. At each time t, the regression error of the algorithm is given by

$$e_t = d_t - \hat{d}_t,$$

and the objective of the algorithm is to minimize the square error loss $\sum_{t=1}^{T} e_t^2$.

2.1. Notation

We first introduce a labeling for the tree nodes following [23]. The root node is labeled with an empty binary string λ and assuming that a node has a label n, where p is a binary string, we label its upper and lower children as $p1$ and $p0$, respectively. Here we emphasize that a string can only take its letters from the binary alphabet $\{0, 1\}$, where 0 refers to the lower child, and 1 refers to the upper child of a node. We also introduce another concept, i.e., the definition of the prefix of a string. We say that a string $n' = q'_1 \ldots q'_{l'}$ is a prefix to string $n = q_1 \ldots q_l$ if $l' \leq l$ and $q'_i = q_i$ for all $i = 1, \ldots, l'$, and the empty string λ is a prefix to all strings. Let $\mathcal{P}(n)$ represent all prefixes to the string n, i.e., $\mathcal{P}(n) \triangleq \{n_0, \ldots, n_l\}$, where $l \triangleq l(n)$ is the length of the string n, n_i is the string with $l(n_i) = i$, and $n_0 = \lambda$ is the empty string, such that the first i letters of the string n forms the string n_i for $i = 0, \ldots, l$.

For a given SOT of depth D, we let $\mathcal{N}_D$ denote all nodes defined on this SOT and $\mathcal{L}_D$ denote all leaf nodes defined on this SOT. We also let β_D denote the number of partitions defined on this SOT. This yields the recursion $\beta_{j+1} = \beta_j^2 + 1$ for all $j \geq 1$, with the base case $\beta_0 = 1$. For a given partition k, we let $\mathcal{M}_k$ denote the set of all nodes in this partition.

For a node $n \in \mathcal{N}_D$ (defined on the SOT of depth D), we define $\mathcal{S}_D(n) \triangleq \{\acute{n} \in \mathcal{N}_D \,|\, \mathcal{P}(\acute{n}) = n\}$ as the set of all nodes of the SOT of depth D, whose set of prefixes include node n.

For a node $n \in \mathcal{N}_D$ (defined on the SOT of depth D) with length $l(n) \geq 1$, the total number of partitions that contain n can be found by the following recursion

$$\gamma_d(l(n)) \triangleq \prod_{j=1}^{l(n)} \beta_{d-j}.$$

For $l(n) = 0$ case (i.e., for $n = \lambda$), one can clearly observe that there exists only one partition containing λ, therefore $\gamma_d(0) = 1$.

For two nodes $n, \acute{n} \in \mathcal{N}_D$ (defined on the SOT of depth D), we let $\rho(n, \acute{n})$ denote the number of partitions that contain both n and $\acute{n}$. Trivially, if $\acute{n} = n$, then $\rho(n, \acute{n}) = \gamma_d(l(n))$. If $n \neq \acute{n}$, then letting $\bar{n}$ denote the longest prefix to both n and $\acute{n}$, i.e., the longest string in $\mathcal{P}(n) \cap \mathcal{P}(\acute{n})$, we obtain

$$\rho(n, \acute{n}) \triangleq \begin{cases} \gamma_d(l(n)), & \text{if } n = \acute{n} \\ \frac{\gamma_d(l(n))\gamma_{d-l(\bar{n})-1}(l(\acute{n})-l(\bar{n})-1)}{\beta_{d-l(\bar{n})-1}}, & \text{if } n \notin \mathcal{P}(\acute{n}) \cup \mathcal{S}_D(\acute{n}). \\ 0, & \text{otherwise} \end{cases} \tag{0.1}$$

Since $l(\bar{n}) + 1 \leq l(n)$ and $l(\bar{n}) + 1 \leq l(\acute{n})$ from the definition of the SOT, we naturally have $\rho(n, \acute{n}) = \rho(\acute{n}, n)$.

2.2. Construction of the Algorithm

For each node n on the SOT, we define a node predictor

$$\hat{d}_{t,n} = \boldsymbol{v}_{t,n}^T \boldsymbol{x}_t, \tag{0.2}$$

whose parameter $\boldsymbol{v}_{t,n}$ is updated using the online gradient descent algorithm. We also define a separator function for each node p on the SOT except the leaf nodes (note that leaf nodes don't have any children) using the sigmoid function

$$s_{t,n} = \frac{1}{1 + \exp(\boldsymbol{\phi}_{t,n}^T \boldsymbol{x}_t)}. \tag{0.3}$$

We then define the prediction of any partition according to the hierarchical structure of the SOT as the weighted sum of the prediction of the nodes in that partition, where the weighting is determined by the separator functions of the nodes between the leaf node and the root node. In particular, the prediction of the kth partition at time t is defined as follows

$$\hat{d}_t^{(k)} = \sum_{n \in \mathcal{M}_k} \left(\hat{d}_{t,n} \prod_{i=0}^{l(n)-1} s_{t,n_i}^{q_i} \right), \tag{0.4}$$

where $n_i \in \mathcal{P}(n)$ is the prefix to string n with length $i - 1$, q_i is the ith letter of the string n, i.e., $n_{i+1} = n_i q_i$, and finally $s_{t,n_i}^{q_i}$ denotes the value of the separator function at node n_i such that

$$s_{t,n_i}^{q_i} \triangleq \begin{cases} s_{t,n_i}, & \text{if } q_i = 0 \\ 1 - s_{t,n_i}, & \text{otherwise} \end{cases} \tag{0.5}$$

with s_{t,n_i} defined as in (0.3). We emphasize that we dropped n-dependency of q_i and n_i to simplify notation. Using these definitions, we can construct the final estimate of our algorithm as

$$\hat{d}_t = \sum_{k \in \beta_D} w_t^{(k)} \hat{d}_t^{(k)}, \tag{0.6}$$

where $w_t^{(k)}$ represents the weight of partition k at time t.

Having found a method to combine the predictions of all partitions to generate the final prediction of the algorithm, we next aim to obtain a low complexity representation since there are $O(1.5^{2^D})$ different partitions defined on the SOT and (0.10) requires a storage and computational complexity of $O(1.5^{2^D})$. To this end, we denote the product terms in (0.4) as follows

$$\hat{\delta}_{t,n} \triangleq \hat{d}_{t,n} \prod_{i=0}^{l(n)-1} s_{t,n_i}^{q_i}, \tag{0.7}$$

where $\hat{\delta}_{t,n}$ can be viewed as the estimate of the node n at time t. Then (0.4) can be rewritten as follows

$$\hat{d}_t^{(k)} = \sum_{p \in \mathcal{M}_k} \hat{\delta}_{t,n}.$$

Since we now have a compact form to represent the tree and the outputs of each partition, we next introduce a method to calculate the combination weights of $O(1.5^{2^D})$ partitions in a simplified manner. To this end, we assign a particular linear weight to each node. We denote the weight of node n at time t as $w_{t,n}$ and then we define the weight of the kth partition as the sum of the weights of its nodes, i.e.,

$$w_t^{(k)} = \sum_{n \in \mathcal{M}_k} w_{t,n},$$

for all $k \in \{1, \ldots, \beta_D\}$. Since we use online gradient descent to update the weight of each partition, the weight of partition k is recursively updated as

$$w_{t+1}^{(k)} = w_t^{(k)} + \mu_t e_t \hat{d}_t^{(k)}.$$

This yields the following recursive update on the node weights

$$w_{t+1,n} = w_{t,n} + \mu_t e_t \hat{\delta}_{t,n}, \tag{0.8}$$

where $\hat{\delta}_{t,n}$ is defined as in (0.7). This result implies that instead of managing $O(1.5^{2^D})$ memory locations, and making $O(1.5^{2^D})$ calculations, only keeping track of the weights of every node is sufficient, and the number of nodes in a depth-D model is $|\mathcal{N}_D| = 2^{D+1} - 1$. Therefore, we can reduce the storage and computational complexity from $O(1.5^{2^D})$ to $O(2^D)$ by performing the update in (0.21) for all $n \in \mathcal{N}_D$.

Using these node predictors and weights, we construct the final estimate of our algorithm as follows

$$\hat{d}_t = \sum_{k=1}^{\beta_d} \left\{ \left(\sum_{n \in \mathcal{M}_k} w_{t,n} \right) \left(\sum_{n \in \mathcal{M}_k} \hat{\delta}_{t,n} \right) \right\}.$$

Here, we observe that for arbitrary two nodes $n, \acute{n} \in \mathcal{N}_d$, the product $w_{t,n}\hat{\delta}_{t,\acute{n}}$ appears $\rho(n, \acute{n})$ times in $\hat{d}_t$ (cf. (0.1)). Hence, the combination weight of the estimate of the node n at time t can be calculated as follows

$$\kappa_{t,n} = \sum_{\acute{n} \in \mathcal{N}_d} \rho(n, \acute{n}) w_{t,\acute{n}}. \tag{0.9}$$

Using the combination weight (0.9), we obtain the final estimate of our algorithm as follows

$$\hat{d}_t = \sum_{n \in \mathcal{N}_D} \kappa_{t,n} \, \hat{\delta}_{t,n}. \tag{0.10}$$

Note that (0.10) is equal to (0.6) with a storage and computational complexity of $O(4^D)$ instead of $O(1.5^{2^D})$.

As we derived all the update rules for the node weights and the parameters of the individual node predictors, what remains is to provide an update scheme for the separator functions. To this end, we use the online gradient descent update

$$\boldsymbol{\phi}_{t+1,n} = \boldsymbol{\phi}_{t,n} - \frac{1}{2}\eta_t \nabla e_t^2(\boldsymbol{\phi}_{t,n}), \tag{0.11}$$

for all nodes $n \in \mathcal{N}_D \setminus \mathcal{L}_D$, where η_t is the learning rate of the algorithm and $\nabla e_t^2(\boldsymbol{\phi}_{t,n})$ is the derivative of $e_t^2(\boldsymbol{\phi}_{t,n})$ with respect to $\boldsymbol{\phi}_{t,n}$. After some algebra, we obtain

$$\begin{aligned}
\boldsymbol{\phi}_{t+1,n} &= \boldsymbol{\phi}_{t,n} + \eta_t e_t \frac{\partial \hat{d}_t}{\partial s_{t,n}} \frac{\partial s_{t,n}}{\partial \boldsymbol{\phi}_{t,n}}, \\
&= \boldsymbol{\phi}_{t,n} + \eta_t e_t \left\{ \sum_{\acute{n} \in \mathcal{N}_D} \kappa_{t,\acute{n}} \frac{\partial \hat{\delta}_{t,\acute{n}}}{\partial s_{t,n}} \right\} \frac{\partial s_{t,n}}{\partial \boldsymbol{\phi}_{t,n}} \\
&= \boldsymbol{\phi}_{t,n} + \eta_t e_t \left\{ \sum_{q=0}^{1} \sum_{\acute{n} \in \mathcal{S}_D(nq)} (-1)^q \kappa_{t,\acute{n}} \frac{\hat{\delta}_{t,\acute{n}}}{s_{t,n}^q} \right\} \frac{\partial s_{t,n}}{\partial \boldsymbol{\phi}_{t,n}},
\end{aligned} \tag{0.12}$$

where we use the logistic regression classifier as our separator function, i.e., $s_{t,n} =$

Algorithm 1 Self-Organizing Tree Regressor (SOTR)

```
 1: for $t = 1$ to $n$ do
 2:     Calculate separator functions $s_{t,p}$, for all $p \in \mathcal{N}_D \setminus \mathcal{L}_D$ using (0.14).
 3:     Calculate node predictors $\hat{d}_{t,p}$, for all $p \in \mathcal{L}_D$ using (0.2).
 4:     Define $\alpha_{t,p} = \prod_{i=1}^{l(p)} s_{t,\nu_i}^{q_i}$ and calculate $\hat{\delta}_{t,p}$, for all $p \in \mathcal{L}_D$ using (0.7).
 5:     Calculate combination weights $\kappa_{t,p}$, for all $p \in \mathcal{L}_D$ using (0.9).
 6:     Construct the final estimate $\hat{d}_t$ using (0.10).
 7:     Observe the error $e_t = d_t - \hat{d}_t$.
 8:     Update the node predictors $\boldsymbol{v}_{t+1,p} = \boldsymbol{v}_{t,p} + \mu_t e_t \alpha_{t,p} \boldsymbol{x}_t$ for all $p \in \mathcal{L}_D$.
 9:     Update the node weights $w_{t+1,p} = w_{t,p} + \mu_t e_t \hat{\delta}_{t,p}$ for all $p \in \mathcal{L}_D$.
10:     Update the separator functions $\boldsymbol{\phi}_{t,p}$ for all $p \in \mathcal{N}_D \setminus \mathcal{L}_D$ using (0.12).
11: end for
```

$\left(1 + \exp(\boldsymbol{x}_t^T \boldsymbol{\phi}_{t,n})\right)^{-1}$. Therefore, we have

$$\begin{aligned}\frac{\partial s_{t,n}}{\partial \boldsymbol{\phi}_{t,n}} &= -\left(1 + \exp(\boldsymbol{x}_t^T \boldsymbol{\phi}_{t,n})\right)^{-2} \exp(\boldsymbol{x}_t^T \boldsymbol{\phi}_{t,n}) \boldsymbol{x}_t \\ &= -s_{t,n}(1 - s_{t,n}) \boldsymbol{x}_t. \qquad (0.13)\end{aligned}$$

Note that other separator functions can also be used in a similar way by simply calculating the gradient with respect to the extended direction vector and plugging in (0.12) and (0.13). We emphasize that $\nabla e_t^2(\boldsymbol{\phi}_{t,n})$ includes the product of $s_{t,n}$ and $1 - s_{t,n}$ terms, hence in order not to slow down the learning rate of our algorithm, we restrict $s^+ \leq s_t \leq 1 - s^+$ for some $0 < s^+ < 0.5$. According to this restriction, we define the separator functions as follows

$$s_t = s^+ + \frac{1 - 2s^+}{1 + e^{\boldsymbol{x}_t^T \boldsymbol{\phi}_t}}. \qquad (0.14)$$

According to the update rule in (0.12), the computational complexity of the introduced algorithm results in $O(m4^d)$. This concludes the construction of the algorithm and a pseudocode is given in Algorithm 1.

2.3. Convergence of the Algorithm

For Algorithm 1, we have the following convergence guarantee, which implies that our predictor (given in Algorithm 1), asymptotically achieves the performance of the best linear combination of the $O(1.5^{2^D})$ different adaptive models that can be represented using a depth-D tree with a computational complexity $O(p4^D)$. We emphasize that while constructing the algorithm, we refrain from any statistical assumptions on the underlying data, and our algorithm works for any sequence of $\{d_t\}_{t\geq 1}$ with an arbitrary length of n. Furthermore, one can use this algorithm to learn the region boundaries and then feed this information to the first algorithm to reduce computational complexity.

Theorem 1. *Let $\{d_t\}_{t\geq 1}$ and $\{\boldsymbol{x}_t\}_{t\geq 1}$ be arbitrary, bounded, and real-valued sequences. The predictor $\hat{d}_t$ given in Algorithm 1 when applied these sequences yields*

$$\sum_{t=1}^{T}(d_t - \hat{d}_t)^2 - \min_{\boldsymbol{w}\in\mathbb{R}^{\beta_d}} \sum_{t=1}^{T}(d_t - \boldsymbol{w}^T\hat{\boldsymbol{d}}_t)^2 \leq O(\log(T)), \tag{0.15}$$

for all T, when $e_t^2(\boldsymbol{w})$ is strongly convex $\forall t$, where $\hat{\boldsymbol{d}}_t = [\hat{d}_t^{(1)}, \ldots, \hat{d}_t^{(\beta_d)}]^T$ and $\hat{d}_t^{(k)}$ represents the estimate of d_t at time t for the adaptive model $k = 1, \ldots, \beta_d$.

Proof of this theorem can be found in Appendix A.1.

3. Self-Organizing Trees for Binary Classification Problems

In this section, we study online binary classification, where we observe feature vectors $\{\boldsymbol{x}_t\}_{t\geq 1}$ and determine their labels $\{y_t\}_{t\geq 1}$ in an online manner. In particular, the aim is to learn a classification function $f_t(\boldsymbol{x}_t)$ with $\boldsymbol{x}_t \in \mathbb{R}^p$ and $y_t \in \{-1, 1\}$ such that when applied in an online manner to any streaming data, the empirical loss of the classifier $f_t(\cdot)$, i.e.,

$$L_T(f_t) \triangleq \sum_{t=1}^{T} \mathbb{1}_{\{f_t(\boldsymbol{x}_t)\neq d_t\}}, \tag{0.16}$$

is asymptotically as small as (after averaging over T) the empirical loss of the best partition classifier defined over the SOT of depth D. To be more precise, we measure the relative performance of f_t with respect to the performance of a partition classifier $f_t^{(k)}$, where $k \in \{1, \ldots, \beta_D\}$, using the following regret

$$R_T(f_t; f_t^{(k)}) \triangleq \frac{L_T(f_t) - L_T(f_t^{(k)})}{T}, \tag{0.17}$$

for any arbitrary length T. Our aim is then to construct an online algorithm with guaranteed upper bounds on this regret for any partition classifier defined over the SOT.

3.1. Construction of the Algorithm

Using the notations described in Section 2.1, the output of a partition classifier $k \in \{1, \ldots, \beta_D\}$ is constructed as follows. Without loss of generality, suppose that the feature $\boldsymbol{x}_t$ has fallen into the region represented by the leaf node $n \in \mathcal{L}_D$. Then, $\boldsymbol{x}_t$ is contained in the nodes $n_0, \ldots, n_D$, where n_d is the i letter prefix of n, i.e., $n_D = n$ and $n_0 = \lambda$. For example, if node n_d is contained in partition k, then one can simply set

$f_t^{(k)}(\boldsymbol{x}_t) = f_{t,n_d}(\boldsymbol{x}_t)$. Instead of making a hard selection, we allow an error margin for the classification output $f_{t,n_d}(\boldsymbol{x}_t)$ in order to be able to update the region boundaries later in the proof. To achieve this, for each node contained in partition k, we define a parameter called *path probability* to measure the contribution of each leaf node to the classification task at time t. This parameter is equal to the multiplication of the separator functions of the nodes from the respective node to the root node, which represents the probability that $\boldsymbol{x}_t$ should be classified using the region classifier of node n_d. This path probability (similar to the node predictor definition in (0.7)) is defined as

$$P_{t,n_d}(\boldsymbol{x}_t) \triangleq \prod_{i=0}^{d-1} s_{t,n_i}^{q_{i+1}}(\boldsymbol{x}_t), \tag{0.18}$$

where $p_{t,n_i}^{q_{i+1}}(\cdot)$ represents the value of the partitioning function corresponding to node n_i towards the q_{i+1} direction as in (0.5). We consider that the classification output of node n_d can be trusted with a probability of $P_{t,n_d}(\boldsymbol{x}_t)$. This and the other probabilities in our development are independently defined for ease of exposition and gaining intuition, i.e., these probabilities are not related to the unknown data statistics in any way and they definitely cannot be regarded as certain assumptions on the data. Indeed, we do not take any assumptions about the data source.

Intuitively, the path probability is low when the feature vector is close to the region boundaries, hence we may consider to classify that feature vector by another node classifier (e.g., the classifier of the sibling node). Using this path probabilities, we aim to update the region boundaries by learning whether an efficient node classifier is used to classify $\boldsymbol{x}_t$, instead of directly assigning $\boldsymbol{x}_t$ to node n_d and lose a significant degree of freedom. To this end, we define the final output of each node classifier according to a Bernoulli random variable with outcomes $\{-f_{t,n_d}(\boldsymbol{x}_t), f_{t,n_d}(\boldsymbol{x}_t)\}$ where the probability of the latter outcome is $P_{t,n_d}(\boldsymbol{x}_t)$. Although the final classification output of node n_d is generated according to this Bernoulli random variable, we continue to call $f_{t,n_d}(\boldsymbol{x}_t)$ the final classification output of node n_d, with an abuse of notation. Then, the classification output of the partition classifier is set to $f_t^{(k)}(\boldsymbol{x}_t) = f_{t,n_d}(\boldsymbol{x}_t)$.

Before constructing the SOT classifier, we first introduce certain definitions. Let the instantaneous empirical loss of the proposed classifier f_t at time t be denoted by $\ell_t(f_t) \triangleq \mathbb{1}_{\{f_t(\boldsymbol{x}_t)\neq y_t\}}$. Then, the expected empirical loss of this classifier over a sequence of length T can be found by

$$L_T(f_t) = E\left[\sum_{t=1}^{T} \ell_t(f_t)\right], \tag{0.19}$$

with the expectation taken with respect to the randomization parameters of the classifier f_t. We also define the effective region of each node n_d at time t as follows

$\mathcal{R}_{t,n_d} \triangleq \left\{ \boldsymbol{x} : P_{t,n_d}(\boldsymbol{x}) \geq (0.5)^d \right\}$. Note that according to the aforementioned structure of partition classifiers, node n_d classifies an instance $\boldsymbol{x}_t$ only if $\boldsymbol{x}_t \in \mathcal{R}_{t,n_d}$. Therefore, the time accumulated empirical loss of any node n during the data stream is given by

$$L_{T,n} \triangleq \sum_{t \leq T : \{\boldsymbol{x}_t\}_{t\geq 1} \in \mathcal{R}_{t,n}} \ell_t(f_{t,n}). \tag{0.20}$$

Similarly, the time accumulated empirical loss of a partition classifier k is $L_T^{(k)} \triangleq \sum_{n \in \mathcal{M}_k} L_{T,n}$.

We then use a mixture-of-experts approach to achieve the performance of the best partition classifier that minimizes the accumulated classification error. To this end, we set the final classification output of our algorithm as $f_t(\boldsymbol{x}_t) = f_t^{(k)}$ with probability $w_t^{(k)}$, where

$$w_t^{(k)} = \frac{1}{Z_{t-1}} 2^{-J(k)} \exp\left(-b\, L_{t-1}^{(k)}\right),$$

$b \geq 0$ is a constant controlling the learning rate of the algorithm, $J(k) \leq 2|\mathcal{L}(k)| - 1$ represents the number of bits required to code the partition k (which satisfies $\sum_{k=1}^{\beta_D} J(k) = 1$), and $Z_t = \sum_{k=1}^{\beta_D} 2^{-J(k)} \exp\left(-b\, L_t^{(k)}\right)$ is the normalization factor.

Although this randomized method can be used as the SOT classifier, in its current form, it requires a computational complexity $O(1.5^{2^D} p)$ since the randomization $w_t^{(k)}$ is performed over the set $\{1, \ldots, \beta_D\}$ and $\beta_D \approx 1.5^{2^D}$. However, the set of all possible classification outputs of these partitions has a cardinality as small as $D + 1$ since $\boldsymbol{x}_t \in \mathcal{R}_{t,n_D}$ for the corresponding leaf node n_D (in which $\boldsymbol{x}_t$ is included) and $f_t^{(k)} = f_{t,n_d}$ for some $d = 0, \ldots, D$, $\forall k \in \{1, \ldots, \beta_D\}$. Hence, evaluating all the partition classifiers in k at the instance $\boldsymbol{x}_t$ to produce $f_t(\boldsymbol{x}_t)$ is unnecessary. In fact, the computational complexity for producing $f_t(\boldsymbol{x}_t)$ can be reduced from $O(1.5^{2^D} p)$ to $O(Dp)$ by performing the exact same randomization over f_{t,n_d}'s using the new set of weights w_{t,n_d}, which can be straightforwardly derived as follows

$$w_{t,n_d} = \sum_{k=1}^{\beta_D} w_t^{(k)} \mathbb{1}_{f_t^{(k)}(\boldsymbol{x}_t) = f_{t,n_d}(\boldsymbol{x}_t)}. \tag{0.21}$$

To efficiently calculate (0.21) with complexity $O(Dp)$, we consider the universal coding scheme and let

$$M_{t,n} \triangleq \begin{cases} \exp\left(-bL_{t,n}\right) & \text{, if } n \text{ has depth } D \\ \frac{1}{2}\left[M_{t,n0} M_{t,n1} + \exp\left(-bL_{t,n}\right)\right] & \text{, otherwise} \end{cases} \tag{0.22}$$

for any node n and observe that we have $M_{t,\lambda} = Z_t$ [23]. Therefore, we can use the recursion (0.22) to obtain the denominator of the randomization probabilities $w_t^{(k)}$. To efficiently calculate the numerator of (0.21), we introduce another intermediate parameter as follows. Letting n'_d denote the sibling of node n_d, we recursively define

$$\kappa_{t,n_d} \triangleq \begin{cases} \frac{1}{2} & , \text{if } d = 0 \\ \frac{1}{2} M_{t-1,n'_d}\, \kappa_{t,n_{d-1}} & , \text{if } 0 < d < D \\ M_{t-1,n'_d}\, \kappa_{t,n_{d-1}} & , \text{if } d = D \end{cases}, \tag{0.23}$$

$\forall d \in \{0, \ldots, D\}$, where $\boldsymbol{x}_t \in \mathcal{R}_{t,n_D}$. Using the intermediate parameters in (0.22) and (0.23), it can be shown that we have

$$w_{t,n_d} = \frac{\kappa_{t,n_d} \exp\left(-b\, L_{t,n_d}\right)}{M_{t,\lambda}}. \tag{0.24}$$

Hence, we can obtain the final output of the algorithm as $f_t(\boldsymbol{x}_t) = f_{t,n_d}(\boldsymbol{x}_t)$ with probability w_{t,n_d}, where $d \in \{0, \ldots, D\}$ (i.e., with a computational complexity $O(D)$).

We then use the final output of the introduced algorithm and update the region boundaries of the tree (i.e., organize the tree) to minimize the final classification error. To this end, we minimize the loss $E\left[\ell_t(f_t)\right] = E[\mathbb{1}_{\{f_t(\boldsymbol{x}_t) \neq y_t\}}] = \frac{1}{4} E\left[(y_t - f_t(\boldsymbol{x}_t))^2\right]$ with respect to the region boundary parameters, i.e., we use the stochastic gradient descent method, as follows

$$\begin{aligned} \boldsymbol{\phi}_{t+1,n_d} &= \boldsymbol{\phi}_{t,n_d} - \eta \nabla E\left[\ell_t(f_t)\right] \\ &= \boldsymbol{\phi}_{t,n_d} - (-1)^{q_{d+1}}\, \eta\, (y_t - f_t(\boldsymbol{x}_t))\, s_{t,n_d}^{q'_{d+1}}(\boldsymbol{x}_t) \left[\sum_{i=d+1}^{D} f_{t,n_i}(\boldsymbol{x}_t)\right] \boldsymbol{x}_t, \end{aligned} \tag{0.25}$$

$\forall d \in \{0, \ldots, D-1\}$, where η denotes the learning rate of the algorithm and q'_{d+1} represents the complementary letter to q_{d+1} from the binary alphabet $\{0, 1\}$. Defining a new intermediate variable

$$\pi_{t,n_d} \triangleq \begin{cases} f_{t,n_d}(\boldsymbol{x}_t) & , \text{if } d = D-1 \\ \pi_{t,n_{d+1}} + f_{t,n_d}(\boldsymbol{x}_t) & , \text{if } d < D-1 \end{cases}, \tag{0.26}$$

one can perform the update in (0.25) with a computational complexity $O(p)$ for each node n_d, where $d \in \{0, \ldots, D-1\}$, resulting in an overall computational complexity of $O(Dp)$ as follows

$$\boldsymbol{\phi}_{t+1,n_d} = \boldsymbol{\phi}_{t,n_d} - (-1)^{m_{d+1}}\, \eta\, (y_t - f_t(\boldsymbol{x}_t))\, \pi_{t,n_d} s_{t,n_d}^{q'_{d+1}}(\boldsymbol{x}_t)\ \boldsymbol{x}_t. \tag{0.27}$$

This concludes the construction of the algorithm and the pseudocode of the SOT classifier can be found in 2.

3.2. Convergence of the Algorithm

In this section, we illustrate that the performance of Algorithm 2 is asymptotically as well as the best partition classifier such that as $T \to \infty$, we have $R_T(f_t; f_t^{(k)}) \to$

Algorithm 2 Self-Organizing Tree Classifier (SOTC)

```
 1: for $t = 1$ to $T$ do
 2:     Propagate $\{\boldsymbol{x}_t\}_{t\geq 1}$ from the root to the leaf and obtain the visited nodes $n_0, \ldots, n_D$.
 3:     Calculate $P_{t,n_d}(\boldsymbol{x}_t)$ for all $d \in 0, ..., D$ using (0.18).
 4:     Calculate $w_{t,n_d}(\boldsymbol{x}_t)$ for all $d \in 0, ..., D$ using (0.24).
 5:     Draw a node among $n_0, \ldots, n_D$ with probabilities $w_{t,n_0}, \ldots, w_{t,n_D}$, respectively; suppose that $n_d$ is drawn.
 6:     Draw a classification output $\{1, -1\}$ with probabilities $P_{t,n_d}(\boldsymbol{x}_t)$ and $1 - P_{t,n_d}(\boldsymbol{x}_t)$, respectively; $f_t(\boldsymbol{x}_t)$ is equated to the selected output.
 7:     Update the region classifiers (perceptron) at the visited nodes [25].
 8:     $\ell_t(f_t) \leftarrow \mathbb{1}_{\{f_t(\boldsymbol{x}_t)\neq y_t\}}$
 9:     Update $L_{t,n_d}$ for all $d \in 0, ..., D$ using (0.20).
10:     Apply the recursion in (0.22) to update $M_{t+1,n_d}$ for all $d \in 0, ..., D$.
11:     Update the separator parameters $\boldsymbol{\phi}$ using (0.27).
12: end for
```

0. Hence, Algorithm 2 asymptotically achieves the performance of the best partition classifier among $O(1.5^{2^D})$ different classifiers that can be represented using the SOT of depth D with a significantly reduced computational complexity of $O(p4^D)$ without any statistical assumptions on data.

Theorem 2. *Let $\{\boldsymbol{x}_t\}_{t\geq 1}$ and $\{y_t\}_{t\geq 1}$ be arbitrary and real-valued sequence of feature vectors and their labels, respectively. Then, Algorithm 2, when applied to these data sequences, sequentially yields*

$$\max_{k\in\{1,\ldots,\beta_D\}} E\left[R_T(f_t; f_t^{(k)})\right] \leq O\left(\sqrt{\frac{2^D}{T}}\right), \tag{0.28}$$

for all T with a computational complexity $O(Dp)$, where p represents the dimensionality of the feature vectors and the expectation is with respect to the randomization parameters.

Proof of this theorem can be found in Appendix A.2.

4. Numerical Results

In this section, we illustrate the performance of SOTs under different scenarios with respect to state-of-the-art methods.

4.1. Numerical Results for Regression Problems

Throughout this section, "SOTR" represents the self-organizing tree regressor defined in Algorithm 1. "CTW" represents the context tree weighting algorithm of [16], "OBR" represents the optimal batch regressor, "VF" represents the truncated Volterra filter [1], "LF" represents the simple linear filter, "B-SAF" and "CR-SAF" represent the Beizer and the Catmul-Rom spline adaptive filter of [2], respectively, "FNF" and "EMFNF" represent the Fourier and even mirror Fourier nonlinear filter of [3], respectively. Finally, "GKR" represents the Gaussian-Kernel regressor and it is constructed using n node regressors, say $\hat{d}_{t,1}, \ldots, \hat{d}_{t,n}$, and a fixed Gaussian mixture weighting (that is selected according to the underlying sequence in hindsight), giving

$$\hat{d}_t = \sum_{i=1}^{n} f\left(\boldsymbol{x}_t; \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i\right) \hat{d}_{t,i},$$

where $\hat{d}_{t,i} = \boldsymbol{v}_{t,i}^T \boldsymbol{x}_t$ and

$$f\left(\boldsymbol{x}_t; \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i\right) \triangleq \frac{1}{2\pi\sqrt{|\boldsymbol{\Sigma}_i|}} e^{-\frac{1}{2}(\boldsymbol{x}_t - \boldsymbol{\mu}_i)^T \boldsymbol{\Sigma}_i^{-1} (\boldsymbol{x}_t - \boldsymbol{\mu}_i)},$$

for all $i = 1, \ldots, n$.

For a fair performance comparison, in the corresponding experiments in Subsection 4.1.2, the desired data and the regressor vectors are normalized between $[-1, 1]$ since the satisfactory performance of the several algorithms require the knowledge on the upper bounds (such as the B-SAF and the CR-SAF) and some require these upper bounds to be between $[-1, 1]$ (such as the FNF and the EMFNF). Moreover, in the corresponding experiments in Subsection 4.1.1, the desired data and the regressor vectors are normalized between $[-1, 1]$ for the VF, the FNF, and the EMFNF algorithms due to the aforementioned reason. The regression errors of these algorithms are then scaled back to their original values for a fair comparison.

Considering the illustrated examples in the respective papers [2, 3, 16], the orders of the FNF and the EMFNF are set to 3 for the experiments in Subsection 4.1.1 and 2 for the experiments in Subsection 4.1.2. The order of the VF is set to 2 for all experiments. Similarly, the depth of the trees for the SOTR and CTW algorithms is set to 2 for all experiments. For these tree based algorithms, the feature space is initially partitioned by the direction vectors $\boldsymbol{\phi}_{t,n} = [\phi_{t,n}^{(1)}, \ldots, \phi_{t,n}^{(p)}]^T$ for all nodes $n \in \mathcal{N}_D \setminus \mathcal{L}_D$, where $\phi_{t,n}^{(i)} = -1$ if $i \equiv l(n) \pmod{D}$, e.g., when $D = p = 2$, we have the four quadrants as the four leaf nodes of the tree. Finally, we use cubic B-SAF and CR-SAF algorithms, whose number of knots are set to 21 for all experiments. We emphasize that both these parameters and the learning rates of these algorithms are selected to give equal rate of performance and convergence.

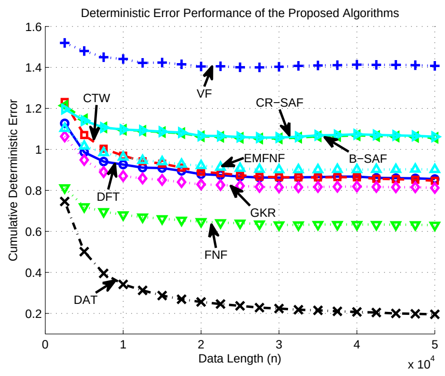

Figure 0.3 Regression error performances for the second order piecewise linear model in (0.29).

4.1.1. Mismatched Partitions

In this subsection, we consider the case where the desired data is generated by a piecewise linear model that mismatches with the initial partitioning of the tree based algorithms. Specifically, the desired signal is generated by the following piecewise linear model

$$d_t = \begin{cases} \boldsymbol{w}^T\boldsymbol{x}_t + \pi_t, & \text{if } \boldsymbol{\phi}_0^T\boldsymbol{x}_t \geq 0.5 \text{ and } \boldsymbol{\phi}_1^T\boldsymbol{x}_t \geq 1 \\ -\boldsymbol{w}^T\boldsymbol{x}_t + \pi_t, & \text{if } \boldsymbol{\phi}_0^T\boldsymbol{x}_t \geq 0.5 \text{ and } \boldsymbol{\phi}_1^T\boldsymbol{x}_t < 1 \\ -\boldsymbol{w}^T\boldsymbol{x}_t + \pi_t, & \text{if } \boldsymbol{\phi}_0^T\boldsymbol{x}_t < 0.5 \text{ and } \boldsymbol{\phi}_2^T\boldsymbol{x}_t \geq -1 \\ \boldsymbol{w}^T\boldsymbol{x}_t + \pi_t, & \text{if } \boldsymbol{\phi}_0^T\boldsymbol{x}_t < 0.5 \text{ and } \boldsymbol{\phi}_2^T\boldsymbol{x}_t < -1 \end{cases}, \tag{0.29}$$

where $\boldsymbol{w} = [1, 1]^T$, $\boldsymbol{\phi}_0 = [4, -1]^T$, $\boldsymbol{\phi}_1 = [1, 1]^T$, $\boldsymbol{\phi}_2 = [1, 2]^T$, $\boldsymbol{x}_t = [x_{1,t}, x_{2,t}]^T$, π_t is a sample function from a zero mean white Gaussian process with variance 0.1, $x_{1,t}$ and $x_{2,t}$ are sample functions of a jointly Gaussian process of mean $[0, 0]^T$ and variance $\boldsymbol{I}_2$. The learning rates are set to 0.005 for SOTR and CTW, 0.1 for FNF, 0.025 for B-SAF and CR-SAF, 0.05 for EMFNF and VF. Moreover, in order to match the underlying partition, the mass points of GKR are set to $\boldsymbol{\mu}_1 = [1.4565, 1.0203]^T$, $\boldsymbol{\mu}_2 = [0.6203, -0.4565]^T$, $\boldsymbol{\mu}_3 = [-0.5013, 0.5903]^T$, and $\boldsymbol{\mu}_4 = [-1.0903, -1.0013]^T$ with the same covariance matrix in the previous example.

Figure 0.3 shows the normalized time accumulated regression error of the proposed algorithms. We emphasize that the SOTR algorithm achieves a better error performance compared to its competitors. Comparing the performances of the SOTR and CTW algorithms, we observe that the CTW algorithm fails to accurately predict the desired data, whereas the SOTR algorithm learns the underlying partitioning of the data, which significantly improves the performance of SOTR. This illustrates the importance of the initial partitioning of the regressor space for tree based algorithms

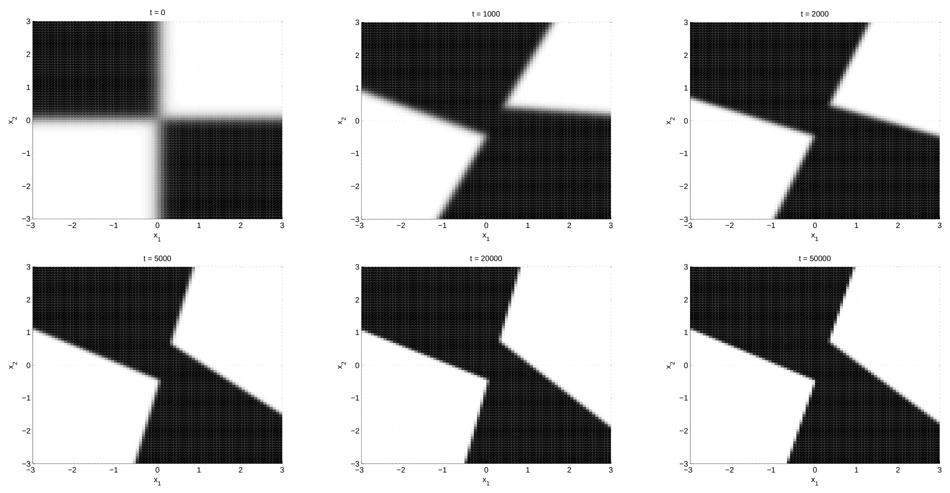

Figure 0.4 Changes in the boundaries of the leaf nodes of the SOT of depth 2 generated by the SOTR algorithm at time instances $t = 0, 1000, 2000, 5000, 20000, 50000$. The separator functions adaptively learn the boundaries of the piecewise linear model in (0.29).

to yield a satisfactory performance.

In particular, the CTW algorithm converges to the best batch regressor having the predetermined leaf nodes (i.e., the best regressor having the four quadrants of two dimensional space as its leaf nodes). However that regressor is sub-optimal since the underlying data is generated using another constellation, hence their time accumulated regression error is always lower bounded by $O(1)$ compared to the global optimal regressor. The SOTR algorithm, on the other hand, adapts its region boundaries and captures the underlying unevenly rotated and shifted regressor space partitioning, perfectly. Figure 0.4 shows how our algorithm updates its separator functions and illustrates the nonlinear modeling power of SOTs.

4.1.2. Chaotic Signals

In this subsection, we illustrate the performance of our algorithm when estimating a chaotic data generated by the Henon map and the Lorenz attractor [26].

First, we consider a zero-mean sequence generated by the Henon map, a chaotic process given by

$$d_t = 1 - \zeta\, d_{t-1}^2 + \eta\, d_{t-2}, \tag{0.30}$$

and known to exhibit chaotic behavior for the values of $\zeta = 1.4$ and $\eta = 0.3$. The desired data at time t is denoted as d_t whereas the extended regressor vector is $\boldsymbol{x}_t = [d_{t-1}, d_{t-2}, 1]^T$, i.e., we consider a prediction framework. The learning rates are set to 0.025 for B-SAF and CR-SAF, whereas it is set to 0.05 for the rest.

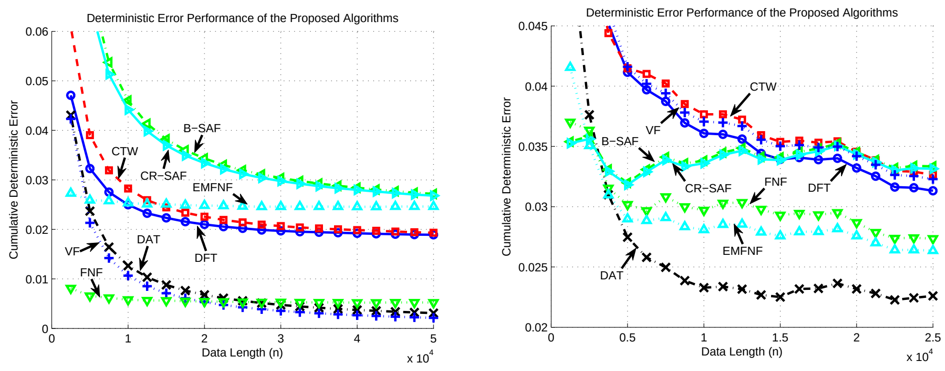

Figure 0.5 Regression error performances of the proposed algorithms for the signal generated by the Henon map in (0.30) (left figure) and for the Lorenz attractor in (0.31) with parameters $dt = 0.01$, $\rho = 28$, $\sigma = 10$, and $\beta = 8/3$ (right figure).

Figure 0.5 (left plot) shows the normalized regression error performance of the proposed algorithms. One can observe that the algorithms whose basis functions do not include the necessary quadratic terms and the algorithms that rely on a fixed regressor space partitioning yield unsatisfactory performance. On the other hand, we emphasize that VF can capture the salient characteristics of this chaotic process since its order is set to 2. Similarly, FNF can also learn the desired data since its basis functions can well approximate the chaotic process. The SOTR algorithm, however, uses a piecewise linear modeling and still achieves the asymptotically same performance as the VF algorithm, while outperforming the FNF algorithm.

Second, we consider the chaotic signal set generated using the Lorenz attractor [26] that is defined by the following three discrete time equations:

$$\begin{aligned} x_t &= x_{t-1} + (\sigma(y - x))dt \\ y_t &= y_{t-1} + (x_{t-1}(\rho - z_{t-1}) - y_{t-1})dt \\ z_t &= z_{t-1} + (x_{t-1}y_{t-1} - \beta z_{t-1})dt, \end{aligned} \tag{0.31}$$

where we set $dt = 0.01$, $\rho = 28$, $\sigma = 10$, and $\beta = 8/3$ to generate the well-known chaotic solution of the Lorenz attractor. In the experiment, x_t is selected as the desired data and the two dimensional region represented by y_t, z_t is set as the regressor space, that is, we try to estimate x_t with respect to y_t and z_t. The learning rates are set to 0.01 for all algorithms.

Figure 0.5 (right plot) illustrates the nonlinear modeling power of the SOTR algorithm even when estimating a highly nonlinear chaotic signal set. As can be observed from Figure 0.5, the SOTR algorithm significantly outperforms its competitors and achieves a superior error performance since it tunes its region boundaries to the opti-

Table 0.1 Average classification errors (in percentage) of algorithms on benchmark datasets.

Data Set	PER	OZAB	OGB	OSB	TNC	SOTC
Heart	24.66	23.96	23.28	23.63	21.75	**20.09**
Breast Cancer	5.77	5.44	5.71	5.23	4.84	**4.65**
Australian	20.82	20.26	19.70	20.01	15.92	**14.86**
Diabetes	32.25	32.43	33.49	31.33	26.89	**25.75**
German	32.45	31.86	32.72	31.86	28.13	**26.74**
BMC	47.09	45.72	46.92	46.37	25.37	**17.03**
Splice	33.42	32.59	32.79	32.81	18.88	**18.56**
Banana	48.91	47.96	48.00	48.84	27.98	**17.60**

mal partitioning of the regressor space, whereas the performances of the other algorithms directly rely on the initial selection of the basis functions and/or tree structures and partitioning.

4.2. Numerical Results for Classification Problems

4.2.1. Stationary Data

In this section, we consider stationary classification problems and compare the SOTC algorithm with the following methods: *Perceptron* - “PER” [25], *Online AdaBoost* - “OZAB” [27], *Online GradientBoost* - “OGB” [28], *Online SmoothBoost* - “OSB” [29], and *Online Tree based Non-adaptive Competitive Classification* - “TNC” [22]. The parameters for all of these compared methods are set as in their original proposals. For the method *Online GradientBoost* - “OGB” [28] which uses K weak learner per M selectors essentially resulting in MK weak learners in total, we use $K = 1$, as in [29], for a fair comparison along with the logit loss that has been shown to consistently outperform other choices in [28]. The TNC algorithm is non-adaptive, i.e., not self organizing, in terms of the space partitioning, which we use in our comparisons to illustrate the gain due the proposed self-organizing structure. We use the perceptron algorithm as the weak learners and node classifiers in all algorithms. We set the learning rate of the SOTC algorithm to $\eta = 0.05$ in all of our stationary as well as non-stationary data experiments. We use $N = 100$ weak learners for the boosting methods, whereas we use a depth-4 tree in SOTC and TNC algorithms, which corresponds to $31 = 2^5 - 1$ local node classifiers. The SOTC algorithm has linear complexity in the depth of the tree, whereas the compared methods have linear complexity in the number of weak learners.

As can be observed in Table 0.1, the SOTC algorithm consistently outperforms the compared methods. In particular, the compared methods essentially fail classifying Banana and BMC datasets, which indicates that these methods are not able to extend to complex nonlinear classification problems. On the contrary, the SOTC algorithm

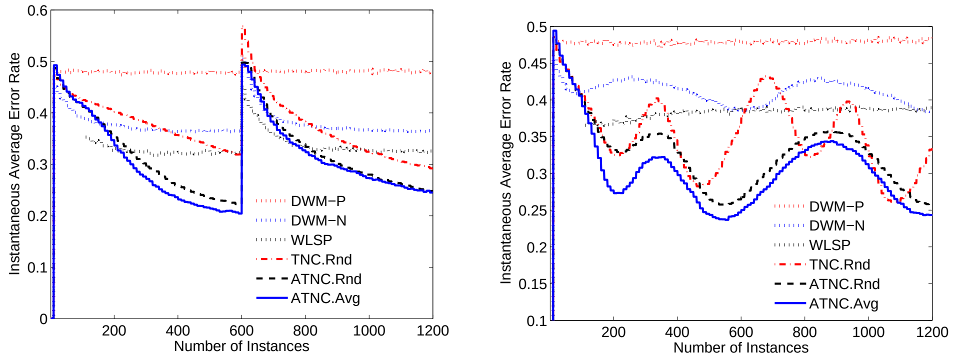

Figure 0.6 Performances of the algorithms in case of the abrupt and continuous concept changes in the BMC dataset. On the left figure, at the 600th instance, there is a 180° clock-wise rotation around the origin that is effectively a label flip. On the right figure, at each instance, there is a $180^{\circ}/1200$ clock-wise rotation around the origin.

successfully models these complex nonlinear relations with piecewise linear curves and provides a superior performance. In general, the SOTC algorithm has a significantly better transient characteristics and the TNC algorithm occasionally performs poorly (such as on BMC and Banana data sets) depending on the mismatch between the initial partitions defined on the tree and the underlying optimal separation of the data. This illustrates the importance of learning the region boundaries in piecewise linear models.

4.2.2. Non-Stationary Data: Concept Change/Drift

In this section, we apply the SOTC algorithm to non-stationary data, where there might be continuous or sudden/abrupt changes in the source statistics, i.e., concept change. Since the SOTC algorithm processes data in a sequential manner, we choose the Dynamically Weighted Majority Algorithm (DWM) [30] with perceptron (DWM-P) or naive bayes (DWM-N) experts for the comparison, since the DWM algorithm is also an online algorithm. Although the batch algorithms do not truly fit into our framework, we still devise an online version of the tree-based local space partitioning algorithm [24] (which also learns the space partitioning and the classifier using the coordinate ascent approach) using a sliding window approach and abbreviate it as the WLSP algorithm. For the DWM method, which allows the addition and removal of experts during the stream, we set the initial number of experts to 1, where the maximum number of experts is bounded by 100. For the WLSP method, we use a window size of 100. The parameters for these compared methods are set as in their original proposals.

We run these methods on the BMC dataset (1200 instances, Figure 0.6), where a sudden/abrupt concept change is obtained by rotating the feature vectors (clock-

Table 0.2 Running times (in seconds) of the compared methods when processing the BMC data set on a daily-use machine (Intel(R) Core(TM) i5-3317U CPU @ 1.70 GHz with 4 GB memory).

PER	OZAB	OGB	OSB	TNC	DWM-P	DWM-N	WLSP	SOTC
0.06	12.90	3.57	3.91	0.43	2.06	6.91	68.40	0.62

wise around the origin) 180° after the 600th instance. This is effectively equivalent to flipping the label of the feature vectors, hence the resulting dataset is denoted as BMC-F. For a continuous concept drift, we rotate each feature vector $180°/1200$ starting from the beginning; and the resulting dataset is denoted as BMC-C. In Figure 0.6, we present the classification errors for the compared methods over 1000 trials. At each 10th instance, we test the algorithms with 1200 instances drawn from the active set of statistics (active concept).

Since BMC data set is non-Gaussian with strongly nonlinear class separations, the DWM method does not perform well on the BMC-F data. For instance, DWM-P operates with an error rate fluctuating around $0.48 - 0.49$ (random guess). This results since the performance of the DWM method is directly dependent on the expert success and observe that both base learners (perceptron or the naive bayes) fail due to the high separation complexity in the BMC-F data. On the other hand, the WLSP method quickly converges to steady state, however it is also asymptotically outperformed by the SOTC algorithm in both experiments. Increasing the window size is clearly expected to boost the performance of WLSP, however at the expense of an increased computational complexity. It is already significantly slower than the SOTC method even when the window size is 100 (for a more detailed comparison see Table 0.2). When the performance of the WLSP method is significantly worse on the BMC-C data set compared to the BMC-F data set, since in the former scenario, WLSP is trained with a batch data of a continuous mixture of concepts in the sliding windows. Under this continuous concept drift, the SOTC method always (not only asymptotically as in the case of the BMC-F data set) performs better than the WLSP nethod. Hence, the sliding window approach is sensitive to the continuous drift. Our discussion about the DWM method on the concept change data (BMC-F) remains valid in the case of the concept drift (BMC-C) as well. In these experiments, the power of self-organizing trees is obvious as the SOTC algorithm almost always outperforms the TNC algorithm. We also observe from Table 0.2 that the SOTC algorithm is computationally very efficient and the cost of region updates (compared with respect to the TNC algorithm) does not increase the computational complexity of the algorithm significantly.

Appendix

A.1. Proof of Theorem 1

For the SOT of depth D, suppose $\hat{d}_t^{(k)}$, $k = 1, \ldots, \beta_d$, are obtained as described in Section 2.2. To achieve the upper bound in (0.15), we use the online gradient descent method and update the combination weights as

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{1}{2}\eta_t \nabla e_t^2(\boldsymbol{w}_t) = \boldsymbol{w}_t + \eta_t e_t \hat{\boldsymbol{d}}_t, \tag{.32}$$

where η_t is the learning rate of the online gradient descent algorithm. We derive an upper bound on the sequential learning regret R_n, which is defined as

$$R_T \triangleq \sum_{t=1}^{T} e_t^2(\boldsymbol{w}_t) - \sum_{t=1}^{T} e_t^2(\boldsymbol{w}_n^*),$$

where $\boldsymbol{w}_T^*$ is the optimal weight vector over T, i.e.,

$$\boldsymbol{w}_T^* \triangleq \underset{\boldsymbol{w} \in \mathbb{R}^{\beta_d}}{\arg\min} \sum_{t=1}^{T} e_t^2(\boldsymbol{w}).$$

Following [31], using Taylor series approximation, for some point $\boldsymbol{z}_t$ on the line segment connecting $\boldsymbol{w}_t$ to $\boldsymbol{w}_T^*$, we have

$$e_t^2(\boldsymbol{w}_T^*) = e_t^2(\boldsymbol{w}_t) + \left(\nabla e_t^2(\boldsymbol{w}_t)\right)^T (\boldsymbol{w}_T^* - \boldsymbol{w}_t) + \frac{1}{2}(\boldsymbol{w}_T^* - \boldsymbol{w}_t)^T \nabla^2 e_t^2(\boldsymbol{z}_t)(\boldsymbol{w}_T^* - \boldsymbol{w}_t). \tag{.33}$$

According to the update rule in (.32), at each iteration the update on weights are performed as $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta_t}{2}\nabla e_t^2(\boldsymbol{w}_t)$. Hence, we have

$$\begin{aligned}\left\|\boldsymbol{w}_{t+1} - \boldsymbol{w}_T^*\right\|^2 &= \left\|\boldsymbol{w}_t - \frac{\eta_t}{2}\nabla e_t^2(\boldsymbol{w}_t) - \boldsymbol{w}_T^*\right\|^2 \\ &= \left\|\boldsymbol{w}_t - \boldsymbol{w}_T^*\right\|^2 - \eta_t \left(\nabla e_t^2(\boldsymbol{w}_t)\right)^T (\boldsymbol{w}_t - \boldsymbol{w}_T^*) + \frac{\eta_t^2}{4}\left\|\nabla e_t^2(\boldsymbol{w}_t)\right\|^2.\end{aligned}$$

This yields

$$\left(\nabla e_t^2(\boldsymbol{w}_t)\right)^T (\boldsymbol{w}_t - \boldsymbol{w}_T^*) = \frac{\left\|\boldsymbol{w}_t - \boldsymbol{w}_T^*\right\|^2 - \left\|\boldsymbol{w}_{t+1} - \boldsymbol{w}_T^*\right\|^2}{\eta_t} + \eta_t \frac{\left\|\nabla e_t^2(\boldsymbol{w}_t)\right\|^2}{4}.$$

Under the mild assumptions that $\left\|\nabla e_t^2(\boldsymbol{w}_t)\right\|^2 \leq A^2$ for some $A > 0$ and $e_t^2(\boldsymbol{w}_T^*)$ is λ-strong convex for some $\lambda > 0$ [31], we achieve the following upper bound

$$e_t^2(\boldsymbol{w}_t) - e_t^2(\boldsymbol{w}_T^*) \leq \frac{\left\|\boldsymbol{w}_t - \boldsymbol{w}_T^*\right\|^2 - \left\|\boldsymbol{w}_{t+1} - \boldsymbol{w}_T^*\right\|^2}{\eta_t} - \frac{\lambda}{2}\left\|\boldsymbol{w}_t - \boldsymbol{w}_T^*\right\|^2 + \eta_t \frac{A^2}{4}. \tag{.34}$$

By selecting $\eta_t = \frac{2}{\lambda t}$ and summing up the regret terms in (.34), we get

$$\begin{aligned} R_n &= \sum_{t=1}^{n} \left\{ e_t^2(\boldsymbol{w}_t) - e_t^2(\boldsymbol{w}_T^*) \right\} \\ &\leq \sum_{t=1}^{n} \left\| \boldsymbol{w}_t - \boldsymbol{w}_T^* \right\|^2 \left(\frac{1}{\eta_t} - \frac{1}{\eta_{t-1}} - \frac{\lambda}{2} \right) + \frac{A^2}{4} \sum_{t=1}^{n} \eta_t \\ &= \frac{A^2}{4} \sum_{t=1}^{n} \frac{2}{\lambda t} \leq \frac{A^2}{2\lambda} \left(1 + \log(n)\right). \end{aligned}$$

A.2. Proof of Theorem 2

Since Z_t is a summation of terms that are all positive, we have $Z_t \geq 2^{-J(k)} \exp\left(-b\, L_t^{(k)}\right)$ and after taking the logarithm of both sides and rearranging the terms, we get

$$-\frac{1}{b} \log Z_T \leq L_T^{(k)} + \frac{J(k) \log 2}{b} \tag{.35}$$

for all $k \in \{1, \ldots, \beta_D\}$ at the (last) iteration at time T. We then make the following observation

$$\begin{aligned} Z_T = \prod_{t=1}^{T} \frac{Z_t}{Z_{t-1}} &= \prod_{t=1}^{T} \sum_{k=1}^{\beta_D} \frac{2^{-J(k)} \exp\left(-b\, L_{t-1}^{(k)}\right)}{Z_{t-1}} \exp\left(-b\, \ell_t(f_t^{(k)})\right) \\ &\leq \exp\left(-b\, L_T(f_t) + \frac{T b^2}{8}\right), \end{aligned} \tag{.36}$$

where the second line follows from the definition of Z_t and the last line follows from the Hoeffding's inequality by treating the $w_t^{(k)} = 2^{-J(k)} \exp\left(-b\, L_{t-1}^{(k)}\right)/Z_{t-1}$ terms as the randomization probabilities. Note that $L_T(f_t)$ represents the expected loss of the final algorithm, cf. (0.19). Combining (.35) and (.36), we obtain

$$\frac{L_T(f_t)}{T} \leq \frac{L_T^{(k)}}{T} + \frac{J(k) \log 2}{T b} + \frac{b}{8},$$

and choosing $b = \sqrt{2^D/T}$, we find the desired upper bound in (0.28) since $J(k) \leq 2^{D+1} - 1$, for all $k \in \{1, \ldots, \beta_D\}$.

ACKNOWLEDGMENTS

The authors would like to thank Huseyin Ozkan for his contributions in this work.

REFERENCE

1. M. Schetzen. *The Volterra and Wiener Theories of Nonlinear Systems*. John Wiley & Sons, NJ, 1980.

2. M. Scarpiniti, D. Comminiello, R. Parisi, and A. Uncini. Nonlinear spline adaptive filtering. *Signal Processing*, 93(4):772 – 783, 2013.
3. A. Carini and G. L. Sicuranza. Fourier nonlinear filters. *Signal Processing*, 94(0):183 – 194, 2014.
4. N. D. Vanli, M. O. Sayin, I. Delibalta, and S. S. Kozat. Sequential nonlinear learning for distributed multiagent systems via extreme learning machines. *IEEE Transactions on Neural Networks and Learning Systems*, 28(3):546–558, March 2017.
5. D. P. Helmbold and R. E. Schapire. Predicting nearly as well as the best pruning of a decision tree. *Mach. Learn.*, 27(1):51–68, 1997.
6. E. Takimoto, A. Maruoka, and V. Vovk. Predicting nearly as well as the best pruning of a decision tree through dyanamic programming scheme. *Theoretical Computer Science*, 261:179–209, 2001.
7. E. Takimoto and M. K. Warmuth. Predicting nearly as well as the best pruning of a planar decision graph. *Theoretical Computer Science*, 288:217–235, 2002.
8. D. P. Bertsekas. *Nonlinear programming*. Athena Scientific, 1999.
9. A. H. Sayed. *Fundamentals of Adaptive Filtering*. John Wiley & Sons, NJ, 2003.
10. S. Dasgupta and Y. Freund. Random projection trees for vector quantization. *IEEE Transactions on Information Theory*, 55(7):3229–3242, 2009.
11. W.-Y. Loh. Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(1):14–23, 2011.
12. L. Brieman, J. Friedman, C. J. Stone, and R. A. Olsen. *Classification and Regression Trees*. Chapman & Hall, 1984.
13. J. Gama. Functional trees. *Machine Learning*, 55(3):219–250, 2004.
14. O. J. J. Michel, A. O. Hero, and A.-E. Badel. Tree-structured nonlinear signal modeling and prediction. *IEEE Trans. Signal Process.*, 47(11):3027–3041, Nov 1999.
15. H. Ozkan, N. D. Vanli, and S. S. Kozat. Online classification via self-organizing space partitioning. *IEEE Transactions on Signal Processing*, 64(15):3895–3908, Aug 2016.
16. S. S. Kozat, A. C. Singer, and G. C. Zeitler. Universal piecewise linear prediction via context trees. *IEEE Trans. Signal Process.*, 55(7):3730–3745, 2007.
17. N. D. Vanli, M. O. Sayin, and S. S. Kozat. Predicting nearly as well as the optimal twice differentiable regressor. *CoRR*, abs/1401.6413, 2014.
18. A. V. Aho and N. J. A. Sloane. Some doubly exponential sequences. *Fibonacci Quarterly*, 11:429–437, 1970.
19. N. D. Vanli, K. Gokcesu, M. O. Sayin, H. Yildiz, and S. S. Kozat. Sequential prediction over hierarchical structures. *IEEE Transactions on Signal Processing*, 64(23):6284–6298, Dec 2016.
20. C. Scott and R. D Nowak. Minimax-optimal classification with dyadic decision trees. *IEEE Trans. Inf. Theory*, 52(4):1335–1353, 2006.
21. C. Strobl, A.-L. Boulesteix, and T. Augustin. Unbiased split selection for classification trees based on the gini index. *Computational Statistics & Data Analysis*, 52(1):483–501, 2007.
22. H. Ozkan, M. A. Donmez, O. S. Pelvan, A. Akman, and S. S. Kozat. Competitive and online piecewise linear classification. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3452–3456, May 2013.
23. F. M. J. Willems, Y. M. Shtarkov, and T. J. Tjalkens. The context-tree weighting method: basic properties. *IEEE Transactions on Information Theory*, 41(3):653–664, 1995.
24. J. Wang and V. Saligrama. Local supervised learning through space partitioning. In *Advances in Neural Information Processing Systems (NIPS)*, pages 91–99. 2012.
25. Y. Freund and R. E Schapire. Large margin classification using the perceptron algorithm. *Mach. Learn.*, 37(3):277–296, 1999.
26. E. N. Lorenz. Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences*, 20(2):130–141, 1963.
27. N. C. Oza and S. Russell. Online bagging and boosting. In *Artificial Intelligence and Statistics*, pages 105–112, 2001.
28. C. Leistner, A. Saffari, P. M. Roth, and H. Bischof. On robustness of on-line boosting-a competitive study. In *IEEE 12th International Conference on Computer Vision Workshops*, pages 1362–1369, 2009.

29. S.-T. Chen, H.-T. Lin, and C.-J. Lu. An online boosting algorithm with theoretical justifications. *International Conference on Machine Learning*, 2012.
30. J. Zico Kolter and Marcus Maloof. Dynamic weighted majority- an ensemble method for drifting concepts. *J. Mach. Learn. Res.*, 8:2755–2790, 2007.
31. E. Hazan, A. Agarwal, and S. Kale. Logarithmic regret algorithms for online convex optimization. *Mach. Learn.*, 69(2-3):169–192, 2007.