Evolving Fuzzy Pattern Trees for Binary Classification on Data Streams

Ammar Shaker, Robin Senge, Eyke Hüllermeier

Department of Mathematics and Computer Science
University of Marburg, Germany
{shaker, senge, eyke}@mathematik.uni-marburg.de

Abstract

Fuzzy pattern trees (FPT) have recently been introduced as a novel model class for machine learning. In this paper, we consider the problem of learning fuzzy pattern trees for binary classification from data streams. Apart from its practical relevance, this problem is also interesting from a methodological point of view. First, the aspect of efficiency plays an important role in the context of data streams, since learning has to be accomplished under hard time (and memory) constraints. Moreover, a learning algorithm should be adaptive in the sense that an up-to-date model is offered at any time, taking new data items into consideration as soon as they arrive and perhaps forgetting old ones that have become obsolete due to a change of the underlying data generating process. To meet these requirements, we develop an evolving version of fuzzy pattern tree learning, in which model adaptation is realized by anticipating possible local changes of the current model, and confirming these changes through statistical hypothesis testing. In experimental studies, we compare our method to a state-of-the-art tree-based classifier for learning from data streams, showing that evolving pattern trees are competitive in terms of performance while typically producing smaller and more compact models.

Keywords: classification, data streams, incremental learning, evolving systems, fuzzy systems, data-driven model design

1. Introduction

Fuzzy pattern tree induction was recently introduced as a novel machine learning method for classification by Huang, Gedeon and Nikravesh [11]. Independently, the same type of model structure was proposed in [23] under the name "fuzzy operator tree". An alternative to the original algorithm for learning pattern trees, as proposed in [11], was developed by Senge and Hüllermeier in [20]. Besides, an FPT variant for regression was introduced in [19].

Roughly speaking, a fuzzy pattern tree is a hierarchical, tree-like structure, whose inner nodes are marked with generalized (fuzzy) logical and arithmetic operators. It implements a recursive function that maps a combination of attribute values, entered in the leaf nodes, to a number in the unit interval, produced as an output by the root of the tree. The model class of fuzzy pattern trees is interesting for several reasons. Apart from some properties that make it appealing from a learning point of view (like a built-in feature selection mechanism and the possibility to guarantee monotonicity in certain attributes), FPTs are arguably attractive from an interpretation point of

view. Generally, each tree can be considered as a kind of (generalized) logical description of a class.[1] In this regard, pattern trees can be considered as a viable alternative to classical fuzzy rule models. Compared to such models, the hierarchical structure of pattern trees further allows for a more compact representation and for trading off accuracy against model simplicity in a seamless manner.

In recent years, the idea of adaptive learning in dynamical environments has received considerable attention, especially under the slogan of "learning from data streams" [8]. Closely related to this, a special branch of data-driven fuzzy systems modeling has emerged under the notion of "evolving fuzzy systems" [2, 15, 1, 16]. Despite small differences regarding the basic assumptions and the technical setting, the emphasis of goals and performance criteria, or the focus on specific types of applications, the key motivation of these and related fields is the idea of a system that learns incrementally, and maybe even in real-time, on a continuous stream of data, and which is able to properly adapt itself to changes of environmental conditions or properties of the data-generating process.

Motivated by these developments, we propose an extended version of fuzzy pattern trees suitable for learning from data streams. More specifically, building on the (batch learning) algorithm for pattern tree induction as proposed in [20], we develop an evolving variant for the problem of binary classification. The rest of the paper is organized as follows. In Section 2, we start with a brief description of the data stream scenario and recall the special requirements it involves for learning. Fuzzy pattern trees are explained in Section 3, in which we also recall the basic algorithm for learning such trees in batch mode. An extension of this algorithm for learning from data streams in then proposed in Section 4. Finally, an empirical evaluation of this method is presented in Section 5, where evolving fuzzy pattern trees are compared with so-called Hoeffding trees [13] on different types of data streams, both in terms of performance and readability.

2. Learning from Data Streams

In recent years, so-called data streams have attracted considerable attention in different fields of computer science, including database systems, data mining, and distributed systems. As the notion suggests, a data stream can roughly be thought of as an ordered sequence of data items, where the input arrives more or less continuously as time progresses [10, 9, 8]. There are various applications in which streams of this type are produced, such as network monitoring, telecommunication systems, customer click streams, stock markets, or any type of multi-sensor systems.

A data stream system may constantly produce huge amounts of data. Regarding aspects of data storage, management, processing, and analysis, the continuous arrival of data items in multiple, rapid, time-varying, and potentially unbounded streams raises new challenges and research problems. Indeed, it is usually not feasible to simply store the arriving data in a traditional database management system in order to perform operations on that data later on. Rather, stream data must generally be processed in an online, incremental manner so as to guarantee that results are up-to-date and that queries can be answered with small time delay.

Domingos and Hulten [5] list a number of properties that an ideal stream mining system should possess, and suggest corresponding design decisions: the system uses only a limited amount of memory; the time to process a single record is short and ideally constant; the data is volatile and a single data record accessed only once; the model produced in an incremental way

[1] Actually, the description is not purely logical, since arithmetic (averaging) operators are also allowed.

is equivalent to the model that would have been obtained through common batch learning (on all data records so far); the learning algorithm should react to concept drift in a proper way and maintain a model that always reflects the current concept.

Apart from processing and querying tools, methods for mining and learning from data streams have attracted a lot of interest [7, 8]. Corresponding algorithms should not only work in an incremental manner, but should also be *adaptive* in the sense of being able to adapt to an evolving environment in which the data (stream) generating process may change over time. Thus, the handling of changing concepts is of utmost importance in mining data streams [3].

A few frameworks and software systems for mining data streams have been released in recent years, including VFML [12] and MOA [4]. VFML is a toolkit for mining high-speed data streams and very large data sets. MOA is a framework for dealing with massive amounts of evolving data streams. It includes data stream generators and several classifiers, and also offers different methods for classifier evaluation. MOA is also able to interact with the popular WEKA machine learning environment [21].

3. Fuzzy Pattern Trees

As already mentioned earlier, a fuzzy pattern tree is a hierarchical, tree-like structure. The inner nodes of an FPT are marked with generalized (fuzzy) operators, either logical and arithmetic, whereas the leaf nodes are associated with fuzzy predicates on input attributes. A pattern tree propagates information from the leaf to the root node: A node takes the values of its descendants as input, combines them using the respective operator, and submits the output to its predecessor. Thus, a pattern tree implements a recursive mapping producing outputs in the unit interval. An exemplary pattern tree is shown in Fig. 1.

3.1. Tree Structure and Model Components

We proceed from the common setting of supervised learning and assume an attribute-value representation of instances, which means that an instance is a vector

$$\boldsymbol{x} \in \mathbb{X} = \mathbb{X}_1 \times \mathbb{X}_2 \times \ldots \times \mathbb{X}_m \ ,$$

where $\mathbb{X}_i$ is the domain of the i-th attribute A_i. Each domain $\mathbb{X}_i$ is discretized by means of a fuzzy partition that consists of n_i fuzzy subsets

$$F_{i,j} : \mathbb{X}_i \to [0,1] \qquad (j = 1, \ldots, n_i) \ ,$$

such that $\sum_{j=1}^{n_j} F_{i,j}(x_i) > 0$ for all $x_i \in \mathbb{X}_i$. The $F_{i,j}$ are often associated with linguistic labels such as "small" or "large", in which case they are also referred to as *fuzzy terms*. In the case of binary classification, each instance is associated with a class label $y \in \mathbb{Y} = \{\ominus, \oplus\}$, where $\oplus$ denotes the positive and $\ominus$ the negative class, respectively. A training example is a tuple $(\boldsymbol{x}, y) \in \mathbb{X} \times \mathbb{Y}$.

Unlike decision trees [17], which assume an input at the root node and output a class prediction at each leaf, pattern trees process information in the reverse direction. The input of a pattern tree is entered at the leaf nodes. More specifically, a leaf node is labeled by an attribute A_i and a fuzzy subset $F_{i,j}$ of the corresponding domain $\mathbb{X}_i$. Given an instance $\boldsymbol{x} = (x_1, \ldots, x_m) \in \mathbb{X}$ as an input, the node produces $F_{i,j}(x_i)$ as an output, that is, the degree of membership of x_i in $F_{i,j}$. This degree of membership is then propagated to the parent node.

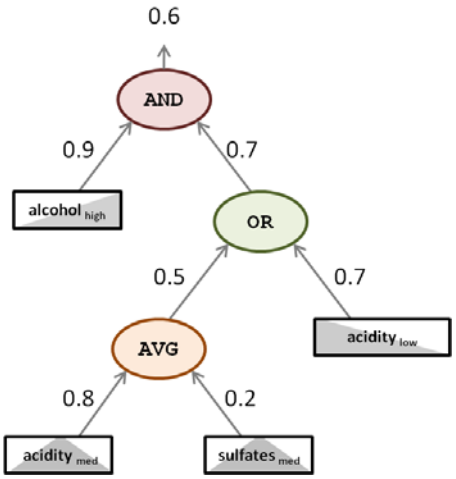


Figure 1: Example of a pattern tree modeling the assessment of a red wine based on chemical properties (see [19]): To quality as a good wine, the level of alcohol must be high and, moreover, either the level of acidity must be low or the average of acidity and sulfates must be medium. The concrete red wine in this example is evaluated as "good" to the degree 0.6; taking $t = 1/2$ as a threshold, it would hence be classified as good if a definite decision (between good and bad) ought to be made.

Internal nodes are labeled by generalized logical or arithmetic operators. More specifically, the set of operators Ψ includes the minimum (MIN), algebraic (ALG), Lukasiewicz (LUK) and Einstein (EIN) t-norms and respective t-conorms [14], as well as the weighted and ordered weighted average [18, 22]. These operators provide a wide spectrum ranging from very strict, conjunctive over averaging to compensatory, disjunctive aggregation:

$$\texttt{EIN} \leq \texttt{LUK} \leq \texttt{ALG} \leq \texttt{MIN} \leq \texttt{WA}(\lambda), \texttt{OWA}(\lambda) \leq \texttt{MAX} \leq \texttt{COALG} \leq \texttt{COLUK} \leq \texttt{COEIN}.$$

The results of the evaluations of internal nodes are propagated to the parents of these nodes in a recursive way. The output eventually produced by a pattern tree is given by the output of its root node; like for all other nodes, it is a number in the unit interval. In the case of binary classification, a discrete prediction can be produced via thresholding: The positive class is predicted if the output exceeds a threshold t (typically $1/2$), otherwise the negative class. For further technical details, we refer to [20].

3.2. Learning Fuzzy Pattern Trees in Batch Mode

The basic algorithm for learning a pattern tree for binary classification in batch mode is presented in pseudo-code in Fig. 2. It implements a beam search and maintains the B best models (trees) so far ($B = 5$ is used as a default value). The algorithm starts by initializing the set of all primitive pattern trees $\mathbf{P}$. A primitive tree is a tree that consists of only one node, labeled by a fuzzy term. Additionally, the first candidate set, $\mathbf{C}^0$, is initialized by the B best primitive pattern trees, i.e., the trees being maximally similar to the target X_0(see Section 3.3).

After initialization, the algorithm iterates over all candidate trees. Starting from line 11, it seeks to improve the currently selected candidate C_i^{t-1} in terms of performance. To this end, new

Top-down Algorithm

{Initialization}
{Every primitive pattern tree is labeled by a Fuzzy subset $F_{i,j}$ associated with attribute A_i}
$\mathbf{P} = \{F_{ij}\}, i = 1, ..., n; j = 1, ..., m$
$\mathbf{C}^0 = argmaxB_{P \in \mathbf{P}}[Sim(P, X_0)]$
$\epsilon = 0.0025$
$t = 0$
{Induction}
{Loop on iterations}
while true **do**
  $t = t + 1$
  $\mathbf{C}^t = \mathbf{C}^{t-1}$
  {Loop on each candidate}
  for all $C_i^{t-1} \in \mathbf{C}^{k-1}$ **do**
    {Loop on each leaf of the chosen candidate}
    for all $l \in leafs(C_i^{t-1})$ **do**
      {Loop on each available operator θ}
      for all $\theta \in \Psi$ **do**
        {Loop on nearly each primitive pattern tree}
        for all $P \in \mathbf{P} \backslash l$ **do**
          $\mathbf{C}^t = \mathbf{C}^t \cup ReplaceLeaf(C_i^{t-1}, l, \theta, P)$
        end for
      end for
    end for
  end for
  $\mathbf{C}^t = argmaxB_{C_i^t \in \mathbf{C}^t}[Perf(C_i^t, X_0)]$
  $perf^*(t) = max_{C_i^t \in \mathbf{C}^t}(Perf(C_i^t, X_0))$
  $perf^*(t-1) = max_{C_i^{t-1} \in \mathbf{C}^{t-1}}(Perf(C_i^{t-1}, X_0))$
  if $perf^*(t) < (1 + \epsilon)perf^*(t-1)$ **then**
    break
  end if
end while
return $argmax_{C_i^t \in \mathbf{C}^t}[Perf(C_i^t, X_0)]$

Figure 2: Top-down algorithm for learning fuzzy pattern trees.

candidates are created by tentatively replacing exactly one leaf node L (labeled by a fuzzy term) of C_i^{t-1} by a new subtree. This new subtree is a three-node pattern tree $N = [L \,|\, \theta \,|\, R]$ that again contains L as one of its leaf nodes, now connected with another primitive tree R by means of an operator θ. The new candidate tree is then evaluated by computing its performance. Having tried all possible replacements of all leaf nodes of the trees in $\mathbf{C}^i$, the B best candidates are selected and passed to the next iteration, unless the termination criterion is fulfilled. More specifically,

our algorithm stops if

$$\text{perf}^*(t) < (1+\epsilon)\text{perf}^*(t-1) \ , \tag{1}$$

i.e., if the relative improvement is smaller than ϵ, where $\epsilon = 0.0025$ by default.

3.3. Performance Evaluation

To evaluate the performance of a pattern tree PT, we compare the output of our pattern tree for each training example $(\boldsymbol{x}^{(i)}, y^{(i)}) \in \mathcal{T}$ to its respective target output. More precisely, a tree will make predictions in the unit interval, which can be considered as membership degrees of a fuzzy subset B of the training data: $B(\boldsymbol{x}^{(i)}) = \text{PT}(\boldsymbol{x}^{(i)})$ for all training instances $\boldsymbol{x}^{(i)}$. This fuzzy subset can then be compared to the true subset of positive (and hence implicitly to the true subset of negative) examples, namely the set A defined by $A(\boldsymbol{x}^{(i)}) = 1$ if $y^{(i)} = \oplus$ and $A(\boldsymbol{x}^{(i)}) = 0$ if i$y^{(i)} = \ominus$. Concretely, the following measure has proved to yield good results [20]:

$$Perf(A,B) = 1 - \sqrt{\frac{1}{|\mathcal{T}|}\sum_{i=1}^{|\mathcal{T}|}(A(\boldsymbol{x}^{(i)}) - B(\boldsymbol{x}^{(i)}))^2} \tag{2}$$

3.4. Fuzzy Partitions

To make pattern tree learning amenable to numeric attributes, these attributes have to be "fuzzified" and discretized beforehand. Fuzzification is needed because fuzzy logical operators at the inner nodes of the tree expect values between 0 and 1 as input, while discretization is needed to limit the number of candidate trees in each iteration of the learning algorithm. Besides, fuzzification may also support the interpretability of the model.

Fuzzy partitions can of course be defined in various ways. In our implementation, we discretize a domain $\mathbb{X}_i$ using three fuzzy sets $F_{i,1}, F_{i,2}, F_{i,3}$ associated, respectively, with the terms "low", "medium" and "high". The first and the third fuzzy set are defined as

$$F_{i,1}(x) = \begin{cases} 1 & x < min \\ 0 & x > max \\ \frac{max-x}{max-min} & otherwise \end{cases}, \quad F_{i,3}(x) = \begin{cases} 1 & x > max \\ 0 & x < min \\ \frac{x-min}{max-min} & otherwise \end{cases},$$

with *min* and *max* being the minimum and the maximum value of the attribute in the training data. Noting that all operators appearing at inner nodes of a pattern tree are monotone increasing in their arguments, it is clear that these fuzzy sets can capture two types of influence of an attribute on the class membership, namely a positive and a negative one: If the value of a numeric attribute increases, the membership of the "high"-term of that attribute also increases (positive influence), whereas the membership of the "low"-term decreases (negative influence).

Apart from monotone dependencies, it is of course possible that a non-extreme attribute value is "preferred" by a class. The fuzzy set $F_{i,2}$ is meant to capture dependencies of this type. It is defined as a triangular fuzzy set with center m:

$$F_{i,2}(x) = \begin{cases} 0 & x \leq min \\ \frac{x-min}{m-min} & min < x \leq m \\ \frac{max-x}{max-m} & m < x < max \\ 0 & x \geq max \end{cases} . \tag{3}$$

The parameter m is determined so as to maximize the absolute (Pearson) correlation between the membership degrees of the attribute values in $F_{i,2}$ and the corresponding class information (encoded by 1 for instances belonging to the class and 0 for instances of other classes) on the training data. In case of a negative correlation, $F_{i,2}$ is replaced by its negation $1 - F_{i,2}$.

Finally, nominal attributes are modeled as degenerated fuzzy sets: For each value v of the attribute, a fuzzy set with the following membership function is introduced:

$$Term_v(x) = \begin{cases} 1 & x = v \\ 0 & otherwise \end{cases} .$$

4. Evolving Fuzzy Pattern Trees

The basic idea of our evolving version of fuzzy pattern tree learning (eFPT) is to maintain an ensemble of pattern trees, consisting of a current (active) model and a set of neighbor models. The current model is used to make predictions, while the neighbor models can be seen as *anticipated adaptations*: they are kept ready to replace the current model in case of a drop in performance, caused, for example, by a drift of the concept to be learned. More generally, the current model is replaced or, say, the anticipated adaptation is realized, whenever its performance appears to be significantly worse than the performance of one of the neighbor models; in this case, the set of neighbors it revised, too.

More specifically, the set of neighbor models is always defined by the set of trees that are "close" to the current model—hence the term "neighbor"—in the sense of being derivable from this model by means of a single "edit operation", namely an expansion or a pruning step; a detailed explanation of how the neighbor trees are generated is given by the algorithm GenerateNeighborTrees shown in Fig. 3. Like in batch learning, an expansion replaces a leaf L of the current tree by a three-node pattern tree $[L \,|\, \theta \,|\, R]$. A pruning step is essentially undoing an expansion. More precisely, each inner node except the root can be replaced by one of its sibling nodes (which means that the subtree rooted by this node is lifted by one level, while the subtree rooted by the other sibling is pruned).

Looking at the neighbor trees as the local neighborhood of the current model in the space of pattern trees, the algorithm is performing a kind of adaptive local search in this space and, therefore, is somewhat comparable to a discrete variant of a swarm-based search procedure (the collective movement of the active model and its "surrounding" neighbor models in the search space is similar, for example, to the flocking of a group of birds).

4.1. Performance Monitoring and Hypothesis Testing

For each time step t, the error rate of the current model PT and, likewise, of all neighbors is calculated on a sliding window consisting of the last n training examples $\left\{(\boldsymbol{x}^{(t-i)}, y^{(t-i)})\right\}_{i=0}^{n-1}$:

$$\tau = \frac{1}{n} \sum_{i=0}^{n-1} \left(y^{(t-i)} - \hat{y}^{(t-i)}\right)^2 \quad , \tag{4}$$

where $\hat{y}^{(i)}$ is the prediction of $y^{(i)}$. The length of the sliding window, n, is a parameter of the method; as a default value, we use $n = 100$, which is large from the point of view of statistical hypothesis testing (see below) and small enough to enable a fast reaction to changes of the data generating process.

Storing the predictions and observed class labels, τ can easily be updated in an incremental way:

$$\tau \leftarrow \tau - \frac{1}{n}\left((y^{(n+1)} - \hat{y}^{(n+1)})^2 - (y^{(1)} - \hat{y}^{(1)})^2\right) \quad , \tag{5}$$

where $y^{(n+1)}$ is a new observation and $y^{(1)}$ the oldest example in the current window.

In order to decide whether or not one of the neighbor trees is superior to the current model, each update of the error rates is followed by a statistical hypothesis test. Let τ_0 and τ_1 denote, respectively, the error rate of the current model and a neighbor tree. We are then testing the null hypothesis $H_0 : \tau_0 \leq \tau_1$ against the alternative hypothesis $H_1 : \tau_0 > \tau_1$. A suitable test statistic for doing so is

$$\frac{\sqrt{n}(\tau_0 - \tau_1)}{\sqrt{2\hat{\tau}(1-\hat{\tau})}} \quad ,$$

where $\hat{\tau} = \frac{\tau_0+\tau_1}{2}$ and n is the sample size (window length). This test statistic approximately follows a normal distribution, and the null hypothesis is rejected if it exceeds a critical threshold $Z_{\alpha'}$; here, $\alpha' = \alpha/c$ is a Bonferroni-corrected significance level (c is the number of neighbor trees). Note that α controls the proneness of the algorithm toward changes of the model: The smaller α, the less often the model will be changed (by default, we use $\alpha = 0.01$).

The above test is conducted for each neighbor tree, and if H_0 is rejected in at least one of these tests, the current model is replaced by the alternative for which the test statistic was the highest. In this case, the fuzzy partitions of the numerical attributes are recomputed, too, applying the approach of Section 3.4 to the data in the current window.

4.2. Summary of the Algorithm

The algorithm for evolving fuzzy pattern tree (eFPT) learning on data streams is summarized in Fig. 4. The main steps of this algorithm are as follows:

1. In the initialization phase, a first pattern tree is learned in batch mode on a small set of training examples. The current model is initialized with this tree.
2. The set of neighbor trees is generated for the current model (see Fig. 3).
3. Upon the arrival of a new example, the sliding window is shifted, the error rates for the current model and all neighbors are updated, and the error rates of the neighbors are compared to the one of the current model.
4. If a neighbor is significantly better than the current model, the latter is replaced by the former; in this case,
   (a) the primitive pattern trees are reinitialized,
   (b) the operators used in the pattern trees are optimized (e.g., by recomputing optimal weight parameters for averaging operators),
   (c) the set of neighbor trees is recomputed (see Fig. 3).
5. Loop at step 3

4.3. Refinements

The computational complexity of our eFPT algorithm critically depends on the size of the model ensemble, i.e., the number of neighbor trees. In fact, while monitoring the performance of a single tree can be done quite efficiently, the overall cost may become high due to the potentially large number of trees that have to be monitored and compared to the current model. Additional

Procedure GenerateNeighborTrees(C)

```
{Initialization}
{Every primitive pattern tree is labeled by a Fuzzy subset F_{i,j} associated with attribute A_i}
P = {F_ij}, i = 1, ..., n; j = 1, ..., m
N = Null
{Creating the neighbor extension trees}
{Loop on each leaf of the current tree}
for all l_chosen ∈ leafs(C) do
   {Loop on each available operator θ}
   for all θ ∈ Ψ do
      {Loop on nearly each primitive pattern tree}
      for all P ∈ P\l_chosen do
         N = N ∪ ReplaceLeaf(C, l_chosen, θ, P)
      end for
   end for
end for
{Creating the neighbor pruning trees}
{Loop on each internal node of the current tree}
for all n_chosen ∈ Internalnodes(C) do
   {Replacing the chosen node by its children nodes}
   N = N ∪ ReplaceNode(C, n_chosen, child1)
   N = N ∪ ReplaceNode(C, n_chosen, child2)
end for
return N
```

Figure 3: Algorithm for generating neighbor trees.

costs are caused by the re-computation of the neighbor models, which becomes necessary after the replacement of the current model.

In the following, we propose two refinements of the above algorithm, both of which are meant to reduce the computational complexity by reducing the number of neighbor models. Since this number mainly depends on two factors, namely the number of leaf nodes of the current model and the number of operators, an obvious idea is to reduce either of these two.

4.3.1. Selecting Leaf Nodes

Recall that a neighbor tree is constructed by either expanding or pruning a leaf node of the current model. Here, our idea is to reduce complexity by allowing these edit operations not for all leafs, but only for a subset of promising candidates. In order to select this subset, we propose a heuristic that estimates the potential influence of a leaf on the overall output of the tree. More specifically, we try to give an approximate answer to the following question: Provided we allow a leaf node L in a pattern tree PT to be expanded, i.e., to replace L by a subtree $N = [L\,|\,\theta\,|\,R]$, what improvement can be expected from this modification?

An optimistic answer to this question can be given by assuming that N will produce optimal outputs, namely $N(\boldsymbol{x}) = 1$ for positive and $N(\boldsymbol{x}) = 0$ for negative examples. Based on this

Evolving Fuzzy Pattern Tree

{Initialization}
C = *BatchPatternTree*
$\mathbf{N}$ = *GenerateNeighborTrees*(C)
{New instance form the stream is present}
while incoming instance t **do**
  {Update the error rate for the current tree}
  $\tau^t_{current} = \tau^{t-1}_{current} - \frac{1}{n}L(y_1, \hat{y}_1) + \frac{1}{n}L_(y_{n+1}, \hat{y}_{n+1})$
  {Loop on each neighbor tree}
  for all $N_k \in \mathbf{N}$ **do**
    {Update the error rate for each neighbor tree}
    $\tau^t_k = \tau^{t-1}_k - \frac{1}{n}L(y_{1,k}, \hat{y}_1) + \frac{1}{n}L_(y_{n+1,k}, \hat{y}_{n+1})$
  end for
  {Testing the null hypothesis that the current error rate is lower than that of all neighbor trees}
  if $\exists N_k \in \mathbf{N}$: Reject $H_0(\tau^t_{current} < \tau^t_k)$ **then**
    {A neighbor tree with a lower error rate is found}
    $C = N_k$
    {Recompute all primitive pattern trees}
    $\mathbf{P} = \{A_{ij}\}, i = 1, ..., n; j = 1, ..., m$
    OptimizeUsedOperator(C)
    $\mathbf{N}$ = *GenerateNeighborTrees*(C)
  end if
end while

Figure 4: Evolving Fuzzy Pattern Trees.

assumption, we define the *potential* of a leaf node L in terms of its *average relative* improvement:

$$\mathrm{POT}(L) = \frac{1}{|\mathcal{T}|} \sum_{(\boldsymbol{x},y)\in\mathcal{T}} \begin{cases} \frac{\mathrm{PT}'(\boldsymbol{x})-\mathrm{PT}(\boldsymbol{x})}{1-L(\boldsymbol{x})} & \text{if } y = \oplus \\ \frac{\mathrm{PT}(\boldsymbol{x})-\mathrm{PT}'(\boldsymbol{x})}{L(\boldsymbol{x})} & \text{if } y = \ominus \end{cases},$$

where PT′ is the pattern tree after expansion of L. Based on this conception of the potential of a leaf, we modify our algorithm by considering only the p leaf nodes with highest potential; p is a parameter that has to be defined by the user (our default value is $p = 3$).

4.3.2. Retaining Operators

Another idea to reduce the number of expansions $N = [L\,|\,\theta\,|\,R]$ is to restrict the set of operators θ. More specifically, we implement a procedure in which some operators are provisionally retained: Instead of trying all logical operators right away, we only try the largest (least extreme) t-norm `MIN` and the smallest t-conorm `MAX` (in addition to the two averaging operators). Only in case `MIN` is selected as an optimal operator, we also try the other (more extreme) t-norms; likewise, if `MAX` is selected, the other t-conorms are tried, and the best one is adopted.

The basic assumption underlying this procedure is that, if any of the t-norms (t-conorms) is the most appropriate operator, the algorithm will select `MIN` (`MAX`) in the first step, since this is the "closest" among the available operators.

5. Empirical Evaluation

In this section, we compare our evolving fuzzy pattern trees (eFPT) with Hoeffding trees [13], a state-of-the-art approach for classification on data streams, in terms of performance, stability, and handling of concept drift. We use eFPT in its default setting (i.e., using default parameters $n = 100$, $\alpha = 0.01$, $p = 3$). Experiments are not only conducted with real data sets, but also with synthetic data. As an important advantage of synthetic data, let us note that it allows for conducting experiments in a *controlled* way and, therefore, to investigate the performance of a method under particular conditions. In particular, synthetic data is useful for simulating a concept drift.

The experiments are performed using the MOA framework, which offers the ConceptDriftStream procedure for simulating concept drift. The idea underlying this procedure is to mix two pure distributions in a probabilistic way, smoothly varying the corresponding probability degrees. In the beginning, examples are taken from the first pure stream with probability 1, and this probability is decreased in favor of the second stream in the course of time. More specifically, the probability is controlled by means of the sigmoid function

$$f(t) = \left(1 + e^{-4(t-t_0)/w}\right)^{-1} \quad .$$

This function has two parameters: t_0 is the mid point of the change process, while w controls the length of this process.

The evaluation of an evolving classifier learning from a data stream is clearly a non-trivial issue. In fact, compared to standard batch learning, simple one-dimensional performance measures such as classification accuracy are not immediately applicable, or at least not able to capture the time-varying behavior of a classifier in a proper way. Besides, additional criteria become relevant, too, such as the handling of concept drift, many of which are rather vague and hard to quantify. In our experiments, we employ a holdout procedure for measuring predictive accuracy, which is offered by the MOA framework. Here, the idea is to interleave the training and the testing phase of a classifier as follows: the classifier is trained incrementally on a block of M instances and then evaluated (but no longer adapted) on the next N instances, then again trained on the next M and tested on the subsequent N instances, and so forth; as parameters, we use $M = 5,000$ and $N = 1,000$ in the first two experiments with synthetic data. For the experiments with real data, these parameters are adapted to the size of the respective data set; see Table. 1 for an overview of the main characteristics of these data sets. The real data sets are standard benchmarks taken from the Statlib archive[2] and the UCI repository [6]. Since they do not have an inherent temporal order, we average the performance curves over 100 randomly shuffled versions of these data sets.

5.1. Synthetic Data

The first experiment uses data taken from a hyperplane generator. Here, the instance space is given by the d-dimensional Euclidean space, and the decision boundary is defined in terms of a

[2] `http://lib.stat.cmu.edu/`

Data Set	Instances	Attributes	Classes	Holdout Evaluation
statlog (shuttle)	58000	9	7	$M = 5000$ and $N = 1000$
red wine	1599	11	[3,8]	$M = 100$ and $N = 25$
white wine	4889	11	[3,9]	$M = 200$ and $N = 50$
adult	32561	14	2	$M = 200$ and $N = 50$

Table 1: Experimental Data Sets Summary.

hyperplane (which is specified by a normal vector $\boldsymbol{w} \in \mathbb{R}^d$ and a value $w_0 \in \mathbb{R}$) in this space. The classification problem is to predict the position of a point $\boldsymbol{x} \in [0, 1]^d$ relative to the hyperplane: $\boldsymbol{x}$ is positive if $\boldsymbol{w}^\top \boldsymbol{x} > w_0$, otherwise it is negative. The ConceptDriftStream procedure mixing streams produced by two different hyperplanes simulates a rotating hyperplane. Using this procedure, we generated 1,200,000 examples connecting two hyperplanes in 4-dimensional space, with $t_0 = 500,000$ and $w = 100,000$.

As can be seen in Fig. 5, eFPT fits the pattern trees quite well to the data. Although there is a visible drop in performance at the beginning of the concept drift, eFPT is able to recover quite quickly, reaching the same performance as before after a short while The Hoeffding tree, on the other hand, needs quite a long time to learn the concept and is more strongly affected by the drift; it recovers only lately, but then reaches almost the same level of accuracy as eFPT.

In the above experiment, the Hoeffding tree was arguably put as a disadvantage, since fitting a hyperplane with a decision tree is a quite difficult problem. In a second experiment, we therefore use a random tree generator to produce examples. This generator constructs a decision tree by making random splits on attribute values and then assigns random class labels to the leaf nodes. Obviously, this generator is favorable for the Hoeffding tree.

Again, the same ConceptDriftStream is used, but this time mixing two random tree generators. As can be seen in Fig. 6, the Hoeffding tree is now able to outperform eFPT in the first phase of the learning process; in fact, it reaches an accuracy of close to 100%, which is not unexpected given that the Hoeffding tree is ideally tailored for this kind of data. Once again, however, the Hoeffding tree is much more affected by the concept drift than the pattern tree learner, showing a more pronounced "valley" in the performance curve.

5.2. Real Data

In this experiment, we used the Shuttle data from the Statlog repository, for which the task is to predict the class of a shuttle. The data set is highly imbalanced, with 80% of the instances belonging to one class and the remaining 20% distributed among six other classes; in order to obtain a binary problem, we grouped these six classes into a single one. The new problem thus consists of predicting whether a shuttle belongs to the majority class or not. Both algorithms were trained at the beginning on 300 instances in batch mode; for the holdout evaluation we used $M = 200$ and $N = 50$. Fig. 7 shows the results averaged over 100 randomly shuffled versions of the data set. As can be seen, eFPT exhibits a rather stable performance from the very beginning, whereas the Hoeffding tree starts to outperform eFPT after seeing about 7,000 instances.

The wine quality data is an ordinal classification problem, in which a wine (characterized by several chemical properties) is put into a discrete category ranging from 10 (best) to 0 (worst). We turned this problem into a binary classification task by grouping the top-5 and bottom-6 classes. Actually, the data set consists of two subsets, one for white wine and one for red wine.

For both data sets, the initial learning is done on 300 instances. For the evaluation on the red wine data, we used $M = 100$ and $N = 25$, because this data set is relatively small (about 1600 examples); for white wine, we used $M = 200$ and $N = 50$. Fig. 8 shows the results of both experiments. As can be seen, eFPT is clearly superior to Hoefdding trees on these data sets.

The adult data set is quite large in size, consisting of about 32,500 instances. The problem is to predict whether or not the income of a person exceeds the $50,000 per year. Our experiment starts with an initial learning phase on the first 1,000 instances, while the rest of the data is used for incremental learning and evaluation with $M = 200$ and $N = 50$. As can be seen in Fig. 9, the performance curves are not very smooth, indicating a strong influence of the order of the data records. Nevertheless, eFPT seems to slightly outperform the Hoeffding tree in the beginning, while the latter becomes better with an increasing volume of data. The results here are averaged over 100 randomly shuffled versions.

5.3. Model Size

Apart from comparing the performance of the methods, we also looked at the size of the models they produce. In this regard, eFPT is clearly superior. In fact, the size of the fuzzy pattern trees is rather stable over time and remains on a low level—the maximum size observed in the two experiments is 19 nodes. As opposed to this, the Hoeffding tree seems to grow linearly with the length of the stream and becomes as large as 747 nodes in the hyperplane and 851 nodes in the random trees setting (see Fig. 10). Needless to say, a model of that size is no longer understandable. Quite similar observations can be made for the real data sets (see Fig.11); the wine data has to be considered with reservation, however, since these data sets are not long enough to convey long-term effects (see Fig. 12).

6. Summary and Conclusions

We have proposed an evolving version of the fuzzy pattern tree classifier that meets the increased requirements of incremental learning on data streams. The key idea of eFPT is to maintain, in addition to the current model, a set of neighbor trees that can replace the current model if the performance of the latter is no longer optimal. Thus, a modification of the current model is realized implicitly in the form of a replacement by an alternative tree. A replacement decision is made on the basis of the performance of all models, which is monitored continuously on a sliding window of fixed length.

In an experimental study, we compared eFPT with Hoeffding trees, a state-of-the-art classifier for learning from data streams, on real and synthetic data. The results we obtained are quite promising. Put in a nutshell, they suggest that eFPT is competitive in terms of accuracy, while being less affected by concept drift and producing smaller, more compact models. These criteria are of course interrelated: The smaller a model is, the more easily and quickly it can be adapted in the case of a concept drift; besides, compactness of a model is of course also desirable from an understandability point of view. On the other hand, producing large models can be advantageous in cases where the target concept to be learned is complex and the data generating process sufficiently stable; in our experiments, Hoeffding trees performed comparatively well especially in these cases.

In future work, we intend to generalize our current version of eFTP from binary to multi-class classification. Moreover, we are also interested in developing an evolving version of fuzzy pattern trees for regression. An implementation of our current algorithm, running under the

MOA framework, can be downloaded at `http://www.uni-marburg.de/fb12/kebi/research`.

References

[1] Angelov, P. P., Filev, D. P., Kasabov, N., 2010. Evolving Intelligent Systems. John Wiley and Sons, New York.

[2] Angelov, P. P., Lughofer, E., Zhou, X., 2008. Evolving fuzzy classifiers using different model architectures. Fuzzy Sets and Systems 159 (23), 3160–3182.

[3] Ben-David, S., Gehrke, J., Kifer, D., 2004. Detecting change in data streams. In: Proceedings of the 30th International Conference on Very Large Data Bases. Toronto, Canada, pp. 180–191.

[4] Bifet, A., Kirkby, R., August 2009. Massive Online Analysis Manual.
URL `http://moa.cs.waikato.ac.nz/`

[5] Domingos, P., Hulten, G., 2001. Catching up with the data: Research issues in mining data streams. 2001 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, Santa Barbara, CA, USA.

[6] Frank, A., Asuncion, A., 2010. UCI machine learning repository.
URL `http://archive.ics.uci.edu/ml`

[7] Gaber, M. M., Zaslavsky, A., Krishnaswamy, S., 2005. Mining data streams: A review. ACM SIGMOD Record 34 (1).

[8] Gama, J., Gaber, M. M., 2007. Learning from Data Streams. Springer-Verlag, Berlin, New York.

[9] Garofalakis, M., Gehrke, J., Rastogi, R., 2002. Querying and mining data streams: you only get one look. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data. Madison, Wisconsin, USA, pp. 635–635.

[10] Golab, L., Ozsu, M. T., 2003. Issues in data stream management. SIGMOD Record 32(2), 5–14.

[11] Huang, Z., Gedeon, T. D., Nikravesh, M., 2008. Pattern trees induction: A new machine learning method. IEEE Transactions on Fuzzy Systems 16(4), 958–970.

[12] Hulten, G., Domingos, P., 2003. VFML a toolkit for mining high-speed time-changing data streams.
URL `http://www.cs.washington.edu/dm/vfml/`

[13] Hulten, G., Spencer, L., Domingos, P., 2001. Mining time-changing data streams. In: Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. San Francisco, CA, USA, pp. 97 – 106.

[14] Klement, E. P., Mesiar, R., Pap, E., 2002. Triangular Norms. Kluwer Academic Publishers, Dordrecht.

[15] Lughofer, E., 2008. FLEXFIS: A robust incremental learning approach for evolving Takagi-Sugeno fuzzy models. IEEE Transactions on Fuzzy Systems 16 (6), 1393–1410.

[16] Lughofer, E., 2011. Evolving Fuzzy Systems: Methodologies, Advanced Concepts and Applications. Springer-Verlag, Berlin, Heidelberg.

[17] Quinlan, J., 1993. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, San Francisco.

[18] Schweizer, B., Sklar, A., 1983. Probabilistic Metric Spaces. Elsevier & North-Holland, New York.

[19] Senge, R., Hüllermeier, E., 2010. Pattern trees for regression and fuzzy systems modeling. In: Proceedings WCCI 2010, World Congress on Computational Intelligence. Barcelona, Spain.

[20] Senge, R., Hüllermeier, E., 2010. Top-down induction of fuzzy pattern trees. IEEE Transactions on Fuzzy Systems 19 (2), 241–252.

[21] Witten, I. H., Frank, E., 2005. Data Mining: Practical machine learning tools and techniques, 2nd Edition. Morgan Kaufmann Publishers, San Francisco.

[22] Yager, R., 1988. On ordered weighted averaging aggregation operators in multi criteria decision making. IEEE Transactions on Systems, Man and Cybernetics 18(1), 183–190.

[23] Yi, Y., Fober, T., Hüllermeier, E., 2008. Fuzzy operator trees for modeling rating functions. International Journal of Computational Intelligence and Applications 8 (4), 413–428.

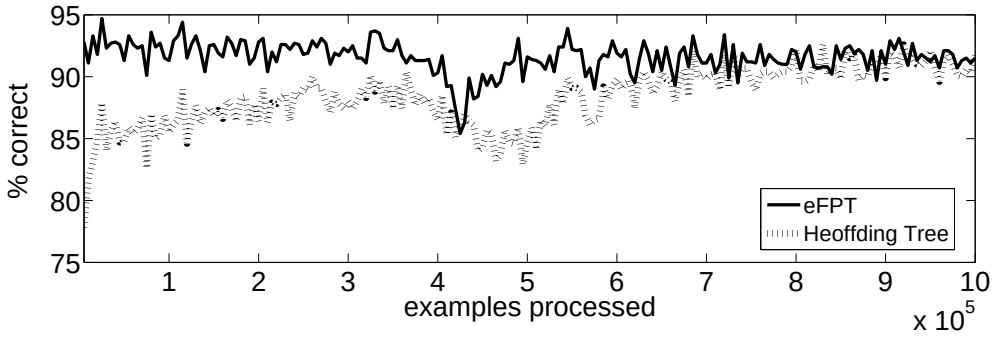


Figure 5: Performance on the hyperplane data.

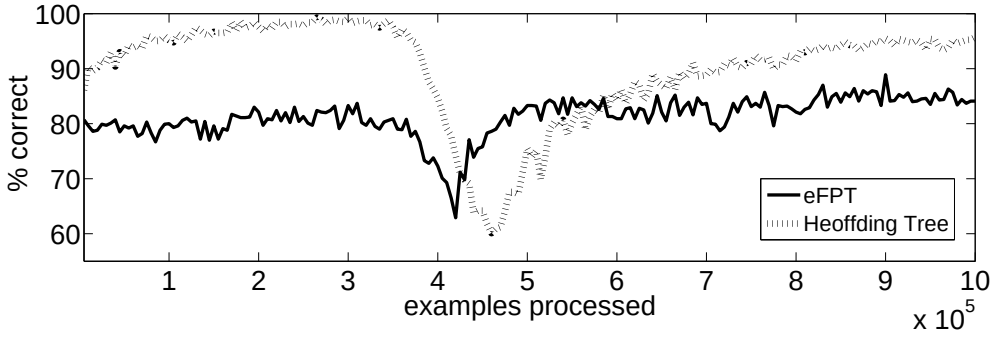


Figure 6: Performance on the random tree data.

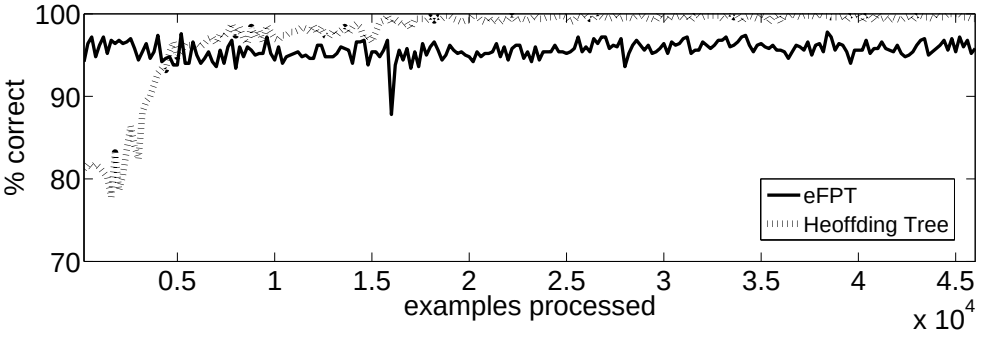
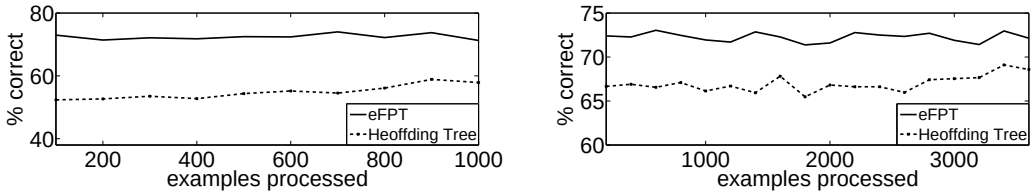


Figure 7: Performance on the shuttle data set.



(a) red wine (b) white wine

Figure 8: Performance on the wine quality data set.

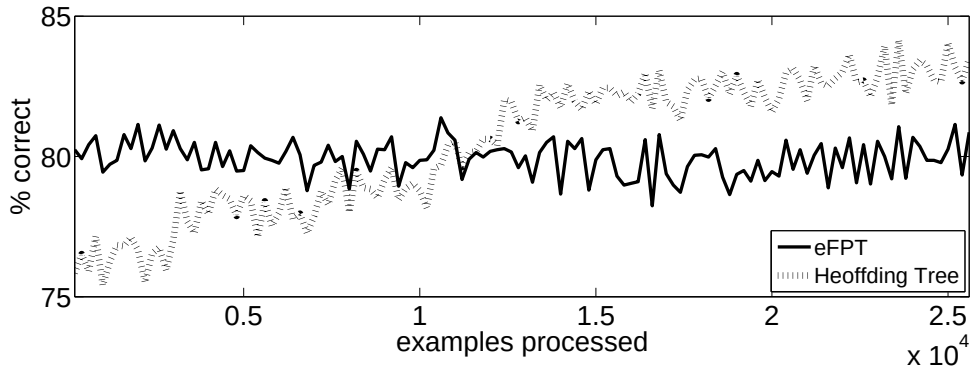
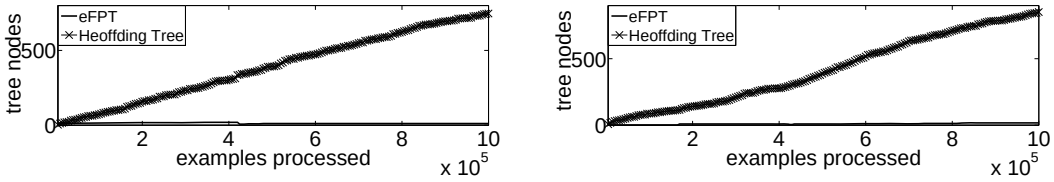
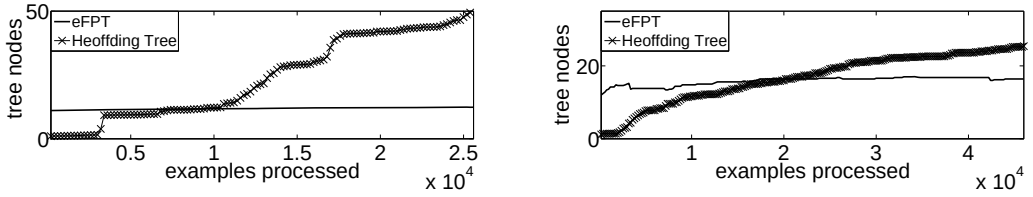


Figure 9: Performance on the adult data set.



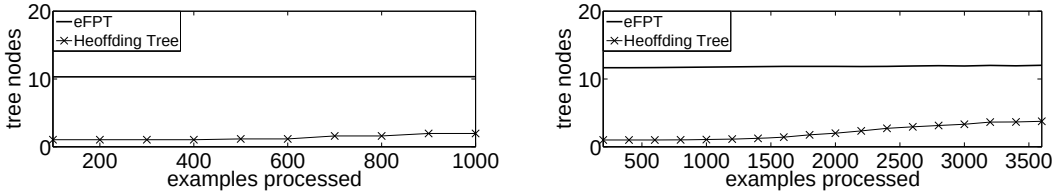
(a) hyperplane data (b) random trees data

Figure 10: Model size of eFPT and Hoeffding trees.



(a) shuttle data (b) adult data

Figure 11: Model size of eFPT and Hoeffding trees on the adult and shuttle data.



(a) red wine (b) white wine

Figure 12: Model size of eFPT and Hoeffding trees on the wine data.