Partial tree-edit distance: a solution to the default class problem in pattern-based tree classification

Maciej Piernik and Tadeusz Morzy

Institute of Computing Science, Poznan University of Technology
ul. Piotrowo 2, 60-965 Poznan, Poland
maciej.piernik@cs.put.poznan.pl

Abstract. Pattern-based tree classifiers are capable of producing high quality results, however, they are prone to the problem of the default class overuse. In this paper, we propose a measure designed to address this issue, called partial tree-edit distance (PTED), which allows for assessing the degree of containment of one tree in another. Furthermore, we propose an algorithm which calculates the measure and perform an experiment involving pattern-based classification to illustrate its usefulness. The results show that incorporating PTED into the classification scheme allowed us to significantly improve the accuracy on the tested datasets.

Keywords: Tree-subtree similarity, tree classification, tree-edit distance

1 Introduction

Rooted, ordered, labeled tree is a popular data structure which finds various applications in many different domains. One of the most important issues concerning this structure is similarity computation. This problem has several practical applications such as XML document similarity [1], comparison of RNA secondary structures [2], natural language processing [3], or data integration [4, 5]. In this paper, we focus on the problem of tree similarity in the context of pattern-based tree classification.

The process of pattern-based tree classification is as follows. First, a training dataset of trees is mined for patterns (frequent subtrees), separately for each class. Next, based on the discovered patterns, a classifier is constructed as a set of rules: $pattern \rightarrow class$, where *pattern* denotes a frequent subtree mined from the training dataset with a given *class*. Once a classifier is created, each new unclassified tree is tested against every rule for pattern containment (e.g., with subtree matching) and assigned to the class from which it contains the highest percentage of patterns.

Pattern-based tree classification is a straightforward method capable of producing high quality results [6]. However, there are two problematic cases for this

approach: (1) if a classified tree contains the same percentage of patterns from two or more classes, (2) if a classified tree does not contain any pattern. The first case can be resolved by creating a ranking of patterns and weighing them according to their importance. In the second case, however, a common approach is to use a so called *default class*, which assigns all unmatched trees to a single class, e.g., a majority class in the training dataset. Clearly, such a situation should be avoided as there is a high chance of deteriorating the classification quality. For traditional, transactional data, other solutions than the default class exist [7], most of which are based on partial similarity of a rule with a classified object. In the tree processing domain, however, to the best of our knowledge, an accurate tree-subtree similarity measure is not available.

In this paper, we analyze the above described problem of tree-subtree similarity defined as follows. Given two trees P (a pattern tree) and D (a document tree), find how much does P need to be modified to become a subtree of D. In order to answer this question, we propose a new distance measure, called *partial tree-edit distance* (PTED), along with an algorithm to calculate it. We will show that incorporating PTED into the classification scheme significantly improves upon the existing methods of dealing with the default class problem.

The remainder of this paper is organized as follows. Section 2 gives the background of related work for the proposed measure. Section 3 introduces necessary notation and definitions. In Section 4 we describe and define the partial tree-edit distance measure. Section 5 presents an efficient algorithm which calculates the proposed measure. In Section 6 we empirically evaluate the algorithm in terms of time complexity and illustrate the usefulness of the proposed measure with an experiment involving pattern-based tree classification. Finally, in Section 7 we conclude the article and draw lines of further research.

2 Related work

One of the first attempts at solving the tree-subtree similarity problem was proposed by Zhang and Shasha [8]. The authors present a generalization of tree-edit distance, which can be stated as follows. Given trees T_1 and T_2, what is the minimum distance between T_1 and T_2 when zero or more subtrees can be removed from T_2 at no cost. This problem is similar to the question stated in this paper, however, it works closely only when the root node of T_1 is mapped to the root node of T_2. Furthermore, it allows for all edit operations to appear in both trees while we allow only for the pattern tree to be modified. Given our motivation, we need our measure to identify subtrees anywhere in the hierarchy of a tree and only by modifying a pattern tree in the least invasive way.

Another problem, similar to the one stated in this paper, was explored by Augsten et al. [5]. The problem concerned finding the best k matches of a small query tree in a large document tree. This approach, however, is also unsuited for our problem because it focuses on subtrees spanning to the bottom (leaf nodes) of a document tree while we need our measure to identify subtrees of any shape and depth.

Cohen and Or [9] recently proposed a framework for solving the subtree similarity-search problem, along with an indexing structure to enhance the efficiency of the searching [10]. Their solution is generic and allows for a wide variety of similarity measures to be used. However, the aim and scope of the framework is different to the problem addressed in this paper. The authors focus on finding several similar subtrees using some subtree similarity measure while we focus on the sole problem of how to measure the subtree similarity. Therefore, the scope of our work is different to that of Cohen and Or, nevertheless, complementary.

Much effort has also been put into tree pattern matching, which is a more general problem than the one stated in this paper [1]. An interesting approach to tree pattern matching, called tree pattern relaxation, was proposed by Amer-Yahia et al. [11]. The authors propose four relaxations of pattern constraints which allow for approximate pattern matching. This approach, however, requires specifically constructed weighted patterns, what makes it unsuited for our problem since our patterns are simple trees.

Another pattern matching related problem is approximate tree matching with variable length don't cares [12]. In 1993 Zhang et al. adopted the idea of VLDC's from string matching to tree matching. However, this approach, similarly to tree pattern relaxation, requires patterns of a specific structure. This requirement makes this method unusable in our case, since we are focusing on simple subtrees.

All of these solutions tackle similar issues to the one stated in this article. However, their detailed characteristics showcase that they are unsuited for our particular problem.

3 Preliminaries

A *tree* T is a connected graph with $|T|$ nodes and $|T|-1$ edges. We call a tree T *rooted* if all edges in T are directed away from one designated node, called a *root* node. We denote a tree T rooted at a node x by T_x and a root node of a tree T by r^T. If two nodes x and y are connected with an edge and x is closer to the root node than y, then x is a *parent* of y and y is a *child* of x. Nodes without any children are called *leaf* nodes. Children of the same node x are called *siblings* and the number of all children of x is denoted by $|x|$. We also designate a special node λ, called *empty node*.

A rooted tree T is *ordered* if there exists a total order among all nodes in T. In our approach, we order the nodes according to the pre-order traversal. The fact that a node x appears in a tree before a node y is expressed by $x < y$.

A tree T is *labeled* if every node in this tree $x \in T$ has a label assigned to it, symbolized by $l(x)$. For convenience, hereinafter, a rooted, ordered, labeled tree will be referred to as *tree*.

An ordered set of trees is called a *forest*. A forest F containing trees rooted at all children nodes of a node x is denoted by F_x. The rightmost tree of a forest F is denoted by $\bar{F}$. A forest F without a tree T is symbolized by $F-T$ and the number of nodes in all trees in forest F is symbolized by $|F|$.

A tree S whose nodes and edges form subsets of nodes and edges of another tree T is called a *subtree* of T. We denote that S is a subtree of T by $S \subseteq T$.

Let us now define the edit operations which can be performed on tree nodes. In general, there are three basic edit operations: *insertion*, *deletion*, and *relabeling*. By inserting a node x into a tree T at a node y, x becomes a child of the parent of y, taking y's place in the sibling order, while y becomes a child of x. When deleting a node x from a tree T, all children of x become the children of the parent of x. Consequently, when x is a root node, the result is a forest F_x.

4 Partial tree-edit distance

4.1 Conceptual description

To illustrate how partial tree-edit distance works, first let us consider an example presented in Fig. 1. In this example, by T^i we will denote the i-th node (according to the pre-order traversal) in tree T. As the question stated in this paper implies, the task is to determine how many operations need to be performed on P for it to become a subtree of D. Looking at the example, clearly, P is not a subtree of D. However, as illustrated with the grey areas, there is a part of P which can be directly mapped into D. Namely, nodes P^2, P^4, and P^5 can be mapped into D^1, D^5, and D^{11}, respectively, as they have the same labels. As a result of this mapping, we also have to map P^3 into D^4. This time, however, we need to use the relabeling operation as the labels are different. Finally, as nodes P^1, P^6, and P^7 have no corresponding nodes in D, they have to be removed using the deletion operation. Therefore, the total number of edit operations required to transform P into a subtree of D is 4 (1 relabeling, 3 deletions).

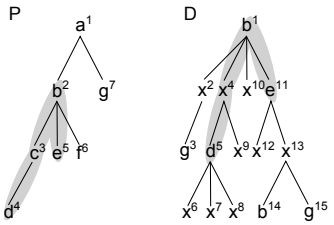


Fig. 1. Example of fitting a pattern tree P into a document tree D. The nodes in P covered by the grey area are relabeled to the corresponding nodes covered by the grey area in D, while the nodes in P uncovered by the grey area are deleted. Numbers represent the order of pre-order traversal.

So far, we have only used relabeling and deletion. Furthermore, we only deleted the root node (P^1) and the leaf nodes (P^6, P^7). Let us now discuss the possible consequences of using other edit operations, namely, deletion of inner nodes and insertion of inner and non-inner nodes. Inserting a non-inner node

into P does not make sense, since it could only increase the number of operations needed to fit P into D. That is why, in partial tree-edit distance insertion of non-inner nodes is forbidden. Deleting or inserting an inner node results in a children nodes' transfer, so the internal structure of a tree is altered. In our case however, given the pattern-based classification motivation, allowing for such operations to appear would alter the inner structure of the patterns. Since patterns are frequent subtrees, by deleting non-inner nodes we are guaranteed to obtain structures which are at least as frequent as the base structure thanks to the anti-monotonicity property of the support measure (any subtree of a frequent subtree will have equal or higher support). However, allowing for an inner node to be inserted or deleted from a pattern results in a subtree of which frequency we know absolutely nothing about, therefore, it cannot be called a pattern anymore. It may even happen that such a pattern does not appear in the dataset at all. Therefore, insertion and deletion of inner nodes into a pattern may lead to wrong class assignments and deteriorate the overall classification quality.

It is worth noting that other applications may benefit from allowing these additional edit operations to appear and exploring such operations constitutes an interesting topic for a future research. However, given our main motivation, they are out of the scope of this paper.

Given the above, partial tree-edit distance is defined around two edit operations: deletion of non-inner nodes and relabeling. Both of these operations have an associated cost, which can be universally expressed with the following formula:

$$c(x, y) = \begin{cases} 0 & x = \lambda \\ w_d & y = \lambda \\ w_r & otherwise \end{cases} \tag{1}$$

where x and y are nodes, λ is an empty node, and w_d and w_r are user-defined weights associated with deletion and relabeling. Let s be a sequence of these two operations. *Partial tree-edit sequence* s between two trees P and D is a sequence which transforms P into any subtree of D. The cost $c(s)$ of partial tree-edit sequence s is the total cost of all operations in s. *Partial tree-edit distance (PTED)* $\Delta(P, D)$ between a pattern tree P and a document tree D is the minimal cost of all possible partial tree-edit sequences between P and D.

As we can see, the measure works as a combination of subtree matching and tree-edit distance, producing a distance equal 0 when a pattern appears in a tree, and a value between 0 and the size of a pattern, otherwise.

4.2 Formal definition

Definition 1. *A partial mapping m between a pattern tree P and a document tree D is a subset of $P \times (D \cup \{\lambda\})$, such that: (1) each node from P appears in m exactly once, (2) each node from D appears in m at most once, (3) for any $(x, x'), (y, y') \in m$ where $x' \neq \lambda$ and $y' \neq \lambda$: x is a parent of $y \Leftrightarrow x'$ is a parent of y', (4) for any $(x, x'), (y, y') \in m$ where x is a sibling of y and x' is a sibling of y': $x < x' \Leftrightarrow y < y'$.*

Each element in the mapping $(x, x') \in m$ represents an edit operation and has an associated cost $c(x, x')$, as defined in Equation 1. An element where $x' = \lambda$ represents a deletion while an element where $x' \neq \lambda$ represents a relabeling. The cost $c(m)$ of a partial mapping m is a sum of costs of all elements in m.

Definition 2. *Partial tree-edit distance $\Delta(P, D)$ between a pattern tree P and a document tree D is the minimal cost of all possible partial mappings between P and D.*

Now, we will introduce a recursive formula which calculates partial tree-edit distance. The formula works in two stages. The purpose of the first stage, performed by a main function Δ and defined in Equation 2, is to place P at each possible position in D.

$$\Delta(P, D) = \min_{x \in P, y \in D} \Big\{ \delta(\{T_x\}, \{T_y\}) + \sum_{\{z \in P : z \notin T_x\}} c(z, \lambda) \Big\} \tag{2}$$

Next, for each placement of P in D, the second stage takes place. The goal of the second stage, performed by an auxiliary function δ defined in Equation 3, is to check how well does P fit in D, at a given placement. The function accepts two forests G and H as parameters and recursively considers 3 cases: ignoring the rightmost tree of H, deleting the rightmost tree of G, and fitting the rightmost tree of G into the rightmost tree of H.

$$\delta(G, H) = \min \begin{cases} \delta\left(G, H - \bar{H}\right) \\ \delta\left(G - \bar{G}, H\right) + \delta\left(\{\bar{G}\}, \emptyset\right) \\ \delta\left(G - \bar{G}, H - \bar{H}\right) + \delta\left(F_{r^{\bar{G}}}, F_{r^{\bar{H}}}\right) + c(r^{\bar{G}}, r^{\bar{H}}) \end{cases} \tag{3}$$

Equation 4 defines the boundary conditions of the auxiliary function δ. The first two cases reflect the fact that the cost of fitting an empty pattern into any tree equals 0, while the third case signifies that the cost of fitting any non-empty pattern into an empty tree equals the cost of removing the whole pattern.

$$\begin{aligned} &\delta(\emptyset, \emptyset) = 0 \\ &\delta(\emptyset, H) = 0 \\ &\delta(G, \emptyset) = \delta\left(G - \bar{G}, \emptyset\right) + \delta\left(F_{r^{\bar{G}}}, \emptyset\right) + c(r^{\bar{G}}, \lambda) \end{aligned} \tag{4}$$

5 Algorithm

In this section we propose an algorithm which calculates partial tree-edit distance. Similarly to the formal definition, the algorithm consists of two main components: (1) the main loop Δ which places P at every possible position in D and (2) the auxiliary function δ which checks the quality of each placement. The algorithm for the main loop is a trivial implementation of Equation 2, so

Algorithm 1 Partial tree edit distance algorithm: $\delta(T_v, T_w)$

Require: trees T_v and T_w,
Ensure: a cost of a partial mapping m between T_v and T_w with restriction $(v, w) \in m$
1: $tab \leftarrow [|v| + 1, |w| + 1]$
2: **for** $j = 0..|w|$ **do**
3: $\quad tab[0, j] \leftarrow 0$;
4: **end for**
5: **for** $i = 1..|v|$ **do**
6: $\quad tab[i, 0] \leftarrow tab[i - 1, 0] + (|T_{v_i}|) \cdot w_d$;
7: **end for**
8: **for** $i = 1..|v|$ **do**
9: $\quad$ **for** $j = 1..|w|$ **do**
10: $\quad\quad tab[i, j] \leftarrow \min\{$
$\quad\quad\quad tab[i - 1, j] + (|T_{v_i}|) \cdot w_d,$
$\quad\quad\quad tab[i, j - 1],$
$\quad\quad\quad tab[i - 1, j - 1] + \delta(T_{v_i}, T_{w_j})$
$\quad\quad\}$;
11: $\quad$ **end for**
12: **end for**
13: **return** $tab[|v|, |w|] + (l(v) = l(w)\ ?\ 0 : w_r)$;

we will skip the pseudocode for this step. The auxiliary function is implemented with a dynamic programming algorithm, given in Algorithm 1.

The algorithm accepts two trees T_v and T_w as parameters and outputs the minimal cost of a partial mapping between T_v and T_w, given that v is mapped into w. Variable *tab* stores the intermediate results of mapping the children nodes of v into the children nodes of w, so it is an $\mathcal{R}^{|v|+1 \times |w|+1}$ matrix (Line 1). In Lines 2-4 the top row in the matrix is initialized to 0. This reflects the fact that the subtrees in the right tree can be removed without any cost (ignored). In practice, it fulfills the second boundary condition from Equation 4. In Lines 5-7, the left column is initialized with the cumulative cost of deleting consecutive subtrees of v ($tab[i, 0]$ = cost of removing $T_{v_1}..T_{v_i}$). These values fulfill the third boundary condition from Equation 4. Lines 8-12 contain the main loop of the auxiliary function. It scans through all children nodes of v and w and for each pair v_i, w_j stores a temporary result $tab[i, j]$ which holds the minimal cost of mapping $v_1..v_i$ into $w_1..w_j$. This cost is computed in Line 10 as the minimum of 3 expressions, reflecting the 3 options in Equation 3:

- $tab[i-1, j] + (|v_i| + 1) \cdot w_d$ accounts for removing the rightmost subtree from the left tree;
- $tab[i, j - 1]$ accounts for ignoring the rightmost subtree from the right tree;
- $tab[i - 1, j - 1] + \delta(T_{v_i}, T_{w_j})$ accounts for mapping the rightmost subtree of the left tree into the rightmost subtree of the right tree.

In the end, $tab[|v|, |w|]$ holds the minimal cost of mapping the children of v into the children of w (with descendants). Finally, in Line 13, by adding the cost of mapping v into w we obtain the total cost of the minimal partial mapping between T_v and T_w with v mapped into w. This concludes the algorithm.

Let us now analyze the complexity of the presented algorithm. It is easy to notice that the auxiliary function algorithm is an adoption of the algorithm for the Levenstein distance between two sequences [13], which has a quadratic

complexity. Here however, the auxiliary function is called within the main loop which is also quadratic in time, so the overall complexity is $O(n^4)$. However, the auxiliary function runs only as deep as the height of the smaller tree, so since pattern trees are usually much smaller than document trees, the algorithm will usually be more efficient than the complexity suggests.

6 Experimental evaluation

6.1 Datasets and experimental setup

During the experiments, we used both real and synthetic datasets containing XML documents represent as rooted, ordered, labeled trees. For the time complexity evaluation, we generated a dataset of 20 documents ranging between 100 and 2000 in the number of elements. To generate this dataset, we used the software developed by Zaki [14].

To test the applicability of PTED in pattern-based classification, we used the datasets created by Zaki and Aggarwal [6]. The synthetic datasets DS1-4, were generated by the aforementioned authors and are composed of a training and a testing set each, containing between 60000 and 100000 documents. The real datasets CS1-3, each consisting of around 8000 documents, contain web logs categorized into two classes (for a detailed description see [6]). Since they were not divided into training and testing sets, we used each for both purposes and cross-validated them with one another. By CSXY we will denote the CSX set used for training and CSY for testing. This gives us a total of 10 tests: 4 on synthetic and 6 on real data. The minimal frequency of a subtree required to consider it a pattern was 0.1% for DS datasets and 1% for CS datasets.

All classifiers were evaluated using a weighted accuracy measure [6], defined as follows:

$$Accuracy = \sum_{c \in \mathcal{C}} \left(w_c \cdot \frac{|\mathcal{D}_c^{test}|}{|\mathcal{D}_c|} \right) \tag{5}$$

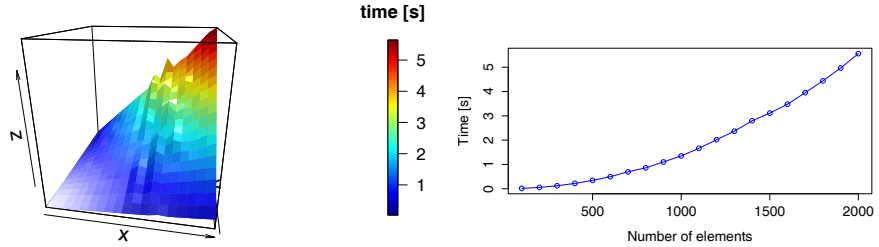
where $\mathcal{C}$ is the set of all classes, $\mathcal{D}_c^{test}$ is the set of documents correctly assigned to class c, $\mathcal{D}_c$ is the set of documents which should be assigned to class c, and w_c is a weight associated with each of the classes. Similarly as Zaki and Aggarwal [6], we analyzed three weighting models:

- *proportional*: $w_c = |\mathcal{D}_c|/|\mathcal{D}|$ — classes weighted proportionally to their distribution in the training dataset,
- *equal*: $w_c = 1/|\mathcal{C}|$ — all classes weighted equally,
- *inverse*: $w_c = \frac{1/\mathcal{D}_c}{\sum_{c' \in \mathcal{C}} 1/|\mathcal{D}_{c'}|}$ — classes weighted inversely to their distribution in the training dataset.

Additionally, we used the Friedman test [15] to determine whether the compared approaches performed significantly differently and a post-hoc Nemenyi test [15] to check if the proposed solution significantly improved the quality of classification.

6.2 Time complexity evaluation

To assess the time complexity of the proposed algorithm we used the generated dataset containing 20 documents of increasing sizes. For each pair of documents, we calculated partial tree-edit distance 100 times and measured the average computing time. The results of this test are presented in Fig. 2.



(a) PTED processing time for varying pattern and document sizes. Axes X and Y represent the sizes of pattern and document trees while the Z axis represents processing time.

(b) PTED processing time for increasing pattern and document size.

Fig. 2. Time complexity of the algorithm calculating partial tree-edit distance.

Fig. 2(a) illustrates how much time it takes to calculate PTED for trees of various sizes. First, let us observe how the algorithm behaves when both pattern and document trees are expanded. As we can see, with increasing sizes of both trees, processing time presents a polynomial growth. This is reflected in the spine on the 3D chart (the diagonal line w.r.t. X and Y axes) which is extracted and visualized in Fig. 2(b) to facilitate the observation. When increasing the size of only one of the trees, the increase in processing time is much slower. Considering the purpose of our measure, this is a very important observation. Since we are assessing the degree of containment of one tree in another, the left (pattern) tree should be usually much smaller than the right (document) tree. This is certainly true in the practical example involving pattern-based classification presented in Section 6.3, as the largest patterns discovered in all experiments contained only 11 nodes. Additionally, it is worth noticing that the chart is symmetrical w.r.t. the X/Y diagonal. This means that the algorithm behaves the same regardless of which tree is bigger.

6.3 Practical application

To empirically evaluate the usefulness of PTED, we performed an experiment involving pattern-based tree classification. Our goal of is to illustrate the importance of the default class problem and show how partial tree-edit distance can be used to address it. We compare four ways of dealing with this problem.

In the first three approaches, we use different methods for determining the default class, as proposed by Zaki and Aggarwal in the state-of-the-art XRules classifier [6]. All three approaches determine the dafault class based on the class distribution in the documents from the training dataset which are not covered by any rule (do not contain any of the discovered frequent subtrees). Moreover, each method maximizes the accuracy measure from Equation 5 w.r.t. one of the three weights: proportional, equal, and inverse. Analogously to the accuracy measure, given that $\mathcal{D}_c$ represents the training documents with class c and $\bar{\mathcal{D}}_c$ represents a portion of these documents uncovered by any of the rules, the method for determining the default class is defined as follows:

$$Class(D) = \arg\max_{c \in \mathcal{C}} \left(w_c \cdot \frac{|\bar{\mathcal{D}}_c|}{|\mathcal{D}_c|} \right) \tag{6}$$

where w_c is one of the three previously defined weights: proportional, equal, or inverse.

In the last approach, we use partial tree-edit distance to assign each ambiguous document D to one of the classes according to the following formula:

$$Class(D) = \arg\max_{c \in \mathcal{C}} \left(\sum_{P \in \mathcal{P}_c} \left(1 - \frac{\Delta(P, D)}{|P|} \right) \right) \tag{7}$$

where $\mathcal{C}$ is a set of classes and $\mathcal{P}_c$ is a set of patterns with class c. Intuitively, this formula measures the similarity of D with all patterns in each class and assigns it to the class with the highest cumulative similarity.

Table 1 presents the results of this experiment. The first column represents the datasets used in each test (the values in square brackets [DC%] will be explained later) while the following columns present the accuracies achieved by each of the described approaches. The results of the proportional and equal methods are presented in a single column as they produced the same outcome on every dataset. Each method was evaluated with three variants of the accuracy measure and the differences in the results were tested for statistical significance. In order to determine whether by using PTED we were able to significantly improve the quality of classification, for every dataset we ranked each algorithm's performance from 1 to 3, where 1 is the highest and 3 is the lowest score. In cases when one or more of the algorithms were tied, average ranks were assigned (e.g., if two algorithms were tied at the 2nd place, each was granted a rank of 2.5). Once created, the ranking (presented in the "Avg. rank" row) was used to perform the Friedman test [15]. The null-hypothesis for this test is that there is no difference in the performance between the tested methods. Moreover, in case of rejecting this null-hypothesis we used the Nemenyi post-hoc test [15] to verify whether the performance of the best approach is statistically different from the remaining approaches.

The results clearly illustrate that by using partial tree-edit distance we were able to improve the classification quality in almost every test, regardless of the applied accuracy measure. This outcome is partially confirmed by the Friedman

Table 1. Comparison of methods for handling unclassified examples in a pattern-based classifier. Bold indicates the best result.

Approach	*Prop./Eq.*	*Inv.*	*PTED*	*Prop./Eq.*	*Inv.*	*PTED*	*Prop./Eq.*	*Inv.*	*PTED*
Dataset [DC%]	Proportional accuracy [%]			Equal accuracy [%]			Inverse accuracy [%]		
DS1 [56]	**53.37**	47.74	47.74	**52.35**	50.47	**52.35**	**56.95**	47.57	**56.95**
DS2 [70]	**62.38**	34.23	34.23	48.74	**52.47**	48.74	**63.25**	42.56	**63.25**
DS3 [74]	54.03	54.03	**59.93**	54.03	54.03	**59.93**	54.03	54.03	**59.93**
DS4 [63]	**54.02**	**54.02**	**54.02**	**53.43**	**53.43**	**53.43**	**52.85**	**52.85**	**52.85**
CS12 [47]	72.32	72.32	**72.44**	64.04	64.04	**64.43**	55.76	55.76	**56.43**
CS21 [49]	**72.78**	**72.78**	**72.78**	62.62	62.62	**62.64**	52.47	52.47	**52.50**
CS13 [48]	72.26	72.26	**72.69**	63.33	63.33	**64.33**	54.40	54.40	**55.96**
CS31 [50]	72.63	72.63	**73.61**	62.68	62.68	**67.27**	52.73	52.73	**60.93**
CS23 [47]	73.61	73.61	**73.64**	63.60	63.60	**63.66**	53.59	53.59	**53.67**
CS32 [50]	73.17	73.17	**73.71**	62.79	62.79	**67.07**	52.42	52.42	**60.44**
Avg. rank	2.10	2.40	**1.50**	2.35	2.35	**1.30**	2.25	2.55	**1.20**

statistical test. The critical value of the Friedman statistic for the analyzed setting at $\alpha = 0.05$ is 3.560 and the F scores for the proportional, equal, and inverse accuracy tests are 2.392, 5.229, and 9.090, respectively. Therefore, the analyzed approaches perform significantly differently according to the two latter measures, but not according to the first one. The additional post-hoc Nemenyi test reveals that the critical distance (difference in average ranks) required to deem an approach significantly superior to others equals 1.048 at $\alpha = 0.05$, so PTED is indeed significantly better than any of the three default class strategies according to equal and inverse accuracy.

In order to emphasize the gravity of the default class problem, we additionally measured how many times the default class had to be used in the analyzed datasets. The numbers in the square brackets in the first column of Table 1 ([DC%]) present the percentage of documents from the test set which were uncovered by any pattern from the classifier. In every test, this problem concerned around half or more documents (e.g., for test DS3 which contains 100000 test documents there were 73906 documents without any matching pattern). By using partial tree-edit distance we are able to treat each of these cases individually instead of assigning them arbitrarily to the same class.

7 Conclusions

In this paper, we introduced a new measure for assessing the tree-subtree similarity, called partial tree-edit distance (PTED), which describes to what extent one tree is included in another. We also proposed an algorithm which calculates the proposed measure in polynomial time. Furthermore, we performed an experiment involving pattern-based tree classification using partial tree-edit distance to illustrate the usefulness of the measure. The results show that by using PTED we were able to significantly improve the classification quality over the classical pattern-based approach.

The measure proposed in this paper opens several possibilities of future research. It could be used to improve the quality of approximate subtree matching,

XML querying, ranking, clustering, or classification. Encouraged by the results achieved in our experiments, we plan on developing a new pattern-based XML classification algorithm designed around partial tree-edit distance.

Acknowledgments

This research is partly funded by the Polish National Science Center under Grant No. DEC-2015/19/B/ST6/02637.

References

1. Hachicha, M., Darmont, J.: A survey of xml tree patterns. IEEE Trans. on Knowl. and Data Eng. **25**(1) (2013) 29–46
2. Dulucq, S., Tichit, L.: Rna secondary structure comparison: Exact analysis of the zhang–shasha tree edit algorithm. Theor. Comput. Sci. **306**(1-3) (2003) 471–484
3. Kouylekov, M., Magnini, B.: Combining lexical resources with tree edit distance for recognizing textual entailment. In: Proceedings of the 1st International Conference on Machine Learning Challenges. MLCW'05 (2006) 217–230
4. Augsten, N., Bohlen, M., Dyreson, C., Gamper, J.: Approximate joins for data-centric xml. In: Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on. (2008) 814–823
5. Augsten, N., Barbosa, D., Bohlen, M., Palpanas, T.: Efficient top-k approximate subtree matching in small memory. IEEE Trans. on Knowl. and Data Eng. **23**(8) (2011) 1123–1137
6. Zaki, M.J., Aggarwal, C.C.: XRules: An Effective Algorithm for Structural Classification of XML Data. Mach. Learn. **62**(1-2) (2006) 137–170
7. Stefanowski, J.: Algorithims of rule induction for knowledge discovery (2001) Habilitation thesis.
8. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM J. Comput. **18**(6) (1989) 1245–1262
9. Cohen, S., Or, N.: A general algorithm for subtree similarity-search. In: Proceedings of the 30th International Conference on Data Engineering. ICDE'14 (2014) 928–939
10. Cohen, S.: Indexing for subtree similarity-search using edit distance. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. SIGMOD '13 (2013) 49–60
11. Amer-Yahia, S., Cho, S., Srivastava, D.: Tree pattern relaxation. In: Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology. EDBT '02 (2002) 496–513
12. Zhang, K., Shasha, D., Wang, J.T.L.: Approximate tree matching in the presence of variable length don't cares. J. Algorithms **16** (1993) 33–66
13. Levenshtein, V.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Soviet Physics Doklady **10** (1966) 707
14. Zaki, M.J.: Efficiently mining frequent trees in a forest: Algorithms and applications. IEEE Trans. on Knowl. and Data Eng. **17**(8) (August 2005) 1021–1035
15. Demsar, J.: Statistical Comparisons of Classifiers over Multiple Data Sets. J. Mach. Learn. Res. **7** (2006) 1–30