Handling Tree-Structured Values in *RapidMiner*

Felix Jungermann
Technical University of Dortmund
Artificial Intelligence Group
felix.jungermann@cs.tu-dortmund.de

Abstract

Attribute value types play an important role in mostly every datamining task. Most learners, for instance, are restricted to particular value types. The usage of such learners is just possible after special forms of preprocessing. *RapidMiner* most commonly distinguishes between nominal and numerical values which are well-known to every *RapidMiner*-user. Although, covering a great fraction of attribute types being present in nowadays datamining tasks, nominal and numerical attribute values are not sufficient for every type of feature. In this work we are focusing on attribute values containing a tree-structure. We are presenting the handling and especially the possibilities to use tree-structured data for modelling. Additionally, we are introducing particular tasks which are offering tree-structured data and might benefit from using those structures for modelling. All methods presented in this paper are contained in the *Information Extraction Plugin*[1] for *RapidMiner*.

1 Introduction

Tree-structured data is very popular in natural language processing (NLP). Constituent or dependency parse trees of sentences can be used as attributes for the sentence or for parts of the sentence. Figure 1 and Figure 2 show a constituent and a dependency parse tree of the sentence *"Felix went to New York to visit the statue of liberty."*. Although, tree-structures are not restricted to NLP tasks. Machine-readable languages like SQL can be parsed and converted into a tree-structure, too. In addition, many other tasks are offering tree-structured data. The classification of *html*-documents for instance can be supported by the examination of the tree-structure given by the *html*-tags. Of

[1] available at http://www-ai.cs.tu-dortmund.de/SOFTWARE/IEPLUGIN

course, it is possible to break down tree-structures to flat features [1]. Scientific research in the field of relation extraction by [2] has shown that using the structure itself delivers more promising results. The rest of this paper is structured as follows. In Section 2 we analyze trees and shows how this structures are handled using tree kernels. In Section 3 we present the most important operators needed to process tree-structures in *RapidMiner*. Section 4 motivates the usage of these operators by presenting exemplary datamining tasks which offer tree-structured values. Finally, Section 5 concludes this paper.

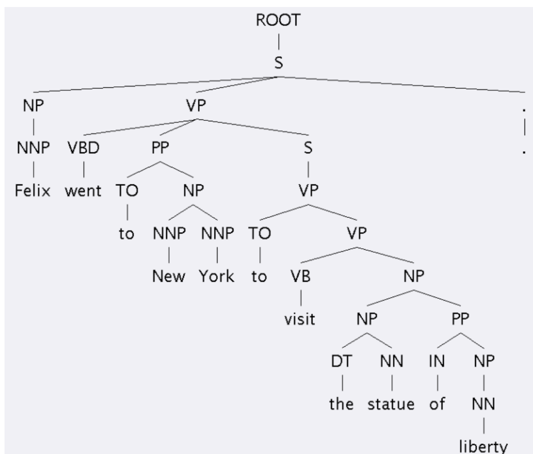

Figure 1: Constituent parse tree

Figure 2: Dependency parse tree

2 Tree-Structures

Tree-structures are special graphs $G = (V, E)$ where V is the set of vertices or nodes and E is the set of edges connecting these nodes. Additionally, trees are recursive in the sense that each node n_i of a tree t represents the root of a subtree t_i of t. It is possible to flatten a tree. Figure 3, for instance, shows the string representation of the tree shown in Figure 1. This representation could be converted into a *bag of words* (BOW) representation. In addition, the nodes could be stored as singular attributes. Documents containing a sequential word-structure can be flattened, too, by creating the BOW representation. Unfortunately, flattening will destroy inherently contained information given by the structure. For document flattening just a sequential structure is destroyed. Trees contain much more information like the depth of a node, the particular direct children, the children of those children, and so on. Flattening

a tree-structure will get very complex or/and it will cast away many information which cannot be handled well by a flat exposition. Tree kernels take this fact in account and can be used to compare two trees recursively. In addition these kernels can be used in kernel methods for data mining.

2.1 Tree-Kernels

In contrast to just using flat features, [3] were the first to use parse tree information for the classification of relations. Their work is based on the convolution kernel presented by [4] for discrete structures. To make the structural information of a tree applicable by a machine learning technique a kernel for the comparison of two trees is used. This kernel compares two trees and delivers a real-valued number which can be used by machine learning techniques.

[3] define a tree kernel as written in eq. (2), where T_1 and T_2 are trees and $I_{subtree_i}(n)$ is an indicator-function that returns 1 if the root of subtree i is at node n.

$$K(T_1, T_2) = \sum_{n_1 \in N_1} \sum_{n_2 \in N_2} \sum_{i} I_{subtree_i}(n_1) I_{subtree_i}(n_2) \quad (1)$$

$$= \sum_{n_1 \in N_1} \sum_{n_2 \in N_2} C(n_1, n_2) \quad (2)$$

$C(n_1, n_2)$ represents the number of common subtrees at node n_1 and n_2. The number of common subtrees finally represents a syntactic similarity measure and can be calculated recursively starting at the leaf nodes in $O(|N_1| * |N_2|)$. In the following we will call this kernel Quadratic Tree Kernel (QTK).

During this recursive calculation three cases are being respected:

1. If the productions at n_1 and n_2 are different, $C(n_1, n_2) = 0$
2. If the productions at n_1 and n_2 are the same and if n_1 and n_2 are preterminals, $C(n_1, n_2) = 1$
3. Else if the productions at n_1 and n_2 are the same and if n_1 and n_2 are not preterminals, $C(n_1, n_2) = \prod_j (\sigma + C(n_{1_j}, n_{2_j}))$, where $\sigma \in \{1, 0\}$ and n_{1_j} is the j-th children of n_1 (in a uniform manner for n_2).

Especially trees containing many nodes will result in large kernel-outputs which makes further processing by machine learning techniques complicated. To overcome that problem, [3] present two possibilities: Normalization and Scaling

Every kernel output can be normalized by using the following equation:

$$K'(T_1, T_2) = \frac{K(T_1, T_2)}{\sqrt[2]{K(T_1, T_1) * K(T_2, T_2)}} \tag{3}$$

Unfortunately, calculating the kernel-outcome is computationally expensive. Calculating this outcome multiple times like this is done during the normalization should be avoided. [3] established a scaling factor $0 < \lambda <= 1$ which is used in the second and third case for the recursive kernel calculation:

2. If the productions at n_1 and n_2 are the same and if n_1 and n_2 are preterminals, $C(n_1, n_2) = \lambda$
3. Else if the productions at n_1 and n_2 are the same and if n_1 and n_2 are not preterminals, $C(n_1, n_2) = \lambda * \prod_j (\sigma + C(n_{1_j}, n_{2_j}))$.

Moschitti [5, 6] is presenting an efficient calculation of eq. (2). Instead of visiting every node of both trees T_1 and T_2 he is building the pairs of nodes n_1 and n_2 for which the result of $C(n_1, n_2)$ is not 0. This node pair set N_p is defined as:

$$N_p = < n_1, n_2 > \in N_{T_1} \times N_{T_2} : p(n_1) = p(n_2), \tag{4}$$

where $p(n_i)$ delivers the production at node n_i. A production for node n_i is a representation of the node n_i itself and of its children. The order of the children has to be respected. If the productions of two nodes are not equal, $C(n_1, n_2) = 0$. The node pair set N_p contains all node pairs which are relevant for the kernel calculation. See [5, 6] for a detailed view on the mechanism collecting the relevant pairs needed to calculate the tree kernel outcome.

Although this approach may require only $O(|N_1|+|N_2|)$ it needs $|N_1|*|N_2|$ cycles in the worst case, which occurs if each of both trees just exists of one particular production (this might occur several times in each tree). The sorting of the production-lists of each tree requires $O(|N_1| * log(|N_1|))$. This sorting can be done once during preprocessing. In the following we will call this kernel Fast Tree Kernel (FTK).

The greater the amount of nodes in a tree the more complex the calculation of the kernel function. We are offering methods for pruning huge trees – refer Section 4.1 for these methods.

A recent extension is the combination of tree kernels on the one hand with linear kernels on the other hand, resulting in a so called composite kernel. The composite kernels have shown to achieve better results than with just one of these kernels [7, 8]. These techniques are not purely based on structural information (parse trees). The composite kernel combines a tree kernel and a linear kernel. [7] present a linear combination for instance:

$$K_1(R_1, R_2) = \alpha \hat{K_L}(R_1, R_2) + (1 - \alpha)\hat{K}(T_1, T_2) \tag{5}$$

where R_1 and R_2 are examples containing structural information and 'flat' features. Tuning the α-value changes the influence of the specific kernels.

3 Tree-Structures in *RapidMiner*

RapidMiner [9] most commonly is restricted to *nominal* and *numerical* attribute types. The attribute type *date* is not appropriate for our purpose and will not be mentioned in this work any further. Tree-structures can be represented as nominal value like it is shown in Figure 3. It would be a computational overhead to parse these nominal values into tree objects for every time they are needed. We developed a generic form of attribute which allows the storage of every type of *Java*-object. This generic object-attribute can be used to work with tree-structures in *RapidMiner*. Like for nominal values, the object-attribute is storing a mapping which maps numerical values to particular objects.

(ROOT (S (NP (NNP Felix)) (VP (VBD went) (PP (TO to) (NP (NNP New) (NNP York))) (S (VP (TO to) (VP (VB visit) (NP (NP (DT the) (NN statue)) (PP (IN of) (NP (NN liberty)))))))) (. .)))

Figure 3: String representation of the constituent parse tree shown in Fig. 1

3.1 Loading Trees

The operator *TreeCreatorAndProcessor* is the first operator to be used. It creates the object-attributes and its tree-structured values. Its parameters are presented in Table 1. If a tree is already given by its string representation like in Figure 3 this string simply will be converted into a tree object which finally will be stored in an object-attribute. If a sentence is contained in the attribute selected by the parameter *valueAttribute* it has to be parsed in order to create a tree. We embedded the open-source library of the *Stanford Parser*[2] into the *Information Extraction Plugin*. A various number of precompiled models are published for this parser which can be used to parse sentences for the creation of tree-structured attributes.

This operator is not very modular combining the parsing and the followed pruning of sentences. We will deskew this behavior in upcoming versions of our plugin to allow more modular setups.

[2] available at `nlp.stanford.edu/software/lex-parser.shtml`

Parameter	Description
valueAttribute	The attribute which contains the tree or the sentence to be parsed.
needParsing	Does the attribute value need to be parsed?
modelfile	The file containing a parser model
parseTreeType	The trees have to be pruned for special tasks (see Section 4.1). Select pruning type here.
FTK	Selecting this will activate the list-creation needed for the FTK.

Table 1: Parameters for *TreeCreatorAndProcessor*

3.2 Tree-Kernel SVM

We enhanced the already available *JMySVM* [10] implementation in *Rapid-Miner* by abilities to process tree-structures. We implemented the Kernel presented by [3], the Fast Tree Kernel by [5, 6] and the Composite Kernel by [7]. The operator *TreeSVM* has some additional parameters in contrast to the *Support Vector Machine* operator. These parameters are shown in Table 2.

Parameter	Description
kernel type	The kernel type to be used (*Collins and Duffy (trivial)*, *Moschitti (FTK)*, *Composite Kernel*)
CollinsDuffy Kernel Lambda	The λ-value to be used (see Section 2.1) for QTK or FTK
Composite Kernel Alpha	If the *Composite Kernel* is used the α value (see eq. (5)) can be adjusted here.
kernel type 1	The first kernel to be used for the *Composite Kernel* (just *Entity* is possible)
attribute list	The list of attributes to be used by the *Entity Kernel*
kernel type 2	The second kernel to be used for the *Composite Kernel* (QTK and FTK possible)
Collins Duffy Kernel Lambda (composite)	The λ-value to be used for QTK or FTK by the *Composite Kernel*

Table 2: Additional parameters for *TreeSVM*

3.3 Tree-Kernel Perceptron

For the task of online-learning we implemented a kernel perceptron, which offers the use of the two already presented kernels QTK and FTK. Algorithm

1 shows how the perceptron is trained.

Algorithm 1 Kernel Perceptron Algorithm

1: **procedure** TrainPerceptron($T \subset \mathcal{X} \times \mathcal{Y}$)
2: Initialize $\mathbf{w} := \mathbf{0}, M := \emptyset$
3: **for all** $(\mathbf{x}, y) \in T$ **do**
4: **if** $\sum_{y' \in Y} y' K(\mathbf{x}, M_{y'}) \neq y$ **then**
5: $M_y := M_y \cup \{\mathbf{x}\}$
6: **end if**
7: **end for**
8: **end procedure**

For every example $(\mathbf{x}, y)$ from the training-set the decision function in line 4 is performed. The possible classes Y are $\{1, -1\}$. This means that for each example $\mathbf{x}$ two kernel calculations are performed – one for the positive and one for the negative class. Each calculation is multiplied by the class y'. The calculation is performed on sets $M_{y'}$. These sets contain the already misclassified examples for the particular class y'. In contrast to perform multiple kernel calculations on all examples already been misclassified this is a more efficient type of kernel which collects all misclassified tree-structured values in one datastructure. This method saves a significant number of calculations for each prediction for practical use. We called this kernel approach *treeceptron*, and offer this approach to be used here as a kernel. Because of space limitations the exact handling of the sets $M_{y'}$ is not described here.

Parameter	**Description**
kernel type	The kernel type to be used (*CollinsDuffy*, *FastTree*, *Treeceptron*, *DAGperceptron*, *OneDAGperceptron*)
attribute	The attribute containing the tree-structures.
lambda	The λ-value to be used (see Section 2.1) for the tree kernel.
sigma	The σ-value to be used (see Section 2.1) for the tree kernel.
bootstrap	If this is selected the at each step a randomly chosen example will be selected.
stopping	After this number of iteration training will stop. Selecting -1 will make the perceptron do one run on the complete exampleset.

Table 3: Parameters for *Kernel Perceptron*

4 Particular Tasks

In this Section we are presenting two particular data mining tasks which benefit from tree-structured attribute values. We will show in an exemplary processes how these tasks can be handled in *RapidMiner*. Figure 4 shows the exemplary process structure for an experiment containing tree-structured attribute values. At first, the data is retrieved. The tree-structured data is converted into tree-objects. Finally, the performance is evaluated using a *Validation* chain. Internally, a model is trained by using the *TreeSVM* or *Kernel Perceptron* operator. If multiple classes are available, the well-known *Polynomial by Binomial Classification* operator will have to be used to encapsulate the binary machine learning methods. We make our experiments by using the *Information Extraction Plugin* for *RapidMiner* [11, 12].

4.1 Relation Extraction

Relation Extraction is well-known since the Automatic Content Extraction (ACE) conferences. The ACE conference of 2004 provided a joint task which included a task for Relation Extraction [13]. If all entities in a sentence have been found, every possible pair of two entities is combined to a relation candidate in order to find out whether there is a relation and to predict the corresponding relation type.

Every relation candidate is stored as an example in *RapidMiner*, and each example contains the parse tree of the (part of the) sentence which contains the relation candidates.

It is important that the relation candidate, the parse tree is used for, just covers a small part of the complete sentence, mostly. This allows to prune the used parse tree without loosing information about the embedded relation candidate but having the advantage of smaller complexity.

[7] inspected five types of pruning methods: The *Minimum Complete Tree* is the the smallest complete subtree containing both entities. Cutting off every node and production except the path between the entities and the nodes in between will lead to the *Path-enclosed Tree*. The *Context-Sensitive Path Tree* contains one word beside each entity, additionally. Non-terminal-nodes which just have one in- and out-arc are removed to create the *Flattened Path-enclosed Tree* out of the *Path-enclosed Tree* and the *Flattened CPT Tree* out of the *Context-Sensitive Path Tree*.

Using a composite-kernel on this task achieved the best performance compared to using just the parse tree or just the entity information [7].

Figure 4: Tree-structures handling process

4.2 Website Classification

Classifying websites is one of the most important tasks in document classification. A typical approach for document classification is the transformation of the documents into its bags of words (BOW) [14]. *Html*-documents are offering an additional tree-structure given by its *html*-tags. In this Section we will show how to use such tree-structural information in *Rapidminer*.

We used the *Syskill and Webert Web Page Ratings* dataset which is available at the UCI machine learning repository [15]. The dataset originally was used to learn user preferences. It contains websites of four domains and user ratings on the particular websites are given. We will just focus on the classification of the four categories. We parsed the websites for the construction of a tree-structured attribute value for each website. Unfortunately, the leafs of the *html*-tree contain huge text fragments in some extent. We converted every leaf which contains text into the token *'leaf'* to solve this problem. The dataset contains 341 examples of four classes. 136 examples belong to class *BioMedical*, 61 to class *Bands*, 64 to class *Goats* and finally, 70 examples belong to class *Sheep*.

Method	Accuracy	Recall	Precision	Time (in s)
Using a perceptron on one- and two-gram features	$66.4 \pm 1.5\%$	$64.1 \pm 1.7\%$	$66.0 \pm 2.1\%$	118.7
Using a perceptron on one-gram features	$62.7 \pm 1.6\%$	$61.3 \pm 1.8\%$	$62.1 \pm 2.3\%$	18.9
Using one DAG handling both classes	$67.9 \pm 1.5\%$	$64.0 \pm 1.6\%$	$68.4 \pm 2.5\%$	**1.78**
Using two DAGs	$67.9 \pm 1.4\%$	$64.1 \pm 1.6\%$	$68.6 \pm 2.4\%$	2.38
Using the treeceptron approach	$68.3 \pm 1.6\%$	$64.5 \pm 1.7\%$	$69.1 \pm 2.4\%$	55.0
Using the FTK approach	$\mathbf{68.3 \pm 1.5\%}$	$\mathbf{64.6 \pm 1.7\%}$	$\mathbf{69.2 \pm 2.4\%}$	78.9

Table 4: Results of the Perceptron approaches on SW

Table 4 contains the results of experiments on the SW dataset using a perceptron with different settings. The first two lines are experiments using the *Perceptron* operator in *RapidMiner* on flat features. We created the flat features by splitting the string representation of the trees into one- and two-grams. This preprocessing destroys the structure of the trees but allows to evaluate the gain of methods using tree-structured values. The other lines of the Table show experiments made by using the *Tree-Kernel Perceptron* operator. It becomes obvious that using tree-structured values is significantly better than just to use one-grams of the string representation of the tree-structure. By applying an ANOVA test we evaluated that using tree-structured values still is significantly better than just to use two- **and** one-grams for the experiments in cases of accuracy and precision. The time noted in Table 4 is the execution time for one ten-fold cross-validation. The more faster execution time for some of the experiments using tree-structured values is rooted in the fact that the internal data structure for storing tree-structures is more efficient (a DAG is a directed acyclic graph used for the storage of tree forests as presented by [16]). Another point which is responsible for the faster execution time might be the greater number of attributes which have to be processed in the case of flat features. The tree-structured value internally is just one attribute for each example.

5 Conclusion

We presented the possibilities to handle tree-structured values in *RapidMiner* by using the *Information Extraction Plugin.* We showed that tree-structured attribute values can be created directly out of a string representation of a tree or by using a parsing operator. Some tasks like relation extraction for instance need the original trees to be pruned which can be done by the presented operators, too. Finally, the already available operators to be used with tree-

structured values for modelling are presented and two particular tasks motivate the usage of trees in datamining tasks.

Our future work will focus on other machine learning techniques in order to allow a more flexible handling of tree-structured values.

Acknowledgements

This work has been supported by the DFG, Collaborative Research Center SFB876, project A1.

References

[1] G. Zhou, J. Su, J. Zhang, and M. Zhang, "Exploring various knowledge in relation extraction," in *Proceedings of the 43rd Annual Meeting of the ACL*, (Ann Arbor), pp. 427–434, Association for Computational Linguistics, June 2005.

[2] G. Zhou, M. Zhang, D. H. Ji, and Q. Zhu, "Tree kernel-based relation extraction with context-sensitive structured parse tree information," in *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, Association for Computational Linguistics, 2007.

[3] M. Collins and N. Duffy, "Convolution kernels for natural language," in *Proceedings of Neural Information Processing Systems, NIPS 2001*, 2001.

[4] D. Haussler, "Convolution kernels on discrete structures," tech. rep., University of California at Santa Cruz, Department of Computer Science, Santa Cruz, CA 95064, USA, July 1999.

[5] A. Moschitti, "Making tree kernels practical for natural language learning," 2006.

[6] A. Moschitti, "Efficient convolution kernels for dependency and constituent syntactic trees," in *Procs. ECML* (J. Fuernkranz, T. Scheffer, and M. Spiliopoulou, eds.), pp. 318 – 329, Springer, 2006.

[7] M. Zhang, J. Zhang, J. Su, and G. Zhou, "A composite kernel to extract relations between entities with both flat and structured features," in *Proceedings 44th Annual Meeting of ACL*, pp. 825–832, 2006.

[8] M. Zhang, W. Che, A. T. Aw, C. L. Tan, G. Zhou, T. Liu, and S. Li, "A grammar-driven convolution tree kernel for semantic role classification," in *Proceedings of the 45th Annual Meeting of the Association of*

Computational Linguistics, pp. 200–207, Association for Computational Linguistics, 2007.

[9] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler, "YALE: Rapid Prototyping for Complex Data Mining Tasks," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2006)* (T. Eliassi-Rad, L. H. Ungar, M. Craven, and D. Gunopulos, eds.), (New York, USA), pp. 935–940, ACM Press, 2006.

[10] S. Rueping, *mySVM Manual.* Universitaet Dortmund, Lehrstuhl Informatik VIII, 2000. http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM/.

[11] F. Jungermann, "Information extraction with rapidminer," in *Proceedings of the GSCL Symposium 'Sprachtechnologie und eHumanities'* (W. Hoeppner, ed.), pp. 50–61, Universität Duisburg-Essen, Abteilung für Informatik und Angewandte Kognitionswissenschaft Fakultät für Ingenieurwissenschaften, 2009.

[12] F. Jungermann, "An information extraction plugin for rapidminer 5," in *Proceedings of the RapidMiner Community Meeting And Conference (RCOMM 2010)*, pp. 67 – 72, 2010.

[13] Linguistic Data Consortium, *The ACE 2004 Evaluation Plan*, 2004.

[14] T. Joachims, "Text categorization with support vector machines: Learning with many relevant features," in *Proceedings of the European Conference on Machine Learning* (C. Nédellec and C. Rouveirol, eds.), (Berlin), pp. 137 – 142, Springer, 1998.

[15] A. Frank and A. Asuncion, "UCI machine learning repository," 2011.

[16] F. Aiolli, G. Da San Martino, A. Sperduti, and A. Moschitti, "Efficient kernel-based learning for trees," in *Computational Intelligence and Data Mining, 2007. CIDM 2007. IEEE Symposium on*, pp. 308 –315, 2007.