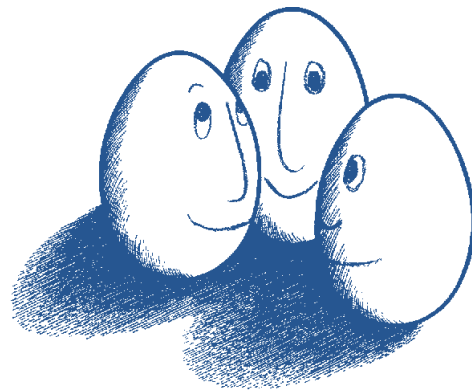


Diploma Thesis

Integration of WebLicht Services for Fast Structural Kernel Generations and Feature Visualization in RapidMiner

Marcel Fitzner

February 2015



Supervisors:

Prof. Dr. Katharina Morik

Dipl.-Inform. Christian Pölitz

Technical University Dortmund

Faculty of Computer Science

Chair of Artificial Intelligence (VIII)

<http://www-ai.cs.uni-dortmund.de/>

Table of content

Chapter 1	Introduction	8
Chapter 2	Annotation and Extraction of Linguistic Features	10
2.1	Overview of linguistic tools for NLP	10
2.2	Introduction to linguistic tools	12
2.2.1	Tokeniser	12
2.2.2	Part-Of-Speech-Taggers	12
2.2.3	Stemmers and lemmatisers	13
2.2.4	Parsers	14
2.2.5	Constituency parsers	14
2.2.6	Dependency parsers	15
2.2.7	Word sense disambiguators	17
2.2.8	Named Entity recognizers	17
2.3	Natural language processing toolkits	18
2.4	Web-Based Linguistic Chaining Tool (WebLicht)	20
2.4.1	The WebLicht services for natural language processing	20
2.4.2	The web environment of WebLicht	21
2.4.3	Communication with WebLicht services	22
2.4.4	Preparing text corpora for WebLicht services	22
2.4.5	Testing the accessibility of WebLicht services	23
2.5	Annotating linguistic features in RapidMiner	25
2.5.1	Flexible tool chain concept for the 'WebLicht Feature Annotator'	25
2.5.2	XML Configuration of the WebLicht tool chain	27
2.5.3	Implementing the WebLicht connector	28
2.5.4	Compatible WebLicht services in the tool chain	29
2.6	Extraction of linguistic features in RapidMiner	31
2.6.1	Motivation for implementing a parser for annotated text corpora	31
2.6.2	Parsing linguistic features from annotated text corpora	31
2.6.3	Feature extraction from XML sections in the TCF document	33
2.7	Discussing linguistic features in the context of a hypothetical task	36
2.7.1	Definition of a metaphor	36
2.7.2	Discussing linguistic features for pattern detection	36
Chapter 3	Feature Visualization	40
3.1	Terminology of graphs and trees	40
3.2	Modeling relational structures of parse trees	41
3.3	Tree drawing	42
3.3.1	Drawing conventions	42
3.3.2	Aesthetics and constraints of a tree drawing	43
3.4	Drawing algorithms for parse trees	45
3.4.1	The "Layered-Tree-Draw" Algorithm	46
3.4.2	The "Reingold & Tilford" Algorithm	48
3.5	Visualization of structural features in RapidMiner	50

Chapter 4	Machine Learning in Text Corpora	52
4.1	Kernel Methods (KMs)	53
4.2	The Support Vector Machine (SVM)	56
4.2.1	The linear separable case.....	57
4.2.2	Karush-Kuhn-Tucker conditions	59
4.2.3	The non-separable case	60
4.3	The String (Subsequence) Kernel	62
4.4	Bag of Words Kernel / n-gram Kernel	63
4.5	The Spectrum Kernel.....	64
4.6	The Tree Kernel.....	64
4.7	Fast Kernels for String and Tree Matching	66
4.7.1	The suffix tree	67
4.7.2	Matching statistics	69
4.7.3	Efficient Kernel computation.....	71
4.7.4	Weight functions.....	73
4.8	The 'Fast String Kernel' operator for RapidMiner.....	74
Chapter 5	Experiments	77
5.1	Experiment I: "Tranches".....	77
5.1.1	Acquiring labeled data.....	77
5.1.2	Training phase.....	80
5.1.3	Testing phase.....	82
5.2	Experiment II: "Literature types".....	84
5.2.1	Acquiring texts from different periods.....	84
5.2.2	Training phase.....	85
5.2.3	Testing phase.....	87
5.3	Experiment III: "Bild vs. Spiegel".....	88
5.3.1	Acquiring sentences from online articles	88
5.3.2	Training phase & Testing phase	89
5.3.3	Testing phase.....	90
5.4	Benchmark test of the 'Fast String Kernel' operator.....	91
Chapter 6	Summary and Outlook.....	94
Appendix	95
A.1	RapidMiner operator 'WebLicht Feature Annotator'	95
A.1.1	Installation and usage in RapidMiner.....	95
A.1.2	Description of parameters.....	95
A.1.3	XML scheme definition for the XML configuration of the tool chain	97
A.1.4	XML configuration for storing available WebLicht services	99
A.1.5	Class diagram of the 'WebLicht Feature Annotator'.....	103
A.2	RapidMiner operator 'WebLicht TCF to ExampleSet'.....	104
A.2.1	Installation and usage in RapidMiner.....	104
A.2.2	Description of parameters.....	104
A.3	RapidMiner operator 'Visualize and Label Parse Trees'	105
A.3.1	Installation and usage in RapidMiner.....	105

A.3.2	Description of parameters	106
A.3.3	Class diagram of the visualization framework for drawing parse trees	107
A.4	RapidMiner operator 'Fast String Kernel'	108
A.4.1	Installation and usage in RapidMiner	108
A.4.2	Description of parameters	109
References	110
Literature	110
URLs	112
List of Figures	113
List of Tables	115

"The whole of life is just like watching a film. Only it's as though you always get in ten minutes after the big picture has started, and no-one will tell you the plot, so you have to work it out all yourself from the clues." *Terry Pratchett*

Acknowledgements:

First of all, my deep thanks goes to my supervisors at the Chair VIII for Artificial Intelligence at the Faculty of Informatics at the Technical University of Dortmund, Mrs. Prof. Dr. Katharina Morik and Dipl.-Inf. Christian Pölitz for the support and direction during my research in the field of computational linguistics and machine learning. I thank Katharina for her comments and ideas about the implemented visualization module, and Christian who provided insights and expertise about learning on text corpora in RapidMiner.

My thanks also go to Thomas Bartz and many other linguists who have spent a lot of time and energy to prepare the many different text corpora that were provided to me for the experiments.

I am also immensely grateful to Sebastian Buschjäger, Sebastian Gerard, Lukas Pfahler, and Jörg Nitschke for their comments on an earlier version of the manuscript, although any errors are my own and should not tarnish the reputations of my esteemed fellow students. Last but not least, I thank my wife Jia for her daily support, love, and the many delicious Chinese meals she cooked.

Chapter 1

Introduction

Computational linguistics is an interdisciplinary research field in which natural language is studied from a computational perspective. It focuses on the development of models for various kinds of linguistic phenomena in order to enable machines to recognize, process, represent, and produce natural language in both spoken and written form.

In the context of *text corpus based learning*, experts in linguistic research often have a distinct question in mind to which computer scientists attempt to provide an answer with the help of machine learning methods. The term 'text corpus' refers to a set of documents where each document may consist of several sentences but often contains only a single sentence. In the following course of this work, the terms 'document' and 'sentence' are used synonymously. Typical tasks could be the association of sentences to specific topics or text corpora, or the distinction of sentences if a specific grammatical phenomenon is present or not.

In this work text classification tasks are considered where sentences may be distinguished according to specific expressions or some type of linguistic feature like tokens, parts-of-speech, lemmas, dependencies or grammatical constituents. Establishing an overall routine that classifies documents of text corpora by means of machine learning methods requires us to first obtain and prepare these features. Figure 1-1 presents a pipeline that consists of different processing steps where each step is covered by a single chapter:



Figure 1-1: A processing pipeline concept combining feature preparation steps and machine learning methods in order to perform a text classification.

In order to perform a machine learning, only the sentences of an acquired text corpus (step 1) could be used, but features like tokens, parts-of-speech, lemmas, dependencies or constituents reveal patterns that may prove relevant for a text classification task. For instance, such patterns can be characterized by a specific word usage, parts-of-speech sequences or relational structures like dependencies or constituents.

In Chapter 2 features and linguistic tools are investigated, followed by a comprehensive presentation of methods that perform a feature processing in various ways (step 2). More precisely, Section 2.1 provides a hierarchy of various linguistic features. A common technique to obtain these features is by means of *annotation tools* that are often designed to annotate a document with a single feature type. In Section 2.2 linguistic tools of the most relevant features are introduced. Furthermore, in Section 2.3 a list of available annotation toolkits is compiled with regard to finding the most suitable for the integration into a *feature annotation tool* for RapidMiner.

Then, Section 2.4 provides a detailed introduction to the chosen toolkit "WebLicht". As the last step in feature preparation, Section 2.6 presents the process for *feature extraction* that is implemented in a RapidMiner operator.

Chapter 3 deals with the implementation of a *visualization module* (step 3) to display *parse trees* that encode the dependencies or grammatical constituents of the sentences. Section 3.3 introduces conventions in order to formalize the optimization problem of obtaining layouts that produce tidy trees spanning a minimal width. Section 3.4 then, presents algorithms to construct tree layouts in *linear time*. Additionally, the implemented visualization module offers the option to manually label sentences of a text corpus which can be used in supervised machine learning (Appendix A.3).

In Chapter 4 *kernel methods* (KMs) are introduced that allow an efficient detection of patterns in a given set of linguistic features (Section 4.1). Conceptually, KMs perform a mapping of features to a *feature space* (step 4). In this space non-linear relations become linear separable which allow the integration of a machine learning method (step 5). Here, the prominent *support vector machine* (SVM) is employed which is presented in Section 4.2. The Sections 4.3-4.7 briefly introduce KMs that are specifically designed for comparing strings and trees. Since classical string kernels are computationally slow, the '*Fast Kernel (Method) for String and Tree Matching*' [Vishwanathan & Smola] is presented where the kernel computation performs in *linear time* (Section 4.7). In addition to the implemented operator for RapidMiner (Section 4.7.3), various weight functions are provided in order to differently emphasize arbitrary matching substrings (Section 4.7.4). Section 4.8 addresses the problem of a high memory consumption during the kernel computation and provide an effective solution by implementing different caching mechanisms. Finally, the runtime performance of the 'Fast String Kernel' operator is measured in a benchmark test and the results are outsourced in Section 5.4.

Chapter 5 presents three machine learning experiments that were run on different corpora in order to investigate which combination of feature type and weight function is the most suitable to achieve the highest possible separability of two different text corpora with regard to each specific text classification task. At the same time, the use of the annotation and extraction operators is shown, and where applicable the visualization operator which additionally allows to assign labels to the annotated sentence.

In the Summary and Outlook a brief review provides an overview of the established linguistic processing capabilities and properties of the learning framework. Further, the advantages of own contributions are outlined while pointing out opportunities for further research.

Chapter 2

Annotation and Extraction of Linguistic Features

This chapter investigates various linguistic features and deals with the preparation of these features in the context of a processing pipeline with the intention to perform a text classification. The preparation includes the annotation of text corpora with features and the extraction of these so that they can be forwarded to a machine learning method. Linguistic features are basically of so called *flat* or *structured* type. A *flat feature* is usually of nominal type. In natural language processing (NLP) these are given by linguistic units like tokens, lemmas or part-of-speech (PoS) tags. *Structured features* refer to data that encodes the representation of a tree containing all the *structural relations* within a linguistic unit like a sentence.

Furthermore, flat features can be incorporated into structural features like *bag-of-terms* or *n-grams*. Here, the bag-of-terms is a vector that contains frequencies of tokens or the corresponding part-of-speech tags. Another structural feature is the *n-gram* which is comprised of a sequences of units (like characters or words) whereas all sequences have the same number of units.

Section 2.1 provides a hierarchical overview of linguistic tools on different levels of analysis. Further, Section 2.2 introduces common linguistic tools used for natural language processing (NLP) tasks. In order to establish NLP for RapidMiner, Section 2.3 presents a list of available toolkits and libraries where their suitability is evaluated with regard to processing English and German text corpora. Then, Section 2.4 introduces 'WebLicht', which is a service oriented architecture (SOA) that allows us to directly communicate with various services to enrich a text corpus with desired features.

A particular problem is to obtain structural features that build upon basic features. Therefore, Section 2.5 presents a flexible tool chain that is implemented into a feature annotation tool and further describes the configuration of WebLicht services. Section 2.6 describes the feature extraction process from annotated corpora and points out important properties of the implemented operator in RapidMiner.

Finally, in Section 2.7 the different types of linguistic features are discussed against the background of a hypothetical task of detecting metaphors in a text corpus.

2.1 Overview of linguistic tools for NLP

Many linguistic tools enrich a linguistic resource with annotations either because a higher accessibility is required or because this resource needs to be passed for further processing. Depending on the task, specific tools annotate paragraphs, sentence parts, phrases or single words with additional data. The data can contain information about a word sense (semantics),

part-of-speech (syntax), references, or any other unit of the linguistic resource that seems useful for the analysis task. Other annotated information may describe the phonetics of single words or consist of markers for a proper identification of *named entities* (e.g. persons, organizations).

Generally, linguistic tools can be described as programs that analyze or process linguistic units like tokens, phrases or sentences. Tokens are not only the words of the text, but may also refer to numbers, named entities or punctuation characters.

Tools that play a major role with regard to a text analysis (highlighted in Table 2-1) are *tokenisers* (Section 2.2.1) that segment sentences into sets of tokens, *lemmatisers* that determine to each word the corresponding lemma (2.2.3), and *part-of-speech (PoS) taggers* (Section 2.2.2) that automatically identify the parts-of-speech and tag the tokens accordingly. Furthermore, *named entity recognizers* (Section 2.2.8) which are often contained in tokenisers, and *word sense disambiguators* are important tools, as well.

	Discipline	Units & Categories	Tools
Higher levels of analysis ↑	Pragmatics, Discourse theory, Rhetoric, Speech act theory	discourse types, genres, classes of speech act, emotions	emotion analyzers, <i>metaphor analyzers</i> , rhetorical coherency analyzers, dependency analyzers, named entity recognizers
	Semantics		word sense disambiguators , semantic role analyzers, coreference and anaphora tools
Lower levels of analysis ↓	Syntax	sentences, phrases, words	constituency parsers, dependency parsers , chunkers
	Morphology and Lexical analysis	words, prefixes and suffixes, singular and plural, conjugations, declensions	stemmers, lemmatisers, tokeniser, part-of-speech taggers
	Phonetics and Phonology	sounds, phonemes, syllables, Intonational categories	speech recognition, spectrograms / sonograms

Table 2-1: Disciplines of linguistic tools distributed along different levels of analysis

Tools of a higher complexity usually depend on tools with a lower one. As shown in Table 2-1, the linguistic tools can be divided into different levels of analysis [hierarchy]. As an example, a syntax analyzer like a parser requires sentences to be clearly separated from each other, words to be clearly delineated by a *tokeniser*, and a *part-of-speech tagger* to have performed first. *Constituency* and *dependency parsers* (Section 2.2.4) that analyse the syntax of a sentence depend on the output of linguistic tool that extract *flat features* from the same sentence beforehand.

2.2 Introduction to linguistic tools

The following subsections describe common, linguistic tools that are used for the annotation of text corpora with relevant features.

2.2.1 Tokeniser

A computational analysis of a text corpus normally starts with the segmentation of the text into a set of individual words, also known as tokenization. Additionally, a sentence-splitting tool is often used in combination with the tokenization. An easy way to tokenize a text corpus into its tokens is by simply decomposing the text along its whitespace characters followed by punctuation marks. In alphabetic texts additional challenges have to be met as there exist many linguistically anomalous elements like numbers, abbreviations, named entities, punctuation (e.g. used in URLs) and many more. Given those difficulties it is often more practical to consider tokens instead of words when segmenting a text, since a token encompasses these anomalous elements [tokeniser].

Most of the tokenisers need to be trained for a large set of idiosyncrasies in a given language. For example it is being expected from a tokeniser to recognize and decompose the entities in a compound word like the famous German word *"Donaudampfschiffahrtselektrizitätenhauptbetriebswerkbauunterbeamtengesellschaft"* or linguistically similar compounds like *"low-budget"* or *"first-class"*. Short phrases like idioms that are composed of multiple words and separated by spaces like *"im Großen und Ganzen"* in German or *"pain in the neck"* are best to be treated as a single term. Furthermore, a tokeniser should also properly recognize terms of the same meaning but that can be written in different ways like *"egg beater"*, *"egg-beater"* and *"eggbeater"*.

After performing the segmentation process, a tokeniser delivers linguistic features as a set of ideally all properly identified tokens. This set of features is usually known under the term *"bag of words"* (for the definition see Section 4.4).

Some tokenisers additionally return a list of sentences in which each token corresponds to a specific sequence of characters in the text - according to the rules and idiosyncrasies of the given language.

2.2.2 Part-Of-Speech-Taggers

The task of a part-of-speech (PoS) tagger is to classify the syntax of words in a given text. However, a tokenization has to be performed beforehand. While considering the context of a word a PoS tagger chooses the parts-of-speech tags from a specific set of parts-of-speech, usually referred to as *tagset*. For German texts a frequently used tagset is the "Stuttgart-Tübingen Tagset" (STTS) [Schiller et al.], and for English texts the Penn Treebank Tagset (PTTS) [Santorini] and the CLAWS Tagset [Garside] are frequently used.

Ambiguity of parts-of-speech tags for a given word poses a frequent problem for a tagger, as shown in the following sentence:

Der	Lehrer	trinkt	einen	Kaffee	.
ART	NN	VVFIN	VVIN VVFIN ART	NN	\$.

Example 2-1: Ambiguous case with possible PoS-tags with regard to the word "einen".
The given sentence translates to "The teacher drinks a coffee".

Without considering the context, the German word "einen" has two possible meanings, either being used as an article in masculine form for accompanying a noun or it is used as a verb with the meaning to "unify" something. Hence the available tags are "ART", "VVINF" (infinitive verb, full) and "VVFIN" (finite verb, full).

Various supervised learning methods have been established to train a PoS tagger to choose the correct tags in case of ambiguity, for example by means of hidden markov models (HMM) [Charniak 1997] and decision trees [Schmid]. Another approach uses a simple, rule-based PoS tagger [Brill]. During training, tags are learned from specific corpora (mostly of the same genre like corpora of newspapers) whose parts-of-speech tags have been manually annotated. For the actual tagging phase it is recommended that the tokens of a text match those tokens to which the tagger has been trained to recognize. Respectively it is important that the tokeniser employed in the preprocessing is used for the training phase as well.

In an unsupervised setting no previously defined tagset can be chosen, where no training data is available and hence no error signal can be computed to evaluate a potential solution. Instead a new tagset is generated during the tagging phase by means of stochastic methods.

While in a naive approach the frequencies of occurring PoS-tags are simply learned from a given training corpus, HMM-based or decision tree based taggers perform far better by considering the context of a tag like the preceding and following tag [Brill]. Following this approach, learning methods of performant PoS taggers make use of sequences of PoS-tags. In this context, a sequence of n linguistic units (like words, parts-of-speech or characters) is referred to as an n -gram. Usually learning methods make use of bi- or trigrams. In the given Example 2-1 the word sequence "trinkt einen" has the bi-grams VVFIN-ART, VVFIN-VVIN and VVFIN-VVFIN. Training from text corpora, the sequence VVFIN-ART likely has the highest probability. Conclusively, a PoS tagger assigns the tag ART to the word "einen".

2.2.3 Stemmers and lemmatisers

Both stemming and lemmatization aim to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form [stemming]. Since the inflectional morphology of most European languages is indicated by suffixes, a *stemmer* is a program that usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational affixes. Stemmers were developed originally to improve information retrieval and are usually very simple programs that

use a catalog of regular expressions to simplify the word-forms found in digital texts. They are linguistically not very sophisticated, and miss many kinds of morphological variations.

Whenever possible, a lemmatiser is preferred to a stemmer. Usually a combination of a lexicon and a set of rules is being used in order to remove inflectional endings and then to return the base (or dictionary) form of a word, which is known as the *lemma*. A lemmatiser using this approach can determine the lemmas to each annotated token in the input text corpus.

2.2.4 Parsers

A parser is a tool that performs syntactic analysis of natural language either in an automated or manual way. Although parsing of natural language superficially resembles parsing in computer science, the former one operates in a very different way, since the diversity and complexity of human language still exceeds those parsers that make use of finite-state grammars and recognition rules.

Far more accurate and robust parsers today incorporate statistical principles to some degree and often these parsers have been trained from manually parsed texts using machine learning techniques, which has been done analogously successfully for the part-of-speech tagger as described in Section 2.2.2. Many common parsers like the Stanford constituency parser, the Stanford phrase structure parser or the Berkeley parser work with a PoS-tagger as a preprocessor. In this context, it is important that the tagset of the integrated PoS-tagger has to match the tagset that is expected from the parser.

Parsers usually require tokenized input text, and they output these tokens wrapped into a structural form. The structured data is known as a *parse tree*, which encodes different syntactic connections between parts of the sentence.

Most parsers implement the syntactical analysis of the two major categories *dependency grammars* and *constituency grammars*. Both types are introduced in the next two sections. Less frequently used parsers combine the results of both analysis categories into a *hybrid* form.

2.2.5 Constituency parsers

In the analysis of *constituency grammar* the relation of constituents derive from an initial binary division by splitting the clause into a subject noun phrase (NP) and a predicate verb phrase (VP). Subclauses are then iteratively decomposed up to the smallest constituents according to a given constituency grammar.

The following Figure 2-1 presents a constituency parse tree of an exemplary sentence in German. Before feeding the sentence to a constituency parser the sentence was tokenized and the parts-of-speech were identified by a PoS-tagger of the NLP project. The parser used tags from the 'Tiger Treebank Tagset' to annotate the nodes in the tree [Tiger]:

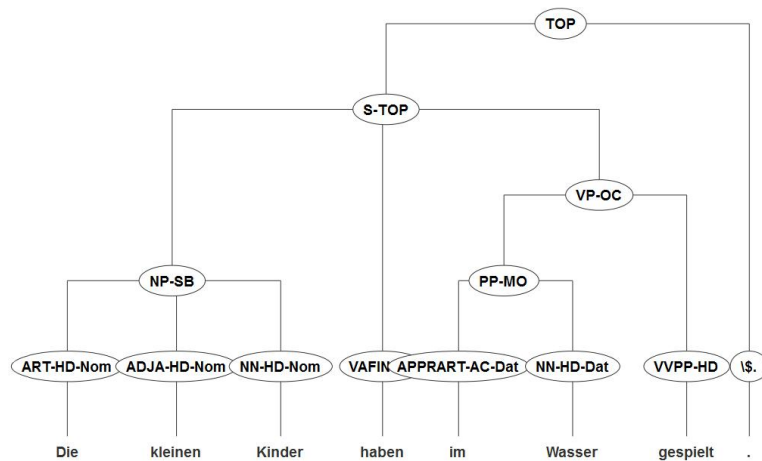


Figure 2-1: The result of a constituency parser for a German sentence.

In the case of structural ambiguous sentences not every constituent parser detects the ambiguity and usually delivers only one interpretations of the parsed sentence.

In the exemplary sentence "Visiting relatives can be dangerous" the constituency parser of the 'OpenNLP project' (left side of Figure 2-2) considers "Visiting relatives" as a noun phrase (NP) with "relatives" as the head while the 'Stanford Core NLP' parser (right side of Figure 2-2) basically treats "Visiting relatives" as a verb phrase (VP) with "Visiting" as its head:

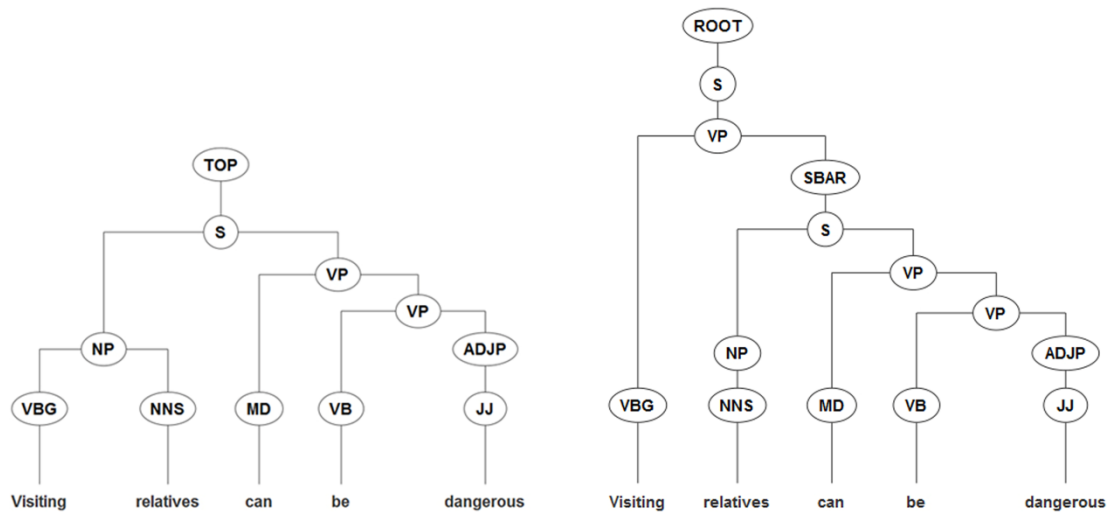


Figure 2-2: Different parsing results of the constituent parser of the NLP project and the Stanford Core NLP parser

2.2.6 Dependency parsers

The analysis of *dependency grammar* considers dependency relations between single words of a given sentence, while for a single word multiple connections to other ones can exist [Neumann]. The principal idea for syntactic connections is to choose the verb as the root of all clause structures. Tokens can then iteratively be connected with a parent node where - according to a given dependency grammar, the PoS-tag of each token is subordinated in a hierarchy of word categories. As an example, Figure 2-3 presents a German sentence in a

dependency tree, with the verb "haben" (which translates to "to have") as the root of the hierarchy of dependency relations. The edges carry PoS-tags from the 'STTS' tagset [Tiger]:

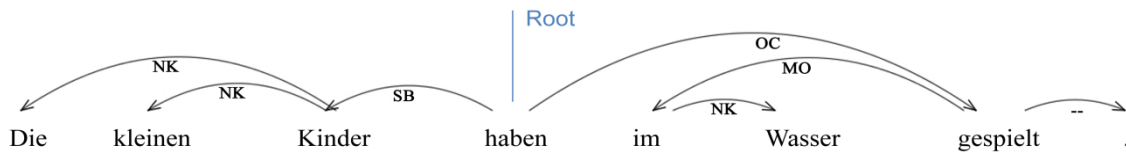


Figure 2-3: Result of the Stuttgart Dependency Parser for an German example sentence.

The above parse tree is only one way to represent dependencies, the following notation schemes illustrate other common conventions [DepConst]:

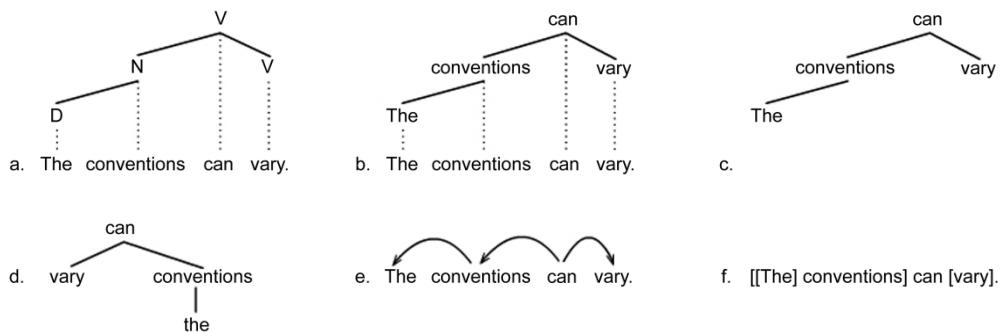


Figure 2-4: Different conventions to draw dependency trees

Convention (f.) in Figure 2-4, also referred to as *bracket notation*, is especially useful since it can practically encode a parse tree to a string which can then be forwarded to a consecutive tool. Alternatively, the bracket notation can integrate both PoS-tags and tokens regarding the above sentence "The conventions can vary":

$$\left[V \text{ can} \left[N \text{ conventions} \left[D \text{ The} \right] \right] \left[V \text{ vary} \right] \right]$$

Example 2-2: A parse tree represented in bracket notation

Representing the parsed dependency grammar as a tree is a *one-to-one relation*, since every element in the sentence corresponds to exactly one node in the tree structure. The result of this correspondence is that dependency grammars are word grammars, as shown in Figure 2-5:



Figure 2-5: Difference between a dependency and a constituency tree

In the dependency tree on the left two words are represented with two nodes, whereas the constituency tree on the right contains three nodes. Constituency trees require the number of nodes to exceed the number of elements in a sentence of at least by one [depVsConst].

2.2.7 Word sense disambiguators

A word sense disambiguator is a tool that is specialized in the automatic identification of the correct meaning of a word (or sense) in a text. Such tools usually use a combination of digitized dictionaries that contain a database of words and their possible meanings, and information about the context in which the given words is likely to have a particular meaning.

As an example, let us consider the words "white" and "snow". A disambiguation tool can associate a set of attributes that describe each of the single words, but the words in the order "Snow white" most likely has a different meaning. The input for word sense disambiguation tools are usually tokenized text, although PoS-tags and even parsing may be required before disambiguation.

2.2.8 Named Entity recognizers

The linguistic phenomenon of a named entity is the generalization of the idea of a proper noun. Examples for named entities refer to places, brand names, non-generic things, people, and sometimes highly subject-specific terms. Named entity recognition plays an important role in information retrieval, machine translation or in topic identification.

Basically, there are no limits that may restrict where named entities can be derived from. Named entities can have frequent usage in texts, and are usually not listed in common dictionaries. The detection of named entities is based on rules, statistical methods, machine learning algorithms, or a combination of these methods.

For example, while analyzing sequences of two or more words, a simple rule may check if these words are written with capital letters. Another possibility to setup named entity recognition is often done by integrating databases that contain large lists of named entities.

2.3 Natural language processing toolkits

In order to establish preprocessing routines that extract various linguistic features, a manageable selection of frameworks is examined that have been developed for natural language processing (NLP) tasks. The complete list of NLP toolkits is too comprehensive in order to be listed here [OutlineNLP], and numerous highly specialized tools perform all kinds of tasks on a wide range [tasksNLP].

For our purposes, the list is restricted to toolkits that are offered as libraries or services, and whose processing capabilities fall into consideration for the implementation of a processing operator in RapidMiner. Tools that come into consideration need to be capable of detecting and annotating text corpora with sentences, tokens, lemmas, part-of-speech tags, named entities, and parsing constituents and dependencies. The following Table 2-2 lists toolkits or libraries that are evaluated on a closer inspection:

Toolkit [URL]	Creator	License	Processing capabilities	Supported language
OpenNLP [TK_OpenNLP]	Apache Software Foundation	Apache License 2.0	sentence segmentation, tokenization, lemmatizing, PoS-tagging, NER, and more	English and others; German: sentence segmentation, tokenizing and PoS tagging only
Stanford CoreNLP [TK_Stanford]	The Stanford NLP Group	GNU Public License v3	sentence segmentation, tokenization, lemmatizing, PoS tagging, constituency parsing, NER, and more	English, Spanish & Chinese; German: PoS tagging, NER, parsing only
Natural Language Toolkit [TK_NLTK]	Team NLTK	Apache License 2.0	n-gram, PoS tagging, tokenization, NER	English, Arabic
LinguaStream [TK_Lingua]	Computer Research Group "GREYC"	Free for research	sentence segmentation, tokenization, PoS tagging, statistical tools	English, French
Mate Tools [TK_Mate]	IMS - Institute for Computational Language Processing	GNU Public License v3	lemmatizing, part-of-speech tagging, morphological tagging, dependency parsing, and semantic role labeling	English, German
MontyLingua [TK_Monty]	MIT	Free for research	tokenization, lemmatizing, PoS tagging, parsing	English
WebLicht [WebLicht]	SfS - University of Tübingen	Free for research	sentence segmentation, tokenization, lemmatizing, PoS tagging, NER, and many more	Various languages

Table 2-2: A selected list of available toolkits for different NLP tasks and languages

All listed toolkits are offered with a license that make them potential candidates for an integration into the implementation of a RapidMiner operator. Unfortunately, not many toolkits offer NLP for German language, hence only toolkits are considered that fully or partially support German:

- The '*OpenNLP*' toolkit offers basic processing (sentence segmentation, tokenizing, PoS tagging), but lacks constituency or dependency parsing.
- The '*Stanford CoreNLP*' toolkit offers PoS tagging, named entity recognition and constituency parsing trained on the Negra corpus [NEGRA]. On a side note, the NEGRA corpus is a large set of tokens and sentences of German newspaper text, taken from the '*Frankfurter Rundschau*'. Still, using only one distinct training corpus like NEGRA may not suffice to represent the German language as editors of the *Frankfurter Rundschau* are probably not using terms, phrases and sayings that are common to people in other cultural regions of Germany. Furthermore, the 'Stanford CoreNLP' does not contain tools that extract German tokens or their lemmas, and offers no dependency parser which we want to integrate into a RapidMiner, as well.
- The toolkit '*Mate Tools*' contains basic linguistic processing as well as parsing and semantic role labeling. The tools provide processing of linguistic units like lemmas and PoS-tags, but no tokenization.
- The language processing environment '*WebLicht*' is a service oriented architecture (SOA) that has been established by partners of the *KobRA* project at the University of Tübingen [Hinrichs et al., WebLicht]. The term WebLicht refers to **Web-Based Linguistic Chaining Tool**. Since WebLicht is offered as a SOA, no libraries need to be integrated into a programming code, but instead all the services can be accessed remotely. The repertoire of services amounts a comprehensive list of processing tools for both English and German (among other languages). Furthermore, the environment allows a chaining of tools so that they consecutively add linguistic features to a text corpus, as presented in Section 2.4.2.

For the integration in a feature preprocessing pipeline in RapidMiner, the choice fell on 'WebLicht' as the advantages above make this SOA an applicable and highly interesting candidate for the implementation of a processing operator in RapidMiner. The next section presents a selection of tools that are available in the large repertoire of NLP tools provided by WebLicht.

2.4 Web-Based Linguistic Chaining Tool (WebLicht)

2.4.1 The WebLicht services for natural language processing

WebLicht offers a large repertoire of tools, many of them process texts in English and German language. The Table 2-3 depicts a list of interesting tools that come into question for a language processing in RapidMiner:

Tool name	Developed by	Supported Languages	Extracted features
Tokeniser	ASV: Uni Leipzig	de	sentences, tokens
Tokeniser	IMS: Uni Stuttgart	en, de, cs, hu, sl, fr, it	sentences, tokens
Tokeniser - OpenNLP Project	SfS: Uni Tübingen	en, de	tokens
Tokeniser/Sentences - OpenNLP Project	SfS: Uni Tübingen	en, de	sentences, tokens
Tokeniser and Sentence Splitter	BBAW: Berlin	de	sentences, tokens
Part-of-Speech-Tagger	BBAW: Berlin	de	PoS-tags
POS Tagger - OpenNLP Project	SfS: Uni Tübingen	en, de	PoS-tags
RFTagger	IMS: Uni Stuttgart	de, cs, hu, sl	PoS-tags
TreeTagger	IMS: Uni Stuttgart	en, de, fr, it	lemmas, PoS-tags
Stanford Core NLP	SfS: Uni Tübingen	en	sentences, tokens, lemmas, PoS-tags, named entities, const. parsing
Berkeley Parser - Berkeley NLP	SfS: Uni Tübingen	de	Parsing
Constituent Parser	IMS: Uni Stuttgart	en, de	const. parsing
Constituent Parser - Open NLP Project	SfS: Uni Tübingen	en	const. parsing, PoS-tags
Stanford Dependency Parser	SfS: Uni Tübingen	en	dep. parsing, PoS-tags
Stanford Phrase Structure Parser	SfS: Uni Tübingen	en, de	parsing , PoS-tags
Stuttgart Dependency Parser	IMS: Uni Stuttgart	de	dep. parsing, lemmas, PoS-tags
German Named Entity Recognizer	SfS: Uni Tübingen	de	named entities
Open NLP Named Entity Recognizer	SfS: Uni Tübingen	en, es	named entities
Person Named Recognizer	BBAW: Berlin	de	named entities

Table 2-3: A list of relevant services of WebLicht that perform NLP tasks for English and German text corpora.

The repertoire of the SOA WebLicht provides all important tools for annotating text corpora with the most common linguistic features. More importantly, all of these tools are able to process English and German texts.

2.4.2 The web environment of WebLicht

WebLicht comes along with an interactive web interface that can be accessed by authorized research users¹.

This section briefly describes how linguistic tools can be chained together in order to perform NLP tasks on texts: First, the user uploads or directly enters a text via a form. In order to assemble a processing chain, the user can drag tools into an appropriate area of the web interface. While doing so, the WebLicht interface interactively adapts the palette of the remaining applicable tools according to the last appended tool.

When starting the execution of a "constructed" tool chain the given text is first converted into an XML2 structure whose scheme follows a specific *text corpus format (TCF)* [Heid et al.].

Depending on the "stage" of the tool chain, each processing tool enriches the text corpus with additional linguistic features. Finally, the finished annotated text and the extracted features can be viewed in a separate web interface or directly downloaded as a TCF file.

The screenshot in Figure 2-6 shows an exemplary processing tool chain in the web environment of WebLicht. It consists of a text loader, a text to TCF converter, a tokenizer, a part-of-speech tagger, and a constituent parser.

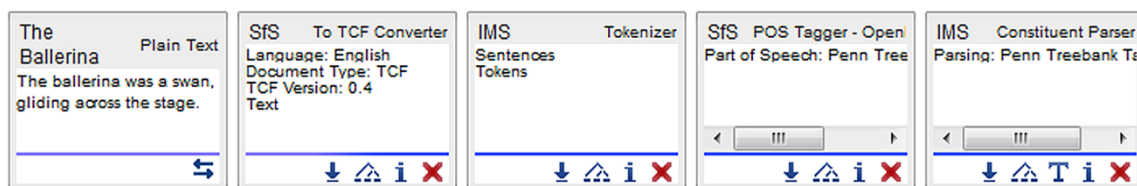


Figure 2-6: An exemplary preprocessing tool chain in WebLicht.

¹ The login mechanism makes use of a so called authentication- and authorization infrastructure of the German research net (DFN-AAI). That way, researchers who have an account at an university or institutions that is connected to the German research net (DFN) are able to login to the environment. This however, does not concern the WebLicht services which can be accessed freely.

² XML is a short term for extensible markup language and basically defines a set of rules for encoding documents in a format that can be read by humans as well as processed by machines [XML]. An XML scheme definition (XSD) aids in formally specifying the elements in an XML document [XSD]. Thus an XML document can be validated against its scheme definition by checking if the elements contained in the document are conform to a given set of rules in the scheme.

2.4.3 Communication with WebLicht services

One aspect that makes WebLicht an ideal choice for language processing capabilities in RapidMiner is due to the possibility to directly communicate with the services over the web via the POST method of the HTTP protocol [HTTP]:

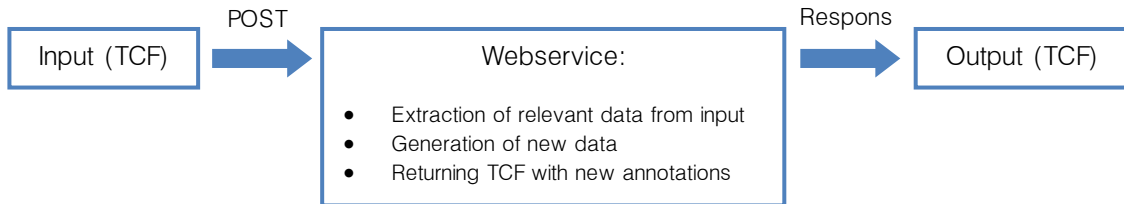


Figure 2-7: Communication with WebLicht services by using the HTTP POST protocol.

The web URL to each available service is given in the properties view of the web environment. This view provides additional information like input and output features that are used in the implementations later (Section 2.5.2). The properties of each service further indicate the content types, one for the data type that the service accepts, and the other type describes what type of data is sent back by the service. Content types are also known under the term *MIME*, short for Multipurpose Internet Mail Extensions³ [MIME]. Simply put, clients that send HTTP requests to a service need to indicate the MIME type expected from a service, so that the transmitted content can be properly processed by the WebLicht service. Respectively, a service sends a MIME type together with the content back to a client. Most of the time, WebLicht services expect and send content of the MIME type 'text/tcf+xml'. The first part 'text' indicates the general type of the file, whereas 'tcf+xml' indicates the subtypes. In this case the services accept file contents in 'tcf' and 'xml' format.

2.4.4 Preparing text corpora for WebLicht services

As mentioned above in Section 2.4.2, a text corpus needs to be converted into TCF first. To this end, requests are sent to either one of the following converter services:

Name of the tool	Developed by	Supported Languages	performed task
BBAoS&H Converter	Berlin Brandenburg Academy of Sciences and Humanities	de	text to text corpus (TCF)
SfS Converter	SfS: University of Tübingen	de, en, fr, it, and many more	text to text corpus (TCF)

Table 2-4: Available conversion tools to process texts to the text corpus format (TCF).

By sending a POST-request with the appended text content to one of the converter services, the returned text is being wrapped into the text corpus format (TCF). The next section demonstrates such a request.

³ Nowadays, MIME do not only describe file extensions used as attachments in emails, but describe numerous different content types in general. In communication protocols like HTTP for the WWW MIME types play an important role for applications that transmit and process file contents.

2.4.5 Testing the accessibility of WebLicht services

By using a simple communication program like "wget" the accessibility of a WebLicht service can be tested [Wget]. *Wget* allows sending files via the HTTP POST request and retrieves a response from the targeted web server. The following example demonstrates how a simple text is converted to an XML file (whose scheme follows the text corpus format (TCF)):

1. Input text (saved in file "Ballerina.txt"):

```
The ballerina was a swan, gliding over the stage.
```

2. HTTP POST request (sent via Wget):

```
wget --post-file="D:/Ballerina.txt" --header='Content-Type: text/plain' -O D:/Ballerina.xml  
"http://weblicht.sfs.uni-tuebingen.de/rws/service-  
converter/convert/?informat=plaintext&language=en&outformat=tcf04"
```

3. Service response (saved in "Ballerina.xml"):

```
<?xml version="1.0" encoding="UTF-8"?>  
<D-Spin xmlns="http://www.dspin.de/data" version="0.4">  
  <MetaData xmlns="http://www.dspin.de/data/metadata">  
    <source></source>  
  </MetaData>  
  <TextCorpus xmlns="http://www.dspin.de/data/textcorpus" lang="en">  
    <text>The ballerina was a swan, gliding over the stage.</text>  
  </TextCorpus>  
</D-Spin>
```

List 2-1: A returned text corpus converted into TCF in version 0.4.

After converting the input text to TCF, the text corpus can now be annotated with further linguistic features. In the next example, "Ballerina.xml" is sent to the tokeniser from the OpenNLP project via:

```
wget --post-file="D:/Ballerina.xml" --header='Content-Type: text/tcf+xml'  
-O D:/Ballerina_tokens.xml "http://weblicht.sfs.uni-tuebingen.de/rws/service-opennlp/annotate/tok-  
sentences?language=en"
```

4. The returned text corpus (saved in "Ballerina_tokens.xml"):

```
<?xml version="1.0" encoding="UTF-8"?>  
<D-Spin xmlns="http://www.dspin.de/data" version="0.4">  
  <MetaData xmlns="http://www.dspin.de/data/metadata"><source></source></MetaData>  
  <TextCorpus xmlns="http://www.dspin.de/data/textcorpus" lang="en">  
    <tc:text xmlns:tc="http://www.dspin.de/data/textcorpus">  
      The ballerina was a swan, gliding over the stage.</tc:text>  
    <tc:tokens xmlns:tc="http://www.dspin.de/data/textcorpus" charOffsets="true">  
      <tc:token end="3" start="0" ID="t_0">The</tc:token>  
      <tc:token end="13" start="4" ID="t_1">ballerina</tc:token>
```

```

<tc:token end="17" start="14" ID="t_2">was</tc:token>
<tc:token end="19" start="18" ID="t_3">a</tc:token>
<tc:token end="24" start="20" ID="t_4">swan</tc:token>
<tc:token end="25" start="24" ID="t_5">,</tc:token>
<tc:token end="33" start="26" ID="t_6">gliding</tc:token>
<tc:token end="38" start="34" ID="t_7">over</tc:token>
<tc:token end="42" start="39" ID="t_8">the</tc:token>
<tc:token end="48" start="43" ID="t_9">stage</tc:token>
<tc:token end="49" start="48" ID="t_10">.</tc:token>
</tc:tokens>
<tc:sentences xmlns:tc="http://www.dspin.de/data/textcorpus">
  <tc:sentence tokenIDs="t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_10"></tc:sentence>
</tc:sentences>
</TextCorpus>
</D-Spin>

```

List 2-2: A returned text corpus annotated with Sections for tokens and sentences.

As shown in List 2-2, a tokeniser does not only extract all the tokens, but also annotates the text corpus with a section for sentences in which the tokens are given in the same order as in each sentence. The next section presents the concept of combining different WebLicht services to a tool chain.

2.5 Annotating linguistic features in RapidMiner

2.5.1 Flexible tool chain concept for the 'WebLicht Feature Annotator'

Although the web environment of WebLicht is not relevant for our purposes, the idea of chaining tools is a useful concept for the software design of a 'Feature Annotation' operator in RapidMiner. Here, a text corpus is also annotated with different linguistic features (by means of WebLicht services) in an incremental way.

The following activity diagram (Figure 2-8) presents the *concept of a flexible tool chain* that is implemented as a RapidMiner operator with the name '**WebLicht Feature Annotator**':

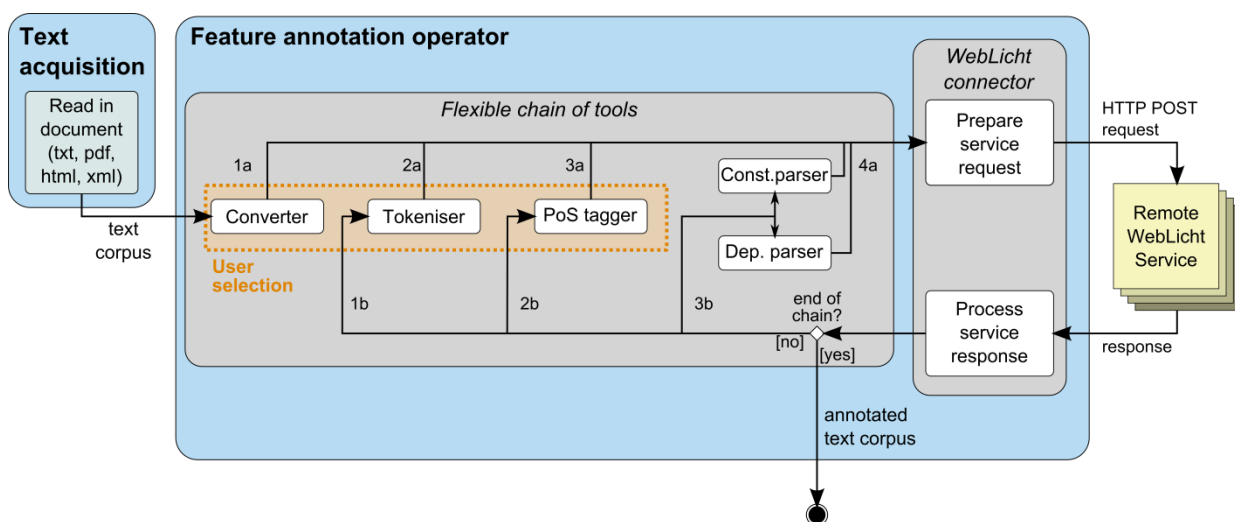


Figure 2-8: The concept of a flexible tool chain for the 'WebLicht Feature Annotator'

After loading a document via the "*read document*" operator in RapidMiner, the text corpus is forwarded to the feature annotation operator. In the parameter settings of this operator a RapidMiner user can select one of the following tool chains (documented in the Appendix A.1.2):

- converter
- converter → tokeniser
- converter → tokeniser → lemmatizer (not shown in Figure 2-8)
- converter → tokeniser → PoS tagger
- converter → tokeniser → PoS tagger → constituency parser
- converter → tokeniser → PoS tagger → dependency parser

The exemplary selection (highlighted with an orange border in Figure 2-8) includes only the first three linguistic tools (converter, tokeniser and PoS tagger). It should be noted, that the depicted tools in the program are placeholders for the distinct tools. Section 2.5.4 describes in detail, how specific WebLicht tools/services can be chosen.

The cascade like nature of the flexible tool chain behaves as follows:

1. In (1a) the converter sends a text corpus to the WebLicht connector (Section 2.5.3) which prepares and sends a HTTP POST request to a WebLicht converter. On a service response the connector converts the received text corpus (in TCF) to a RapidMiner document and forwards it to the next tool in the selected chain (1b), provided that the end of the selected chain is not reached.
2. In (2a) the tokeniser triggers the connector to send the text corpus to a tokeniser WebLicht service. The received TCF content contains an XML section that lists all the tokens in the text corpus (2b).
3. In the last step of the selected tool chain the tokenized text corpus is sent to a PoS-tagger and obtain a corpus that is annotated with the PoS-tags according to the tokens.

However, not every service can be connected with one another in the tool chain. For example, the tokeniser from the OpenNLP project annotates the text corpus with tokens only while different PoS taggers additionally require an XML section for sentences. When a specific language parameter for a text corpus should be defined, the problem is that not every service supports that language. Due to these restrictions, all the dependencies of services among one another have to be considered.

In order to implement a RapidMiner operator that is easily manageable and allows adjustments in the case of future changes of any of the WebLicht services, or allows adding new WebLicht services, it is desirable to establish a *configurable tool chain* that has the following properties:

- **Flexibility:** Allow to easily modify the configuration of a WebLicht service
- **Scalability:** Add or remove a tool (in a tool category⁴) that is added/removed in the tool chain
- **Adjustments:** Easily change any parameter of the service: URL, in-/output features, supported languages, and more
- **Syntactical correctness:** Using XML & XSD allows to ensure a correct configuration setup

One possible way to provide a flexible setup is achieved with an XML configuration file [XML]. Furthermore, by defining an own XML Scheme Definition [XSD] (listed in Appendix A.1.3) it is ensured that the elements and attributes of the configuration XML remain syntactically correct so that a configuration loader can properly read the necessary information from each service.

How the different service parameters are incorporated into the XML configuration is presented in the next section.

⁴ Due to programmatic limitations in the implementation of the RapidMiner operator we have to restrict the flexibility of the configuration to the set of known tool categories.

2.5.2 XML Configuration of the WebLicht tool chain

According to the properties view that is given along with each service in the web environment of WebLicht, the following service properties are mapped into single XML elements:

- Creator
- Contact (email address)
- Description
- Input features
- Output features
- PID (basically, a service ID registered at WebLicht)
- URL (server address in the WWW)

Based on the XML Scheme Definition in the Appendix A.1.3, an **XML structure** allows to configure a specific tool. In List 2-3 the exemplary configuration of the "IMS PoS tagger" from the University of Stuttgart is shown:

```
<tool_group category="pos-tagger">
  <tool id="1"> </tool>
  <tool id="2">
    <creator>IMS: University of Stuttgart</creator>
    <contact>clarin@ims.uni-stuttgart.de</contact>
    <description lang="en">[IMS] PoS TreeTagger(2008): Italian, English, French,
      and German part-of-speech tagger and lemmatiser
    </description>
    <input_features lang="it,en,fr,de" mime_type="text/tcf+xml"
      type_description="tokens" version="0.4" />
    <output_features lang="" mime_type="" postags.tagset="stts"
      type_description="lemmas, POStags" version="" />
    <pid>http://hdl.handle.net/11858/00-247C-0000-0007-3739-5</pid>
    <url>http://clarin05.ims.uni-stuttgart.de/treetagger2008</url>
    <url_params></url_params>
  </tool>
  <tool id="3"> </tool>
</tool_group>
```

List 2-3: An extract of the XML configuration for a WebLicht service that is available in the tool chain

The only attribute in the element *description* defines the language that is used for the description text of the given service. The *input_features* and *output_features* only contain XML attributes that define the language ("lang"), the MIME type ("mime_type") and the linguistic features ("type_description") accepted and returned by the service. In the case of a PoS-tagger tool, the element *output_features* specifically contains the attribute "postags.tagset". This information is later on used during the visualization of linguistic features (Chapter 3). Lastly, the XML element *url_params* is a placeholder for parameters that can additionally be appended to the *URL* used in an HTTP request to the WebLicht service. Other service categories have similar XML sections. The full XML configuration is listed in the Appendix A.1.4.

2.5.3 Implementing the WebLicht connector

In order to let the 'WebLicht Feature Annotator' communicate with WebLicht services that are selected in the tool chain, a "WebLicht connector" is implemented that sends HTTP requests to a service in a simple way. For this purpose, the RESTful services *JAX-RS 2.0* [JAX-RS] provides a comfortable application programming interface (API). This framework is included in the *JAX-RPC* API which is part of the Java Platform since Java Enterprise Edition 5 [JAX-RPC]. In Code Snippet 2-1 the implementation of a straightforward connection setup is listed:

```
final ServiceTool serviceTool = webLichtServices.getServiceTool(toolCategory, toolID); // (1)
final Client client = ClientBuilder.newClient(configuration); // (2)
WebTarget target = client.target( "http://" +
                                serviceTool.getHost() + ":" +
                                serviceTool.getPort() ).path( serviceTool.getPath() ); // (3)

for ( Entry<String, String> param : serviceTool.getUrlParams().entrySet() ) // (4)
    target = target.queryParam(param.getKey(), param.getValue()); // (5)

Entity<String> uploadEntity = Entity.entity( inputDocument.getText(),"text/tcf+xml"); // (6)
Response response = target.request("text/tcf+xml").post(uploadEntity); // (7)

String responseText = response.readEntity(String.class); // (8)
Document returnDoc = new Document(responseText); // (9)
```

Code Snippet 2-1: Connection setup of the 'WebLicht Feature Annotator' which allows uploading content to a WebLicht service and receiving a response.

The Code Snippet 2-1 demonstrates the steps of a HTTP POST communication with a WebLicht service: In the first step (1), the tool with a specific category and identifier is fetched from the list of available tools (previously loaded from the configuration XML). Then, an instance of the JAX-RS specific class *Client* is created (2) which is used to build and execute client requests and which consumes responses from a WebLicht service. In (3) a *WebTarget* object is created which targets a WebLicht service by a specified URL (composed of the parts host address, access port and the local path under which the service can be found on the destination server). The current *serviceTool* accesses all the required information (which are read from the XML configuration) via the methods *getHost()*, *getPort()* and *getPath()*. With *getUrlParams()* in (4) service specific parameters like "?language=de" are obtained. With *queryParam()* in (5) pairs of keys and values of each parameter are appended to the URL (several key-value-pairs are separated by "&").

In (6) the input document is prepared for sending. This is done via *Entity.entity(...)* where the input text is passed as the first parameter, and the MIME type as the second parameter. With *target.request().post(uploadEntity)* not only the entity is sent to the destination service, but also a *Response* object is created that silently waits for a response from the service (7). Via *target.request("text/tcf+xml")* the MIME types 'TCF' and 'XML' are defined that the client is allowed to receive. On a service response in (8) the client reads the received entity via

response.readEntity() and stores the saved string into a new document that now contains the annotated text corpus (9).

2.5.4 Compatible WebLicht services in the tool chain

Based on the parameters of each service that a configuration loader reads in from the XML configuration file (Section 2.5.2), the 'WebLicht Feature Annotator' operator determines allowable tool chain combinations. The list of valid combinations is presented in the following Figure 2-9:

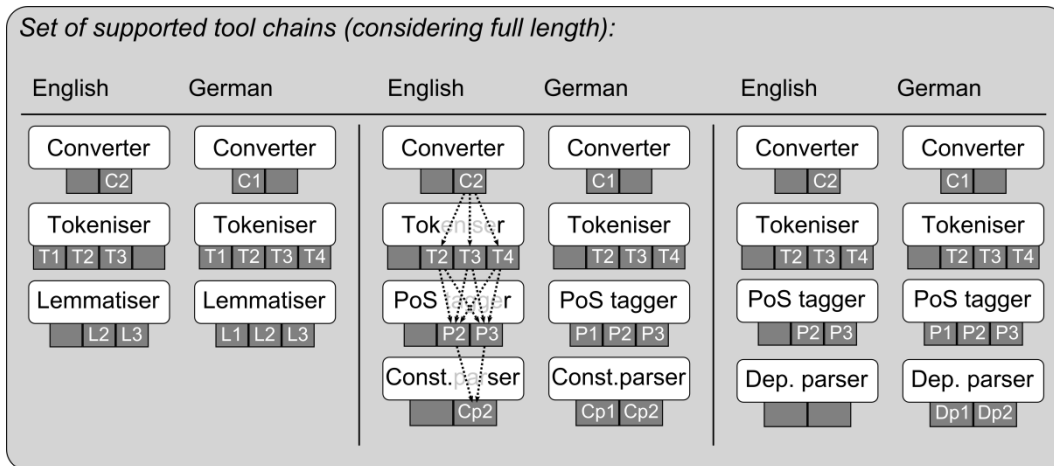


Figure 2-9: Combinations of supported tool chains

As shown in Figure 2-9, the tool chains (mentioned in Section 2.5.1) are divided into the language parameter 'English' and 'German'. Services that support other languages can be added, and the configuration actually integrates services to process French and Italian texts, as well (for brevity these are not shown in Figure 2-9).

By defining the language parameter for the input text corpus, the operator determines the available tool chains together with the list of valid tools that support the chosen language.

Then, in the parameters of this operator (see Appendix A.1.2) tool lists of specific categories (converter, tokeniser, lemmatizer, PoS tagger, and parser) contain available services. Since these lists never contain tools that are not supported, a RapidMiner user can only choose valid combinations to form a tool chain⁵. For instance, by setting the language parameter to English and choosing the first tool chain "Converter→Tokeniser→Lemmatiser" (first column in Figure 2-9) the list of converters only consists of C2. Figure 2-9 represents this by hiding C1.

Regarding tokenisers in the last four columns in Figure 2-9, the tokeniser from the OpenNLP project does not deliver sentences as its output feature, but PoS-taggers depend on this feature. Therefore, the user may only choose from services T2 - T3.

⁵ The set of services shown in Figure 2-9 does not match with those in Table 2-3 since (at the time of writing) few services are declared as "in development" and thus not suitable for productive use.

As mentioned in section (2.5.1), the implemented 'WebLicht Feature Annotator' includes a dependency mechanism to check if the required *input features* of a selected service are provided by previous tools in the chain. The dependency mechanism is realised by the specific method "checkToolCompatibility()" as shown in the class diagram in Appendix A.1.5. It checks if the input features of a selected service are contained in the set of output features that have been collected in the tool chain so far.

One shortcoming is that both dependency parsers offered by WebLicht do not support analyzing English text corpora (second last column in Figure 2-9). This, may change in the future if such a parser is added to the repository of tools in WebLicht.

Overall, the implementation of the 'WebLicht Feature Annotator' appears justified as the benefits of the tool chain outweigh the single disadvantage of a (currently) lacking dependency parser for English texts:

- **The RapidMiner user / linguistic researcher has the possibility to quickly replace tools that deliver unsatisfactory results.** For example, each Weblicht service may annotate a text corpus with features differently since it has been trained on a specific text corpora or makes use of different detection mechanisms. This is the case for tokenisers that have named entity recognizers included, and as such may return different recognized named entities for the same tokens. Another example is where two constituency parsers deliver different results in the case of structural ambiguous sentences as shown in Figure 2-2.
- **The flexibility of the tool chain allows the user to easily try out different combinations of services** while the implementation ensures the validity and correctness of the chosen tool chain.
- **The configuration is straightforward to manage and allows adding, editing or removing of WebLicht services.**

The detailed documentation to the 'WebLicht Feature Annotator' operator is given in the Appendix A.1.

2.6 Extraction of linguistic features in RapidMiner

2.6.1 Motivation for implementing a parser for annotated text corpora

The developers of WebLicht at the University of Tübingen provide a library that contains a TCF parser [TCF0.3Parser]. However, the parser is incompatible with the XML corpora delivered by the WebLicht services as it only accepts input in TCF version 0.3 (specification 2009) while the WebLicht services produce XML corpora in TCF version 0.4 (specification 2011) [TCFSpec]. If the version were intentionally changed from 0.3 to 0.4 for all input text corpora, this would result in an unknown number of exceptions in which the provided parsers could not properly recognize specific XML content.

Due to the above reason and for easier adaptability in the future, a parser is implemented in order to read all the different XML elements, attributes and data contents that may occur in an annotated text corpus. For processing of XML content the implementation makes use of *JAXP*, the Java API for XML processing, which is a standard component in many Java development kits [JAXP].

2.6.2 Parsing linguistic features from annotated text corpora

After an annotated text corpus has been received, a process is required that extracts and outputs linguistic features for further use in RapidMiner. More precisely, an XML parser is implemented and integrated into the 'WebLicht TCF to ExampleSet' operator (see Appendix A.2) that properly detects the relevant sections in the XML structures of a given text corpus. Afterwards, the operator places the features of each type into distinct columns of a new *ExampleSet*⁶. Figure 2-10 depicts the involved input and output of this operator:

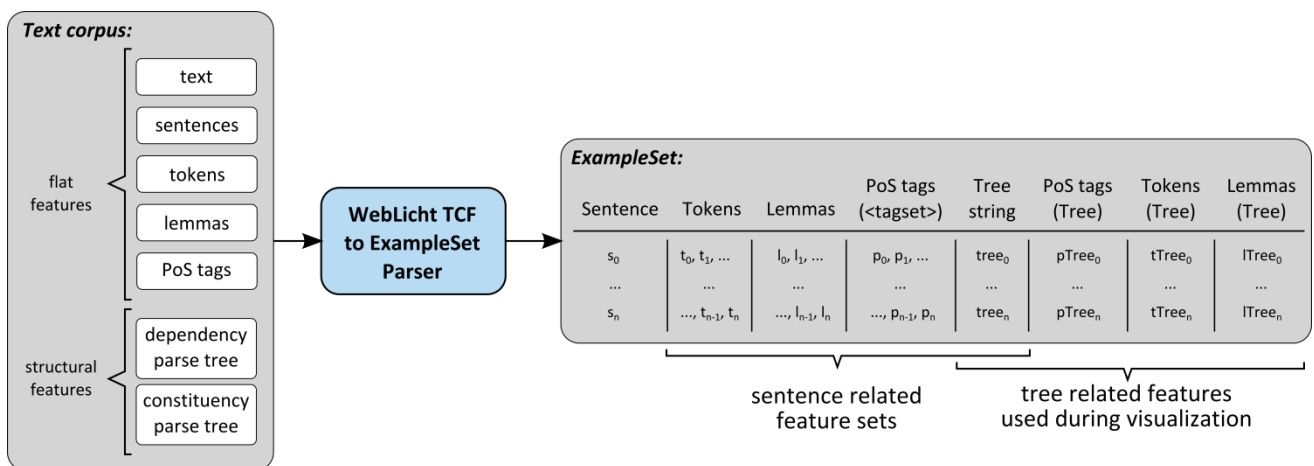


Figure 2-10: The extraction process of linguistic features from a text corpus

⁶ An ExampleSet is the most frequently used data structure in RapidMiner and allows the user to forward data to various operators for further processing.

As shown in Figure 2-10, the original text is not forwarded to the output ExampleSet, but instead the sentences s_0, \dots, s_n are placed one after the other in the column "Sentence". Flat features associated with each sentence are placed in the columns "Tokens", "Lemmas" and "PosTags". The label of the column "PoS-tags" additionally carries the information which tagset the PoS-tags belong to. Note that features are always stored as strings in the cells of the ExampleSet.

If a text corpus contains parse trees, the parser detects the type of the trees. The trees are then transformed into string representations that use the bracket notation as described in Section 2.2.6. In the following course of this work, the term '*tree string*' refers to this representation. The encoded tree strings $tree_0, \dots, tree_n$ are then placed in the column "Tree string" in the corresponding order of the sentences given in the first column.

While the parser performs the feature extraction from corresponding XML sections in the corpus (Section 2.6.3), it is important to take into account that WebLicht services may occasionally return incomplete sets of features. This problem occurs when a requested service is unable to properly analyze a specific sentence. Unfortunately, in this situation the service denies to further process the remaining sentences and only delivers the *partial set of features* according to the initial sentences that were properly parsed so far.

Since it is desirable to associate tokens, lemmas or PoS-tags with each according sentence, the parser deals with *incomplete feature sets* by processing each feature type *independently*. The parsing process is proceeded on the following assumptions:

1. The selected tool chain contains a tokeniser. Since tokenization usually involves no complex analytical processing (as no text corpora with Asian language is processed), and therefore each chosen tokeniser service is robust enough to provide a full list of tokens that occur in the given text corpus.
2. The selected tokeniser annotates the corpus with *a section for sentences*.

In the exceptional case where the parser notices a partial set of a distinct feature type (lemmas, PoS-tags, parse trees) the highest token ID i in the text corpus is determined. Then, only the first i features are associated with the corresponding sentences, and the last $(n - i)$ entries of the corresponding feature type remain empty. With these circumstances at hand, the parser is implemented with the following properties:

- **Preserve the order** of features as occurring in the XML text corpus
- **Flexible parsing:** Only parse existing feature sections in the XML text corpus, and produce the output columns in the ExampleSet accordingly
- **Robustness:** Continue the parsing process despite errors in the XML text corpus like missing XML elements or different naming conventions
- **Readability:** Encode parse trees via string bracket notation
- **Enrich parse trees:** If parse trees are contained in the XML text corpus, add columns in the ExampleSet for each existing flat feature type (tokens, lemmas, PoS-tags). Then in

each column add a list of features as they would occur in the parse tree when performing a *pre-order* traversal of that parse tree.

The visualization operator for parse trees makes use of the last property (see Section 3.5). Since parse trees only contain tags from a specific tagset it is desirable to visualize tokens and lemmas in the nodes, as well. By simply traversing each parse tree in *pre-order* the parser can match each tag with the corresponding token or lemma (if existing⁷) and appends the token or lemma to a 'tree string (tokens)', or 'tree string (lemmas)' respectively. The *pre-order traversal* is a recursive method that is defined as follows [treeTraversal]:

1. Read the data of the root element of the current tree (or subtree)
2. traverse to the left subtree by recursively calling the pre-order method
3. traverse the right subtree by recursively calling the pre-order method.

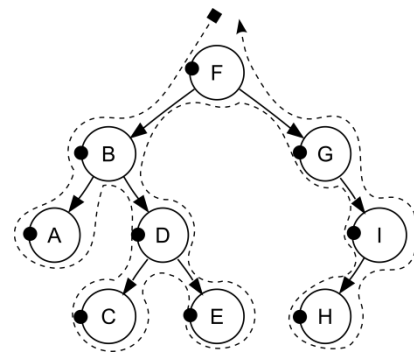


Figure 2-11: Pre-order traversal of an exemplary tree

Figure 2-11 depicts an exemplary tree whose pre-order traversal is given by: F, B, A, D, C, E, G, I, H. Appending the data in the visited nodes to a 'tree string' yields:

$$\left[F \left[B \left[A, D \left[C, E \right] \right], G \left[I \left[H \right] \right] \right] \right]$$

By matching the tokens, lemmas or PoS-tags according to the tags in each leaf of the parse tree, the parser produces the tree strings according to Figure 2-11 in the following way:

TreeString:	<code>[F[B[A,D[C,E]],G[I[H]]]]</code>
Tokens (Tree):	<code>, , token_A, , token_C, token_E, , token_H</code>
Lemmas (Tree):	<code>, , lemma_A, , lemma_C, lemma_E, , lemma_H</code>
PoS-tags (Tree):	<code>, , PoS-tag_A, , PoS-tag_C, PoS-tag_E, , PoS-tag_H</code>

2.6.3 Feature extraction from XML sections in the TCF document

The following list briefly describes the parsing of different XML sections in an text corpus that has been annotated with various features. Since not all WebLicht services follow the TCF 0.4 specification [TCFSpec], different cases are considered in which the implemented parser takes special care in order to properly extract annotated features:

1. **Tokens** (a mandatory section in the input text corpus):

```
<tc:tokens xmlns:tc="http://www.dspin.de/data/textcorpus" charOffsets="true">
  <tc:token end="3" start="0" ID="t_0">The</tc:token>
```

⁷ An unmatched tag in the parse tree results in an empty string.

```
</tc:tokens>
```

By searching for an XML element that contains "tokens" the parser can find the token section. Because not every WebLicht service follows the same naming convention prepended strings like "tc:" are ignored. When the section is found, simply one token after the other is parsed and stored in an internal list.

2. **Sentences** (a mandatory section in the input text corpus):

```
<tc:sentences>
  <tc:sentence tokenIDs="t_0 t_1 ..." />
  ...
</tc:sentences>
```

This sentence section is added by every tokeniser service in the tool chain (with the exception of the tokeniser from the OpenNLP project). By parsing the attribute "tokenIDs" the parser is able to associate a set of tokens to the appropriate sentence. XML data has to be parsed carefully as WebLicht services use different naming and numbering conventions. Some services start the list of "tokenIDs" with "t0", while other services start it with "t_1". Since all features like lemmas or PoS-tags carry a reference to the *tokenID* all features can be associated with the according sentence.

3. **PoS-tags** (an optional section in the input text corpus):

```
<tc:POStags tagset="STTS">
  <tc:tag tokenIDs="t_0">ART</tc:tag>
</tc:POStags>
```

While scanning for "POStags", the parser additionally reads the attribute "tagset" indicating the tagset of the given PoS-tags. Again, special care needs to be taken during the parsing: For example, when indicating sentence punctuation, some services prepend "\$" or "\\$" to the tag and other services, however, return them directly as is.

4. **Lemmas** (an optional section in the input text corpus):

```
<tc:lemmas>
  <tc:lemma ID="l_0" tokenIDs="t_0">the</tc:lemma>
</tc:lemmas>
```

5. **Constituency parse trees** (an optional section in the input text corpus):

```
<tc:parsing tagset="tuebadztb"><<parse>
  <constituent cat="VROOT" ID="c_56">
    <constituent tokenIDs="t_1" cat="ART" ID="c_1"></constituent>
  ...
</tc:parsing>
```

```

    </constituent>
</parse>
...
</parsing>

```

The section for constituency parse trees usually starts with an XML element "tc:parsing". Because some services omit the leading "tc:" or use a different naming scheme, the parser only searches for "parsing" in order to find this section.

The constituents in each sentence are enclosed by a pair of "<parse>...</parse>" elements. The parser inherits the tree structure by following the constituents nested within other constituents in a pre-order traversal. A leaf is simply represented by a pair of opening and closing "constituent" tags without having any content in between. Siblings on the same level of the tree are represented by placing the "constituent" elements one after another.

6. **Dependency parse trees** (an optional section in the input text corpus):

```

<ns3:depparsing emptytoks="false" multigovs="false" tagset="tiger">
  <ns3:parse>
    <ns3:dependency govIDs="t_2" deplDs="t_0" func="MO"/>
    <ns3:dependency govIDs="t_0" deplDs="t_1" func="NK"/>
    <ns3:dependency deplDs="t_2" func="ROOT"/>
    <ns3:dependency govIDs="t_4" deplDs="t_3" func="NK"/>
  </ns3:parse>
  ...
</ns3:depparsing>

```

The section for dependency parse trees is recognized by an XML element that contains "depparsing". Again, some services omit the leading "nc:" or make use of a different naming scheme. In the same way as for constituency parse trees, each sentence is enclosed by a pair of opening and closing "parse" elements. The parser follows the tree structure by starting from the dependency whose attribute "func" has the value "ROOT". Then, the depending ID "deplDs" indicates the child element in the tree one level below. The dependent elements can be found by examining the values of the attribute "govID". While parsing down the tree, the attribute "func" of each element carries a PoS tag or grammar related tag (depending on the used tagset of the WebLicht service) and is determined by the parser.

2.7 Discussing linguistic features in the context of a hypothetical task

This section deals with the *idea of detecting metaphors in sentences* according to the types of linguistic features that can be obtained from the 'WebLicht Feature Annotator'.

2.7.1 Definition of a metaphor

A metaphor is a figure of speech in which an implied comparison is made between two unlike things that in fact have something in common. According to the simple model by [Richards] a metaphor consists of an unfamiliar part (*tenor*) and a familiar part (*vehicle*). The *tenor* is the subject to which attributes are ascribed while the *vehicle* is the object to which these attributes are borrowed. In the German sentence "Der Mann ist ein Schrank" which translates to "the man is a cupboard" the tenor is "Mann" and the vehicle is "Schrank". Further examples are: "He is a walking dictionary" or "The ballerina was a swan, gliding across the stage".

For the discussion in the next section let us only consider metaphors of the above simple model where both *tenor* and *vehicle* only consist of nouns. Other metaphorical expressions use a wide range of possibilities to combine nouns with verbal phrases or particles which in turn would be far more difficult to analyze. Another condition that should be excluded is when the tenor is omitted since the speaker assumes that the recipient concludes the tenor from the given context.

Furthermore, metaphors need to be distinguished between *creative* and *dead* ones: On one hand, *creative metaphors* are essentially of innovative nature either because they occur very rarely (e.g. "reshuffle the deckchairs on the Titanic") or in specific contexts (e.g. "Pyrrhic victory") or they have found their way in language use at some time (e.g. "to google").

On the other hand, *dead metaphors* subsume two types: In the first type the sense of a transferred image is absent, like in the German verb "begreifen" which means "to understand" where the physical action is used as a metaphor for understanding. Such metaphors are no more visualized, and usually go unnoticed. The second type of metaphors have such a high frequency in linguistic usage that they have been lexicalized (e.g. the German noun "Fundgrube" which translates to "bonanza").

The distinction between *creative* and *dead* metaphors is usually achieved by dictionaries containing a large set of dead metaphors so that these can simply filtered out from a given text corpus.

2.7.2 Discussing linguistic features for pattern detection

In the context of the processing pipeline for a classification task (Figure 1-1), the following Figure 2-12 depicts how linguistic features are obtained right at the start, then transform them in a processing step, and finally forward them to a machine learning method in order to perform a pattern detection. Generally, these steps also resemble a typical data mining task that bridges **computational linguistics** with **machine learning** in an interdisciplinary manner:

Computational linguistics

Machine Learning

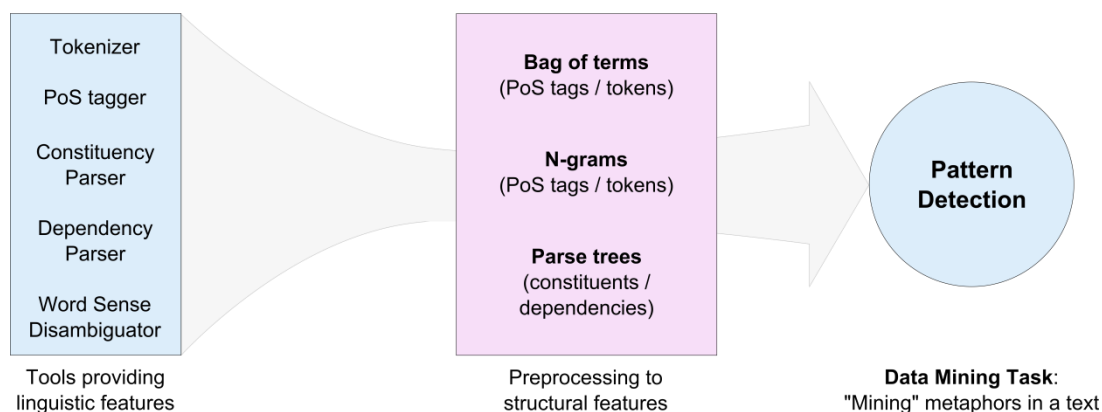


Figure 2-12: An interdisciplinary perspective on an exemplary data mining task

With different types of linguistic features at hand (Figure 2-12, center box), first it is unclear which feature type or combination of feature types is most applicable for a given NLP task like the exemplary idea of detecting metaphors. The following Table 2-5 demonstrates the case where parts-of-speech tags are considered alone, in the attempt of finding patterns that correspond to a possible metaphor in a sentence:

Example	Sequence of PoS-tags	interesting PoS-tags	Metaphor?
Der Mann ist groß <i>This man is tall</i>	ART NN VAFIN ADJD	NN ADJD	no
Der Mann ist Lehrer <i>Wolfgang is (a) teacher</i>	ART NN VAFIN NN	ART NN NN	no
Der Mann ist ein Schrank <i>This man is a cupboard</i>	ART NN VAFIN ART NN	ART NN ART NN	yes
Julia war ein Schwan <i>Julia was a swan</i>	NE VAFIN ART NN	NE ART NN	yes

Table 2-5: A set of examples with sequences of PoS-tags; the tags are taken from the "Stuttgart-Tübingen Tagset" (STTS); the translations are given in the second lines

The column with "interesting PoS-tags" shows possible subsets of part-of-speech tags that are considered relevant in the given sentences. The last two sentences contain metaphors and show that the *tenor* ("Der Mann", "Wolfgang", "Julia") either is a common noun (NN) or a named entity (NE), and that the *vehicle* of the metaphor is tagged with NN or NE, as well. Concerning the German language, the *vehicle* of a metaphor often appears in combination with an article (ART).

In order to decide whether a sentence contains a metaphor or not, first of all an exemplary case is required that provides a distinct set of features with an unambiguous pattern. In terms of machine learning, this detection task resembles a binary classification problem in which a given set of training data needs to be separated in two classes. In order to perform a machine learning, we need to know if a specific set of features exists such that a decision rule can unambiguously decide to which class a given sentence belongs. If such a set exists, a decision rule could be learned and later on a classification could be performed on unseen data.

Let us consider n-grams comprised of sequences of parts-of-speech tags (with n=5):

1. Der Mann ist Lehrer	<u>5-gram of PoS-tags</u> →	ART NN VAFIN NN	(no metaphor)
2. Der Mann ist ein Schrank	<u>5-gram of PoS-tags</u> →	ART NN VAFIN ART NN	(metaphor)

Example 2-3: Sequence of n-grams with n=5; the blue coloring indicates the subject or the tenor of a metaphor. The orange coloring indicates the object phrase or the vehicle of a metaphor.

In Example 2-3 the only difference between the n-grams is given by the tags 'NN' (for "Lehrer") and 'ART NN' (for "ein Schrank"). Another factor we have to take into account are tokens that are not related to a metaphorical expression. These tokens potentially feature arbitrary PoS-tags, and potentially add more ambiguous cases which further permits any clear distinction.

Another example is given below to demonstrate a shortcoming of n-gram features due to their fix length. Since sentences are usually arbitrary long, interesting words can potentially have arbitrary words in between. For example, by choosing n=4 the second sentence yields two combinations of 4-grams:

1. Der Mann ist Lehrer	<u>4-gram</u> →	ART NN VAFIN NN
2. Der Mann ist ein	<u>4-gram, 1st sequence</u> →	ART NN VAFIN ART
Mann ist ein Schrank	<u>4-gram, 2nd sequence</u> →	NN VAFIN ART NN

Example 2-4: Sequence of 4-grams; the second sentence yields two 4-grams of PoS-tags.

By comparing the 4-grams of the first sentence and the first sequence of the second sentence, a difference is obtained in the tags 'NN' (vehicle) and 'ART'. However, this does not give any clue at all. When we compare the n-gram of the first sentence with the second sequence of PoS-tags, we obtain the matching tags 'ART NN' (tenor) from the first example and 'ART NN' (vehicle) from the second sequence, although we want to compare 'NN' (vehicle) with 'ART NN' (vehicle). This problem would additionally require to determine correct start positions of interesting PoS-tag sequence in order to compare only *relevant pairs* of PoS-tags.

Considering the first sentence, there are no rules in German that define when to add or omit an indefinite article (like "ein") to a common noun. Both in written and colloquial language the difference is sensed very subtly. Thus, the first sentence could equally be written as:

1. Der Mann ist ein Lehrer	<u>interesting PoS tags</u> →	ART NN VAFIN ART NN	(no metaphor)
2. Der Mann ist ein Schrank	<u>interesting PoS tags</u> →	ART NN VAFIN ART NN	(metaphor)

Example 2-5: Variation of Example 2-3 with an indefinite article added to the common noun in the first sentence.

Example 2-5 proves that PoS-tags alone are not suitable for an unambiguous distinction between sentences. Furthermore, if structural features like constituency parse trees (Section 2.2.5) are considered, we could determine the proper tag for each constituent that forms a

subject or object phrase, and still end up in ambiguous cases. This leads to the conclusion that no feature type, that has been presented so far, would allow a proper binary classification.

However, the following concept may offer an approach to examine and decide ambiguous cases: First, we determine pairs of "candidate" tokens that possibly form a metaphorical expression. This can be achieved by only searching for pairs of their corresponding PoS-tags. If we constrain sentences with metaphors to only be made up of nouns or named entities in the subject and object phrases, we could simply search for pairs of 'NN' or 'NE' tags.

In the given example above, the tag 'NN' refers to the token "Mann" and the 'NN' tags to the tokens "Lehrer" and "Schrank". In the next step, a *word sense disambiguation tool* could retrieve the *senses* to each corresponding tokens.

The basic idea is as follows: If the sense of one of the "candidate tokens" is *not* related to the *semantic field*⁸ of the other token then the given pair of tokens are likely to form a metaphorical expression.

Figure 2-13 shows an exemplary semantic field of the word "purple". In linguistics, a specific term (here "purple") whose semantic field is included within the semantic field of another more general term ("color"), is called a *hyponym*. Respectively, the semantic field of a *hypernym* subsumes instances of more specific terms ("purple", "red", and so on). Terms on the same level in the field are called *co-hyponyms*.

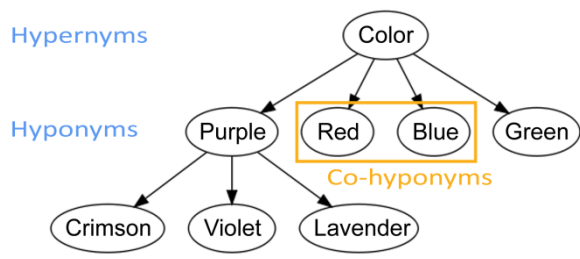


Figure 2-13: An exemplary semantic field of the term "purple"

Regarding the distinction between the sentences from Example 2-3 a **word sense disambiguation tool** could provide the following senses (given in brackets after the PoS-tags):

1. Der Mann ist Lehrer	interesting PoS tags →	NN (person) NN (job)
2. Der Mann ist ein Schrank	interesting PoS tags →	NN(person) NN (object)

Example 2-6: The exemplary sentences are now annotated with PoS-tags and their word senses. These senses can then be compared regarding their semantic fields.

With this detection concept, the second sentence in Example 2-6 could be unambiguously classified as a metaphorical expression since the term "person" for the token "Mann" does not share the same semantic field with the term "object" for the token "Schrank".

⁸ A semantic field is a set of words grouped by a meaning that is referred to a specific subject [Jackson et al.]. From an intuitive point of view, words in a semantic field are not simply synonyms, but rather are possible words that describe the same general phenomenon [Akmajian et al.]. Furthermore, the sense of a word is partly depending on its relation to other words in the same conceptual area [Hintikka].

Chapter 3

Feature Visualization

This chapter deals with the visualization of structural features extracted by the "WebLicht TCF to ExampleSet" operator (Section 2.6). A commonly used representation for linguistic structures is the tree graph, also referred to as *tree bank*. As already described in previous chapters these tree banks mainly come in two types: Constituency trees that display hierarchies of phrases (Section 2.2.5) and dependency trees that represent relations by drawing lines between dependencies within a text corpus (Section 2.2.6).

Before different concepts for tidy tree drawing are presented, various definitions of graphs and trees are given in Section 3.1. Afterwards, Section 3.2 describes the representation of relational data used to describe parse trees. According to [Battista], the drawing of a graph in a pleasing and tidy way can be divided into the categories of *drawing convention* (Section 3.3.1), *aesthetic standards*, and the constraints on aesthetics in a drawing (Section 3.3.2). Following these constraints, two different drawing algorithms are presented in Section 3.4: First, the "Layered-Tree-Draw" algorithm that constructs tidy tree layouts (Section 3.4.1). However, because this algorithm does not produce tree layouts with minimal breadth, the drawing algorithm proposed by Reingold & Tilford is shown in Section 3.4.2. Finally, Section 3.5 presents the implemented visualization operator that provides visual insights into structural relations of parse trees.

3.1 Terminology of graphs and trees

This section contains the most relevant definitions of graphs and trees that are used later on. We start with the general type of a graph $G = (V, E)$ which consists of a finite set V of vertices and a finite set E of edges consisting of unordered pairs of vertices. A vertex is often called *node*, and the terms *arc*, *link*, or *connection* are used instead of edge. The end-vertices of an edge $e = (u, v)$ are the vertices u and v . These nodes are also called *adjacent* to each other and the edge e is *incident* to u and v . The *neighbors* of v are its adjacent vertices. The *degree* of v is the number of its neighbors. An edge (u, v) with $u = v$ is a self-loop. Furthermore, an edge that is contained more than once in E is called a *multiple edge*. A *simple* graph does not contain any self-loops and no multiple edges.

A *directed graph* (*digraph*) is defined similarly to a graph. The only exception is that the set of edges E , called *directed edges*, contain ordered pairs of vertices. A directed edge (u, v) is defined by an *outgoing edge* of u and an *incoming edge* of v . Usually a directed edge is drawn as an arrow. Vertices without outgoing edges are called *sinks*. Respectively, vertices without incoming edges are called *sources*.

A (directed) path in a (directed) graph $G = (V, E)$ is a sequence (v_1, v_2, \dots, v_h) of distinct vertices of G , such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq h - 1$. A path is a *cycle* if $(v_h, v_1) \in E$. In this context, a graph in which any two vertices are connected by exactly one path is called *acyclic*.

A *tree* is a connected acyclic (and thus simple) graph. Furthermore, a tree is called a *rooted tree* if one vertex has been designated as the *root*. Commonly, a tree is considered as a directed graph while all edges are oriented away from the root. One property of a rooted tree is that the *parent*(v_i) of any vertex v_i lies on the path (v_{root}, v_i) to the root v_{root} . Furthermore, every vertex $v_i \neq v_{root}$ has a unique parent. A rooted tree in which each vertex has at most n children is also referred to as *n-ary tree*. In the case of two children the tree is called *binary tree*. Vertices that have no children at all are called *leaves*.

A graph $G' = (V', E')$ with $V' \subseteq V$ and $E' = E \cap (V' \times V')$ is called a *subgraph* of $G = (V, E)$. Respectively, a *subtree* T' rooted at vertex $v \neq root(T)$ consists of the subgraph induced by all vertices on paths originating from v , with $v = root(T')$. The *depth* of a vertex v of a tree T is defined by the number of edges that lead on a path from $root(T)$ to v . The height of T is the maximum depth of a vertex of T .

We call an edge (u, v) of a digraph *transitive* if a directed path from u to v exists while $(u, v) \notin E$. The transitive closure G' of a digraph G has an edge (u, v) for every path from u to v in G .

Next, we define the term *drawing*: A drawing Γ of a graph (digraph) G is a function that maps each vertex $v \in V$ to a distinct point $\Gamma(v)$. Every edge (u, v) is mapped to a so called *simple open Jordan curve*, with endpoints $\Gamma(u)$ and $\Gamma(v)$ [Battista, JordanCurve]. Furthermore, we call Γ *planar* if no two Jordan curves intersect. Thus, a graph is planar if it admits a planar drawing.

3.2 Modeling relational structures of parse trees

In order to express parse trees of annotated text corpora (Sections 2.2.5 and 2.2.6) an appropriate model is required that represents these trees. A parse tree is given as a relational structure which consists of entities and the relationship between them. These entities are linguistic units like words (in a dependency parse trees) or constituents (in a constituency parse tree).

Modeling relational structures as trees can be done in many ways. The representation of an entity is usually a vertex (drawn as points or boxes), while the relationship between two entities is visualized by an edge (drawn as straight or curved lines) that connects the associated vertices.

One way to describe parse trees is by using the *list notation*⁹ with list entries L_u of edges that are incident to vertex u for each $u \in V$. The following Table 3-1 shows an exemplary tree T (which is the parse tree shown Figure 3-4 below):

L_1	(1,2)
L_2	(2,3), (2,4), (2,5)
L_3	(3,6), (3,7), (3,8), (3,9)
L_4	(4,10), (4,11)
L_{11}	(11,12), (11,13)
L_{13}	(13,14), (13,15), (13,16)

Table 3-1: The list notation of an exemplary tree T

A more compact description of a graph is a $n \times n$ *adjacency matrix* A whose columns and rows correspond to n vertices, with $A_{uv} = 1$ if $(u, v) \in E$ and $A_{uv} = 0$ otherwise:

	1	2	...	13	14	15	16
1	0	1	...	0	0	0	0
2	1	0	...	0	0	0	0
...	0
13	0	0	...	0	1	1	1
14	0	0	...	1	0	0	0
15	0	0	...	1	0	0	0
16	0	0	...	1	0	0	0

Table 3-2: The adjacency matrix of an exemplary graph T ; the entries $A_{uv} = 0$ indicate that the vertex $u = v$ has no self-loop.

With the set of relational data at hand, we can now deal with drawing trees according to some specific drawing conventions, as defined in the next section.

3.3 Tree drawing

3.3.1 Drawing conventions

A *drawing convention* is a basic constraint of geometrical representations of nodes and edges. For example, in flow diagrams (Figure 2-8), vertices are drawn as boxes and edges as orthogonal chains consisting of horizontal and vertical lines. Drawing conventions vary depending on the field of application and thus involve many different details of the drawing. This section outlines a few central conventions for the visualization of parse trees.

Representations of vertices include boxes, circles, diamonds, parallelograms, ellipses, or filled dots (thicker than the edge lines). For our purposes vertices are drawn as boxes so that labels can be placed within. However, instead of displaying a simple label, this box can be used to place extracted features of the parse tree.

⁹ The WebLicht parsers make use of list notation in the XML structures of the annotated text corpus, as shown in 0, 5) and 6)

Since all tags in a parse tree carry a reference to the token ID, all flat features can be matched with the corresponding nodes in the parse tree. As shown in Figure 3-1, each node can display a PoS tag, token and lemma if a match is found and the feature type has been annotated in the text corpus. The ID is additionally displayed in order to track the alphabetical order of tokens.

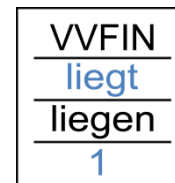


Figure 3-1:
The implemented representation of a vertex in parse trees

Common conventions for drawing edges include orthogonal, straight or bent lines. Straight lines or lines with a sharp bent are preferred in order to achieve parallel lines.

Along with conventions for edges and vertices, different options are available to arrange the elements of a tree in a two dimensional space. In graph drawing theory, a *graphical construction* is a grid if the relations between all elements are expressed by the same graphical component in a two dimensional plane [Bertin]. The following Figure 3-2 presents a set of grid conventions that use different notation types for the edges and vertices, and arrange the grid onto an ordered field:

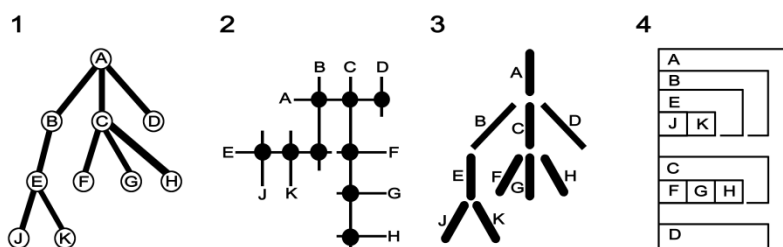


Figure 3-2: Grid types for trees with different ways to draw components and connections.

When visualizing trees according to the grid types (2) and (4), it would become difficult to visually grasp its relational structures if a tree contains a large number of hierarchies. For our purposes trees are drawn according to the common and visually comprehensible grid type (1).

3.3.2 Aesthetics and constraints of a tree drawing

A major factor that influences the usefulness of a drawn tree is its *readability*. Although all possible drawings contain the same information, the conveyance of the relational structures can be very different and strongly depends on the following properties of a drawing [Battista]:

- Bends
- Crossings (planarity)
- Positioning of vertices
- Edge Length
- Symmetry
- Area

Regarding the aesthetics **bends** and **crossings** it is desirable to produce drawings with as fewest bends and crossings as possible so that the readability of a tree is increased [Bhanji et al., Purchase et. al]. Additionally, these are directly influenced by the **positioning of the vertices**, as

demonstrated in Figure 3-3 below. Here, the drawing is a constituency parse tree of the processed sentence "The quick brown fox jumps over the lazy dog.". To emphasize the impact of the positioning of the vertices, Figure 3-3 shows the transitive closure of the tree $T = (V, E)$ with V being the extracted constituents and E the relations between them:

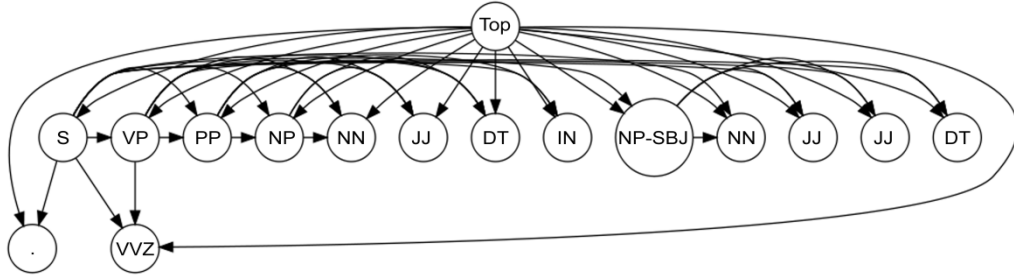


Figure 3-3: An exemplary tree T with a very low readability

The numerous overlapping edges and crossings make it impossible to comprehend the relational structures of the parsed tree. On the opposite, Figure 3-4 presents the reduced graph T' (the *transitive reduction* of T) with a high readability:

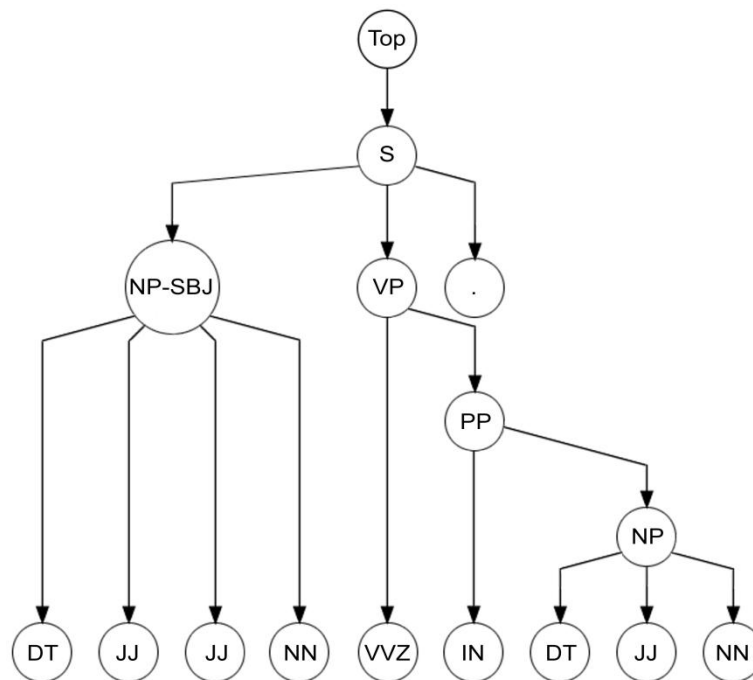


Figure 3-4: The reduced tree T' of T with a high readability

The tree T is drawn in such a way that the leaves are placed on the lowest line and the internal vertices with a specific depth are arranged on the same height in the drawing. The parallel edges are achieved by placing bends in between the edges (on the same x-coordinate as the node below) and preferably on the same height in order to achieve a maximum display of **symmetry**. This aesthetics is, as we see, directly influenced by positioning vertices and bends on the same height, by using as many parallel edges as possible, and furthermore by creating isomorphic subtrees that have the same drawing.

Regarding the aesthetics *area*, it is of utmost importance that a **spatial layout** is produced that covers an **area with minimal breadth**. Since parse trees can be very large, a drawing should not waste space on a computer screen (or printed medium) so that it is still viewable.

With all the central aesthetic criteria at hand, the following parameters for a *multi-objective optimization problem* can be formalized in order to produce a tree drawing with a maximum readability:

- minimization of the number of crossings
- minimization of the number of bends of edges
- minimization of the number of different gradients
- minimization of the covered surface
- maximization of displaying symmetries

Generally, drawing algorithms for graphs cannot satisfy all constraints, and thus deliver trade-off solutions [Battista]. However, since we deal with trees, an optimum can be achieved regarding all optimization objectives due to a specific drawing algorithm introduced below (Section 3.4.2).

3.4 Drawing algorithms for parse trees

This section introduces specialized algorithms that are suitable to produce tidy drawings of parse trees received from WebLicht. A natural way of visualizing these rooted trees is by constructing downward planar drawings.

In the first step of a drawing process a *layer assignment* is performed in which each vertex of a tree T is assigned to a layer $L_i \in \{L_1, \dots, L_h\}$ such that an edge (u, v) with $u \in L_i$ and $v \in L_j$ goes from layer L_i to a layer L_j below, with $i < j$.

In the general case of acyclic graphs, the goals during the layer assignment are to simultaneously produce a small number of layers, as few edges as possible that span large numbers of layers, and a balanced assignment of vertices to layers [Battista].

In the case of trees the layer assignment is simple: The number of layers is given by the maximum depth of a vertex in the tree. Therefore, a vertex with depth i is directly placed into layer L_i , thus each vertex can be assigned the y-coordinate $y(v) = -i$. The drawing starts with $v = \text{root}(T)$ by assigning v to L_0 and y-coordinate $y(v) = 0$, which is the top of the drawing. If T has more vertices than the root, the assignment continues with its children. Each child vertex w with depth $i + 1$ is then assigned to a new layer L_{i+1} below.

The edges of a parse tree can be drawn without crossings by ensuring that the *left-to-right relative order* of any two vertices v and w in layer L_i is the same order of their parents v' and

w' in layer L_{i-1} . As mentioned in Section 2.6.3, each vertex of a parse tree is indicated by a unique number (id) and the order of vertices is given by an increasing numbering.

Due to the layer assignment, all vertices have prescribed y-coordinates so that an algorithm for constructing a drawing only needs to compute the x-coordinates. An intuitive requirement is to position the x-coordinate of a parent vertex within the horizontal span of its children.

3.4.1 The "Layered-Tree-Draw" Algorithm

In order to construct a layered drawing of an n-ary rooted tree, the '*Layered-Tree-Draw*' Algorithm is present first. It is a basic recursive approach that makes use of a divide-and-conquer strategy to draw subtrees (divide) and their children (conquer) in the tree. The Algorithm 3-1 to this approach is defined as follows:

Algorithm 3-1: The "Layered-Tree-Draw" Algorithm

Input: Rooted n-ary tree T
Output: Layered drawing Γ of T

1. **Trivial case:** If T consists of only one vertex, output the trivial drawing.
2. **Divide:** Apply the algorithm recursively on each subtree T_i (e.g. in a left-to-right order).
3. **Conquer:** Draw each subtree T_i for $i = 2, \dots, m$ separately. Then, place the drawing of T_i to the right of the drawing of T_{i-1} so that their bounding boxes are not overlapping. Now, shift T_i to the left until the leftmost node in T_i has a distance of $d = 2$ units to the rightmost node in T_{i-1} .
Finally, the root r is positioned vertically one unit above and horizontally in the center of the drawing of a new subtree. If r only has one subtree, then place the root directly above that subtree.

The term *bounding box* (mentioned in the conquer step of Algorithm 3-1) refers to a rectangle that embraces the drawing of a subtree. According to this approach, an exemplary rooted tree is presented in the next Figure 3-5.

The properties of a drawing Γ produced by the '*Layered-Tree-Draw*' Algorithm are given as follows:

- Γ clearly **encodes the depth levels** by using a layered layout in which each vertex with depth i is placed on a layer L_i with y-coordinate $y(v) = -i$.
- Γ is **planar**, and consists of strictly downward straight lines.
- Γ contains **no crossings** since the left-to-right order of the children of each vertex is preserved.
- **Minimum horizontal and vertical distance** of at least 1 unit between any two vertices.
- The area covered by Γ is $O(n^2)$.
- Every parent vertex is placed horizontally in the center of a subtree.

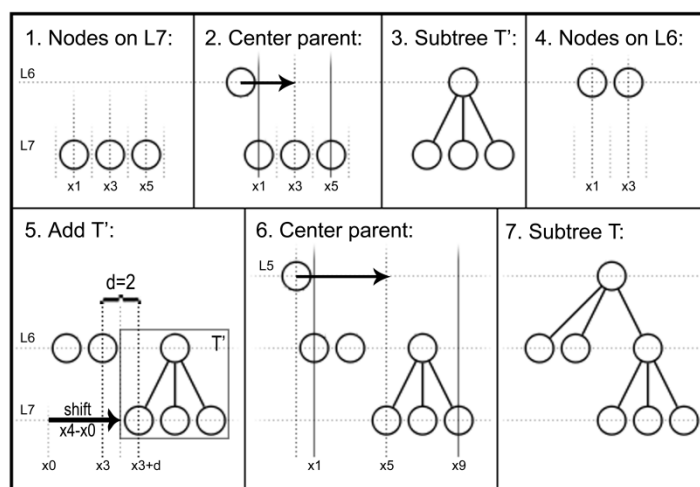
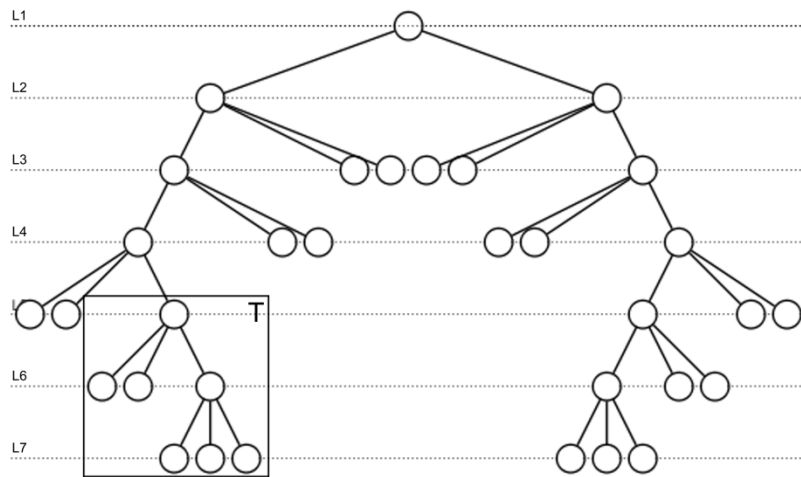


Figure 3-5: An exemplary rooted tree whose nodes are placed by the *Layered-Tree-Draw* Algorithm along the x-axis and by the layer assignment along the y-axis. The steps beneath the drawing describe the construction of subtree T .

As we can see in Figure 3-5, a major disadvantage of Algorithm 3-1 is that the produced drawing spans a large breadth, and even larger trees lead to an exponential growth in the width of the drawing.

Let us briefly describe the steps of constructing the subtree T : Due to the recursive nature of the divide-and-conquer approach, the algorithm traverses down to the lowest level L_7 (1) and positions the nodes along the x-axis with a distance of $d = 2$ towards each other. Then, in the same conquer step, the parent vertex is centered above its children at the x-coordinate x_3 (2). In the conquer step of the previous step during the recursive run, the leaf nodes on layer L_6 are drawn (4). In (5) the subtree T' can be added and properly shifted to the right by so many units that the leftmost node in T' has a distance of $d = 2$ units to the rightmost node that is drawn at the x-coordinate x_3 . Note, that the vertex at x_3 could equally contain a subtree to which T' would be placed with $d = 2$ to the rightmost node in that subtree. Finally, in (6) the parent of the nodes on L_6 is positioned in the center of the resulting drawing which spans from the x-coordinate x_1 to x_9 . After that Algorithm 3-1 continues with L_5 analogously.

3.4.2 The "Reingold & Tilford" Algorithm

Reingold and Tilford modified the *Layered-Tree-Draw* Algorithm to produce a layout that makes smarter use of space, maximizes the density and still displays symmetries in the drawing [Reingold&Tilford]. Since their original algorithm processes binary trees only, [Walker] extended it to draw n-ary rooted trees as well. In order to perform the drawing in linear time, [Buchheim et al.] further improved Walker's algorithm.

The '**Reingold & Tilford** Algorithm' follows the same divide-and-conquer strategy as the 'Layered-Tree-Draw' Algorithm (Section 3.4.1). However, at each conquer step it makes use of a local optimization heuristic in order to reduce the width, and centers a parent vertex horizontally with regards to its children. The modified Algorithm 2 is defined as follows:

Algorithm 3-2: The "Reingold & Tilford" Algorithm

Input: Rooted n-ary tree T
 Output: Compact layered drawing Γ of T

1. **Trivial case:** If T consists of only one vertex, output the trivial drawing.
2. **Divide:** Apply the algorithm recursively on each subtree T_i (e.g. in a left-to-right order).
3. **Conquer:** First, draw each subtree T_i for $i = 2, \dots, m$ separately. Then, place the drawing of T_i to the right of the drawing of T_{i-1} . Now, shift T_i to the left until its left contour has a horizontal distance of $d = 2$ to the right contour of T_{i-1} . Finally, the root r is positioned vertically one unit above and horizontally in the center of its children. If r only has one subtree, then position the root directly above that subtree.

The modification in Algorithm 3-2 compared to the "Layered-Tree-Draw" Algorithm is the positioning of subtrees according to their contours. A *left contour* of a tree T with height h is defined as the sequence of vertices v_0, \dots, v_h such that each v_i is the leftmost vertex with depth i in T . The right contour is defined analogously. In the conquer step the right contour of the left subtree T_{i-1} and the left contour of the right subtree T_i need to be followed simultaneously while ensuring that both contours keep the minimum distance of two units. This compacting step is visualized in the following Figure 3-6:

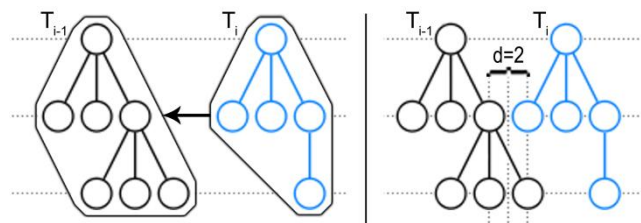


Figure 3-6: Compacting subtrees along their contours during the conquer step in the "Reingold&Tilford" Algorithm

The basic steps of the implementation of the "Reingold & Tilford" Algorithm consists of two traversals of the input tree T : In the first traversal the horizontal shifts of each child vertex relative to its parent vertex are determined. Then, in the second traversal the x-coordinates of the

vertices are computed by accumulating the shifts on the path from each vertex to the root. Finally, the same graph as in Figure 3-6 is obtained according to the "Reingold & Tilford" Algorithm:

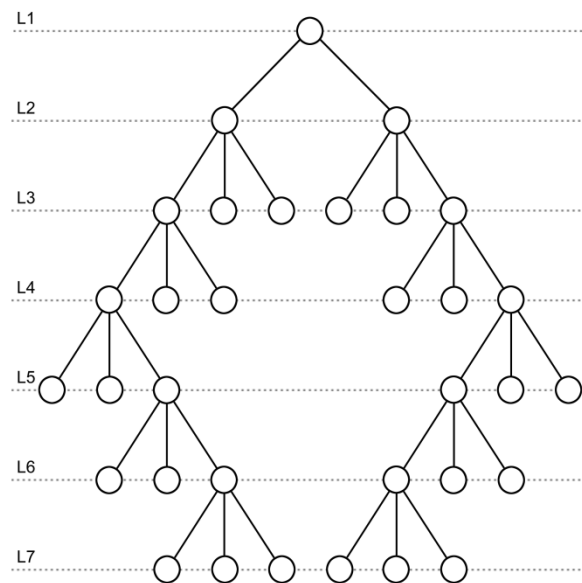


Figure 3-7: The exemplary tree from Figure 3-5 drawn by the "Reingold&Tilford" Algorithm.

Figure 3-7 shows an aesthetically pleasing drawing of a rooted tree constructed by the "*Reingold & Tilford*" Algorithm. Note, that the y -coordinates are calculated by the layer assignment as described in the introduction of Section 3.4.

To conclude this section, the properties of a drawing Γ according to the "Reingold & Tilford" layout are given as follows:

- Γ clearly encodes the depth level
- Γ is **planar** with strictly downward straight lines.
- Γ contains **no crossings** since the left-to-right order of the children of each vertex is preserved.
- Γ is **compact**
- A **minimum distance of 1** unit between any two vertices
- Γ covers an area of $O(n^2)$
- Γ **preserves symmetry** by producing simply isomorphic structures that have congruent drawings, up to a translation in the drawing.

3.5 Visualization of structural features in RapidMiner

This section presents concepts and results of the implemented **'Visualize and Label Parse Trees'** operator for RapidMiner. The idea is to provide linguistic research experts a visual insight into structural relations of a given sentence based on the annotations of a constituency or dependency parser.

In order to visualize the parse tree of a sentence, a given sentence has to be annotated with the **'WebLicht Feature Annotator'** with the tool chain containing either a constituency or a dependency parse service from WebLicht (see Section 2.5). After that, the **'WebLicht TCF to ExampleSet'** operator has to be employed in order to extract the **'tree string'** which encodes the parse tree (see Section 2.6.2). The tree string can then be forwarded to the **'Visualize and Label Parse Trees'** operator¹⁰.

Figure 3-8 depicts a concept for presenting drawings of both constituency and dependency parse trees:

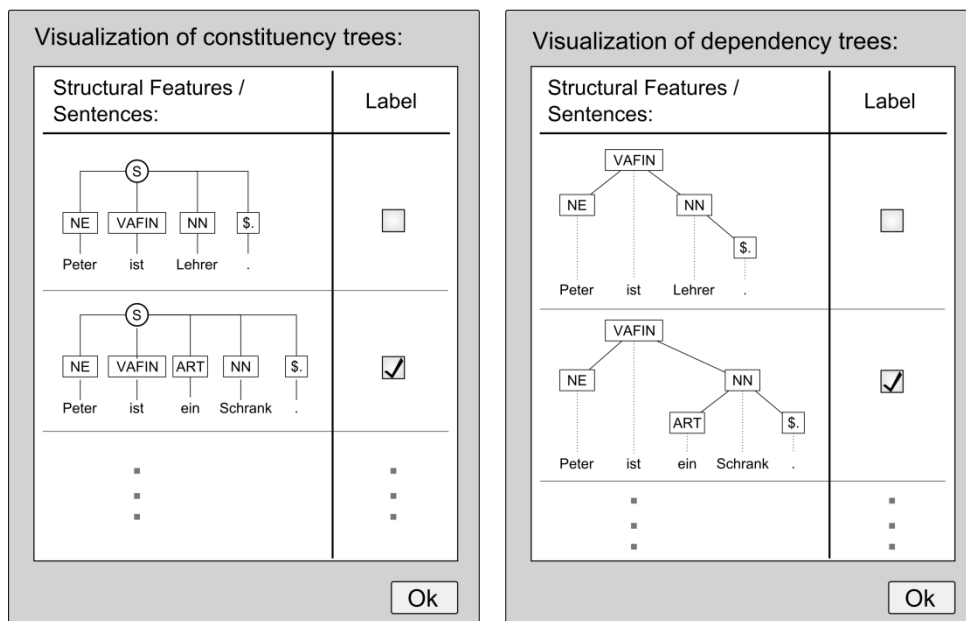


Figure 3-8: A conceptual dialog presenting lists of visualized parse trees that can be labeled.

Basically, the **'Visualize and Label Parse Trees'** operator reconstructs parse trees from **'tree strings'**. Since "[", "]" and "," are reserved characters in the bracket notation, the reconstruction is achieved by simply parsing "words" that are delimited by these special characters. Another feature of this operator is the option to assign a label to each sentence.

¹⁰ Alternatively, the parse trees could have been forwarded as serialized objects, where each object would be represented as a sequence of bytes. However, by encoding/decoding trees to strings in bracket notation, these **'tree strings'** can directly be viewed and used in learning methods.

The following Figure 3-9 presents the visualization of a dependency parse tree of an exemplary German sentence:

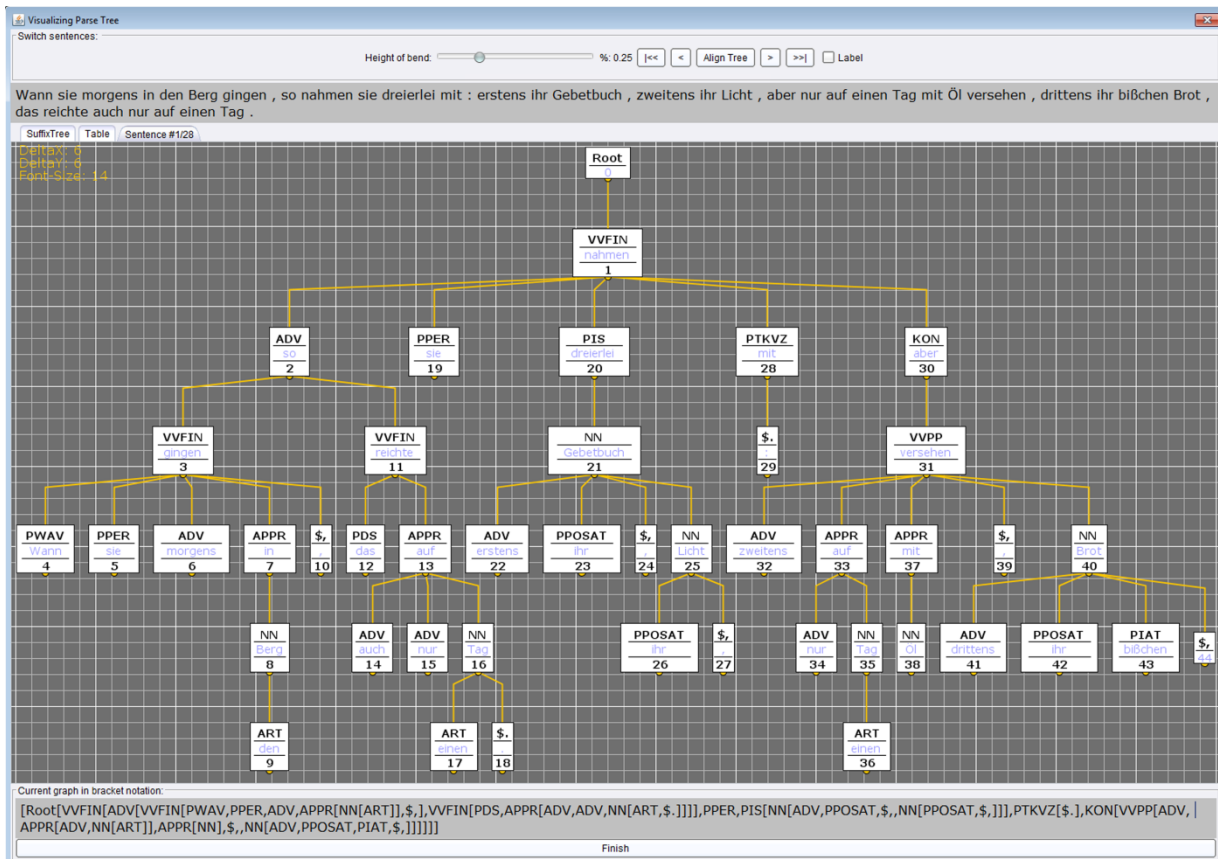


Figure 3-9: A tidy and compact drawing of an exemplary dependency parse tree for an exemplary German sentence.

Compared to the concept in Figure 3-8 the leaf nodes are not placed on the lowest line in the drawing. Doing so would not yield a compact "Reingold & Tilford" layout (as shown in Figure 3-7), and as such the resulting tree would span even more horizontal space. Also, instead of drawing a list of trees beneath one another (which would be difficult to navigate), only one tree at a time is shown in a panel. By providing intuitive controls the user can quickly rotate through the list of trees.

This section is concluded with a brief presentation of the performance of this operator: The implemented visualization operator runs in about *linear runtime*. The visualization process includes the parsing of the encoded 'tree strings', the internal construction of trees, the caching of all parse trees, and the final visualization of the first tree. For 1.000 encoded strings (with a random length between 500 and 1000 characters) the operator uses roughly one second on a mobile platform equipped with an Intel i7-3630QM@2.4Ghz CPU, with 2GB of limited memory space, while RapidMiner running on a single core.

The full documentation of the implemented RapidMiner operator 'Visualize and Label Parse Trees' is given in the Appendix A.3. A class diagram is added in Appendix A.3.3 that depicts the major classes involved in the implementation.

Chapter 4

Machine Learning in Text Corpora

This chapter presents machine learning methods that are intended to be used for pattern detection in text corpora. A text corpus usually consists of sentences given as text strings. The central idea is to detect patterns in linguistic features like tokens, lemmas, PoS-tags or 'tree strings' that have previously been obtained by the feature extraction tool (Section 2.6).

For the analysis of text corpora *kernel-based learning methods* (KMs) are introduced in Section 4.1 which provide a powerful approach to efficiently detect nonlinear relations without the problem of *overfitting* [Shawe-Taylor & Cristianini]¹¹. Since classification tasks are the main focus in this work, KMs are combined with the prominent *support vector machine* (SVM) (Section 4.2). With this machine learning concept, the following kernel methods are introduced that come into question for the analysis of text corpora:

- **String Subsequence Kernel** (Section 4.3)
- **Bag of Words Kernel and N-gram Kernel** (Section 4.4)
- **Spectrum Kernel** (Section 4.5)
- **Tree Kernel** (Section 4.6)
- **Fast Kernel for String and Tree Matching** (Section 4.7)

Figure 4-1 depicts the process in which preprocessed linguistic data can be used in machine learning:

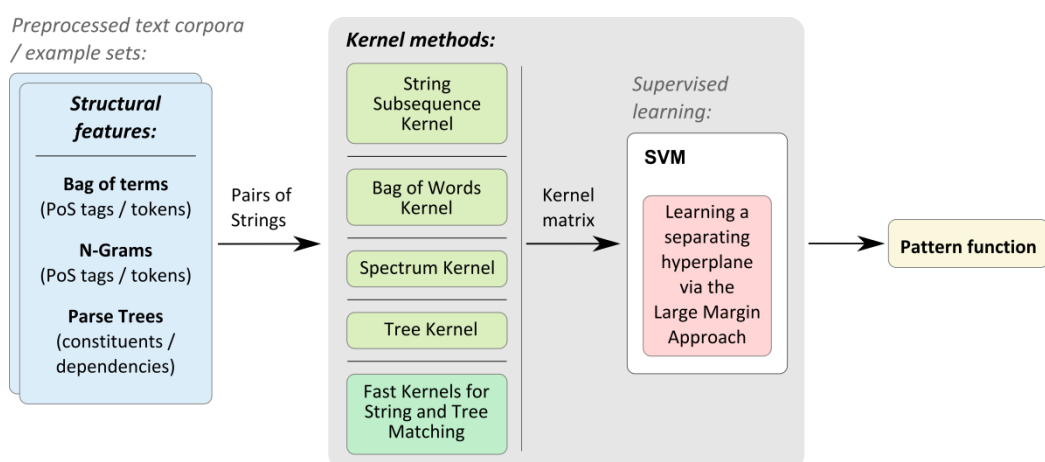


Figure 4-1: A machine learning framework with different kernel methods used by the SVM in a supervised learning surrounding

¹¹ Overfitting occurs when a learned model is too complex, that is too many parameters (relative to the number of examples in the training set) are used to memorize data rather than to learn to generalize from trend. The predictive performance of such a model is usually poor on unseen data since it exaggerates minor fluctuations in the data [StatLearn]

In order to feed a KM with examples, the extracted features regarding each sentence are first concatenated to text strings. Then, a chosen KM computes the similarity values between pairs of strings. The result is a so called *kernel matrix* (Section 4.1) that consists of similarity values between all examples, which is then forwarded to the SVM.

A particular focus in this chapter is put on the '**Fast Kernel Method for String and Tree Matching**' (Section 4.7) [Vishwanathan & Smola] which is implemented and made use of during the machine learning experiments in Chapter 5. Compared to classic string kernels, this 'Fast String Kernel' method computes string kernels in *linear time* in the size of the arguments (Section 4.7.3), independent of any weights that can be associated with matching substrings. In this context, Section 4.7.4 presents various weight functions that allow a different emphasis of matching substrings. Finally, in Section 4.8 various aspects are outlined that concern the implementation of the 'Fast String Kernel' operator for RapidMiner.

4.1 Kernel Methods (KMs)

Kernel methods (KMs) provide a powerful way of detecting nonlinear relations in data $d_1, \dots, d_n \in D$ that is transformed to feature vectors living in an m -dimensional Euclidian *feature space* \mathcal{F} (with m possibly being infinite). General types of relations analyzed by KMs are clusters, principal components, correlations and classifications.

In the case of character strings gained from text corpora, these features cannot readily be described by explicit feature vectors. Constructing a module that transforms data to feature vectors in \mathcal{F} is a difficult problem since important information can get lost during that process. It is clear that the transformation of features plays a key role in the effectiveness of detecting patterns [Lodhi et al.].

Furthermore, the explicit computation of the coordinates of features in \mathcal{F} has a very high computational cost which is implicit in the number of dimensions of \mathcal{F} . To circumvent this problem, *kernel-based learning methods* (KMs) offer an effective alternative. The building block of a KM is a function known as the *kernel function* (or short *kernel*) K that efficiently computes the inner product between mapped examples in the feature space:

$$\begin{aligned} \Phi : D &\rightarrow \mathcal{F}, \\ (d_i, d_j) &\mapsto K(d_i, d_j) = \langle \Phi(d_i), \Phi(d_j) \rangle = \Phi(d_i) \cdot \Phi(d_j) \end{aligned}$$

The function Φ maps a feature $d_i \in D$ into some feature space F . In this work, these features are given as sets of linguistic units. For a vector space \mathbb{R}^n the inner product $\langle \cdot, \cdot \rangle$ is defined as:

$$\langle x, y \rangle = x \cdot y = \sum_{i=1}^n x_i y_i$$

The inner product is an appropriate measure for the *similarity* between two data items. More generally, an *inner product space* is a vector space X over the real values \mathbb{R} if there exists a real-valued symmetric bilinear map $\langle \cdot, \cdot \rangle$ which is linear in each argument and satisfies $\langle x, x \rangle \geq 0$. This bilinear map is known as the *dot* or *scalar* product [Shawe-Taylor & Cristianini].

The mapping of features $d_i \in D$ to feature vectors $x_i \in X \subseteq \mathcal{F}$ has not yet been specified, but this is actually not necessary since only the inner products need to be computed:

$$K(d_i, d_j) = \Phi(d_i) \cdot \Phi(d_j)$$

This mathematical shortcut, often known as *kernel trick*, allows learning in *implicit* feature space without ever computing coordinates of data points in that space. In the context of classification, the learning refers to a linear decision function represented by a weight vector in \mathcal{F} . This weight vector is a linear combination of feature vectors of the training points. For some point z we can look up a function f_z via $f_z(x) = \langle x, z \rangle$ and thus find the corresponding weight vector. Therefore, finding the weight vector is equivalent to identifying the corresponding element in feature space \mathcal{F} [Shawe-Taylor & Cristianini].

In the optimal case that the data becomes linearly separable in \mathcal{F} , a linear classifier like the support vector machine (see Section 4.2) can learn a linear decision function f with an associated weight vector \vec{w} . Figure 4-2 and Figure 4-3 depict a simply case: Instead of mapping each feature to this space we use the kernel trick to directly compute the inner product of two features according to some *kernel function*.

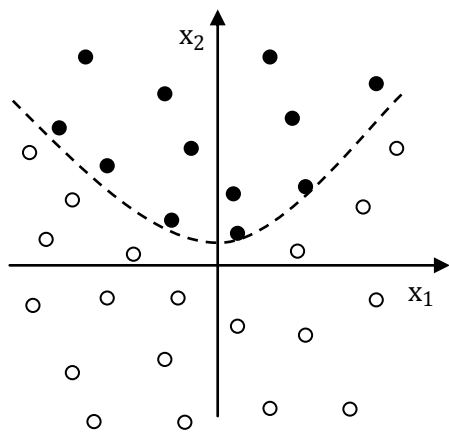


Figure 4-2: An exemplary set $X \subseteq \mathbb{R}^2$ of non-linear separable examples

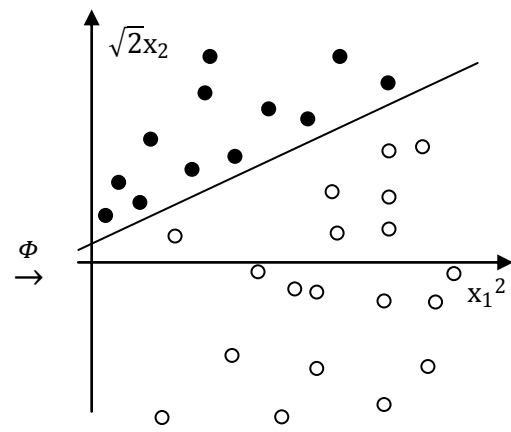


Figure 4-3: Data set X mapped to the feature space \mathcal{F} after applying a mapping function Φ

The learned decision function, also called hypothesis, can then be applied to previously unseen examples in the same vector space in order to make predictions.

A major advantage of KMs is that this approach allows decoupling algorithms for similarity computation from the specification of a feature space. Numerous kernels were developed to

compute similarities of arbitrary data types. Of particular importance in this work are kernels that are able to compare text strings.

Alternatively, the similarity computed by $K(x, y)$ can be seen as the angle between two vectors of the mapped inputs x and y which is also known as the *cosine similarity*. The cosine between two vectors can be derived from the Euclidean dot product $\langle x, y \rangle = x \cdot y = \|x\| \cdot \|y\| \cos(\theta)$.

$$\cos(\theta) = \frac{x \cdot y}{\|x\| \cdot \|y\|} \quad \text{with} \quad \theta = \begin{cases} 0, & \text{if } \cos(\theta) = 1 \text{ and } x \cdot y = \|x\| \cdot \|y\| \\ \frac{\pi}{2}, & \text{if } \cos(\theta) = 0 \text{ and } x \cdot y = 0 \end{cases}$$

For text matching, the angle between vectors of similar strings is small with a possible minimum of zero, while dissimilar strings have vectors that are orthogonal towards each other.

The central data structure of all kernel-based algorithms that holds the similarities of all compared examples is the so called *Gram matrix*, or simply *kernel matrix* [Shawe-Taylor & Cristianini]. The Gram matrix G is defined as an $m \times m$ matrix whose entries are the similarities/inner products between each two examples:

$$G_{ij} = \langle \Phi(d_i), \Phi(d_j) \rangle = K(d_i, d_j)$$

G can be displayed as follows:

K	1	2	...	m
1	$K(d_1, d_1)$	$K(d_1, d_2)$...	$K(d_1, d_m)$
2	$K(d_2, d_1)$	$K(d_2, d_2)$...	$K(d_2, d_m)$
...
m	$K(d_m, d_1)$	$K(d_m, d_2)$...	$K(d_m, d_m)$

In the case that the features d_i and d_j originate from the same data set D the gram matrix is symmetric due to $G_{ij} = G_{ji}$ which means that the transposed matrix G^T equals G . Furthermore, the Gram matrix is *positive semi-definite*. A symmetric matrix $A \in \mathbb{R}^{n \times n}$ is positive semi-definite, if its eigenvalues are all non-negative. This is true only if $x'Ax \geq 0$, with the vectors $x \in \mathcal{F}$.

Lastly, when comparing features like two strings s and t then their lengths directly impact the similarity value that is computed by $K(s, t)$. Therefore, a given kernel function is normalized as follows [Shawe-Taylor & Cristianini]:

$$\hat{K}(d_i, d_j) = \left\langle \frac{\Phi(s)}{\|\Phi(s)\|}, \frac{\Phi(t)}{\|\Phi(t)\|} \right\rangle = \frac{K(s, t)}{\sqrt{K(s, s) \cdot K(t, t)}}$$

4.2 The Support Vector Machine (SVM)

The SVM is a very universal learner due to the fact that its integral kernel function, or simply called *kernel*, can be exchanged like a simple "plug-in" [Joachims2000]. The SVM implements a large margin approach in which a separating hyperplane is optimally placed between two classes of examples that have been mapped by some KM to a feature space beforehand. The idea is then to maximize a large margin between two classes in order to obtain a model that generalizes well to unseen data without having learned too closely to the set of training examples (also known as *overfitting*). The classification is then done as follows: By applying the learned model on a previously unseen example the according decision function determines on which side of the hyperplane a new example lies and associates the according class to that example. In the context of classification, the SVM results in a non-probabilistic binary linear classifier.

Since the SVM performs *supervised learning* the sentences of a text corpus have to be labeled according to some classes. However, the SVM is not restricted to *binary classification*, but also allows *multiclass classification*. For instance, for N classes N binary classifications can be performed in which each SVM considers in a *one-vs.-rest strategy* one class as the positive examples and separates it from the $N - 1$ other classes that are treated as negative examples. Afterwards new examples are predicted according to the class with the largest confidence, that is where distance to the corresponding hyperplane is the largest [Joachims 2000].

In supervised learning the SVM analyzes previously labeled training data (e.g. with the labels $y \in \{-1,1\}$ in binary classification) to infer a function for predicting the labels of new examples (either $y = 1$ or $y = -1$). The discrepancy between the true labels and the predicted label is measured by a loss function [Shawe-Taylor]. This loss is called the *classification error*. Basically, in supervised learning the SVM iteratively predicts examples of a training set while adjusting the large margin until the classification error converges to a minimum.

The advantage of the SVM lies in a low generalization error¹² if a large margin can be determined. That is, depending on the separability of a training set, the SVM generalizes well enough when classifying unseen examples.

In the next section the ideal case is described where examples are linearly separable, and in Section 4.2.3 the realistic scenario is shown where non-separable data is used for learning.

¹² The generalization error is defined as mean-square error $\frac{1}{n} \sum_{i=1}^n (\hat{y} - y)^2$, for the examples $i = 1, \dots, n$, with the predicted label \hat{y} and the true label y . The generalization is a theoretical concept to measure the distance between the error on the training set and the test set and is averaged over the entire set of possible training data that can be generated after each iteration of the learning process. This theoretical model assumes the true probability distribution of the examples by means of a hypothetical function that predicts the labels without error.

4.2.1 The linear separable case

Regarding the linear separable case the SVM learns a decision function that accurately separates the data according to their associated labels [Burges]. Figure 4-4 shows a set of training examples residing in feature space $\mathcal{F} = \mathbb{R}^2$ that can be separated by a hyperplane H in two classes. In space $\mathcal{F} = \mathbb{R}^n$ a hyperplane is given by the normal vector $\vec{w} \in \mathbb{R}^n$ and some bias $b \in \mathbb{R}$, as follows:

$$H = \{x | \langle \vec{w}, \vec{x} \rangle + b = 0\}$$

$\langle \vec{w}, \vec{x} \rangle$ is again the inner product, with $\langle \vec{w}, \vec{x} \rangle = \sum_{i=1}^n w_i \cdot x_i$.

Any point x which would lie on H satisfies $\vec{w} \cdot \vec{x} + b = 0$,

where \vec{w} is the normal to H . The perpendicular distance from H to the origin is given by $\frac{|-b|}{\|\vec{w}\|}$,

while $\|\vec{w}\| = \sqrt{\langle \vec{w}, \vec{w} \rangle}$ is the Euclidean norm of \vec{w} . Let d_+ (d_-) be the shortest distance from H to a positive (negative) example. The working principle of the support vector algorithm is to determine the separating hyperplane H in such a way that a maximum margin $d_+ + d_-$ can be obtained. As shown in Figure 4-4 the hyperplanes H_1 and H_2 delimit the margin. With the same normal vector \vec{w} these planes run parallel to H . By scaling \vec{w} and b , the hyperplanes H_1 and H_2 can be expressed in normal form as follows:

$$H_1: \langle \vec{w}, \vec{x} \rangle + b = -1 \quad \text{and} \quad H_2: \langle \vec{w}, \vec{x} \rangle + b = +1$$

In the case of linear separability, all examples x_i satisfy the following conditions:

$$\langle \vec{w}, \vec{x} \rangle + b \geq +1 \quad \text{for } y_i = +1 \quad (1)$$

$$\langle \vec{w}, \vec{x} \rangle + b \leq -1 \quad \text{for } y_i = -1 \quad (2)$$

Points that lie between origin and H_1 or lie on H_1 satisfy the inequality (2) and are assigned the class -1 (white points). Points on H_2 or beyond H_2 receive the class label $+1$ (black points). By construction, no points lie in between H_1 and H_2 . The perpendicular distance from H_1 (H_2) to the origin amounts to $\frac{|-1+b|}{\|\vec{w}\|}$ ($\frac{|-1-b|}{\|\vec{w}\|}$). Hence, the width of the margin is given by $\frac{2}{\|\vec{w}\|}$. By formalizing the margin, we can determine the hyperplanes H_1 and H_2 . Maximizing $\frac{2}{\|\vec{w}\|}$ is achieved by minimizing $\|\vec{w}\|$ with regard to the constraints (1) and (2). By combining these constraints, we obtain a single inequation:

$$y_i(\langle \vec{w}, \vec{x}_i \rangle + b) - 1 \geq 0 \quad \forall i \in \{1, \dots, l\} \quad (3)$$

All points that satisfy (3) are lying on one of the hyperplanes H_1 or H_2 and are denoted by *support vectors* (in Figure 4-4 these points are drawn with an extra circle). By solving \vec{w} and b ,

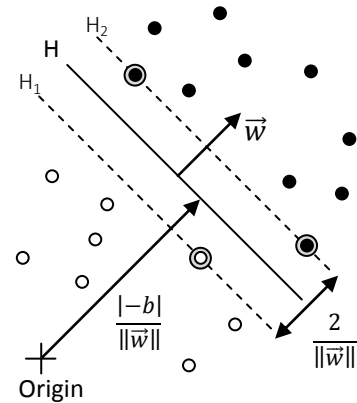


Figure 4-4: Linear separating hyperplanes for the separable case. Encircled points represent the support vectors.

we obtain the separating hyperplane. By means of a simple decision function, the SVM can then classify new examples:

$$f(\vec{x}, \vec{w}, b) = \text{sgn}(\langle \vec{w}, \vec{x} \rangle + b) \quad (4)$$

In order to compute \vec{w} and b , the given minimization problem under the constraints (3) can be turned into a *Lagrangian formalization*. A Lagrangian function is obtained by subtracting the sum of constraints from a target function. Here, $\frac{1}{2} \|\vec{w}\|^2$ is used as the target function since we want to minimize $\|\vec{w}\|$. For each of the l constraints (3) the Lagrangian formalization requires to introduce *Lagrange multipliers* $0 \leq \alpha_i \leq \infty$:

$$\alpha_i y_i (\langle \vec{w}, \vec{x}_i \rangle + b) - \alpha_i \geq 0 \quad \forall i \in \{1, \dots, l\} \quad (5)$$

By summarizing these l constraints, we obtain the Lagrangian function of the problem, also known as the *primal optimization function* L_P :

$$L_P = \frac{1}{2} \|\vec{w}\|^2 - \sum_{i=1}^l \alpha_i y_i (\langle \vec{w}, \vec{x}_i \rangle + b) + \sum_{i=1}^l \alpha_i \quad (6)$$

In order to compute the minimum of L_P , the gradients of L_P regarding the unknowns \vec{w} and b need to be determined as follows:

$$\frac{dL_P}{d\vec{w}} \stackrel{!}{=} 0 \quad \Leftrightarrow \quad \vec{w} = \sum_{i=1}^l \alpha_i y_i \vec{x}_i \quad (7)$$

$$\frac{dL_P}{db} \stackrel{!}{=} 0 \quad \Leftrightarrow \quad \sum_{i=1}^l \alpha_i y_i = 0 \quad (8)$$

L_P is a convex quadratic programming problem due to the fact that the points that satisfy the constraints (7) and (8) form a convex set [Burges].

An easier optimization problem is obtained by turning the primal form into the so called *Wolfe dual* as it only requires to solve the Lagrangian multipliers $\alpha_i \geq 0$, as shown below. The Wolfe dual has the property that a *maximum* of L_P in (6), subject to a set of constraints $C_1 = \{\alpha_i | \alpha_i \geq 0, i \in l\}$, occurs for the same values of \vec{w} , b and α , as the *minimum* of L_P , subject to constraints $C_2 = \{\alpha_j | \alpha_j \geq 0, j \in l\}$ [Fletcher]. Therefore, instead of minimizing L_P with regard to \vec{w} and b we can now maximize L_P . By substituting (7) and (8) in L_P we obtain the *dual optimization function* L_D :

$$L_D = \sum_{i=1}^l \alpha_i - \frac{1}{2} \sum_{i=1}^l \sum_{j=1}^l \alpha_i \alpha_j y_i y_j \langle \vec{x}_i, \vec{x}_j \rangle \quad (9)$$

After solving the maximization problem L_D the normal \vec{w} to a hyperplane H can be determined by simply inserting the α_i in constraint (7). For any $\alpha_i > 0$ the according points x_i are the support vectors, whereas for $\alpha_i = 0$ the points satisfying (3) lie on one side of the margin defined by H .

The advantage of using a Lagrangian function is the operational simplicity for the calculation of a possible decision function (4). With the given maximization problem L_D , the training data can now be given as inner products between vectors which is the essential property of *kernel methods*, as presented in Section 4.1.

4.2.2 Karush-Kuhn-Tucker conditions

The so called Karush-Kuhn-Tucker (KKT) conditions which are basically a generalization of the Lagrangian multipliers are necessary conditions to guarantee an optimal solution of the non-linear optimization problem [Fletcher]. In the case of the primal optimization function L_P there is an optimal solution since all side constraints are linear. Furthermore, the KKT conditions are sufficiently fulfilled since the objective function $\frac{1}{2} \|\vec{w}\|^2$ is convex and all side conditions yield a convex feasible region:

$$\frac{dL_P}{d\vec{w}_v} = \vec{w}_v - \sum_{i=1}^l \alpha_i y_i \vec{x}_{iv} = 0, \text{ with } \dim v = 1, \dots, d \quad (10)$$

$$\frac{dL_P}{db} = \sum_{i=1}^l \alpha_i y_i = 0 \quad (11)$$

$$y_i (\langle \vec{w}, \vec{x}_i \rangle + b) - 1 \geq 0 \quad i = 1, \dots, l \quad (12)$$

$$\alpha_i \geq 0 \quad \forall i \in \{1, \dots, l\} \quad (13)$$

$$\alpha_i (y_i (\langle \vec{w}, \vec{x}_i \rangle + b) - 1) = 0 \quad \forall i \in \{1, \dots, l\} \quad (14)$$

The solution of the optimization problem of the SVM is equivalent to the solution of the set of KKT conditions. This approach is a starting point for algorithms that solve the linear restricted, quadratic convex problem. Details about nonlinear programming are given in [Fletcher]. Note that solving the Lagrangian function only yields the normal vector \vec{w} , but the bias b is not calculated explicitly. However, the bias can be determined via condition (14) by simply using an arbitrary example \vec{x}_i that is a support vector in the found solution (with $\alpha_i \neq 0$). According to [Burges] it is recommended to finally average over the values b from all equations with such examples.

4.2.3 The non-separable case

Realistic data sets of two classes are usually not separable by a linear hyperplane since many examples (either as outliers or due to errors in the training set) may lie on the side that belongs to the other class. In such a case the previously presented SVM with a separating hyperplane would fail to deliver valid solutions. In order to retain this concept, the constraints (1) and (2) are relaxed by integrating slack variables ξ_i into the description of the hyperplanes H_1 and H_2 :

$$\langle \vec{w}, \vec{x} \rangle + b \geq +1 - \xi_i \quad \text{for } y_i = +1 \quad (15)$$

$$\langle \vec{w}, \vec{x} \rangle + b \leq -1 + \xi_i \quad \text{for } y_i = -1 \quad (16)$$

$$\xi_i \geq 0 \quad \forall i \quad (17)$$

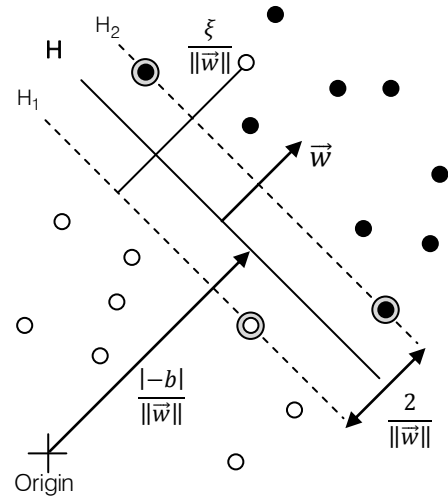


Figure 4-5: Hyperplanes in the non-separable case. Encircled points represent the support vectors.

The slack variables only appear in those equations where the corresponding example is on the side of the opposite class, as shown in Figure 4-5. This is reflected in equations (15) and (16) if the ξ_i exceeds the value one. The upper bound of the training error is then given as $\sum_{i=1}^n \xi_i$. By introducing a penalty factor C this error sum can be weighted and the resulting product added to the *objective function* which results to:

$$\frac{1}{2} \|\vec{w}\|^2 + C \left(\sum_{i=1}^n \xi_i \right)^k$$

A high value for factor C increases the weight for the penalty, while for a constant $k \geq 1$ we obtain a convex programming problem. Setting $k = 2$ yields a quadratic programming problem, and $k = 1$ has the advantage that neither the slack variables ξ_i nor the Lagrangian multipliers appear in the Lagrangian function, thus allowing us to continue to use the dual problem L_D :

$$L_D = \sum_{i=1}^l \alpha_i - \frac{1}{2} \sum_{i=1}^l \sum_{j=1}^l \alpha_i \alpha_j y_i y_j \langle \vec{x}_i, \vec{x}_j \rangle \quad (18)$$

Nonetheless, a new constraint has to be added, in which C is the upper bound for the Lagrangian multiplier α :

$$0 \leq \alpha_i \leq C \quad (19)$$

Furthermore, in order to find a maximum for L_D , the constraint (8) has to be satisfied, as well.

Finally, the solution of the separating hyperplane is given by the normal vector, with 'SV' indicating the number of support vectors:

$$\vec{w} = \sum_{i=1}^{SV} \alpha_i y_i \vec{x}_i \quad (20)$$

In order to guarantee a solution for the optimization problem, the following necessary KKT conditions need to be satisfied with regard to the primal problem L_P :

$$L_P = \frac{1}{2} \|\vec{w}\|^2 + C \sum_{i=1}^n \xi_i - \sum_{i=1}^l \alpha_i \{y_i (\langle \vec{w}, \vec{x}_i \rangle + b) - 1 + \xi_i\} - \sum_{i=1}^l \mu_i \xi_i \quad (21)$$

Furthermore, new Lagrangian multipliers μ_i are introduced in order to enforce the positivity of ξ_i . Therefore, the KKT conditions are given as follows:

$$\frac{dL_P}{d\vec{w}_v} = \vec{w}_v - \sum_{i=1}^l \alpha_i y_i \vec{x}_{iv} = 0 \quad \text{with } \dim v = 1, \dots, d \quad (22)$$

$$\frac{dL_P}{db} = \sum_{i=1}^l \alpha_i y_i = 0 \quad (23)$$

$$\frac{dL_P}{d\xi_i} = C - \alpha_i - \mu_i = 0 \quad (24)$$

$$y_i (\langle \vec{w}, \vec{x}_i \rangle + b) - 1 + \xi_i \geq 0 \quad i = 1, \dots, l \quad (25)$$

$$\xi_i \geq 0 \quad \forall i \in \{1, \dots, l\} \quad (26)$$

$$\alpha_i \geq 0 \quad \forall i \in \{1, \dots, l\} \quad (27)$$

$$\mu_i \geq 0 \quad \forall i \in \{1, \dots, l\} \quad (28)$$

$$\alpha_i (y_i (\langle \vec{w}, \vec{x}_i \rangle + b) - 1 + \xi_i) = 0 \quad \forall i \in \{1, \dots, l\} \quad (29)$$

$$\mu_i \xi_i = 0 \quad \forall i \in \{1, \dots, l\} \quad (30)$$

Analogously to the separable case (Section 4.2.1), the bias b can be determined via the conditions (29) and (30). Combining condition (24) with (30) shows that $\xi_i = 0$ if $\alpha_i < 0$ since $C - \alpha_i = \mu_i$, with $\mu_i > 0$. Thus, any example for which $0 < \alpha_i < C$ is true can be chosen to calculate b .

4.3 The String (Subsequence) Kernel

The string subsequences kernel (SSK) considers the number of subsequences shared by two strings [Lodhi et al.]. These strings are a finite sequence of symbols that do not need to be of the same length. Intuitively, the SSK can be understood as a function to measure the similarity between pairs of strings. The more substrings are in common, the more similar the strings are.

We start by defining the feature space as $\mathcal{F}_n = \mathbb{R}^{\Sigma^n}$ where Σ^n is the set of all strings of length n of a finite alphabet Σ . Let us consider u as a subsequence $s[i:j]$ of s with the start position i and the end position j , or $u = s[i]$ for short. The length $l(i)$ of u is given by $j - i + 1$. Then, by defining the u coordinate $\Phi_u(s)$ for each $u \in \Sigma^n$, we obtain a feature mapping Φ . Furthermore, the length of a substrings with some start position i can be weighted by a parameter λ , with $0 < \lambda \leq 1$, as follows:

$$\Phi_u(s) = \sum_{i:u=s[i]} \lambda^{l(i)}$$

Simply, a feature in \mathcal{F}_n is a measure of the number of occurrences of subsequences in a string s which are weighted according to their length. Weighting a subsequence by an exponentially decaying factor up to the full length in the text, allows to put a stronger emphasis on those occurrences that are close to contiguous [Lodhi et al.]. More clearly [Croce et al.]:

- longer subsequence receive lower weights
- gaps contribute to a weight since the exponent of λ is the number of characters *and* gaps between the first and last character
- characters like gaps *can* be omitted

The kernel function that calculates the inner product of two feature vectors of string s and t is then expressed as the sum over all common subsequences weighted according to their frequency of occurrence and lengths:

$$\begin{aligned} SSK_n(s, t) &= \sum_{u \in \Sigma^n} \Phi_u(s) \cdot \Phi_u(t) = \\ &= \sum_{u \in \Sigma^n} \sum_{i:u=s[i]} \lambda^{l(i)} \sum_{j:u=t[j]} \lambda^{l(j)} = \sum_{u \in \Sigma^n} \sum_{i:u=s[i]} \sum_{j:u=t[j]} \lambda^{l(i)+l(j)} \end{aligned}$$

These features have the complexity $O(|\Sigma|^n) = O(mnp)$ both in computational time and storage space where m and n are the lengths of the two strings while p is the length of the largest subsequence [Croce et al.]. Furthermore, in the works of [Lodhi et al.] analytical steps are described to obtain an efficient computation of the inner products via a dynamic programming technique.

4.4 Bag of Words Kernel / n-gram Kernel

The *bag-of-words (BoW) model*, also known as the vector space model (VSM) is a simplifying representation that is commonly applied in the area of NLP as well as in information retrieval (IR). Representing a document as a word vector is generally understood as a *bag-of-words*. Herein, each word that occurs in a document is given by its frequency in a document, while the ordering of words as well as characters for text structuring are ignored [Joachims 2000]. The representation of a bag-of-word vector is the mapping function Φ in feature space $\mathcal{F} = \mathbb{R}^N$:

$$\Phi : d \mapsto \Phi(d) = \begin{pmatrix} tf(t_1,d) \\ tf(t_2,d) \\ \dots \\ tf(t_n,d) \end{pmatrix} \in \mathbb{R}^N$$

$tf(t_i, d)$ is the *term frequency* of each term t_i , $i = \{1, \dots, N\}$ in a document d whereas 'term' and word is used synonymously. Given this representation, a document is mapped to an n -dimensional vector $\Phi(d) \in \mathcal{F} = \mathbb{R}^N$ that has the size of the dictionary with N being the number of terms occurring in the whole text corpus. When considering distinct words the size of the vocabulary is usually very large. Hence, each BoW vector is usually an extremely sparse histogram of that vocabulary.

Inspired by the *attribute-value representation* used in [Joachims 2000, Section 2.2.1] we can abstract from words and instead make use of other linguistic features like lemmas or part-of-speech-tags.

Table 4-1 shows exemplary sequences of part-of-speech tags processed to '*bag of PoS-tags*'. The dictionary consists of the terms t_1 =ADJD, t_2 =ART, t_3 =NE, t_4 =NN, and t_5 =VAFIN:

Document	Example	Sequence of PoS-tags / terms	$\Phi(d)$
d_1	Der Mann ist groß.	ART NN VAFIN ADJD	(1,1,0,1,1)
d_2	Der Mann ist Lehrer.	ART NN VAFIN NN	(0,1,0,1,1)
d_3	Der Mann ist ein Schrank.	ART NN VAFIN ART NN	(0,2,0,2,1)
d_4	Julia war ein Schwan.	NE VAFIN ART NN	(0,1,1,1,1)

Table 4-1: Representing the part-of-speech tags as bag of terms

For a given set of documents (sentences) a practicable representation is the *document-term matrix* D in which the column stores the frequencies of all terms that occur in the given documents while each row holds a bag of term vector according to each document.

The similarity computation between two documents s and t is achieved by calculating the inner product between the according 'bag of term' vectors. The BoW kernel is defined as follows [Sonnenburg et al.]:

$$BoW(s, t) = \langle \Phi(s), \Phi(t) \rangle = \sum_{i \in words} \Phi_i(s) \cdot \Phi_i(t)$$

In order to not depend on the length of the 'bag of term'-vectors the kernel is normalized:

$$\widehat{BoW}(s, t) = \left\langle \frac{\Phi(s)}{\|\Phi(s)\|}, \frac{\Phi(t)}{\|\Phi(t)\|} \right\rangle = \frac{k(s, t)}{\|\Phi(s)\| \cdot \|\Phi(t)\|}$$

Instead of bag of terms, we can use *n-grams* as an alternative characterization of a document. Here, sequences of n consecutive characters (n-grams) can be mapped into a feature space \mathcal{F} which is spanned by all possible strings of length n . The computation is the same as for the bag-of-word kernel. Since *n-grams* take any character into account, a single mismatching character leads to only n affected n-gram kernels, while the surrounding kernels remain intact [Sonnenburg et al.].

4.5 The Spectrum Kernel

The basic idea of the *spectrum kernel* (SpK) is to count the occurrences of a *k-spectrum* of contiguous subsequences in two given sequences s and t . A *k-spectrum* of a sequence is defined as the set of all k -length contiguous subsequences (also called '*k-mers*') that this sequence contains. For example the 3-spectrum of the sequence "gattaca" has the contiguous subsequences ["gat", "att", "tta", "tac", "aca"]. These counts are then mapped to a feature space \mathcal{F} which is spanned by $|\Sigma|^k$ many dimensions, with Σ being the alphabet of the text corpus. The mapping is done by indexing a *feature map* of all possible k -mers $u \in \Sigma^k$ and simply storing the number of occurrence $\#u(x)$ at the according index in this map. The spectrum kernel is then defined as:

$$SpK(s, t) = \Phi(s) \cdot \Phi(t) = \sum_{u \in \Sigma^k} \#u(s) \cdot \#u(t)$$

The spectrum kernel can efficiently be computed in $O(k \cdot (|s| + |t|))$ by using tries [Leslie et. al] and in $O(|s| + |t|)$ by using suffix trees [Vishwanathan & Smola] (Section 4.7).

4.6 The Tree Kernel

Given two trees (e.g. parse trees), the *tree kernel* (TK) captures common structural information by considering all tree fragments that occur in both trees [Collins et al.]. More precisely, the similarity between two trees is expressed by counting the subtrees they share. For trees X and Z , the *tree kernel* is defined as:

$$TK(X, Z) = \Phi(X) \cdot \Phi(Z) = \sum_{x \in X} \sum_{z \in Z} c(x, z)$$

The function $c(x, z)$ recursively counts the number of shared subtrees that are rooted in the nodes x and z :

$$c(x, z) = \lambda \prod_{i=1}^{|x|} (1 + c(x_i, z_i))$$

For the nodes x and z that are not derived from the same production, we define $c(x, z) = 0$. If x and z are leaf nodes of the same production, we obtain $c(x, z) = \lambda$ where λ is a trade-off parameter with $0 < \lambda \leq 1$ that balances the contribution of subtrees. For instance, choosing a small value for λ causes the contribution of lower nodes in large subtrees to decay. Figure 4-6 illustrates the shared subtrees of the trees X and Z :

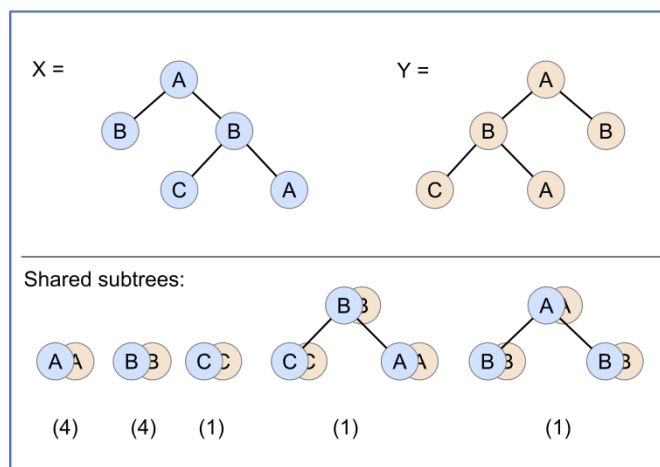


Figure 4-6: Shared subtrees in two parse trees; the numbers in brackets indicate the number of occurrences for each shared subtree pair

Due to the recursive definition of $c(x, z)$ and the identity of $\Phi(X) \cdot \Phi(Z) = \sum_{x \in X} \sum_{z \in Z} c(x, z)$, tree kernels are computed in worst case in $O(|X| \cdot |Z|)$ time. In the best case the computation time is close to linear in the number of nodes $(x, z) \in X \times Z$ [Collins et al.].

4.7 Fast Kernels for String and Tree Matching

This section presents the *linear time* algorithm of [Vishwanathan & Smola] to compute 'Fast Kernels for String and Tree Matching', or short 'fast string kernels' (FSK).

A few notations need to be introduced: Let Σ be a finite set of characters forming the alphabet. Any $x \in \Sigma^k$ with $k = 0, 1, \dots, n$ is called a string. Σ^* then represents the set of all non-empty strings defined over the alphabet Σ . Furthermore, $s, t, u, v, w, x, y, z \in \Sigma^*$ denote strings and $a, b, c \in \Sigma$ single characters.

Similarly to string kernels (Section 4.3), the FSK is expressed as the sum over all common subsequences $s \in \Sigma^*$ between two strings. The kernel function is furthermore extended by a weight parameter w_s which allows to weight arbitrary matching substrings s :

$$FSK(x, y) := \sum_{s \in x, s' \in y} w_s \delta_{s, s'} = \sum_{s \in \Sigma^*} w_s \cdot num_s(x) \cdot num_s(y) \quad (1)$$

Here, the function $num_s(t)$ denotes the number of occurrences of s in some string t . In order to use trees like constituency or dependency parse trees, a 'tree to string' conversion needs to be performed beforehand (Section 2.6.2).

When two strings x and y are compared, the algorithm makes use of two central data structures: The *suffix tree* (Section 4.7.1) which is created for a string x in *linear time*, and the *matching statistics* (Section 4.7.2) based on a string y with regard to x , built in *linear time* as well. Then, these structures are queried during an efficient kernel computation (Section 4.7.3).

Albeit arbitrary weights w_s can be associated to any matching substrings, the kernel computation still performs in linear time. The weights can be defined either *a priori*, for instance via a dictionary, or after seeing the data by employing one of the weight functions presented in Section 4.7.4.

Implementation details about the 'Fast String Kernel' as an operator for RapidMiner are presented in Section 4.8. It computes the kernel matrix for two example sets that can be forwarded to learners like the SVM. Due to the consumption of $O(n)$ space for each suffix tree, the memory quickly becomes a bottle neck for large sets. Therefore, different caching strategies are implemented that allow an efficient computation on limited memory.

Finally, a benchmark test of the operator was performed while differently sized example sets were used. The results are outsourced to Chapter 5, Section 5.4.

4.7.1 The suffix tree

The suffix tree is a *compact trie* that contains all the suffixes of a given text string [Knuth]. In the following, a brief summary is given about tries: A *standard trie* for a set of strings S is an ordered tree that has the following properties:

- Each node except for the root is labeled with a character.
- The children of each node are sorted in alphabetic order.
- Following a path from root to one of the leaves yields one of the strings of S .

Obtained from a standard trie, the nodes in a *compressed trie* have a degree of at least two. Furthermore, the chains of redundant nodes are compressed to single nodes. Figure 4-8 shows an exemplary compressed trie obtained from the standard trie in Figure 4-7 with the set of strings $S = \{\text{bear, bell, bid, bull, buy, sell, stock, stop}\}$:

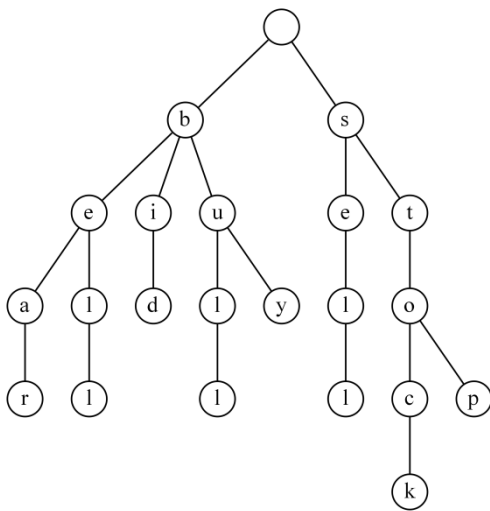


Figure 4-7: The standard trie of a set S of strings

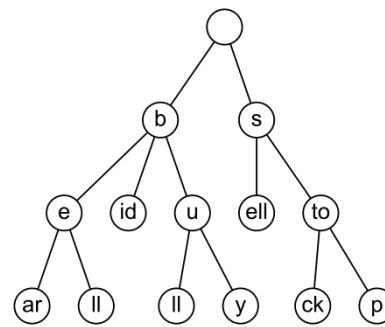


Figure 4-8: The compressed trie of a set S of strings

Now, a suffix tree is the *compact version of a compressed trie* which uses index ranges at the nodes. This concept is made more clearly with an example further below.

For the construction of a suffix tree a sentinel character $\$ \notin \Sigma$ is introduced which is lexicographically smaller than all the elements in Σ . Then, each input string x (pattern string) is enhanced by appending $\$$ at the end. The length of a string x is given by $|x|$. Again, the notation $x[i:j]$ describes a substring of string x with the start position i and end the position j (both inclusive), with $1 \leq i \leq j \leq |x|$. For some string $x = uvw$, u is known as a *prefix* of x while v is called the substring, and w the *suffix* of x .

Figure 4-9 illustrates the suffix-tree $S(x)$ of an enhanced exemplary string $x = ababc\$$. The compact representation of string x with index range $[0; |x| = 5]$ is given in Figure 4-10:

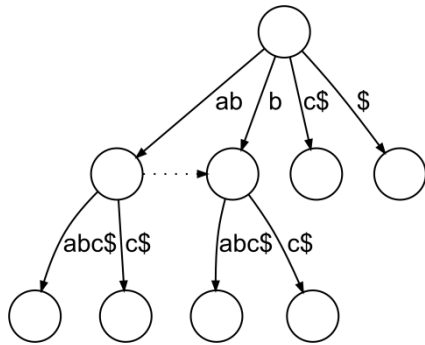


Figure 4-9: The suffix tree $S(x)$ for the string $x = "ababc\$"$

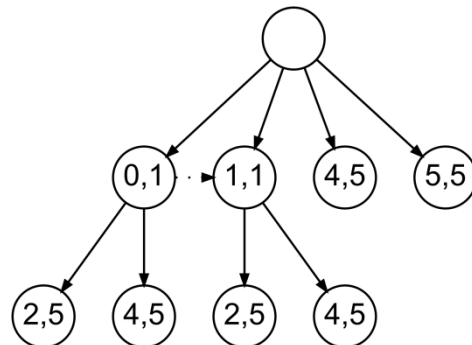


Figure 4-10: The compact representation of the suffix tree $S(x)$

For the kernel computation further below the following notations are required beforehand:

- \bar{w} denotes a path from root to a node while parsing the tree for a string w .
- $T_{\bar{w}}$ denotes the subtree rooted at a node \bar{w} .
- $lvs(\bar{w})$ yields the number of leaves of $T_{\bar{w}}$.
- $words(S(x))$ denotes the set of all non-empty prefixes w for some (possibly empty) string u such that $\bar{wu} \in nodes(S(x))$. Thus, $words(S(x))$ is the set of all possible substrings of $x\$$.
- For every $t \in words(S(x))$ we define $ceil(t)$ as the node \bar{w} such that $w = tu$ and u is the shortest (possibly empty) substring such that $\bar{w} \in nodes(S(x))$. Thus, when leading up the path to t then $ceil(t) = \bar{w}$ is the immediate next node.
- For every $t \in words(S(x))$ we define $floor(t)$ as the node \bar{w} such that $t = wu$ and u is the shortest *non-empty* substring such that $\bar{w} \in nodes(S(x))$. Thus, when leading up the path to t then $floor(t) = \bar{w}$ is the last encountered node.

For some internal node \bar{w} with $ceil(w) = \bar{w}$ the parent node is $floor(w)$. As shown in Figure 4-9, $floor("ba")$ is the second node with depth one reached by following the edge "b" from root. The only word $"ba" \in words(S("ba"))$ is found along the path $t = "b"u$, with $u = "a"$. $Ceil("ba")$ is the third node with depth two, following the edge "b" from root and then further the edge "abc\$".

Suffix trees can be constructed by employing the algorithms of [McCreight], [Ukkonen], and [Weiner] in linear time. For the implementation of the 'Fast String Kernel' operator, Ukkonen's online construction algorithm for suffix trees¹³ is adapted and further enhanced (Section 4.8, bullet 1).

¹³ For brevity, an introduction to this algorithm is omitted. [Gusfield] and [SO_Ukkonen] provide a comprehensible and extensive description of Ukkonen's linear time algorithm.

Figure 4-9 shows a *suffix link* from the internal node \overline{ab} to the node \overline{b} . Ukkonen's algorithm produces suffix links as an intermediate step in order to achieve a linear time construction of the tree. Furthermore, suffix links prove to be useful for an efficient string matching. Suppose that we found a substring aw of the string x by parsing the suffix tree $S(x)$. Trivially, w is also a substring of x . If in a later query we need the corresponding node to w , we find it in $O(1)$ time via suffix links instead of parsing the tree once again.

A suffix tree has an **important property**: If two substrings of x (e.g. $x = "ababc\$"$ with $s = "abab"$ and $t = "abc\$"$) have a common prefix (" ab ") they share the same path up to a common *ceil* node.

Generally, if we want to count the number of occurrences of a substring w in string x we first have to determine $ceil(w)$. Since all suffixes of w have to pass through $ceil(w)$, we could simply count the occurrences of the sentinel character '\$' which can only be found in the leaves. Instead, we access the number of leaves in each node \overline{w} in $O(1)$ time by precalculating $lvs(\overline{w})$ via a depth first search (DFS) and storing the value in each node \overline{w} .

Finally, in order to obtain $ceil(w)$ for a matching substring in $O(1)$ time, the *matching statistics* need to be prepared, as described next.

4.7.2 Matching statistics

The *matching statistics* are the central data structure in the computation of *fast string kernels*. They are calculated for a string y with respect to a pattern string x for which the suffix tree is created beforehand. In order to construct the matching statistics in *linear time* the algorithm of [Chang & Lawler] is integrated and adapted¹⁴ in the 'Fast String Kernel' operator for RapidMiner.

The idea is as follows: For each substring $y[i]$ defined by start positions $i := 1, \dots, |y|$ we determine the length v_i of the longest substring of x that matches a prefix of $y[i]$. Given these lengths, we can identify the *ceil* nodes $c_i = ceil(y[i: \hat{v}_i])$ and *floor* nodes $c_i' = floor(y[i: \hat{v}_i])$ that correspond to each match in the suffix tree $S(x)$.

In a nutshell, the matching statistics consist of the following vectors:

- $c \in nodes(S(x))^{|y|}$: Each i -th component c_i stores a *ceil node* $c_i \in nodes(S(x))$
- $c' \in nodes(S(x))^{|y|}$: Each i -th component c_i' store a *floor node* $c_i' \in nodes(S(x))$
- $v \in \mathbb{N}^{|y|}$: Each i -th component v_i denotes the length of the longest common substring (LCS) of x that matches a prefix of $y[i]$ with $c_i = ceil(y[i: \hat{v}_i])$ and $c_i' = floor(y[i: \hat{v}_i])$. Here, \hat{v}_i denotes the end location of a LCS, with $\hat{v}_i := i + v_i - 1$.

¹⁴ As explained in Section 4.8, bullet 1, the algorithm had to be enhanced in order to operate with 'items'.

Table 4-2 depicts the matching statistics of an exemplary string $y = "bcbab"$ with respect to the suffix tree $S(x = "ababc")$:

start pos. i	1	2	3	4	5
$y[i: \hat{v}_i]$	bc	c	bab	ab	b
v_i	2	1	3	2	1
c_i (ceil)	bc\$	c\$	babc\$	ab	b
c_i' (floor)	b	<i>root</i>	b	<i>root</i>	<i>root</i>

Table 4-2: Matching statistics of the string "bcbab" with respect to the suffix tree $S("ababc")$

For a start position i , $1 \leq i \leq |x|$, a substring that occurs in both x and y is a prefix of $y[i: \hat{v}_i]$. To see this, let us consider a substring $z = "ba"$ that occurs in both $x = "ababc"$ and $y = "bcbab"$. This implies that $z = y[i: j]$, here with $i = 3$ and $j = 4$. But the longest prefix of $y[i]$ that matches a substring of x is $y[i: \hat{v}_i]$, with $i = 3$ and $\hat{v}_3 = 3 + v_3 - 1 = 5$. Therefore, it can only be that $j \leq \hat{v}_i$ and the substring z is a prefix of $y[i: \hat{v}_i]$.

For brevity, the central idea of the algorithm of [Chang & Lawler] is outlined: The key observation is that the lengths v_i of matching substrings behave for consecutive starting positions i and $i + 1$ as follows:

$$v_i - 1 \leq v_{i+1}$$

This is true for all positions i , because if $y[i: \hat{v}_i]$ is a substring of x then $y[i + 1: \hat{v}_i]$ is trivially a substring of x , as well. Besides this, each matching substring in x must have $y[i + 1: \hat{v}_i]$ as a prefix.

The algorithm of [Chang & Lawler] uses this observation by walking down the suffix links in $S(x)$ in an intelligent manner to compute the matching statistics in $O(|y|)$ time:

Let us consider that for some matching substring $y[i: \hat{v}_i]$ a floor node c_i' is given, and that we further want to determine c_{i+1} , c'_{i+1} , and v_{i+1} . Therefore, we first find an intermediate node $p'_{i+1} = \text{floor}(y[i + 1: \hat{v}_i])$ by walking down the suffix link of c_i' and then walking along the edges that correspond to the remaining portion of $y[i + 1: \hat{v}_i]$ until we reach that node p'_{i+1} . Now, by simply following the edges that match $y[\hat{v}_i + 1: n]$ the next node that we encounter is the ceil node c_{i+1} . When we can no further parse elements in c_{i+1} , we obtain the length v_{i+1} . In order to determine v_1 , we can simply walk down the suffix tree $S(x)$ and find the longest prefix of y that matches a substring of x .

To summarize this section: Let x and y be two strings with the lengths $|x| = n$ and $|y| = m$, with $n \geq m$. By using the online construction algorithm of [Ukkonen] a pattern string x is read in

and the suffix tree $S(x)$ is constructed in $O(n)$ steps¹⁵. Then, the matching statistics are built for y with regard to x in $O(|y|)$ time, and we can read off the lengths \hat{v}_i of matching substrings for each start position i in constant time. Thus, it takes $O(m)$ steps, to determine if a substring $y[i:\hat{v}_i]$ of y is found in x or not. If so, each match contributes to the total sum in the similarity computation described in the next section.

4.7.3 Efficient Kernel computation

For the kernel computation a function $W(x, \cdot)$ is defined that assigns a weight to any matching substring $t = uv$ of x in constant time:

$$W(x, t) = \sum_{z \in \text{prefix}(v)} w_{uz} - w_u \quad (2)$$

Let us assume that we want to determine the sum of weights for the substring $t = uv = "ba"$ according to the suffix tree $S(x)$ shown in Figure 4-11. Recall that \bar{u} is the floor node which is the last encountered node on the path leading up to $t = uv$. Hence "b" resembles the prefix u . We now consider the weights w_{uz} for all substrings uz : By defining z as the set of all prefixes of v , we "catch" all the substrings that follow u on the path $t = uv$. In this trivial example $z \in \text{prefix}(v)$ is "a" of the ceil node \bar{v} labeled with "abc\$". In a large suffix tree,

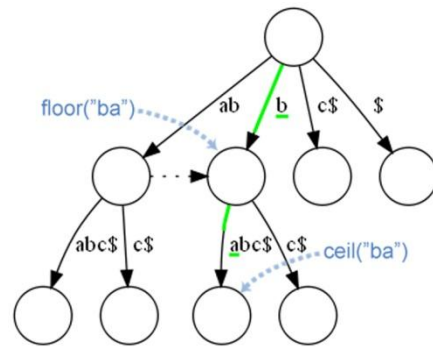


Figure 4-11: Weighting an exemplary matching substring $y = "ba"$ in the suffix tree $S(x)$ for the string $x = "ababc\$"$

however, this results in a set of suffixes that share the same substring u , also known as the *longest common prefix (LCP)*.

Briefly, $W(x, t)$ is the summed weight of all prefixes of substrings t in x reduced by the summed weight of all prefixes of their LCP. Thus, only weights are taken into account whose elements in the ceil node correspond to the remaining portion of a match.

Finally, the function to compute *fast string kernels* in $O(|x| + |y|)$ time is given as follows:

$$FSK(x, y) = \sum_{i=1}^{|y|} [val(c'_i) + lvs(c_i) \cdot W(x, y[i:\hat{v}_i])] \quad (3)$$

with $val(\bar{t}) = val(\text{parent}(\bar{t})) + lvs(\bar{t}) \cdot W(x, t) \quad (4)$

and $val(\text{root}) := 0 \quad (5)$

¹⁵ If the two strings have different length, then the suffix tree is created from the longer string.

Here, $val(\bar{t})$ computes the value of the parent of the ceil node \bar{t} plus the contributions due to all strings that end on the edge connecting the ceil node \bar{t} to its parent which is the floor node.

Claim: The kernel function (3) can be computed in *linear time*.

Proof: As mentioned above, the number of leaves in each node of $S(x)$ can be computed in $O(|x|)$ time via a DFS. Furthermore, the matching statistics algorithm constructs the vectors v , c and c' in $O(|y|)$ time. We assume a weight function $W(y, t)$ and utilize the recursive nature of $val(\bar{t})$ to precompute $W(y, w)$ for all $\bar{w} \in nodes(S(x))$ by a top down procedure in the suffix tree in $O(|x|)$ time. Now, in order to compute each term in (3) in constant time we can simply look up the precomputed $val(c'_i)$ and $lvs(c_i)$ and compute for each substring the weight $W(x, y[i: \hat{v}_i])$ in constant time. Due to the sum that iterates through $i = 1, \dots, |y|$, we have $|y|$ many terms that are calculated in $O(|y|)$. Therefore, the complexity of (3) is $O(|x| + |y|)$. ■

Claim: The kernel function (3) computes the string kernel (1).

Lemma 4-1:

The set of matching substrings of x and y is the set of all prefixes of $y[i: \hat{v}_i]$.

Proof: By applying the Lemma 4-1 to the basic kernel function (1), we can decompose the sum into a *sum over matches* between the pattern string x and each of the prefixes $y[i: \hat{v}_i]$. With this, we only need to show that each term in the sum of (3) corresponds to the contributions of all prefixes of $y[i: \hat{v}_i]$.

The following observation plays a key role in the computation: All substrings of x that share the same ceil node \bar{t} in the suffix tree $S(x)$ also have the same number of occurrences in x . This value is exactly reflected by $lvs(ceil(\bar{t}))$. Therefore, we can factor out $lvs(\bar{t})$ in (3) in order to properly scale up the contribution of each of the prefixes in $y[i: \hat{v}_i]$.

For instance, if we want to compute $val("bab")$ in the suffix tree $S(x = "ababc")$ we need to take the contributions due to "bab", "ba" and "b" into account. Trivially, "b" appears twice in the string x , namely as a prefix of "babc" and "bc", which in the suffix tree is equally reflected by $lvs(ceil("b")) = 2$. Hence, its contribution must be counted twice, while "ba" and "bab" occur only once in y and thus their contributions must be counted once.

For each $\bar{w} \in nodes(S(x))$ the recursive function $val(\bar{w})$ uses the above observation and calculates the contribution to the kernel due to \bar{w} and *all* its prefixes.

Let $t = uv$ be an arbitrary substring that has the floor node $\bar{u} = floor(t)$. In order to compute the contribution of t to the kernel, we have to consider:

- the contributions due to u and all its prefixes of u
- all strings of the form uz with $z \in \text{prefix}(v)$

As mentioned above, because each substring occurs exactly $\text{lvs}(\text{ceil}(\bar{t}))$ times in x , the kernel function $k(x, y)$ can use an efficient bracketing and a weight function $W(y, t)$. ■

Constructing the suffix tree $S(y)$ once and annotating each internal node with its *val* beforehand, reduces the time for further kernel computations to $O(|y|)$. This option is considered in the implementation of **caching mechanisms** which are presented in Section 4.8.

4.7.4 Weight functions

Various weight functions $W(x, y[i: \hat{v}_i])$ are suitable for the computation of *fast string kernels*:

1. **Length Dependent Weights:** We simply let the weights w_s depend on the length of $|s|$ by setting $w_s = w_{|s|}$. Further, we define $w_j := \sum_{i=1}^j w_i$. We precompute these weights beforehand up to w_j where $j \geq |x|$ for all x . Because the sums in $W(x, t)$ telescope, the weight function can be simplified to:

$$W(x, t) = \sum_{j=|\text{floor}(t)|}^{|t|} w_j - w_{|\text{floor}(t)|} = \sum_{j=1}^{|t|} w_j - \sum_{j=1}^{|\text{floor}(t)|} w_j = w_{|t|} - w_{|\text{floor}(t)|}$$

The values in the final step can simply be looked up in constant time from the precomputed matching statistics of the string x .

2. **Exponential decay:** In this weighting scheme exponential decay factors $w_i = \lambda^{-i}$ are used while $\tau := |t|$ and $\gamma := |\text{floor}(t)|$ denote the string boundaries:

$$W(x, t) = \frac{\lambda^{-\gamma} - \lambda^{-\tau}}{\lambda - 1} \quad \text{with} \quad \lambda \neq 1$$

The value of a decaying factor for some $0 < \lambda < 1$ becomes smaller the larger a string boundary is. On the opposite, for some given τ and γ , with $\gamma < \tau$, the resulting $W(x, t)$ puts a stronger emphasis on the matching substring the smaller λ is.

3. **Constant weight:** This approach simply measures the difference between the string boundaries τ and γ :

$$W(x, t) = \tau - \gamma$$

4. **Bag-of-characters:** When matching sequences of characters, we can simply set the weight $w_s = 0$ for all strings s with $|s| > 1$ and obtain a *bag-of-characters kernel* (Section 4.4):

$$W(x, t) = \begin{cases} 1, & \text{if } \gamma = 0 \text{ and } \tau = 1 \\ 0, & \text{otherwise} \end{cases}$$

5. **Bag-of-Words:** When matching single words (like part-of-speech tags), we may assign weights $w_s \neq 0$ for all strings s bounded by whitespace, and thus obtain the *bag-of-words kernel* (Section 4.4):

$$W(x, t) = \begin{cases} 1, & \text{if } \gamma = 0 \\ 0, & \text{otherwise} \end{cases}$$

6. **K-spectrum:** By setting specifically $w_s = 0$ for all strings s with $|s| \neq k$ for some sequence length k we obtain the *k-spectrum kernel* (Section 4.5).

7. **TF-IDF weights:** Another possibility is to choose *TF-IDF* weights that are achieved by first creating a list of all strings s including their frequencies of occurrence and then by rescaling the weights w_s accordingly. *TF-IDF*, short for *term frequency-inverse document frequency* (TF-IDF), is a measure to reflect how important a term is to a document d (sentence) in a collection D (text corpus). The simple form of the *term frequency* $TF(t, d) = |t \in d|$ counts the number of occurrence of a term t in a given document d . The *inverse document frequency* $IDF(t, D) = \log(N/|d \in D: t \in d|)$ with $TF(t, d) \neq 0$ measures how much information the term provides, that is if the term is common or rare across all N documents. Then, the *TF-IDF* then is defined as the multiplication:

$$TIDF(t, d, D) = TF(t, d) \cdot IDF(t, D)$$

4.8 The 'Fast String Kernel' operator for RapidMiner

The implementation of the 'Fast String Kernel' (FSK) as an operator in RapidMiner is the central building block in order to obtain a kernel method that is able to process text corpora in linear time. Details about this operator are documented in Appendix A.4. In the following list various important circumstances are outlined that have a direct impact on the practical implementation of the FSK operator:

1. **Atomic words vs. character sequences:** When analyzing sequences of tags (e.g. PoS-tags), no substring matching can be performed on these tags. Therefore, the implementation has to consider words as *atomic elements* that we denote by *items*. By implementing a generic approach the FSK operator either accepts sequences of characters (strings) or sequences of items. In the latter case the labels of the edges in a suffix tree carry items instead of sequences of characters. Further, all implemented methods had to be extended in order to deal with items, as well. An important preparation step again is to

enhance each sequence of items with the sentinel character \$, for instance $s = "NE, VVFIN, ADV, NE, \$"$. Finding matches between two sequences of items works in the same way as with matching strings. Lastly, the FSK operator accepts sequences of items as strings (by internally detecting the commas and appending each items to a list).

2. **Normalising the kernel:** In order to measure the similarities of two strings x and y on a common scale, the operator always computes normalized kernels (Section 4.1).
3. **The upper triangular matrix:** When comparing only the strings / itemsets x_i and x_j with $1 \leq i, j \leq n$ within an example set E , the operator computes the upper gram matrix due to $FSK(x_i, x_j) = FSK(x_j, x_i)$. Since the normalized similarity value of a string x_i compared with itself is 1, the diagonal entries are directly set to $FSK(x_i, x_i) = 1$.
4. **Weight functions:** The FSK operator provides weighting of strings and items according to the functions 1 - 5 given in Section 4.7.4.
5. **Caching strategies:** Let us consider two different example sets E_1 with the strings x_i ($1 \leq i \leq n$) and E_2 with the strings y_j ($1 \leq j \leq m$). Since a suffix tree consumes $O(n)$ memory space that is linear to the length of the input string, it is desirable to avoid rebuilding any of the suffix trees when comparing E_1 with E_2 . Because all kernels are normalized, the suffix trees have to be constructed for all input strings. Therefore, the minimum amount of suffix trees is $\theta(n + m)$. While iterating through the kernel matrix K in order to compute $\theta(n \cdot m)$ similarities, different caching strategies are implemented to keep or discard once constructed suffix trees:

- **No caching:** While iterating through both $i = 1, \dots, n$ and $j = 1, \dots, m$ no suffix tree is kept in memory. Hence, the number of tree constructions is $\theta(n \times m)$.
- **Caching suffix trees from E_2 :** The set of n suffix trees is constructed and kept in memory. While iterating through the rows $j = 1, \dots, m$ each j -th suffix tree is constructed on demand. Thus, only $\theta(n + m)$ trees are required in total.

FSK	$S(x_1)$	$S(x_2)$...	$S(x_n)$
$S(y_1)$	$FSK(y_1, x_1)$	$FSK(y_1, x_2)$...	$FSK(y_1, x_n)$
$S(y_2)$				
...				
$S(y_m)$				

Table 4-3: Caching strategy where n suffix trees are kept in memory while iterating through the rows $j = 1, \dots, m$

- **Windowing:** In this caching strategy, a rectangular window is defined with width k and height l on the interval ranges $1 \leq i + k \leq n$ and $1 \leq j + l \leq m$. In this window a set of $k + l$ suffix trees is constructed and the computation is performed line-by-line. Afterwards the window is shifted to the right, while the suffix trees $S(y_i), \dots, S(y_l)$ are kept in memory. If the window exceeds the outmost right column it is shifted back to the first column and down by l many rows. No trees are kept in memory, then.

Iterating through the rows requires $O\left(l \left\lceil \frac{m}{l} \right\rceil\right)$ trees and $O\left(\left\lceil \frac{m}{l} \right\rceil \cdot \left[\left\lceil \frac{n}{k} \right\rceil \cdot \lceil k \rceil\right]\right)$ when shifting the window along the columns. Therefore, the total sum of tree constructions amounts to $O\left(\lceil m \rceil + \left\lceil \frac{m}{l} \right\rceil \cdot \lceil n \rceil\right)$.

FSK	...	$S(x_i)$...	$S(x_{i+k})$...
...					
$S(y_j)$		$FSK(y_j, x_i)$...	$FSK(y_j, x_{i+k})$	
...		
$S(y_{j+l})$		$FSK(y_{j+l}, x_i)$...	$FSK(y_{j+l}, x_{i+k})$	

Table 4-4: Caching strategy with a window of size $k \cdot l$

Chapter 5

Experiments

This chapter presents three experiments in which the implemented operators annotate documents (Appendix A.1) and extract linguistic features (Appendix A.2). Where applicable the visualization operator (Appendix A.3) demonstrates a comfortable way to label each sentence. After performing a learning on linguistic features, the effectiveness of the implemented weight functions could be tested. More precisely, the classification rates for unseen examples were measured after a binary classification model was trained by the SVM (Section 4.2).

5.1 Experiment I: "Tranches"

5.1.1 Acquiring labeled data

In the first experiment a set of sentences from various German newspapers was considered where each sentence had been manually obtained from [DWDS]. Each sentence was given a topic word whereas the word was also contained in the corresponding sentence.

Next, two *tranches* were taken from this set: The *first tranche*, to be used for training, consisted of 6650 sentences with the initial letters of topics ranging from Q to T. The *second tranche* contained 6697 sentences with topics ranging from T to Z and was used for the testing phase.

The idea for a classification task was to assign a positive label to a sentences where the topic word was appropriately describing the sentence. In the case that the topic word was different than the actual topic of the sentence, a negative label was given. Two exemplary sentences are shown in Table 5-1 regarding the topic 'quality criteria':

Topic:	Example describing the topic?	Label:	Source:	Date:
Qualitätskriterien	Es sei an der Zeit, Qualitätskriterien für die Pflege festzulegen.	+1	Frankfurter Allgemeine Zeitung	27.09.1995
Qualitätskriterien	Wie ehrlich ist es, die Privattheater nach strengeren Qualitätskriterien zu bewerten?	-1	Die Welt	02.03.2001

Table 5-1: Two examples that differently fit to a given topic.

The first example which translates to 'It is about time to determine the rules quality criteria for (health) care' is labeled with '+1' as it fits the topic well. The second sentence which translates to 'How honest is it to evaluate private theaters according to more stringent quality criteria?' is labeled with a '-1' since the topic is about private theaters.

A comfortable way to label the sentences from both tranches is to use the 'Visualize and Label Parse Trees' operator (Appendix A.3). However, this operator expects features like tokens and parse trees as a minimum. Therefore, the 'WebLicht Feature Annotator' (Appendix A.1) was

first applied on each sentence. In the settings of this operator the language parameter was set to 'de' and the following tool chain was chosen (see Appendix A.1.2 for each employed WebLicht service):

Converter#1 → Tokenizer#3 → PoS-Tagger#2 → Dependency-Parser#1

The operator annotated the sentences with tokens, lemmas (annotated by PoS-Tagger#2), PoS-tags and dependency parse trees. Afterwards, the 'WebLicht TCF to ExampleSet' operator extracted these features to strings. Parse trees were converted to 'tree strings' and 'tree string (tokens)' (Section 2.6.2). Table 5-2 lists these features according to the first example sentence:

Feature type:	Extracted string:
original sentence	Es sei an der Zeit, Qualitätskriterien für die Pflege festzulegen.
tokens	Es,sei,an,der,Zeit,,,Qualitätskriterien,für,die,Pflege,festzulegen,.,
lemmas	es,sein,an,der,Zeit--,Qualitätskriterium,für,der,Pflege,festlegen,--
PoS-tags	PPER,VAFIN,APPR,ART,NN,\$,,NN,APPR,ART,NN,VVIZU,\$.
tree string	[Root[VAFIN[PPER[VVIZU[NN[APPR[NN[ART]]],\$.],APPR[NN[ART,\$,]]]]]
tree string (PoS-tags)	, VAFIN, PPER, VVIZU, NN, APPR, NN, ART, \$., APPR, NN, ART, \$,
tree string (tokens)	, sei, Es, festzulegen, Qualitätskriterien, für, Pflege, die, ., an, Zeit, der, ,
tree string (lemmas)	, sein, es, festlegen, Qualitätskriterium, für, Pflege, der, --, an, Zeit, der, --

Table 5-2: Extracted features of an exemplary sentence.

The 'Visualize and Label Parse Trees' operator could then present the dependency parse trees with the option to label the according sentence, like in Figure 5-1. Note that the checkbox "Label" indicates a positive label "+1" when checked, and respectively, "-1" when left unchecked.

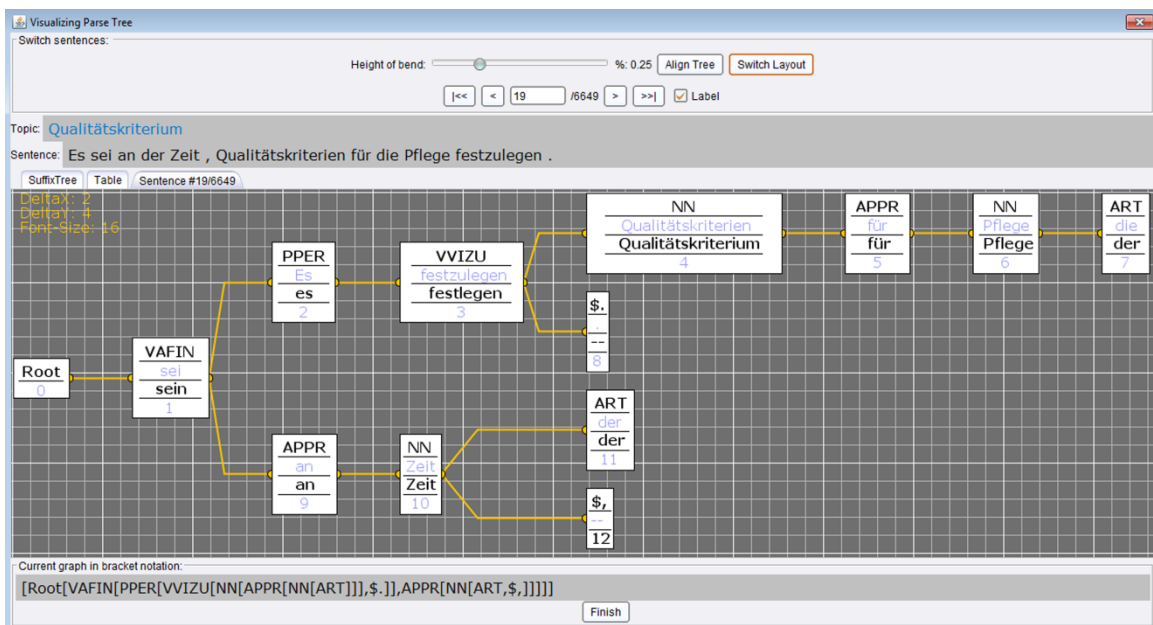


Figure 5-1: Labeling sentences with the 'Visualize and Label Parse Trees' operator.

To use space optimally, the parse tree is shown in a horizontal layout.

For the sake of completeness, the parse tree for the second exemplary sentence above is shown in Figure 5-2:

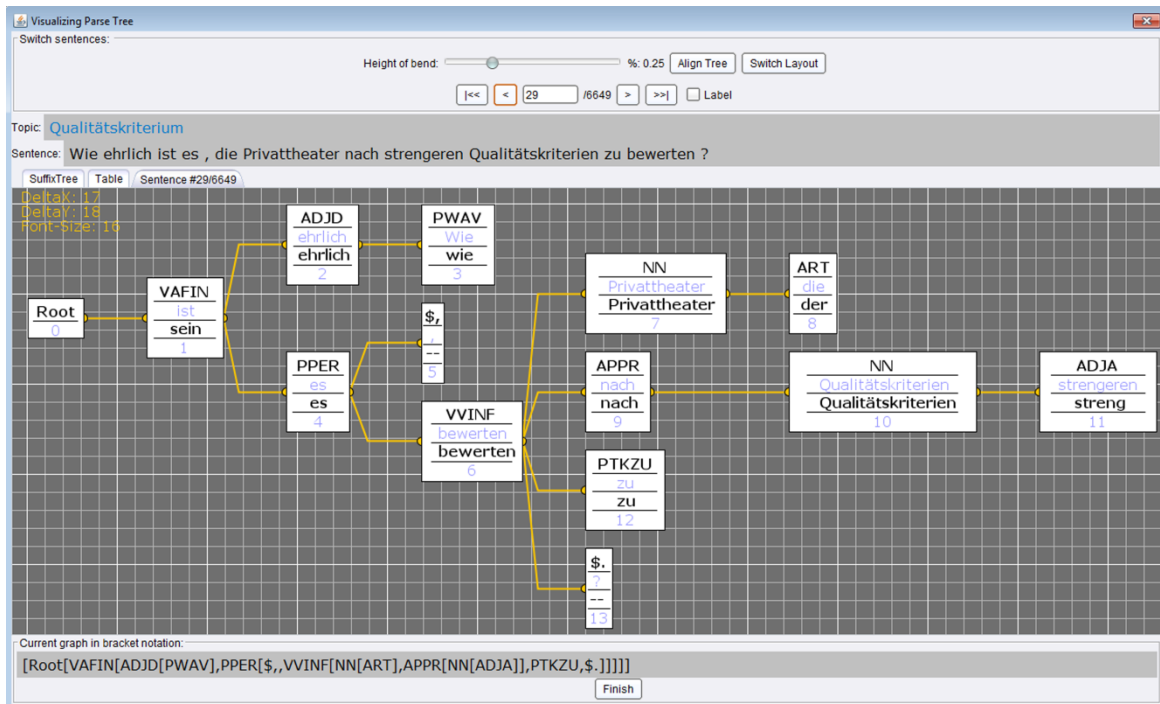


Figure 5-2: Assigning a negative label by leaving the label control field unchecked

The files to this experiment containing the tranches for training and testing, the RapidMiner processes for annotating the original sentences, and the output files with the preprocessed sentences are placed in the folder 'Experiments\Tranche' on the CD that is enclosed with this work:

Input file:	RapidMiner process file:	Output file:
Tranche_TestSet.csv	Annotate&Extract_TrancheTest_Write2CSV.rmp	TranchePreppedTestSet.csv
Tranche_TrainingSet.csv	Annotate&Extract_TrancheTraining_Write2CSV.rmp	TranchePreppedTrainingSet.csv

Table 5-3: List of files used or created during Experiment I

The annotation process took around seven hours for each set containing roughly 6.700 sentences. Then, the files with the prepared examples were examined and those examples that provided no information were removed. For instance, this was the case when the set of tokens only consisted of a single token like a punctuation character.

In cases where the WebLicht services could not properly detect the sentence boundaries the original sentences were manually fixed by extending abbreviations or removing chapter numbers like "X.". Afterwards, the RapidMiner process for feature annotation and extraction was executed for that particular sentence again.

5.1.2 Training phase

To produce a training set, 3000 examples were randomly taken from the first tranche, that is from 'TranchePreppedTrainingSet.csv'. More precisely, via *stratified sampling*¹⁶ a randomized subset was obtained from the list of examples.

Since we want to study the influence of linguistic features on the prediction ability, that is, to determine if a sentence is or is not a "good" example for the corresponding topic, one particular feature type was considered in each training run. Figure 5-3 shows the list of different training data used for Experiment I:

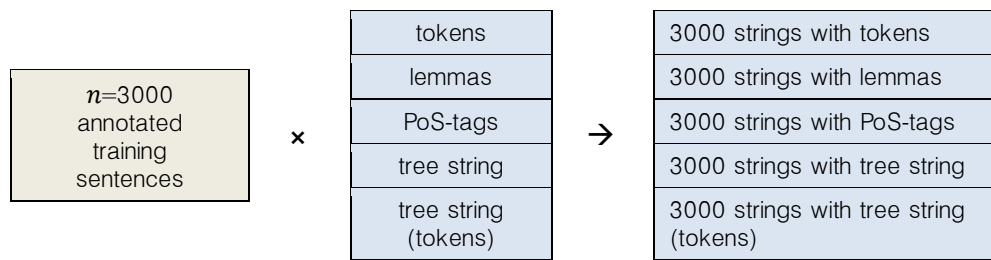


Figure 5-3: Training sets in Experiment I with each set considering one particular feature type

Then, the 'Fast String Kernel' (FSK) operator calculated a kernel matrix of the n examples for each feature type. In order to learn models, the matrix was forwarded to the SVM learner which is available as the 'LibSVM' operator in RapidMiner¹⁷.

It is important to note that in both the learning and testing runs the FSK operator treated PoS-tags and 'tree strings' as *items* (described in Section 4.8). For the other features the FSK operator performed a substring matching on the full strings.

During the training of the SVM learner a 'Parameter Optimization' operator was used to find the optimum value for the penalty term C that leads to the largest possible margin between two classes regarding the training set. In order to find the optimum in each training run, the optimization process performed 40 classification runs on the training set with an interval of $C = [0.01; 10]$. After the highest accuracy and the optimal value for C was determined, the interval range was fine tuned and the optimization process was started once again.

Additionally, in each optimization step, a (*leave-k-out*) *cross-validation* technique was applied to avoid overfitting with regard to a specific subset of the data. Therefore, the 'X-validation' operator performed a partitioning of the training data in $k = 10$ randomized subsets of equal size. In each validation step a single subset was retained as testing data while $k - 1$ remaining subsets were used for the training of the SVM learner. The cross-validation process was then

¹⁶ In stratified sampling a random subset is chosen in such a way that the class distribution in the subset is the same as in the whole example set.

¹⁷ In the parameter settings of the SVM we chose 'precomputed' as the kernel type and set the cache size to a sufficient memory limit of 1024MB.

repeated k times while each of the k subsets was used once as testing data. At the end of each cross-validation the performance results were averaged to provide an estimation of how accurately each learned model may perform on the testing data.

Furthermore, each feature type was combined with a particular weight function (Section 4.7.4) in order to observe how a weight function or feature type influences the resulting prediction accuracies. Table 5-4 lists the optimized C and λ values with the corresponding accuracies obtained from all training runs:

Optimized C in Exp. I:	Constant weight	Length Dependent	Exponential Decay	B-o-C	B-o-W
Tokens	55.77% \pm 2.39%, C=2.3275	56.50% \pm 1.99%, C=5.045	57.57% \pm 2.26%, C=1.778, $\lambda=1,0000000000000002$	54.27% \pm 0.39%, C=4.06	57.53% \pm 1.28%, C=6.8505
Lemmas	55.90% \pm 1.32%, C=8.312	56.43% \pm 0.65%, C=1.09	57.40% \pm 2.59%, C=2.3275, $\lambda=1,0000000000000002$	54.17% \pm 0.87%, C=6.835	56.73% \pm 1.71%, C=8.312
PoS-Tags	56.57% \pm 1.41%, C=3.956	56.20% \pm 1.27%, C=100.0	56.57% \pm 1.41%, C=3.956, $\lambda=1,0000000000000002$	54.63% \pm 0.82%, C=9.851	57.70% \pm 1.29%, C= 5.05
Tree string	57.33% \pm 2.13%, C=8.312	56.67% \pm 2.23%, C=2.3275	57.33% \pm 2.13%, C=8.312, $\lambda=1,0000000000000002$	54.43% \pm 0.50%, C=1.195	57.30% \pm 1.77%, C=3.956
Tree string (tokens)	56.73% \pm 2.26%, C=0.01	57.37% \pm 1.72%, C=1.2385	56.73% \pm 2.26%, C=0.001, $\lambda=1,0000000000000002$	54.47% \pm 1.01%, C=2.893	57.53% \pm 1.01%, C=0.689

Table 5-4: The optimized C values and with the according accuracies (and standard deviations) obtained from the training runs in Experiment I.

In general, the training data appears not to be easily separable due to the overall low accuracies measured in all combinations (lowest: 54.17%, highest: 57.70%). Using the 'bag-of-words' (B-o-W) function in combination with PoS-tags is particular interesting as the matching series of PoS-tags between two examples appear to be useful enough to lead to the highest accuracy.

However, the differences between the accuracies (with standard deviations up to 2.59%) are still too little to make a reliable statement when comparing each feature type with a weight function. The only exception hereto is the 'bag-of-character' (B-o-C) function that achieves the lowest accuracies for all feature types.

The file to the RapidMiner process for training on various features can be found in the folder 'Experiments\Tranche' on the CD:

RapidMiner process file:
Experiments\Tranche\FSK_Tranche_X-Validation_Of_TrainingSet.rmp

Table 5-5: RapidMiner file used for the training phase in Experiment I

5.1.3 Testing phase

For the testing phase all 6697 sentences of the second tranche ('TranchePreppedTestSet.csv') have been used. Furthermore, in each run the following steps were performed:

In the first step, the string kernels from the training data were computed and the resulting kernel matrix was then forwarded to the SVM learner in which the optimized parameter C was applied. This re-established the optimized models that were determined in the training phases before. When using the 'exponential decay' function, the optimized λ were applied in the FSK operator.

In the second step, a second FSK operator was employed to compute the kernels of training examples compared with unseen examples. The kernel matrix was then forwarded to the 'Apply Model' operator which made use of the optimized SVM models in order to perform the testing on unseen data.

Figure 5-4 depicts the RapidMiner process in which the 3000 training examples are obtained in the upper subprocess and the 6697 test examples in the lower subprocess:

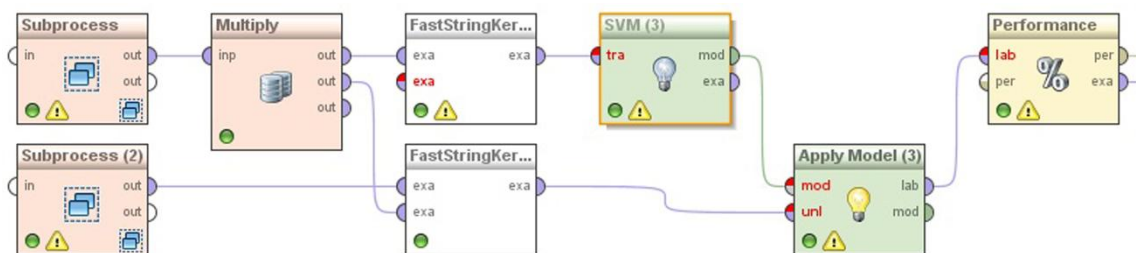


Figure 5-4: The RapidMiner process in which the optimized SVM models are applied to unseen test examples.

The file containing the RapidMiner process for performing the classification tests can be found in 'Experiments\Tranche' on the CD:

RapidMiner process file:

Experiments\Tranche FSK_Tranche_PREDICT_LABELs_Testset.rmp

Table 5-6: RapidMiner file used for the testing phase in Experiment I

Table 5-7 shows the classification results with all the combinations of feature type and weight function:

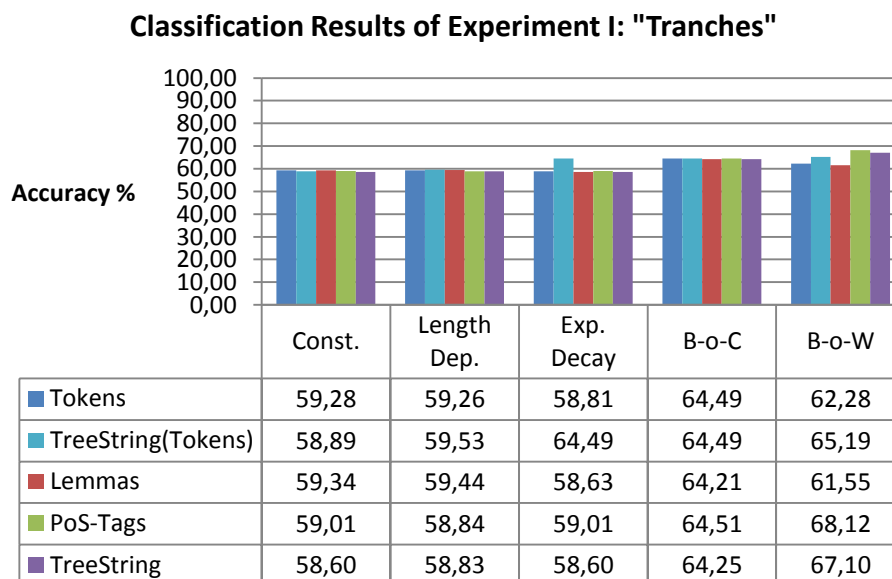


Table 5-7: Classification results (%) of Experiment I "Tranches" obtained from different combinations of linguistic features and weight functions

The classification on unseen examples achieves accuracies with the lowest value at 58.6% and a peak value of 68.12%. The 'bag-of-characters' (B-o-C) and 'bag-of-words' (B-o-W) functions achieve in most combinations far better accuracies than their competitors. This is surprising for the B-o-C function as the accuracies achieved in the training phase did not stand out from the other values. Recall that the B-o-C function considers only the first character in a matching substring and in case of PoS-tags and 'tree strings' the first item is considered.

The best result is achieved by combining the B-o-W function with PoS-tags where a prediction probability of up to 0.68 was obtained, and with 'tree strings' a prediction probability of 0.67. With the assumption in the training phase above, these pleasing results here lead us to the conclusion that unseen sentences (that are "good" examples for the corresponding topic word) can be predicted best by considering the PoS tags and the underlying grammar of the sentences rather than by considering matching tokens.

Interestingly, the combination of the 'exponential decay' function with the 'tree string (tokens)' feature achieved an accuracy of (64.49%) which is 5.48% better than the second best accuracy reached with PoS-Tags (59.01%). This slightly outstanding classification result might be due to the fact that the 'tree string (tokens)' feature combines the structural information of a dependency parse tree with the tokens in each sentence.

5.2 Experiment II: "Literature types"

5.2.1 Acquiring texts from different periods

In the second experiment two data sets with sentences containing the German word "Leiter" (English: ladder, leader or conductor) were analyzed. The first set consisted of 3188 sentences collected from *general literature* (containing the subcategories: functional writing, science and belletristic) of the 19th century and earlier, up to the 16th century. The second corpus consisted of 3097 sentences from *contemporary literature* (mainly taken from newspaper texts) starting from around the second half of the 20th century up to the present. Table 5-8 lists two exemplary sentences that represent both types of literature:

Example sentence:	Literature type:	Label:
Bey diesem Gesicht aber bleibet es nicht sondern daß jm das liebste ist so ist Gott selber verhanden vnd stehet oben auff der Leiter vnd thut dem Jacob ein tröstliche Predigt.	general literature	-1
Der Leiter des Komitees lebt mit seiner Familie in einem Stadtteil, der vom Regime kontrolliert wird.	contemporary literature	+1

Table 5-8: Two examples of general and contemporary literature

Similarly to Experiment I, the sentences from both data sets were annotated with features in roughly seven hours and afterwards written to output files. Also here, the list of annotated sentences was purged from irrelevant examples and, in cases of erroneously detected sentence boundaries, the sentences were manually fixed.

The original data sets, the RapidMiner processes for annotating the original sentences, and the output files can be found in the folder 'Experiments\Leiter' on the enclosed CD:

Input file:	RapidMiner process file:	Output file:
LeiterAllg.csv	Annotate&Extract_LeiterAllg_Write2CSV.rmp	preppedLeiterAllg.csv
LeiterZeit.csv	Annotate&Extract_LeiterZeit_Write2CSV.rmp	preppedLeiterZeit.csv

Table 5-9: List of files used or created during Experiment II

5.2.2 Training phase

For the classification task a binary label was assigned to each sentence in order to indicate the type of literature. During testing, the SVM should predict the label of a previously unseen sentence regarding to which type of literature it belongs.

In order to produce a training set, the first 1500 examples of a distinct feature type were taken from each data set, that is, 1500 examples from general literature plus 1500 examples from contemporary literature.

The remaining sentences were then used for testing. Figure 5-5 shows the different training data obtained for Experiment II:

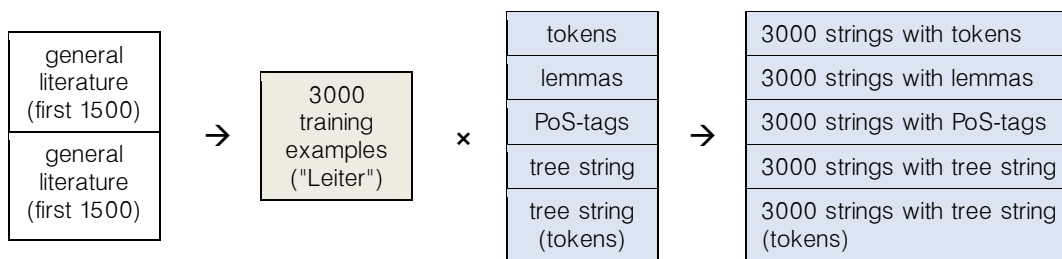


Figure 5-5: Training sets for Experiment II

In the same way as for Experiment I, the computed kernel matrix was forwarded to the SVM. Again, the SVM was wrapped by an optimization process in order to determine optimal values for the parameters C and λ , and a cross validation was performed in each optimization step.

The file to the RapidMiner process for training on various features can be found in the folder 'Experiments\Tranche' on the CD:

RapidMiner process file:

Experiments\Tranche\FSK_Leiter_X-Validation_Of_TrainingSet.rmp

Table 5-10: RapidMiner file used for the training phase in Experiment II

Again, by combining each feature type with a particular weight function (Section 4.7.4) the following training accuracies were obtained, as shown in Table 5-4:

Optimized C in Exp. II:	Constant weight	Length Dependent	Exponential Decay	B-o-C	B-o-W
Tokens	87.00% \pm 1.80%, C=11.0	82.75% \pm 2.19%, C=8.48	87.05% \pm 1.72%, C=3.7, $\lambda=1,0000000000000002$	72.00% \pm 2.66%, C=0.0632	82.85% \pm 1.55%, C=19.204
Lemmas	87.10% \pm 1.55%, C=4.75	82.05% \pm 1.81%, C=6.4036	87.10% \pm 1.46%, C=3.565, $\lambda=1,0000000000000002$	74.45% \pm 2.69%, C=0.1099	83.40% \pm 1.81%, C=7.5136
PoS-tags	81.30% \pm 2.38%, C=1.8355	80.50% \pm 1.63%, C=2.008	81.90% \pm 2.51%, C=4.515, $\lambda=1,0000000000000002$	60.50% \pm 2.82%, C=1.75825	78.95% \pm 3.11%, C= 15.75
TreeString	80.93% \pm 1.99%, C=3.0705	79.45% \pm 3.39%, C=6.8675	80.90% \pm 1.98%, C=3.07, $\lambda=1,0000000000000002$	56.80% \pm 2.54%, C=7.123	78.13% \pm 2.58%, C=9.575
TreeString (Tokens)	87.93% \pm 1.59%, C=2.3275	84.50% \pm 1.45%, C= 2.3275	87.90% \pm 1.71%, C=7.2775, $\lambda=1,0005$	52.77% \pm 3.62%, C= 9.2575	83.27% \pm 1.43%, C=9.7525

Table 5-11: The optimized C values with the according accuracies (and standard deviations) obtained from the training runs in Experiment II.

Based on the high trend of all accuracies we may assume that the training data is generally very well separable. Certainly, this can be traced back to the fact that the written language used in *general literature* from the 19th century and earlier (up to the 16th century) differs greatly from the word usage in contemporary texts of modern times (20th century up to the present).

By comparing the weight functions we observe that the 'exponential decay' function achieves the highest accuracies. Similar high values are also achieved by the 'constant weight' function. Both the 'length dependent' and 'bag-of-words' (B-o-W) functions achieved roughly similar accuracies. The 'bag-of-characters' (B-o-C) function obtained a prediction performance that is far below average. Regarding the B-o-C function this appears comprehensible as each matching substring only consists of the first character, or they consist of the first item in case of PoS-tags and tags used in dependency parse trees.

When comparing the different linguistic features, we notice that the highest accuracies are achieved by using tokens, lemmas, and 'tree strings (tokens)' ¹⁸ (again, with exception to the B-o-C function). Particular pleasing is that using the 'tree strings (tokens)' resulted to the highest possible accuracies. We can further observe that using PoS-tags and the 'tree strings' lead to the lowest accuracies - independent of the chosen weight function.

This leads to the assumption that the best accuracies for predicting the memberships of unseen sentences to one of the two literature types can be expected by considering tokens, lemmas and 'tree strings (tokens)'.

¹⁸ 'Tree string (tokens)' are tree strings in which the tags in the leaves of the original parse tree have been replaced by the tokens of the corresponding sentence.

5.2.3 Testing phase

In the testing phase the remaining n-1500 examples of a distinct feature type were taken from each data set, that is, the last 1688 sentences from the first set (general literature) plus the last 1597 sentences from the second set (contemporary literature).

Again, in each classification run each feature type was combined with each of the weight functions. The results are presented in Table 5-12:

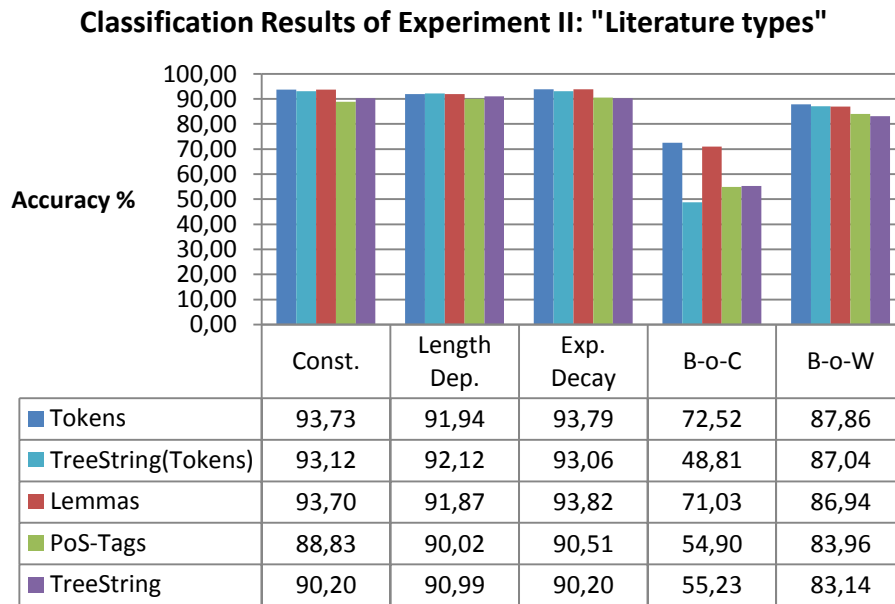


Table 5-12: Classification results (%) of Experiment II "Literature types" obtained from different combinations of linguistic features and weight functions

The classification of unseen examples performs very well with the exception of the B-o-C function. As expected in the training phase, the highest accuracies are achieved when the 'Fast String Kernel' operator makes use of tokens, 'tree strings (tokens)', and lemmas. Here, the 'constant weights' function and the 'exponential decay' function compete on accuracies between 93.06% and 93.82%.

Using PoS-tags or 'tree strings' (consisting of grammar tags) leads to classification rates that are generally worse than the winners, and except for B-o-C all functions achieve a prediction performance that is up to 4,9% worse than considering tokens, lemmas and tree string (tokens).

This further supports the assumption that both data sets containing contemporary and general literature can be separated very well due to the word usage in these literature types.

5.3 Experiment III: "Bild vs. Spiegel"

5.3.1 Acquiring sentences from online articles

The idea for a classification task in the third experiment was to distinguish arbitrary sentences taken from articles of the German newspaper "Bild" from sentences obtained from articles of the German magazine "Spiegel" [Bild, Spiegel].

More precisely, the sentences of 30 online articles from "Bild.de", and the sentences of 33 online articles from "Spiegel.de" were manually collected and stored in two separate data sets. The articles were from March 2015, and only the text bodies were considered while headlines and subtitles were omitted since both publishers use catching phrases that may have similar grammatical structure.

A negative binary label was then assigned to all the sentences originating from articles of "Bild.de" and a positive label to the sentences taken from "Spiegel.de" articles. During testing, the SVM should predict the label of a previously unseen sentence and therefore decide from which source the sentence originally came.

By using the 'WebLicht Feature Annotator' (Appendix A.1), the collected articles were annotated with features and afterwards written to output files. Since the 'WebLicht TCF to ExampleSet' operator is outputting single annotated sentence, all the annotated articles were extracted to sets of sentences. In the end, for the first data set 784 sentences could be obtained from "Bild.de" articles, and for the second set 932 sentences could be extracted from "Spiegel.de" articles. Table 5-13 lists an exemplary sentence from each source:

Example sentence:	Source:	Date:	Label:
Ausgerechnet am Tag, an dem die Welt in Sotschi den olympischen Geist beschwor, ordnete Putin die geheime Krim-Operation an.	Bild.de	09.03.2015	-1
Ein mögliches Übereinkommen würde Teil internationalen Rechtes und könnte von einem neuen US-Präsidenten nicht so einfach wieder aufgehoben werden.	Spiegel.de	12.03.2015	+1

Table 5-13: Two exemplary sentences taken from online articles from Bild.de and Spiegel.de

The original data sets, the RapidMiner processes for annotating the original sentences, and the output files can be found in the folder 'Experiments\Bild_vs_Spiegel' on the enclosed CD:

Input file:	RapidMiner process file:	Output file:
Bild_article.csv	Annotate&Extract_Bild-Articles_Write2CSV.rmp	Bild_article_prepped.csv
Spiegel_article.csv	Annotate&Extract_Spiegel-Articles_Write2CSV.rmp	Spiegel_article_prepped.csv

Table 5-14: List of files used or created during Experiment III

5.3.2 Training phase & Testing phase

In order to produce a training set, the first 350 examples of a distinct feature type were taken from each preprocessed data set, that is, 350 examples from "Bild.de" plus 350 examples from "Spiegel.de". The remaining sentences were then used for testing. Figure 5-5 shows the list of different training data obtained for Experiment II:

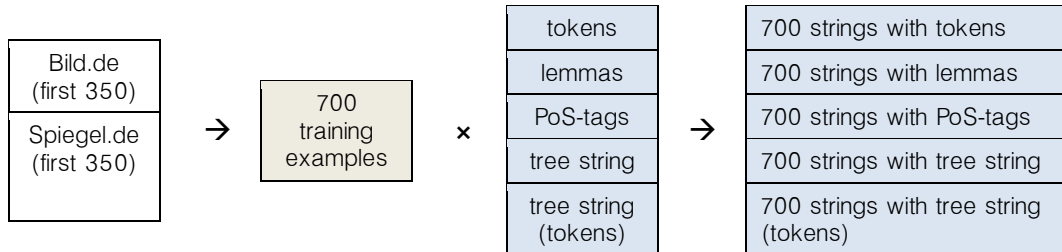


Figure 5-6: Training sets for Experiment III

In the same way as in the previous experiments, the computed kernel matrix was forwarded to the SVM. The SVM was wrapped by an optimization process as well, in order to determine optimal parameters C and λ , and a cross validation was performed in each optimization step.

The file to the RapidMiner process for training on various features can be found in the folder 'Experiments\Tranche' on the CD:

RapidMiner process file:

Experiments\Bild_vs_Spiegel\FSK_Bild_vs_Spiegel_X-Validation_Of_TrainingSet.rmp

Table 5-15: RapidMiner file used for the training phase in Experiment III

In each training run one feature type was combined with a particular weight function (Section 4.7.4). Table 5-4 presents the training accuracies:

Optimized C in Exp. III:	Constant weight	Length Dependent	Exponential Decay	B-o-C	B-o-W
Tokens	78.00% \pm 5.39%, C=3.751	73.86% \pm 5.07%, C=3.751	78.00% \pm 5.39%, C=3.751, $\lambda=1.0000000000000002$	57.71% \pm 7.59%, C=1.338	68.86% \pm 3.66%, C=9.258
Lemmas	76.29% \pm 4.34%, C=3.565	73.57% \pm 4.10%, C=6.683	76.29% \pm 4.25%, C=3.385, $\lambda=1.0000000000000002$	59.57% \pm 5.31%, C=0.4775	65.29% \pm 5.19%, C=8.772
PoS-tags	68.86% \pm 5.06%, C=2.264	67.57% \pm 6.79%, C=3.859	68.86% \pm 5.06%, C=2.264, $\lambda=1.0000000000000002$	60.43% \pm 5.15%, C=10.0	66.29% \pm 4.66%, C= 9.575
TreeString	66.29% \pm 5.83%, C=2.117	65.71% \pm 3.44%, C=1.7425	66.29% \pm 5.83%, C=2.117, $\lambda=1.0000000000000002$	56.29% \pm 2.80%, C=1.110	66.29% \pm 3.27%, C=3.698
TreeString (Tokens)	79.00% \pm 4.19%, C=1.213	74.14% \pm 3.35%, C= 9.201	79.00% \pm 4.19%, C=1.213, $\lambda=1.0000000000000002$	52.86% \pm 3.94%, C= 3.485	68.00% \pm 3.90%, C=7.875

Table 5-16: The optimized C values with the according accuracies (and standard deviations) obtained from the training runs in Experiment III.

Except for the B-o-C function, the obtained accuracies range from the lowest value of 65.71% up to a peak value of 79.00% which indicates that the training data can be separated quite well. Particularly outstanding is the 'tree string (tokens)' feature which achieves the highest accuracies in combination with all functions other than B-o-C. However, since the standard deviation is generally high with values up to 7.59% no reliable statements can be made regarding a particular weight function or feature type. On a side note: Because the parameter λ of the 'exponential decay' function has been learned to be close to 1 with regards to all features, the accuracies for 'constant weight' and 'exponential decay' in combination with a distinct feature are the same.

5.3.3 Testing phase

For all runs in the testing phase the last 435 examples from the first data set ('Bild.de') plus the last 582 examples from the second data set ('Spiegel.de') were used. Table 5-12 presents the classification results with all the combinations of feature type and weight function:

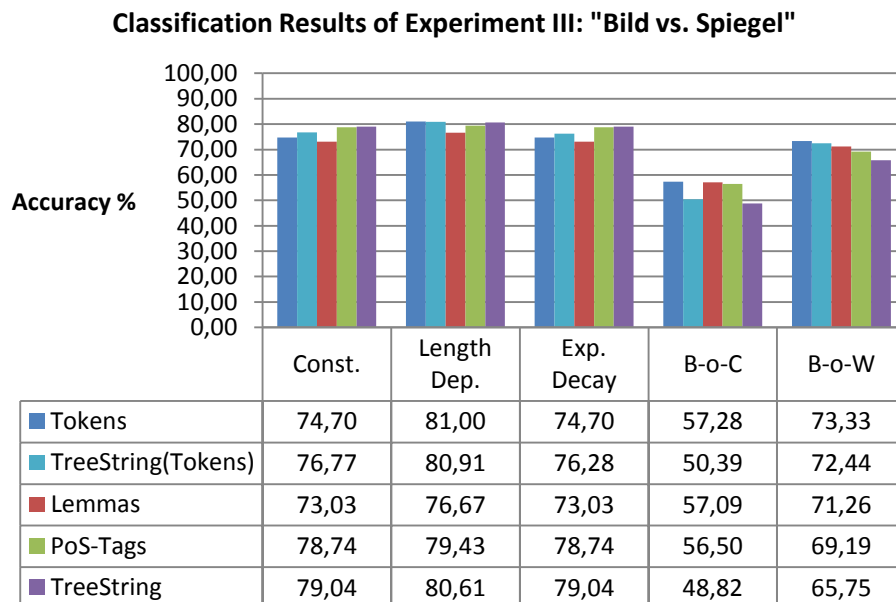


Table 5-17: Classification results (%) of Experiment III "Bild vs. Spiegel" obtained from different combinations of linguistic features and weight functions

The classification of unseen examples performs very well with the exception of the 'B-o-C' function, as before. The highest accuracies are achieved by the 'length dependent' function with the peak probability of 0.81 to predict the source of a previously unseen sentence correctly. The second highest values are achieved by the 'constant' and 'exponential decay' functions (again with same values as λ is close to 1).

When we take a look at the combination of the 'tree string' feature with the 'constant' and 'exponential decay' functions, we observe that this feature leads to the highest prediction performance (79,04%). This is particularly interesting as the 'tree string' feature consists of the grammatical structure of each sentence and is sufficient to achieve a good data separation.

5.4 Benchmark test of the 'Fast String Kernel' operator

In the last experiment the runtime performance of the FSK operator was measured. Therefore, very large example sets were prepared with sentences containing fix lengths of 500 and 1000 characters. In order to obtain such strings, a set of roughly 18,000 documents was annotated in the same way as in the previous experiments. Then, each sentence plus all the extracted features was appended as a single line to an output file. Since each document consisted of several sentences, the extraction operator obtained a resulting set of around 54,000 sentences.

The corresponding RapidMiner process, as well as the original and annotated output file can be found in the folder 'Experiments\ FSK_benchmark_test' on the CD enclosed with this work:

Input file:	RapidMiner process file:	Output file:
bringenB-N_K2.csv	Annotate&Extract_BringenB-N_K2_Write2CSV.rmp	bringenB-N-K2_annotated_sentences.csv

Table 5-18: The list of files used or created for preparing sentences of fix length

Then, this output file was duplicated twice. In the first copy each line was cut off after the 500th character with the help of an external text editing tool, and in the second copy each line was cut off after the 1000th character. Sentences that were shorter than the required lengths were removed.

In a simple RapidMiner process 1,000 to 5,000 strings were loaded from both prepared files, and the FSK operator computed the full kernel matrix while outputting the runtime statistics. The corresponding files are added to the folder 'Experiments\ FSK_benchmark_test', as well:

Input file:	RapidMiner process file:
Bringen_Sentences_500chars.csv	FSK_benchmark_test.rmp
Bringen_Sentences_1000chars.csv	

Table 5-19: The list of prepared text files used for the RapidMiner benchmark process

The FSK operator then computed the full kernel matrix between example sets of the same size ($n = 1000, 2000, \dots, 5000$). Further, the 'constant' weight function was chosen and the caching strategy set to 'cache_exampleSet_2' (Appendix A.4.2). The test was performed on Windows 7, on a mobile platform equipped with an Intel i7-3630QM@2.4Ghz CPU while RapidMiner was running on a single CPU core with a limited memory of 4 GB. No other processes were running in the background during the test.

Figure 5-7 shows the computation runtimes of different sized kernel matrices:

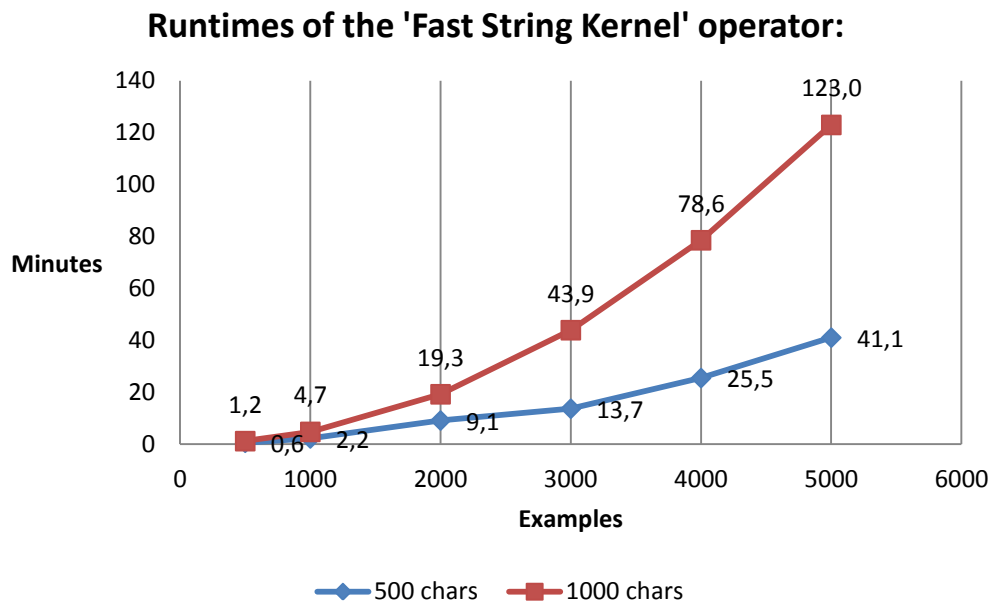


Figure 5-7: Benchmark results of the 'Fast String Kernel' operator with differently sized example sets with each set consisting of strings with 500 and 1000 characters

One may expect that the computation has a quadratic increase since the FSK operator always computes $n \times n$ many kernels, for a given set of n examples. However, we observe that the computation is noticeably below that which shows that the runtime clearly benefits from the implemented caching strategies.

Next, the runtimes of different processes involved in the kernel computation were measured. These processes are the **parsing** of input strings, the construction of **suffix trees** (including the computation of contributions), the building of **matching statistics**, computation of **similarities** (including the computation of weight function $W(x, t)$), and the time spent during **other processes** (function calls, object creation/destruction, etc.). Figure 5-8 presents the **averaged runtime fractions**¹⁹ of these subprocesses:

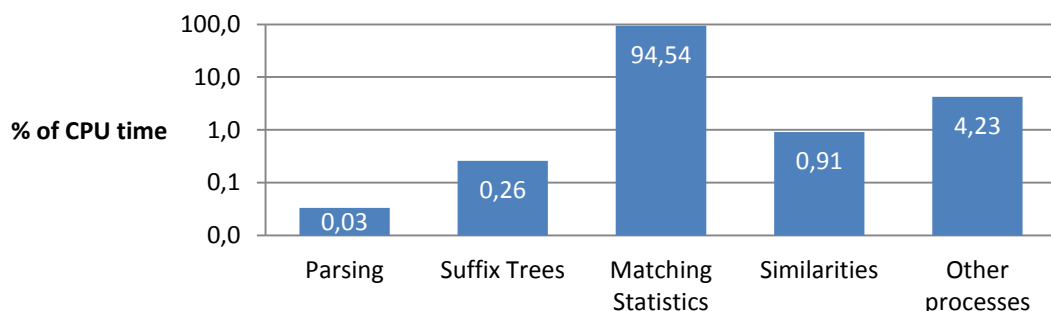


Figure 5-8: Runtime fractions during the kernel computations

¹⁹ These are the averaged runtimes obtained during the benchmark test.

Clearly, most of the runtime is spent on the computation of the matching statistics. These runtime relations are, however, biased due to the time spent on comparing $\theta(n \times m)$ many strings and due to caching strategies like 'cache_examplest_2' where only $\theta(n + m)$ many suffix trees are constructed.

The runtime performance of the process for matching statistics as shown by Figure 5-9 proves that this process is nonetheless independent of the amount of similarities computations. Regarding the slight increase in the runtime we may suppose that this was caused due to the increasing amount of matching substrings that were found with the increasing number of examples:

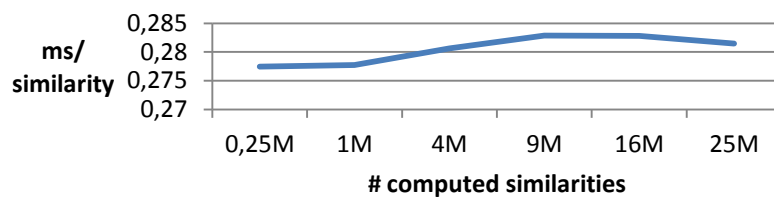


Figure 5-9: Runtime performance of the matching statistics process per similarity

Lastly, Figure 5-10 shows that the implemented algorithm for suffix tree construction runs in generally constant time. Here, we may assume that the slow, but steady decrease in the runtime may be due to caching effects caused by the operating system.

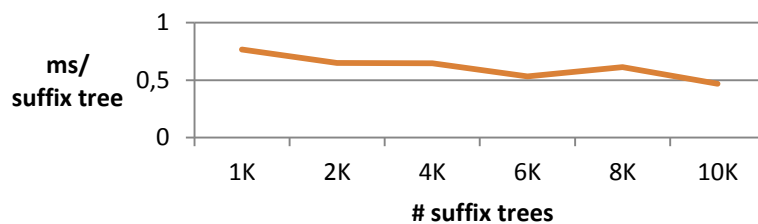


Figure 5-10: Runtime performance of the suffix construction process per suffix tree

Chapter 6 Summary and Outlook

The implemented operators for feature annotation and extraction make up a fast and robust preprocessing pipeline which allows us to obtain linguistic features from annotated texts. Furthermore, the extraction operator combines basic and structural features in order to produce variations of the 'tree strings' features which may prove useful in a given machine learning task. As shown in the experiments, the 'tree string (tokens)' feature lead to high classification performances that could compete with other basic features.

Special care has been taken to establish a flexible configuration framework that allows to enhance the annotation process with other WebLicht services, and with little adaptation effort other linguistic features could simply be integrated, as well.

The visualization module displays structural data given by dependency and constituency parse trees and additionally makes use of the basic features to enrich the visual representation of each sentence. At the same time, this tool can be used to manually label sentences.

With regards to machine learning, the 'Fast String Kernel' (FSK) operator has been implemented in order to achieve linear runtimes while computing string kernels of linguistic features. In combination with the SVM a machine learning framework was established that allows to perform text classification tasks based on freely selectable linguistic feature types.

The FSK operator builds suffix trees and matching statistics, but the implementation makes no use of compressed data structures like *suffix arrays* as suggested by [Vishnawathan & Teo]. However, the memory costs for constructed suffix trees could be circumvented by implementing effective caching mechanisms. This allows the operator to work with very little memory while still computing string kernels in linear time.

Based on pleasing performance results in the experiments, the implemented weight functions proved to be great assets for the FSK operator. However, in the limited time *TF-IDF weights* could not be integrated. These weights would have allowed to emphasize single terms regarding their frequency and information value against the background of all other matching terms. Other approaches like *dictionary weights* which make use of static dictionary matching could neither be taken into consideration. These weights would not only require to extend the algorithm for suffix tree construction, but would as well afford a manual preparation of dictionaries by tailoring them to the analyzed text corpora.

Due to the fact that not only the choice of a weight function, but also the choice of a specific feature type has a direct impact on the classification results, more room is assumed for studying other sophisticated features that could to be used in combination with the 'Fast String Kernel' method.

Appendix

A.1 RapidMiner operator 'WebLicht Feature Annotator'

A.1.1 Installation and usage in RapidMiner

The implementation of this RapidMiner operator is bundled in a jar-file "rmx_kobra_wlst-0.7.2.jar" (at the time of writing in the version 0.7.2). The file can simply be copied into the plugin folder of RapidMiner. After that, the list of operators in RapidMiner contains the group "KoBra WebLicht Service Tools" in which the "WebLicht Feature Annotator" can be found. This operator expects the RapidMiner data type "document" as its input, and produces the same output type that contains the annotated XML corpus according to the TCF specification.

Especially when annotating a large text corpus, it is recommended to save the output of the 'WebLicht Feature Annotator' to an XML file with the 'Write Document' operator,



Figure A - 1: An exemplary RapidMiner process using the 'WebLicht Feature Annotator'

and load the text later in a separate process (as shown in Figure A - 1). This is especially useful as it keeps the processing time in RapidMiner short, and additionally avoids unnecessary server load on the WebLicht services. The **synopsis of this operator** is given in the help description as follows:

Uploads a text document to a chain of remote WebLicht services and receives the response as an annotated text corpus (TCF). However, this operator delivers the TCF data as a document (doc). It is recommended to save the returned data as an XML with a "Write Document" operator. IMPORTANT: All WebLicht parsers require sentences in the TCF. Please make sure to select the proper Tokeniser!

A.1.2 Description of parameters

Figure A - 2 shows a screenshot of the parameters of the 'WebLicht Feature Annotator'. In this screenshot the settings are chosen to annotate a German text corpus with tokens, PoS-tags and a dependency parse trees.

A screenshot of the 'WLFeatureAnnotator (WebLicht Feature Annotator (Text to WebLicht TCF))' configuration window. It contains several dropdown menus for parameter selection:

- language: de
- WL ToolChain Selection: Conv. + Tok. + PoSTagger + DepP...
- WL Converter Selection (de): Converter #1
- WL Tokenizer Selection (de): Tokenizer #2
- WL PosTagger Selection (de): PoS-Tagger #1
- WL DepParser Selection (req.Sentenc...): Dependency Parser #1

Figure A - 2: Interactive list of parameters in the 'WebLicht Feature Annotator' realizing the concept of the flexible tool chain

The parameters of this operator parameter are as follows:

- **'language'**: This parameter is a drop list with all the available languages as defined in the attribute `availableLanguages="de,en,fr,it"` of the XML element `<services>` in the XML configuration (Section A.1.4). In the case that a WebLicht service does not support the selected language, the corresponding drop list below will not contain this service. Also, if none of the services of a specific category supports the selected language, the parameter drop list is not shown at all. Accordingly, the available languages are shown in the help description of the operator.
- **'WL Tool chain Selection'**: This parameter is a drop list with the following WebLicht (WL) tool chains:

- converter
- converter → tokeniser
- converter → tokeniser → lemmatiser
- converter → tokeniser → PoS tagger
- converter → tokeniser → PoS tagger → constituency parser
- converter → tokeniser → PoS tagger → dependency parser

Depending on which tool chain is selected, the according tool categories (converter, tokeniser, PoS tagger, dependency and constituency parser) appear as parameter drop lists where each drop list contains the WebLicht services of that category. For brevity, the parameters lists contain labels consisting of the tool category followed by the tool id of that category. The association of each label to a specific WebLicht service is displayed in the help description of this operator and uses the description of service as defined in the XML configuration.

- **'WL Converter Selection(<lang>')** – A list of converters:

Converter #1: [BBAoS&H] conversion of plaintext to TCF0.4,
Converter #2: [SfS] Converter, converts text in different document formats to TCF. If the language is specified as "unknown", the language is guessed from the text content. The following languages can be guessed: it, is, hu, th, sv, fr, ru, fi, ro, es, en, el, ee, pt, de, da, pl, bg, no, nl, lv

- **'WL Tokeniser Selection(<lang>')** – A list of tokenisers:

Tokeniser #1: [SfS] Tokeniser from the OpenNLP Project. ***No sentences are delivered!***,
Tokeniser #2: [SfS] Tokeniser/sentences from the OpenNLP project. The 'newlineBounds' parameter treats newlines as a hard break (a sentence boundary).
Tokeniser #3: [IMS] Czech, Slovenian, Hungarian, Italian, French, German, English tokeniser and sentence boundary detector,
Tokeniser #4: [BBAoS&H] tokenizes a text and splits it up into sentences

- **'WL Lemmatiser Selection(<lang>')** – A list of lemmatisers:

Lemmatizer #1: [IMS] SMOR lemmatizer: produces possible STTS tags and lemmas for a given list of words,

Lemmatizer #2: [IMS] PoS TreeTagger(2008): Italian,English,French,German part-of-speech tagger and lemmatiser,
Lemmatizer #3: [IMS] PoS TreeTagger(2013): Italian,English,French,German part-of-speech tagger and lemmatiser

- 'WL PoSTagger Selection(<lang>)' – A list of PoS taggers:

PoS-Tagger #1: [BBAoS&H] Part of Speech Tagger for German,
PoS-Tagger #2: [IMS] PoS TreeTagger(2008): Italian,English,French,German part-of-speech tagger and lemmatiser,
PoS-Tagger #3: [IMS] PoS TreeTagger(2013): Italian,English,French,German part-of-speech tagger and lemmatiser

- 'WL ConstParser Selection(requires Sentences!)(<lang>)' – A list of constituency parsers:

Constituency Parser #1: [SfS] Constituent Parser from the Berkeley NLP Project,
Constituency Parser #2: [IMS] German and English constituent parser

- 'WL DepParser Selection(requires Sentences!)(<lang>)' – A list of dependency parsers:

Dependency Parser #1: [IMS] Stuttgart Dependency Parser,
Dependency Parser #2: [SfS] MaltParser is a system for data-driven dependency parsing, which can be used to induce a parsing model from treebank data and to parse new data using an induced model. MaltParser is developed by Johan Hall, Jens Nilsson and Joakim Nivre at Växjö University and Uppsala University, Sweden.

A.1.3 XML scheme definition for the XML configuration of the tool chain

The following XML scheme definition [XSD] (bundled in the .jar file of this operator) formally specifies the allowable elements in the XML document for the configuration of the tool chain presented in the next Section 2.5.2:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="services">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="tool_group" maxOccurs="unbounded">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="tool" maxOccurs="unbounded">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="creator" type="xsd:string" maxOccurs="1" minOccurs="1" />
<xsd:element name="contact" type="emailAddress" maxOccurs="1" minOccurs="0" />
<xsd:element name="description" maxOccurs="1" minOccurs="1">
<xsd:complexType>
<xsd:simpleContent>
<xsd:extension base="xsd:string">
<xsd:attribute name="lang" type="xsd:string" />
```

```

    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
</xsd:element>
<xsd:element name="input_features" maxOccurs="1" minOccurs="1">
  <xsd:complexType>
    <xsd:attribute name="lang" type="xsd:string" />
    <xsd:attribute name="mime_type" type="xsd:string" />
    <xsd:attribute name="posttags.tagset" type="xsd:string" />
    <xsd:attribute name="type_param" type="xsd:string" />
    <xsd:attribute name="type_description" type="xsd:string" />
    <xsd:attribute name="version" type="xsd:string" />
  </xsd:complexType>
</xsd:element>
<xsd:element name="output_features" maxOccurs="1" minOccurs="1">
  <xsd:complexType>
    <xsd:attribute name="depparsing.emptytoks" type="xsd:string" />
    <xsd:attribute name="depparsing.multigovs" type="xsd:string" />
    <xsd:attribute name="depparsing.tagset" type="xsd:string" />
    <xsd:attribute name="lang" type="xsd:string" />
    <xsd:attribute name="mime_type" type="xsd:string" />
    <xsd:attribute name="parsing.tagset" type="xsd:string" />
    <xsd:attribute name="posttags.tagset" type="xsd:string" />
    <xsd:attribute name="type_description" type="xsd:string" />
    <xsd:attribute name="version" type="xsd:string" />
  </xsd:complexType>
</xsd:element>
<xsd:element name="pid" type="xsd:string" maxOccurs="1" minOccurs="1" />
<xsd:element name="url" type="xsd:anyURI" maxOccurs="1" minOccurs="1" />
<xsd:element name="url_params" type="xsd:string" maxOccurs="1" minOccurs="0" />
</xsd:sequence>
<xsd:attribute name="id" type="xsd:int" />
</xsd:complexType>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="category" type="xsd:string" />
</xsd:complexType>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="availableLanguages" type="xsd:string" />
</xsd:complexType>
</xsd:element>
<xsd:simpleType name="emailAddress">
  <xsd:restriction base="xsd:string">
    <xsd:pattern
      value="([0-9a-zA-Z][-\.\w]*[0-9a-zA-Z])*@[([0-9a-zA-Z][-\w]*[0-9a-zA-Z]\.)+[a-zA-Z]{2,9}" />
  </xsd:restriction>
</xsd:simpleType>
</xsd:schema>

```

A.1.4 XML configuration for storing available WebLicht services

```
<?xml version="1.0" encoding="UTF-8"?>
<services xsi:noNamespaceSchemaLocation="weblight_urls.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" availableLanguages="de,en,fr,it">
<tool_group category="converter">
  <tool id="1">
    <creator>Berlin-Brandenburg Academy of Sciences and Humanities</creator>
    <contact>didakowski@bbaw.de</contact>
    <description lang="en">[BBAoS&H] conversion of plaintext to TCF0.4</description>
    <input_features lang="de" mime_type="text/plain" type_description="text" version="" />
    <output_features lang="de" mime_type="text/tcf+xml" type_description="text" version="0.4" />
    <pid>http://fedora.deutschestextarchiv.de:8088/fedora/objects/WebLichtWebServices:2/
      datastreams/cmd/instance?asOfDateTime=2014-01-20T15:36:57Z</pid>
    <url>http://dspin.dwds.de:8080/services/rohling_v_0_4</url>
    <url_params></url_params>
  </tool>
  <tool id="2">
    <creator>SfS: Uni-Tuebingen</creator>
    <contact>wlsupport@sfs.uni-tuebingen.de</contact>
    <description lang="en">[SfS] Converter, converts text in different document formats to TCF. If the
      language is specified as "unknown", the language is guessed from the text content. The following languages
      can be guessed: it, is, hu, th, sv, fr, ru, fi, ro, es, en, el, ee, pt, de, da, pl, bg, no, nl, lv.</description>
    <input_features lang="de,en,fr,it,is,hu,th,sv,ru,fi,ro,es,el,ee,pt,da,pl,bg,no,nl,lv"
      mime_type="application/msword,application/vnd.openxmlformats-officedocument.wordprocessingml.
        document,application/pdf,text/plain,application/rtf"
      type_param="doc,pdf,plaintext,rtf" type_description="Word-Doc,Office-Doc,PDF,TXT,RTF" version="" />
    <output_features lang="" mime_type="text/tcf+xml" type_description="text" version="0.4" />
    <pid>11858/00-1778-0000-0004-BA56-7</pid>
    <url>http://weblight.sfs.uni-tuebingen.de/rws/service-converter/convert/qp</url>
    <url_params>informat=plaintext&language=de&outformat=tcf04</url_params>
  </tool>
</tool_group>
<tool_group category="tokeniser">
  <tool id="1">
    <creator>SfS: Uni-Tuebingen</creator>
    <contact>wlsupport@sfs.uni-tuebingen.de</contact>
    <description lang="en">[SfS] Tokeniser from the OpenNLP Project. ***No sentences are
      delivered!***</description>
    <input_features lang="de,en" mime_type="text/tcf+xml" type_description="text" version="0.4" />
    <output_features lang="" mime_type="" type_description="tokens" version="" />
    <pid>11858/00-1778-0000-0004-BA63-7</pid>
    <url>http://weblight.sfs.uni-tuebingen.de/rws/service-opennlp/annotate/tokens</url>
    <url_params></url_params>
  </tool>
  <tool id="2">
    <creator>SfS: Uni-Tuebingen</creator>
    <contact>wlsupport@sfs.uni-tuebingen.de</contact>
    <description lang="en">[SfS] Tokeniser/sentences from the OpenNLP project. The 'newlineBounds'
      parameter treats newlines as a hard break (a sentence boundary).</description>
    <input_features lang="de,en" mime_type="text/tcf+xml" type_description="text" version="0.4" />
    <output_features lang="" mime_type="" type_description="sentences,tokens" version="" />
    <pid>11858/00-1778-0000-0004-BA7B-4</pid>
  </tool>
</tool_group>
</services>
```

```

<url>http://weblicht.sfs.uni-tuebingen.de/rws/service-opennlp/annotate/tok-sentences</url>
<url_params>newlineBounds=false</url_params>
</tool>
<tool id="3">
  <creator>IMS: University of Stuttgart</creator>
  <contact>clarin@ims.uni-stuttgart.de</contact>
  <description lang="en">[IMS] Czech, Slovenian, Hungarian, Italian, French, German, English tokeniser
    and sentence boundary detector</description>
  <input_features lang="cz,si,hu,it,fr,de,en" mime_type="text/tcf+xml" type_description="text"
    version="0.4" />
  <output_features lang="" mime_type="" type_description="sentences,tokens" version="" />
  <pid>http://hdl.handle.net/11858/00-247C-0000-0007-3736-B</pid>
  <url>http://clarin05.ims.uni-stuttgart.de/cgi-bin/dspin/tokeniser4.perl</url>
  <url_params></url_params>
</tool>
<tool id="4">
  <creator>Berlin-Brandenburg Academy of Sciences and Humanities</creator>
  <contact>didakowski@bbaw.de</contact>
  <description lang="en">[BBAoS&H] tokenizes a text and splits it up into sentences</description>
  <input_features lang="de" mime_type="text/tcf+xml" type_description="text" version="0.4" />
  <output_features lang="" mime_type="" type_description="sentences,tokens" version="" />
  <pid>http://hdl.handle.net/11858/00-203C-0000-0023-21B9-7</pid>
  <url>http://dspin.dwds.de:8080/services/tokeniser_v_0_4</url>
  <url_params></url_params>
</tool>
</tool_group>
<tool_group category="lemmatizer">
  <tool id="1">
    <creator>IMS: University of Stuttgart</creator>
    <contact>clarin@ims.uni-stuttgart.de</contact>
    <description lang="en">[IMS] SMOR lemmatizer: produces possible STTS tags and lemmas for a given
      list of words</description>
    <input_features lang="de" mime_type="text/tcf+xml" type_description="tokens" version="0.4" />
    <output_features lang="" mime_type="" posttags.tagset="stts" type_description="lemmas" version="" />
    <pid>http://hdl.handle.net/11858/00-247C-0000-0007-373A-3</pid>
    <url>http://clarin05.ims.uni-stuttgart.de/cgi-bin/dspin/smor-lemmatizer4.perl</url>
    <url_params></url_params>
  </tool>
  <tool id="2">
    <creator>IMS: University of Stuttgart</creator>
    <contact>clarin@ims.uni-stuttgart.de</contact>
    <description lang="en">[IMS] PoS TreeTagger(2008): Italian,English,French,German part-of-speech
      tagger and lemmatiser</description>
    <input_features lang="it,en,fr,de" mime_type="text/tcf+xml" type_description="tokens" version="0.4" />
    <output_features lang="" mime_type="" posttags.tagset="stts" type_description="POStags, lemmas"
      version="" />
    <pid>http://hdl.handle.net/11858/00-247C-0000-0007-3739-5</pid>
    <url>http://clarin05.ims.uni-stuttgart.de/treetagger2008</url>
    <url_params></url_params>
  </tool>
  <tool id="3">
    <creator>IMS: University of Stuttgart</creator>

```

```

<contact>clarin@ims.uni-stuttgart.de</contact>
<description lang="en">[IMS] PoS TreeTagger(2013): Italian,English,French,German part-of-speech
tagger and lemmatiser</description>
<input_features lang="it,en,fr,de" mime_type="text/tcf+xml" type_description="tokens" version="0.4" />
<output_features lang="" mime_type="" posttags.tagset="stts" type_description="POStags, lemmas"
version="" />
<pid>http://hdl.handle.net/11858/00-247C-0000-0022-D906-1</pid>
<url>http://clarin05.ims.uni-stuttgart.de/treetagger</url>
<url_params></url_params>
</tool>
</tool_group>
<tool_group category="pos-tagger">
<tool id="1">
<creator>Berlin-Brandenburg Academy of Sciences and Humanities</creator>
<contact>didakowski@bbaw.de</contact>
<description lang="en">[BBAoS&H] Part of Speech Tagger for German</description>
<input_features lang="de" mime_type="text/tcf+xml" type_description="sentences,tokens" version="0.4" />
<output_features lang="" mime_type="" posttags.tagset="stts" type_description="POStags" version="" />
<pid>http://hdl.handle.net/11858/00-203C-0000-0023-21B4-2</pid>
<url>http://dspin.dwds.de:8080/services/tagger_v_0_4</url>
<url_params></url_params>
</tool>
<tool id="2">
<creator>IMS: University of Stuttgart</creator>
<contact>clarin@ims.uni-stuttgart.de</contact>
<description lang="en">[IMS] PoS TreeTagger(2008): Italian,English,French,German part-of-speech
taggerand lemmatiser</description>
<input_features lang="it,en,fr,de" mime_type="text/tcf+xml" type_description="tokens" version="0.4" />
<output_features lang="" mime_type="" posttags.tagset="stts" type_description="lemmas, POStags"
version="" />
<pid>http://hdl.handle.net/11858/00-247C-0000-0007-3739-5</pid>
<url>http://clarin05.ims.uni-stuttgart.de/treetagger2008</url>
<url_params></url_params>
</tool>
<tool id="3">
<creator>IMS: University of Stuttgart</creator>
<contact>clarin@ims.uni-stuttgart.de</contact>
<description lang="en">[IMS] PoS TreeTagger(2013): Italian,English,French,German part-of-speech
tagger and lemmatiser</description>
<input_features lang="it,en,fr,de" mime_type="text/tcf+xml" type_description="tokens" version="0.4" />
<output_features lang="" mime_type="" posttags.tagset="stts" type_description="lemmas, POStags"
version="" />
<pid>http://hdl.handle.net/11858/00-247C-0000-0022-D906-1</pid>
<url>http://clarin05.ims.uni-stuttgart.de/treetagger</url>
<url_params></url_params>
</tool>
</tool_group>
<tool_group category="constituency-parser">
<tool id="1">
<creator>SfS: Uni-Tuebingen</creator>
<contact>wlsupport@sfs.uni-tuebingen.de</contact>
<description lang="en">[SfS] Constituent Parser from the Berkeley NLP Project</description>

```

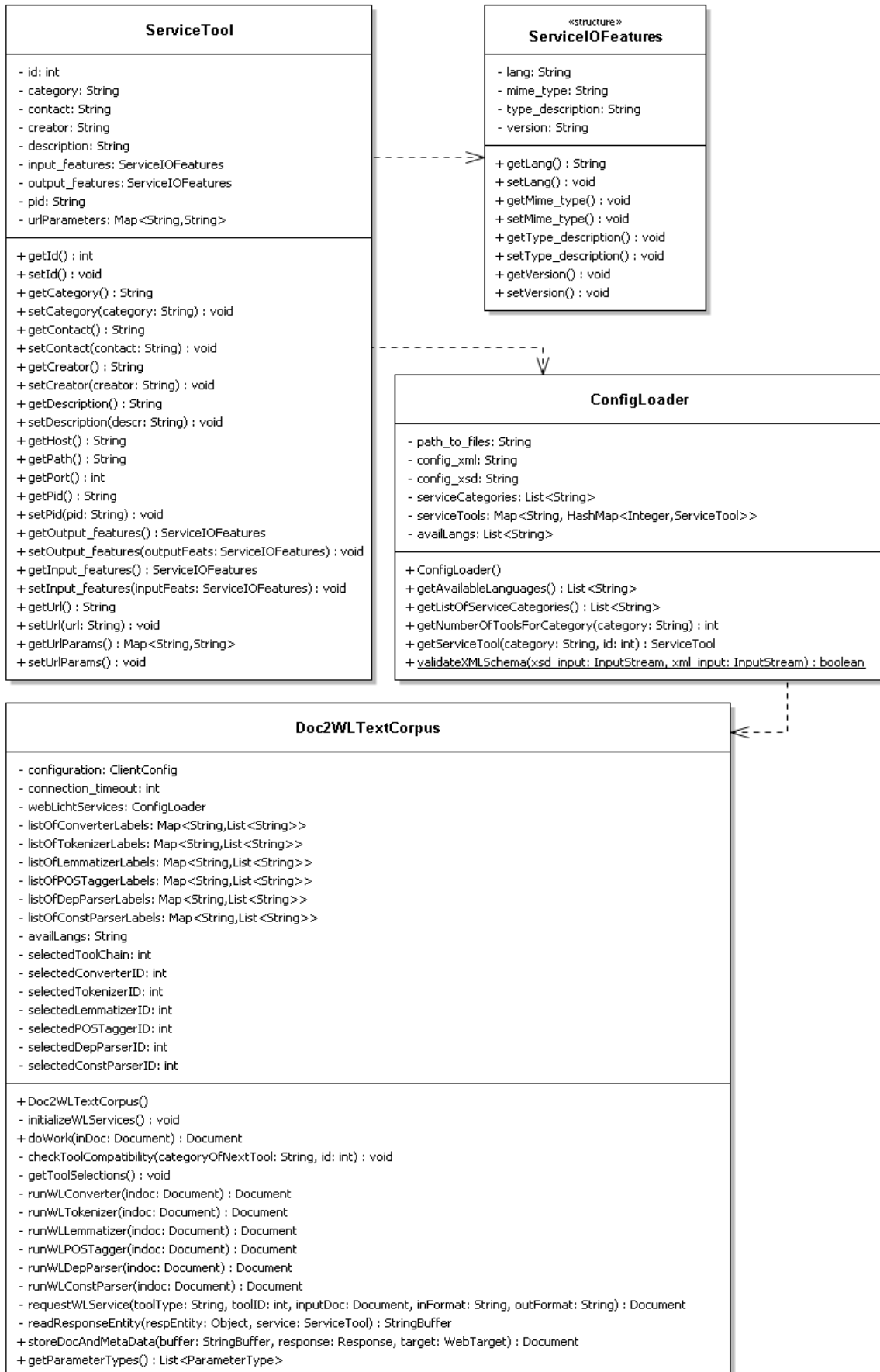
```

<input_features lang="de" mime_type="text/tcf+xml" type_description="sentences,tokens" version="0.4" />
<output_features lang="" mime_type="" parsing.tagset="tuebadztb" posttags.tagset="stts"
  type_description="constituents" version="" />
<pid>http://hdl.handle.net/11022/0000-0000-1CB2-8</pid>
<url>http://weblicht.sfs.uni-tuebingen.de/rws/BerkeleyParser_04/resources/parser</url>
<url_params></url_params>
</tool>
<tool id="2">
  <creator>IMS: University of Stuttgart</creator>
  <contact>clarin@ims.uni-stuttgart.de</contact>
  <description lang="en">[IMS] German and English constituent parser</description>
  <input_features lang="de,en" mime_type="text/tcf+xml" type_description="sentences,tokens"
    version="0.4" />
  <output_features lang="" mime_type="" parsing.tagset="tigertb" type_description="constituents"
    version="" />
  <pid>http://hdl.handle.net/11858/00-247C-0000-0007-3738-7</pid>
  <url>http://clarin05.ims.uni-stuttgart.de/cgi-bin/dspin/bitpar4.perl</url>
  <url_params></url_params>
</tool>
</tool_group>
<tool_group category="dependency-parser">
  <tool id="1">
    <creator>IMS: University of Stuttgart</creator>
    <contact>clarin@ims.uni-stuttgart.de</contact>
    <description lang="en">[IMS] Stuttgart Dependency Parser</description>
    <input_features lang="de" mime_type="text/tcf+xml" type_description="sentences,tokens" version="0.4" />
    <output_features depparsing.emptytoks="false" depparsing.multigovs="false" depparsing.tagset="tiger"
      lang="" mime_type="" posttags.tagset="stts" type_description="dependencies" version="" />
    <pid>http://hdl.handle.net/11858/00-247C-0000-0007-3734-F</pid>
    <url>http://ws1-clarind.esc.rzg.mpg.de/webservice-parser</url>
    <url_params></url_params>
  </tool>
  <tool id="2">
    <creator>SfS: Uni-Tuebingen</creator>
    <contact>wlsupport@sfs.uni-tuebingen.de</contact>
    <description lang="en">[SfS] MaltParser is a system for data-driven dependency parsing, which can be
      used to induce a parsing model from treebank data and to parse new data using an induced model.
      MaltParser is developed by Johan Hall, Jens Nilsson and Joakim Nivre at Växjö University and Uppsala
      University, Sweden.</description>
    <input_features lang="de" mime_type="text/tcf+xml" posttags.tagset="stts"
      type_description="sentences,tokens" version="0.4" />
    <output_features depparsing.emptytoks="false" depparsing.multigovs="true"
      depparsing.tagset="tuebadz" lang="" mime_type="" type_description="dependencies" version="" />
    <pid>http://hdl.handle.net/11022/0000-0000-1D4D-B</pid>
    <url>http://ws1-clarind.esc.rzg.mpg.de/webservice-parser</url>
    <url_params>depparsing.multigovs=true&amp;depparsing.tagset=tuebadz</url_params>
  </tool>
</tool_group>
</services>

```

A.1.5 Class diagram of the 'WebLicht Feature Annotator'

Depending classes involved in the implementation of the 'WebLicht Feature Annotator' (implemented by the class *'Doc2WL TextCorpus'*):



A.2 RapidMiner operator 'WebLicht TCF to ExampleSet'

A.2.1 Installation and usage in RapidMiner

Equally to the 'WebLicht Feature Annotator', the implementation of 'WebLicht TCF to ExampleSet' operator is bundled in the jar-file "rmx_kobra_wlst-0.7.2.jar" and installed in the same way as described in Section A.1.1.

This RapidMiner operator accepts the data type "document" as input and produces an 'ExampleSet'. Figure A - 3 shows the beginning of a process in which an annotated text corpus is loaded by a 'Read Document' operator. Directly after, the 'WLTCF2ExampleSet' operator processes all the linguistic features that are available in the document (as described in Section 2.6.2), and extracts them to different columns of a new 'ExampleSet' which is then the output of this operator.

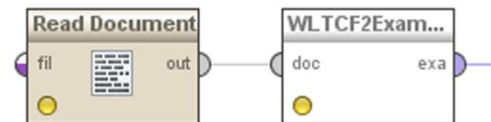


Figure A - 3: An exemplary RapidMiner process using the operator 'WebLicht TCF to ExampleSet'

The columns of the resulting 'ExampleSet' can contain the following features (depending on the annotated text corpus): Sentence, Tokens, Lemmas, PosTags, TreeString, PoSTags(tree), Tokens(tree), Lemmas(tree), Label. The **synopsis of this operator** is given as follows:

Extracts features from a text corpus (TCF) (previously annotated by using the 'WebLicht Feature Annotator'), and outputs them to an ExampleSet. Additionally, offers the option to label an example with a user-defined attribute. The default value of the label can be defined as "-1" or "1".

A.2.2 Description of parameters

The parameters of the 'WebLicht TCF to ExampleSet' operator shown in Figure A - 4 are described as follows:

- **'add a label attribute'**: If checked, the next two parameter fields appear and a binary label can be associated to each example in the ExampleSet.
- **'label attribute'**: Defines the name of the label attribute.
- **'default value'**: Default value of the label. Accepts a binomial value of either "-1" or "1".

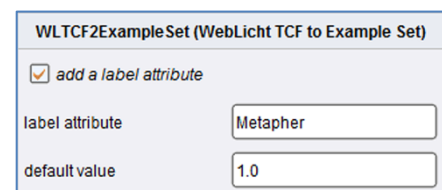


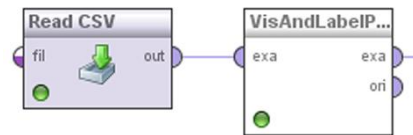
Figure A - 4: Parameters of the 'WebLicht TCF to ExampleSet' operator.

A.3 RapidMiner operator 'Visualize and Label Parse Trees'

A.3.1 Installation and usage in RapidMiner

The implementation of the RapidMiner operator 'Visualize and Label Parse Trees' is bundled in the jar-file "rmx_kobra_wlst-0.7.2.jar" and the operator is installed as described in Section A.1.1.

Figure A - 5 shows a RapidMiner process in which an ExampleSet that contains various linguistic features is loaded by a 'Read CSV' operator, and directly after



visualized by the 'Visualize and Label Parse Trees' operator.

Figure A - 5: An exemplary RapidMiner process using the 'Visualize and Label Parse Trees' operator

The **synopsis of this operator** is given as follows:

Visualize and label parse trees of sentences that were previously annotated by a WebLicht parser. This operator can only be used directly after the 'WebLicht TCF to Example Set' operator or by providing an ExampleSet that contains the attributes produced by the 'WebLicht TCF to Example Set' operator (...). Other feature types that have been annotated by the 'WebLicht Feature Annotator', are associated to the nodes in that parse tree and can additionally be shown in the visualization.

An exemplary visualized (dependency) parse tree is shown in Figure A - 6. By clicking on a checkbox in the controls of the panel the user can manipulate the label of the currently presented sentence/parse tree if the user has added a label in the parameter settings of the 'WebLicht TCF to ExampleSet' operator. By clicking the 'Finish' button, the set of all examples is then outputted.

The control buttons in the top area of the panel allow the user to switch to the first, previous, next or last sentence. This triggers the panel to clear the drawing canvas and update it with a parse tree that corresponds to the new selected sentence.

By clicking and holding the mouse on the canvas, the drawing can be translated inside the panel. Using the mouse wheel allows the user to resize the parse tree. Therefore, in cases of too large drawings, the user is able to navigate all of the subtrees or scale down the graph in order to view the full structure of the parse tree.

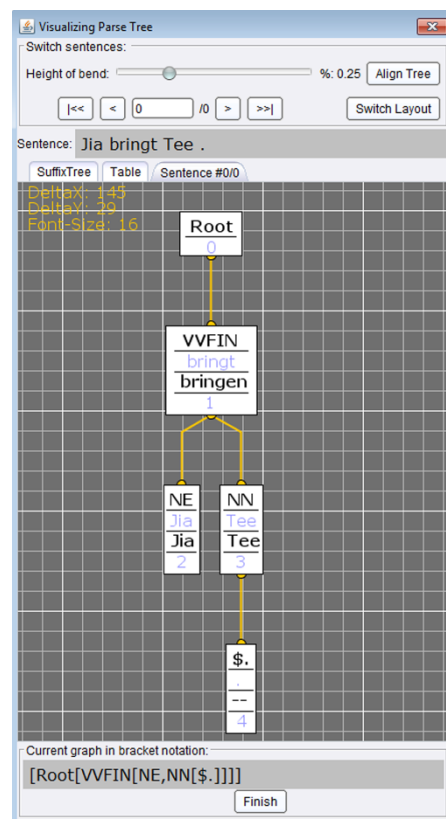


Figure A - 6: The visualization panel with an exemplary parse tree

The button 'Switch Layout' toggles the tree layout to change from a vertical arrangement of the nodes to a horizontal one. That is, nodes on the same depth are now placed on the same x-coordinate while keeping an alphabetical order in a top-down manner.

The control button "Align Tree" places the tree to the left top edge of the panel. To adjust the edges of the tree to be direct or parallel lines, the position of the bends can be changed so that all edges either connect two boxes in straight lines or the bend is positioned on the x-coordinate of the lower box and gradually between the y-coordinates of both boxes.

The drawing of the parse tree (for the current selected sentence) is placed in a sub window which is also called "tab widget". This tab widget carries the label "Sentence" together with the number of the selected sentence, followed by the maximum number of sentences. Additionally, the tab widget "Table" provides a list with the extracted features. Note that the PoS-tags in this table are not necessarily the same tags produced by a parser service in the tool chain, especially in the case where the text corpus is annotated by a constituency parser.

A.3.2 Description of parameters

The parameters of the 'Visualize and Label Parse Trees' operator shown in Figure A - 7 are described as follows:

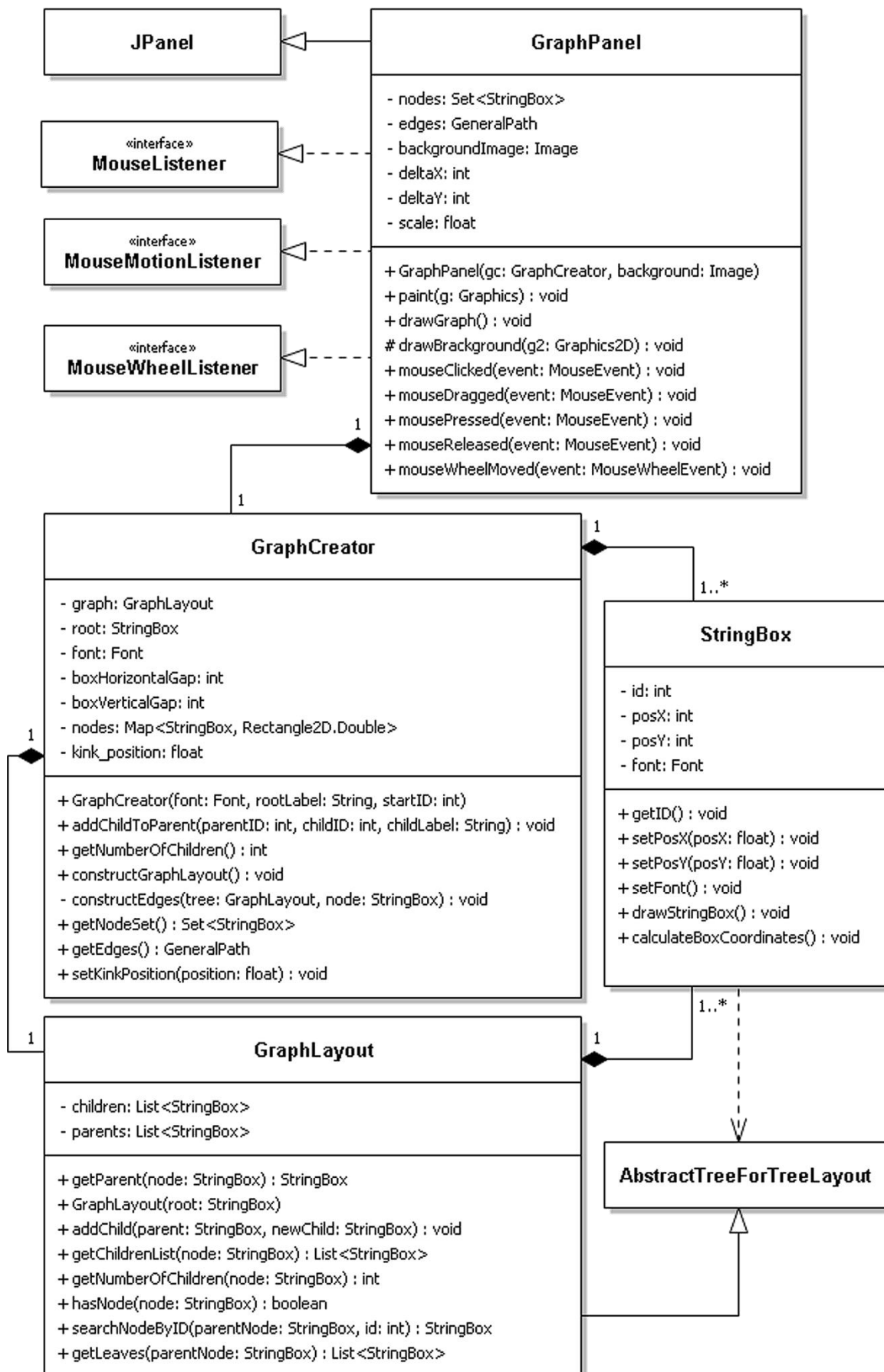
- 'first example' / 'last example': These parameters allow the user to define a range of examples for the viewing process.
- 'invert filter': This parameter inverts the selection of examples which is the ExampleSet without the range of Examples from first example i to last example j .
- 'unique char for string termination': This parameter defines the termination symbol during the visualization of the parse tree as suffix tree. This character must not be contained in any of the strings in the ExampleSet.

VisAndLabelParseTrees (Visualize and Label Parse Trees)	
first example	1
last example	5000
<input type="checkbox"/> invert filter	
unique char for string termination	\$
<input checked="" type="checkbox"/> show lemmas	
<input checked="" type="checkbox"/> show tokens	
sentences attribute	Sentence
treestring attribute	TreeString
posTags(tree) attribute	PoSTags(tree)
lemma(tree) attribute	Lemmas(tree)
tokens(tree) attribute	Tokens(tree)
label attribute	Label

Figure A - 7: Parameters of the 'Visualize and Label Parse Trees' operator

- 'show lemmas' / 'show tokens': If available these feature types will be placed in the boxes of the visualized parse tree.
- 'sentence attribute' / 'treestring attribute' / 'posTags(tree) attribute' / 'lemma(tree) attribute' / 'tokens(tree) attribute': The attribute name for each feature type in the ExampleSet as provided by the 'WebLicht TCF to ExampleSet' operator.
- 'label attribute': The label of the ExampleSet as chosen by the user in the 'WebLicht TCF to ExampleSet' operator

A.3.3 Class diagram of the visualization framework for drawing parse trees



A.4 RapidMiner operator 'Fast String Kernel'

A.4.1 Installation and usage in RapidMiner

The implemented '*Fast String Kernel*' operator is bundled in the jar-file "rmx_fast_string_kernels-0.7.1.jar". In order to use this operator, the file has to be placed into the plugin folder of RapidMiner. After that, a new group 'Fast String Kernels' appears in the of list of RapidMiner operators in which the '*Fast String Kernel*' can then be found. This operator expects one or two "ExampleSet(s)" as its input, and outputs the same data structure. Furthermore, the data type of the attribute from the ExampleSet(s) that is required for the string kernel computation needs to be *text*.

This operator expects that either the first or both input ports are connected. If two ExampleSets are connected, the 'Fast String Kernel' computes the similarities $sim(i, j)$ between the n strings from the first set and the m strings from the second set, with $1 \leq i \leq n$ and $1 \leq j \leq m$. If only one ExampleSet is connected, the similarity computation compares all strings with all other strings in that set while the operator only needs to calculate the upper triangle matrix with $1 \leq i \leq n$ and $i \leq j \leq n$ and mirrors the values to the lower triangle matrix. In this special case, the warning shown in RapidMiner regarding this operator can be ignored. The result output of each operator is an ExampleSet that stores the computed kernel matrix with the entries $sim(i, j)$ in n columns and m rows. Additionally, the operator adds a column "id" to the ExampleSet with integer values that indicate the current row in the kernel matrix.

Figure A - 8 shows an exemplary RapidMiner process in which two 'ExampleSets' (each contains a column with strings) are loaded by a 'Read CSV' operator. The 'Multiply' operator duplicates the first ExampleSet which is then forwarded to the upper 'Fast String Kernel' for computing the similarities between strings in that set, while the second 'Fast String Kernel' computes the similarities between the first and second set.

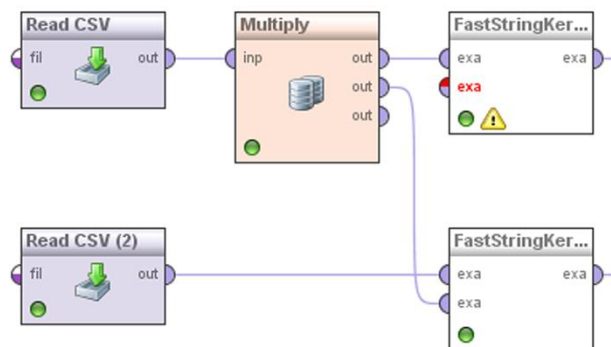


Figure A - 8: An exemplary RapidMiner process with two 'Fast String Kernel' operators.

The **synopsis of the 'Fast String Kernel' operator** is given as follows:

Extracts features from a text corpus (TCF) (previously annotated by using the 'WebLicht Feature Annotator'), and outputs them to an ExampleSet. Additionally, offers the option to label an example with a user-defined attribute. The default value of the label can be defined as "-1" or "1".

A.4.2 Description of parameters

The parameters of the 'Fast String Kernel' operator shown in Figure A - 9 are described as follows:

- 'compare string x/y': Defines the name of the input strings x and y .
- 'unique char for string termination': This parameter defines the termination symbol for the suffix tree. This character must not be contained in any of the input strings in the ExampleSet.
- 'remove characters from string': If checked, the next parameter field defines the set of characters to be removed from the input string.
- 'weight function': Any found match can be weighted by one of the implemented functions (see Section 4.7.4): "constant", "length_dependent", "exponential", "bag_of_characters" and "bag_of_words".
- 'weight per item': Weighting parameter depending on the chosen weight function.
- 'log total computation time': Logs the total runtime of the fast string kernel operator.
- 'log similarities': Logs the calculated similarity values of all compared pairs of input strings.
- 'log kernel computation': Outputs the summed and normalized kernel value after each comparison of the strings (x_i, y_j) , with $x_{1 \leq i \leq n}$ and $y_{1 \leq j \leq m}$.
- 'log kernel computation details': Outputs any matched substring that the kernel finds during the similarity computation. Enabling this option usually leads to a reduced runtime performance of RapidMiner, due to the amount of produced logging messages. This option should be used for "debugging" data or for small data sets only.
- 'precache strategy': We provide the following caching strategies that are applied during the kernel computation: "Cache_ExampleSet_2" (default), "window" and "no_caching". Detailed descriptions are given in Section (4.8).

FastStringKernel (Fast String Kernel)	
compare string x	stringX
compare string y	stringX
unique char for string termin...	\$
<input checked="" type="checkbox"/> remove characters from string	
characters to remove from in...	", : ; ' " ^ \ { } € \$ % & @ #
weight function	LENGTH_DEPENDENT
weight per item	1.0
<input type="checkbox"/> log total computation times	
<input type="checkbox"/> log similarities	
<input type="checkbox"/> log kernel computation	
<input type="checkbox"/> log kernel computation details (use with caution)	
precache strategy	CACHE_EXAMPLESET_2

Figure A - 9: Parameters of the 'Fast String Kernel' operator.

References

Literature

- [Akjaman et al.] A. Akmajian, R. A. Demers, A. K. Farmer, R. M. Harnish: *Linguistics*, MIT Press, p239, 2001.
- [Baeza-Yates et al.] R. A. Baeza-Yates, B. Ribeiro-Neto: *Modern Information Retrieval*. Addison Wesley Longman, 1999.
- [Battista et al.] G. Di Battista, P. Eades, R. Tamassia, I.G. Tollis: *Graph Drawing*. Prentice Hall, 1999.
- [Bertin] J. Bertin: *Graphische Semiologie*. Walter de Gruyter, Berlin, 1974.
- [Bhanji et al.] S. Bhanji, H. C. Purchase, R. F. Cohen and M. James: *Validating Graph Drawing Aesthetics: A Pilot Study*. Technical Report 336, University of Queensland, Department of Computer Science, 1995.
- [Brill] E. Brill: *A simple rule-based part-of-speech tagger*. In: Proceedings of the 3rd Conference on Applied Natural Language Processing (ANLP-92), pages 152-155, 1992.
- [Buchheim et al.] C. Buchheim, M. Jünger, S. Leipert: *Improving Walker's Algorithm to Run in Linear Time*. In: Proc. GD'02, Springer LNCS 2528, pp. 344 353, 2002.
- [Burges] J. C. Burges: *A tutorial on Support Vector Machines for Pattern Recognition*. In: Journal of Data Mining and Knowledge Discovery, Volume 2 Issue 2, pages 121-167, June 1998.
- [Chang & Lawler] W. I. Chang, E. L. Lawler: *Sublinear approximate string matching and biological applications*. Algorithmica, 12(4/5), 327 344, 1994.
- [Collins et al.] M. Collins and N. Duffy: *Convolution kernel for natural language*. In Advances in Neural Information Processing Systems (NIPS), volume 16, pages 625 632, 2002.
- [Croce et al.] D. Croce, A. Moschitti, R. Basili: *Structured Lexical Similarity via Convolution Kernels on Dependency Trees*. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 1034 1046, 2011.
- [Fletcher] R. Fletcher: *Practical Methods of Optimization*. John Wiley and Sons, Inc., 1987.
- [Garside] R. Garside, G. Leech, A. McEnery: *Corpus annotation. Linguistic information from computer text corpora*. Addison Wesley Longman, 1997.
- [Gusfield] D. Gusfield: *Algorithms on Strings, Trees and Sequences. Computer Science and Computational Biology*. Cambridge University Press New York, 1997.
- [Heid et al.] Ulrich Heid, Helmut Schmid, Kerstin Eckart, and Erhard Hinrich: *A corpus representation format for linguistic web services: The D-SPIN text corpus format and its relationship with ISO standards*. In: Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10), 2010.
- [Hinrichs] E. W. Hinrichs, M. Hinrichs, T. Zastrow: *WebLicht: Web-Based LRT Services for German*. In: Proceedings of the ACL 2010 System Demonstrations. pages 25 29, 2010.
- [Hintikka] J. Hintikka: *Aspects of Metaphor*, Springer, 1994, p41.
- [Hofmann et al.] T. Hofmann; B. Schölkopf; A. J. Smola: *Kernel methods in machine learning*. In: The Annals of Statistics 36, no. 3, 1171 1220, 2008.
- [Hoffmann 2006] H. Hoffmann: *Kernel PCA for novelty detection*. In: Journal of Pattern Recognition, Volume 40 Issue 3, pages 863 874, March 2007.
- [Jackson et al.] H. Jackson, E. Zñ Amvela: *Words, Meaning, and Vocabulary*, Continuum, p14, 2000.
- [Joachims 1998] T. Joachims: *Text categorization with support vector machines*. In: European Conference on Machine Learning (ECML). Springer Verlag, 1998.
- [Joachims 2000] T. Joachims: *The Maximum-Margin Approach to Learning Text Classifiers - Methods, Theory and Algorithms*. Dissertation at the LS8 Faculty of Computer Sciences, Technical University Dortmund, 2000.
- [Knuth] D. E. Knuth: *The Art of Computer Programming. Fundamental Algorithms, Vol. 1*. Addison-Wesley, Reading, Massachusetts, 2nd Ed., 1995.
- [Leslie et al.] C. Leslie, E. Eskin, W. Noble: *The Spectrum Kernel: A String Kernel for SVM Protein Classification*. Pacific Symposium of Biocomputing, 2002.
- [Lodhi et al.] H. Lodhi, C. Saunders, J. Shawe-Taylor; N. Cristianini, C. Watkins: *Text classification using string kernels*. In: Journal of Machine Learning Research, 419 444, 2002.

- [Markou1] M. Markou, S. Singh: *Novelty detection: A review, part 1*. In: Statistical approaches, Signal Processing 83 (12), 2481–2497, 2003.
- [Markou2] M. Markou, S. Singh: *Novelty detection: A review, part 2*. In: Neural network based approaches, Signal Processing 83 (12), 2499–2521, 2003.
- [Neumann] G. Neumann: *Einführung in die Dependenzgrammatik*. LT lab, DFKI. University Saarland, 2. July 2013.
- [Purchase et al.] H. C. Purchase, R. F. Cohen and M. James: *Validating Graph Drawing Aesthetics*, In: Graph Drawing (Proc. GD '95), vol. 1027 of Lecture Notes Comput. Sci., pp. 435–446, Springer Verlag, 1996.
- [Reingold&Tilford] E.M. Reingold, J.S. Tilford: *Tidier drawing of trees*. IEEE Trans. on Software Engineering, Vol. SE-7(2), pp. 223–228, 1981.
- [Richards] I. A. Richards: *The Philosophy of Rhetoric*. Oxford University Press, 1936.
- [Santorini] B. Santorini: *Part-of-speech tagging guidelines for the Penn Treebank project*. 3rd revision, 1990.
- [Shawe-Taylor & Cristianini] J. Shawe-Taylor, N. Cristianini: *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.
- [Sonnenburg et al.] S. Sonnenburg, K. Rieck, Fraunhofer First, Ida, G. Rätsch: *Large Scale Learning with String Kernels*. MIT Press, pp. 73–103, 2007.
- [Schiller et al.] A. Schiller, S. Teufel, C. Stöckert, C. Thielen: *Guidelines für das Tagging deutscher Textcorpora mit STTS (Kleines und großes Tagset)*, 1999.
- [Schmid] H. Schmid: *Probabilistic part-of-speech tagging using decision trees*. In: Proceedings of the International Conference on New Methods in Language Processing, 1994.
- [Schölkopf1998] B. Schölkopf, A. J. Smola, K.-R. Müller: *Nonlinear component analysis as a kernel eigenvalue problem*. In: Neural Computation 10, 1299–1319, 1998.
- [Schölkopf2002] B. Schölkopf, A. J. Smola: *Learning with Kernels*. MIT Press, Cambridge, MA, 2002.
- [Stanford CoreNLP] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, D. McClosky: *The Stanford CoreNLP Natural Language Processing Toolkit*. In Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pp. 55–60, 2014.
- [StatLearn] T. Hastie, R. Tibshirani, J. Friedman: *The Elements of Statistical Learning*. Springer Verlag, 2nd ed., 2009.
- [Ukkonen] E. Ukkonen: *On-line construction of suffix trees*. Algorithmica 14 (3), 249–260, 1995.
- [Walker] J. Q. Walker II: *A Node-Positioning Algorithm for General Trees*. Software Practice and Experience 10, 553–561, 1990.
- [Vishwanathan & Smola] A.J. Smola, S.V.N. Vishwanathan: *Fast Kernels for String and Tree Matching*. In: Neural Information Processing Systems, 569–576, 2002.
- [Vishwanathan & Teo] C. H. Teo, S. V. N. Vishwanathan: *Fast and space efficient string kernels using suffix arrays*. In: Proceedings, 23rd ICMP, 929–936, ACM Press, 2006.

URLs

[Bild]	http://www.bild.de
[BSD]	https://en.wikipedia.org/wiki/BSD_licenses
[Clarin-d]	http://clarin-d.de/de/home.html
[dependencies]	http://en.wikipedia.org/wiki/Dependency_grammar#Representing_dependencies
[DepConst]	http://de.wikipedia.org/wiki/Dependenzgrammatik#Dependenz_vs._Konstituenz
[DWDS]	http://www.dwds.de/
[hierarchy]	http://weblicht.sfs.uni-tuebingen.de/weblichtwiki/index.php/Tools_in_Detail#Hierarchies_of_Linguistic_Tools
[HTTP]	https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
[hyponymy & hypernymy]	https://en.wikipedia.org/wiki/Hyponymy_and_hypernymy
[JAX-RPC]	https://java.net/projects/jax-rpc/
[JAX-RS]	https://jax-rs-spec.java.net/
[JAXP]	https://jaxp.java.net/
[JordanCurve]	http://en.wikipedia.org/wiki/Jordan_curve_theorem
[KobRA]	http://www.kobra.tu-dortmund.de/mediawiki/index.php?title=Projektbeschreibung/Methode
[MIME]	http://en.wikipedia.org/wiki/MIME
[NEGRA]	http://www.coli.uni-saarland.de/projects/sfb378/negra-corpus/negra-corpus.html
[TK_Lingua]	http://www.linguastream.org/
[TK_Mate]	http://www.ims.uni-stuttgart.de/forschung/ressourcen/werkzeuge/matetools.html
[TK_Monty]	http://web.media.mit.edu/~hugo/montylingua/
[TK_NLTK]	http://www.nltk.org/
[TK_OpenNLP]	http://opennlp.apache.org/index.html
[TK_Stanford]	http://nlp.stanford.edu/software/corenlp.shtml
[OutlineNLP]	http://en.wikipedia.org/wiki/Outline_of_natural_language_processing
[SO_Ukkonen]	http://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english/
[Spiegel]	http://www.spiegel.de
[stemming]	http://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html
[tasksNLP]	http://en.wikipedia.org/wiki/Natural_language_processing
[TCFSpec]	http://weblicht.sfs.uni-tuebingen.de/englisch/tutorials/html/index.html
[TCF0.3Parser]	http://weblicht.sfs.uni-tuebingen.de/englisch/tutorials/html/wlservices/index.html
[Tiger]	http://www.ims.uni-stuttgart.de/forschung/ressourcen/korpora/TIGERCorpus/annotation/tiger_introduction.pdf
[tokeniser]	http://weblicht.sfs.uni-tuebingen.de/weblichtwiki/index.php/Tools_in_Detail#Tokeniser
[treeLayout]	https://code.google.com/p/treelayout/
[treeTraversal]	https://en.wikipedia.org/wiki/Tree_traversal#Pre-order
[WebLicht]	CLARIN-D/SfS-University of Tübingen: <i>WebLicht: Web-Based Linguistic Chaining Tool.</i> https://weblicht.sfs.uni-tuebingen.de/
[Wget]	https://en.wikipedia.org/wiki/Wget , https://www.gnu.org/software/wget/
[XML]	http://en.wikipedia.org/wiki/XML
[XSD]	http://en.wikipedia.org/wiki/XML_Schema_(W3C)

List of Figures

Figure 1-1: A processing pipeline concept combining feature preparation steps and	8
Figure 2-1: The result of a constituency parser for a German sentence.....	15
Figure 2-2: Different parsing results of the constituent parser of the NLP project and the Stanford Core NLP parser.....	15
Figure 2-3: Result of the Stuttgart Dependency Parser for an German example sentence.....	16
Figure 2-4: Different conventions to draw dependency trees.....	16
Figure 2-5: Difference between a dependency and a constituency tree.....	16
Figure 2-6: An exemplary preprocessing tool chain in WebLicht.....	21
Figure 2-7: Communication with WebLicht services by using the HTTP POST protocol.....	22
Figure 2-8: The concept of a flexible tool chain for the 'WebLicht Feature Annotator'.....	25
Figure 2-9: Combinations of supported tool chains.....	29
Figure 2-10: The extraction process of linguistic features from a text corpus.....	31
Figure 2-11: Pre-order traversal of an exemplary tree.....	33
Figure 2-12: An interdisciplinary perspective on an exemplary data mining task.....	37
Figure 2-13: An exemplary semantic field of the term "purple".....	39
Figure 3-2: Grid types for trees with different ways to draw components and connections.....	43
Figure 3-1: The implemented representation of a vertex in parse trees.....	43
Figure 3-3: An exemplary tree T with a very low readability.....	44
Figure 3-4: The reduced tree T' of T with a high readability.....	44
Figure 3-5: An exemplary rooted tree whose nodes are placed by the <i>Layered-Tree-Draw</i> Algorithm along the x-axis and by the layer assignment along the y-axis. The steps beneath the drawing describe the construction of subtree T	47
Figure 3-6: Compacting subtrees along their contours during the conquer step in the "Reingold&Tilford"-Algorithm.....	48
Figure 3-7: The exemplary tree from Figure 3-5 drawn by the "Reingold&Tilford" Algorithm.....	49
Figure 3-8: A conceptual dialog presenting lists of visualized parse trees that can be labeled.....	50
Figure 3-9: A tidy and compact drawing of an exemplary dependency parse tree for an exemplary German sentence.....	51
Figure 4-1: A machine learning framework with different kernel methods used by the SVM in a supervised learning surrounding.....	52
Figure 4-2: An exemplary set $X \subseteq \mathbb{R}^2$ of non-linear separable examples.....	54
Figure 4-3: Data set X mapped to the feature space \mathcal{F} after applying a mapping function Φ	54
Figure 4-4: Linear separating hyper-planes for the separable case. Encircled points represent the support vectors.....	57
Figure 4-5: Hyperplanes in the non-separable case. Encircled points represent the support vectors.....	60
Figure 4-6: Shared subtrees in two parse trees; the numbers in brackets indicate the number of occurrences for each shared subtree pair.....	65
Figure 4-7: The standard trie of a set S of strings.....	67
Figure 4-8: The compressed trie of a set S of strings.....	67
Figure 4-9: The suffix tree $S(x)$ for the string $x = "ababc\$"$	68
Figure 4-10: The compact representation of the suffix tree $S(x)$	68

Figure 4-11: Weighting an exemplary matching substring $y = "ba"$ in the suffix tree $S(x)$ for the string $x = "ababc\$"$	71
Figure 5-1: Labeling sentences with the 'Visualize and Label Parse Trees' operator. To use space optimally, the parse tree is shown in a horizontal layout.	78
Figure 5-2: Assigning a negative label by leaving the label control field unchecked.....	79
Figure 5-3: Training sets in Experiment I with each set considering one particular feature type	80
Figure 5-4: The RapidMiner process in which the optimized SVM models are applied to unseen test examples.....	82
Figure 5-5: Training sets for Experiment II	85
Figure 5-6: Training sets for Experiment III.....	89
Figure 5-7: Benchmark results of the 'Fast String Kernel' operator with differently sized example sets with each set consisting of strings with 500 and 1000 characters.....	92
Figure 5-8: Runtime fractions during the kernel computations.....	92
Figure 5-9: Runtime performance of the matching statistics process per similarity	93
Figure 5-10: Runtime performance of the suffix construction process per suffix tree.....	93
Figure A - 1: An exemplary RapidMiner process using the 'WebLicht Feature Annotator'	95
Figure A - 2: Interactive list of parameters in the 'WebLicht Feature Annotator' (that reflects the concept of the flexible tool chain).....	95
Figure A - 3: An exemplary RapidMiner process using the operator 'WebLicht TCF to ExampleSet'	104
Figure A - 4: Parameters of the 'WebLicht TCF to ExampleSet' operator.	104
Figure A - 5: An exemplary RapidMiner process using the 'Visualize and Label Parse Trees' operator ...	105
Figure A - 6: The visualization panel with an exemplary parse tree	105
Figure A - 7: Parameters of the 'Visualize and Label Parse Trees' operator.....	106
Figure A - 8: An exemplary RapidMiner process with two 'Fast String Kernel' operators.....	108
Figure A - 9: Parameters of the 'Fast String Kernel' operator.....	109

List of Tables

Table 2-1: Disciplines of linguistic tools distributed along different levels of analysis	11
Table 2-2: A selected list of available toolkits for different NLP tasks and languages.....	18
Table 2-3: A list of relevant services of WebLicht that perform NLP tasks for English and German text corpora.	20
Table 2-4: Available conversion tools to process texts to the text corpus format (TCF).	22
Table 2-5: A set of examples with sequences of PoS-tags; the tags are taken from the "Stuttgart-Tübingen Tagset" (STTS); the translations are given in the second lines.....	37
Table 3-1: The list notation of an exemplary tree T	42
Table 3-2: The adjacency matrix of an exemplary graph T ; the entries $A_{uv} = 0$ indicate that the vertex $u = v$ has no self-loop.	42
Table 4-1: Representing the part-of-speech tags as bag of terms.....	63
Table 4-2: Matching statistics of the string "bcbab" with respect to the suffix tree $S("ababc")$	70
Table 4-3: Caching strategy where n suffix trees are kept in memory	75
Table 4-4: Caching strategy with a window of size $k \cdot l$	76
Table 5-1: Two examples that differently fit to a given topic.	77
Table 5-2: Extracted features of an exemplary sentence.....	78
Table 5-3: List of files used or created during Experiment I.....	79
Table 5-4: The optimized C values and with the according accuracies (and standard deviations) obtained from the training runs in Experiment I.....	81
Table 5-5: RapidMiner file used for the training phase in Experiment I.....	81
Table 5-6: RapidMiner file used for the testing phase in Experiment I.....	82
Table 5-7: Classification results (%) of Experiment I "Tranches" obtained from different combinations of linguistic features and weight functions.....	83
Table 5-8: Two examples of general and contemporary literature	84
Table 5-9: List of files used or created during Experiment II.....	84
Table 5-10: RapidMiner file used for the training phase in Experiment II.....	85
Table 5-11: The optimized C values with the according accuracies (and standard deviations) obtained from the training runs in Experiment II.....	86
Table 5-12: Classification results (%) of Experiment II "Literature types" obtained from different combinations of linguistic features and weight functions.....	87
Table 5-13: Two exemplary sentences taken from online articles from Bild.de and Spiegel.de	88
Table 5-14: List of files used or created during Experiment III	88
Table 5-15: RapidMiner file used for the training phase in Experiment III.....	89
Table 5-16: The optimized C values with the according accuracies (and standard deviations) obtained from the training runs in Experiment III.....	89
Table 5-17: Classification results (%) of Experiment III "Bild vs. Spiegel" obtained from different combinations of linguistic features and weight functions.....	90
Table 5-18: The list of files used or created for preparing sentences of fix length.....	91
Table 5-19: The list of prepared text files used for the RapidMiner benchmark process	91

Eidesstattliche Versicherung

Fitzner, Marcel

85453

Name, Vorname

Matrikel-Nummer

Ich versichere hiermit an Eides statt, dass ich die vorliegende Diplomarbeit mit dem Titel

**"Integration of WebLicht Services for Fast Structural Kernel Generations
and Feature Visualization in RapidMiner"**

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt so wie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund.

Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft. Die Technische Universität Dortmund wird ggfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift