Text Preparation through Extended Tokenization

Marcus Hassler, Günther Fliedl
marcus.hassler@uni-klu.ac.at, fliedl@ifit.uni-klu.ac.at
Alps-Adria University Klagenfurt

Abstract

Tokenization is commonly understood as the first step of any kind of natural language text preparation. The major goal of this early (pre-linguistic) task is to convert a stream of characters into a stream of processing units called tokens. Beyond the text mining community this job is taken for granted. Commonly it is seen as an already solved problem comprising the identification of word borders and punctuation marks separated by spaces and line breaks. But in our sense it should manage language related word dependencies, incorporate domain specific knowledge, and handle morphosyntactically relevant linguistic specificities. Therefore, we propose rule-based Extended Tokenization including all sorts of linguistic knowledge (e.g., grammar rules, dictionaries). The core features of our implementation are identification and disambiguation of all kinds of linguistic markers, detection and expansion of abbreviations, treatment of special formats, and typing of tokens including single- and multi-tokens. To improve the quality of text mining we suggest linguistically-based tokenization as a necessary step preceeding further text processing tasks. In this paper, we focus on the task of improving the quality of standard tagging.

Keywords: text preparation, natural language processing, tokenization, tagging improvement, tokenization prototype

1 Introduction

Nearly all researchers concerned with text mining presuppose tokenizing as first step during text preparation [1, 2, 3, 4, 5]. Good surveys about tokenization techniques are provided by Frakes, Baeza-Yates, and Ribeiro-Neto in [6, 7], and Manning and Schütze in [8, pp.124–136]. But – as we know – only very few reflect tokenization as a task of multi-language text processing with far-reaching impact [9]. This involves language-related knowledge about linguistically motivated and domain specific patterns on many levels of linguistic analysis (i.e. sentence border disambiguation, composita identification, abbreviation handling) [10, 11].

Extended Tokenization in our sense does not only separate strings into basic processing units, but also interprets and groups isolated tokens to create higher level tokens. This is done by exploiting so-called token types assigned through an elaborated machinery of heuristic and linguistically motivated rules, and – if available – minimized dictionary knowledge. The early identification of multi-tokens (see Sec. 2) partially covers the well known "named entity recognition" task based on an empirically motivated categorization of proper names as defined by MUC-6[1] and MUC-7[2], also standardized through the TEI[3] Guidelines[4].

Our implementation (see Sec. 4) provides a language independent core tokenizer which is easily adaptable for language specific needs through incremental rule set expansion. Operating on plain text data only, it supports any kind of regular expression pattern matching and several methods for context dependent constraint checking. Thus, the complexity of subsequent processing tasks (e.g., tagging, chunking) is reduced dramatically by early made decisions.

Figure 1 shows where Extended Tokenization is located within the text preparation and processing framework. First, raw texts are preprocessed and segmented into textual units. This step comprises cleansing and filtering (e.g., whitespace collapsing, stripping extraneous control characters) [12] and removal of all kinds of structural or layout relevant markup (e.g., XML tags). Then, Extended Tokenization segments the plain text into appropriate processing units. Subsequent tasks like tagging are applied on the tok-

[1] Message Understanding Conference, http://www.cs.nyu.edu/cs/faculty/grishman/muc6.html (03.03.2006)

[2] Message Understanding Conference, http://www.itl.nist.gov/iaui/894.02/related_projects/muc/proceedings/muc_7_toc.html (03.03.2006)

[3] Text Encoding Initiative, http://www.tei-c.org (03.03.2006)

[4] TEI Guidelines, http://www.tei-c.org/Guidelines2/index.html (03.03.2006)

Figure 1: The task of text preparation and processing

enized output and thus should be supported as far as possible (e.g., format normalization, consistent terminology).

In Sec. 2 we give definitions of token concepts. Section 3 discusses the procedure of token typing and describes our rule-based approach. The architecture and functionality of our online implementation JavaTok is covered in Sec. 4. In the conclusion we outline some features of JavaTok with respect to our theoretical considerations.

2 Definitions of Token Concepts

The most simple form of a token is the *single-token*. It is defined as a character string not containing any non-printable or delimiting characters (blank, tabulator, line-feed, new line, etc.), corresponding to the traditional concept of a token. Examples of single-tokens are words, numbers, internet addresses, most abbreviations, etc. See Guo [4, 13] and Mikheev [14] for more examples.

Written texts also contain more complex language constructs that do not fit into the singe-token concept. Such tokens may be specially formatted using blanks (the standard delimiter for token boundaries) or belong to semantically motivated groups of tokens. In this case the blank is part of a token chain fixed together through interpretation.We define such tokens containing token delimiter characters for formatting (e.g., blanks, tabs, new lines, line feeds) as *multi-tokens*. Well known representants are composite nouns, special formats ('`+43 463 2700-3531`'), named entities (names, locations, institutions), and idioms (formulas). Traditionally they have been identified as a sequence of atomic tokens glued together during a later processing phase - mainly using dictionary lookup. In our approach these tokens are multi-tokens mainly through heuristic interpretation or, in other words, they are tokens through rule-based typing.

The early treatment of multi-tokens as (semantic) concepts during text preparation benefits the overall quality of data- and text-mining tasks. The representation of a text using multi-tokens leads to better intermediate results, hence structurally and (semantically) grouped tokens are treated as atomic units. If subsequent tasks do not support multi-tokens, a simple reinterpretation into standard tokens is possible.

3 The Procedure of Token Typing

As far as we know rule-based typing of tokens at this early stage of NLP has not been introduced in the literature so far. In this paper typing of tokens is defined as a pre-linguistic classification process that assigns type identifiers to both, single-tokens and multi-tokens.

The typing process (see Alg. 1) distinguishes two levels: At the the *first level* (1-3) token and sentence borders are identified. This step results in a sequence of single-tokens, involving basic typing of tokens. At the *second level* (4-6) contextually motivated reinterpretation (retyping) of tokens is done to fit user-specified requirements. Beyond that rules for merging (resp. splitting) of single-tokens (resp. multi-tokens) are applied recursively. The aim is to improve the accuracy of preliminary first level tokenization.

Algorithm 1 Tokenization and typing of tokens

1: identify single-tokens
2: type single-tokens
3: split sentence end markers
4: reinterpret single-token types
5: merge and split tokens recursively
6: reinterpret all token types

The tokenization algorithm starts with basic text segmentation, separating strings into single-tokens (step 1 in Alg. 1) using standard delimiters (blanks, tabs, new lines, line feeds). Each identified single-token is typed (step 2 in Alg. 1) using a predefined set of basic token types. Examples of basic types and subtypes are

- **alphabetics** T_a**:** no letters capitalized (T_{a1}), first letter capitalized (T_{a2}), all letters capitalized (T_{a3}), mixed cases (T_{a4}) etc.
- **numerics** T_n**:** plain numbers (T_{n1}), numbers containing periods or colons (T_{n2}) etc.
- **punctuation marks** T_p**:** sentence ending markers (T_{p1}), pairwise marks lick brackets and quotes (T_{p2}), single sentence-internal marks like commas (T_{p3}) etc.
- **mixtures** T_m**:** ending with sentence end marker (T_{m1}), ending with hyphen (T_{m2}), starting with hyphen (T_{m3}), containing slashes/hyphens (T_{m4}), containing numbers (T_{m5}) etc.

These basic types are assigned to tokens straightforward, utilizing a classification of characters into distinct categories (see Tab. 1).

During the next step punctuation marks are identified and separated (step 3 in Alg. 1). Only tokens typed as mixtures (T_{m1}) are investigated. If a token

Table 1: Example of character definitions

Category	Characters
alpha	abcdefghijklmnopqrstuvwxyzüäöß
alpha captial	ABCDEFGHIJKLMNOPQRSTUVWXYZÜÄÖ
numeric	0123456789
sentence end	.?!
punctuation	,:;"'()[]<>
hyphen	-
delimiters	\U0003 \U0009 \U000A \U000B \U000C \U000D \U0020

string does not match an entry in one of the repositories (e.g., abbreviations, acronyms, regular expressions rules for single-token or multi-token typing), the last character is split and builds a new token together with its corresponding token type (see 1 in Fig. 2). To assure the correctness of this splitting operation basic context-specific rules are applied. A token ended by a period and followed by a lower case token is not split, because the period does not mark the end of a sentence (see 2 in Fig. 2).

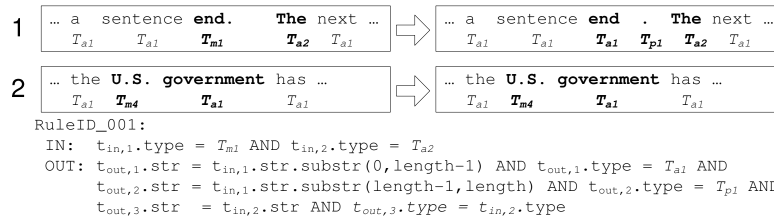

Figure 2: Tokenization examples

A set of user-defined token types is used to reinterpret and group (basic) token types and strings (step 4-6 in Alg. 1). The user is enabled to define custom types to support domain-specific needs. Such token types are simply expressed through strings, which are assigned to recognized tokens. The definition of token types can be related to different sources of knowledge about the motivation for token interpretation. This includes domain knowledge (i.e., structure of an organization, knowledge about data warehouses), gazetteer knowledge (i.e., country names, river names), expert knowledge (i.e., medicine, astronomy), and pure linguistic knowledge (i.e., morphological and syntactical rules, subject of a sentence).

Examples of user-defined types are stopwords (U_1), abbreviations (U_2), dates and times (U_3), phone numbers (U_4), email addresses (U_5), a sequence of capitalized single-tokens (U_6, in many cases extended keywords) etc. These types are identified by applying two strategies: First, tokens are compared with an repository of reliable `(string; token type)` entries created by a human expert or any kind of (semi-) automatic machinery. If no match is found, an ordered list of rules is applied to process the sequence of tokens. The rules include regular expression matching of token strings (see 3 in Fig. 2), matching of token types (see 4 in Fig. 2), and combinations (see 5 and 6 in Fig. 2).

The examples in Fig. 2 and Fig. 3 also outline the rule syntax. Each rule consists of a condition part (input sequence of typed tokens) and a consequence part (output sequence of typed tokens). The numbered indices of tokens indicate relative token positions. Our rule-based approach is based on simple and pure linguistic functional interpretation of basic-token types and token strings in a given context. Example types of rules may cover morphological, syntactical, and general patterns like

- suffix identification of well-known endings (e.g., "-ly", "-ness").
- identification and reconcatenation of hyphenated words at line breaks
- sentence border disambiguation
- multi-token identification
- special character treatment (e.g., apostrophes, slashes, ampersand etc.)

4 JavaTok

This section describes the architecture of JavaTok 1.0, a free-configurable tokenizer developed in JAVA. To cope with language dependent occurrence of special characters (country specific characters like Slavic diacritics, French accents, umlauts and sharp s in German, etc.), JavaTok enables a Unicode[5]-conform initialization and input/output processing. For the purpose of convenient higher-level tokenization the following features are necessary:

- free configuration and adaptation (character definitions, tokenization strategies)
- string replacements (abbreviation resolution, zero elimination, string and thesaurus-like substitution of multiple length)
- user-defined token type definition
- rule-based token typing (credit card numbers, phone numbers, dates, internet addresses, special IDs, . . .)

[5] http://www.unicode.org (30.03.2006)

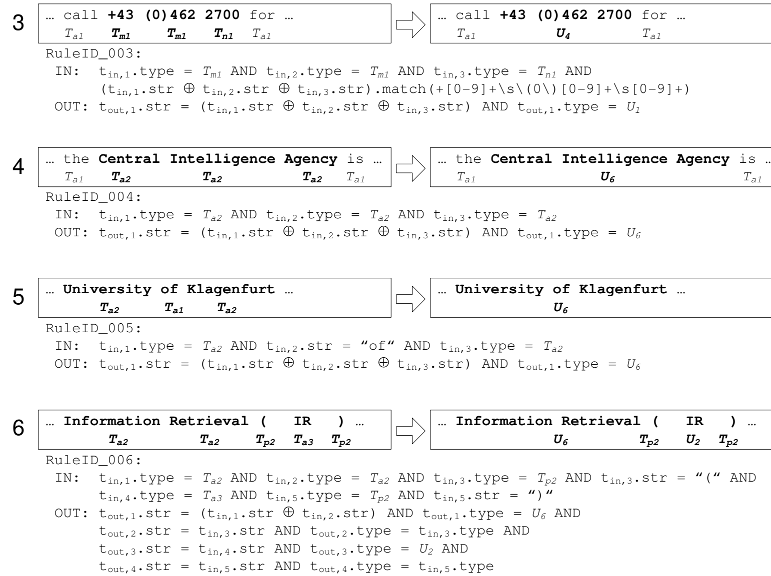

Figure 3: Tokenization rules

- pre-tagging functionality (based on token types)
- compound noun and proper name identification
- multi language support
- process statistics and runtime performance measurement
- multiple output formats

One of the main aims was to support web-based processing, easy integration in existing software systems and good overall performance. An online version of JavaTok is available at our NLP web portal[6].

Extended Tokenization in our sense includes basic linguistic analysis performed through rules described in the previous section. In case of ambiguity JavaTok does not make assumptions or guesses. Uncertain token or sentence borders are not further interpreted, marked, or split. The tokenizer behaves the same way as standard tokenizers do, considering only separators defined for splitting. This is because misinterpretation of tokens at an early stage leads to poor overall performance of NLP systems. JavaTok also does not carry out any kind of character conversion automatically, hence other tools using the tokenizer output may loose important information (such as the case of letters during tagging).

[6]http://nlp.ifit.uni-klu.ac.at (30.03.2006)

Output	S	M	R
The Red Cross is aka. RK .			
The Red Cross is **also known as** RK .			x
The (Red Cross)/INST is aka. RK .		x	
The (Red Cross)/INST is (**also known as**)/ABBR RK .		x	x
The/T_{a2} Red/T_{a2} Cross/T_{a2} is/T_{a1} aka./ABBR RK/T_{a3} ./T_{p1}	x		
The/T_{a2} Red/T_{a2} Cross/T_{a2} is/T_{a1} **also**/T_{a1} **known**/T_{a1} **as**/T_{a1} RK/T_{a3} ./T_{p1}	x		x
The/T_{a2} (Red/T_{a2} Cross/T_{a2})/INST is/T_{a1} aka./ABBR RK/T_{a3} ./T_{p1}	x	x	
The/T_{a2} (Red/T_{a2} Cross/T_{a2})/INST is/T_{a1} (**also**/T_{a1} **known**/T_{a1} **as**/T_{a1})/ABBR RK/T_{a3} ./T_{p1}	x	x	x

Figure 4: Sample outputs: `The Red Cross is aka. RK.`

4.1 Sample Tokenization Outputs

In the example given in Fig. 4 the basic token types used are '/T_{a1}', '/T_{a2}', '/T_{a3}', '/T_{m1}', and '/T_{p1}' (see Sec. 3). The user-defined token types are '/ABBR' (abbreviation) and '/INST' (institution). The mode describes whether single-token typing is enabled (S), whether multi-token typing is enabled (M) and whether known abbreviations are to be replaced (R). The only known abbreviation in the example is 'aka.', standing for 'also known as'. Also, 'Red Cross' is a known institution. 'RK' (Red Cross) is an unknown abbreviation. Hence words do not contain uppercase letters in between, it is marked as irregular by a rule.

4.2 Tagging Optimization Using JavaTok

Figure 5 shows the effect of Extended Tokenization on state-of-the-art tagging outputs. For evaluation purposes we used three freely available taggers: *QTag*[7] developed at the University of Birmingham, *POStaggerME*[8] available as part of the OpenNLP package provided by SourceForge, and *Stanford POS Tagger*[9] implemented by the Stanford NLP Group.

In cell (1) you see the initial input sentence comprising some of the previously discussed delimitation problems, which motivate our token typing and multi-token concepts. Cell (2) contains *QTag* outputs, cell (3) *POStaggerME* outputs, and cell (4) *Stanford POS Tagger* outputs. Lines 2a), 3a) and 4a) refer to standard tagging output, whereas lines 2b), 3b) and 4b) show the improved tagger output including preceded Extended Tokenization done by JavaTok. Underlines reflect differences between standard and JavaTok processed tagging input units. Bold written tags are changed through

[7] http://www.english.bham.ac.uk/staff/omason/software/qtag.html (30.03.2006)
[8] http://opennlp.sourceforge.net (39.03.2006)
[9] http://nlp.stanford.edu/software/tagger.shtml (30.03.2006)

```
1    Straight $n x n$ mapping doesn't fit into ... [5, 7].

2 a) Straight $n  x  n$ mapping doesn't  fit into ... [  5  ,  7  ]  .
     JJ       "   NN NN VBG     DOZ      JJ  IN   CD  NN CD ,  CD NN .
  b) Straight $n x n$   mapping does not fit into ... [5, 7]         .
     JJ       "         NN      DOZ  XNOT VB IN   CD  NN             .

3 a) Straight $n  x  n$ mapping doesn't  fit into ... [  5  ,  7  ]  .
     JJ       NN JJ NN  NN      RB       VB  IN   :   IN CD ,  CD NN .
  b) Straight $n x n$   mapping does not fit into ... [5, 7]         .
     JJ       NN        NN      VBZ  RB  VB  IN   :   CD             .

4 a) Straight $n  x  n$ mapping doesn't  fit into ... [   5  ,  7  ]  .
     JJ       NN LS FW  VBG     JJ       NN  IN   :   SYM CD ,  CD NN .
  b) Straight $n x n$   mapping does not fit into ... [5, 7]         .
     JJ       NN        NN      VBZ  RB  VB  IN   :   CD             .
```

Figure 5: Tagging improvements through Extended Tokenization

JavaTok-specific groupings and/or splittings, thus resulting in optimized tagging inputs. Tagging improvement is achieved in two respects:

- Direct changes of tags through empirically more adequate input units
- Indirect changes of tags through changes of linguistic contexts

5 Conclusion

Extended Tokenization can be seen as one of the core steps of any kind of text preparation. It is crucial for all following text processing tasks. To cope with NLP difficulties we introduced the notion of Extended Tokenization, including token definitions and user-defined token types. With our multi-token concept we are able to classify, split and recombine tokens and token chains to semantic units for further processing. Our rule-based token typing approach carries out reinterpretation and substitution of token strings and token types on two different levels.

Our implementation, JavaTok 1.0, allows proper treatment of both, general and language-related tokenization difficulties. To circumvent early misinterpretation of tokens the tokenizer can leave segmentation decisions open, avoiding hypothetically motivated decisions in ambiguous contexts. JavaTok is optimized for reducing data and time complexity with respect to further processing tasks (e.g., named entity recognition, tagging etc.). A short draft about optimization of tagging output through our tokenization method is

outlined at the end of the paper, showing promising results. However, more empirical work is certainly needed, together with an examination of methods for automatic rule elicitation.

References

[1] Webster, J.J. & Kit, C., Tokenization as the initial phase in NLP. University of Trier, volume 4, pp. 1106–1110, 1992.

[2] Fox, C., Lexical analysis and stoplists. pp. 102–130, 1992.

[3] Grefenstette, G. & Tapanainen, P., What is a word, what is a sentence? problems of tokenization. *The 3rd Conference on Computational Lexicography and Text Research (COMPLEX'94)*, pp. 79–87, 1994.

[4] Guo, J., Critical tokenization and its properties. *Computational Linguistics*, **23(4)**, pp. 569–596, 1997.

[5] Barcala, F.M., Vilares, J., Alonso, M.A., Graa, J. & Vilares, M., Tokenization and proper noun recognition for information retrieval. *3rd International Workshop on Natural Language and Information Systems (NLIS '02)*, pp. 246–250, 2002.

[6] Frakes, W.B. & Baeza-Yates, R., *Information Retrieval: Data Structures and Algorithms*. Prentice Hall, Englewood Cliffs, NJ, USA, 1992.

[7] Baeza-Yates, R. & Ribeiro-Neto, B., *Modern Information Retrieval*. Addison Wesley, ACM Press, New York: Essex, England, 1999.

[8] Manning, C.D. & Schütze, H., *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, Massachusetts: London, England, 5th edition, 2002.

[9] Giguet, E., The stakes of multilinguality: Multilingual text tokenization in natural language diagnosis. *Proceedings of the 4th Pacific Rim International Conference on Artificial Intelligence Workshop Future issues for Multilingual Text Processing*, Cairns, Australia, 1996.

[10] Jackson, P. & Moulinier, I., *Natural Language Processing for Online Applications: Text Retrieval, Extraction and Categorisation*. John Benjamins, msterdam, Netherlands: Wolverhampton, United Kingdom, 2002.

[11] Say, B. & Akman, V., An information-based approach to punctuation. *Proceedings ICML '96: Second International Conference on Mathematical Linguistics*, Tarragona, Spain, pp. 93–94, 1996.

[12] Palmer, D.D., Tokenisation and sentence segmentation. *Handbook of Natural Language Processing*, eds. R. Dale, H. Moisl & H. Somers, Marcel Dekker, Inc., pp. 11–35, 2000.

[13] Guo, J., One tokenization per source. *Proceedings of the Thirty-Sixth Annual Meeting of the Association for Computational Linguistics and Seventeenth International Conference on Computational Linguistics*, pp. 457–463, 1998.

[14] Mikheev, A., Periods, capitalized words, etc. *Comput Linguist*, **28(3)**, pp. 289–318, 2002.