Tokenization: Returning to a Long Solved Problem
A Survey, Contrastive Experiment, Recommendations, and Toolkit

Rebecca Dridan & Stephan Oepen
Institutt for Informatikk, Universitetet i Oslo
{ rdridan | oe } @ifi.uio.no

Abstract

We examine some of the frequently disregarded subtleties of tokenization in Penn Treebank style, and present a new rule-based preprocessing toolkit that not only reproduces the Treebank tokenization with unmatched accuracy, but also maintains exact stand-off pointers to the original text and allows flexible configuration to diverse use cases (e.g. to genre- or domain-specific idiosyncrasies).

1 Introduction—Motivation

The task of *tokenization* is hardly counted among the grand challenges of NLP and is conventionally interpreted as breaking up "natural language text [...] into distinct meaningful units (or tokens)" (Kaplan, 2005). Practically speaking, however, tokenization is often combined with other string-level preprocessing—for example normalization of punctuation (of different conventions for dashes, say), disambiguation of quotation marks (into opening vs. closing quotes), or removal of unwanted mark-up—where the specifics of such pre-processing depend both on properties of the input text as well as on assumptions made in downstream processing.

Applying some string-level normalization *prior* to the identification of token boundaries can improve (or simplify) tokenization, and a sub-task like the disambiguation of quote marks would in fact be hard to perform *after* tokenization, seeing that it depends on adjacency to whitespace. In the following, we thus assume a *generalized* notion of tokenization, comprising all string-level processing up to and including the conversion of a sequence of characters (a string) to a sequence of token objects.[1]

Arguably, even in an overtly 'separating' language like English, there can be token-level ambiguities that ultimately can only be resolved through parsing (see § 3 for candidate examples), and indeed Waldron et al. (2006) entertain the idea of downstream processing on a token *lattice*. In this article, however, we accept the tokenization conventions and sequential nature of the Penn Treebank (PTB; Marcus et al., 1993) as a useful point of reference—primarily for interoperability of different NLP tools.

Still, we argue, there is remaining work to be done on PTB-compliant tokenization (reviewed in§ 2), both methodologically, practically, and technologically. In § 3 we observe that state-of-the-art tools perform poorly on re-creating PTB tokenization, and move on in § 4 to develop a modular, parameterizable, and transparent framework for tokenization. Besides improvements in tokenization accuracy and adaptability to diverse use cases, in § 5 we further argue that each token object should unambiguously link back to an underlying element of the original input, which in the case of tokenization of text we realize through a notion of *characterization*.

2 Common Conventions

Due to the popularity of the PTB, its tokenization has been a de-facto standard for two decades. Approximately, this means splitting off punctuation into separate tokens, disambiguating straight quotes, and separating contractions such as *can't* into *ca* and *n't*. There are, however, many special cases—

[1]Obviously, some of the normalization we include in the tokenization task (in this generalized interpretation) could be left to downstream analysis, where a tagger or parser, for example, could be expected to accept non-disambiguated quote marks (so-called straight or typewriter quotes) and disambiguate as part of syntactic analysis. However, on the (predominant) point of view that punctuation marks form tokens in their own right, the tokenizer would then have to adorn quote marks in some way, as to whether they were split off the left or right periphery of a larger token, to avoid unwanted syntactic ambiguity. Further, increasing use of Unicode makes texts containing 'natively' disambiguated quotes more common, where it would seem unfortunate to discard linguistically pertinent information by normalizing towards the poverty of pure ASCII punctuation.

documented and undocumented. In much tagging and parsing work, PTB data has been used with gold-standard tokens, to a point where many researchers are unaware of the existence of the original 'raw' (untokenized) text. Accordingly, the formal definition of PTB tokenization[2] has received little attention, but reproducing PTB tokenization automatically actually is not a trivial task (see § 3).

As the NLP community has moved to process data other than the PTB, some of the limitations of the PTB tokenization have been recognized, and many recently released data sets are accompanied by a note on tokenization along the lines of: *Tokenization is similar to that used in PTB, except ...* Most exceptions are to do with hyphenation, or special forms of named entities such as chemical names or URLs. None of the documentation with extant data sets is sufficient to fully reproduce the tokenization.[3]

The CoNLL 2008 Shared Task data actually provided two forms of tokenization: that from the PTB (which many pre-processing tools would have been trained on), and another form that splits (most) hyphenated terms. This latter convention recently seems to be gaining ground in data sets like the Google 1T n-gram corpus (LDC #2006T13) and OntoNotes (Hovy et al., 2006). Clearly, as one moves towards a more application- and domain-driven idea of 'correct' tokenization, a more transparent, flexible, and adaptable approach to string-level pre-processing is called for.

3 A Contrastive Experiment

To get an overview of current tokenization methods, we recovered and tokenized the raw text which was the source of the (Wall Street Journal portion of the) PTB, and compared it to the gold tokenization in the syntactic annotation in the treebank.[4] We used three common methods of tokenization: (a) the original PTB tokenizer.sed script; (b) the tokenizer from the Stanford CoreNLP tools[5]; and (c) tokenization from the parser of Charniak & Johnson (2005). Table 1 shows quantitative differences between each of the three methods and the PTB, both in terms of the number of sentences where the tokenization differs, and also in the total Levenshtein distance (Levenshtein, 1966) over tokens (for a total of 49,208 sentences and 1,173,750 gold-standard tokens).

Tokenization Method	**Differing Sentences**	**Levenshtein Distance**
tokenizer.sed	3264	11168
CoreNLP	1781	3717
C&J parser	2597	4516

Table 1: Quantitative view on tokenization differences.

Looking at the differences qualitatively, the most consistent issue across all tokenization methods was ambiguity of sentence-final periods. In the treebank, final periods are always (with about 10 exceptions) a separate token. If the sentence ends in *U.S.* (but not other abbreviations, oddly), an extra period is hallucinated, so the abbreviation also has one. In contrast, C&J add a period to all final abbreviations, CoreNLP groups the final period with a final abbreviation and hence lacks a sentence-final period token, and the sed script strips the period off *U.S.* The 'correct' choice in this case is not obvious and will depend on how the tokens are to be used.

The majority of the discrepancies in the sed script tokenization come from an under-restricted punctuation rule that incorrectly splits on commas within numbers or ampersands within names. Other than that, the problematic cases are mostly shared across tokenization methods, and include issues with currencies, Irish names, hyphenization, and quote disambiguation. In addition, C&J make some additional modifications to the text, lemmatising expressions such as *won't* as *will* and *n't*.

4 REPP: A Generalized Framework

For tokenization to be studied as a first-class problem, and to enable customization and flexibility to diverse use cases, we suggest a non-procedural, rule-based framework dubbed REPP (Regular

[2]See `http://www.cis.upenn.edu/~treebank/tokenization.html` for available 'documentation' and a sed script for PTB-style tokenization.

[3]Øvrelid et al. (2010) observe that tokenizing with the GENIA tagger yields mismatches in one of five sentences of the GENIA Treebank, although the GENIA guidelines refer to scripts that may be available on request (Tateisi & Tsujii, 2006).

[4]The original WSJ text was last included with the 1995 release of the PTB (LDC #95T07) and required alignment with the treebank, with some manual correction so that the same text is represented in both raw and parsed formats.

[5]See `http://nlp.stanford.edu/software/corenlp.shtml`, run in '`strictTreebank3`' mode.

```
>wiki
#1
!([^␣])([])}?!,;:”’])␣([^␣]|$)     \1␣\2␣\3
!(^|[^␣])␣([[({“‘])([^␣])          \1␣\2␣\3
#
>1
:[[:space:]]+
```

Figure 1: Simplified examples of tokenization rules.

Expression-Based Pre-Processing)—essentially a cascade of ordered finite-state string rewriting rules, though transcending the formal complexity of regular languages by inclusion of (a) full perl-compatible regular expressions and (b) fixpoint iteration over groups of rules. In this approach, a first phase of string-level substitutions inserts whitespace around, for example, punctuation marks; upon completion of string rewriting, token boundaries are stipulated between all whitespace-separated substrings (and only these).

For a good balance of human and machine readability, REPP tokenization rules are specified in a simple, line-oriented textual form. Figure 1 shows a (simplified) excerpt from our PTB-style tokenizer, where the first character on each line is one of four REPP operators, as follows: (a) ‘#’ for group formation; (b) ‘>’ for group invocation, (c) ‘!’ for substitution (allowing capture groups), and (d) ‘:’ for token boundary detection.[6] In Figure 1, the two rules stripping off prefix and suffix punctuation marks adjacent to whitespace (i.e. matching the tab-separated left-hand side of the rule, to replace the match with its right-hand side) form a numbered group (‘#1’), which will be iterated when called (‘>1’) until none of the rules in the group fires (a fixpoint). In this example, conditioning on whitespace adjacency avoids the issues observed with the PTB sed script (e.g. token boundaries within comma-separated numbers) and also protects against infinite loops in the group.[7]

REPP rule sets can be organized as modules, typically each in a file of its own, and invoked selectively by name (e.g. ‘>wiki’ in Figure 1); to date, there exist modules for quote disambiguation, (relevant subsets of) various mark-up languages (HTML, LaTeX, wiki, and XML), and a handful of robustness rules (e.g. seeking to identify and repair ‘sandwiched’ inter-token punctuation). Individual tokenizers are configured at run-time, by selectively activating a set of modules (through command-line options). An open-source reference implementation of the REPP framework (in C^{++}) is available, together with a library of modules for English.

5 Characterization for Traceability

Tokenization, and specifically our notion of generalized tokenization which allows text normalization, involves changes to the original text being analyzed, rather than just additional annotation. As such, full *traceability* from the token objects to the original text is required, which we formalize as ‘characterization’, in terms of character position links back to the source.[8] This has the practical benefit of allowing downstream analysis as direct (stand-off) annotation on the source text, as seen for example in the ACL Anthology Searchbench (Schäfer et al., 2011).

With our general regular expression replacement rules in REPP, making precise what it means for a token to link back to its ‘underlying’ substring requires some care in the design and implementation. Definite characterization links between the string before ($\mathcal{I}$) and after ($\mathcal{O}$) the application of a single rule can only be established in certain positions, viz. (a) *spans not matched by the rule*: unchanged text in $\mathcal{O}$ outside the span matched by the left-hand side regex of the rule can always be linked back to $\mathcal{I}$; and (b) *spans caught by a regex capture group*: capture groups represent the same text in the left- and right-hand sides of a substitution, and so can be linked back to $\mathcal{O}$.[9] Outside these text spans, we can only make definite statements about characterization links at boundary points, which include the start and end of the full string, the start and end of the string

[6]Strictly speaking, there are another two operators, for line-oriented comments and automated versioning of rule files.

[7]For this example, the same effects seemingly could be obtained without iteration (using greatly more complex rules); our actual, non-simplified rules, however, further deal with punctuation marks that can function as prefixes or suffixes, as well as with corner cases like *factor(s)* or *Ca[2+]*. Also in mark-up removal and normalization, we have found it necessary to ‘parse’ nested structures by means of iterative groups.

[8]If the tokenization process was only concerned with the identification of token boundaries, characterization would be near-trivial.

[9]If capture group references are used out-of-order, however, the per-group linkage is no longer well-defined, and we resort to the maximum-span ‘union’ of boundary points (see below).

matched by the rule, and the start and end of any capture groups in the rule.

Each character in the string being processed has a start and end position, marking the point before and after the character in the original string. Before processing, the end position would always be one greater than the start position. However, if a rule mapped a string-initial, PTB-style opening double quote (``) to one-character Unicode “, the new first character of the string would have start position 0, but end position 2. In contrast, if there were a rule

```
!wo(n't)     will␣\1                (1)
```

applied to the string *I won't go!*, all characters in the second token of the resulting string (*I will n't go!*) will have start position 2 and end position 4. This demonstrates one of the formal consequences of our design: we have no reason to assign the characters *ill* any start position other than 2.[10] Since explicit character links between each $\mathcal{I}$ and $\mathcal{O}$ will only be established at match or capture group boundaries, any text from the left-hand side of a rule that should appear in $\mathcal{O}$ must be explicitly linked through a capture group reference (rather than merely written out in the right-hand side of the rule). In other words, rule (1) above should be preferred to the following variant (which would result in character start and end offsets of 0 and 5 for *both* output tokens):

```
!won't    will␣n't                 (2)
```

During rule application, we keep track of character start and end positions as offsets between a string before and after each rule application (i.e. all pairs $\langle\mathcal{I},\mathcal{O}\rangle$), and these offsets are eventually traced back to the original string at the time of final tokenization.

6 Quantitative and Qualitative Evaluation

In our own work on preparing various (non-PTB) genres for parsing, we devised a set of REPP rules with the goal of following the PTB conventions. When repeating the experiment of § 3 above using REPP tokenization, we obtained an initial difference in 1505 sentences, with a Levenshtein distance of 3543 (broadly comparable to CoreNLP, if marginally more accurate).

[10]This subtlety will actually be invisible in the final token objects if *will* remains a single token, but if subsequent rules were to split this token further, all its output tokens would have a start position of 2 and an end position of 4. While this example may seem unlikely, we have come across similar scenarios in fine-tuning actual REPP rules.

Examining these discrepancies, we revealed some deficiencies in our rules, as well as some peculiarities of the ‘raw’ Wall Street Journal text from the PTB distribution. A little more than 200 mismatches were owed to improper treatment of currency symbols (*AU$*) and decade abbreviations (*’60s*), which led to the refinement of two existing rules. Notable PTB idiosyncrasies (in the sense of deviations from common typography) include ellipses with spaces separating the periods and a fairly large number of possessives (*’s*) being separated from their preceding token. Other aspects of gold-standard PTB tokenization we consider unwarranted ‘damage’ to the input text, such as hallucinating an extra period after `U.S.` and splitting `cannot` (which adds spurious ambiguity). For use cases where the goal were *strict* compliance, for instance in pre-processing inputs for a PTB-derived parser, we added an optional REPP module (of currently half a dozen rules) to cater to these corner cases—in a spirit similar to the CoreNLP mode we used in § 3. With these extra rules, remaining tokenization discrepancies are contained in 603 sentences (just over 1 %), which gives a Levenshtein distance of 1389.

7 Discussion—Conclusion

Compared to the best-performing off-the-shelf system in our earlier experiment (where it is reasonable to assume that PTB data has played at least some role in development), our results eliminate two thirds of the remaining tokenization errors—a more substantial reduction than recent improvements in parsing accuracy against the PTB, for example.

Of the remaining differences, over 350 are concerned with mid-sentence period ambiguity, where at least half of those are instances where a period was separated from an abbreviation in the treebank—a pattern we do not wish to emulate. Some differences in quote disambiguation also remain, often triggered by whitespace on both sides of quote marks in the raw text. The final 200 or so differences stem from manual corrections made during treebanking, and we consider that these cases could not be replicated automatically in any generalizable fashion.

References

Charniak, E., & Johnson, M. (2005). Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics* (pp. 173–180). Ann Arbor, USA.

Hovy, E., Marcus, M., Palmer, M., Ramshaw, L., & Weischedel, R. (2006). Ontonotes. The 90% solution. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics* (pp. 57–60). New York City, USA.

Kaplan, R. M. (2005). A method for tokenizing text. Festschrift for Kimmo Koskenniemi on his 60th birthday. In A. Arppe, L. Carlson, K. Lindén, J. Piitulainen, M. Suominen, M. Vainio, H. Westerlund, & A. Yli-Jyrä (Eds.), *Inquiries into words, constraints and contexts* (pp. 55 – 64). Stanford, CA: CSLI Publications.

Levenshtein, V. (1966). Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physice – Doklady*, *10*, 707–710.

Marcus, M. P., Santorini, B., & Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English. The Penn Treebank. *Computational Linguistics*, *19*, 313 – 330.

Øvrelid, L., Velldal, E., & Oepen, S. (2010). Syntactic scope resolution in uncertainty analysis. In *Proceedings of the 23rd international conference on computational linguistics* (pp. 1379 – 1387). Beijing, China.

Schäfer, U., Kiefer, B., Spurk, C., Steffen, J., & Wang, R. (2011). The ACL Anthology Searchbench. In *Proceedings of the ACL-HLT 2011 system demonstrations* (pp. 7–13). Portland, Oregon, USA.

Tateisi, Y., & Tsujii, J. (2006). *GENIA annotation guidelines for tokenization and POS tagging* (Technical Report # TR-NLP-UT-2006-4). Tokyo, Japan: Tsujii Lab, University of Tokyo.

Waldron, B., Copestake, A., Schäfer, U., & Kiefer, B. (2006). Preprocessing and tokenisation standards in DELPH-IN tools. In *Proceedings of the 5th International Conference on Language Resources and Evaluation* (pp. 2263 – 2268). Genoa, Italy.