THE LISTER HILL NATIONAL CENTER FOR BIOMEDICAL COMMUNICATIONS

A research division of the National Library of Medicine

TECHNICAL REPORT
LHNCBC-TR-2006-003

A Comparison of 13 Tokenizers on MEDLINE

December 2006

Ying He, Ph.D.
Mehmet Kayaalp, M.D., Ph.D.

U.S. National Library of Medicine, LHNCBC
8600 Rockville Pike, Building 38A
Bethesda, MD 20894

Abstract

This report describes a study on tokenization of MEDLINE® abstracts by 13 different software packages that are freely available. In literature, there is little or no comparative evaluation studies on general purpose tokenizers, nor is there any such study on tokenizers that are specific to biomedical text. Biomedical text processing in general and tokenization in particular are quite challenging as biomedical text contains a wide variety of domain-specific terms.

This study explores various scenarios taken from actual MEDLINE abstracts, and provides critical evaluation on the observed performances of the tested tokenizers. The results of this study show that there is a wide variance among outputs of these tokenizers and choosing a right tokenizer requires detailed information that this report is aimed to compile.

The target audience of this report may be those people who are interested in using any particular tokenizer and want to know what types of behavior are expected from general purpose and biomedical tokenizers. The report is prepared with the intention to aid the decision making process of the reader on choosing the right tokenizer and/or devising algorithms that can effectively use the resulting tokens with a minimum loss of information. The reader can find a list of factors that need to be taken into account in such decision. The report also discusses various pros and cons of the tokenizers that are tested.

Table of Contents

1. Introduction

This report describes our study on 13 software packages that tokenize text into words. The text of this study comes from the abstracts of MEDLINE®, a corpus of biomedical literature compiled by the U.S. National Library of Medicine. In linguistics, tokenization is the process of breaking a sequence of characters into words, numbers, punctuations, and other symbols. These words and expression sequences are called tokens, and the tools performing such tokenization are called tokenizers. The following example illustrates the basic function of a tokenizer, where each line of the output is a distinct token.

Input

```
This is a test.
```

Output

```
This
is
a
test
.
```

A number of tokenizers are available as open source software. Preliminary examination shows that, given a MEDLINE abstract as the input, outputs of these tokenizers differ—sometimes significantly. For example, some tokenizers split hyphenated compound words (e.g., `alpha-T`) into two or more tokens (e.g., `alpha`, `-`, and `T`), but other tokenizers keep it as one token. Such differences may considerably alter the course of many natural language processing (NLP) procedures such as indexing and part of speech (POS) tagging. If an inverted index is built based on a tokenization protocol that generates three separate tokens for the word `alpha-T`, a search engine would not be able to find the word in the index directly. More importantly, other concepts denoted by `alpha` or `T` might be erroneously associated with `alpha-T`.

As outputs of word tokenizers are distinct from each other, choosing a right tokenizer is a non-trivial task. Tokenizing biomedical literature poses an additional challenge as it contains many domain-specific words such as names of genes, gene sequences, and chemical substances. So far, we only found one paper on comparisons of word tokenizers [1], which, however, discusses mainly the needs of such comparison rather than actually comparing them.

We installed 13 freely available software packages that can tokenize sentences and studied their performance comparatively. Although we have done this study in order to choose the right tokenizer(s) for an intramural research project [2], we believe that others might benefit from our experience as well. We here present our findings in a technical report format and hope that the documentation of our experience would help others to choose a tokenizer that suits to their needs.

2. Experiment Settings

In this study, we initially downloaded 18 software packages, some of which were designed for different purposes but all can be utilized for tokenization. We then had to put the following tokenizers aside due to technical reasons, which we briefly state below:

- Gate tokenizer
- LingPipe tokenizer
- Masao Utiyama's tokenizer
- YamCha tokenizer
- FreeLing tokenizer

Gate, LingPipe, and Masao Utiyama' tokenizers need extra coding to run. YamCha tokenizer requires training sets to be created by users. We had difficulty to install FreeLing due to its complicated installation procedure.

After eliminating the above mentioned packages, we ended up with a set of 13 tokenizers as shown in Table 1 (see Appendix A for detailed descriptions on the software packages that contain these tokenizers):

Table 1: Tokenizers

No.	Tokenizer
1	NLTK tokenizer [3]
2	OpenNLP tokenizer [4]
3	Mallet tokenizer [5]
4	SPECIALIST NLP tokenizer [6]
5	Gump tokenizer [7]
6	Dan Melamed's tokenizer [8]
7	Qtoken [9]
8	UIUC word splitter [10]
9	LT TTT tokenizer [11]
10	MedPost tokenizer [12]
11	Brill's POS tagger [13]
12	Stanford POS tagger [14]
13	MXPOST tagger [15]

All taggers in the above table perform tokenization. In order to compare their tokenization performances side-by-side, we discarded the tags if they were produced automatically.

Some of the above software packages have two or more tokenizers, and we only chose the one that needs the least additional effort to run. For those tokenizers that require supervision (i.e., a training set), we used the trained language model supplied with corresponding tokenizer package. We exclude those tokenizers that require users to formulate regular expressions. The performance of a tokenizer that requires customized regular expression depends heavily on its user's ability (more specifically, domain and

linguistic knowledge of the user and the user's skills on formulating regular expressions); therefore, inclusion of such tokenizers would severely hamper our efforts for objective comparisons.

3. Comparison of Tokenizers

In our comparison, we observed that all tokenizers are different, except SPECIALIST NLP tokenizer and Qtoken, which produce identical results in our test. For a test document with 78 MEDLINE abstracts, the number of tokens varies from 14488 to 17117. The tokenizer with the smallest number of tokens was Mallet tokenizer, followed by NLTK tokenizer with 14977 tokens. The strategies of these two tokenizers were similar in delimiting tokens by white spaces only. Mallet tokenizer also removed punctuations and numbers from the output. The SPECIALIST NLP tokenizer and the Qtoken yield the largest number of tokens. They used both white spaces and punctuations as token delimiters. The outputs of other systems were differing when tokenization was involving punctuations, compound words, numbers, parentheses and other non-alphanumerical symbols, abbreviations, Unicode, chemical substances, other named entities (website, email address, date, unit, equation, reference etc.), or foreign language terms.

In Appendix B, we present outputs from each tokenizer for a sample input sentence from MEDLINE:

```
Independent of current body composition, IGF-I levels at 5 yr
were significantly associated with rate of weight gain
between 0-2 yr (beta = 0.19; P &lt; 0.0005), and children
who showed postnatal catch-up growth (i.e. those who
showed gains in weight or length between 0-2 yr by >0.67 SD
score) had higher IGF-I levels than other children (P = 0.02).
```

From the sample outputs, we observe that tokenizers (see Table 1 where each tokenizer is associated with a unique number) break sentences into tokens

- at white spaces only (tokenizers 1, 3 and 13),
- at white spaces and punctuations (tokenizers 4 and 7).

Tokenizer 3 also removes all punctuations and numbers from the output.

Outputs of other tokenizers (2, 5, 6, 8, 9, 10, 11, and 12) do not follow such simple rules and their outputs differ when the text in question involves hyphenated compound words such as `catch-up`, decimal numbers such as `0.19`, range numbers such as `0-2`, abbreviations such as `i.e.` or HTML symbols such as `<` (a substitute for ASCII "less than" sign <).

We have the following observations on tokenizers 2, 5, 6, 8, 9, 10, 11, and 12 (The ˆ mark indicates token boundaries):

- Tokenizers 6 and 9 break the hyphenated word `catch-up` into three tokens, and the other tokenizers keep it as one token.
- All tokenizers except tokenizer 11 in this set keep the decimal number `0.19` as one token. Tokenizers 6 and 9 break the range number `0-2` into three tokens while the others kept it as one token.

- For the abbreviation `i.e.`, there are three output variants. Tokenizer 2 breaks it into three tokens (`i␣.e␣.`), tokenizers 5, 8, 9, 10, and 12 keep it as one token, and tokenizer 6 and 11 break it into four tokens.
- Only tokenizers 10 and 12 recognize the HTML symbol `<` as a whole unit.
- All tokenizers treat parentheses as separate tokens. However, tokenizers 8 and 12 use other symbols replacing parentheses.

4. Tokenization Choices

As noticed in the previous example, tokenizers make different choices on what constitutes a token. In this section, we are going to elaborate these different tokenization schemes.

First, we present a few simple cases (Set I), where usually only one type of punctuation is involved. We want to use these cases to show the differences among tokenizers from punctuation point of view.

Then, we present a second set (Set II), which constitutes more complex cases, most of which can be categorized as named entities such as chemical substances, mathematical formulas, and URLs, which usually involve several types of punctuations. These cases show differences among tokenizers from named entity point of view.

Here the term "word" refers to those words separated by white spaces in the original text; whereas, the term "token" refers to smaller pieces of text constructing a "word".

Case Set I

In this set, we include the following categorized examples (see Appendix C for outputs from each tokenizer):

- Hyphenated compound words (e.g. `bottle-fed`, `Bottle-fed`)
- Words with letters and slashes (e.g. `insertion/deletion`, `mg/day`, `Bethesda/MD`)
- Words with letters and apostrophes (e.g. `years'`, `Freud's`, `haven't`, `o'clock`, `'reportorie cloning'`, `O'Neill`)
- Words with letters and brackets (e.g. `(tissue)`, `(Dipnoi)`, `(TP) [GABAergic]`)
- Words with letters and periods (e.g. `i.e.`, `e.g.`)
- Words with letters and numbers (e.g. `CO2`, `12th`)
- Words with numbers and one type of punctuations (e.g. `3,000,000`, `1/2`, `76%`, `1.4`, `1-20`)

There are two main tokenization schemes that are operational on Case Set I: One scheme keeps the word as a whole unit; whereas, the other breaks it into several tokens at punctuations or numbers. However, there are a few exceptions. For example, for the word `haven't`, there is an extra scheme that breaks it into `have` and `n't`, where `n't` is treated as a special form of `not`.

When breaking word into tokens, the places where to break are not always consistent even in the same category. For example, hyphenated compound words `bottle-fed` and `Bottle-fed` are treated by tokenizer 9 differently. The former is kept as a single token, while the latter is broken into three tokens. Another inconsistent example is tokenizer 2 on dealing with parentheses, where `(tissue)` is split into two tokens, `(tissue` and `)`, and `(Dipnoi)` is split into two tokens, `(` and `Dipnoi)`, but `(TP)` is split into three tokens `(`, `TP` and `)`.

By examining these simple cases, we acquired a basic understanding about the choices, assumptions, and behavior of each tokenizer. Readers who are interested in choosing a tokenizer have a simple list of what to look for in a tokenizer. Additionally, if readers consider using the set of tokenizers that are studied here, they can easily eliminate those tokenizers that do not fit their requirements. On the other hand, the analysis that has been outlined above is merely the tip of the iceberg. When inputs are more complex than the ones presented above, to make a decision on the right/acceptable tokenizer is not a trivial one. The following case set and the rest of the report address to those complex issues and intend to shed some light on the types of tokenization schemes that are involved.

Case Set II

In this set, we include the following categorized examples:

- a DNA sequence: `5'-ATGCAAAT-3'`
- a chemical substance: `3,4-epoxy-3-methyl-1-butyl-diphosphate`
- two arithmetical expressions: `76.8+/-14.2` and `8.4-13.8%`
- a hypertext markup symbol: `<`
- a URL: `http://www.tobaccoarchives.com`

Outputs of tokenizers for Case Set II are provided in Appendix C: Case Set II. As seen in the outputs, there are a number of different sets of token—in one case, nine different outputs were produced by 13 tokenizers.

As these words are named entities, users usually have preferences on how to tokenize them. Those preferences may be driven by word senses, document context, and/or other conventions. We hope the Case Set II would be a starting point for developing a guideline for the reader. It is a starting point, since the limited scope of our effort prevents us from developing a more complete set of named entities such as gene symbols (e.g. `PrP33-35C`, `F23H11.1`), drug names, temporal expressions (date, time and other temporal references), spatial references (e.g., geographical locations and mailing addresses), organization names, email addresses, and various units combined with their measures (e.g., 100mg/dl/day). Different tokenizers output different token sets for these entities. Some outputs can be misleading in later NLP stages. For these cases, it may be helpful if a tokenizer can first recognize them as named entities and use the information to perform tokenization.

5. Factors for Decision Making

We observed a wide variance among outputs of these 13 tokenizers of this study. Awareness of the details of the tokenization schemes is critical to the user in order to choose the right tokenizer and/or devise algorithms that can effectively use the resulting tokens with the minimum loss of information. Below, we listed some of those details and factors that need to be taken into account when the user makes decision on choosing a particular tokenizer.

1. In many NLP tasks, tokenization is usually followed by a POS tagging; therefore, the choice of a tagger directly affects the choice on tokenization. For example, in a Penn Treebank tagger, `years'` would be tagged as `years/NNS` and `'/POS`. It is not appropriate to choose a tokenizer that would treat `years'` as one token. Therefore, if the user has a particular preference on a POS tagger, the user would be better of to use POS tagger as the tokenizer.

2. Since the choice of tokenizer would affect indexing, it is necessary to plan the indexing strategy considering the tokenization schemes at hand. As discussed in the introduction, given the word `alpha-T`, if an inverted index is built using three separate tokens `alpha`, `-`, and `T`, this word would not be directly accessed, but a combination strategy for search needs to be devised.

3. The tokenization scheme must be consistent with the scheme of each of the following NLP steps. For example, in tokenizer 9, capital hyphenated compound words are broken into three or more tokens but lowercase ones are kept as one token. If any of the following NLP processes expects all hyphenated compound words be treated uniformly, the results would be problematic to the least. For other word forms such as abbreviations, the situation may become more complex.

4. Frequently, the ability of reconstructing the text into its original format is necessary. In these cases, tokenizers that perform normalization may not be good candidates. For example, tokenizer 8 converts both parenthesis `(` and square bracket `[` to `-LBR-`, and it is impossible to recover the original text from such normalized token set.

5. The domain, in our case biomedical and clinical sciences, may play a significant role in choosing the right tokenizer. Many tokenizers are trained using texts from the *Wall Street Journal* and they usually perform better than others on newspapers and/or on financial texts.

Other factors such as programming languages, training prerequisites, and lexicon prerequisites may also affect the choice of tokenizer.

6. Discussion

In this section, we discuss pros and cons of the tokenizers we have tested, see Appendix D for a tabulated comparison.

Tokenizer 1 (NLTK tokenizer) is too simplistic for many realistic tokenization tasks. For users familiar with Python, in which the software is written, it might be a platform for building research prototypes.

Tokenizer 2 (OpenNLP tokenizer) is a tokenizer with a grammar model that is trainable. The user can either train the system with a customized training set or apply only a default syntax model provided with the OpenNLP system. It preserves hyphenated compound words and various numerical forms within a single token boundary. However, it treats parentheses inconsistently as we show in Case Set I, which is the main concern of this tokenizer. It may be corrected by using different trained language model. This tokenizer comes with a Java API.

Tokenizer 3 (Mallet tokenizer) is a bare-bone word tokenizer, which is hardly suitable for any real-world NLP task.

Tokenizer 4 (SPECIALIST NLP tokenizer) breaks a given text into small pieces by delimiting at both white spaces and punctuations. It is possible to reassemble these tokens to larger lexical items, using another module that comes with SPECIALIST NLP Tools software. Another feature of this tokenizer is that it also outputs the number of white spaces between successive tokens, which facilitates reconstructing tokens back to the original form of the text. The tool is written in Java and comes with a Java API as well as the source code.

Tokenizer 5 (Gump tokenizer) is designed for Brill's POS Tagger, and its output is similar to that of tokenizer 2. Although it may be sufficient for general purpose tokenization, its source code is written in Gump, which might be undesirable for users who plan to customize the tokenization process.

Tokenizer 6 (Dan Melamed's tokenizer) breaks text into small tokens as those in tokenizer 4, except for some word forms such as hyphenated compound words and arithmetical expressions, which, unlike in tokenizer 4, are broken into multiple tokens as well. It usually keeps numbers with decimal points and commas together. It also conserves genitive word forms in single tokens. It is written in Perl and its source code is available.

Tokenizer 7 (Qtoken) has produced outputs identical to tokenizer 4 in our tests. It is written in Java, but its source code is not available.

Tokenizer 8 (UIUC word splitter) performs similarly to tokenizers 2 and 5, though it breaks numbers from letters (e.g., `1st` → `1` and `st`, and `CO2` → `CO` and `2`). The input of this tokenizer can only be one sentence, which means that it is necessary to preprocess the text with a sentence splitter before using this tokenizer. Another drawback of this

tokenizer is that it converts parentheses and squared brackets into the same normalized tokens (-LBR- or –RBR-).

Tokenizer 9 (LT TTT tokenizer) has some distinguishing features in treating hyphenated compound words and in treating words that contain both letters and numbers.

Tokenizer 10 (MedPost tokenizer) is part of the MedPost tagger package. MedPost is designed as a tagger, but its "–token" option makes it work as a tokenizer. It preserves hyphenated compound words and most of the numeric word forms in a single token structure. However, it breaks words at slash and percentage signs. It is the only tokenizer in our set of tokenizers that is designed to deal with Unicode. It is written in C++ and Perl.

Tokenizer 11 (Brill's POS tagger) is designed as a POS tagger. It is not suitable to be a tokenizer since it removes some tokens, and performs poorly for words in Case Set II.

Tokenizer 12 (Stanford POS tagger) is designed as a POS tagger and can be used as a tokenizer. It does not tokenize complex words, such as those tested in Case Set II. It is written in Java. This tokenizer also converts parentheses and squared brackets into the same normalized tokens (-LRB- or –RRB-).

Tokenizer 13 (MXPOST tagger) is a trainable, maximum entropy POS tagger, but its tokenization scheme is too simplistic as it breaks text only at white spaces. It is written in Java, but the source code is not available.

7. Conclusion

In this study, we tested and evaluated 13 tokenizers, which are freely available. We examined their various treatments on several scenarios and provided some qualitative guidelines for choosing a word tokenizer that may suit to the user needs. We outlined major differences in various tokenization schemes, which hopefully serve well to the NLP community and others who are interested in tokenization of free text.

Acknowledgement

This research was supported in part by the Intramural Research Program of the NIH, National Library of Medicine and the Oak Ridge Institute of Science and Education.

Reference

[1] B. Haber, et al. (1998) *Towards Tokenization Evaluation.* Proceedings of the First International Conference on Language Resources and Evaluation, Volume I, pp. 427-431, Granada, Spain.

[2] Kayaalp, M. (2004) *Modeling and Learning Methods*. A report to the Board of Scientific Counselors. Lister Hill National Center for Biomedical Communications, U.S. National Library of Medicine. LHNCBC-TR-2004-002, Bethesda, Maryland.

[3] NLTK, http://nltk.sourceforge.net/

[4] OpenNLP, http://opennlp.sourceforge.net/

[5] Mallet, http://mallet.cs.umass.edu/index.php/Main_Page

[6] SPECIALIST NLP Tools, http://lexsrv3.nlm.nih.gov/LexSysGroup/Projects/textTools/current/index.html

[7] Gump, http://www.mozart-oz.org/mogul/doc/lager/gump-tokenizer/

[8] Dan Melamed's Tool, http://www.cs.nyu.edu/~melamed/tools.html

[9] Qtoken, http://www.english.bham.ac.uk/staff/omason/software/qtoken.html

[10] UIUC word splitter, http://l2r.cs.uiuc.edu/~cogcomp/atool.php?tkey=WS

[11] LT TTT, http://www.ltg.ed.ac.uk/software/ttt/

[12] MedPost, http://bioinformatics.oxfordjournals.org/cgi/content/abstract/20/14/2320

[13] Brill's POS Tagger, http://www.umiacs.umd.edu/~jimmylin/downloads/brill-javadoc/index-all.html

[14] Stanford POS tagger, http://nlp.stanford.edu/software/tagger.shtml

[15] MXPOST tagger, http://www.cogsci.ed.ac.uk/~jamesc/taggers/MXPOST.htm

Appendix A

Additional information on the software packages that are mentioned in this report are provided below:

1. NLTK (Natural language Toolkit) developed at the University of Pennsylvania is an NLP package written in Python. We tested the tokenizer by running tokenize.simple.space() provided in NLTK-LITE version 0.6.3.
2. OpenNLP Tools is an open source project that contains a variety of NLP tools written in Java. It is based on the maximum entropy model. We tested the software package (version 1.3) by running opennlp.tools.lang.english.Tokenizer with EnglishTok.bin.gz as the underlying model of English.
3. Mallet developed at the University of Massachusetts at Amherst is a collection of Java codes for statistical NLP, document classification and clustering, information extraction, and various machine learning applications to text. We tested package edu.umass.cs.mallet.base.pipe.CharSequence2TokenSequence from Mallet version 0.4.
4. SPECIALIST NLP Tools developed at the U.S. National Library of Medicine constitutes a set of Java objects for the analysis of free text documents and identification words, terms, phrases, sentences and sections in the document. The tokenizer we tested was gov.nih.nlm.nls.nlp.tokenizer.TokenizerMain from textTools_V2.4.A.
5. Gump is a tokenizer written in a programming language with the same name. It was developed by Torbjörn Lager.
6. Dan Melamed's tokenizer, written in Perl (File name: tokenize), is one of his NLP tools.
7. Qtoken is a tokenizer developed by Oliver Mason at the University of Birmingham. It is written in Java.
8. Word Splitter was developed at the University of Illinois at Urbana Champaign. It requires running the Sentence Segmentation Tool (developed by the same group), prior running the tokenization module. It is a Perl module which inputs one sentence at a time and outputs the words and punctuation marks delimited by spaces.
9. LT TTT (Text Tokenisation Tool) is a toolset developed at the University of Edinburgh. It provides individually-tailored tokenization of text. We tested the sample script runplain from TTT version 2.
10. MedPost is a POS tagger developed at the U.S. National Library of Medicine and is written in Perl/C++. We tested it via `sh medpost -text -specialist`.

11. Brill's POS tagger algorithm was designed by Eric Brill. The Java package that we used in our tests was developed by Jimmy Lin. We tested the software by running edu.mit.csail.brill.BrillTagger command.
12. Stanford POS tagger is developed at Stanford University. It is written in Java. We tested this package (version 2006-1-20) by running edu.stanford.nlp.tagger.maxent.MaxentTagger with train-wsj-0-18 as the underlying model.
13. MXPOST tagger, developed by Adwait Ratnaparkhi, is a maximum entropy POS tagger. We tested the package by running command tagger.TestTagger.

Appendix B

In the following, we present outputs from each tokenizer for a sample sentence from MEDLINE. The ▲ mark indicates token boundaries.

The input sentence:

```
Independent of current body composition, IGF-I levels at 5 yr
were significantly associated with rate of weight gain
between 0-2 yr (beta = 0.19; P &lt; 0.0005), and children
who showed postnatal catch-up growth (i.e. those who
showed gains in weight or length between 0-2 yr by >0.67 SD
score) had higher IGF-I levels than other children (P = 0.02).
```

(1) NLTK tokenizer:

```
Independent▲of▲current▲body▲composition,▲IGF-I▲levels▲at▲5▲yr▲
were▲significantly▲associated▲with▲rate▲of▲weight▲gain▲
between▲0-2▲yr▲(beta▲=▲0.19;▲P▲&lt;▲0.0005),▲and▲children▲
who▲showed▲postnatal▲catch-up▲growth▲(i.e.▲those▲
who▲showed▲gains▲in▲weight▲or▲length▲between▲0-2▲yr▲by▲>0.67▲SD▲
score)▲had▲higher▲IGF-I▲levels▲than▲other▲children▲(P▲=▲0.02).
```

(2) OpenNLP tokenizer:

```
Independent▲of▲current▲body▲composition▲,▲IGF-I▲levels▲at▲5▲yr▲
were▲significantly▲associated▲with▲rate▲of▲weight▲gain▲
between▲0-2▲yr▲(▲beta▲=▲0.19▲;▲P▲&lt;▲0.0005▲)▲,▲and▲children▲
who▲showed▲postnatal▲catch-up▲growth▲(▲i▲.▲e▲.▲those▲who▲
showed▲gains▲in▲weight▲or▲length▲between▲0-2▲yr▲by▲>0.67▲SD▲
score▲)▲had▲higher▲IGF-I▲levels▲than▲other▲children▲(▲P▲=▲0.02▲)▲.
```

(3) Mallet tokenizer:

```
Independent▲of▲current▲body▲composition▲IGF▲I▲levels▲at▲yr▲
were▲significantly▲associated▲with▲rate▲of▲weight▲gain▲
between▲yr▲beta▲P▲lt▲and▲children▲
who▲showed▲postnatal▲catch▲up▲growth▲i▲e▲those▲who▲
showed▲gains▲in▲weight▲or▲length▲between▲yr▲by▲SD▲
score▲had▲higher▲IGF▲I▲levels▲than▲other▲children
```

(4) SPECIALIST NLP tokenizer:

```
Independent▲of▲current▲body▲composition▲,▲IGF▲-▲I▲levels▲at▲5▲yr▲
were▲significantly▲associated▲with▲rate▲of▲weight▲gain▲
between▲0▲-▲2▲yr▲(▲beta▲=▲0▲.▲19▲;▲P▲&▲lt▲;▲0▲.▲0005▲)▲,▲and▲children▲
who▲showed▲postnatal▲catch▲-▲up▲growth▲(▲i▲.▲e▲.▲those▲who▲
showed▲gains▲in▲weight▲or▲length▲between▲0▲-▲2▲yr▲by▲>▲0▲.▲67▲SD▲
score▲)▲had▲higher▲IGF▲-▲I▲levels▲than▲other▲children▲(▲P▲=▲0▲.▲02▲)▲.
```

(5) Gump tokenizer:

```
Independent▲of▲current▲body▲composition▲,▲IGF-I▲levels▲at▲5▲yr▲
were▲significantly▲associated▲with▲rate▲of▲weight▲gain▲
between▲0-2▲yr▲(▲beta▲=▲0.19▲;▲P▲&▲lt▲;▲0.0005▲)▲,▲and▲children▲
who▲showed▲postnatal▲catch-up▲growth▲(▲i.e.▲those▲who▲
showed▲gains▲in▲weight▲or▲length▲between▲0-2▲yr▲by▲>▲0.67▲SD▲
score▲)▲had▲higher▲IGF-I▲levels▲than▲other▲children▲(▲P▲=▲0.02▲)▲.
```

(6) Dan Melamed's tokenizer:

```
Independent␣of␣current␣body␣composition␣,␣IGF␣-␣I␣levels␣at␣5␣yr␣
were␣significantly␣associated␣with␣rate␣of␣weight␣gain␣
between␣0␣-␣2␣yr␣(␣beta␣=␣0.19␣;␣P␣&lt;␣0.0005␣)␣,␣and␣children␣
who␣showed␣postnatal␣catch␣-␣up␣growth␣(␣i␣.␣e␣.␣those␣who␣
showed␣gains␣in␣weight␣or␣length␣between␣0␣-␣2␣yr␣by␣>␣0.67␣SD␣
score␣)␣had␣higher␣IGF␣-␣I␣levels␣than␣other␣children␣(␣P␣=␣0.02␣)␣.
```

(7) Qtoken:
same as (4).

(8) UIUC word splitter:

```
Independent␣of␣current␣body␣composition␣,␣IGF-I␣levels␣at␣5␣yr␣
were␣significantly␣associated␣with␣rate␣of␣weight␣gain␣
between␣0-2␣yr␣-LBR-␣beta␣=␣0.19␣;␣P␣&␣lt␣;␣0.0005␣-RBR-
␣,␣and␣children␣
who␣showed␣postnatal␣catch-up␣growth␣-LBR-␣i.e.␣those␣who␣
showed␣gains␣in␣weight␣or␣length␣between␣0-2␣yr␣by␣>␣0.67␣SD␣
score␣-RBR-␣had␣higher␣IGF-I␣levels␣than␣other␣children␣-LBR-
␣P␣=␣0.02␣-RBR-␣.
```

(9) LT TTT tokenizer:

```
Independent␣of␣current␣body␣composition␣,␣IGF␣-␣I␣levels␣at␣5␣yr␣
were␣significantly␣associated␣with␣rate␣of␣weight␣gain␣
between␣0␣-␣2␣yr␣(␣beta␣=␣0.19␣;␣P␣&␣lt␣;␣0.0005␣)␣,␣and␣children␣
who␣showed␣postnatal␣catch-up␣growth␣(␣i.e.␣those␣who␣
showed␣gains␣in␣weight␣or␣length␣between␣0␣-␣2␣yr␣by␣>0.67␣SD␣
score␣)␣had␣higher␣IGF␣-␣I␣levels␣than␣other␣children␣(␣P␣=␣0.02␣)␣.
```

(10) MedPost tokenizer:

```
Independent␣of␣current␣body␣composition␣,␣IGF-I␣levels␣at␣5␣yr␣
were␣significantly␣associated␣with␣rate␣of␣weight␣gain␣
between␣0-2␣yr␣(␣beta␣=␣0.19␣;␣P␣&lt;␣0.0005␣)␣,␣and␣children␣
who␣showed␣postnatal␣catch-up␣growth␣(␣i.e.␣those␣who␣
showed␣gains␣in␣weight␣or␣length␣between␣0-2␣yr␣by␣>␣0.67␣SD␣
score␣)␣had␣higher␣IGF-I␣levels␣than␣other␣children␣(␣P␣=␣0.02␣)␣.
```

(11) Brill's POS tagger

```
Independent␣of␣current␣body␣composition␣,␣IGF-I␣levels␣at␣5␣yr␣
were significantly␣associated␣with␣rate␣of␣weight␣gain␣
between␣0-2␣yr␣(␣beta␣=␣0␣.␣19␣;␣P␣&␣lt␣;␣0␣.␣0005␣)␣,␣and␣children␣
who␣showed␣postnatal␣catch-up␣growth␣(␣i␣.␣e␣.␣those␣who␣
showed␣gains␣in␣weight␣or␣length␣between␣0-2␣yr␣by␣>␣0␣.␣67␣SD␣
score␣)␣had␣higher␣IGF-I␣levels␣than␣other␣children␣(␣P␣=␣0␣.␣02␣)␣.
```

(12) Stanford POS tagger

```
Independent␣of␣current␣body␣composition␣,␣IGF-I␣levels␣at␣5␣yr␣
were significantly␣associated␣with␣rate␣of␣weight␣gain␣
between␣0-2␣yr␣-LRB-␣beta␣=␣0.19␣P␣&lt;␣0.0005␣-RRB-␣,␣and␣children␣
who␣showed␣postnatal␣catch-up␣growth␣-LRB-␣i.e.␣those␣who␣
showed␣gains␣in␣weight␣or␣length between␣0-2␣yr␣by␣-RRB-␣0.67␣SD␣
score␣-RRB-␣had␣higher␣IGF-I␣levels␣than␣other␣children␣-LRB-
␣P␣=␣0.02␣-RRB-␣.
```

(13) MXPOST tagger
Its output is the same as (1) for this example. Their outputs differ only if the input contains symbols such as ê.

Appendix C

Case Set I

Hyphenated compound words

There are two main ways of tokenizing hyphenated compound words: to keep each compound word as one token or to break it to two or more tokens. Tokenizers 1, 2, 5, 8, 10, 11, 12, and 13 follow the first strategy, and tokenizers 4, 6, and 7 follow the second strategy. However, tokenizers 3 and 9 are different. Tokenizer 3 removes hyphens from the output. Tokenizer 9 treats lower case hyphenated compound words differently from the upper case ones, with the former being one token and the latter being multiple tokens. For example:

- `bottle-fed`
  - → `bottle-fed` (no change) by tokenizers 1, 2, 5, 8, 9, 10, 11, 12, and 13
  - → `bottle␣-␣fed` by tokenizers 4, 6, and 7
  - → `bottle␣fed` by tokenizer 3
- `Bottle-fed`
  - → `Bottle-fed` (no change) by tokenizers 1, 2, 5, 8, 10, 11, 12, and 13
  - → `Bottle␣-␣fed` by tokenizers 4, 6, 7, and 9
  - → `Bottle␣fed` by tokenizer 3

Words with letters and slashes

Words with slashes can indicate alternatives (`insertion/deletion`), units (`mg/day`) and locations (`Bethesda/MD`) etc. There are also two main strategies for a word with slashes:

1. Producing one token as done by tokenizers 1, 2, 5, 8, 9, 11, 12, and 13, or
2. Producing two or more tokens by delimiting at slashes by tokenizers 3, 4, 6, 7, and 10 (tokenizer 3 does not keep slash as token).

The first approach fails when a component on either side of slash is a hyphenated compound word (`blood-air/water`), or when a component on either side may form a phrase with the word before or after (`North Carolina/USA`).

Words with letters and apostrophes

Words with apostrophes can indicate possessive (e.g. `years'` and `Freud's`), contraction (e.g. `haven't` and `o'clock`), words with single quotation (e.g. `'repertorie cloning'`) and name (e.g. `O'Neill`) etc. There are a variety of tokenization strategies.

- `years'`
  - → `years'` (no change) by tokenizers 1, 5, 6, and 13
  - → `years␣'` by tokenizers 2, 4, 7, 8, 9, 10, 11, and 12
  - → `years` by tokenizer 3
- `Freud's`
  - → `Freud's` (no change) by tokenizers 1, 6, and 13
  - → `Freud␣s` by tokenizer 3
  - → `Freud␣'␣s` by tokenizers 4 and 7
  - → `Freud␣'s` by tokenizers 2, 5, 8, 9, 10, 11, and 12
- `haven't`
  - → `haven't` (no change) by tokenizers 1, 6, 9, 10, and 13

→ have␣n't by tokenizers 2, 5, 11, and 12
→ haven␣t by tokenizer 3
→ haven␣'␣t by tokenizers 4 and 7
→ haven␣'t by tokenizer 8

- o'clock
  → o'clock (no change) by tokenizers 1, 2, 6, 9, 10, 11, 12, and 13
  → o␣'clock by tokenizer 8
  → o␣clock by tokenizer 3
  → o␣'␣clock by tokenizers 4, 5, and 7
- 'reportorie␣cloning'
  → 'reportorie␣cloning' (no change) by tokenizers 1, 6, and 13
  → 'reportorie␣cloning␣' by tokenizers 2 and 11
  → 're␣portorie␣cloning␣' by tokenizer 5
  → reportorie␣cloning by tokenizer 3
  → '␣reportorie␣cloning␣' by tokenizers 4, 7, 8, 9, 10, and 12
- O'Neill
  → O'Neill (no change) by tokenizers 1, 2, 6, 9, 10, 11, and 13
  → O␣'Neill by tokenizer 8
  → O␣Neill by tokenizer 3
  → O␣'␣Neill by tokenizers 4, 5, 7, and 12

Words with letters and one type of brackets

There are basically four types of brackets: parentheses (), square brackets [], braces {}, and angle brackets <>. Tokenizers except 1, 2, 3, 9, and 13 separate parentheses and square brackets from the word attached to it; whereas, tokenizers 8 and 12 convert brackets to other symbols. Tokenizer 3 removes brackets altogether. Tokenizers 1, 9, and 13 always keep parentheses with word together if there is no space between them. Tokenizer 2 is inconsistent. For example, (tissue) is split into two tokens, (tissue and), and (Dipnoi) is split into two tokens (and Dipnoi), but (TP) is split into three tokens (, TP, and). Note that all left angle brackets are replaced by symbol < in MEDLINE. For angle brackets, however, it is still difficult to know if an angle bracket is truly a bracket or a *less than* or *greater than* sign.

- (tissue)
  → (tissue) (no change) by tokenizers 1, 9, and 13
  → (tissue␣) by tokenizer 2
  → tissue by tokenizer 3
  → (␣tissue␣) by tokenizers 4, 5, 6, 7, 10, and 11
  → -LBR-␣tissue␣-RBR- by tokenizer 8
  → -LRB-␣tissue␣-RRB- by tokenizer 12
- (Dipnoi)
  → (Dipnoi) (no change) by tokenizers 1, 9, and 13
  → (␣Dipnoi) by tokenizer 2
  → Dipnoi by tokenizer 3
  → (␣Dipnoi␣) by tokenizers 4, 5, 6, 7, 10, and 11
  → -LBR-␣Dipnoi␣-RBR- by tokenizer 8
  → -LRB-␣Dipnoi␣-RRB- by tokenizer 12

- `(TP)`
  - → `(TP)` (no change) by tokenizers 1, 9, and 13
  - → `TP` by tokenizer 3
  - → `(␣TP␣)` by tokenizers 2, 4, 5, 6, 7, 10, and 11
  - → `-LBR-␣TP␣-RBR-` by tokenizer 8
  - → `-LRB-␣TP␣-RRB-` by tokenizer 12
- `[GABAergic]`
  - → `[GABAergic]` (no change) by tokenizers 1, 9, and 13
  - → `[GABAergic␣]` by tokenizer 2
  - → `GABAergic` by tokenizer 3
  - → `[␣GABAergic␣]` by tokenizers 4, 5, 6, 7, 10, and 11
  - → `-LBR-␣GABAergic␣-RBR-` by tokenizer 8
  - → `-LRB-␣GABAergic␣-RRB-` by tokenizer 12

Words with letters and periods

Words with a period at the end usually indicate end of sentence. However, they may merely be abbreviations, such as `i.e.` and `e.g.`. There is no uniform scheme to differentiate tokenizers. Some tokenizers, such as tokenizer 9, use dictionary to identify abbreviated words and some others tokenizers, such as tokenizer 2, are trained using tokenized text.

- `i.e.`
  - → `i.e.` (no change) by tokenizers 1, 5, 8, 9, 10, 12, and 13
  - → `i␣e` by tokenizer 3
  - → `i␣.␣e␣.` by tokenizers 4, 6, 7, and 11
  - → `i␣.e␣.` by tokenizer 2
- `e.g.`
  - → `e.g.` (no change) by tokenizers 1, 2, 5, 8, 9, 10, 12, and 13
  - → `e␣g` by tokenizer 3
  - → `e␣.␣g␣.` by tokenizers 4, 6, 7, and 11

Words with only letters and numbers

Tokenizers 8 and 9 separate numbers and letters: both split `CO2` into `CO` and `2`, but only tokenizer 8 splits `12th` into `12` and `th`. Tokenizer 3 removes numeric portions from the word. All other tokenizers keep such words as whole unit.

- `CO2`
  - → `CO` by tokenizer 3
  - → `CO␣2` by tokenizers 8 and 9
  - → `CO2` (no change) by others
- `12th`
  - → `th` by tokenizer 3
  - → `12␣th` by tokenizer 8
  - → `12th` (no change) by others

Words with numbers and one type of punctuations

Some simple examples for numbers are large numbers (e.g. `3,000,000`), fractions (e.g. `1/2`), percentages (e.g. `76%`), decimals (e.g. `1.4`), and ranges (e.g. `1-20`) etc. Most tokenizers either keep them as a whole or separate them at punctuations with some

exceptions—tokenizer 3 outputs no tokens, and tokenizers 11 and 12 sometimes miss some tokens in their outputs.

- `3,000,000`
  - → `3,000,000` (no change) by tokenizers 1, 2, 5, 6, 8, 9, 10, 12, and 13
  - → (nothing) by tokenizer 3
  - → `3␣,␣000␣,␣000` by tokenizers 4, 7, and 11
- `1/2`
  - → `1/2` (no change) by tokenizers 1, 2, 5, 9, and 13
  - → (nothing) by tokenizer 3
  - → `1␣/␣2` by tokenizers 4, 6, 7, 8, and 10
  - → `1` by tokenizer 11
  - → `1\` by tokenizer 12
- `76%`
  - → `76%` (no change) by tokenizers 1, 5, and 13
  - → (nothing) by tokenizer 3
  - → `76␣%` by tokenizers 2, 4, 6, 7, 8, 9, 10, and 11
  - → `76` by tokenizer 12
- `1.4`
  - → `1.4` (no change) by tokenizers 1, 2, 5, 6, 8, 9, 10, 12, and 13
  - → (nothing) by tokenizer 3
  - → `1␣.␣4` by tokenizers 4, 7, and 11
- `1–20`
  - → `1–20` (no change) by tokenizers 1, 2, 5, 8, 10, 11, 12, and 13
  - → (nothing) by tokenizer 3
  - → `1␣–␣20` by tokenizers 4, 6, 7, and 9

Case Set II

The following cases illustrate finer differences among tokenizers, from named entity point of view. In these cases, each input involves more than one type of punctuations, and where to break depends on the punctuation processing priority and/or order in tokenizers.

DNA sequences

- `5′-ATGCAAAT-3′`
  - → `5′-ATGCAAAT-3′` (no change) by tokenizers 1, 2, and 13
  - → `ATGCAAAT` by tokenizer 3
  - → `5′-ATGCAAAT-3␣′` by tokenizers 10 and 11
  - → `5′␣-␣ATGCAAAT␣-␣3′` by tokenizer 6
  - → `5␣′␣-␣ATGCAAAT-3␣′` by tokenizers 5, 8, and 12
  - → `5␣′␣-␣ATGCAAAT-␣3␣′` by tokenizer 9
  - → `5␣′␣-␣ATGCAAAT␣-␣3␣′` by tokenizers 4 and 7

Chemical substances

- `3,4-epoxy-3-methyl-1-butyl-diphosphate`
  - → `3,4-epoxy-3-methyl-1-butyl-diphosphate` (no change) by tokenizers 1, 2, 8, 12, and 13
  - → `3,4␣-␣epoxy-3-methyl-1-butyl-diphosphate` by tokenizer 5

→ `3␣,␣4-epoxy-3-methyl-1-butyl-diphosphate` by tokenizers 10 and 11
→ `epoxy␣methyl␣butyl␣diphosphate` by tokenizer 3
→ `3,4␣-␣epoxy-3-methyl␣-␣1␣-␣butyl-diphosphate` by tokenizer 9
→ `3,4␣-␣epoxy␣-␣3␣-␣methyl␣-␣1␣-␣butyl␣-␣diphosphate` by tokenizer 6
→ `3␣,␣4␣-␣epoxy␣-␣3␣-␣methyl␣-␣1␣-␣butyl␣-␣diphosphate` by tokenizers 4 and 7

Arithmetic expressions
Arithmetic expressions are words with numbers and multiple types of operators. There are usually many ways to tokenize them. As operators imply special meanings and precedences, the correctness of a particular tokenization approach is less subjective compared to that in free text. For example, the outputs from tokenizers 9 and 11 of the first example below and the outputs from tokenizers 5 and 9 of the second example below are undesirable.

- `76.8+/-4.2`
  → `76.8+/-4.2` (no change) by tokenizers 1, 2, and 13
  → (nothing) by tokenizer 3
  → `76.8␣+/-4.2` by tokenizer 9
  → `76.8␣+/-␣4.2` by tokenizer 10
  → `76.8␣+␣/␣-4.2` by tokenizer 12
  → `76.8␣+␣/␣-␣4.2` by tokenizers 5, 6, and 8
  → `76␣.␣8+/-4␣.␣2` by tokenizer 11
  → `76␣.␣8␣+␣/␣-␣4␣.␣2`. by tokenizers 4 and 7
- `8.4-13.8%`
  → `8.4-13.8%` (no change) by tokenizers 1 and 13
  → (nothing) by tokenizer 3
  → `8.4-13.8␣%` by tokenizers 2, 8, and 10
  → `8.4␣-13.8%` by tokenizer 5
  → `8.4-13␣.8␣%` by tokenizer 9
  → `8.4␣-13.8␣%` by tokenizer 12
  → `8.4␣-␣13.8␣%` by tokenizer 6
  → `8␣.␣4-13␣.␣8␣%` by tokenizer 11
  → `8␣.␣4␣-␣13␣.␣8␣%` by tokenizers 4 and 7

Hypertext markup symbols
Some of the frequently observed hypertext markup symbols are `<` and `&quot;` (for the double quotation mark). Only tokenizers 10 and 12 recognize these symbols as a whole unit. Tokenizers 1 and 13 keep them together with neighboring words. Other tokenizers break them into different set of tokens or remove punctuations.

- `<`
  → `<` (no change) by tokenizers 1, 10, 12, and 13
  → `lt` by tokenizer 3
  → `&lt␣;` by tokenizers 2 and 6
  → `&␣lt␣;` by tokenizers 4, 5, 7, 8, and 11

→ `&#38;␣lt␣;` by tokenizer 9

- `(P<0.001)`
  - → `(P<0.001)` (no change) by tokenizers 1 and 13
  - → `P␣lt` by tokenizer 3
  - → `(␣P&lt␣;0.001␣)` by tokenizer 2
  - → `(␣P&lt␣;␣0.001␣)` by tokenizers 5 and 6
  - → `(␣P␣<␣0.001␣)` by tokenizer 10
  - → `-LRB-␣P␣<␣0.001␣-RRB-` by tokenizer 12
  - → `-LBR-␣P␣&␣lt␣;␣0.001␣-RBR-` by tokenizer 8
  - → `(␣P␣&#38;␣lt␣;␣0.001␣)` by tokenizer 9
  - → `(␣P␣&␣lt␣;0␣.␣001␣)` by tokenizers 4, 7, and 11

URL

- `http://www.tobaccoarchives.com`
  - → `http://www.tobaccoarchives.com` (no change) by tokenizers 1, 2, 5, 12, and 13
  - → `http␣:␣//www.tobaccoarchives.com` by tokenizer 9
  - → `http␣www␣tobaccoarchives␣com` by tokenizer 3
  - → `http␣:␣/␣/␣www.tobaccoarchives.com` by tokenizers 8 and 10
  - → `http␣:␣/␣www␣.␣tobaccoarchives␣.␣com` by tokenizer 11
  - → `http␣:␣/␣/␣www␣.␣tobaccoarchives␣.␣com` by tokenizers 4, 6, and 7

Appendix D

Algorithms for these tokenizers can be categorized into two broad classes, using machine-learning techniques or using rules (see Table 2). Tokenizers 2, 11, 12, and 13 belong to the first class. Tokenizers 2, 12, and 13 are based on maximum entropy approach, and are trainable if customized annotations provided (In our test, tokenizer 13 acts like a simple splitter[*]). Tokenizer 11 uses a transformation-based error-driven learning approach. All other tokenizers are rule-based. As mentioned in Section 3, tokenizers 1, 3, 4, and 7 follow simple rules. Other rule-based tokenizers follow more complicated rules; e.g., using regular expressions. To compare speed of tokenizers, we run each tokenizer for a collection of 50 MEDLINE abstracts (about 6000 words) on a Linux machine with a 3GHz Intel Pentium D processor. Processing 6000 words takes less than 1 second by some tokenizers, such as NLTK. As seen on Table 2, source codes are available for most tokenizers but not for tokenizers 7, 9, and 13. For tokenizers 7 and 13, only Java .jar files are made available. All tokenizers use some scripting languages, sometimes along with C/C++.

Table 2: Comparison of Tokenizers

No	Name	Algorithm	Speed word/sec	Open Source	Language	Other
1	NLTK tokenizer	Simple splitter[*]	> 6000	Y	Python	
2	OpenNLP tokenizer	Maximum entropy	~ 400	Y	Java	Trainable
3	Mallet tokenizer	Simple splitter[*]	> 6000	Y	Java	
4	SPECIALIST NLP tokenizer	Simple splitter[*]	~ 600	Y	Java	
5	Gump tokenizer	Linguistic rules	> 6000	Y	Gump	
6	Dan Melamed's tokenizer	Linguistic rules	> 6000	Y	Perl	
7	Qtoken	Simple splitter[*]	~1500	**N**	Java	
8	UIUC word splitter	Linguistic rules	> 6000	Y	Perl	Sentence input
9	LT TTT tokenizer	Linguistic rules	~1000	Partial	Perl & others	Multiple input formats
10	MedPost tokenizer	Linguistic rules	~750	Y	Perl/C++	Trainable
11	Brill's POS tagger	Transformation-based error-driven learning	~300	Y	Java/C	Java wrapper
12	Stanford POS Tagger	Maximum entropy	~50	Y	Java	Trainable
13	MXPOST tagger	Maximum entropy	~200	**N**	Java	Trainable

[*] A simple splitter follows a simple rule to break sentence into tokens either at white space only or at white space and punctuations.