# The Xerox Part-of-Speech Tagger
# Version 1.0

Doug Cutting <cutting@parc.xerox.com> and
Jan Pedersen <pedersen@parc.xerox.com>

Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto, CA 94304, USA

April 12, 1993

## 1 Background

This document describes how to use the Xerox Part of Speech Tagger, both as delivered, for tagging English text with the Brown tagset, and how to retarget it to other tagsets and languages.

This document does not describe how the tagger works, nor does it tell much about the coding conventions used. For information on these, the reader is directed towards [1] and [2] respectively. For convenience, Postscript versions of these documents are included in the distribution, filed as `doc/anlp92.ps` and `papers/riao91.ps`.

## 2 Getting Started

This tagger release is available via anonymous FTP from the host `parcftp.xerox.com` in the directory `pub/tagger` in a file named `tagger-1-0.tar.Z`.

The tagger is written in ANSI Common Lisp. Development was done in Franz Allegro Common Lisp version 4.1, though the code should port more or less unchanged to other ANSI Common Lisp implementations. We have included in the distribution (in the `src/common-lisp/` subdirectory) a compatibility package intended to bring non-ANSI Common Lisp implementations into conformance.

### 2.1 Compilation

To compile the tagger, perform the following steps:

1. Make a directory, connect to it, and upack the distribution.
```
% mkdir tagger
% cd tagger
% tar xvf ../tagger.tar
```

2. Start Common Lisp.

3. Compile and load `pdefsys`:
```
> (compile-file "src/pdefsys")
> (load "src/pdefsys")
```

4. If running in a pre-ANSI Common Lisp (e.g., CMU CL v16):
```
> (pdefsys:compile-system :common-lisp)
```

5. Compile and load the system `tdb-sysdcl`:
   ```
   > (pdefsys:compile-system :tdb-sysdcl)
   > (pdefsys:load-system :tdb-sysdcl)
   ```

5. Compile and load the system `tag-english`:
   ```
   > (pdefsys:compile-system :tag-english :propagate t)
   > (pdefsys:load-system :tag-english)
   ```

## 2.2 Loading

To load the tagger, perform the following steps.

1. Start Common Lisp.

2. Load `pdefsys`:
   ```
   > (load "src/pdefsys")
   ```

3. Load the system `tdb-sysdcl`:
   ```
   > (pdefsys:load-system :tdb-sysdcl)
   ```

4. Load the system `tag-english`:
   ```
   > (pdefsys:load-system :tag-english)
   ```

We recommend loading at least `pdefsys` in your lisp init file.
The rest of this file assumes that the tagger has been loaded.

# 3 Running the Tagger

In addition to the source code, we include:

- a tokenizer for plain ASCII english;
- a lexicon for English induced from the Brown corpus [3];
- a table mapping word suffixes to likely ambiguity classes (to handle words not in the lexicon) inferred from the Brown corpus; and
- an HMM trained on the odd numbered sentences in the Brown corpus.

These are provided in the system `:tag-english`, used in the examples in this section. With this lexicon and HMM, correct tags are assigned to 96% of the words in the even numbered sentences of the Brown corpus. Note however that, when training this HMM, no unknown forms were encountered, providing no training data for forms assigned the open class (i.e., unknown words with unknown suffixes). One should consider retraining on representative text to defeat this problem. An HMM trained on the full text of the novel Moby Dick is included in the subdirectory `data/moby-dick`.

The functions `tag-analysis:tag-file` and `tag-analysis:tag-string` provide convenient diagnostic interfaces to the tagger. Both functions print an annotated version of the text passed to them. Examples of their use follows.

```
> (tag-analysis:tag-file "copy.txt")

A    copying machine is  a  device that  produces copies of original documents.
at/2 vbg     nn      bez at nn     wps/3 vbz      nns    in jj/2     nns

> (tag-analysis:tag-string "I saw the man on the hill with the telescope.")
```

```
I      saw   the man on the hill with the telescope.
ppss/2 vbd/3 at  nn  in at  nn   in/2 at  nn/2
```

(The number following the tag is the arity of the ambiguity class assigned by the lexicon. Words without a number are unambiguous.)

### 3.1 Programmatic Tagging

To use the tagger in a program, create a `tagging-ts` and use the values of calls to the generic function `next-token`. Note that `reinitialize-instance` redirects tagging to a new text with minimal initialization overhead.

For example, the following function, `my-tag-files`, calls `my-process-token-and-tag` on each token/tag pair generated by tagging each file in the argument `files`:

```
(use-package :tdb)
(use-package :tag-analysis)

(defun my-tag-files (files)
  (let ((token-stream (make-instance 'tagging-ts)))
    (dolist (file files)
      (with-open-file (char-stream file)
        (reinitialize-instance token-stream :char-stream char-stream)
        (loop (multiple-value-bind (token tag)
                  (next-token token-stream)
                (unless token (return))
                (my-process-token-and-tag token tag)))))))
```

## 4 Modifying the Tagger

In this section we describe how to retrain the tagger on different text, and how to change the tokenizer, lexicon, and various other parameters of the tagger.

All these parameters are specified by setting the variable `*tokenizer-class*` to a class of token stream which, when given text, returns tokens paired with ambiguity classes. This is typically done by defining a subclass of `basic-tag-ts` which specifies values for its parameters as follows:

```
(use-package :tag-basics)

(defclass my-tag-tokenizer (basic-tag-ts) ()
  (:default-initargs
    :tagger-pathname #p"/users/me/my-tagger/my.hmm"
    :tokenizer-fsa *my-tokenizer-fsa*
    :lexicon (or *my-lexicon*
                 (setq *my-lexicon* (make-instance 'my-lexicon)))
    :transition-biases *my-transition-biases*
    :symbol-biases *my-symbol-biases*))

(setq *tokenizer-class* (find-class 'my-tag-tokenizer))
```

These parameters are described further in the following sections.

## 4.1 Training

The parameter `:tagger-pathname` indicates where the HMM will be read from and written to.

To train a tagger, use the function `tag-trainer:train-on-files`, for example:

```
> (train-on-files (directory "*.txt"))
```

## 4.2 The Tokenizer FSA

The parameter `:tokenizer-fsa` specifies the finite state automaton to be used to break up the text to be tagged into tokens.

These are usually defined with the macro `fsa-tokenizer:def-tokenizer-fsa`, for example:

```
(use-package :fsa-tokenizer)
(def-tokenizer-fsa *my-tokenizer-fsa*
  (let* ((digit `(/ #\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9))
         (small `(/ #\a #\b #\c #\d #\e #\f #\g #\h #\i #\j #\k #\l #\m #\n
                    #\o #\p #\q #\r #\s #\t #\u #\v #\w #\x #\y #\z))
         (cap `(/ #\A #\B #\C #\D #\E #\F #\G #\H #\I #\J #\K #\L #\M #\N
                  #\O #\P #\Q #\R #\S #\T #\U #\V #\W #\X #\Y #\Z))
         (alpha `(/ ,small ,cap)))
    `((:word (+ ,alpha))
      (:number (! (+ ,digit) (* ,alpha)))
      (:sent (! (/ #\. #\? #\!)
                (+ (/ #\newline #\space))
                ,cap)
             1))))
```

A tokenizer is specified as a list of *rules*. Each rule consists of three parts: a type (named by a keyword), a regular expression and, optionally a *backoff*.

The types `:word` and `:sent` are treated specially by the tagger. Tokens of type `:word` will be looked up in the lexicon. All others will not. Tokens of type `:sent` identify sentence boundaries. The tagger processes text one sentence at a time.

All other types identify tokens as members of singleton ambiguity classes, i.e., unambiguously tag a token with that type.

Regular expressions are recursively composed of strings and characters with the following operators:

- `*` kleene star, matches zero or more occurrences of its argument;
- `+` matches one or more occurrences of its argument;
- `?` matches zero or one occurence of its argument;
- `/` matches any of its arguments;
- `!` matches the sequence of its arguments; and
- `-` matches those strings which match its first argument, but not its second argument.

Starting at a some point in the input, the tokenizer finds the longest string which matches a rule, returning the string and the type of the rule. The input is advanced to the next character after this match, and the tokenizer restarts. If no rule matches the input is advanced one character and the tokenizer restarts. If two rules match the same string, the one earlier in the definition is used.

If the matching rule has a backup specified, then the input is backed off that number of characters when that rule fires. This allows rules to specify right context.

The function `tag-analysis:tokenize-file` tokenizes a file, producing output in a format similar to that of `tag-analysis:tag-file`.

## 4.3 Specifying a lexicon

A lexicon may be specified by defining a class subclassing `vector-lexicon` and `suffix-lexicon`. The class `vector-lexicon` allows the specification of lexica by exhaustive enumeration, while `suffix-lexicon` infers part-of-speech ambiguity classes based upon word suffixes. The ordering of these superclasses determines which lexicon is consulted first.

For example:

```
(use-package :vector-lexicon)
(use-package :class-guesser)

(defvar *my-open-class* '(:noun :proper-noun :adjective :verb))

(defclass my-lexicon (vector-lexicon suffix-lexicon) ()
  (:default-initargs
    :lexicon-open-class *my-open-class*
    :lexicon-pathname #p"/users/me/my-tagger/lexicon.txt"
    :lexicon-size 15000
    :suffix-pathname #p"/users/me/my-tagger/suffix.trie")
```

The lexicon parameter `:lexicon-open-class` lists the open class tags. This is used when all other lookups fail.

The lexicon parameter `:lexicon-pathname` specifies a file containing the entries for the `vector-lexicon`. This file should consist of lines containing three forms, separated by spaces: *surface*, *tag* and *lexical*. The file must be lexicographically sorted, as by the Unix program `sort`. The surface forms are used to match input, while lexical forms are returned (with tag) by the tagger.

The lexicon parameter `:lexicon-size` provides a hint to `vector-lexicon` of the number of distinct surface forms in the file. Accurate specification of this improves efficiency, but is not required.

The lexicon parameter `:suffix-pathname` names the file where `suffix-lexicon` will store its suffix table. To create such a file, based on a lexicon and some training text, use the function `class-guesser:train-guesser-on-files`.

The function `tdb:lexicon-lookup`, given a surface form and a lexicon, returns as multiple values two parallel vectors containing lexical forms and corresponding tags. Note that if all of the lexical forms are identical, the first value is simply that string.

## 4.4 Specifying Biases

The parameters `:transition-biases` and `:symbol-biases` enable tuning of the tagger by influencing initial probabilities when training. For example:

```
(setq *my-transition-biases*
  '((:valid :to-infinitive :verb)
    (:invalid :determiner :verb :preposition)))
(setq *my-symbol-biases*
  `((:valid (:determiner :noun) :determiner)
    (:valid ,*my-open-class* :noun :proper-noun)))
```

Here `*my-transition-biases*` indicates that the infinitive marker is likely to be followed by a verb. Similarly, determiners are not likely to be followed by verbs or prepositions. A `:valid` clause is equivalent to an `:invalid` clause whose tail is the complement of the tags in the `:valid` clause, and vice versa.

And `*my-symbol-biases*` indicates that words in the ambiguity class `(:determiner :noun)` are likely to be determiners, and words which are assigned the open class (i.e., words otherwise unknown to the lexicon) are more likely to be nouns or proper nouns than anything else.

## References


[1] D. Cutting, J. Kupiec, J. Pedersen, and P. Sibun. A practical part-of-speech tagger. In *Proceedings of the Third Conference on Applied Natural Language Processing*, Trento, Italy, April 1992. ACL. Also available as Xerox PARC technical report SSL-92-01.

[2] D.R. Cutting, J. Pedersen, and P.-K. Halvorsen. An object-oriented architecture for text retrieval. In *Conference Proceedings of RIAO'91, Intelligent Text and Image Handling, Barcelona, Spain*, pages 285–298, April 1991. Also available as Xerox PARC technical report SSL-90-83.

[3] W. N. Francis and F. Kučera. *Frequency Analysis of English Usage*. Houghton Mifflin, 1982.