ISSN 2033-3544

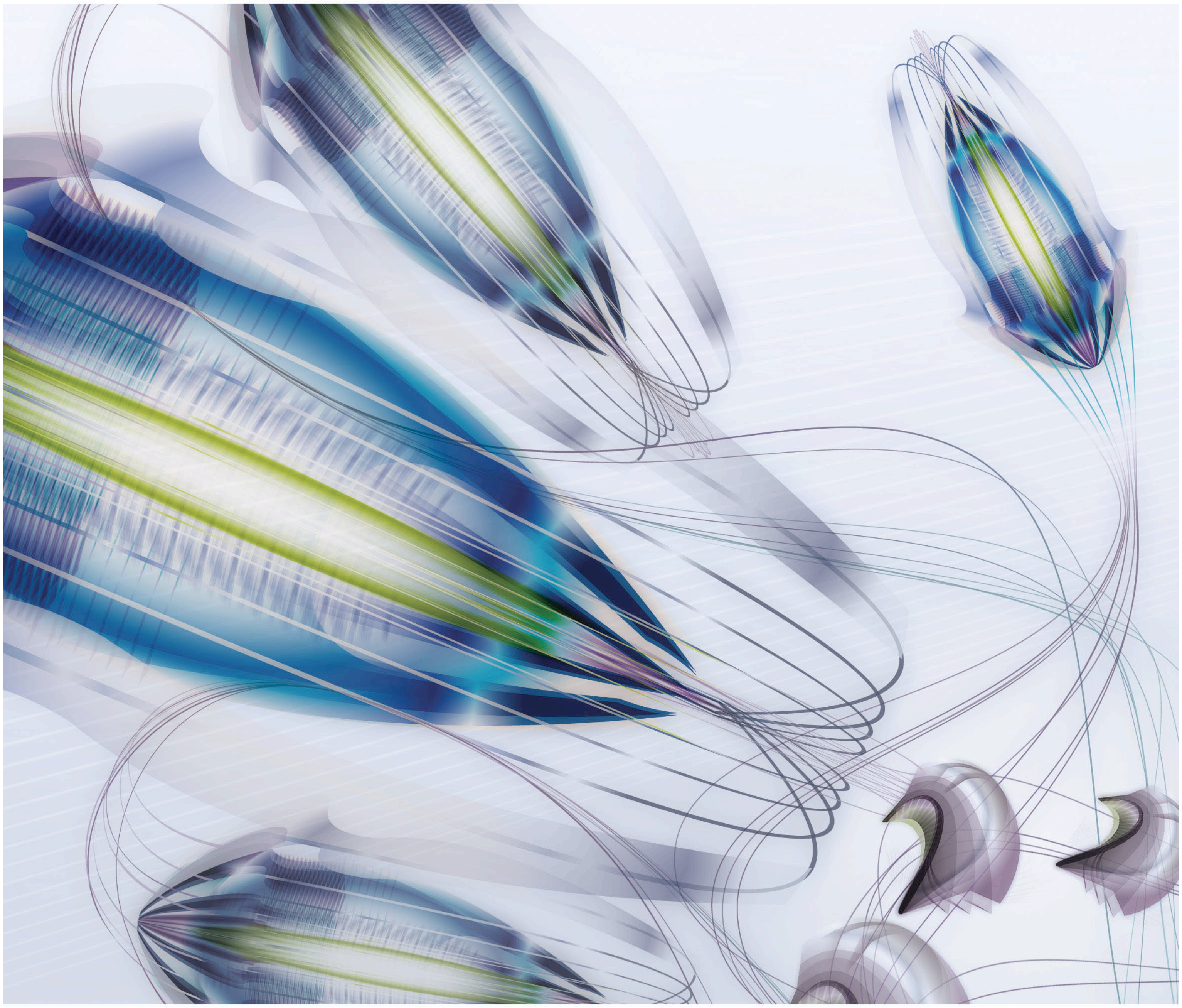
CLiPS

COMPUTATIONAL LINGUISTICS & PSYCHOLINGUISTICS
TECHNICAL REPORT SERIES, CTRS-002, SEPT 2010

WWW.CLIPS.UA.AC.BE/CTRS

Memory-based Shallow Parser for Python

Tom De Smedt, Vincent Van Asch, Walter Daelemans



Universiteit Antwerpen

COMPUTATIONAL LINGUISTICS & PSYCHOLINGUISTICS RESEARCH CENTER CLiPS

CTRS-002
September 7, 2010

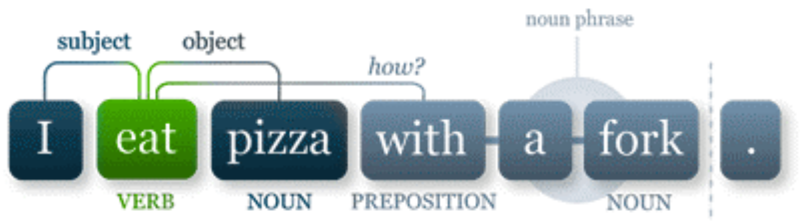
MBSP for Python

Tom De Smedt, Vincent Van Asch, Walter Daelemans
Computational Linguistics & Psycholinguistics Research Center, University of Antwerp
tom.desmedt | vincent.vanasch | walter.daelemans @ua.ac.be
http://www.clips.ua.ac.be/pages/MBSP

MBSP for Python

MBSP is a text analysis system based on the TiMBL and MBT memory based learning applications developed at CLiPS and ILK. It provides tools for Tokenization and Sentence Splitting, Part of Speech Tagging, Chunking, Lemmatization, Relation Finding and Prepositional Phrase Attachment.

The general English version of MBSP has been trained on data from the Wall Street Journal corpus. The Python implementation of MBSP is open source and freely available.



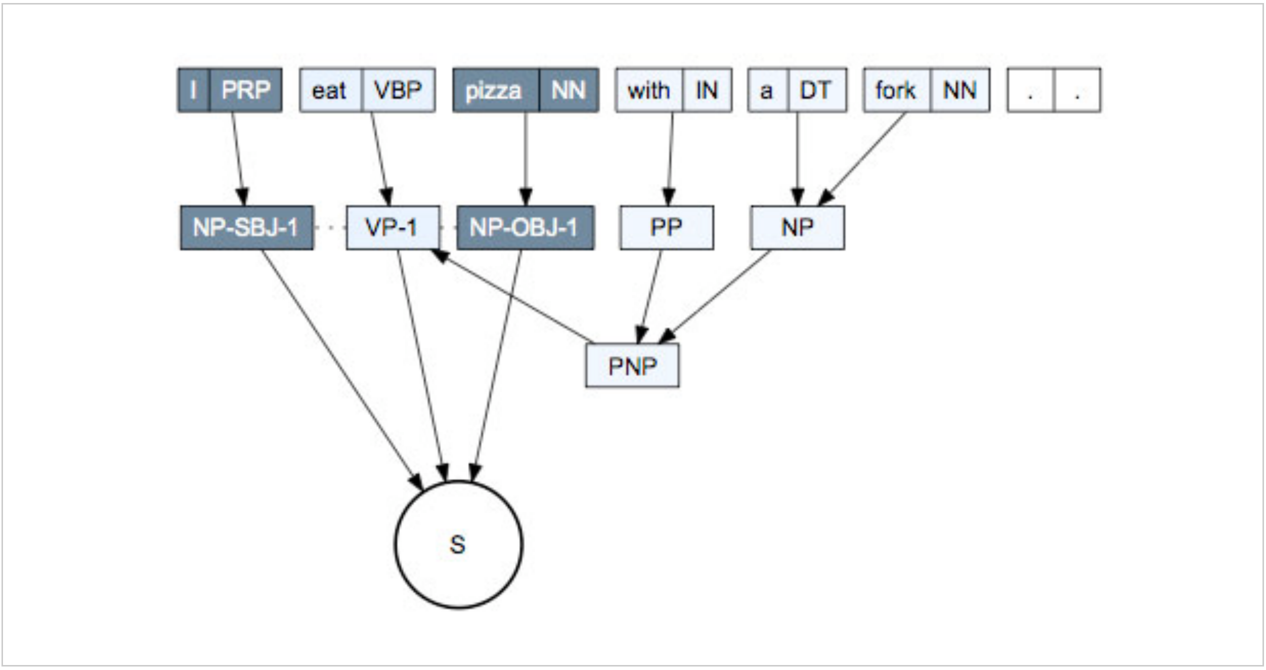
- DOWNLOAD: http://www.clips.ua.ac.be/pages/MBSP
- LICENSE: GPL
- PACKAGE AUTHORS:
  Vincent Van Asch (vincent.vanasch at ua.ac.be),
  Tom De Smedt (tom at organisms.be)

Reference: Daelemans, W., & Van den Bosch, A. (2005). *Memory-based language processing*. Cambridge University Press, Cambridge, UK. ISBN-13: 9780521808903 | ISBN-10: 0521808901

Preface

This reference guide describes a particular implementation of a “parser”: a software tool that transforms a sentence into a syntactic structure, as a first step in analyzing the meaning of the sentence. The aim of this preface is to provide some background on the concept of "shallow parsing" in the context of syntactic analysis methods in general.

A parser is a system that transforms sentences (strings of characters) into a representation that describes the groupings of words (phrases) and their relations (e.g. subject and object). The representation of choice for such information is a syntactic tree in which nodes refer to phrases, word categories, or words, and links refer to relations between these objects:



Sample output from MBSP visualized with GraphViz (see Section 12).

In Computational Linguistics, different approaches exist to building a parser. One option is to construct a grammar – a set of rules that describes the syntactic structures that are *expected* to occur in the sentences of a language. Combined with a computer-readable dictionary and a search algorithm a grammar can produce (generate) all possible syntactic structures a sentence can have. Unfortunately, there can be many of them (most of them nonsensical) and the parser will not assign more weight to the more relevant ones. A grammar can be constructed by hand by a linguist, but can also be induced automatically from a “treebank” (a text corpus in which each sentence is annotated with its syntactic structure).

Although the annotation of a sizable corpus of text with syntactic structures is time-consuming, the effort pays off: from the treebank a grammar can be induced based on actual language use rather than on the often idealized image a linguist has about what is "grammatical". Moreover, the induced grammar and dictionary are probabilistic. They contain statistics on the grammar rules, and on the association of words with syntactic categories. The many syntactic analyses that can be constructed for any sentence can at the same time be ordered according to probability.

The type of syntactic analysis a sentence gets depends on the type of grammar used. Constituent-based grammars focus on the hierarchical phrase structure of a sentence. Dependency-based grammars focus on (grammatical) relations between words. However, all of them attempt to provide a syntactic analysis that is as complete as possible. Given that syntactic analysis is mainly useful as an intermediate step toward semantic interpretation, detailed syntactic information is not always necessary. What we need for applications in many text-mining applications is a robust, efficient, accurate, and deterministic analysis of a sentence in terms of its main constituents and the relations between them.

Shallow parsing

Shallow parsing is an approach to achieve this. The syntactic parsing process is carved up into a set of classification problems, each of which can be separately learned using standard supervised machine learning methods. The MBSP modules include part of speech tagging, phrase labeling and grammatical relation finding for English. In combination, these modules produce a syntactic analysis of sentences that is detailed enough to drive further semantic and application-oriented processing. Especially in applications such as information retrieval, question answering, and information extraction, where large volumes of (often ungrammatical) text have to be analyzed in an efficient and robust way, shallow parsing is useful.

The concept of shallow parsing, proposed by Abney[1] (1991) has no clearly defined meaning, and is used sometimes in a very limited sense, referring only to tagging and chunking, and sometimes in a broader sense, referring also to semantic tasks such as named-entity recognition. It can best be interpreted as a family of related tasks attempting to recover some syntactic-semantic information in a robust and deterministic way at the expense of ignoring detailed configurational syntactic information. In our approach to shallow parsing, we use memory-based learning as machine learning method, and defined the following modules in the current version:

- PART-OF-SPEECH TAGGING AND CHUNKING.
  After tokenization and sentence splitting, each word in each sentence of an input text is assigned its grammatical category. At the same time, constituents like noun phrases, verb phrases, prepositional phrases etc. are detected.
- GRAMMATICAL RELATIONS.
  On the basis of the predicted constituents, grammatical relations between them are predicted (e.g. subject, object, etc.)
- PREPOSITIONAL PHRASE ATTACHMENTS.
  Prepositional phrases are related to the constituent to which they belong.

A machine learning approach to automatic text analysis is only as good as the treebank material on which it is trained. It is also the case that any available treebank is not representative for language as a whole but only (if large enough) for the domain from which it was sampled. This means, for example, that a shallow parser trained on a corpus that contains descriptions of machine parts will not perform very well on text about astronomy. Our shallow parser currently provides two versions: one trained on general newspaper language, and one on biomedical language (included in the module as an example).

[1] Abney, S. (1991). *Parsing By Chunks*. In: Robert Berwick, Steven Abney and Carol Tenny (eds.), Principle-Based Parsing. Kluwer Academic Publishers, Dordrecht.

Table of contents

1 Introduction

1.1 Quick overview

MBSP parses a string of characters into words and sentences, and determines the grammatical structure of the sentence. You will need Python 2.5+ to run it (http://python.org/download, already installed on Mac OS X). The module uses a client-server architecture for performance. It includes binaries (TiMBL, MBT, MBLEM) precompiled for Mac OS X, so on Mac it works out-of-the-box. Otherwise, if you're on a Unix system, the module has a `setup.py` file that should compile everything for you. Open the terminal and type:

```
> cd MBSP
> python setup.py
```

If that doesn't work you'll need to follow the steps in the installation instructions.

Put the MBSP folder in the same folder as your Python script and import the module. By default, the servers are configured to start automatically. Once they are up and running you can use the `parse()` command to analyze texts:

```
>>> import MBSP
>>> print MBSP.parse('cats with hats')

cats/NNS/I-NP/O/O/A1/cat with/IN/I-PP/B-PNP/O/P1/with hats/NNS/I-NP/I-PNP/O/P1/hat
```

Each word has been tagged with grammatical information. For example, MBSP determined that *cats* is a plural noun (**NNS**). It has a prepositional noun phrase (**PNP**) attached to it (**A1** is the anchor of **P1**), so the *hats* go *with* the *cats*. To a human this might seem pretty straightforward, but without analysis, for a machine the sentence is just a sequence of characters with no meaning. The tag codes can appear cryptic at first, but consider that it is more concise to say **NNS** than **PLURAL NOUN** over and over. The tag codes are common in natural language processing, it's a good idea to get acquainted with them – see the Appendix for an overview.

Something went wrong? Probably the servers didn't have enough time to start:

```
>>> MBSP.start(timeout=120)
>>> print MBSP.parse('cats with hats')
```

The output of the `parse()` command is a tagged string that can be manipulated in many ways. With the `split()` command it can be transformed into a tree of linked Python objects:

```
>>> s = MBSP.parse('black cats with striped hats')
>>> s = MBSP.split(s)
>>> for sentence in s:
>>>     for chunk in sentence.chunks:
>>>         print [word.lemma for word in chunk.words], chunk.attachments

[u'black', u'cat'] [Chunk('with striped hats/PNP')]
[u'with'] []
[u'striped', u'hat'] []
```

1.2 Purpose

MBSP stands for "Memory-Based Shallow Parser". Shallow parsing (i.e. automatic discovery of a sentence constituents) is an important component of many text analysis systems, in applications such as information extraction and summary generation. The Memory-Based Learning (MBL) approach has the advantage of avoiding the need for manual definition of patterns (for example, using regular expression syntax) and of being reusable across different corpora and sublanguages. MBSP is a so-called *lazy learner*: it keeps all the initial training data available (including exceptions which may sometimes be productive). This technique has been shown to achieve higher accuracy than *eager* (or *greedy*) methods for many language processing tasks. For the Wall Street Journal corpus (WSJ), Fβ = 1 is 93.8% for **NP** chunking, 94.7% for **VP** chunking, 77.1% for **SBJ** detection and 79.0% for **OBJ** detection. MBSP is based on the `IB1-IG` and `IGTREE` algorithms bundled in our MBL software package, called TiMBL.

Reference: Daelemans, W., Buchholz, S., & Veenstra, J. (1999). *Memory-Based Shallow Parsing*. In: Proceedings of CoNLL, Bergen, Norway.

The parser provides functionality for tokenization and sentence splitting, part-of-speech tagging, chunking, relation finding, prepositional phrase attachment and lemmatization:

- **Tokenization**: splits sentence periods and punctuation marks from words.
- **Tagging**: assigns part-of-speech tags to words (e.g. *cat* → noun → **NN**, *eat* → verb → **VB**).
- **Chunking**: assigns chunk tags to groups of words (e.g. *the black cat* → noun phrase → **NP**).
- **Relation finder**: finds relations between chunks, sentence subject, object and predicates.
- **PNP finder**: finds prepositional noun phrases (e.g. *under the table*).
- **PP-attachment**: finds prepositional noun phrase anchors (e.g. *eat pizza* → *with fork*).
- **Lemmatization**: finds word lemmata (e.g. *was* → *be*).

1.3 Grammar basics

Sentences are made up of words. Words have a syntactic role (noun, verb, adjective, ...) depending on their location in the sentence. For example, *can* can be a verb or a noun, depending on the context (*the can*, *I can*).

- **Sentence**: the basic unit of writing, expected to have a subject and a predicate.
- **Word**: a string of characters that expresses a meaningful concept.
- **Token**: a specific word with grammatical tags: *the can*/**NN**, *I can*/**VB**.
- **Chunk**: a group of words (=phrase) that contains a single thought (e.g. *a sumptuous banquet*).
- **Head**: the word that determines the syntactic type of the chunk: *the black cat* → **NP**.
- **Subject**: the person/thing *doing* or *being*, usually a noun phrase (**NP**): *the cat is black*.
- **Predicate**: the remainder of the sentence tells us what the subject does: *the cat sits on the mat*.
- **Clause**: subject + predicate.
- **Argument**: a chunk that is related to a verb in a clause, i.e. subject and object.
- **Object**: the person/thing affected by the action: *the cat eats fish*. Poor fish.
- **Preposition**: temporal, spatial or logical relationship: *the cat sits on the mat*.
- **Copula**: a word used to link subject and predicate, typically the verb *to be*.
- **Lemma**: canonical form of a word: *run*, *runs*, *running* are part of a lexeme, *run* is the lemma.
- **POS**: part-of-speech, the syntactic role of a word or phrase in the sentence, e.g. adjective = **JJ**.

1.4 Acknowledgements

This version of MBSP has been developed by the computational linguistics group of CLiPS (Computational Linguistics & Psycholinguistics, department of Linguistics, University of Antwerp, Belgium) on the basis of earlier versions developed at the University of Antwerp and Tilburg University.

CONTRIBUTING AUTHORS

- Walter Daelemans
- Jakub Zavrel
- Sabine Buchholz
- Jorn Veenstra
- Antal van den Bosch
- Ko van der Sloot
- Bertjan Busser
- Erik F. Tjong Kim Sang
- Jo Meyhi
- Vincent Van Asch
- Tom De Smedt

2 Installation

MBSP is a Python module. On Mac OS X, the Python programming language is already installed. On other systems you need to download and install it yourself: http://python.org/download. MBSP works with Python 2.5, but support for processes is better in version 2.6. It should also work with version 2.4. MBSP is bundled with three required dependencies written in C/C++ (TiMBL 6.1.5, MBT 3.1.3 and MBLEM). Binaries have been precompiled for Mac OS X 10.5, but these may not work on your machine. In that case you need to compile binaries manually from the source code.

2.1 Compiling with setup.py

The module comes with a `setup.py` script that compiles the C/C++ binaries automatically. If this works for you, you're in luck – no manual compilation is necessary. Also, if you run `setup.py` with the `install` argument, it will first compile the binaries and then install a copy of MBSP in Python's `/site-packages` folder so that the module is available in any Python script.

```
> cd MBSP
> python setup.py install
```

2.2 Compiling from source

You'll need a gcc compiler (http://gcc.gnu.org). On Windows you need cygwin (http://cygwin.com). In the cygwin installer (`setup.exe`), be sure to select the "`devel`" packages for installation.

BUILDING MBLEM

- Go to the `MBSP/mblem` folder.
- Delete all files with a "`.o`" extension + the current executable binary `mblem_english_bmt`.
- From the command line, do `make` in the `MBSP/mblem` folder:

```
> cd MBSP/mblem
> make
```

BUILDING TIMBL

- Go to the `MBSP/timbl` folder.
- Uncompress the source code from the `timbl-6.1.5.tar` archive.
- From the command line, do `configure` and `make` in the `MBSP/timbl/timbl-6.1.5` folder:

```
> cd MBSP/timbl/timbl-6.1.5
> ./configure --enable-shared=no --enable-static=no --prefix=[FOLDER]
> make install
```

- The `Timbl` executable will be in `[FOLDER]/bin` → copy it to `MBSP/timbl`.
- Now build MBT in the same `[FOLDER]` location:

BUILDING MBT

- Go to the `MBSP/mbt` folder.
- Uncompress the source code from the `mbt-3.1.3.tar` archive.
- From the command line, do `configure` and `make` in the `MBSP/mbt/mbt-3.1.3` folder:

```
> cd MBSP/mbt/mbt-3.1.3
> ./configure  --enable-shared=no --enable-static=no --prefix=[FOLDER]
> make install
```

- The `Mbt` executable will be `in [FOLDER]/bin` → copy it to `MBSP/mbt`.
- Delete the build `[FOLDER]`, it is no longer needed. That's it!

MULTITHREADING

MBSP can be configured to work with multithreading, which can increase performance by 25% up to 200%. `MBSP.config.threading` needs to be set to `True`. You also need to build the newer TiMBL 6.3+, MBT 3.2+ and TimblServer 2+ (or up) from source. Installation is mostly the same:

- TimblServer can be compiled in the same way as MBT.
- Put the `TimblServer` executable in `MBSP/timbl`, remove any `Timbl` executable.

Older systems may complain that `pkg-config` is outdated. In this case, before building Timbl, compile pkg-config 0.25+ (http://pkgconfig.freedesktop.org) from source with:

```
> sudo ./configure
> sudo make
> sudo make install
```

2.3 Module folder location

To be able to `import MBSP` in your scripts, Python needs to know where the module is located. There are three basic ways to accomplish this:

- Put the `MBSP` folder in the same folder as your script.
- Put the `MBSP` folder in the standard location for modules so it is available to *all* scripts.
  The standard location depends on your operating system, for example:
  `/Library/Python/2.5/site-packages/` on Mac,
  `/usr/lib/python2.5/site-packages/` on Unix,
  `c:\python25\Lib\site-packages\` on Windows.
- Add the location of MBSP to the `sys.path` list in your script, before importing it:

```
>>> MODULE = '/users/tom/desktop/MBSP'
>>> import sys; if MODULE not in sys.path: sys.path.append(MODULE)
>>> import MBSP
```

3 Parser

MBSP uses a client-server architecture. This way, the corpus data is loaded only once (during server startup) and stays available while the servers sleep in the background. Before tagging jobs can be sent to the parser, the servers have to be started. By default, this will happen automatically when you import MBSP in your script. Otherwise, the `start()` command starts the four servers (named `CHUNK`, `LEMMA`, `RELATION` and `PREPOSITION`) manually. The `started()` command yields True if a given server has been started. The `stop()` command will stop the servers.

```
MBSP.start(timeout=60)
MBSP.started(name=ALL) # CHUNK | LEMMA | RELATION | PREPOSITION
MBSP.stop()
```

The `parse()` command takes a string of sentences and returns a tagged Unicode string.
Sentences in the output are separated by newline characters (`\n`).

```
MBSP.parse(string,
     tokenize = True,
         tags = True,
       chunks = True,
    relations = True,
      anchors = True,
      lemmata = True,
     encoding = 'utf-8')
```

For example:

```
>>> print MBSP.parse('I ate pizza with a fork.')

I/PRP/I-NP/O/NP-SBJ-1/O/i
ate/VBD/I-VP/O/VP-1/A1/eat
pizza/NN/I-NP/O/NP-OBJ-1/O/pizza
with/IN/I-PP/B-PNP/O/P1/with
a/DT/I-NP/I-PNP/O/P1/a
fork/NN/I-NP/I-PNP/O/P1/fork ././O/O/O/O/.
```

Each token (i.e. tagged word) in a sentence has a number of annotations: `tags=True` includes the word part-of-speech tag, `chunks` the chunk tag + **PNP** tag (prepositional noun phrase), `relations` the chunk relation tag, `anchors` the **PNP** anchor tag. With `tokenize` set to `False`, no tokenization is carried out (so the input string is expected to be tokenized). The `encoding` parameter defines the character encoding of the input string, "utf-8" is fine in most cases.

3.1 Parser tags

Let's examine the word *ate* and the tags assigned by the parser in the previous example:

WORD	PART-OF-SPEECH	CHUNK	PNP	RELATION	ANCHOR	LEMMA
ate	**VBD**	**I-VP**	**O**	**VP-1**	**A1**	eat

The word's part-of-speech tag is **VBD**, which means that it is a verb in the past tense. The word occurs in a **VP** chunk, a verb phrase. It is not part of a prepositional noun phrase. Its relation tag is **VP-1**, which means it is linked to the words tagged **as NP-SBJ-1** (*I*, sentence subject) and **NP-OBJ-1** (*pizza*, sentence object). Its anchor tag is **A1**, meaning it is the anchor of the prepositional noun phrase **P1**, *with a fork*. How did I eat pizza? → with a fork. The base form or lemma of *ate* is *eat*.

- Common part-of-speech tags include **NN** (noun), **JJ** (adjective) and **VB** (verb).
- Common chunk tags include **NP** (noun phrase) and **VP** (verb phrase).
- Common relations include **SBJ** (subject) and **OBJ** (object).

The Appendix gives an overview of all the possible tags generated by the parser.
A description and an example can also be acquired with the `taginfo()` command:

```
>>> description, example = MBSP.taginfo('NN')
>>> print description, example

('noun, singular or mass', 'tiger, chair, laughter')
```

3.2 Parser shortcuts

Below is a set of concise commands that internally call `parse()` with the required parameters.

- The `tag()` command returns a string annotated with part-of-speech tags.
- The `chunk()` command returns a string annotated with part-of-speech, chunk and **PNP** tags.
- The `lemma()` command takes a single word and returns its lemma.
- The `nouns()`, `verbs()` or `adjectives()` command returns a list of nouns, verbs or adjectives retrieved from the given string.

```
MBSP.tokenize(string)
MBSP.tag(string, tokenize=True, lemmata=False)
MBSP.chunk(string, tokenize=True, lemmata=False)
MBSP.lemma(word)
MBSP.nouns(string, lemmatize=False)
MBSP.verbs(string, lemmatize=False)
MBSP.adjectives(string, lemmatize=False)
```

Like `parse()`, each of the commands has an optional `encoding` parameter that is "utf-8" by default.

3.3 Parser output

The output of the `parse()` command is a string of sentences in which each token is annotated with the requested tags. The `pprint()` command (extra `p` is for pretty) gives a good overview of the tags:

```
>>> s = MBSP.parse('I eat pizza.')
>>> MBSP.pprint(s)

    WORD   TAG    CHUNK   ROLE   ID     PNP    ANCHOR   LEMMA
       I   PRP    NP      SBJ    1      -      -        i
     eat   VBP    VP      -      1      -      -        eat
   pizza   NN     NP      OBJ    1      -      -        pizza
       .   .      -       -      -      -      -        .
```

The output string is in fact a `TokenString` object that behaves as a Python unicode string, but with extra functionality. Most notably it has a `TokenString.split()` method that yields a `TokenList` object: a list of sentences, where each sentence is a list of tokens, in which each token is a list of tags. This is useful when you want to extend the parser and need to make some modifications to the output (see Section 11).

If you want to analyze the output (i.e. examine the relations between words and groups of words), it is more convenient to construct a parse tree from the output (see Section 7).

Section 3 summarized the basics you need to start running your own tagging jobs. In Sections 4-6 we discuss the tokenizer, lemmatizer and PP-attacher in more detail. For a full overview of how the tagger/chunker works, we refer to the textbook (*Memory-Based Language Processing*, Daelemans & van den Bosch, 2005). Section 7 demonstrates important functionality to traverse sentences as syntactic trees. Sections 8-12 offers in-depth technical information for those wishing to extend and/or integrate MBSP with other software.

4 Tokenizer

MBSP includes a regular expressions-based tokenizer that divides sequences of characters into words and sentences. Special care is given to punctuation marks. We need to guess which punctuation marks are part of a word (e.g. periods in abbreviations) and which mark word and sentence breaks. Once tokenized, the parser can then determine categories for words (tokens) in the string.

```
MBSP.tokenizer.split(string, tags=False, replace={}, ignore=[])
```

The `split()` command returns a list of Unicode strings where punctuation marks have been split from words as individual tokens. Each string in the list is a sentence. By default, all SGML-tags (i.e. anything wrapped in <...>) are stripped from the input string. The `replace` dictionary is used to map unicode quotes and ellipsis to standard quotes and periods. The `ignore` list defines ranges of words that require special attention (for example, abbreviations and URL's – see below).

For example:

```
>>> MBSP.tokenizer.split('The U.N. is considering banning "defamation of religion."
 The U.N. president said such a move would not limit free speech.')

[u'The U.N. is considering banning " defamation of religion . "',
 u'The U.N. president said such a move would not limit free speech .']
```

4.1 Tokenization process

In the simplest case, words are marked by spaces. A number of exceptions are then handled:

- **Handle missing space**: punctuation inside a word can indicate a missing space.
- **Handle punctuation**: punctuation marks at the head or tail of a word can indicate a sentence break, the start of a citation, the start of an explanation in parenthesis. In this case the punctuation is split from the word. Punctuation that is part of the word (e.g. *5p.m.*) needs to be kept intact.
- **Handle contractions**: apostrophes in contractions (*he's*) and possessives *(father's*) mark word boundaries. The suffix is split from the word.
- **Handle lists**: list item markers such as *1. 1)* * - *a. a)* at the start of a new line indicate the start of a new sentence, even if the previous sentence did not end with a period.
- **Handle hyphenation**: hyphens at the end of a line mark words that have been split across lines: *mar-* + *ket* becomes *market*, *Great-* + *Britain* becomes *Great-Britain*.
- **Handle sentence breaks**: periods, exclamation marks, question marks and ellipsis end a sentence if they are followed by a capitalized letter. Parenthesis and quotes can be part of the sentence even if the period precedes it.

4.2 Tokenizer word ranges

The tokenizer defines a `Range` class for matching sets of words: all abbreviations, all numeric strings, all hyperlinks, etc. The `Range` class is a list of known words enriched with regular expression patterns. In the example below, an abbreviation range is created that matches *U.K.*, *U.N.* and any single letter abbreviation (as in *T. De Smedt*).

```
>>> abbreviations = MBSP.tokenizer.Range(['U.K.', 'U.N.'])
>>> abbreviations.patterns.append(re.compile('^[A-Za-z]\.$'))
>>> print 'U.K.' in abbreviations, 'T.' in abbreviations

True True
```

Custom ranges can be passed to the `ignore` parameter of the `split()` command.
By default, `split()` will use a list of predefined ranges:

```
MBSP.tokenizer.ignore = [abbreviations, numeric, URI, entities, biomedical]
```

Predefined ranges:

- **Abbreviations**: the simple rule is that every point is a sentence break. This is 93.2% correct for Brown corpus. Fixing decimal points, single letter abbreviations, alternating letters and capital letter followed by consonants (*Dept.*) improves sentence break correctness to 97.7%. Additionally, the abbreviations range defines a list of other well-known abbreviations.
- **Numeric**: matches anything starting with a digit followed by a chain of digits and .,:/ separators. The range will also recognize units of measurement (length, mass, volume, time, epoch, temperature, storage capacity, data transfer rate, percentage): *US$100*, *1.2MB*, *31/12/2010*, ...
- **URI**: matches URL's, links and e-mail addresses.
- **Entities**: matches HTML and Unicode entities.
- **Biomedical**: guesses biomedical-specific words, such as *1',2'-trifenol*.

4.3 Tokenizer modes

The tokenizer can be configured in different modes. By default it uses Penn Treebank specifications for tokenization and uses special corrections for biomedical use.

```
MBSP.tokenizer.PENN_TREEBANK = True
```

```
MBSP.tokenizer.BIOMEDICAL = True
```

In Penn Treebank mode, only *%* is split from numbers and contractions are not substituted: *won't* becomes *wo n't* instead of *will not*. In biomedical mode, less magic is used when finding missing spaces so that *3(R),3a(S),6a(R)-bis-tetrahydrofuranylurethane* is not split at the comma or the parenthesis.

5 Lemmatizer

MBSP uses the MBLEM C module for word lemmatization. MBLEM (Memory-Based Lematization) transforms a word form into a canonical lexical form: *was → be*, *booking → book*.

```
>>> print MBSP.lemmatize('The cats were sleeping.', tokenize=True)

the cat be sleep .
```

The goal of lemmatization is to provide better (generalizable) lexical information that can be used by other components in the pipeline, and (possibly) also for indexing the processed text in a search engine. Lemmatization is not to be confused with *stemming*, in which frequent suffixes such as *-ing* and *-ed* are simply removed: *having* is stemmed to *hav*, while it is lemmatized to *have*.

5.1 Lemmatization process

Words can have more than one lemma. The lemmatizer's task is to find the most suitable:

- **Find lemmata**: determines for each word in a sentence which lemmata it could be mapped to. The word *saw* is lemmatized to the noun lemma *saw* and the verb lemma *see*. In case of producing more than one lemma, the lemmatizer is unable to determine which lemma is appropriate in the current sentence, as it does not use context.
- **Disambiguate**: consults the output of the part-of-speech tagger. If the tagger has identified saw as a past-tense verb form (**VBD**), MBLEM concludes that the appropriate lemma is see, and selects this one as the correct output. MBLEM therefore always generates a single lemma per word.

MBLEM is trained on the CELEX English lexical database, and will simply retrieve the lemmatizations of words that occur in the database. Words in the text that are not in CELEX (so-called "unknown" words, typically constituting 5% of the words in a text) are lemmatized by analogy to stored word lemmatizations.

Reference: Van den Bosch, A., & Daelemans, W. (1999). *Memory-based morphological analysis*, In: Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics, pp. 285-292.

6 PP-attacher

A shallow parsing approach sometimes has its shortcomings, an important one being that prepositional phrases, which contain important semantic information for interpreting events (e.g. *I sleep* vs. *I sleep under a bridge*), are left unattached. MBSP comes with a memory-based PP-attacher (MBPA) trained on sections 2 through 21 of the Penn Treebank II Wall Street Journal corpus (WSJ). It will link **PNP** chunks to other "anchor" chunks.

With the PP-attacher enabled, performance is slower but you gain a lot of useful information.

```
>>> s = MBSP.parse('I sleep under a bridge', anchors=True)
>>> s = MBSP.split(s)
>>> print s[0].pnp
>>> print s[0].pnp[0].anchor

[Chunk('under a bridge/PNP')]
sleep
```

6.1 PP-attachment process

A **PNP**-chunk can be attached to different candidate chunks. However, there is a semantic difference between *I eat pizza with olives* and *I eat pizza with olives*. Which one is correct? And which one is correct here: *I eat pizza with a fork* or *I eat pizza with a fork*?

- **Find prepositional noun phrases**: these are retrieved by a regular expression-like algorithm. All **PP** + **NP** sequences are considered to be **PNP**'s. Two exceptions: **PP** + **"** + **NP** (e.g. *in "modest amounts"*) and **PP** + **VBG** + **NP** (*in making paper*) are also considered **PNP**'s.
- **Classify**: the core of the PP-attacher is a memory-based classifier. Candidate anchors are the **NP**'s and **VP**'s of the sentence that are not part of the **PNP** itself. For example, *I eat a pizza with olives* induces three classification tasks (*I-with*, *eat-with*, *pizza-with*) in which the machine learner will have to decide if the pair suggests a true anchor or not – taking into account the distance to the candidate anchor and intermediary punctuation, the number of intermediary **NP**'s, and so on.
- **Heuristic decision making**: when the classifier identifies multiple anchor candidates, an extra step is taken to pick one unique anchor, using a baseline and entropy algorithm (details are summarized in the reference paper).

In 50% of the cases a noun phrase (**NP**) is chosen as anchor, in about 45% a verb phrase (**VP**).

Reference: Van Asch, V., & Daelemans, W. (2009). *Prepositional Phrase Attachment in Shallow Parsing*. In: Proceedings of the 7th International Conference on Recent Advances in Natural Language Processing (RANLP), pp. 12-17.

7 Parse trees

A parse tree stores a parsed string as a network of linked Python objects that can be traversed to analyze the sentences in the string. The output of the parser can be passed to the `split()` command, which produces a `Text` object. Essentially, a `Text` is a list of `Sentence` objects. Each `Sentence` consists of `Word` objects. `Word` objects are also grouped in `Chunk` objects, which are related to other `Chunk` objects in various ways.

```
MBSP.split(parsed_string, token=[WORD, POS, CHUNK, PNP, RELATION, ANCHOR, LEMMA])
```

We'll run the sentence "*The cat sat on the mat.*" through the parse tree:

```
>>> s = MBSP.parse('The cat sat on the mat.')
>>> s = MBSP.split(s)
>>> print repr(s)

[Sentence(
 'The/DT/B-NP/O/NP-SBJ-1/O/the
  cat/NN/I-NP/O/NP-SBJ-1/O/cat
  sat/VBD/B-VP/O/VP-1/A1/sit
  on/IN/B-PP/B-PNP/PP-CLR/P1/on
  the/DT/B-NP/I-PNP/NP-CLR/P1/the
  mat/NN/I-NP/I-PNP/NP-CLR/P1/mat
  ././O/O/O/O/.')]
```

```
>>> print s[0].chunks

[Chunk('The cat/NP-SBJ-1'),
 Chunk('sat/VP-1'),
 Chunk('on/PP-CLR'),
 Chunk('the mat/NP-CLR')]
```

7.1 Text

A `Text` is a list of `Sentence` objects.

```
text = MBSP.Text(parsed_string, token=[WORD, POS, CHUNK, PNP, REL, ANCHOR, LEMMA])
```

```
text = MBSP.Text.from_xml(xml)
```

```
text.string                 # 'The cat sat on the mat .'
text.sentences              # [Sentence('The cat sat on the mat .')]
text.append(sentence)
text.copy()
text.xml
```

Since `Text` behaves as a Python list it is easy to traverse all the contained sentences:

```
for sentence in text:
    print sentence
```

7.2 Sentence

A `Sentence` is a list of `Word` objects with attributes that organize words in `Chunk` objects.

```
sentence = MBSP.Sentence(string='', token=[WORD,POS,CHUNK,PNP,REL,ANCHOR,LEMMA])
```

```
sentence = MBSP.Sentence.from_xml(xml)
```

```
sentence.parent              # Slices refer to the sentence they are part of.
sentence.id                  # Unique for each sentence.
sentence.start               # 0
sentence.stop                # 6
sentence.token               # [WORD, POS, CHUNK, PNP, REL, ANCHOR, LEMMA]
```

```
sentence.string              # 'The cat sat on the mat .'
sentence.words               # [Word('The/DT'), Word('cat/NN'), ... ]
sentence.lemmata             # [u'the', u'cat', u'sit', u'on', u'the', u'mat', u'.']
sentence.tagged              # [(u'The', u'DT'), (u'cat', u'NN'), ...]
sentence.parts_of_speech     # [u'DT', u'NN', u'VBD', u'IN', u'DT', u'NN', u'.'
```

```
sentence.chunks              # [Chunk('The cat/NP-SBJ-1'), Chunk('sat/VP-1'), ... ]
sentence.subjects            # [Chunk('The cat/NP-SBJ-1')]
sentence.objects             # []
sentence.verbs               # [Chunk('sat/VP-1')]
sentence.relations           # {'SBJ': {1: Chunk('the cat/NP-SBJ-1')},
                             #    'VP': {1: Chunk('wants/VP-1')},
                             #   'OBJ': {}}
```

```
sentence.pnp                 # [Chunk('on the mat/PNP')]
sentence.anchors             # [Chunk('sat/VP-1')]
```

```
sentence.constituents(pnp=False)
```

```
sentence.get(index, tag=LEMMA)
sentence.loop(tag1, tag2, ...)
sentence.indexof(value, tag=WORD)
```

```
sentence.slice(start, stop)
sentence.copy()
```

```
sentence.xml
sentence.nltk_tree()
```

- `Sentence.constituents()` returns an in-order list of `Word` and `Chunk` objects.
  With `pnp=True`, also contains `PNPChunk` objects whenever possible.
- `Sentence.get()` returns the requested tag of the word at the given index.
  The tag can be `WORD`, `LEMMA`, `POS`, `CHUNK`, `PNP`, `RELATION`, `ROLE`, `ANCHOR` or a custom word tag.
- `Sentence.loop()` is an iterator over a list of tuples containing the requested tags for each word.
- `Sentence.indexof()` returns the indices of words for which the given `tag` equals `value`. For example, `Sentence.indexof("NN*", tag=POS)` returns a list of indices of the words whose part-of-speech is **NN**, **NNS**, **NNP** or **NNPS**.
- `Sentence.slice()` returns a new, partial sentence starting with the word at index start and containing all the words up to (before) index stop.

CREATING A SENTENCE

Normally, a sentence is constructed from the output of the `parse()` command. Since this output is a `TokenString` that stores the order in which tags appear in a token, `Sentence` can figure out how to construct a parse tree by itself. If you want to construct sentences from a different source, you need to specify the `token` parameter in the constructor, or use `Sentence.append()` to add words manually. If you have tokens in a slash-formatted string like MBSP (e.g. "cats/**NNS**/**I-NP**/**O**/**O**/**O**/cat") you can use `Sentence.parse_token()`, which returns the arguments for `Sentence.append()`.

```
sentence.append(word,
       lemma = None,
        type = None,
       chunk = None,
        role = None,
    relation = None,
         pnp = None,
      anchor = None,
         iob = None,
      custom = {})
```

```
sentence.parse_token(token, tags=[WORD, POS, CHUNK, PNP, REL, ANCHOR, LEMMA])
```

For example:

```
>>> s = MBSP.Sentence('cats/NNS', token=[MBSP.WORD, MBSP.POS])
>>> print s.words

[Word('cats/NNS')]
```

```
>>> s = MBSP.Sentence()
>>> s.append(word='cats', lemma='cat', type='NNS', chunk='NP')
>>> print s.words

[Word('cats/NNS')]
```

```
>>> s = MBSP.Sentence()
>>> s.append(*s.parse_token('cats/NNS/I-NP/O/O/O/cat'))
>>> print s.words

[Word('cats/NNS')]
```

7.3 Sentence chunks

A `Chunk` is a list of `Word` objects that belong together.
`Chunks` can be part of a `PNPChunk`, which starts with a **PP** chunk followed by **NP** chunks.

```
chunk = MBSP.Chunk(sentence, words=[], type=None, role=None, relation=None)

chunk.sentence                # Sentence object (e.g. 'The cat sat on the mat .')
chunk.start                   # 0
chunk.stop                    # 2

chunk.string                  # 'The cat'
chunk.words                   # [Word('The/DT'), Word('cat/NN')]
chunk.lemmata                 # ['the', 'cat']
chunk.tagged                  # [(u'The', u'DT'), (u'cat', u'NN')]
chunk.head                    # Word('cat/NN')

chunk.type                    # 'NP'
chunk.role                    # 'SBJ'
chunk.relation                # 1
chunk.relations               # [(1, u'SBJ')]
chunk.related                 # [Chunk('sat/VP-1')]
chunk.subject                 # None
chunk.object                  # None
chunk.verb                    # Chunk('sat/VP-1')
chunk.modifiers               # []
chunk.conjunctions            # []

chunk.pnp                     # None
chunk.anchor                  # None
chunk.attachments             # [Chunk('on the mat/PNP')]

chunk.append(word)

chunk.previous(type=None)     # None
chunk.next(type=None)         # Chunk('sat/VP-1')
chunk.nearest(type="VP")      # Chunk('sat/VP-1')
```

- `Chunk.head` yields the last (i.e. principal) `Word` in the chunk.
- `Chunk.relations` contains all relations the chunk is involved in.
  Some chunks (about 15%) have multiple relations, for example functioning as both **SBJ** and **OBJ**, or being the **OBJ** of multiple **VP** chunks.
- For **VP** chunks, `Chunk.modifiers` is a list of nearby adjectives and adverbs with no relations.
  For example: in *“the cat really wants out*”, *really* and *out* are **ADVP** with no relations.
  The parse tree will assume that they have something to do with the **VP** wants.
  What does the cat want? → out.
  How badly does the cat want out? → really.
- `Chunk.conjunctions` is a list of chunks linked by ‘and’ or ‘or’ to this chunk.
  For example: '*going up and down*', the *up* chunk has conjunctions: `[(Chunk('down'), AND)]`
- `Chunk.pnp` is an alias for the parent `Chunk.prepositional_noun_phrase`.
- `Chunk.attachments` contains related prepositional noun phrases.
- `Chunk.anchor` references the chunk that is the anchor of this **PNP**.

7.4 Prepositional noun phrases

`PNPChunk` is a subclass of `Chunk`, it has the same attributes and methods.
It groups multiple chunks in a prepositional noun phrase (**PNP**).

```
pnp = MBSP.PNPChunk(sentence, words=[], type=None, role=None, relation=None)

pnp.string                 # 'on the mat'
pnp.chunks                 # [Chunk('on/PP-CLR'), Chunk('the mat/NP-CLR')]
pnp.preposition            # Chunk('on/PP-CLR')
pnp.anchor                 # Chunk('sat/VP-1')

pnp.guess_anchor()         # Returns the nearest VP chunk.
```

Words and chunks that are part of a **PNP** will have their `Word.pnp` and `Chunk.pnp` attribute set. All the prepositional noun phrases in a sentence can be retrieved with `Sentence.pnp`.

7.5 Sentence words

A `Sentence` is made up of `Word` objects, which are also grouped in `Chunk` objects:

```
word = MBSP.Word(sentence, string, lemma=None, type=None, index=0)

word.sentence              # Sentence object (e.g. 'The cat sat on the mat .')
word.index                 # 2
word.string                # 'sat'
word.lemma                 # 'sit'
word.type                  # 'VBD'
word.chunk                 # Chunk('sat/VP-1')
word.pnp                   # None

word.custom_tags           # User-defined tags, e.g. {SENTIMENT: 'lazy'}

word.tags()                # ['sat', 'VBD', 'B-VP', 'O', 'VP-1', 'O', 'sit']
```

- `Word.type` is an alias for `Word.part_of_speech`.
- `Word.pnp` is an alias for `Word.prepositional_noun_phrase`.
- `Word.custom_tags` is a dictionary of additional, user-defined tags that can occur when the parser has been extended. If the word has (for example) an additional sentiment tag, it is also available as `Word.sentiment`.

`Word.tags()` returns a list of token tags as they appear in the output of the parser. The order of tags is determined by the `Sentence.token` attribute.

8 Clients and servers

MBSP uses a client-server architecture so that the data only has to be loaded once. From then on it is available as a server that can be contacted with the TCP protocol. MBSP has a `Client` and a `Server` class that communicate using the Python socket module. `Server` is a wrapper around `subprocess.Popen` and starts TiMBL or MBT as a background process. MBSP starts four servers:

NAME	ADDRESS	PROCESS	DESCRIPTION
`MBSP.CHUNK`	localhost:6061	MBT	Server for part-of-speech and chunk tags.
`MBSP.LEMMA`	localhost:6062	TiMBL	Server used by MBLEM for word lemmata.
`MBSP.PREPOSITION`	localhost:6063	TiMBL	Server for **PNP** anchors.
`MBSP.RELATION`	localhost:6064	TiMBL	Server for verb-argument chunk relations.

- **TiMBL** (http://ilk.uvt.nl/timbl): memory-based learning is an elegantly simple and robust machine-learning method applicable to a wide range of tasks in Natural Language Processing.
- **MBT** (http://ilk.uvt.nl/mbt): a memory-based tagger-generator and tagger in one. MBT can, for instance, be used to generate part-of-speech taggers or chunkers for natural language processing.

8.1 Client

The `Client` base class can be used to contact a TiMBL or MBT server running at a given host address, at a given TCP port. `Client` is then subclassed with `Timbl` and `Mbt` classes.

```
client = MBSP.Client(
        host = LOCALHOST,
        port = 6060,
        name = None,
         log = False,
     request = lambda v:v.strip()+'\n',
    response = lambda v:v)
```

```
client.name
client.host
client.port
client.connected # True after Client.connect() is called.
```

```
client.connect()
client.send(request, timeout=None)
client.disconnect()
```

- `Client.connect()` raises a `ServerConnectionError` if the server can't be contacted.
- `ServerConnectionError.code` can contain additional information, such as `CONNECTION_RESET_BY_PEER`, `CONNECTION_REFUSED` or `BROKEN_PIPE`.
- `Client.send()` prepares the request, sends it to the server and returns the response. `Client.send()` raises a `ClientDisconnectedError` if `Client.connect()` is not called first. If `timeout` is given and no response returns in the given time, `ClientTimeoutError` is raised.
- Make sure to clean up with `Client.disconnect()` when the client is no longer needed.

REQUEST AND RESPONSE FORMATTERS

Note the `request` and `response` parameters in the `Client` constructor. These are formatter functions that will be called by `Client.send()` to 1) prepare the data before sending it to the server and 2) to clean up the server response before returning it.

For example, if we send a raw request to the chunk server:

```
>>> chunker = MBSP.Client(port=6061)
>>> print chunker.send('cat')

cat/NN/I-NP <utt>
```

The MBT server's response ends with a delimiter which we may want to clean from the output. This can be achieved by using a response formatter:

```
>>> chunker = MBSP.Client(port=6061, response=lambda v: v.rstrip(' <utt>\n'))
>>> print chunker.send('cat')

cat/NN/I-NP
```

8.2 TiMBL and MBT client

MBSP has `Timbl` and `Mbt` subclasses of `Client` with the right request and response formatters.

```
timbl = MBSP.Timbl(host=LOCALHOST, port=6060, name=None, log=False, verbosity=[])
```

```
mbt = MBSP.Mbt(host=LOCALHOST, port=6060, name=None, log=False)
```

TiMBL's output depends on the way the server *verbosity* (+v) is configured. It typically yields a category string: "`CATEGORY {NP-SBJ}\n`". Depending on the verbosity it can also yield different output. For example, verbosity option `+v di+db` yields a category-distance-distribution string:
`"CATEGORY {VP} DISTRIBUTION { VP 3.1150, n-VP 2.5823 } DISTANCE {2.0206}\n".`

The TiMBL client has an additional `verbosity` parameter which can be used to format the response:

```
>>> timbl = MBSP.Timbl(port=6064, verbosity=['di','db'])
>>> print timbl.send('0 0 2 - eat VBP pizza NN with fork NN 1 0')

['VP', 2.0206599999999999, {'VP': 3.1150600000000002, 'n-VP': 2.5823200000000002}]
```

The server at port 6064 is the preposition server, which loads with option `+v di+db`, so responses include distance and distribution metrics. The TiMBL client in this example sets the right verbosity options so that distance is correctly parsed as a float, and distribution as a dictionary.

See the TiMBL manual for all verbosity options:
http://ilk.uvt.nl/downloads/pub/papers/Timbl_6.3_Manual.pdf

8.3 Client batch requests

Since the servers are multithreaded, multiple request can be sent at the same time to speed up the lookup process. MBSP's client module has a `client.batch()` command to achieve this. It takes a list of requests (i.e. instances) and a "client definition". This definition is a tuple with the necessary parameters that allows the `batch()` command to generate threaded `Client` objects. The `client.define()` command can be used to generate the tuple.

```
MBSP.client.batch(instances, client, timeout=None, retries=1)
```

```
MBSP.client.define(client, host=LOCALHOST, port=6060, name=None, log=False)
```

For example:

```
>>> from MBSP.client import batch, define, Mbt
>>> print batch(['cat', 'sat', 'mat'], client=define(Mbt, port=6061))

['cat/NN/I-NP', 'sat/VBN/I-VP', 'mat/NN/I-NP']
```

8.4 Client request log

If `config.log=True`, all client requests and server responses are logged. This is useful for evaluation. Also, when logs are enabled, a cached request can be reused without having to contact the server. The performance hit is negligible however – except for very repetitive texts. The log is a dictionary indexed by server name (`CHUNK`, `LEMMA`, `RELATION` and `PREPOSITION`). Each server log in the dictionary is an ordered dictionary that stores the last 1000 requests.

```
MBSP.client.log
```

For example:

```
>>> MBSP.config.log = True
>>> MBSP.parse('I eat pizza with a fork.')
>>> print MBSP.client.log[MBSP.RELATION]

{'-1 0 0 eat VBP - - - - - - - I PRP NP VBP VP ?': 'CATEGORY {NP-SBJ}\n',
  '1 0 0 eat VBP - - - eat VBP VP - pizza NN NP NN PNP ?': 'CATEGORY {NP-OBJ}\n',
  '2 0 0 eat VBP eat VBP VP pizza NN NP - fork NN PNP - - ?': 'CATEGORY {-}\n'}
```

Three requests were sent to the relation server to find the sentence **SBJ** and **OBJ**. More complex sentences require more lookup requests and more complex request with more instance features require more time.

8.5 Server

MBSP has a `Server` class that is used to start TiMBL or MBT as a background process. A server can be queried by creating a client. Typically, servers run locally on your own machine (e.g. at "localhost") but they can also be configured to work with an IP-address over a network. You can also create new servers from scratch and work with their responses in Python code.

```
server = MBSP.Server(
        name,
        host = LOCALHOST,
        port = 6060,
     process = TIMBL,
        ping = None,
    features = {})
```

```
server.name                 # LEMMA
server.host                 # LOCALHOST
server.port                 # 6062
server.process              # TIMBL
server.features             # {'-f': '/models/em.data', '-m': 'M', '-w': 2, '-k-: 5}
server.ping                 # ('c = = = = = = = = = = = = = = = = = = = = B ?\n',
                            #  'CATEGORY {ABB-X}\n')
```

```
server.program              # The shell command used to start the process.
server.pid                  # Process id, needed when stop() is called.
```

```
server.started              # True when ping request yields ping response.
```

```
server.start(timeout=60)
server.stop()
```

- `Server.features` specifies the training data file and various options (see the TiMBL manual).
- `Server.start()` raises a `ServerTimeoutError` when the server doesn't start in the given time. `Server.start()` will continually check `Server.started` until it returns `True`.
- `Server.started` determines if the server is up and running by sending a ping request.
- `Server.ping` is a tuple with a sample request and its desired (e.g. correct) response.
  If it is `None`, Server.started just sends "`x ?`" and accepts whatever response.
  Sending a sample request and validating the response is the only safe way to figure out you are addressing the right server.

STOPPING A SERVER

When a server is stopped, the background process is killed. The `Server.pid` process id is stored as a temporary file so that MBSP can retrieve which servers are up and running across Python sessions – you don't need to stop and restart a server each time you run a new Python script. If the process id is lost (for example, the system did a cleanup of temporary files), the server background process will have to be killed manually. The server module has a `force_quit()` that terminates all running TiMBL and MBT processes:

```
MBSP.server.force_quit(processes=(TIMBL, MBT))
```

This only works on Unix-systems though. Otherwise, the TiMBL and MBT processes will have to be terminated manually from the Windows Task Manager or Mac OS X's Activity Monitor. MBSP can also be configured to automatically stop its servers when Python exits.

9 Configuration

A number of settings in config.py can be adjusted. Most notably, IP addresses for the servers (`CHUNK`, `LEMMA`, `RELATION`, `PREPOSITION`) can be defined in the `config.hosts` list, so MBSP can be configured to work over a network.

By default, the servers will start automatically when MBSP is imported. Servers are always started at localhost, i.e. the `config.hosts` list contains the addresses needed for the clients. So if your MBSP is set up to contact servers over a network, `config.autostart` should be set to `False` so that no servers are started on your own machine. Servers can also be configured to stop automatically when Python exits, by setting `config.autostop` to `True`.

The `config.log` option controls whether client requests and the servers' responses are logged in the `MBSP.client.log` dictionary.

9.1 config.py options

OPTION	DEFAULT	DESCRIPTION
`MODULE`	`os.path.dirname(os.path.abspath(__file__))`	MBSP folder path.
`LOCALHOST`	`'localhost'`	Localhost IP address.
`hosts`	`[LOCALHOST, LOCALHOST, LOCALHOST, LOCALHOST]`	Server IP addresses.
`ports`	`[6061, 6062, 6063, 6064]`	Server ports.
`autostart`	`True`	Auto start servers?
`autostop`	`False`	Auto stop servers?
`log`	`False`	Log server requests?
`verbose`	`True`	Server startup message?

`paths`	`{'timbl': MODULE+'/timbl/Timbl'),`
	`'mbt': MODULE+'/mbt/Mbt'),`
	`'mblem': MODULE+'/mblem/mblem_english_bmt'),`
	`'models': MODULE+'/models')}`
		Binaries, training data.

`encoding`	`'utf-8'`	Character encoding.

10 Command-line interface

The parser can be run from the command-line by invoking `mbsp.py` inside the MBSP library folder. The parsed output is printed in the terminal window. Note that the servers will not be started automatically, so you have to start them manually:

```
> cd MBSP
> python mbsp.py start
> python mbsp.py parse -f camelot.txt
> python mbsp.py parse -s "It's only a model."
> python mbsp.py parse xml -s 'It is a silly place.'
> python mbsp.py stop
```

10.1 Command-line options

If no options are given a full parse is executed (i.e. tokenization, tagging, chunking, relations, prepositions and lemmata). Otherwise, you need to explicitly list every required option:

`-O`	`--tokenize`	Tokenize the input.
`-T`	`--tags`	Parse part-of-speech tags.
`-C`	`--chunks`	Parse chunks and PNP tags.
`-R`	`--relations`	Find verb/predicate relations.
`-A`	`--anchors`	Find PP-attachments.
`-L`	`--lemmata`	Find word lemmata.
`-f`	`--file`	Input filename.
`-s`	`--string`	Input string.
`-e`	`--encoding`	Specify character encoding (utf-8 by default).
`-v`	`--version`	Current version of MBSP.

Short options can be concatenated, e.g.: `python mbsp.py parse -OTL -f camelot.txt`

11 Extending MBSP

11.1 Event framework

MBSP has a built-in event framework that offers a convenient way to customize the module. For example, each completed step in the parser fires an event. A user-defined event handler can be assigned to an event, instead of messing around in the source code. Server event handlers take a `Server` object as input. Events are fired when a server is registered in a Servers group (see below), or when a server is successfully started or stopped.

```
def event_handler(server):
    return None
```

```
MBSP.events.server.on_register               = event_handler
MBSP.events.server.on_start                  = event_handler
MBSP.events.server.on_stop                   = event_handler
```

Parser event handlers take a `TokenString` object (see below) as input and return the modified `TokenString`. Events are fired when each step in the parsing process is completed (for example, tokenization is done).

```
def event_handler(tokenstring):
    return tokenstring
```

```
MBSP.events.parser.on_tokenize               = event_handler
MBSP.events.parser.on_parse_tags_and_chunks = event_handler
MBSP.events.parser.on_parse_prepositions     = event_handler
MBSP.events.parser.on_parse_relations        = event_handler
MBSP.events.parser.on_parse_pp_attachments   = event_handler
MBSP.events.parser.on_lemmatize              = event_handler
```

11.2 Customizing the parser

Suppose you need to correct some part-of-speech tags. Since the part-of-speech tag influences the relation and anchor tag annotation, you'd need to do this somewhere halfway down the parsing process. This is possible by injecting an event handler – without modifying the source code and thus keeping it easy to install package updates:

```
>>> def my_tagger_chunker(tokenstring):
>>>     T = tokenstring.split()
>>>     i = T.tags.index(MBSP.PART_OF_SPEECH)
>>>     for sentence in T:
>>>         for token in sentence:
>>>             print token[i] # modify POS-tag here
>>>     return T.join()
>>>
>>> MBSP.events.parser.on_parse_tags_and_chunks = my_tagger_chunker
```

Since the output of the event handler is then processed further, it needs to maintain the order of tags and it can't delete tags (they are still needed by the parser to determine other tags).

11.3 Customizing the parser output

The output of the `parse()` command is a string of sentences in which each token has been annotated with the requested tags. This string is in fact a `TokenString` object that behaves as a Python string, but with extra functionality. Most notably, it has a `TokenString.split()` method that yields a `TokenList` object: a list of sentences, where each sentence is a list of tokens, in which each token is a list of tags.

```
tokenstring = MBSP.TokenString(string, tags=[WORD])
```

```
tokenstring.tags              # [WORD, POS, CHUNK, PNP, RELATION, ANCHOR, LEMMA]
tokenstring.split(sep=TOKENS) # Yields a TokenList.
tokenstring.copy()
```

```
tokenlist = MBSP.TokenList(sentences, tags)
```

```
tokenlist.tags
tokenlist.tags.insert(i, tag, values=None)
tokenlist.tags.append(tag, values=None)
tokenlist.tags.remove(tag)
tokenlist.tags.pop(i)
tokenlist.tags.swap(tag1, tag2)
```

```
tokenlist.join()              # Yields a TokenString.
tokenlist.copy()
tokenlist.reduce(tags=[])
tokenlist.filter(
        word = None,
         tag = None,
       chunk = None,
    relation = None,
      anchor = None,
       lemma = None)
```

The `TokenString` facilitates the conversion from a tagged string to a list – where tokens can then be manipulated. The `TokenList.tags` list not only stores the order of tags in each token, it also keeps the tokens in synch. If you delete an item from `TokenList.tags`, all tokens in each sentence are automatically modified as well. In the same way, `TokenList.tags.append()` has an additional `values` parameter where you can enter the values of the new tag for each token in each sentence (by default, **O**).

For example:

```
>>> s = MBSP.parse('I eat pizza')
>>> s = s.split()
>>> s.tags.append('semantic', values=[['person', 'action', 'food']])
>>> s = s.join()
>>> print s

I/PRP/I-NP/O/NP-SBJ-1/O/i/person eat/VBP/I-VP/O/VP-1/O/eat/action
pizza/NN/I-NP/O/NP-OBJ-1/O/pizza/food
```

When you create a parse tree from this output, the *semantic* tag will end up in `Word.custom_tags`.

To insert a tag from one `TokenString` into another you can use `TokenString1.tags.pop()`. It extracts (and removes) the tag value from each token. You can use this return value for the values parameter in the other `TokenString2.tags.append()`.

`TokenList.reduce()` returns a copy with the given tags removed from all tokens.
`TokenList.filter()` returns a list of all tokens that match the given criteria. Wildcards can be used at the head, tail or middle of a constraint:

```
>>> s = MBSP.parse('I eat pizzas with a fork.')
>>> print s.split().filter(tag="NN*")

[[u'pizzas', u'NNS', u'I-NP', u'O', u'NP-OBJ-1', u'O', u'pizza'],
 [u'fork', u'NN', u'I-NP', u'I-PNP', u'O', u'P1', u'fork']]
```

11.4 Customizing the servers

MBSP's servers are grouped in a `Servers` object: a Python list with some additional functionality:

- `Servers` can automatically start and stop a server (i.e. `Server` object) that is added to the group.
- `Servers` will check that all servers in the group use a different port and a different name.
- When a server is added, its `Server.group` attribute will refer to the `Servers` object.
- When a server is added, the `on_register` event will be fired.

```
servers = MBSP.Servers(start=False, stop=False)

servers.append(server)

servers.[server_name]       # Retrieve a server by name.

servers.started             # True when all servers in the group have started.
servers.start(timeout=60)
servers.stop()
```

The correct way to install a custom server is to add it to the module's `active_servers` group to ensure that it starts and stops together with the other servers (e.g. when `MBSP.stop()` is called). Assume we have a tagger trained for biomedical use that overrides the default part-of-speech tags:

```
>>> import MBSP
>>> MBSP.active_servers.append(
>>>    MBSP.Server(
>>>            name = 'biomedical_pos',
>>>            port = 6065,
>>>         process = MBSP.MBT,
>>>        features = {'-s' : 'biopos/biopos.settings'}))
```

We can contact the server with an event, after MBSP is done tagging and chunking:

```
>>> def update_pos(tokenstring):
>>>     client = MBSP.Mbt(port=6065)
>>>     s1 = tokenstring.split()
>>>     s2 = s1.reduce([MBSP.WORD])
>>>     s2 = MBSP.TokenString(client.send(s2.join()), tags=[MBSP.WORD, MBSP.POS])
>>>     s2 = s2.split()
>>>     s2.tags.append(MBSP.CHUNK, values=s1.tags.pop(s1.tags.index(MBSP.CHUNK)))
>>>     client.disconnect()
>>>     return s2.join()
>>>
>>> MBSP.events.parser.on_parse_tags_and_chunks = update_pos
```

We have installed a new server and then contact it in the `update_pos()` event, right after MBSP has finished tagging and chunking with the default servers. The event extracts the original words in the sentence and feeds them to our biomedical MBT server. We append the original chunk tags to the updated output and pass it back to MBSP. MBSP will continue to look for relations and prepositions with the updated tags.

Replacing a default server before it starts

Assume you want to *replace* a default server with one of your own. If MBSP's servers are configured to start automatically, there is no way to stop them from loading before you can replace them. The way to do this is to disable MBSP's `autostart` feature, add replacement servers to `active_servers`, and then implement your own autostart in your own script.

12 Exporting to XML, NLTK, GraphViz

12.1 Exporting to XML

With the `xml()` command the output of the `parse()` command can be dumped as an XML string:

```
>>> s = MBSP.parse('I eat pizza with a fork.')
>>> print MBSP.xml(s)

<?xml version="1.0" encoding="UTF-8"?>
<text>
<sentence id="1"
       token="word, part-of-speech, chunk, preposition, relation, anchor, lemma">
      <chunk type="NP" relation="SBJ" of="1.1">
             <word type="PRP" lemma="i">I</word>
      </chunk>
      <chunk type="VP" id="1.1" anchor="1.A1">
             <word type="VBP" lemma="eat">eat</word>
      </chunk>
      <chunk type="NP" relation="OBJ" of="1.1">
             <word type="NN" lemma="pizza">pizza</word>
      </chunk>
      <chunk type="PNP" of="1.A1">
             <chunk type="PP">
                    <word type="IN" lemma="with">with</word>
             </chunk>
             <chunk type="NP">
                    <word type="DT" lemma="a">a</word>
                    <word type="NN" lemma="fork">fork</word>
             </chunk>
      </chunk>
      <chink>
             <word type="." lemma=".">.</word>
      </chink>
</sentence>
</text>
```

A `<text>` element consists of one or more `<sentence>` elements. Each `<sentence>` has a unique `id` attribute, a `token` descriptor and consists of `<chunk>` and `<chink>` elements. A `<chunk>` can have a `type` attribute (e.g. **NP**, **VP**) and/or a `relation` attribute (e.g. **SBJ**). In this case it can also have an `of` attribute with the `id` of the related **VP** chunk. A `<chunk type="`**PNP**`">` element can also have an `of` attribute, with the `anchor` of the chunk to which it is attached. A `<chunk>` consists of `<word>` elements, or other `<chunk>` elements in the case of a **PNP**. A `<word>` element contains the actual word and can have a `type` attribute (e.g. **NN**) and a `lemma` attribute. If MBSP was extended with custom tags these will be listed in the `<sentence> token` attribute and as `<word>` attributes.

```
MBSP.xml(string, token=[WORD, POS, CHUNK, PNP, RELATION, ANCHOR, LEMMA])
MBSP.Text.xml            # Returns the Text as an XML string.
MBSP.Text.from_xml(xml) # Returns a new Text from the given XML string.
```

12.2 Exporting to NLTK

The `MBSP.tree` module has an `nltk_tree()` command that returns an `nltk.tree.Tree` object from the given `MBSP.Sentence` object. This way the output from MBSP can be integrated into NLTK (http://nltk.org) for further processing.

```
>>> s = MBSP.parse('I eat pizza with a fork.')
>>> s = MBSP.split(s)
>>> print MBSP.tree.nltk_tree(s.sentences[0])

(S
  (NP (PRP I))
  (VP (VBP eat) (PNP (PP (IN with)) (NP (DT a) (NN fork))))
  (NP (NN pizza)))
```

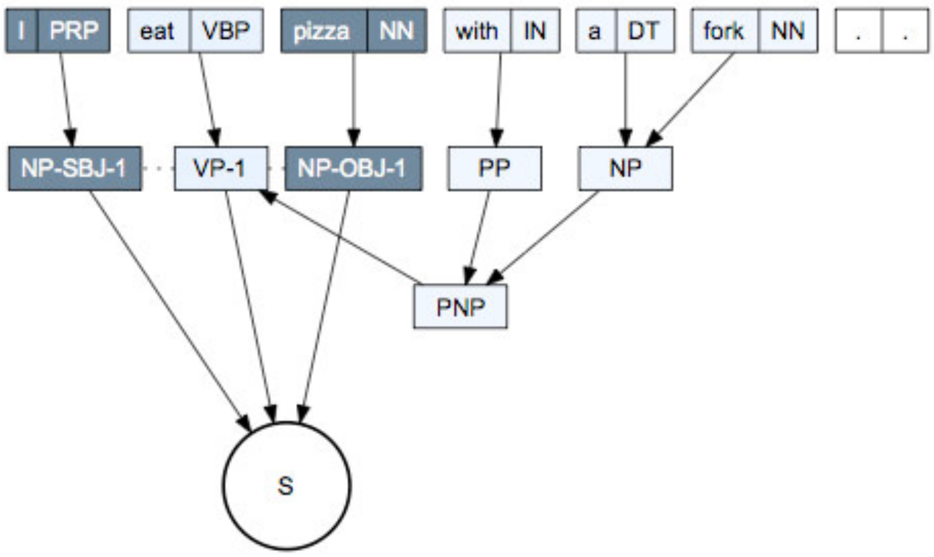
NLTK needs to be installed on your systeem for this to work.

12.3 Exporting to GraphViz

The `MBSP.tree` module has a `graphviz_dot()` command that returns a string in the DOT language (a simple way to describe graphs) from the given `MBSP.Sentence` object. DOT files can be visualized with GraphViz (http://graphviz.org), a free application for drawing graphs and exporting them as images (PDF, PNG, ...)

```
>>> s = MBSP.parse('I eat pizza with a fork.')
>>> s = MBSP.split(s)
>>> f = open('pizza.dot', 'w')
>>> f.write(MBSP.tree.graphviz_dot(s.sentences[0])
>>> f.close()
```

When `pizza.dot` is then openend in GraphViz it displays the following syntax tree:



13 Licensing

MBSP uses a duel license model and offers licenses for two distinct purposes – commercial and open source development.

- **Open source**: for open source purposes MBSP uses the GNU General Public License version 3 (http://gnu.org/licenses/gpl.html). This option requires that you contribute to the open source community by placing your application that uses MBSP under an open source license (i.e. GPLv3). This option secures all users the rights to obtain the application's full source code, modify it, and redistribute it.
- **Commercial**: for commercial purposes you can directly contact prof. Walter Daelemans (walter.daelemans at ua.ac.be). This option requires that you contribute to the continued development of the product by purchasing a commercial license. This option secures you the right to distribute your application under the license terms of your choice.

14 Appendix

MBSP assigns meaningful tags to words and groups of words in a sentence. Each tag is a short code (such as "**DT**" for "determiner"). The tag set is based on the Penn Treebank Tagging Guidelines: ftp://ftp.cis.upenn.edu/pub/treebank/doc/tagguide.ps.gz

14.1 Part-of-speech tags

Part-of-speech tags are assigned to a single word according to its role in the sentence. Traditional grammar classifies words based on eight parts of speech: the verb (**VB**), the noun (**NN**), the pronoun (**PR**), the adjective (**JJ**), the adverb (**RB**), the preposition (**PP**), the conjunction (**CC**), and the interjection (**UH**).

TAG	DESCRIPTION	EXAMPLE
CC	conjunction, coordinating	*and, or, but*
CD	cardinal number	*five, three, 13%*
DT	determiner	*the, a, these*
EX	existential there	*there were six boys*
FW	foreign word	*mais*
IN	conjunction, subordinating or preposition	*of, on, before, unless*
JJ	adjective	*nice, easy*
JJR	adjective, comparative	*nicer, easier*
JJS	adjective, superlative	*nicest, easiest*
LS	list item marker	
MD	verb, modal auxillary	*may, should*
NN	noun, singular or mass	*tiger, chair, laughter*
NNS	noun, plural	*tigers, chairs, insects*
NNP	noun, proper singular	*Germany, God, Alice*
NNPS	noun, proper plural	*we met two Christmases ago*
PDT	predeterminer	*both his children*
PRP	pronoun, personal	*me, you, it*
PRP$	pronoun, possessive	*my, your, our*
RB	adverb	*extremely, loudly, hard*
RBR	adverb, comparative	*better*
RBS	adverb, superlative	*best*
RP	adverb, particle	*about, off, up*
SYM	symbol	*%*

TO	infinitival to	*what to do?*
UH	interjection	*oh, oops, gosh*
VB	verb, base form	*think*
VBZ	verb, 3rd person singular present	*she thinks*
VBP	verb, non-3rd person singular present	*I think*
VBD	verb, past tense	*they thought*
VBN	verb, past participle	*a sunken ship*
VBG	verb, gerund or present participle	*thinking is fun*
WDT	wh-determiner	*which, whatever, whichever*
WP	wh-pronoun, personal	*what, who, whom*
WP$	wh-pronoun, possessive	*whose, whosever*
WRB	wh-adverb	*where, when*
.	punctuation mark, sentence closer	. ; ? *
,	punctuation mark, comma	,
:	punctuation mark, colon	:
(	contextual separator, left paren	(
)	contextual separator, right paren	)

14.2 Chunk tags

Chunk tags are assigned to groups of words that belong together (i.e. phrases). The most common phrases are the noun phrase (**NP**, for example *the black cat*) and the verb phrase (**VP**, for example *is purring*).

TAG	DESCRIPTION	WORDS	EXAMPLE	%
NP	noun phrase	**DT** + **RB** + **JJ** + **NN** + **PR**	*the strange bird*	51
PP	prepositional phrase	**TO** + **IN**	*in between*	19
VP	verb phrase	**RB** + **MD** + **VB**	*was looking*	9
ADVP	adverb phrase	**RB**	*also*	6
ADJP	adjective phrase	**CC** + **RB** + **JJ**	*warm and cosy*	3
SBAR	subordinating conjunction	**IN**	*whether or not*	3
PRT	particle	**RP**	*up the stairs*	1
INTJ	interjection	**UH**	*hello*	0

The IOB prefix marks whether a word is inside or outside of a chunk.

TAG	DESCRIPTION
I-	inside the chunk
B-	inside the chunk, preceding word is part of a different chunk
O	not part of a chunk

A prepositional noun phrase (**PNP**) is a group of chunks starting with a preposition (**PP**) followed by noun phrases (**NP**), for example: *under the table.*

TAG	DESCRIPTION	CHUNKS	EXAMPLE
PNP	prepositional noun phrase	**PP** + **NP**	*as of today*

14.3 Relation tags

Relations tags describe the relation between different chunks, and clarify the role of a chunk in that relation. The most common roles in a sentence are **SBJ** (subject noun phrase) and **OBJ** (object noun phrase). They link **NP** to **VP** chunks. The subject of a sentence is the person, thing, place or idea that is *doing* or *being* something. The object of a sentence is the person/thing affected by the action.

TAG	DESCRIPTION	CHUNKS	EXAMPLE	%
-SBJ	sentence subject	**NP**	*the cat sat on the mat*	35
-OBJ	sentence object	**NP** + **SBAR**	*the cat grabs the fish*	27
-PRD	predicate	**PP** + **NP** + **ADJP**	*the cat feels warm and fuzzy*	7
-TMP	temporal	**PP** + **NP** + **ADVP**	*arrive at noon*	7
-CLR	closely related	**PP** + **NP** + **ADVP**	*work as a researcher*	6
-LOC	location	**PP**	*live in Belgium*	4
-DIR	direction	**PP**	*walk towards the door*	3
-EXT	extent	**PP** + **NP**	*drop 10 %*	1
-PRP	purpose	**PP** + **SBAR**	*die as a result of*	1

14.4 Anchor tags

Anchor tags describe how prepositional noun phrases (**PNP**) are attached to other chunks in the sentence. For example, in the sentence, *I eat pizza with a fork*, the anchor of *with a fork* is *eat* because it answers the question: "In what way do I eat?"

TAG	DESCRIPTION	EXAMPLE
A1	anchor chunks that corresponds to **P1**	*eat with a fork*
P1	**PNP** that corresponds to **A1**	*eat with a fork*

OCCURENCE ESTIMATE

The given percentages for chunk and relations tags are based on tenfold cross validation on sections 10 to 19 of the WSJ Corpus of the Penn Treebank II by Sabine Buchholz, from which we derived a rough indication. The estimate means that if a 100 chunk tags are found, about 50 are **NP** tags and 35 have a **SBJ** relation tag. About 30 of the chunks would be tagged as **NP-SBJ**, and 15 as **NP-OBJ**.

Reference: Buchholz, S. (2002). *Memory-Based Grammatical Relation Finding*. ILK, Tilburg University.