Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Natural Language Processing and the Web
Practice Class 4

Prof. Chris Biemann and Seid M. Yimam

08 November 2016

In this tutorial, you will be introduced to part-of-speech tagging, lemmatization, and chunking, and then build a UIMA aggregate analysis engine which uses a POS tagger, lemmatizer, and chunker to discover lexical relationships in a corpus.

1 Part-of-speech tagging

Part-of-speech tagging (**POS tagging**) is the process of marking up the words in a text with their corresponding part of speech (e.g., noun, verb, adjective). For example, take the following sentence:

A dog had seen the cutest ferrets.

A tokenizer would split it into the following tokens:

$\boxed{\text{A}}$ $\boxed{\text{dog}}$ $\boxed{\text{had}}$ $\boxed{\text{seen}}$ $\boxed{\text{the}}$ $\boxed{\text{cutest}}$ $\boxed{\text{ferrets}}$

A part-of-speech tagger could then assign labels, or **tags**, to the tokens according to their respective parts of speech:

$\boxed{\text{A}}_{\text{DT}}$ $\boxed{\text{dog}}_{\text{NN}}$ $\boxed{\text{had}}_{\text{VBD}}$ $\boxed{\text{seen}}_{\text{VBN}}$ $\boxed{\text{the}}_{\text{DT}}$ $\boxed{\text{cutest}}_{\text{JJS}}$ $\boxed{\text{ferrets}}_{\text{NNS}}$

The inventory from which these POS tags are drawn varies from language to language, and from application to application. In this example, the tags are those used by the Penn Treebank corpus.[1]

Many natural language processing applications assume that the input text is not only tokenized but also POS-tagged. Fortunately, computers can POS-tag text with high accuracy.

In this exercise we will use the Stanford Parser [2], which outputs POS tags as a by-product. For this Parser a DKPro Wrapper exists. For English text the POS tags are a variant of the Penn Treebank tagset. DKPro defines a small hierarchical type system to represent these tags: `POS` is the supertype of all other part-of-speech annotations, with generic parts of speech like `ADJ` (adjective) and `V` (verb) as subtypes. Each of these subtypes has a string feature `value` which more precisely specifies the part of speech. The complete list of English POS types and values is shown in Table 1.

Go to Moodle and download the file `starterCode.zip`. Like in the previous tutorial, integrate the contents of the ZIP file into your solution for exercise 3, or the solution from Moodle. Open DemoPipelineTutorial4.java and examine the code and follow the instructions in the comments. The AAE contains three writers. The first one outputs POS tags. Examine its code and inspect its output.

[1] The Penn Treebank tags used here are as follows:

DT	determiner	NN	noun, singular or mass	VBD	verb, past tense
JJS	adjective, superlative	NNS	noun, plural	VBN	verb, past participle

The full Penn Treebank tagset can be seen at `http://www.computing.dcu.ie/~acahill/tagset.html`.

[2] `http://nlp.stanford.edu/software/lex-parser.shtml`

Type	Value	Part of speech	Example
ADJ	JJ	adjective	green
ADJ	JJR	adjective, comparative	greener
ADJ	JJS	adjective, superlative	greenest
ADV	RB	adverb	however, usually
ADV	RBR	adverb, comparative	better
ADV	RBS	adverb, superlative	best
ADV	WRB	*wh*-abverb	where, when
ART	DT	determiner	the
ART	EX	existential *there*	*there* is
ART	PDT	predeterminer	both the boys
ART	WDT	*wh*-determiner	which
CARD	CD	cardinal number	1, third
CONJ	CC	coordinating conjunction	and
NN	NN	noun, singular or mass	table
NN	NNS	noun plural	tables
NP	NP	proper noun, singular	John
NP	NPS	proper noun, plural	Vikings
O	(various)	other/unknown	to
PP	IN	preposition, subordinating conjunction	in, of, like
PP	RP	particle	give *up*
PR	PP	personal pronoun	I, he, it
PR	PP$	possessive pronoun	my, his
PR	WP	*wh*-pronoun	who, what
PR	WP$	possessive *wh*-pronoun	whose
PUNC	SENT	sentence-break punctuation	. ! ?
V	MD	modal	could, will
V	VB	verb *be*, base form	be
V	VBD	verb *be*, past tense	was, were
V	VBG	verb *be*, gerund/present participle	being
V	VBN	verb *be*, past participle	been
V	VBP	verb *be*, singular present, non-3d	am, are
V	VBZ	verb *be*, 3rd person singular present	is
V	VH	verb *have*, base form	have
V	VHD	verb *have*, past tense	had
V	VHG	verb *have*, gerund/present participle	having
V	VHN	verb *have*, past participle	had
V	VHP	verb *have*, singular present, non-3d	have
V	VHZ	verb *have*, 3rd person singular present	has
V	VV	verb, base form	take
V	VVD	verb, past tense	took
V	VVG	verb, gerund/present participle	taking
V	VVN	verb, past participle	taken
V	VVP	verb, singular present, non-3d	take
V	VVZ	verb, 3rd person singular present	takes

Table 1: DKPro English POS annotation types and values

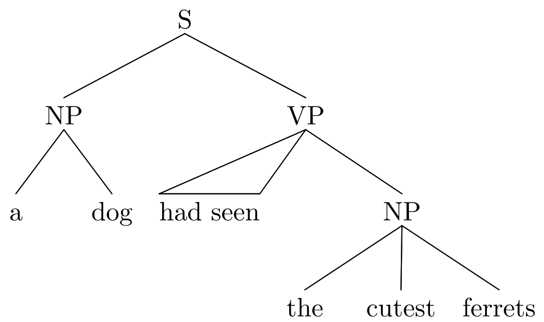
2 Lemmatization

A **lemma** is the canonical, uninflected or "dictionary" form of a word. For example, the lemma of *smallest* is *small*, and the lemma of *eating* is *eat*. In many languages, the lemma for nouns is the nominative singular form, the lemma for adjectives is the nominative singular positive form, and the lemma for verbs is the infinitive. But given an inflected form, finding the lemma (a process called **lemmatization**) is not always as easy. Words often undergo regular spelling changes when inflected—for example, in English, verbs and adjectives ending in `-`*e* often drop this letter when inflecting: *bake* → *baking*. Sometimes final consonants are doubled, as in (British) English *travel* → *travelling*. An accurate algorithm for lemmatization must be aware of these sorts of inflectional rules and know how to undo them to arrive at the base form of the word. It must also know about completely irregular cases, such as *go* → *went*, *mouse* → *mice*, and *good* → *better*. Lemmatization is a difficult task for computers, and requires some basic understanding of the grammatical context and properties of the word. For example, the lemma of *dove* depends on whether the word is being used as a noun (as in the bird) or a verb (as in the past tense of *dive*). However, lemmatization is an important task because, as with part-of-speech tagging, many NLP applications rely on lemmatized text.

Since Stanford Parser also outputs lemmas, there is no need for an additional component. The lemmas for each token are stored in the string `value` of the annotation type `Lemma`. Have a look at `LemmaWriter` and its output. Are all words lemmatized correctly?

3 Parsing vs. chunking

Parsing is the process of analyzing a text to determine its grammatical structure. It goes beyond part-of-speech tagging (though that is often a first step) by grouping words within sentences into hierarchical grammatical structures. Here is a possible parse tree for the example sentence introduced in Problem 1:



Proper parsing is a hard problem in computational linguistics. While identifying some sort of sentence structure is important for many NLP applications, not all of them require something as detailed and complicated as a parse tree. **Chunking**, also known as **shallow parsing**, is a simplified form of sentence analysis which identifies basic constituents (noun groups, verb groups, etc.) but does not specify their internal structure.

For our example POS-tagged sentence, a chunker might identify noun chunks (NC) and verb complexes (VC) as follows:

[[A]$_{\text{DT}}$ [dog]$_{\text{NN}}$]$_{\text{NC}}$ [[had]$_{\text{VBD}}$ [seen]$_{\text{VBN}}$]$_{\text{VC}}$ [[the]$_{\text{DT}}$ [cutest]$_{\text{JJS}}$ [ferrets]$_{\text{NNS}}$]$_{\text{NC}}$

Chunks can be extracted from the output of the Stanford Parser. The DKPro wrapper for the Parser annotates English chunks with the annotation types shown in Table 2; all these are subtypes of `Chunk`. `ChunkWriter` contains an example for accessing chunks. Examine its code and inspect its output.

4 Hearst patterns

In this problem, you will employ the POS, lemma and chunking information to discover lexical relationships in a corpus.

Hearst patterns are lexico-syntactic patterns first used by Marti Hearst[3] to discover hyponyms in

---

[3]Marti Hearst. "Automatic Acquisition of Hyponyms from Large Text Corpora." In *Proceedings of the 14th International Conference on Computational Linguistics (COLING-1992)*. doi:10.3115/992133.992154. `http://people.ischool.berkeley.edu/~hearst/papers/coling92.pdf`

Type	Chunk
NP	noun chunks/phrase
VP	verb chunks
ADJP	adjective chunks (not inside of noun chunks)
ADVP	adverb chunks (not inside of noun or adjective chunks)
PP	prepositional chunk (usually embeds a noun chunk)
SBAR	subordinated clause
PRT	particles
LST	enumeration symbol
S	sentence
ROOT	ROOT node (most times this is a sentence)
INTJ	interjection
UCP	unlike coordinated chunk

Table 2: DKPro English chunking annotation types

large text corpora. (A **hyponym** is a term which denotes a more specific or subordinate group of another term, called a **hypernym**. For example, *tiger* is a hyponym of *mammal*, which is in turn a hyponym of *animal*. Therefore *animal* is a hypernym of *mammal*, and *mammal* is a hypernym of *tiger*.) Hearst observed that certain linguistic constructions can be used to infer hyponymy relationships. For example, in the phrase "works by such authors as Herrick, Goldsmith, and Shakespeare", it is obvious that *Herrick*, *Goldsmith*, and *Shakespeare* are all hyponyms of *author*. In general, any phrase of the pattern "such NP_0 as NP_1, ..., and NP_n" implies that the noun phrases NP_1 through NP_n are hyponyms of NP_0.

Table 3 shows some patterns originally proposed by Hearst, along with examples. Using the introduced Dkpro Wrappers, write a UIMA application which looks for hyponyms by finding Hearst patterns in a collection of documents.[4]

4.1

Write a `HearstAnnotator`, which finds Hearst patterns in documents and annotates them with HearstAnnotations. Therefore, first generate the annotation class from `HearstAnnoation.xml`, which is provided with this exercise. HearstAnnotation has the fields `TypeOf`, `Hypernym` and `Hyponym`. Use TypeOf to store which of the five Hearst patterns was found. For this exercise you do not need to worry about the bracketed parts of the Hearst patterns. E.g. in the example for the second Hearst pattern your HearstAnnotator only has to find `Herrick` as a hyponym of author (and not necessarily `Goldsmith` and `Shakespeare` as a hyponym of author). One way to implement this task is to iterate over all sentences in the document and then iterate over all noun phrases in one sentence and look for Hearst patterns. Most likely you will find the methods of JCasUtil.java very helpful.

4.2

Implement a HearstWriter, which prints the follwing interesting statistics:

a)Let the HearstWriter print out the most commonly found hyponym-hypernym relations.
An example result could look like this:

`count`	`hyponym`	`hypernym`
4	house	building
2	Herrick	author
1	France	country

b) Let the HearstWriter print the top five most productive Hearst patterns.
An example result could look like this:

`count`	`Hearst pattern`
3	NP such as NP
3	such NP as NP
2	NP, inculding NP
1	NP, especially NP
1	NP and/or other NP

[4]See Moodle for links to some good large corpora you can use.

Hearst pattern	**Example**
NP_0 such as NP (… and/or NP)	… played stringed instruments, such as the guitar, with …
such NP_0 as NP (… and/or NP)	… works by such authors as Herrick, Goldsmith, and Shakespeare …
NP (…) and/or other NP_0	… bruises, wounds, broken bones or other injuries …
NP_0, including NP (… and/or NP)	… all common-law countries including Canada and England …
NP_0, especially NP (… and/or NP)	… most European countries, especially France, England, and Spain …

Table 3: Hearst patterns