Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Natural Language Processing and the Web
Practice Class 3

Prof. Chris Biemann and Seid M. Yimam

01 November 2016

`Tokenization`, also known as `word segmentation`, is the task of separating (i.e., *tokenizing*) words in a text. Tokenization is an essential preprocessing step before any linguistic analysis of text is done, and it is a non-trivial problem. For example, think of the expressions *New York* and *rock 'n' roll*. They are individual words, despite the fact that they contain spaces. On the other hand, the expression *I'm*, which is called a *clitic contraction*, consists of the two separate words *I* and *am*.

In the last tutorial we already used existing tokenizers in our UIMA project. In this tutorial we will examine and implement some basic tokenization methods and discuss the difficulties encountered.

1 A naïve whitespace tokenizer

In languages like English or German (in contrast to others like Chinese or Ancient Greek) words are usually separated from each other by whitespace. Spaces, however, are not the only characters which may stand between words in the text. In a sentence like

"I'm Sam," he said.

segmenting purely on whitespace would erroneously include punctuation along with the words:

$\boxed{\text{“I'm}}\ \boxed{\text{Sam,”}}\ \boxed{\text{he}}\ \boxed{\text{said.}}$

Log into Moodle and download the file `Tutorial3Code.zip`. Integrate the java classes into your existing project from the last tutorial (or the solution from Moodle). Place the XML files in `src/main/resources/desc/type`. Place also the `META-INF` folder under `src/main/resources/`. Then open each of the XML files with the `Component Description Editor` (right click → select Open With → Component Descriptor Editor) and navigate to the tab `Type System` and press the button `JCasGen`. This will create Java code in the package `type` that can be referenced from annotators. The class `TokenizerExample` contains a naïve tokenizer which performs segmentation solely on spaces. Examine and run the code. What problems in the output do you see?

2 Tokenization with the `java.text` package

Some of the errors mentioned above could be addressed by treating punctuation, in addition to whitespace, as part of the word boundary. But punctuation often occurs *within* words, as in *U.S.* or *3.14159*.

Implement a tokenizer using `java.text.BreakIterator`. (You can find the API specification at http://docs.oracle.com/javase/8/docs/api/java/text/BreakIterator.html.) What are the improvements over the previous naïve approach? What are the issues that have not been solved?

3 Tokenization in UIMA

You will now adapt the two tokenizers you have written into UIMA annotators. The class `WhitespaceTokenizer` contains a skeleton code for a UIMA aggregate analysis engine (implemented using uimaFIT). This class already contains an example for adding an annotation to the CAS. In this case the annotation type is WSToken, which you previously have created from one of the XML files.

3.1

Adapt the naïve whitespace tokenizer into a UIMA annotator `WhitespaceTokenizer` which annotates the input with the annotation type `WSToken`. Create a new Pipeline (`e.g. PipelineTutorial3`), which should run the reader and the writer that you created in the previous tutorial and your tokenizer. Run the Pipeline and observe the output. Extend the CAS consumer if needed.

3.2

Adapt your `BreakIterator` tokenizer into a UIMA annotator `e.g. BreakIteratorTokenizer` which annotates the input with the annotation type `BIToken`. Integrate this tokenizer into your pipeline and observe its output.

4 Iterating over annotations

The `JCasUtil`, which is provided by the `org.apache.uima.fit.util` package has two different methods to access annotations. In the previous tutorial examples for using these methods have been shown. While `JCasUtil.select()` allows to select all annotation of a particular type from a CAS, `JCasUtil.selectCovered()` can retrieve all annotations of a particular type that are fully contained in the span of a given annotation.

4.1

The Java `BreakIterator` class can also be used to split a string into sentence boundaries instead of word boundaries; using the method `getSentenceInstance()` instead of `getWordInstance()`. Adapt `BreakIteratorTokenizer` into a new annotator `SentenceSplitter` which annotates input with the annotation type `Sentence`. Modify the aggregate analysis engine (your pipeline) to include `SentenceSplitter` and inspect the results.

For this task you will need to first define a new annotation type using the UIMA Eclipse plugin's *component descriptor editor*:

1. In the Package Explorer, right-click on the `src/main/resources/desc/type` folder and select New → Other... → UIMA → Type System Descriptor File → Next. A “New Type System Descriptor File” dialog should appear.
2. The Parent Folder field should be prepopulated with the path to the `type` folder. If not, enter it yourself.
3. In the File name field, enter `Sentence.xml` and hit Finish.
4. Eclipse should automatically open the XML file with the UIMA Component Description Editor. If not (or if you want to open it again later), right-click on the `Sentence.xml` file and select Open With → Component Descriptor Editor.
5. Select the Type System tab and click the Add Type button in the Types (or Classes) section. An “Add a Type” dialog should appear.
6. For the Type Name, enter `de.unihamburg.informatik.nlp4web.tutorial.tut3.type.Sentence` and for the Supertype, enter `uima.tcas.Annotation`. Then click OK.
7. Hit the JCasGen button. This automatically generates a Java class for your new type which you will see appear in your project in the Package Explorer.

4.2

Implement a consumer `TokensPerSentenceWriter` which prints the number of `BITokens` in each sentence. Modify the aggregate analysis engine to include `TokensPerSentenceWriter` and inspect the results.

5 Accessing DOM objects from a webpage – Optional

In the last tutorial, you created a webpage reader with the `jsoup` package, which retrieved a webpage, parsed its raw HTML markup to a DOM (`http://www.w3.org/DOM/#what`), extracted the text from the DOM and sent the text as a document to the pipeline. Besides having a method to extract the entire text of a webpage, `jsoup` allows to access DOM elements at a more fine-grained level, using a CSS-like syntax, so-called *selectors* (`http://jsoup.org/cookbook/extracting-data/selector-syntax`).

5.1

Extend your webpage reader (or the version from Moodle) to allow an optional parameter for passing a selector to jsoup. If set, the reader should output every element that jsoup returns for this selector as a document. If the parameter is not set, the reader should output the `body` element of the webpage. Test your reader with an appropriate URL-selector pair (e.g. the headlines of a news website).