The CTB pretokenizer

Technical and end-user documentation of the DK-CLARIN WP 2.1 pretokenizer
DK-CLARIN WP 2 Technical Report.
Jakob Halskov, DSN, and Jørg Asmussen, DSL
Final version of November 23, 2009[1]

Document history

23.11.2009:

- Minor corrections.

Outline

[1] The most recent version can be downloaded from:
http://korpus.dsl.dk/clarin/corpus-doc/ctb-tokeniser.pdf

1 Introduction

The pretokenizer is a tool, implemented as a web service, which takes as input a raw text (preferably with <p> annotations), a prefix letter (for prefixing token IDs) and finally a text ID which has to be unique within the DK-CLARIN project. The input must comply with a specific XML structure (cf. the demo in Section 4.1). On the basis of this input, a pretokenized version of the text (no header) is returned to the client. The pretokenized format complies with TEI-P5 and will be referred to as TEI-P5-WP2 format in this document.

For some tasks the TEI-P5-WP2 format will constitute an adequate degree of tokenization, but for other tasks it may not. For example, common multiword expressions like *i går* ('yesterday') are not recognized and annotated as a single token. For this reason we refer to the tokenizer as a "pretokenizer" and encourage users to produce separate annotation layers (span groups) on the basis of TEI-P5-WP2 in case they need more sophisticated or customized tokenization, see Asmussen (2009a).

2 Definitions

A key issue when implementing a tokenizer is which characters should be considered punctuation (i.e. non-word characters). Punctuation characters should be characterized by having no semantic impact on the words in the text. In other words, if a punctuation character is removed from the text, this should in no way change the meaning of the words in the text (denotational, connotational or otherwise). If, for example, currency symbols, degree symbols and so on were to be deleted, it would cause a loss of meaning. For this reason such characters are not considered punctuation.

2.1 List of punctuation characters

On the basis of the discussion in Section 2 (and discussions in the WP2 project group which agreed upon using the interpunction characteres listed in Wikipedia, cf. `http://da.wikipedia.org/wiki/Tegns\T1\aetning` and `http://en.wikipedia.org/wiki/Punctuation`) the list of punctuation characters used in the pretokenizer is limited to the following:

' apostrophe

(bracket, round, opening

) bracket, round, closing

[bracket, square, opening

] bracket, square, closing

{ bracket, curly, opening

} bracket, curly, closing

: colon

, comma

– dash, short

— dash, long

/ slash

**** backslash

... ellipsis

! exclamation mark

. full stop

‹ guillemet, single, left

› guillemet, single, right

« guillemet, double, left

» guillemet, double, right

- hyphen [missing in CST list]

? question mark

‘ quotation mark, single, left

’ quotation mark, single, right

“ quotation mark, double, left

” quotation mark, double, right

" quotation mark, double, upright

; semicolon

Whitespace characters are captured by a regular expression which conflates whitespace sequences into a single whitespace character (`\s+`). The pretokenizer does not record the number or types of whitespace in the input.

3 Technical implementation

3.1 Choice of architectures, platforms and technologies

Two equally popular web service architectures are REST (Representational State Transfer) and SOAP (Simple Object Access Protocol). They each have their merits and drawbacks.

A major advantage of SOAP is that it can satisfy a wide range of non-functional requirements, in particular Quality of Service (QoS) requirements like secure, reliable and protocol-independent messaging. For SOAP to work there must be an

HTTP body in which to place the SOAP envelope (containing the payload of the message and its metadata). This means that SOAP webservices always rely on the POST method even for so-called idempotent operations (i.e. operations which do not change anything on the server, e.g. simple requests for information). This violates a fundamental principle of the HTTP protocol, and must be considered a drawback of SOAP. Another drawback of SOAP is that it involves the use of a quite verbose XML format (literally hundreds of different specifications) which may introduce overhead and needless complexity.

While REST does not allow for much QoS, its key merit is exactly its simplicity and transparency. According to the REST webservice design pattern everything is considered a resource, and all resources are organized like a standard file system (which, in fact, resembles the structure of the early WWW). The operations which can be performed on resources are limited to the standard HTTP methods, i.e. POST, PUT, GET and DELETE.

Since the eXist open source XML database system is already being used in DK-CLARIN as a text bank, see Asmussen (2009b), and since eXist features an integrated REST-style HTTP server interface, REST was selected as the architectural style for the pretokenizer webservice. Another reason for avoiding SOAP is that QoS aspects are not an issue, so using SOAP and WSDL would only introduce needless complexity.

Having decided on a native XML database as the platform for the text bank in DK-CLARIN WP2.1 and WP2.2, it seemed only logical to stay with the XML family and select the XQuery technology as the implementation language for the pretokenizer tool. It only made the choice even more obvious that XQuery scripts which are stored in an eXist database collection can, in fact, be executed by simply pointing your web browser to the REST URL.[2]

3.2 The pretokenizer

The pretokenizer is implemented as an XQuery script which in addition to built-in functions like `tokenize` and `normalize-space` makes use of the following five self-defined functions:

1. `tok:tokenize-doc(text,ID-prefix)`
2. `local:isolate-punctuation(string)`
3. `local:punct(string,ID)`
4. `local:space(ID)`
5. `local:token(string,ID)`

[2] See http://exist.sourceforge.net/devguide_xquery.html#storedxq for more details.

Four of the self-defined functions are local, but the `tokenize-doc` function resides in the `tok` namespace and is the main function which calls the other four functions to carry out subtasks of the tokenization process. All five functions are grouped into an XQuery module `pretokenize.xqm`, and this module is included from a single XQuery script `pretokenize` which is invoked by requests addressed to the pretokenizer webservice at http://ctbws.dsl.dk/pretokenize.

The `tokenize-doc` function takes two arguments, namely the <p> annotated text and an ID prefix. This function iterates through the <p> elements of the input document, calls the `local:isolate-punctuation` function on each <p> element (which inserts ^ characters around all punctuation characters and whitespace sequences as defined in Section 2.1) and then tokenizes the text string in the <p> element using the ^ character as delimiter.

For each token, `tokenize-doc` calls either `local:punct`, `local:space`, or `local:token` depending on the contents of the token. These three functions in turn insert <c> or <w> elements in the output with appropriate ID numbers as attributes. The main XQuery script then finally wraps everything in a <text> element and returns it to the client.

3.2.1 Source code

The XQuery source code of the tokenizer service and the tokenizer function module can be downloaded from the following URLs respectively:

- ▷ http://ctbws.dsl.dk/pretokenize.zip
- ▷ http://ctbws.dsl.dk/lib/pretokenize.xqm.zip

Please contact Jørg Asmussen at `ja@dsl.dk` before modifying the code!

3.3 The webservice

As described in the online developer's guide for eXist,[3] eXist databases can be deployed in various ways, one of them being a stand-alone server process accessible through a REST-style API through HTTP. This is the simplest and quickest way to access the database because eXist features a built-in web server which (conveniently) treats all HTTP request paths as paths to a database collection.

The default listen address for the eXistServlet is

- ▷ `http://localhost:8080/exist/rest`

but when running as a stand-alone process the server listens to port 8088, e.g.:

- ▷ `http://localhost:8088/db/ctb/xq`

XPath expressions or XQueries can either be added directly to the request string as values of the request parameter, `_query`, e.g.

[3] See http://exist.sourceforge.net/deployment.html.

▷ `http://localhost:8088/exist/rest/db/shakespeare?_query=//SPEECH[SPEAKER=%22JULIET%22]&_start=3&_howmany=5`

or they can be stored on the server in a database collection and called in a similar fashion using a simple HTTP GET request, e.g.:

▷ `http://localhost:8088/exist/rest/db/test/guess.xql`

In both cases the server returns raw XML to the browser (unless otherwise specified in the query).

In the current implementation of the pretokenizer webservice the public endpoint

▷ http://ctbws.dsl.dk/pretokenize

is mapped to an otherwise 'hidden' eXist database server via DNS.

4 Use of the pretokenizer webservice

The endpoint of the pretokenizer webservice is the following URL:

▷ http://ctbws.dsl.dk/pretokenize

Data upload is via the POST method and the MIME type of the data must be application/xml. Also, the XML content which is posted to the webservice must comply with the following structure – the element contents given are for illustration purposes only:

```
<request>
  <idPrefix>A</idPrefix>
  <textId>2100000000</textId>
  <text>
    <p>A paragraph.</p>
    <p>Another paragraph.</p>
  </text>
</request>
```

The corresponding output is:

```
<text>
  <body>
    <p>
      <w xml:id="A-2100000000-2-1">A</w>
      <c xml:id="A-2100000000-2-2" type="s"/>
      <w xml:id="A-2100000000-2-3">paragraph</w>
      <c xml:id="A-2100000000-2-4" type="p">.</c>
```

```
    </p>
    <p>
      <w xml:id="A-2100000000-4-1">Another</w>
      <c xml:id="A-2100000000-4-2" type="s"/>
      <w xml:id="A-2100000000-4-3">paragraph</w>
      <c xml:id="A-2100000000-4-4" type="p">.</c>
    </p>
  </body>
</text>
```

4.1 Demo

An interactive demo describing the XML structure of the input and giving an example of the output returned by the webservice is available at:

▷ http://korpus.dsl.dk/clarin/demo/pretokenize/

4.1.1 Source code

The Flex source code of the demo application can be viewed and downloaded by right-clicking on the demo application and choosing the menu option *View Source*.

Please contact Jørg Asmussen at `ja@dsl.dk` before modifying the code!

4.2 Use example (a simple Java client)

There are many ways of generating and dispatching an HTTP POST request programmatically (as opposed to using an HTML form with action and method attributes), and there are multiple programming languages which can be used to implement a simple client which takes an XML structure as input, builds and dispatches the POST request and prints the response received from the server.

The following code illustrates how one could implement a simple Java client which tokenizes a text file using the DK-CLARIN WP2.1 pretokenizer webservice:

```
*** TokenizeText.java ***
package mystuff;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.URL;
import java.net.URLConnection;
public class TokenizeText {
  public static void main(String[] args) {
    if (args.length < 1) {
      System.err.println("Usage: <XML text>");
    } else {
      StringBuilder sb = new StringBuilder();
      String s = "";
      String input = "";
```

```
      URL url;
      URLConnection urlConn;
      try {
        //Read the XML document and store it in a String object
        BufferedReader fin =
          new BufferedReader(new FileReader(args[0]));
        while ((s = fin.readLine()) != null) {
          sb.append(s);
        }
        input = sb.toString();
        fin.close();
        // URL of pretokenizer WS
        url = new URL("http://ctbws.dsl.dk/pretokenize");
        // URL connection channel.
        urlConn = url.openConnection();
        // Let the run-time system (RTS) know that we want input.
        urlConn.setDoInput(true);
        // Let the RTS know that we want to do output.
        urlConn.setDoOutput(true);
        // Specify the content type.
        urlConn.setRequestProperty("Content-Type",
                                   "application/xml");
        // Send POST output. No need to use
        // setRequestMethod("POST") since only POST requests have
        // HTTP bodies with a particular content-type
        OutputStreamWriter oswr =
          new OutputStreamWriter(urlConn.getOutputStream(),
                                 "UTF8");
        oswr.write(input);
        oswr.flush();
        oswr.close();
        // Get response data.
        BufferedReader br2 =
          new BufferedReader(new InputStreamReader(
                             urlConn.getInputStream(), "UTF8"));
        String str = "";
        while ((str = br2.readLine()) != null) {
          // Output could also be printed to a file
          System.out.println(str);
        }
        br2.close();
      }
      catch (Exception e) {
        System.err.println(e.getMessage());
      }
    }
  }
}
```

5 References

Asmussen, J. (2009a). Format and markup of corpus texts. Technical report, DK-CLARIN, korpus.dsl.dk/clarin/corpus-doc/text-format.pdf.

Asmussen, J. (2009b). Textbank software. Technical report, DK-CLARIN, korpus.dsl.dk/clarin/corpus-doc/textbank-software.pdf.