

Specifying orthography: harmonization, tokenization and transliteration

Michael Cysouw

2014-11-06

1 Introduction

Given any collection of linguistic strings, there are various issues that often arise in using these linguistic strings in the computational processing of such data. This vignette will give a short practical introduction to the solutions offered in the `qlcTokenize` package. For a full theoretical discussion of all issues involved, see Moran & Cysouw (forthcoming).

All proposals made here (and in the paper by Moran & Cysouw) are crucially rooted in the structure and technologies developed over the last few decades by the Unicode Consortium. Specifically the implementation as provided by the UCI and their porting to R in the `stringi` package are crucial for the functions described here. One might even question, whether there is any need for the functions in this package, and whether the functionality of `stringi` is not already sufficient. We see our additions as high-level functionality that (hopefully) is easily enough to be applied to also allow non-technically-inclined linguists to use it.

Specifically, we offer an approach to document *tailorder grapheme clusters* (as they are called by the Unicode consortium). To deal consistently with such clusters, the official Unicode route would be to produce *Unicode Local Descriptions*, which are overly complex for the use-cases that we have in mind. In general, our goal is to allow for quick and easy processing, which can be used for dozens (or even hundreds) of different languages/orthographies without becoming a life-long project.

We see various use-cases for the `qlcTokenize` package, e.g.:

- checking consistency of the orthographic representation in some data;
- checking for consistent application of a pre-defined orthography structure (e.g. the IPA);
- tokenization of the orthography into functional units (“graphemes”), which is highly useful in language comparison (e.g. character alignment);
- transliteration of orthography to another orthographic representation, specifically in cases in which the transliteration is geared towards reducing orthographic complexity (e.g. sound classes).

In general, our solutions will not be practical for ideosyncratic orthographies like English or French, nor for character-based orthographies like Chinese or Japanese, but is mostly geared towards practical orthographies as used in the hundreds (thousands) of other languages in the world.

2 Installing the package

The current alpha-version of the package `qlcTokenize` is not yet available on CRAN (*Comprehensive R Archive Network*) for easy download and application. If you haven’t done so already, please install the package `devtools` and then install the package `qlcTokenize` directly from github.

```
# install devtools from CRAN
install.packages("devtools")
# install qlcTokenize from github using devtools
devtools::install_github("cysouw/qlcTokenize")
# load qlcTokenize package
```

```
library(qlcTokenize)
# access help files of the package
help(qlcTokenize)
```

3 Orthography Profiles

The basic object in `qlcTokenize` is the *Orthography Profile*. This is basically just a simple tab-separated file listing all (tailored) graphemes in some data. An orthography profile can be easily made by using `write.orthography.profile`. The result of this function is an R-dataframe, but it can also be directly written to a file by using the option `file = path/filename`.

```
test <- "háillo háll "
```

```
write.orthography.profile(test)
```

graphemes	replacements	frequency	codepoints	names
		1	U+0020	SPACE
á	á	1	U+00E1	LATIN SMALL LETTER A WITH ACUTE
á	á	1	U+0061, U+0301	LATIN SMALL LETTER A, COMBINING ACUTE ACCENT
h	h	2	U+0068	LATIN SMALL LETTER H
l	l	4	U+006C	LATIN SMALL LETTER L
o	o	1	U+006F	LATIN SMALL LETTER O
o	o	1	U+043E	CYRILLIC SMALL LETTER O

There are a few interesting aspects in this orthography profile.

- First note that spaces are included in the orthography profile. Space is just treated as any other character in this bare-bones function.
- Second, note that there are two different “o” characters. Looking at the Unicode codepoints and names it becomes clear that the one is a latin letter and the other a cyrillic letter. On most computer screens/fonts these symbols look completely identical, so it is actually easy for such a thing to happen (e.g. when writing with a russian keyboard-setting you might type a cyrillic “o”, but when copy-pasting something from some other source, you might end up with a latin “o”). The effect is that some words might look identical, but that they are not identical for the computer

```
# the differenec between various "o" characters is mostly invisible on screen
"o" == "o" # these are the same "o" characters, so this statement is true
```

```
## [1] TRUE
```

```
"o" == " " # this is one latin and and cyrillic "o" character, so this statement is false
```

```
## [1] FALSE
```

- Third, there are two different “á” characters, one being composed of two elements (the small letter a with a separate combining acute accent), the second being a single “precomposed” element (called “small letter a with acute”). The same problem as with the “o” occurs here: they look identical, but they are not (always) identical to

the computer. For this second problem there is an official Unicode solution (called ‘normalisation’, more on that below). It might even happen that when you just copy-paste the above test-string into your own R-console, that the problem automatically vanishes (because the clipboard might automatically do so-called NFC-normalisation).

- By default, this function lists all the Unicode codepoints and names. If you don’t want them, add the option `info = FALSE`.
- By default, this functions adds a column “replacements” which will be used for transliteration later. If you don’t want this columns, add the option `replacements = FALSE`
- Finally, note that the function also accepts vectors of strings:

```
test <- c("this thing", "is", "a", "vector", "with", "many", "strings")
```

```
write.orthography.profile(test)
```

graphemes	replacements	frequency	codepoints	names
		1	U+0020	SPACE
a	a	2	U+0061	LATIN SMALL LETTER A
c	c	1	U+0063	LATIN SMALL LETTER C
e	e	1	U+0065	LATIN SMALL LETTER E
g	g	2	U+0067	LATIN SMALL LETTER G
h	h	3	U+0068	LATIN SMALL LETTER H
i	i	5	U+0069	LATIN SMALL LETTER I
m	m	1	U+006D	LATIN SMALL LETTER M
n	n	3	U+006E	LATIN SMALL LETTER N
o	o	1	U+006F	LATIN SMALL LETTER O
r	r	2	U+0072	LATIN SMALL LETTER R
s	s	4	U+0073	LATIN SMALL LETTER S
t	t	5	U+0074	LATIN SMALL LETTER T
v	v	1	U+0076	LATIN SMALL LETTER V
w	w	1	U+0077	LATIN SMALL LETTER W
y	y	1	U+0079	LATIN SMALL LETTER Y

Normally, you won’t type your data directly into R, but load the data from some file with functions like `scan` or `read.table`, and then perform `write.orthography.profile` on the data. Given the information as provided by the orthography profile, you might then want to go back to the original file and correct the inconsistencies, and then check again to see if everything is consistent now.

There is also a corresponding function `read.orthography.profile` in case you have a profile made yourself, or got a profile from somebody else. However, in most practical situations, you will only use both these low-level read/write-functions as part of a more powerful function called `tokenize` that will be described next.

4 Tokenization

In most cases you will probably want to use the function `tokenize`. Besides creating orthography profiles, it will also check orthography profiles against new data (and give warnings if there is something), it will separate the input strings into graphemes, and even perform transliteration. Let’s run through a typical workflow using `tokenize`.

Given some data in a specific orthography, you can call `tokenize` on the data to create an initial orthography profile (just like with `write.orthography.profile` discussed above, though there are a few small differences. For example: spaces are not included by default here).

The output of `tokenize` always is a list of three elements: `$strings`, `$orthography.profile`, and `$warnings`. The second element in the list `$orthography.profile` is the table we already encountered above. The first element `$strings` is a table with the original strings, and the tokenization into graphemes as specified by the orthography profile (which in the case below was automatically produced, so there is nothing strange happening here, just a splitting into letters). The `$warnings` are just empty at this stage, but it will contain information about strings that cannot be tokenized with a pre-established profile.

```
tokenize(test)
```

```
## $strings
##   originals      tokenized
## 1 this thing t h i s # t h i n g
## 2      is          i s
## 3      a          a
## 4   vector      v e c t o r
## 5     with      w i t h
## 6    many      m a n y
## 7  strings      s t r i n g s
##
## $orthography.profile
##   graphemes replacements frequency codepoints      names
## 1      a          a          2    U+0061 LATIN SMALL LETTER A
## 2      c          c          1    U+0063 LATIN SMALL LETTER C
## 3      e          e          1    U+0065 LATIN SMALL LETTER E
## 4      g          g          2    U+0067 LATIN SMALL LETTER G
## 5      h          h          3    U+0068 LATIN SMALL LETTER H
## 6      i          i          5    U+0069 LATIN SMALL LETTER I
## 7      m          m          1    U+006D LATIN SMALL LETTER M
## 8      n          n          3    U+006E LATIN SMALL LETTER N
## 9      o          o          1    U+006F LATIN SMALL LETTER O
## 10     r          r          2    U+0072 LATIN SMALL LETTER R
## 11     s          s          4    U+0073 LATIN SMALL LETTER S
## 12     t          t          5    U+0074 LATIN SMALL LETTER T
## 13     v          v          1    U+0076 LATIN SMALL LETTER V
## 14     w          w          1    U+0077 LATIN SMALL LETTER W
## 15     y          y          1    U+0079 LATIN SMALL LETTER Y
##
## $warnings
## NULL
```

Now, you can work further with this profile inside R, but we find it easier to write the results to files, then correct/change these files, and use R again to process the data again. In this vignette we will not start writing anything to your disk (so the following commands will not be executed), but you should try something like the following:

```
dir.create("~/Desktop/tokenize")
setwd("~/Desktop/tokenize")
tokenize(test, file="test")
```

We are going to add two new graphemes to the profile: open the file “test.prf” (in the folder “tokenize” on your Desktop) with a text editor like Textmate, Textwrangler or Notepad ++ (don’t use Microsoft Word!!!). First, add a new line with only “th” on it and, second, add another line with only “ng” on it. The file will then roughly look like this:

graphemes	replacements	frequency	codepoints	names
a	a	2	U+0061	LATIN SMALL LETTER A
c	c	1	U+0063	LATIN SMALL LETTER C
e	e	1	U+0065	LATIN SMALL LETTER E
g	g	2	U+0067	LATIN SMALL LETTER G
h	h	3	U+0068	LATIN SMALL LETTER H
i	i	5	U+0069	LATIN SMALL LETTER I
m	m	1	U+006D	LATIN SMALL LETTER M
n	n	3	U+006E	LATIN SMALL LETTER N
o	o	1	U+006F	LATIN SMALL LETTER O
r	r	2	U+0072	LATIN SMALL LETTER R
s	s	4	U+0073	LATIN SMALL LETTER S
t	t	5	U+0074	LATIN SMALL LETTER T
v	v	1	U+0076	LATIN SMALL LETTER V
w	w	1	U+0077	LATIN SMALL LETTER W
y	y	1	U+0079	LATIN SMALL LETTER Y
th				
ng				

Now try to use this profile with the function `tokenize`. Note that you will get a different tokenization of the strings (“th” and “ng” are now treated as a complex grapheme) and you will also obtain an updated orthography profile, which you could also immediately use to overwrite the existing profile on your disk.

```
tokenize(test, orthography.profile = "test")

# with overwriting of the existing profile:
# tokenize(test, orthography.profile = "test", file = "test")

# note that you can abbreviate this in R:
# tokenize(test, o = "test", f = "test")
```

```
## $strings
##   originals      tokenized
## 1 this thing th i s # th i ng
## 2      is          i s
## 3      a          a
## 4   vector    v e c t o r
## 5    with      w i t h
## 6   many      m a n y
## 7  strings  s t r i n g s
##
## $orthography.profile
##   graphemes replacements frequency    codepoints
## 1      a          a          2      U+0061
## 2      c          c          1      U+0063
## 3      e          e          1      U+0065
## 4      i          i          5      U+0069
## 5      m          m          1      U+006D
## 6      n          n          1      U+006E
## 7     ng         ng          2 U+006E, U+0067
## 8      o          o          1      U+006F
## 9      r          r          2      U+0072
```

```

## 10      s      s      4      U+0073
## 11      t      t      2      U+0074
## 12     th     th      3 U+0074, U+0068
## 13      v      v      1      U+0076
## 14      w      w      1      U+0077
## 15      y      y      1      U+0079
##
##                                names
## 1                                LATIN SMALL LETTER A
## 2                                LATIN SMALL LETTER C
## 3                                LATIN SMALL LETTER E
## 4                                LATIN SMALL LETTER I
## 5                                LATIN SMALL LETTER M
## 6                                LATIN SMALL LETTER N
## 7  LATIN SMALL LETTER N, LATIN SMALL LETTER G
## 8                                LATIN SMALL LETTER O
## 9                                LATIN SMALL LETTER R
## 10                               LATIN SMALL LETTER S
## 11                               LATIN SMALL LETTER T
## 12  LATIN SMALL LETTER T, LATIN SMALL LETTER H
## 13                               LATIN SMALL LETTER V
## 14                               LATIN SMALL LETTER W
## 15                               LATIN SMALL LETTER Y
##
## $warnings
## NULL

```

[more to follow]