Package 'qlcData'

August 29, 2016

Type Package

Title Processing Data for Quantitative Language Comparison (QLC)

Version 0.1.0

Date 2015-10-20

Author Michael Cysouw

Maintainer Michael Cysouw <cysouw@mac.com>

Description This is a collection of functions to read, recode, and transcode data.

License GPL-3

Imports stringi (>= 0.2-5), yaml (>= 2.1.11)

Encoding UTF-8

Suggests knitr

VignetteBuilder knitr

NeedsCompilation no

Repository CRAN

Date/Publication 2015-10-21 10:06:14

R topics documented:

qlcData-package *Processing data for quantitative language comparison (QLC)*

Description

The package offers various functions to read, transcode and process data. There are many different function to read in data. Also a general framework to recode nominal data is included. Further, there is a general approach to describe orthographic systems through so-called Orthography Profiles. It offers functions to write such profiles based on some actual written text, and to test and correct profiles given concrete data. The main end-use is to produce tokenized texts in so-called tailored grapheme clusters.

Details

Package:	qlcData
Type:	Package
Version:	0.1.0
Date:	2015-10-20
License:	GPL-3

Various functions to read specific data formats of QLC are documented in `read.align`, `read.profile`, `read.recoding`.

The `recode` function allows for an easy and transparent way to specify a recoding of an existing nominal dataset. The specification of the recoding-decisions is preferably saved in an easily accessible YAML-file. There are utility function `write.profile` for writing and reading such files included.

For processing of strings using orthography profiles, the central function is `tokenize`. A basic sceleton for an orthography profile can be produced with `write.profile`

Author(s)

Michael Cysouw <cysouw@mac.com>

expandValues *Internal helper*

Description

produce combinations of nominal variables

Usage

```
expandValues(attributes, data)
```

Arguments

attributes	a list of attributes to be recoded. Vectors (as elements of the list) are possible to specify combinations of attributes to be recoded as a single complex attribute.
data	a data frame with nominal data, attributes as columns, observations as rows.

Details

Just a helper.

Value

expandValues is an internal help function to show the various value-combinations when combining attributes.

Author(s)

Michael Cysouw

---

join.align *Join various multialignments into one combined dataframe*

Description

Multialignments are mostly stored in separate files per cognateset. This function can be used to bind together a list of multialignments read by read.align.

Usage

```
join.align(alignments)
```

Arguments

alignments	A list of objects as returned from read.align.

Details

The alignments have to be reordered for this to work properly. Also, duplicate data (i.e. multiple words from the same language) will be removed. Simply the first occurrence is retained. This is not ideal, but it is currently the best and easiest solution.

Value

The result will be a dataframe with doculects as rows and alignments as columns.

Author(s)

Michael Cysouw <cysouw@mac.com>

`read.align` *Reading different versions of linguistic multialignments.*

Description

Multialignments of strings are a central step for historical linguistics (quite similar to multialignments in bioinformatics). There is no consensus (yet) about the file-structure for multialignments in linguistics. Currently, this functions offers to read various flavours of multialignment, trying to harmonize the internal R-structure.

Usage

```
read.align(file, flavor)
```

Arguments

`file`	Multialignment to be read
`flavor`	Currently two flavours are implemented `"PAD"` and `"BDPA"`

Details

The flavor `"PAD"` refers to the Phonetische Atlas Deutschlands, which provides multialignments for german dialects. The flavor `"BDPA"` refers to the Benchmark Database for Phonetic Alignments.

Value

Multialignment-files often contain various different kinds of information. An attempt is made to turn the data into a list with the following elements:

`meta`	: Metadata
`align`	: The actual alignments as a dataframe. When IDs are present in the original file, they are used as rownames. Some attempt is made to add useful column names.
`doculects`	: The rows of the alignment normally are some kind of doculects ("languages", "dialects"). However, because these doculects might occur more than once (when two different, but cognate words from a languages are included) these names are not used as rownames of `$align`, but presented separately here.
`annotations`	: The columns of a multialignment can have annotations, e.g. metathesis or orthographic standard. These annotations are saved here as a dataframe with the same number of columns as the `$align` dataframe. The name of the annotation is put in the rownames.

Author(s)

Michael Cysouw <cysouw@mac.com>

References

BDPA is available at http://alignments.lingpy.org. PAD is available at http://github.com/cysouw/PAD/

---

recode *Recoding nominal data*

Description

Nominal data ('categorical data') are data that consist of attributes, and each attribute consists of various discrete values ('types'). The different values that are distinguished in comparative linguistics are mostly open to debate, and different scholars like to make different decisions as to the definition of values. The recode function allows for an easy and transparent way to specify a recoding of an existing dataset.

Usage

```
recode(data, recoding)
```

Arguments

data	a data frame with nominal data, attributes as columns, observations as rows.
recoding	a recoding data structure, specifying the decisions of the recoding. It can also be a path to a file containing the specifications in YAML format. See Details.

Details

Recoding nominal data is normally considered too complex to be performed purely within R. It is possible to do it completely within R, but it is proposed here to use an external YAML document to specify the decisions that are taken in the recoding. The typical process of recoding will be to use write.recoding to prepare a skeleton that allows for quick and easy YAML-specification of a recoding. Or a YAML-recoding is written manually using various shortcuts (see below), and read.recoding is used to turn it into a full-fledged recoding that can also be used to document the decisions made. The function recode then combines the original data with the recoding, and produces a recoded dataframe.

The recoding data structure in the YAML document basically consists of a list of recodings, each of which describes a new attribute, based on one or more attributes from the original data. Each new attribute is described by:

- *attribute*: the new attribute name.
- *values*: a character vector with the new value names.
- *link*: a numeric vector with length of the original number of values. Each entry specifies the number of the new value. Zero can be used for any values that should be ignored in the new attribute.

- *recodingOf*: the name(s) of the original attribute that forms the basis of the recoding. If there are multiple attributes listed, then the new attribute will be a combination of the original attributes.
- *OriginalValues*: a character vector with the value names from the original attribute. These are only added to the template to make it easier to specify the recoding. In the actual recoding the listing in this file will be ignored. It is important to keep the ordering as specified, otherwise the linking will be wrong. The ordering of the values follows the result of `levels`, which is determined by the current locale.

There is a vignette available with detailed information about the process of recoding, check `recoding nominal data`.

Value

`recode` returns a data frame with the recoded attributes

Author(s)

Michael Cysouw <cysouw@mac.com>

References

Cysouw, Michael, Jeffrey Craig Good, Mihai Albu and Hans-Jörg Bibiko. 2005. Can GOLD "cope" with WALS? Retrofitting an ontology onto the World Atlas of Language Structures. *Proceedings of E-MELD Workshop 2005*, http://emeld.org/workshop/2005/papers/good-paper.pdf

See Also

The World Atlas of Language Structure (WALS) contains typical data that most people would very much like to recode before using for further analysis. See Cysouw et al. 2005 for a discussion of various issues surrounding the WALS data.

`tokenize` *Tokenization and transliteration of character strings based on an orthography profile*

Description

To process strings it is often very useful to tokenise them into graphemes (i.e. functional units of the orthography), and possibly replace those graphemes by other symbols to harmonize the orthographic representation of different orthographic representations ('transcription/transliteration'). As a quick and easy way to specify, save, and document the decisions taken for the tokenization, we propose using an orthography profile.

This function is the main function to produce, test and apply orthography profiles.

Usage

```
tokenize(strings,
  profile = NULL, transliterate = NULL,
  method = "global", ordering = c("size", "context", "reverse"),
  sep = " ", sep.replace = NULL, missing = "\u2047", normalize = "NFC",
  regex = FALSE, silent = FALSE,
  file.out = NULL)
```

Arguments

strings	Vector of strings to the tokenized. It is also possibly to pass a filename, which will then simply be read as `scan(strings, sep = "\n", what = "character")`.
profile	Orthography profile specifying the graphemes for the tokenization, and possibly any replacements of the available graphemes. Can be a reference to a file or an R object. If NULL then the orthography profile will be created on the fly using the defaults of `write.profile`.
transliterate	Default `NULL`, meaning no transliteration is to be performed. Alternatively, specify the name of the column in the orthography profile that should be used for replacement.
method	Method to be used for parsing the strings into graphemes. Currently two options are implemented: `global` and `linear`. See Details for further explanation.
ordering	Method for ordering. Currently three different methods are implemented, which can be combined (see Details below): `size`, `context`, `reverse` and `frequency`. Use `NULL` to prevent ordering and use the top to bottom order as specified in the orthography profile.
sep	Separator to be inserted between graphemes. Defaults to space. This function assumes that the separator specified here does not occur in the data. If it does, unexpected things might happen. Consider removing the chosen seperator from your strings first, e.g. by using gsub.
sep.replace	Sometimes, the chosen separator (see above) occurs in the strings to be parsed. This is technically not a problem, but the result might show unexpected sequences. When `sep.replace` is specified, this marking is inserted in the string at those places where the `sep` marker occurs. Typical usage in linguistics would be `sep = " ", sep.replace = "#"` adding spaces between graphemes and replacing spaces in the input string by hashes in the output string.
missing	Character to be inserted at transliteration when no transliteration is specified. Defaults to DOUBLE QUESTION MARK at U+2047. Change this when this character appears in the input string.
normalize	Which normalization to use before tokenization, defaults to "NFC". Other option is "NFD". Any other input will result in no normalisation being performed.
regex	Logical: when `regex = FALSE` internally the matching of graphemes is done exact, i.e. without using regular expressions. When `regex = TRUE` ICU-style regular expression (see stringi-search-regex) are used, so any reserved characters have to be escaped in the orthography profile. Specifically, add a slash "\"

before any occurrence of the characters `[](){}|+*.-!?^$\` in your profile (except of course when these characters are used in their regular expression meaning).

Note that this parameter also influences whether contexts should be considered in the tokenization (internally, contextual searching uses regular expressions). By default, when `regex = FALSE`, context is ignored. If `regex = TRUE` then the function checks whether there are columns called `Left` (for the left context) and `Right` (for the right context), and optionally a column called `Class` (for the specification of grapheme-classes) in the orthography profile. These are hard-coded column-names, so please adapt your orthography profile accordingly. The columns `Left` and `Right` allow for regular expression to specify context.

`silent` Logical: by default missing characters in the strings are reported with a warning. use `silent = TRUE` to supress these warnings.

`file.out` Filename for results to be written. No suffix should be specified, as various different files with different suffixes are produced (see Details below). When `file.out` is specified, then the data is written to disk AND the R dataframe is returned invisibly.

Details

Given a set of graphemes, there are at least two different methods to tokenize strings. The first is called `global` here: this approach takes the first grapheme, matches this grapheme globally at all places in the string, and then turns to the next string. The other approach is called `linear` here: this approach walks through the string from left to right. At the first character it looks through all graphemes whether there is any match, and then walks further to the end of the match and starts again. In some special cases these two methods can lead to different results (see Examples for an example).

The ordering or the lines in the ortography profile is of crucial importance, and different orderings will lead to radically different results. To simply use the top to bottom ordering as specified in the profile, use `order = NULL`. Currently, there are four different ordering strategies implemented: `size`, `context`, `reverse` and `frequency`. By specifying more than one in a vector, these orderings are used to break ties, e.g. `c("size, frequency", "reverse")` will first order by size, and for those with the same size, it will order by frequency. For lines that are still tied (i.e. the have the same size and frequency) the order will reverse order as attested in the profile. Reversing order can be useful, because hand-written profiles tend to put general rules before specific rules, which mostly should be applied in reverse order.

- `size`: order the lines in the profile by the size of the grapheme, largest first. Size is measured by number of Unicode characters after normalization as specified in the option `normalize`. For example, é has a size of 1 with `normalize = "NFC"`, but a size of 2 with `normalize = "NFD"`.
- `context`: order the lines by whether they have any context specified, lines with context coming first. Note that this only works when the option `context = TRUE` is also chosen.
- `reverse`: order the lines from bottom to top.
- `frequency`: order the lines by the frequency with which they match in the specified strings before tokenization, least frequent coming first. This frequency of course depends crucially on the available strings, so it will lead to different orderings when applied to different data. Also note that this frequency is (necessarily) measured before graphemes are identified, so

these ordering frequencies are not the same as the final frequencies shown in the outpur. Frequency of course also strongly differs on whether context is used for the matching through `context = TRUE`.

Value

Without specificatino of `file.out`, the function `tokenize` will return a list of three:

`strings`	a dataframe with the original and the tokenized/transliterated strings
`profile`	a dataframe with the graphemes with added frequencies. The dataframe is ordered according to the order that resulted from the specifications in `ordering`.
`errors`	a dataframe with all original strings that contain unmatched parts.
`missing`	a dataframe with the graphemes that are missing from the original orthography profilr, as indicated in the errors. Note that the report of missing characters does currently not lead to correct results for transliterated strings.

When `file` is specified, these three tables will be written to three different tab-separated files (with header lines): `file_strings.tsv` for the strings, `file_profile.tsv` for the orthrography profile, `file_errors.tsv` for the strings that have unidentifyable parts, and `file_missing.tsv` for the graphemes that seem to be missing. When there is nothing missing, then no file for the missing strings is produced.

Note

When `regex = TRUE`, regular expressions are acceptable in the columns 'Grapheme', 'Left' and 'Right'. Backreferences in the transliteration column are not possible (yet). When regular expressions are allowed, all literal uses of special regex-characters have to be escaped! Any literal occurrence of the following characters has then to be preceded by a backslash `\` .

- - (U+002D, HYPHEN-MINUS)
- ! (U+0021, EXCLAMATION MARK)
- ? (U+003F, QUESTION MARK)
- . (U+002E, FULL STOP)
- ((U+0028, LEFT PARENTHESIS)
-) (U+0029, RIGHT PARENTHESIS)
- \[(U+005B, LEFT SQUARE BRACKET)
- \] (U+005D, RIGHT SQUARE BRACKET)
- { (U+007B, LEFT CURLY BRACKET)
- | (007C, VERTICAL LINE)
- } (U+007D, RIGHT CURLY BRACKET)
- * (U+002A, ASTERISK)
- \ (U+005C, REVERSE SOLIDUS)
- ^ (U+02C6, MODIFIER LETTER CIRCUMFLEX ACCENT)
- + (U+002B, PLUS SIGN)

- $ (U+0024, DOLLAR SIGN)

Note that overlapping matching does not (yet) work with regular expressions. That means that for example "aa" is only found once in "aaa". In some special cases this might lead to problems that might have to be explicitly specified in the profile, e.g. a grapheme "aa" with a left context "a". See examples below. This problem arises because overlap is only available in literal searches stri_opts_fixed, but the current function uses regex-searching, which does not catch overlap stri_opts_regex.

Author(s)

Michael Cysouw <cysouw@mac.com>

References

Moran & Cysouw (forthcoming)

See Also

See also write.profile for preparing a skeleton orthography profile.

Examples

```
# simple example with interesting warning and error reporting
# the string might look like "AABB" but it isn't...
(string <- "\u0041\u0410\u0042\u0412")
tokenize(string,c("A","B"))

# make an ad-hoc orthography profile
profile <- cbind(
    Grapheme = c("a","ä","n","ng","ch","sch"),
    Trans = c("a","e","n","N","x","sh"))
# tokenization
tokenize(c("nana", "änngschä", "ach"), profile)
# with replacements and a warning
tokenize(c("Naná", "änngschä", "ach"), profile, transliterate = "Trans")

# different results of ordering
tokenize("aaa", c("a","aa"), order = NULL)
tokenize("aaa", c("a","aa"), order = "size")

# regexmatching does not catch overlap, which can lead to wrong results
# the second example results in a warning instead of just parsing "ab bb"
# this should occur only rarely in natural language
tokenize("abbb", profile = c("ab","bb"), order = NULL)
tokenize("abbb", profile = c("ab","bb"), order = NULL, regex = TRUE)

# different parsing methods can lead to different results
# note that in natural language this is VERY unlikely to happen
tokenize("abc", c("bc","ab","a","c"), order = NULL, method = "global")$strings
tokenize("abc", c("bc","ab","a","c"), order = NULL, method = "linear")$strings
```

write.profile *Writing (and reading) of an orthography profile skeleton*

Description

To process strings, it is often very useful to tokenise them into graphemes (i.e. functional units of the orthography), and possibly replace those graphemes by other symbols to harmonize the orthographic representation of different orthographic representations ('transcription'). As a quick and easy way to specify, save, and document the decisions taken for the tokenization, we propose using an orthography profile.

Provided here is a function to prepare a skeleton for an orthography profile. This function takes some strings and lists detailed information on the Unicode characters in the strings.

Usage

```
write.profile(strings,
    normalize = NULL, info = TRUE, editing = FALSE, sep = NULL,
    file.out = NULL, collation.locale = NULL)

read.profile(profile)
```

Arguments

strings	A vector of strings on which to base the orthography profile. It is also possibly to pass a filename, which will then simply be read as `scan(strings, sep = "\n", what = "character")`.
normalize	Should any unicode normalization be applied before making a profile? By default, no normalization is applied, giving direct feedback on the actual encoding as observed in the strings. Other options are `NFC` and `NFD`. In combination with `sep` these options can lead to different insights into the structure of your strings (see examples below).
info	Add columns with Unicode information on the graphemes: Unicode code points, Unicode names, and frequency of occurrence in the input strings.
editing	Add empty columns for further editing of the orthography profile: left context, right context, class, and translitation. See `tokenize` for detailed information on their usage.
sep	separator to separate the strings. When NULL (by default), then unicode character definitions are used to split (as provided by UCI, ported to R by `stringi::stri_split_boundaries`. When `sep` is specified, strings are split by this separator. Often useful is `sep = ""` to split by unicode codepoints (see examples below).
file.out	Filename for writing the profile to disk. When `NULL` the profile is returned as an R dataframe consisting of strings. When `file.out` is specified (as a path to a file), then the profile is written to disk and the R dataframe is returned invisibly.
collation.locale	Specify to ordering to be used in writing the profile. By default it uses the ordering as specified in the current locale (check `Sys.getlocale("LC_COLLATE")`).

`profile` An orthography profile to be read. Has to be a tab-delimited file with a header. There should be at least a column called "Grapheme".

Details

String are devided into default grapheme clusters as defined by the Unicode specification. Underlying code is due to the UCI as ported to R in the `stringi` package.

Value

A dataframe with strings representing a skeleton of an orthography profile.

Author(s)

Michael Cysouw <cysouw@mac.com>

References

Moran & Cysouw (forthcoming)

See Also

`tokenize`

Examples

```
# produce statistics, showing two different kinds of "A"s in Unicode.
# look at the output of "example" in the console to get the point!
(example <- "\u0041\u0391\u0410")
write.profile(example)

# note the differences. Again, look at the example in the console!
(example <- "\u00d9\u00da\u00db\u0055\u0300\u0055\u0301\u0055\u0302")
# default settings
write.profile(example)
# split according to unicode codepoints
write.profile(example, sep = "")
# after NFC normalization unicode codepoints have changed
write.profile(example, normalize = "NFC", sep = "")
# NFD normalization gives yet another structure of the codepoints
write.profile(example, normalize = "NFD", sep = "")
# note that NFC and NFD normalization are identical under unicode character definitions!
write.profile(example, normalize = "NFD")
write.profile(example, normalize = "NFC")
```

`write.recoding` *Reading and writing of recoding files.*

Description

Nominal data ('categorical data') are data that consist of attributes, and each attribute consists of various discrete values ('types'). The different values that are distinguished in comparative linguistics are mostly open to debate, and different scholars like to make different decisions as to the definition of values. The `recode` function allows for an easy and transparent way to specify a recoding of an existing dataset. The current functions help with the preparations and usage of recoding specifications.

Usage

```
write.recoding(attributes, data, file, yaml = TRUE)
read.recoding(recoding, file = NULL, data = NULL)
```

Arguments

`data`	a data frame with nominal data, attributes as columns, observations as rows.
`recoding`	a `recoding` data structure, specifying the decisions of the recoding. It can also be a path to a file containing the specifications in YAML format. See Details.
`attributes`	a list of attributes to be recoded. Vectors (as elements of the list) are possible to specify combinations of attributes to be recoded as a single complex attribute.
`file`	file in which the recoding should be written.
`yaml`	the recoding template is by default written to a file in YAML format. When `yaml=FALSE`, the template is not converted to YAML, but returned inside R as a nested list.

Details

Recoding nominal data is normally considered too complex to be performed purely within R. It is possible to do it completely within R, but it is proposed here to use an external YAML document to specify the decisions that are taken in the recoding. The typical process of recoding will be to use `write.recoding.template` to prepare a skeleton that allows for quick and easy YAML-specification of a recoding. Or a YAML-recoding is written manually using various shortcuts (see below), and `read.recoding` is used to turn it into a full-fledged recoding that can also be used to document the decisions made. The function `recode` then combines the original data with the recoding, and produces a recoded dataframe.

The `recoding data structure` in the YAML document basically consists of a list of recodings, each of which describes a new attribute, based on one or more attributes from the original data. Each new attribute is described by:

- *attribute*: the new attribute name.
- *values*: a character vector with the new value names.

- *link*: a numeric vector with length of the original number of values. Each entry specifies the number of the new value. Zero can be used for any values that should be ignored in the new attribute.
- *recodingOf*: the name(s) of the original attribute that forms the basis of the recoding. If there are multiple attributes listed, then the new attribute will be a combination of the original attributes.
- *OriginalValues*: a character vector with the value names from the original attribute. These are only added to the template to make it easier to specify the recoding. In the actual recoding the listing in this file will be ignored. It is important to keep the ordering as specified, otherwise the linking will be wrong. The ordering of the values follows the result of `levels`, which is determined by the current locale.

For writing recodings by hand, there are various shortcuts allowed:

- the names `attributes`, `values`, etc. can be abbreviated. The first letter should be sufficient.
- the `recodingOf` can be the full name of the attribute in the original data, or simply a number of the column in the data frame.
- the specification of `attribute` and `values` can be left out, although the result will be uninformative names like 'Att1' and 'Val1'.
- it is also possible to add an item `doNotRecode` with a vector of original attribute names (or column numbers). These original attributes will then be included unchanged in the recoded data table.

A minimal recoding consist thus of a specification of `recodingOf` and `link`. Without `link` nothing will be recoded. Omitting `recodingOf` will lead to an error.

There is a vignette available with detailed information about the process of recoding, check `recoding nominal data`.

Value

`write.recoding.template` by default (when `yaml=TRUE`) writes a YAML structure to the specified file. When `yaml=FALSE` the same structure is returned inside R as a nested list.

`read.recoding` either reads a recoding from file, or a list structure within R, and cleans up all the shortcuts used. The output is by default a list structure to be used in `recode`, though it is also possible to write the result to a YAML-file (when `file` is specified). When `data` is specified, the output will be embelished with all the original names from the original data, which makes for an even better documentation of the recoding.

Author(s)

Michael Cysouw <cysouw@mac.com>

References

Cysouw, Michael, Jeffrey Craig Good, Mihai Albu and Hans-Jörg Bibiko. 2005. Can GOLD "cope" with WALS? Retrofitting an ontology onto the World Atlas of Language Structures. *Proceedings of E-MELD Workshop 2005*, http://emeld.org/workshop/2005/papers/good-paper.pdf

See Also

The World Atlas of Language Structure (WALS) contains typical data that most people would very much like to recode before using for further analysis. See Cysouw et al. 2005 for a discussion of various issues surrounding the WALS data.

Index