Package 'tau'

July 2, 2014

Version 0.0-18

Encoding UTF-8

Title Text Analysis Utilities

Description Utilities for text analysis

Suggests tm

License GPL-2

Author Christian Buchta [aut],Kurt Hornik [aut, cre],Ingo Feinerer [aut],David Meyer [aut]

Maintainer Kurt Hornik <Kurt.Hornik@R-project.org>

Depends R (>= 2.10)

NeedsCompilation yes

Repository CRAN

Date/Publication 2014-06-11 15:06:12

R topics documented:

encoding *Adapt the (Declared) Encoding of a Character Vector*

Description

Functions for testing and adapting the (declared) encoding of the components of a vector of mode `character`.

Usage

```
is.utf8(x)
is.ascii(x)
is.locale(x)

translate(x, recursive = FALSE, internal = FALSE)
fixEncoding(x, latin1 = FALSE)
```

Arguments

`x`	a vector (of character).
`recursive`	option to process list components.
`internal`	option to use internal translation.
`latin1`	option to assume `"latin1"` if the declared encoding is `"unknown"`.

Details

`is.utf8` tests if the components of a vector of character are true UTF-8 strings, i.e. contain one or more valid UTF-8 multi-byte sequence(s).

`is.locale` tests if the components of a vector of character are in the encoding of the current locale.

`translate` encodes the components of a vector of `character` in the encoding of the current locale. This includes the `names` attribute of vectors of arbitrary mode. If `recursive = TRUE` the components of a `list` are processed. If `internal = TRUE` multi-byte sequences that are invalid in the encoding of the current locale are changed to literal hex numbers (see FIXME).

`fixEncoding` sets the declared encoding of the components of a vector of character to their correct or preferred values. If `latin1 = TRUE` strings that are not valid UTF-8 strings are declared to be in `"latin1"`. On the other hand, strings that are true UTF-8 strings are declared to be in `"UTF-8"` encoding.

Value

The same type of object as `x` with the (declared) encoding possibly changed.

Note

Currently `translate` uses `iconv` and therefore is not guaranteed to work on all platforms.

Author(s)

Christian Buchta

References

FIXME PCRE, RFC 3629

See Also

Encoding and iconv.

Examples

```
## Note that we assume R runs in an UTF-8 locale
text <- c("aa", "a\xe4")
Encoding(text) <- c("unknown", "latin1")
is.utf8(text)
is.ascii(text)
is.locale(text)
## implicit translation
text
##
t1 <- iconv(text, from = "latin1", to = "UTF-8")
Encoding(t1)
## oops
t2 <- iconv(text, from = "latin1", to = "utf-8")
Encoding(t2)
t2
is.locale(t2)
##
t2 <- fixEncoding(t2)
Encoding(t2)
## explicit translation
t3 <- translate(text)
Encoding(t3)
```

---

readers *Read Byte or Character Strings*

Description

Read byte or character strings from a connection.

Usage

```
readBytes(con)
readChars(con, encoding = "")
```

Arguments

con	a connection object or a character string naming a file.
encoding	encoding to be assumed for input.

Details

Both functions first read the raw bytes from the input connection into a character string. readBytes then sets the Encoding of this to "bytes"; readChars uses iconv to convert from the specified input encoding to UTF-8 (replacing non-convertible bytes by their hex codes).

Value

For readBytes, a character string marked as "bytes". For readChars, a character string marked as "UTF-8" if containing non-ASCII characters.

See Also

Encoding

---

textcnt *Term or Pattern Counting of Text Documents*

Description

This function provides a common interface to perform typical term or pattern counting tasks on text documents.

Usage

```
textcnt(x, n = 3L, split = "[[:space:][:punct:][:digit:]]+",
        tolower = TRUE, marker = "_", words = NULL, lower = 0L,
        method = c("ngram", "string", "prefix", "suffix"),
        recursive = FALSE, persistent = FALSE, useBytes = FALSE,
        perl = TRUE, verbose = FALSE, decreasing = FALSE)

## S3 method for class 'textcnt'
format(x, ...)
```

Arguments

x	a (list of) vector(s) of character representing one (or more) text document(s).
n	the maximum number of characters considered in ngram, prefix, or suffix counting (for word counting see details).
split	the regular expression pattern (PCRE) to be used in word splitting (if NULL, do nothing).
tolower	option to transform the documents to lowercase (after word splitting).

marker	the string used to mark word boundaries.
words	the number of words to use from the beginning of a document (if NULL, all words are used).
lower	the lower bound for a count to be included in the result set(s).
method	the type of counts to compute.
recursive	option to compute counts for individual documents (default all documents).
persistent	option to count documents incrementally.
useBytes	option to process byte-by-byte instead of character-by-character.
perl	option to use PCRE in word splitting.
verbose	option to obtain timing statistics.
decreasing	option to return the counts in decreasing order.
...	further (unused) arguments.

Details

The following counting methods are currently implemented:

`ngram` Count all word n-grams of order 1,...,n.

`string` Count all word sequence n-grams of order n.

`prefix` Count all word prefixes of at most length n.

`suffix` Count all word suffixes of at most length n.

The n-grams of a word are defined to be the substrings of length `n = min(length(word), n)` starting at positions 1,...,`length(word)-n`. Note that the value of `marker` is pre- and appended to word before counting. However, the empty word is never marked and therefore not counted. Note that `marker = "\1"` is reserved for counting of an efficient set of ngrams and `marker = "\2"` for the set proposed by Cavnar and Trenkle (see references).

If `method = "string"` word-sequences of and only of length n are counted. Therefore, documents with less than n words are omitted.

By default all documents are preprocessed and counted using a single C function call. For large document collections this may come at the price of considerable memory consumption. If `persistent = TRUE` and `recursive = TRUE` documents are counted incrementally, i.e., into a persistent prefix tree using as many C function calls as there are documents. Further, if `persistent = TRUE` and `recursive = FALSE` the documents are counted using a single call but no result is returned until the next call with `persistent = FALSE`. Thus, `persistent` acts as a switch with the counts being accumulated until release. Timing statistics have shown that incremental counting can be order of magnitudes faster than the default.

Be aware that the character strings in the documents are translated to the encoding of the current locale if the encoding is set (see `Encoding`). Therefore, with the possibility of `"unknown"` encodings when in an `"UTF-8"` locale, or invalid `"UTF-8"` strings declared to be in `"UTF-8"`, the code checks if each string is a valid `"UTF-8"` string and stops if not. Otherwise, strings are processed bytewise without any checks. However, embedded `nul` bytes are always removed from a string. Finally, note that during incremental counting a change of locale is not allowed (and a change in method is not recommended).

Note that the C implementation counts words into a prefix tree. Whereas this is highly efficient for n-gram, prefix, or suffix counting it may be less efficient for simple word counting. That is, implementations which use hash tables may be more efficient if the dictionary is large.

`format.textcnt` pretty prints a named vector of counts (see below) including information about the rank and encoding details of the strings.

Value

Either a single vector of counts of mode `integer` with the names indexing the patterns counted, or a list of such vectors with the components corresponding to the individual documents. Note that by default the counts are in prefix tree (byte) order (for `method = "suffix"` this is the order of the reversed strings). Otherwise, if `decreasing = TRUE` the counts are sorted in decreasing order. Note that the (default) order of ties is preserved (see `sort`).

Note

The C functions can be interrupted by `CTRL-C`. This is convenient in interactive mode but comes at the price that the C code cannot clean up the internal prefix tree. This is a known problem of the R API and the workaround is to defer the cleanup to the next function call.

The C code calls `translateChar` for all input strings which is documented to release the allocated memory no sooner than when returning from the `.Call`/`.External` interface. Therefore, in order to avoid excessive memory consumption it is recommended to either translate the input data to the current locale or to process the data incrementally.

`useBytes` may not be fully functional with R versions where `strsplit` does not support that argument.

If `useBytes = TRUE` the character strings of `names` will never be declared to be in an encoding.

Author(s)

Christian Buchta

References

W.B. Cavnar and J.M. Trenkle (1994). N-Gram Based Text Categorization. In Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval, 161–175.

Examples

```
## the classic
txt <- "The quick brown fox jumps over the lazy dog."

##
textcnt(txt, method = "ngram")
textcnt(txt, method = "prefix", n = 5L)

r <- textcnt(txt, method = "suffix", lower = 1L)
data.frame(counts = unclass(r), size = nchar(names(r)))
format(r)
```

```
## word sequences
textcnt(txt, method = "string")

## inefficient
textcnt(txt, split = "", method = "string", n = 1L)

## incremental
textcnt(txt, method = "string", persistent = TRUE, n = 1L)
textcnt(txt, method = "string", n = 1L)

## subset
textcnt(txt, method = "string", words = 5L, n = 1L)

## non-ASCII
txt <- "The quick br\xfcn f\xf6x j\xfbmps \xf5ver the lazy d\xf6\xf8g."
Encoding(txt) <- "latin1"
txt

## implicit translation
r <- textcnt(txt, method = "suffix")
table(Encoding(names(r)))
r
## efficient sets
textcnt("is",     n = 3L, marker = "\1")
textcnt("is",     n = 4L, marker = "\1")
textcnt("corpus", n = 5L, marker = "\1")
## CT sets
textcnt("corpus", n = 5L, marker = "\2")
```

`translate` *Translate Unicode Latin Ligatures*

Description

Translate Unicode "Latin ligature" characters to their respective constituents.

Usage

```
translate_Unicode_latin_ligatures(x)
```

Arguments

x	a character vector in UTF-8 encoding.

Details

In typography, a ligature occurs where two or more graphemes are joined as a single glyph. (See http://en.wikipedia.org/wiki/Typographic_ligature for more information.)

Unicode (http://www.unicode.org/) lists the following "Latin" ligatures:

Code	Name
0132	LATIN CAPITAL LIGATURE IJ
0133	LATIN SMALL LIGATURE IJ
0152	LATIN CAPITAL LIGATURE OE
0153	LATIN SMALL LIGATURE OE
FB00	LATIN SMALL LIGATURE FF
FB01	LATIN SMALL LIGATURE FI
FB02	LATIN SMALL LIGATURE FL
FB03	LATIN SMALL LIGATURE FFI
FB04	LATIN SMALL LIGATURE FFL
FB05	LATIN SMALL LIGATURE LONG S T
FB06	LATIN SMALL LIGATURE ST

`translate_Unicode_latin_ligatures` translates these to their respective constituent characters.

`util` *Preprocessing of Text Documents*

Description

Functions for common preprocessing tasks of text documents,

Usage

```
tokenize(x, lines = FALSE, eol = "\n")
remove_stopwords(x, words, lines = FALSE)
```

Arguments

`x`	a vector of character.
`eol`	the end-of-line character to use.
`words`	a vector of character (tokens).
`lines`	assume the components are lines of text.

Details

`tokenize` is a simple regular expression based parser that splits the components of a vector of character into tokens while protecting infix punctuation. If `lines = TRUE` assume `x` was imported with `readLines` and end-of-line markers need to be added back to the components.

`remove_stopwords` removes the tokens given in `words` from `x`. If `lines = FALSE` assumes the components of both vectors contain tokens which can be compared using `match`. Otherwise, assumes the tokens in `x` are delimited by word boundaries (including infix punctuation) and uses regular expression matching.

Value

The same type of object as `x`.

Author(s)

Christian Buchta

Examples

```
txt <- "\"It's almost noon,\" it@dot.net said."
## split
x <- tokenize(txt)
x
## reconstruct
t <- paste(x, collapse = "")
t

if (require("tm", quietly = TRUE)) {
    words <- readLines(system.file("stopwords", "english.dat",
                       package = "tm"))
    remove_stopwords(x, words)
    remove_stopwords(t, words, lines = TRUE)
} else
    remove_stopwords(t, words = c("it", "it's"), lines = TRUE)
```

Index