Using the koRpus Package for Text Analysis

m.eik michalke

May 8, 2017

The R package `koRpus` aims to be a versatile tool for text analysis, with an emphasis on scientific research on that topic. It implements dozens of formulae to measure readability and lexical diversity. On a more basic level `koRpus` can be used as an R wrapper for third party products, like the tokenizer and POS tagger TreeTagger or language corpora of the Leipzig Corpora Collection. This vignette takes a brief tour around its core components, shows how they can be used and gives some insight on design decisions.

1 What is koRpus?

Work on `koRpus` started in February 2011, primarily with the goal in mind to examine how similar different texts are. Since then, it quickly grew into an R package which implements dozens of formulae for readability and lexical diversity, and wrappers for language corpus databases and a tokenizer/POS tagger.

2 Recommendations

2.1 TreeTagger

At the very beginning of almost every analysis with this package, the text you want to examine has to be sliced into its components, and the components must be identified and named. That is, it has to be split into its semantic parts (tokens), words, numbers, punctuation marks. After that, each token will be tagged regarding its part-of-speech (POS). For both of these steps, `koRpus` can use the third party software TreeTagger (Schmid, 1994).[1] Especially for Windows users installation of TreeTagger might be a

[1] `http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/DecisionTreeTagger.html`

little more complex – e. g., it depends on Perl[2], and you need a tool to extract .tar.gz archives[3]. Detailed installations instructions are beyond the scope of this vignette.

If you don't want to use TreeTagger, `koRpus` provides a simple tokenizer of its own called `tokenize()`. While the tokenizing itself works quite well, `tokenize()` is not as elaborate as is TreeTagger when it comes to POS tagging, as it can merely tell words from numbers, punctuation and abbreviations. Although this is sufficient for most readability formulae, you can't evaluate word classes in detail. If that's what you want, a TreeTagger installation is needed.

2.2 Word lists

Some of the readability formulae depend on special word lists (like Dale & Chall, 1948; Bormuth, 1968; Spache, 1953). For copyright reasons these lists are not included as of now. This means, as long as you don't have copies of these lists, you can't calculate these particular measures, but of course all others. The expected format to use a list with this package is a simple text file with one word per line, preferably in UTF-8 encoding.

2.3 Language corpora

The frequency analysis functions in this package can look up how often each word in a text is used in its language, given that a corpus database is provided. Databases in Celex format are support, as is the Leipzig Corpora Collection (Quasthoff, Richter, & Biemann, 2006) file format. To use such a database with this package, you simply need to download one of the .zip/.tar files.

2.4 Translated Human Rights Declaration

If you want to estimate the language of a text, reference texts in known languages are needed. In `koRpus`, the Universal Declaration of Human Rights with its more that 350 translations[4] is used.

3 A sample session

From now on it is assumed that the above requirements are correctly installed and working. If an optional component is used it will be noted. Further, we'll need a sample text to analyze. We'll use the section on defense mechanisms of Phasmatodea from Wikipedia[5] for this purpose.

[2]For a free implementation try `http://strawberryperl.com`
[3]Like `http://7-zip.org`
[4]`http://www.unicode.org/udhr/download.html`
[5]`http://en.wikipedia.org/wiki/Phasmatodea#Defense_mechanisms`

3.1 Tokenizing and POS tagging

As explained earlier, splitting the text up into its basic components can be done by TreeTagger. To achieve this and have the results available in R, the function `treetag()` is used.

3.1.1 `treetag()`

At the very least you must provide it with the text, of course, and name the language it is written in. In addition to that you must specify where you installed TreeTagger. If you look at the package documentation you'll see that `treetag()` understands a number of options to configure TreeTagger, but in most cases using one of the built-in presets should suffice. TreeTagger comes with batch/shell scripts for installed languages, and the presets of `treetag()` are basically just R implementations of these scripts.

```
> tagged.text <- treetag("~/docs/sample_text.txt", treetagger="manual",
    lang="en", TT.options=list(path="~/bin/treetagger/", preset="en"))
```

The first argument (file name) and `lang` should explain themselves. The `treetagger` option can either take the full path to one of the original TreeTagger scripts mentioned above, or the keyword "manual", which will cause the interpretation of what is defined by `TT.options`. To use a preset, just put the `path` to your local TreeTagger installation and a valid `preset` name here.[6]

The resulting S4 object is of a class called `kRp.tagged`. For this class of objects, `koRpus` provides some comfortable methods to extract the portions you're interested in. For example, the main results are to be found in the slot `TT.res`. In addition to TreeTagger's original output (token, tag and lemma) `treetag()` also automatically counts letters, assigns tokens to global word classes and explains the rather cryptic POS tags. To get to these results, use the getter method `taggedText()`:

```
> taggedText(tagged.text)
```

```
       token tag     lemma lttr      wclass                                desc
[...]
30        an  DT        an    2  determiner                          Determiner
31    attack  NN    attack    6        noun              Noun, singular or mass
32       has VBZ      have    3        verb Verb, 3rd person singular present
33      been VBN        be    4        verb               Verb, past participle
34 initiated VBN  initiate    9        verb               Verb, past participle
35         (   (         (    1 punctuation                     Opening bracket
36 secondary  JJ secondary    9   adjective                           Adjective
37   defense  NN   defense    7        noun              Noun, singular or mass
```

[6] As of 0.04-38, English, French, Italian, German, Spanish and Russian are implemented, refer to package documentation. Additional languange support is possible by installing the respective `koRpus.lang.*` package, e. g. from `https://reaktanz.de/R/`

```
38         )   )         )    1 punctuation                    Closing bracket
[...]
```

Once you've come this far, i. e., having a valid object of class `kRp.tagged`, all following analyses should run smoothly.

Troubleshooting If `treetag()` should fail, you should first re-run it with the extra option `debug=TRUE`. Most interestingly, that will print the contents of `sys.tt.call`, which is the TreeTagger command given to your operating system for execution. With that it should be possible to examine where exactly the erroneous behavior starts.

3.1.2 Alternative: `tokenize()`

If you don't need detailed word class analysis, you should be fine using `koRpus`' own function `tokenize()`. As you can see, `tokenize()` comes to the same results regarding the tokens, but is rather limited in recognizing word classes:

```
> tagged.text <- tokenize("~/docs/sample_text.txt", lang="en")
> taggedText(tagged.text)
```

```
       token tag     lemma lttr      wclass                              desc
[...]
30        an word.kRp          2        word             Word (kRp internal)
31    attack word.kRp          6        word             Word (kRp internal)
32       has word.kRp          3        word             Word (kRp internal)
33      been word.kRp          4        word             Word (kRp internal)
34 initiated word.kRp          9        word             Word (kRp internal)
35         (     (kRp          1 punctuation Opening bracket (kRp internal)
36 secondary word.kRp          9        word             Word (kRp internal)
37   defense word.kRp          7        word             Word (kRp internal)
38         )     )kRp          1 punctuation Closing bracket (kRp internal)
[...]
```

3.1.3 Accessing data from `koRpus` objects

In case you want to access a subset of the data in the resulting object, e. g., only the column with number of letters or the first five rows, you'll be happy to know there's special `[` and `[[` methods for these kinds of objects:

```
> tagged.text[["lttr"]]
```

```
  [1]  7 10 11  7  7 10  3  7  4  9  4  4  7  2  6  4  9  2  3  5  5  1  7  7  1
 [26]  3  3  8  5  2  6  3  4  9  1  9  7  1  1  3  7  9  4  7 12  4 11  2 10  1
 [51]  4  8  3  5  3 11 11  3  5  2  6  3  6  1  3  3  6  2  4  7  1  4  2  2  9
 [76]  3  8  9  1  3  7  2  5  2  9 10  4 10  5  8  1  4  7  4  3  7  2  6  5  2
[101]  5 12  5  1  2 10  1  2 11  1  1  2  1  7 10 10  2 10  7  1  1  6  2  7  4
[...]
```

```
> tagged.text[1:5,]
```

```
        token      tag lemma lttr wclass                desc stop stem
1     Defense word.kRp     7   word Word (kRp internal)   NA   NA
2  mechanisms word.kRp    10   word Word (kRp internal)   NA   NA
3 Phasmatodea word.kRp    11   word Word (kRp internal)   NA   NA
4     species word.kRp     7   word Word (kRp internal)   NA   NA
5     exhibit word.kRp     7   word Word (kRp internal)   NA   NA
```

3.1.4 Descriptive statistics

All results of both `treetag()` and `tokenize()` also provide various descriptive statistics calculated from the analyzed text. You can get them by calling `describe()` on the object. If you deal with these for the first time, it's a good idea to first look at its structure:

```
> str(describe(tagged.text))
```

Amongst others, you will find several indices describing the number of characters:

`all.chars`: Counts each character, including all space characters

`normalized.space`: Like `all.chars`, but clusters of space characters (incl. line breaks) are counted only as one character

`chars.no.space`: Counts all characters except any space characters

`letters`: Counts only letters, excluding(!) digits (which are counted seperately as `digits`)

You'll also find the number of `words` and `sentences`, as well as average word and sentence lengths, and tables describing how the word length is distributed throughout the text:

```
> describe(tagged.text)[["lttr.distrib"]]
```

```
                1         2         3         4         5         6
num      19.000000  92.00000  74.00000  80.00000  51.000000  49.00000
cum.sum  19.000000 111.00000 185.00000 265.00000 316.000000 365.00000
cum.inv 537.000000 445.00000 371.00000 291.00000 240.000000 191.00000 [...]
pct       3.417266  16.54676  13.30935  14.38849   9.172662   8.81295
cum.pct   3.417266  19.96403  33.27338  47.66187  56.834532  65.64748
pct.inv  96.582734  80.03597  66.72662  52.33813  43.165468  34.35252
```

For instance, we can learn that the text has 74 words with three letters, 185 with three or less, and 371 with more than three. The last three lines show the percentages, respectively.

3.2 Lexical diversity (type token ratios)

To analyze the lexical diversity of our text we can now simply hand over the tagged text object to the `lex.div()` function:[7]

```
> lex.div(tagged.text)

Language: "en"
TTR.char: Calculate TTR values
 reached token 1 to 50...
 reached token 1 to 100...
 reached token 1 to 150...
 reached token 1 to 200...
[...]
Total number of tokens: 556
Total number of types:  294

Type-Token Ratio
   TTR: 0.53

TTR characteristics:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.5297  0.5443  0.5895  0.6139  0.6429  1.0000


Mean Segmental Type-Token Ratio
               MSTTR: 0.72
        Segment size: 100
      Tokens dropped: 56

Hint: A segment size of 92 would reduce the drop rate to 4.
      Maybe try ?segment.optimizer()
[...]
HD-D
   HD-D: 0.85

HD-D characteristics:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
 0.7677  0.8436  0.8463  0.8426  0.8504  0.8574  8.0000


Measure of Textual Lexical Diversity
                MTLD: 97.18
```

[7] Please note that as of version 0.04-18, the correctness of some of these calculations has not been extensively validated yet. The package was released nonetheless, also to find outstanding bugs in the implemented measures. Any information on the validity of its results is very welcome!

```
   Number of factors: 5.72
         Factor size: 0.72
    SD tokens/factor: 36.67 (all factors)
                      29.27 (complete factors only)

MTLD characteristics:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  14.00   76.14   86.00   81.96   93.21  103.60    1.00
```

The above output is only a small sample and really much longer. Let's look at some particular parts: At first we are informed of the language, which is read from the tagged object. Following that is a growing feedback stream, letting you know how far calculations of a measures' characteristics[8] have progressed. This was added because if all possible measures are being calculated and the text is rather long, this can take quite some time, and it might be uplifting to see that R didn't just freeze. After that the actual results are being printed, using the package's `show()` method for this particular kind of object. As you can see, it prints the actual value of each measure before a summary of the characteristics[9].

Some measures return more information than only their actual index value. For instance, when the Mean Segmental Type-Token Ratio is calculated, you'll be informed how much of your text was dropped and hence not examined. A small feature tool of `koRpus`, `segment.optimizer()`, automatically recommends you with a different segment size if this could decrease the number of lost tokens.

By default, `lex.div()` calculates every measure of lexical diversity that was implemented. Of course this is fully configurable, e. g. to completely skip the calculation of characteristics just add the option `char=NULL`. If you're only interested in one particular measure, it might be more convenient to call the according wrapper function instead of `lex.div()`. For example, to calculate the measures proposed by Maas (1972):

```
> maas(tagged.text)

Language: "en"

Total number of tokens: 556
Total number of types:  294

Maas' Indices
       a: 0.19
    lgV0: 5.64
   lgeV0: 12.99
```

[8] Characteristics can be looked at to examine each measure's dependency on text length. They are calculated by computing each measure repeatedly, beginning with only the first token, then adding the next, progressing until the full text was analyzed.

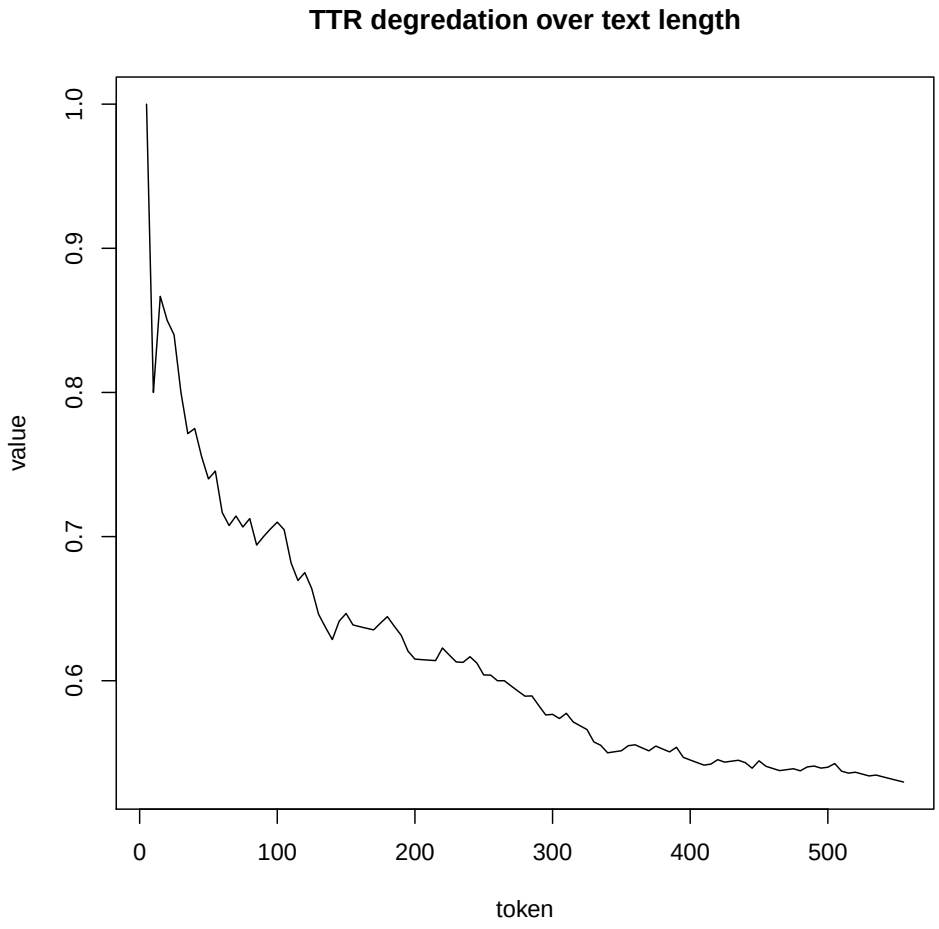
[9] For informtaion on the measures shown see Tweedie and Baayen (1998); McCarthy and Jarvis (2007, 2010).

```
Relative vocabulary growth (first half to full text)
       a: 0.79
    lgV0: 6.93
      V': 0.43 (43 new types every 100 tokens)
```

All wrapper functions have characteristics turned off by default. The following example demonstrates how to calculate the classic type-token ratio with characteristics:

```
> ttr.res <- TTR(tagged.text, char=TRUE)
> plot(ttr.res@TTR.char, type="l", main="TTR degredation over text length")
```

The plot then shows the typical degredation of TTR values with increasing text length:



Since this package is intended for research, it should be possible to directly influence all relevant values of each measure and examine the effects. For example, as mentioned before `segment.optimizer()` recommended a change of segment size for MSTTR to drop less words, which is easily done:

```
> MSTTR(tagged.text, segment=92)

Language: "en"

Total number of tokens: 556
Total number of types:  294

Mean Segmental Type-Token Ratio
               MSTTR: 0.75
        Segment size: 92
      Tokens dropped: 4
```

Please see to the documentation for more detailed information on the available measures and their references.

3.3 Frequency analysis

3.3.1 Importing language corpora data

This package has rudimentary support to import corpus databases.[10] That is, it can read frequency data for words into an R object and use this object for further analysis. Next to the Celex[11] database format (`read.corp.celex()`), it can read the LCC flatfile format footnoteActually, it unterstands two different LCC formats, both the older .zip and the newer .tar archive format. (`read.corp.LCC()`). The latter might be of special interest, because the needed database archives can be freely downloaded.[12] Once you've downloaded one of these archives, it can be comfortably imported:

```
> LCC.en <- read.corp.LCC("~/downloads/corpora/eng_news_2010_1M-text.tar")
```

`read.corp.LCC()` will automatically extract the files it needs from the archive. Alernatively, you can specify the path to the unpacked archive as well. To work with the imported data directly, the tool `query()` was added to the package. It helps you to comfortably look up certain words, or ranges of interesting values:

[10] The package also has a function called `read.corp.custom()` which can be used to process language corpora yourself, and store the results in an object of class `kRp.corp.freq`, which is the class returned by `read.corp.LCC()` and `read.corp.celex()` as well. That is, if you can't get any already analyzed corpus database but have a huge language corpus at hand, you can create your own frequency database. But be warned that depending on corpus size and your hardware, this might take ages. On the other hand, `read.corp.custom()` will provide inverse document frequency (idf) values for all types, which is necessary to compute tf-idf with `freq.analysis()`

[11] `http://celex.mpi.nl`

[12] `http://corpora.informatik.uni-leipzig.de/download.html`

```
> query(LCC.en, "word", "what")

    num word  freq         pct pmio    log10 rank.avg rank.min rank.rel.avg
160 210 what 16396 0.000780145  780 2.892095   260759   260759     99.95362
    rank.rel.min
160     99.95362

> query(LCC.en, "pmio", c(780, 790))

    num  word  freq          pct pmio    log10 rank.avg rank.min rank.rel.avg
156 206  many 16588 0.0007892806  789 2.897077   260763   260763     99.95515
157 207   per 16492 0.0007847128  784 2.894316   260762   260762     99.95477
158 208  down 16468 0.0007835708  783 2.893762   260761   260761     99.95439
159 209 since 16431 0.0007818103  781 2.892651   260760   260760     99.95400
160 210  what 16396 0.0007801450  780 2.892095   260759   260759     99.95362
    rank.rel.min
156     99.95515
157     99.95477
158     99.95439
159     99.95400
160     99.95362
```

3.3.2 Conduct a frequency analysis

We can now conduct a full frequency analysis of our text:

```
> freq.analysis.res <- freq.analysis(tagged.text, corp.freq=LCC.en)
```

The resulting object holds a lot of information, even if no corpus data was used (i. e., `corp.freq=NULL`). To begin with, it contains the two slots `TT.res` and `lang`, which are copied from the analyzed tagged text object. In this way analysis results can always be converted back into `kRp.tagged` objects.[13] However, if corpus data was provided, the tagging results gained three new columns:

```
> taggedText(freq.analysis.res)

       token tag     lemma lttr  [...] pmio rank.avg rank.min
[...]
30        an  DT        an    2        3817 99.98735 99.98735
31    attack  NN    attack    6         163 99.70370 99.70370
32       has VBZ      have    3        4318 99.98888 99.98888
33      been VBN        be    4        2488 99.98313 99.98313
34 initiated VBN  initiate    9          11 97.32617 97.32137
35         (   (         (    1         854 99.96013 99.96013
36 secondary  JJ secondary    9          21 98.23846 98.23674
```

[13] This can easily be done by calling `as(freq.analysis.res, "kRp.tagged")`.

```
37   defense  NN   defense    7         210 99.77499 99.77499
38         )   )         )    1         856 99.96052 99.96052
[...]
```

Perhaps most informatively, `pmio` shows how often the respective token appears in a million tokens, according to the corpus data. Adding to this, the previously introduced slot `desc` now contains some more descriptive statistics on our text, and if we provided a corpus database, the slot `freq.analysis` lists summaries of various frequency information that was calculated.

If the corpus object also provided inverse document frequency (i. e., values in column `idf`) data, `freq.analysis()` will automatically compute tf-idf statistics and put them in a column called `tfidf`.

3.3.3 New to the `desc` slot

Amongst others, the descriptives now also give easy access to character vectors with all words (`$all.words`) and all lemmata (`$all.lemmata`), all tokens sorted into word classes (e. g., all verbs in `$classes$verb`)[14], or the number of words in each sentece:

```
> describe(freq.analysis.res)[["sentc.length"]]

 [1] 34 10 37 16 44 31 14 31 34 23 17 43 40 47 22 19 65 29
```

As a practical example, the list `$classes` has proven to be very helpful to debug the results of TreeTagger, which is remarkably accurate, but of course not free from making a mistake now and then. By looking through `$classes`, where all tokens are grouped regarding to the global word class TreeTagger attributed to it, at least obvious errors (like names mistakenly taken for a pronoun) are easily found:[15]

```
> describe(freq.analysis.res)$classes

$conjunction
 [1] "both" "and"  "and"  "and"  "and"  "or"   "or"   "and"  "and"  "or"
[11] "and"  "or"   "and"  "or"   "and"  "and"  "and"  "and"

$number
[1] "20"  "one"

$determiner
 [1] "an"      "the"     "an"      "The"     "the"     "the"     "some"
 [8] "that"    "Some"    "the"     "a"       "a"       "a"       "the"
[15] "that"    "the"     "the"     "Another" "which"   "the"     "a"
[22] "that"    "a"       "The"     "a"       "the"     "that"    "a"
[...]
```

[14] This sorting depends on proper POS-tagging, so this will only contain useful data if you used `treetag()` instead of `tokenize()`.

[15] And can then be corrected by using the function `correct.tag()`

3.4 Readability

The package comes with implementations of several readability formulae. Some of them depend on the number of syllables in the text.[16] To achieve this, the method `hyphen()` takes objects of class `kRp.tagged` and applies an hyphenation algorithm (Liang, 1983) to each word. This algorithm was originally developed for automatic word hyphenation in LaTeX, and is gracefully misused here to fulfill a slightly different service.[17]

```
> hyph.txt.en <- hyphen(tagged.text)
```

This seperate hyphenation step can actually be skipped, as `readability()` will do it automatically if needed. But similar to TreeTagger, `hyphen()` will most likely not produce perfect results. As a rule of thumb, if in doubt it seems to behave rather conservative, that is, is underestimates the real number of syllables in a text. This, however, would of course affect the results of several readability formulae.

So, the more accurate the end results should be, the less you should rely on the automatic hyphenation alone. But it sure is a good starting point, for there is a function called `correct.hyph()` to help you clean these results of errors later on. The most comfortable way to do this is to call `hyphenText(hyph.txt.en)`, which will get you a data frame with two colums, `word` (the hyphenated words) and `syll` (the number of syllables), in a spread sheet editor:[18]

```
>  hyphenText(hyph.txt.en)

   syll     word
[...]
20    1    first
21    1    place
22    1  primary
23    2 de-fense
24    1      and
[...]
```

You can then manually correct wrong hyphenations by removing or inserting "-" as hyphenation indicators, and call the function without further arguments, which will cause it to recount all syllables:

```
> hyph.txt.en <- correct.hyph(hyph.txt.en)
```

Of course the function can also be used to alter entries on its own:

```
> hyph.txt.en <- correct.hyph(hyph.txt.en, word="primary", hyphen="pri-ma-ry")
```

[16] Whether this is the case can be looked up in the documentation.

[17] The `hyphen()` method was originally implemented as part of the `koRpus` package, but was later split off into its own package called `sylly`.

[18] For example, this can be comfortably done with RKWard: `http://rkward.kde.org`

```
Changed

   syll    word
22    1 primary

  into

   syll      word
22    3 pri-ma-ry
```

The hyphenated text object can now be given to `readability()`, to calculate the measures of interest:[19]

```
> readbl.txt <- readability(tagged.text, hyphen=hyph.txt.en, index="all")
```

Similar to `lex.div()`, by default `readability()` calculates almost[20] all available measures:

```
> readbl.txt

Flesch Reading Ease
  Parameters: en (Flesch)
          RE: 39.76
       Grade: >= 13 (college)


Flesch.PSK Reading Ease
  Parameters: Powers-Sumner-Kearl
       Grade: 7.5
         Age: 12.5
[...]
Gunning Frequency of Gobbledygook (FOG)
  Parameters: Powers-Sumner-Kearl
       Score: 7.39
[...]
Coleman Formulas
  Parameters: default
    Pronouns: 1.62 (per 100 words)
     Prepos.: 13.31 (per 100 words)
   Formula 1: 39% cloze completions
   Formula 2: 37% cloze completions
   Formula 3: 35% cloze completions
```

[19] Please note that as of version 0.04-18, the correctness of some of these calculations has not been extensively validated yet. The package was released nonetheless, also to find outstanding bugs in the implemented measures. Any information on the validity of its results is very welcome!

[20] Measures which rely on word lists will be skipped if no list is provided.

```
   Formula 4: 36% cloze completions
[...]
```

To get a more condensed overview of the results try the `summary()` method:

```
> summary(readbl.txt)

Text language: en
                   index                      flavour    raw               grade  age
1                 Flesch                   en (Flesch)  39.76     >= 13 (college)
2                 Flesch           Powers-Sumner-Kearl                      7.5 12.5
3                 Flesch                   de (Amstad)  55.26 >= 10 (high school)
4                 Flesch          es (Fernandez-Huerta)  79.07                   7
5                 Flesch              fr (Kandel-Moles)  64.38                 8-9
6                 Flesch                    nl (Douma)  73.11                   7
7         Flesch-Kincaid                                                    15.39
8  Farr-Jenkins-Paterson                                33.76
9  Farr-Jenkins-Paterson           Powers-Sumner-Kearl                      7.37
10                  FOG                                                     18.9
11                  FOG           Powers-Sumner-Kearl                       7.39
12                  FOG                 New FOG (NRI)                        19
13          Coleman-Liau                                    32               14.26
14                 SMOG                                                     15.97
15                 SMOG de ("Qu", Bamberger-Vanecek)                        10.32
16                  LIX                                  65.24
17                  RIX                                  10.61      > 12 (college)
[...]
```

If you're interested in a particular formula, again a wrapper function might be more convenient:

```
> flesch.res <- flesch(tagged.text, hyphen=hyph.txt.en)
> lix.res <- LIX(tagged.text)   # LIX doesn't need syllable count
> lix.res

Läsbarhetsindex (LIX)
  Parameters: default
       Index: 65.24

Text language: en
```

3.4.1 Readability from numeric data

It is possible to calculate the readability measures from the relevant key values directly, rather than analyze an actual text, by using `readability.num()` instead of `readability()`. If you need to reanalyze a particular text, this can be considerably faster. Therefore, all objects returned by `readability()` can directly be fed to `readability.num()`, since all relevant data is present in the `desc` slot.

3.5 Language detection

Another feature of this package is the detection of the language a text was (most probably) written in. This is done by gzipping reference texts in known languages, gzipping them again with addition of a small sample of the text in unknown language, and determining the case where the additional sample causes the smallest increase in file size (as described in Benedetto, Caglioti, & Loreto, 2002). By default, the compressed objects will be created in memory only.

To use the function `guess.lang()`, you first need to download the reference material. In this implementation, the Universal Declaration of Human Rights in unicode formatting is used, because the document holds the world record of beeing the text translated into the most languages, and is publicly available[21]. Please get the zipped archive with all translations in .txt format. You can, but don't have to unzip the archive. The text to find the language of must also be in a unicode .txt file:

```
> guess.lang("~/docs/sample_text.txt", udhr.path="~/downloads/udhr_txt.zip")

  Estimated language: English
          Identifier: en
             Country: UK (Europe)

377 different languages were checked.
```

4 Extending koRpus

The language support in this package was designed almost modular, so with a little effort you should be able to add new languages yourself. You need the package sources for this, then basically you will have to add a new file to it and rebuild/reinstall the package. More details on this topic can be found in `inst/README.languages`. Once you got a new language to work with `koRpus`, I'd be happy to include your module in the official distribution.

5 Analyzing full corpora

Despite its name, the scope of `koRpus` is single texts. If you would like to do analysis an a full corpus of texts, have a look at the plugin package `tm.plugin.koRpus`[22].

References

Benedetto, D., Caglioti, E., & Loreto, V. (2002). Language trees and zipping. *Physical Review Letters*, *88*(4), 048702.

[21] `http://www.unicode.org/udhr/download.html`

[22] `https://reaktanz.de/R/pckg/tm.plugin.koRpus/` (package repo) or `https://github.com/unDocUMeantIt/tm.plugin.koRpus` (source code)

Bormuth, J. R. (1968). Cloze test readability: Criterion reference scores. *Journal of Educational Measurement*, *5*(3), 189–196.

Dale, E., & Chall, J. S. (1948). A formula for predicting readability. *Educational research bulletin*, 11–28.

Liang, F. M. (1983). *Word hy-phen-a-tion by com-put-er* (Unpublished doctoral dissertation). Stanford University, Dept. Computer Science, Stanford.

Maas, H. D. (1972). Über den Zusammenhang zwischen Wortschatzumfang und Länge eines Textes. *Zeitschrift f\"ur Literaturwissenschaft und Linguistik*, *2*(8), 73–79.

McCarthy, P. M., & Jarvis, S. (2007). vocd – a theoretical and empirical evaluation. *Language Testing*, *24*(4), 459–488.

McCarthy, P. M., & Jarvis, S. (2010). MTLD, vocd-D, and HD-D: a validation study of sophisticated approaches to lexical diversity assessment. *Behavior research methods*, *42*(2), 381–392.

Quasthoff, U., Richter, M., & Biemann, C. (2006). Corpus portal for search in monolingual corpora. In *Proceedings of the fifth international conference on language resources and evaluation* (pp. 1799–1802). Genoa.

Schmid, H. (1994). Probabilistic part-of-speec tagging using decision trees. In *International conference on new methods in language processing* (pp. 44–49). Manchester, UK.

Spache, G. (1953). A new readability formula for primary-grade reading materials. *The Elementary School Journal*, *53*(7), 410–413.

Tweedie, F. J., & Baayen, R. H. (1998). How variable may a constant be? measures of lexical richness in perspective. *Computers and the Humanities*, *32*(5), 323–352.