Quantitative Content Analysis: Lecture 8

Matthias Haber

05 April 2017

Today's outline

- Text analysis workflow
- Creating a text corpus
- Processing

Basic Principles

- Corpus texts are text repositories.
  - Should not have their texts modified as part of preparation or analysis
  - Subsetting or redefining documents is allowable
- A corpus should be capable of holding additional objects that will be associated with the corpus, such as dictionaries, stopword, and phrase lists, etc.
- A document-feature matrix (dfm) is a sparse matrix that is always documents in rows by features in columns
- Encoding of texts should be done in the corpus, and recorded as meta-data in the corpus
  - This encoding should be `UTF-8` by default (problem for Windows machines)

Quanteda

quanteda is an R package for managing and analyzing text, created by Kenneth Benoit, Kohei Watanabe, Paul Nulty, Adam Obeng, Haiyan Wang, Ben Lauderdale, and Will Lowe. You can install quanteda from inside RStudio, from the Tools. . . Install Packages menu, or simply using

```
install.packages("quanteda")
```

You can also install the developers version directly from Github

```
# the devtools package is required
devtools::install_github("kbenoit/quanteda")
```

Note that on Windows platforms, it is also recommended that you install the RTools suite, and for OS X, that you install XCode from the App Store.

Text analysis workflow

The goal is to simplify text and reduce dimensionality of the dfm created from it. In a nutshell, we want to filter relevant information and discard irrelevant information.

1. Creating the corpus
   - reading files
   - creating a corpus
   - adding document variables and metadata
2. Defining and delimiting documents
   - defining what are "documents" and what are "sentences"

Text analysis workflow (II)

3. Defining and delimiting textual features, using:
   - indentify instances of defined features ("tokens") and extract them as vectors
   - usually these will consist of terms, but may also consist of:
     - `ngrams` and `skipgrams`, sequences of adjacent or nearby tokens
     - multi-word expressions, through `phrasetotoken`
   - in this step we also apply rules that will keep or ignore elements, such as
     - punctuation
     - numbers, including or currency-prefixed digits
     - URLs
     - Twitter tags
     - inter-token separators

Text analysis workflow (III)

4. Further feature selection

- Once defined and extracted from the texts (the tokenization step), features may be:
  - removed or kept through use of predefined lists or patterns
    - converting to lower case
    - stemming

Analysis of documents and features

1. From a corpus.

These steps don't necessarily require the processing steps above:

- `quanteda::kwic`
- `quanteda::textstat_lexdiv`
- `summary`

Creating a corpus object

You can load texts from different file sources. The quanteda package can construct a corpus object from several input sources:

1. a character vector object

```
myTinyCorpus <- corpus(inaugTexts[1:2], notes = "G.W.")
```

2. a VCorpus object from the **tm** package

```
data(crude, package = "tm")
myTmCorpus <- corpus(crude)
```

3. a corpusSource object, created by the readtext() function from the **readtext** package (most likely scenario). Currently, you have to download the package from Github:

```
devtools::install_github("kbenoit/readtext")
```

Using `readtext()` to import texts

In the simplest case, we would like to load a set of texts in plain text files from a single directory. To do this, we use the `readtext` command, and use the 'glob' wildcard operator * to indicate that we want to load multiple files:

```
library(readtext)
myCorpus <- corpus(readtext("data/inaugural/*"))
```

Using `readtext()` to import texts (II)

Often, we have metadata encoded in the names of the files. For example, the inaugural addresses contain the year and the president's name in the name of the file. With the `docvarsfrom` argument, we can instruct the `readtext` command to consider these elements as document variables.

```
mytf <- readtext("data/inaugural/*", docvarsfrom="filenames",
                 dvsep="-",
                 docvarnames=c("Year", "President"))
inaugCorpus <- corpus(mytf)
```

Using readtext() to import texts (III)

If the texts and document variables are stored separately, we can easily add document variables to the corpus, as long as the data frame containing them is of the same length as the texts:

```
SOTUdocvars <- read.csv("data/SOTU_metadata.csv",
                        stringsAsFactors = F)
SOTUdocvars$Date <- as.Date(SOTUdocvars$Date, "%B %d, %Y")
SOTUdocvars$delivery <- as.factor(SOTUdocvars$delivery)
SOTUdocvars$type <- as.factor(SOTUdocvars$type)
SOTUdocvars$party <- as.factor(SOTUdocvars$party)
SOTUdocvars$nwords <- NULL

sotuCorpus <- corpus(readtext(file='data/sotu/*.txt',
                              encoding = "UTF-8-BOM"))
docvars(sotuCorpus) <- SOTUdocvars
```

Exploring a corpus

The quanteda packages comes with a built-in set of inaugural addresses from US Presidents. The `summary` command will output the name of each text along with the number of types, tokens and sentences contained in the text.

```
library(quanteda)
summary(data_corpus_inaugural[1:3])
##                Text Types Tokens Sentences
## 1 1789-Washington   626   1540        23
## 2 1793-Washington    96    147         4
## 3      1797-Adams   826   2584        37
oneText <- data_corpus_inaugural[1]
nchar(oneText)
## 1789-Washington
##            8618
```

Tokenization

To tokenize a text is to split it into units, most commonly words, which can be counted and to form the basis of a quantitative analysis. The quanteda function `tokenize` can be used on a character vector, a vector of character vectors, or a corpus.

```
tokens <- tokenize("Do not use semicolons. They are
transvestite hermaphrodites representing
absolutely nothing. All they do is show
you've been to college.")
vec <- c(one = 'This is text 1.',
two = 'This is the 2nd text.')
```

```
tokenize(vec)
```

Tokenization (II)

The `tokenize` function has a number useful parameters. To remove punctuation, set the `removePunct` argument to be TRUE. We can combine this with the `removeNumbers` function to also remove numbers.

```
tokenize(vec, removePunct = T, removeNumbers = T)
## tokenizedTexts from 2 documents.
## one :
## [1] "This" "is"   "text"
##
## two :
## [1] "This" "is"   "the"  "2nd"  "text"
```

Tokenization in quanteda is very *conservative*: by default, it only removes separator characters. Using this function with the inaugural addresses we can set the `what` parameter to tokenize by sentences.

```
inaugTokens <- tokenize(inaugTexts, what = "sentence")
```

Conversion to lower case

`toLower` and `toUpper` from the quanteda package are methods to change the case of character vectors. Both methods are defined for many classes of quanteda objects.

```
methods(toLower)
## [1] toLower.character*      toLower.NULL*           toLower
## [4] toLower.tokens*
## see '?methods' for accessing help and source code
s1 <- 'NASA sent a rocket into space.'

quanteda::toLower(s1)
## [1] "nasa sent a rocket into space."

quanteda::toUpper(s1)
## [1] "NASA SENT A ROCKET INTO SPACE."
```

Conversion to lower case (II)

`toLower` includes options designed to preserve acronyms:

```
char_tolower(s1, keep_acronyms = TRUE)
## [1] "NASA sent a rocket into space."
```

`toLower` is based on stringi, and is therefore nicely Unicode compliant.

```
# Russian
head(char_tolower(stopwords("russian")), 20)

# Arabic
head(char_tolower(stopwords("arabic")), 20)
```

Note: `dfm` converts to lower case by default, but this can be turned off using the `toLower = FALSE` argument.

Creating a dfm

Once each text has been split into words, we can use the `dfm` function to create a matrix of counts of the occurrences of each word in each document. `dfm()` works on a variety of object types, including character vectors, corpus objects, and tokenized text objects.

```
inaugDfm <- dfm(inaugTokens)
```

Analyzing a dfm

Methods for analyzing a dfm:

```
dfm                    print
convert                removeFeatures
docfreq                similarity
docnames               dfm_sort
featnames              textmodel
textstat_lexdiv        topfeatures
ndoc                   dfm_trim
ntoken                 weight
plot                   settings
show
```

Removing and selecting features: stopwords

We often want to delete stopwords, articles, prepositions, etc. to limit our analysis to words that carry actual meaning. For this, we can use the `remove` parameter.

```
testText <- 'The quick brown fox named Seamus jumps over
the lazy dog also named Seamus,
with the newspaper in his mouth.'

testCorpus <- corpus(testText)
featnames(dfm(testCorpus,
remove = stopwords("english")))
##  [1] "quick"     "brown"     "fox"       "named"     "seamu
##  [6] "jumps"     "lazy"      "dog"       "also"      ","
## [11] "newspaper" "mouth"     "."
```

Removing and selecting features: frequent words

```
library(dplyr)
# keep only words occuring <=10 times and
# in at most 3/4 of the documents
inaugDfm <- dfm(inaugTokens) %>%
dfm_trim(max_count = 10, max_docfreq = 0.75)
```

Removing and selecting features: other

```
# keep only words ending in s
dfm(testCorpus, select = "*s", verbose = FALSE)
## Document-feature matrix of: 1 document, 3 features (0% spar
## 1 x 3 sparse Matrix of class "dfmSparse"
##        features
## docs    seamus jumps his
##   text1      2     1   1

# # keep only hashtags
testTweets <- "My homie @justinbieber #justinbieber
shopping in #LA yesterday #beliebers"
dfm(testTweets, select = "#*", removeTwitter = FALSE)
## Document-feature matrix of: 1 document, 3 features (0% spar
## 1 x 3 sparse Matrix of class "dfmSparse"
##        features
## docs    #justinbieber #la #beliebers
```

Stemming

Stemming relies on the `SnowballC` package's implementation of the Porter stemmer, and is available for the following languages:

```
SnowballC::getStemLanguages()
##  [1] "danish"     "dutch"      "english"    "finnish"    "f
##  [6] "german"     "hungarian"  "italian"    "norwegian"  "p
## [11] "portuguese" "romanian"   "russian"    "spanish"    "s
## [16] "turkish"
```

It's not perfect:

```
r   char_wordstem(c("win", "winning", "wins", "won",
"winner"))   ## [1] "win"    "win"    "win"    "won"
"winner"
```

but it's fast.

Extract corpus objects

To extract texts from a quanteda corpus object we can use the `texts` function.

```
mytexts <- texts(subset(inaugCorpus,
President == "Washington"))
```

Adding metadata

If we are interested in analysing the texts with respect to some other variables, we can create a corpus object to associate the texts with this metadata. For example, we can use the `docvars` option to the `corpus` command to record the party with which each text is associated:

```
dv <- data.frame(Party = c('dem', 'rep', 'rep', 'dem'))
recentCorpus <- corpus(inaugTexts[53:56], docvars = dv)
summary(head(recentCorpus))
##             Text Types Tokens Sentences
## 1 1997-Clinton   773   2451       111
## 2    2001-Bush   622   1810        97
## 3    2005-Bush   772   2325       100
## 4   2009-Obama   939   2729       110
```

Combining features

We can use this metadata to combine features across documents when creating a dfm:

```
partyDfm <- dfm(recentCorpus, groups = 'Party',
ignoredFeatures = (stopwords('english')))
plot(partyDfm, comparison = TRUE)
```

Investigating character vectors

The fundamental type in which R stores text is the character vector. The most simple case is a character vector of length one. The `nchar` function returns the number of characters in a character vector.

```
s1 <- 'This is my example text'
length(s1)
## [1] 1
nchar(s1)
## [1] 23
```

The `nchar` function is vectorized, meaning that when called on a vector it returns a value for each element of the vector.

```
s2 <- c('This is', 'my example text.', 'So imaginative.')
nchar(s2)
## [1]  7 16 15
```

Investigating character vectors example

Which were the longest and shortest inaugural addresses speeches?

```
which.max(nchar(inaugTexts))
## 1841-Harrison
##            14

which.min(nchar(inaugTexts))
## 1793-Washington
##               2
```

String extraction

It is not possible to index into a string in R:

```
s1 <- 'This is a very informative example sentences.'
s1[6:9]
## [1] NA NA NA NA
```

To extract a substring, instead we use the `substr` function.

```
substr(s1, 6,9)
## [1] "is a"
```

String extraction (II)

Often we would like to split character vectors to extract a term of interest. This is possible using the `strsplit` function.

```
s1 <- 'split this string'
strsplit(s1, 'this')
## [[1]]
## [1] "split "  " string"
```

Consider the names of the inaugural texts:

```
names(inaugTexts)[1:3]
## [1] "1789-Washington" "1793-Washington" "1797-Adams"
parts <- strsplit(names(inaugTexts), '-')
years <- sapply(parts, function(x) x[1])
pres <-  sapply(parts, function(x) x[2])
```

Joining character vectors

The `paste` function is used to join character vectors together. The way in which the elements are combined depends on the values of the `sep` and `collapse` arguments:

```
paste('one','two','three')
## [1] "one two three"
paste('one','two','three', sep='_')
## [1] "one_two_three"
paste(years[1:3], pres[1:3], sep='-')
## [1] "1789-Washington" "1793-Washington" "1797-Adams"
paste(years[1:3], pres[1:3], collapse='-')
## [1] "1789 Washington-1793 Washington-1797 Adams"
```

Comparing character vectors

Character vectors can be compared using the == and %in% operators:

```
char_tolower(s1) == char_toupper(s1)
## [1] FALSE
'apples'=='oranges'
## [1] FALSE
'pears' == 'pears'
## [1] TRUE

c1 <- c('apples', 'oranges', 'pears')
'pears' %in% c1
## [1] TRUE
c2 <- c('bananas', 'pears')
c2 %in% c1
## [1] FALSE  TRUE
```

Searching and replacing within text

The base functions for searching and replacing within text are grep and gsub. The grep command tests whether a pattern occurs within a string:

```
grep('pear', 'these are oranges')
## integer(0)
grep('orange', c('apples', 'oranges', 'pears'))
## [1] 2
grep('pears', c('apples', 'oranges', 'pears'))
## [1] 3
```

The gsub command substitutes one pattern for another within a string:

```
gsub('oranges', 'apples', 'these are oranges')
## [1] "these are apples"
```

String replacement

The `stringr` and `stringi` packages provide more extensive and more organized interfaces for string manipulation.

For an overview of the most frequently used functions, see the vignette: `https://cran.r-project.org/web/packages/stringr/vignettes/stringr.html`.

```
library(stringr)
fruits <- c("one apple", "two pears", "three bananas")
str_replace(fruits, "[aeiou]", "-")
## [1] "-ne apple"     "tw- pears"     "thr-e bananas"
str_replace_all(fruits, "[aeiou]", "-")
## [1] "-n- -ppl-"     "tw- p--rs"     "thr-- b-n-n-s"
str_replace(fruits, "([aeiou])", "\\1\\1")
## [1] "oone apple"     "twoo pears"     "threee bananas"
str_replace(fruits, "[aeiou]", c("1", "2", "3"))
## [1] "1ne apple"     "tw2 pears"     "thr3e bananas"
```

Word boundaries

```
words <- c("These are some words. These are more words.")
str_count(words, boundary("word"))
## [1] 8
str_count(words, boundary("sentence"))
## [1] 2

str_split(words, boundary("sentence"))[[1]]
## [1] "These are some words. " "These are more words."
```

Trim whitespace

```
str_trim("  String with trailing and leading white space\t")
## [1] "String with trailing and leading white space"
```

Next Session

- Regular expressions
- Dictionary-based content analysis