Text mining in practice - Bag of Words - corpus creation

Digital Humanities DSC 105 Spring 2023

Marcus Birkenkrahe

January 25, 2023

README

- Creation of a text source using the `tm` package
- Creation of a volatile corpus from a vector
- Creation of a volatile corpus from a data frame
- Checking metadata and inspecting a corpus
- Building a data frame from scratch

TODO Indexing list structures

- A good way to repeat the `list` structure is this (very short) lesson of the "Introduction to R" DataCamp course.
- Lists can contain any other R object (called *elements*): matrices, vectors, data frames, or other lists.
- A list element is addressed (indexed) with the `[[ ]]` operator, while values of the list element are addressed with the `[ ]` operator as usual.
- Example: let's create a list `LIST` of the following elements
  1. a 2 x 2 `matrix` `m` with four `integer` numbers
  2. a `character` vector `foo` that contains your first names
  3. a vector `bar` with today's date

4. a `numeric` vector `baz` of the number of characters of `bar`

```
m <- matrix(data=1:4,nrow=2)
foo <- c("Marcus", "Martin", "Bernhard", "Wolfgang", "Heinrich")
bar <- date()
baz <- nchar(bar)
LIST <- list(m,"name"=foo,"date"=bar,baz)
LIST
```

```
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$name
[1] "Marcus"   "Martin"   "Bernhard" "Wolfgang" "Heinrich"

$date
[1] "Wed Jan 25 21:55:17 2023"

[[4]]
[1] 24
```

- Printing the list shows four elements - two of them are named, two are unnamed. The structure shows this, too:

```
str(LIST)
```

```
List of 4
 $     : int [1:2, 1:2] 1 2 3 4
 $ name: chr [1:5] "Marcus" "Martin" "Bernhard" "Wolfgang" ...
 $ date: chr "Wed Jan 25 21:55:17 2023"
 $     : int 24
```

- When extracting elements or values within elements, you can use names if they exist.
- How can you extract the stored date?

```
LIST[[3]] # using the element index
LIST[[3]][1] # pointing at the first value of the 3rd element
LIST$date # this works because the element is named
LIST$date[1] # pointing at the first value of the 'date' element
```

```
[1] "Wed Jan 25 21:55:17 2023"
[1] "Wed Jan 25 21:55:17 2023"
[1] "Wed Jan 25 21:55:17 2023"
[1] "Wed Jan 25 21:55:17 2023"
```

- How can you extract the 2nd of the first names, "Martin"?

```
LIST[[2]][2]
LIST$name[2]
LIST[[2]][-c(1,3,4,5)] # remove all the other names
```

```
[1] "Martin"
[1] "Martin"
[1] "Martin"
```

TODO The `tm` text mining package

- The `tm` package for text mining comes with a *vignette* (Feinerer, 2022). Its date reveals that the paper is up to date.
- Load `tm` and check the loaded package list with `search()`:

```
library(tm)
search()
```

```
 [1] ".GlobalEnv"              "package:tm"
 [3] "package:NLP"             "package:stringr"
 [5] "package:stringi"         "package:lubridate"
 [7] "package:timechange"      "package:qdap"
 [9] "package:RColorBrewer"    "package:qdapTools"
[11] "package:qdapRegex"       "package:qdapDictionaries"
[13] "ESSR"                    "package:stats"
[15] "package:graphics"        "package:grDevices"
[17] "package:utils"           "package:datasets"
[19] "package:methods"         "Autoloads"
[21] "package:base"
```

- There is no separate data package. Check which functions `tm` contains:

```
ls("package:tm")
```

```
 [1] "as.DocumentTermMatrix"   "as.TermDocumentMatrix"
 [3] "as.VCorpus"              "Boost_tokenizer"
 [5] "content_transformer"     "Corpus"
 [7] "DataframeSource"         "DirSource"
 [9] "Docs"                    "DocumentTermMatrix"
[11] "DublinCore"              "DublinCore<-"
[13] "eoi"                     "findAssocs"
[15] "findFreqTerms"           "findMostFreqTerms"
[17] "FunctionGenerator"       "getElem"
[19] "getMeta"                 "getReaders"
[21] "getSources"              "getTokenizers"
[23] "getTransformations"      "Heaps_plot"
[25] "inspect"                 "MC_tokenizer"
[27] "nDocs"                   "nTerms"
[29] "PCorpus"                 "pGetElem"
[31] "PlainTextDocument"       "read_dtm_Blei_et_al"
[33] "read_dtm_MC"             "readDataframe"
[35] "readDOC"                 "reader"
[37] "readPDF"                 "readPlain"
[39] "readRCV1"                "readRCV1asPlain"
[41] "readReut21578XML"        "readReut21578XMLasPlain"
[43] "readTagged"              "readXML"
[45] "removeNumbers"           "removePunctuation"
[47] "removeSparseTerms"       "removeWords"
[49] "scan_tokenizer"          "SimpleCorpus"
[51] "SimpleSource"            "stemCompletion"
[53] "stemDocument"            "stepNext"
[55] "stopwords"               "stripWhitespace"
[57] "TermDocumentMatrix"      "termFreq"
[59] "Terms"                   "tm_filter"
[61] "tm_index"                "tm_map"
[63] "tm_parLapply"            "tm_parLapply_engine"
[65] "tm_reduce"               "tm_term_score"
[67] "URISource"               "VCorpus"
[69] "VectorSource"            "weightBin"
[71] "WeightFunction"          "weightSMART"
```

```
[73] "weightTf"                "weightTfIdf"
[75] "writeCorpus"             "XMLSource"
[77] "XMLTextDocument"         "Zipf_plot"
[79] "ZipSource"
```

- Texts are processed at different levels:
  1. Strings like "Hello world"
  2. Documents like a text of many strings stored as vector, dataframe
  3. Corpora as collections of documents
- We'll get back to this paper when we learn to clean text data
- Use `VectorSource` to create a *source* from a *character* vector:

```
doc <- c("This is a text.", "This is another one.")
doc_source <- VectorSource(doc)
doc_source

$encoding
[1] ""

$length
[1] 2

$position
[1] 0

$reader
function (elem, language, id)
{
    if (!is.null(elem$uri))
id <- basename(elem$uri)
    PlainTextDocument(elem$content, id = id, language = language)
}
<bytecode: 0x000001c24c3ced70>
<environment: namespace:tm>

$content
[1] "This is a text."      "This is another one."
```

```
attr(,"class")
[1] "VectorSource" "SimpleSource" "Source"
```

- The source `doc_source` is a `list` of five elements and an attribute:
  1. `encoding` says that the content is encoded with apostrophs.
  2. `length = 2` is the length of the input vector
  3. `position = 0` means that there is no other document in the corpus
  4. `reader` is the function used to process the vector
  5. `content` is the content of the corpus - two strings
  6. `attr` is a vector that says what type of source this is

```
typeof(doc_source)
```

```
[1] "list"
```

- To turn the `VectorSource` into a volatile (in-memory) corpus, use `VCorpus` (also a `list`):

```
doc_corpus <- VCorpus(VectorSource(doc))
doc_corpus
typeof(doc_corpus)
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 2
[1] "list"
```

- A corpus can have metadata - this only only has two "documents", i.e. the two strings. A corpus can have thousands of documents.
- You can inspect the corpus with `inspect`. This provides information about each of the documents -

```
inspect(doc_corpus)
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 2

[[1]]
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 15

[[2]]
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 20
```

- Individual documents can be accessed with the `[[` operator or via their name:

```
meta(doc_corpus[[2]]) # metadata for document no. 2 (list index)
meta(doc_corpus[[2]],"language") # metadata for document language
```

```
  author       : character(0)
  datetimestamp: 2023-01-26 03:55:17
  description  : character(0)
  heading      : character(0)
  id           : 2
  language     : en
  origin       : character(0)
[1] "en"
```

- Accessing the corpus document content with `content`:

```
content(doc_corpus[[2]])
```

```
[1] "This is another one."
```

TODO Getting the `coffee.csv` data (again)

- Dataframes and vectors created during a session are deleted once the session is ended unless the session is stored (then they can be found in an `.RData` file) - so we need to re-import the data.

- The coffee tweets still sit in the `CSV` file. We import them into a data frame `tweets` and check that the file is okay with `str`:

```
tweets <- read.csv(file="../data/coffee.csv")
str(tweets)
```

```
'data.frame': 1000 obs. of  15 variables:
 $ num         : int  1 2 3 4 5 6 7 8 9 10 ...
 $ text        : chr  "@ayyytylerb that is so true drink lots of coffee" "RT @bryz
 $ favorited   : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
 $ replyToSN   : chr  "ayyytylerb" NA NA NA ...
 $ created     : chr  "8/9/2013 2:43" "8/9/2013 2:43" "8/9/2013 2:43" "8/9/2013 2
 $ truncated   : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
 $ replyToSID  : num  3.66e+17 NA NA NA NA ...
 $ id          : num  3.66e+17 3.66e+17 3.66e+17 3.66e+17 3.66e+17 ...
 $ replyToUID  : int  1637123977 NA NA NA NA NA NA 1316942208 NA NA ...
 $ statusSource: chr  "<a href=\"http://twitter.com/download/iphone\" rel=\"nofoll
 $ screenName  : chr  "thejennagibson" "carolynicosia" "janeCkay" "AlexandriaOOTD"
 $ retweetCount: int  0 1 0 0 2 0 0 0 1 2 ...
 $ retweeted   : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
 $ longitude   : logi  NA NA NA NA NA NA ...
 $ latitude    : logi  NA NA NA NA NA NA ...
```

  - Be mindful that this only works if the computer can find the file: in my code example, I stored it in the `data` directory, which is at the same level as the directory this Org-mode file is in, `org`:

- This Org-mode file `3_corpus.org` expects to find the R console in the buffer `*R*`. If the current working directory, which you can get with `getwd()` is not `org`, it will produce a connection error:

- Find the current working directory for your R session with `getwd`:

```
getwd()
```

```
[1] "C:/Users/birkenkrahe/Documents/GitHub/tm/org"
```

- This should be the same directory that this file is currently in:

```
shell("cd",intern=TRUE)
```

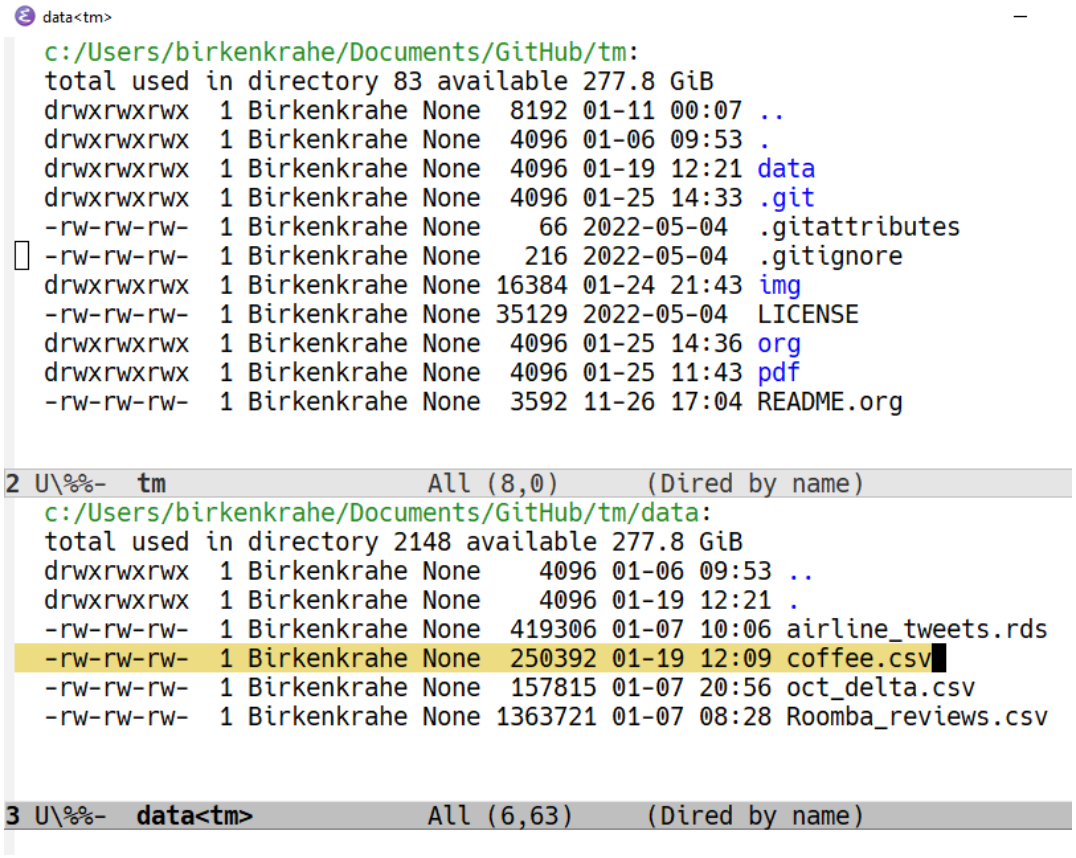

Figure 1: Directories tm and tm/data with coffee.csv

```
#+RESULTS:
: Error in file(file, "rt") : cannot open the connection
: In addition: Warning message:
: In file(file, "rt") :
:   cannot open file '../data/coffee.csv': No such file or directory
```

Figure 2: Directories tm and tm/data with coffee.csv

```
[1] "C:\\Users\\birkenkrahe\\Documents\\GitHub\\tm\\org"
```

- You can check that without reading the output of the commands:

```
identical(getwd(), # current R working directory
  gsub("\\\\", # find \\ and replace it by
       "/",    # /
       shell("cd",intern=TRUE))) # where you are
```

```
[1] TRUE
```

TODO Making a `VectorSource` from `tweets`

- Now we have the data frame and extract the `text` from it. To be sure, we print the first three tweets.

```
coffee_tweets <- tweets$text
head(coffee_tweets, n=3)
```

```
[1] "@ayyytylerb that is so true drink lots of coffee"
[2] "RT @bryzy_brib: Senior March tmw morning at 7:25 A.M. in the SENIOR lot. Get
[3] "If you believe in #gunsense tomorrow would be a very good day to have your co
```

- Now we have a vector with text. The steps to get a source are:

  1. load the `tm` package (re-loading does no harm)
  2. make a source from the vector using `VectorSource`
  3. display structure of the source

```
library(tm)
coffee_source <- VectorSource(coffee_tweets)
str(coffee_source)
```

```
Classes 'VectorSource', 'SimpleSource', 'Source'  hidden list of 5
 $ encoding: chr ""
 $ length  : int 1000
 $ position: num 0
 $ reader  :function (elem, language, id)
 $ content : chr [1:1000] "@ayyytylerb that is so true drink lots of coffee" "RT @
```

- We recognize the familiar list elements from the general explanation of the `tm` package.
- Print the first 2 tweets in `coffee_source`

```
head(coffee_source,n=2)
```

```
[1] "@ayyytylerb that is so true drink lots of coffee"
[2] "RT @bryzy_brib: Senior March tmw morning at 7:25 A.M. in the SENIOR lot. Get
```

- Print the 999th tweet in `coffee_source`

```
coffee_source$content[999] # with the list element and the index
coffee_source[[999]] # list element only
coffee_source[999]  # index only
```

```
[1] "First morning coffee after Ramadan http://t.co/ZEu6cl9qGY"
[1] "First morning coffee after Ramadan http://t.co/ZEu6cl9qGY"
[1] "First morning coffee after Ramadan http://t.co/ZEu6cl9qGY"
```

TODO Making a `VCorpus` from a vector of tweets

- Use `VCorpus`, to create a corpus `coffee_corpus` from `coffee_source`, then print `coffee_corpus`:

```
coffee_corpus <- VCorpus(coffee_source)
coffee_corpus
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 1000
```

- The corpus is a *container*, hence the content is not printed, only indicated.

TODO Accessing the corpus list with index or `content`

- Look at its structure to see how to get to the content - but not the structure of the whole thing since the metadata are overwhelming - instead only at the structure of the first list item.

```
str(coffee_corpus[[1]])
```

```
List of 2
 $ content: chr "@ayyytylerb that is so true drink lots of coffee"
 $ meta   :List of 7
  ..$ author       : chr(0)
  ..$ datetimestamp: POSIXlt[1:1], format: "2023-01-26 03:55:18"
  ..$ description  : chr(0)
  ..$ heading      : chr(0)
  ..$ id           : chr "1"
  ..$ language     : chr "en"
  ..$ origin       : chr(0)
  ..- attr(*, "class")= chr "TextDocumentMeta"
 - attr(*, "class")= chr [1:2] "PlainTextDocument" "TextDocument"
```

- Inspect the data - select the 15th tweet from the corpus:

```
inspect(coffee_corpus[[15]])
```

```
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 111

@HeatherWhaley I was about 2 joke it takes 2 hands to hold hot coffee...then I rea
```

- To extract the content of the 15th tweet in this volatile corpus, you can either use your `list` indexing powers, or use `content`:

```
coffee_corpus[[15]][1] # select list element by index and entry
coffee_corpus[[15]]["content"] # select by index and name
coffee_corpus[[15]]$content # select by name
content(coffee_corpus[[15]]) # select with content function
```

```
$content
[1] "@HeatherWhaley I was about 2 joke it takes 2 hands to hold hot coffee...then
$content
[1] "@HeatherWhaley I was about 2 joke it takes 2 hands to hold hot coffee...then
[1] "@HeatherWhaley I was about 2 joke it takes 2 hands to hold hot coffee...then
[1] "@HeatherWhaley I was about 2 joke it takes 2 hands to hold hot coffee...then
```

- How many characters does the 15th tweet have? (You already know this value from the `inspect` above):

```
nchar(coffee_corpus[[15]]$content)
```

```
[1] 111
```

TODO Making a `DataframeSource` from tweets

- Often, larger amounts of data are in dataframes (i.e. tables of vectors) rather than individual vectors.
- To demonstrate, turn the vector `coffee_tweets` into a dataframe with the function `data.frame`, and show its structure:

```
coffee_tweets.df <- data.frame(coffee_tweets)
str(coffee_tweets.df)
```

```
'data.frame': 1000 obs. of  1 variable:
 $ coffee_tweets: chr  "@ayyytylerb that is so true drink lots of coffee" "RT @bry
```

- This dataframe has one feature (`coffee_tweets.df$coffee_tweets`) and 1000 records or lines.
- However, to turn a dataframe into a source, the dataframe must have a very specific structure:
  1. Column 1 must be called `doc_id` with a unique string for each row.
  2. Column 2 must be called `text` with standard `"UTF-8"` encoding.
  3. Columns 3+ are metadata and will be retained as such
- `coffee_tweets.df` does **not** fulfil these conditions - the first column is called `coffee_tweets`. But we can reformat it:

1. add a column 1 that is called `doc_id` and contains a record ID
2. change the column name to `text`

```
df <- data.frame(
  "doc_id" = 1:1000,
  "text" = coffee_tweets.df$coffee_tweets)
str(df)
```

```
'data.frame': 1000 obs. of  2 variables:
 $ doc_id: int  1 2 3 4 5 6 7 8 9 10 ...
 $ text  : chr  "@ayyytylerb that is so true drink lots of coffee" "RT @bryzy_bril
```

- Now we're good to go for `DataframeSource`:

```
df_source <- DataframeSource(df)
str(df_source)
```

```
Classes 'DataframeSource', 'SimpleSource', 'Source'  hidden list of 5
 $ encoding: chr ""
 $ length  : int 1000
 $ position: num 0
 $ reader  :function (elem, language, id)
 $ content :'data.frame': 1000 obs. of  2 variables:
  ..$ doc_id: int [1:1000] 1 2 3 4 5 6 7 8 9 10 ...
  ..$ text  : chr [1:1000] "@ayyytylerb that is so true drink lots of coffee" "RT
```

- The source looks similar to the output of `VectorSource`, of course, except that the content is a 1000 x 2 table, not a 1000 element vector.

TODO Making a `VCorpus` from the dataframe source

- Let's turn this monster frame into a corpus and access some tweets:

```
df_corpus <- VCorpus(df_source)
df_corpus
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 1000
```

- Compare this with `coffee_corpus` that we derived from a vector:

```
coffee_corpus # got this from VCorpus(coffee_source)

<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 1000
```

TODO Checking metadata with `meta`

- The metadata for our examples, `coffee_corpus` and `df_corpus` are minimal, because we extracted the text only from the dataframe `tweets`:

```
meta(coffee_corpus)
meta(df_corpus)

data frame with 0 columns and 1000 rows
data frame with 0 columns and 1000 rows
```

- Let's construct an example dataframe with some metadata to illustrate the use of `meta`. This is the table we wish to construct - it already fulfils the conditions to build a source from a dataframe:

	doc_id	text	author	date
1	1	Text mining is a great time.	Author1	1514953399
2	2	Text analysis provides insights	Author2	1514866998
3	3	qdap and tm are used in text mining	Author3	1514780598

- We use the `data.frame` function to build this table from scratch:

```
example <-
  data.frame( "doc_id"=c(1,2,3),
     "text"=c("Text mining is a great time.",
      "Text analysis provides insights",
      "qdap and tm are used in text mining"),
     "author"=c("Author1","Author2","Author3"),
     "date"=c("1514953399","1514866998","1514780598"))
example
```

```
  doc_id                                text  author       date
1      1        Text mining is a great time. Author1 1514953399
2      2     Text analysis provides insights Author2 1514866998
3      3 qdap and tm are used in text mining Author3 1514780598
```

- Success! Now the usual steps to build our corpus:
  1. build source `list` with `DataframeSource`
  2. build volatile corpus `list` with `VCorpus`

```
example_source <- DataframeSource(example)
example_corpus <- VCorpus(example_source)
example_corpus
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 2
Content:  documents: 3
```

- Inspect the corpus with `inspect`:

```
inspect(example_corpus)
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 2
Content:  documents: 3

[[1]]
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 28

[[2]]
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 31

[[3]]
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 35
```

- Finally, extraction of the metadata with `meta`:

```
meta(example_corpus)
```

```
   author       date
1 Author1 1514953399
2 Author2 1514866998
3 Author3 1514780598
```

TODO TM Glossary - concepts and code

TERM	MEANING
`tm`	Text mining package
`[[`	List element index
`[`	Vector element index
`List[[2]][5]`	Extracts 5th value of 2nd element of `List`
`x[-n]`	Removes nth element of vector `x`
Vignette	Documentation for an R package (paper)
`ls()`	List all objects in current session
`ls('package:tm')`	List all objects in package `tm`
`tm::VectorSource`	Build source `list` from vector
`tm::VCorpus`	Build corpus `list` from source
`tm::DataframeSource`	BUild source `list` from dataframe
`data.frame`	Create data frame
`typeof`	Return R data type or data structure
`tm::inspect`	Get information about each corpus element
`tm::meta`	Extract metadata from corpus
`tm::content`	Extract `content` element from corpus `list`