Text mining in practice - Bag of Words

Digital Humanities DSC 105 Spring 2023

Marcus Birkenkrahe

January 25, 2023

README

- Text mining with Bag of Words explained
- Text mining with syntactic or semantic parsing explained
- Extended example: airline customer service tweets
- R environment information: console, packages and help
- String manipulation functions

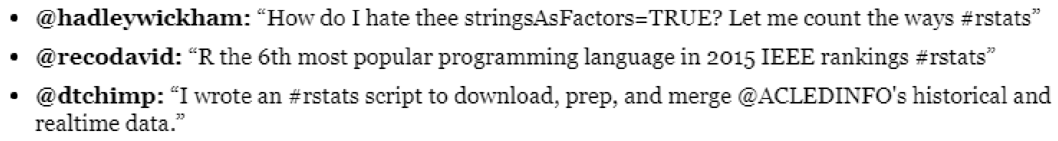
Two techniques

- Recall: tm takes unorganized sources of text and reorganizes them, then applies a sequence of analytical steps to gain insights.
- There are two types of text mining: Bag of Words and Syntactic/Semantic Parsing.
- **Bag of Words** treats every word or group of words (n-gram) as unique. It is easy to understand, can be done quickly, and provides useful input for machine learning frameworks.
- **Syntactic Parsing** is the process of analyzing the structure of a sentence to determine its grammatical structure.
- **Semantic Parsing** is similar but may be built on the task for which inference is required rather than just tag words according to their grammatical function.

Figure 1: Source: 10 practical text mining examples (2022)

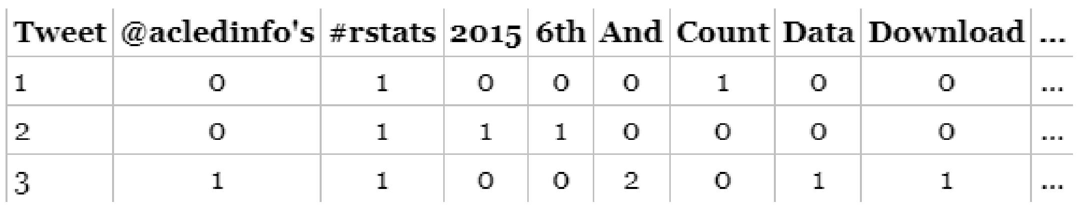
Bag of Words - simple example

- Bag of Words generates document term matrics (DTM) or their transposition (TDM).
- In a DTM, each row represents a document or individual *corpus*, e.g. a tweet, and each column represents a word. In a TDM, rows and columns are switched.
- Example: three tweets form a *corpus* or body of text for analysis



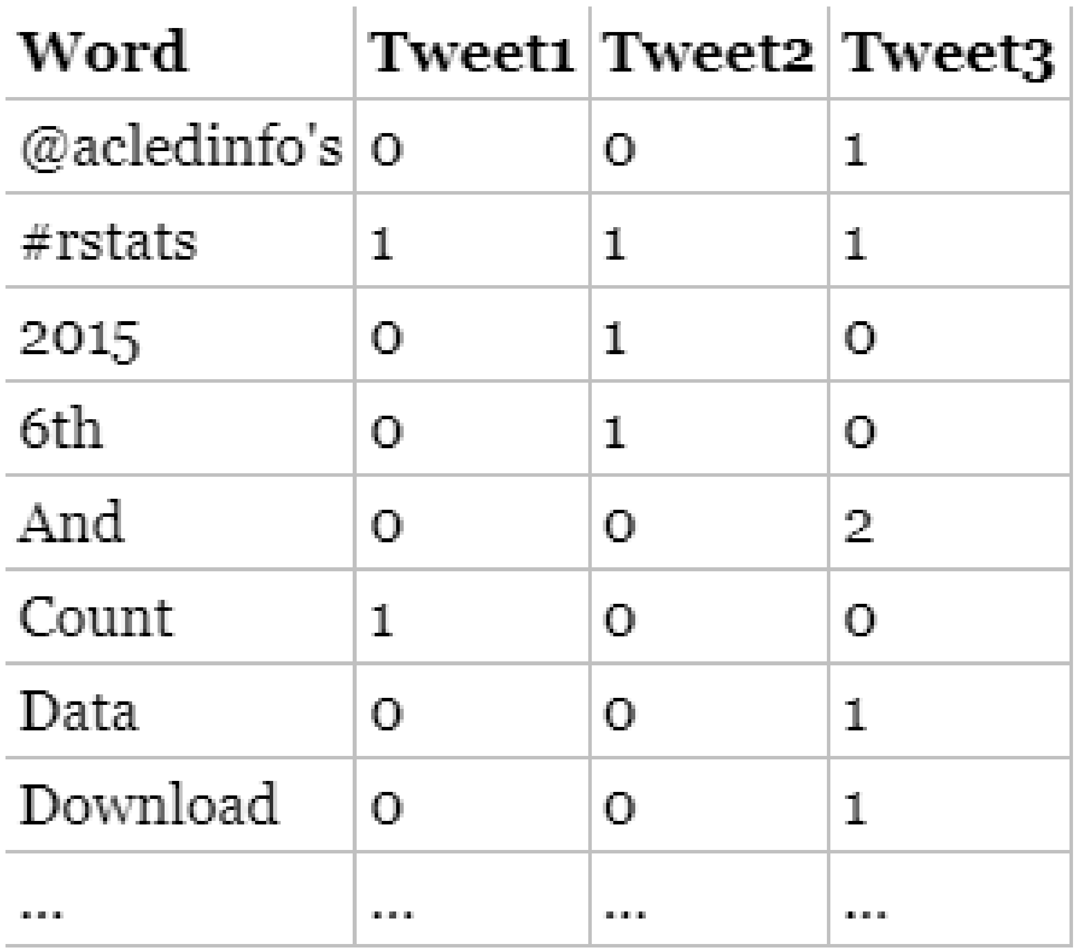
- **@hadleywickham:** “How do I hate thee stringsAsFactors=TRUE? Let me count the ways #rstats”
- **@recodavid:** “R the 6th most popular programming language in 2015 IEEE rankings #rstats”
- **@dtchimp:** “I wrote an #rstats script to download, prep, and merge @ACLEDINFO's historical and realtime data.”

- A document term matrix (DTM) for this corpus:



Tweet	@acledinfo's	#rstats	2015	6th	And	Count	Data	Download	...
1	0	1	0	0	0	1	0	0	...
2	0	1	1	1	0	0	0	0	...
3	1	1	0	0	2	0	1	1	...

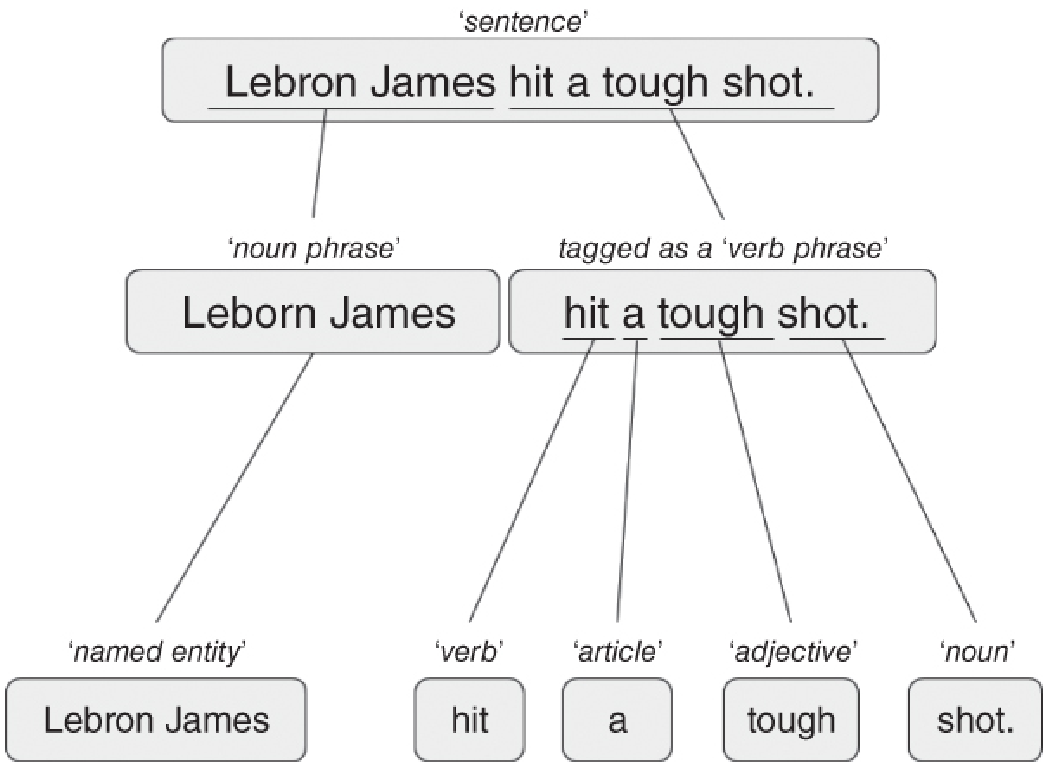
- The transposed DTM or transposed document matrix (TDM):



Word	Tweet1	Tweet2	Tweet3
@acledinfo's	0	0	1
#rstats	1	1	1
2015	0	1	0
6th	0	1	0
And	0	0	2
Count	1	0	0
Data	0	0	1
Download	0	0	1
...	...	...	...

- These DTM and TDM examples only show word counts. Now, without reading all the tweets (perhaps a much larger number than three), you can surmise that the tweets are related to R.

Syntactic parsing - simple example



- Syntactic or semantic parsing has many more attributes assigned to a sentence than Bag of Words, captures and retains more information.
- Syntactic parsing involves determining the roles that each word plays in a sentence (e.g. noun, verb, adjective, etc.) and their relations.
- It is often used as a first step in natural language processing (NLP), before more advanced analysis can be applied.
- Semantic parsing is the process of interpreting natural language input and determining its meaning.
- To do that, sentences have to be mapped to a representation, e.g. by tagging parts of speech (POS) as building blocks.
- R has a package linked to Apache OpenNLP, a library for the processing of natural language text.

- Tags are captured as *meta-data* of the original sentence.

Example: airline customer service tweets

Figure 2: Delta airline

1. **Define problem and specific goals**: understand Delta Airline's customer service tweets to launch a competitive team:
   (a) What is the average length of a social customer service reply?
   (b) What links were referenced most often in the tweets?
   (c) How large should a social media customer service team be?
2. **Identify the text that needs to be collected:** analysis is restricted to Twitter but could be expanded to other feeds.
3. **Organize the text:** The Delta tweets were taken from the Twitter API between Oct 1 to Oct 15, 2015. The original JSON object was transformed to a smaller CSV file with only tweets and date information, `oct_delta.csv`. The raw file can be downloaded online from this URL and stored in a file with this name on your PC.

4. **Extract features:** We will introduce basic string manipulation and bag of words text cleaning functions to extract the features.

5. **Analyze:** We will analyze the corpus using R functions to answer our questions.

6. **Reach insight or recommendation:**

   (a) Using `nchar` and `mean` to assess average tweet length

   (b) Using `grep`, `grepl` and `sum` to find out what links were most often referenced.

   (c) Analyze the agents' signature as a time series `ts` to find out how many people should be on a team.

Get the practice file

- To practice, we use Emacs Org-mode files. Save the practice file for this lecture from tinyurl.com/4f56t7uz to the `Downloads` directory on your PC.

- The file is probably in the directory `Downloads` and it's called `2_bag_of_words_practice.org.txt`

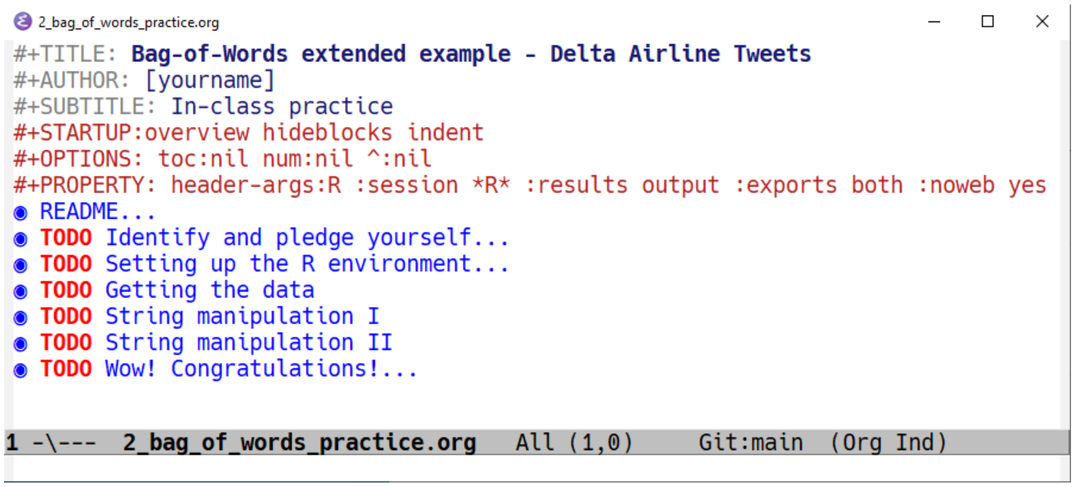
- Open the Windows CMD line terminal and at the prompt enter `cd Downloads` to get to the directory that contains the file `2_bag_of_words_practice.org.txt` (a text file).

- enter `DIR *org*` to check if the file is actually there.

Open the practice file in Emacs

- At the terminal prompt, open the GNU Emacs editor without GUI by entering `emacs -nw`

- In Emacs, find the file with `C-x C-f` and then save it as an Org-mode file with `C-x C-w` as `2_bag_of_words_practice.org` - you should see something like this:



```
#+TITLE: Bag-of-Words extended example - Delta Airline Tweets
#+AUTHOR: [yourname]
#+SUBTITLE: In-class practice
#+STARTUP:overview hideblocks indent
#+OPTIONS: toc:nil num:nil ^:nil
#+PROPERTY: header-args:R :session *R* :results output :exports both :noweb yes
◉ README...
◉ TODO Identify and pledge yourself...
◉ TODO Setting up the R environment...
◉ TODO Getting the data
◉ TODO String manipulation I
◉ TODO String manipulation II
◉ TODO Wow! Congratulations!...

1 -\---  2_bag_of_words_practice.org   All (1,0)    Git:main  (Org Ind)
```

- If you have not used Emacs on your current PC before, you now need to download this configuration file: tinyurl.com/3amrdz55 and install it.
- Open it in Emacs as the other one, with `C-x C-f` followed by the file name and rename it with `C-x C-w` to `~/.emacs` so that it is located in Emacs' home directory `~/`.
- Now either shut Emacs down (`C-x C-c`) and re-open it, or run the functions in `.emacs` with `M-x eval-buffer`.
- **If you have never used the Emacs editor before**, you should have a look at the on-board tutorial. Open it with the keyboard sequence `CTRL-h t`. It teaches you the basics and will take about 1 hour[1].

Setting up literate programming in R

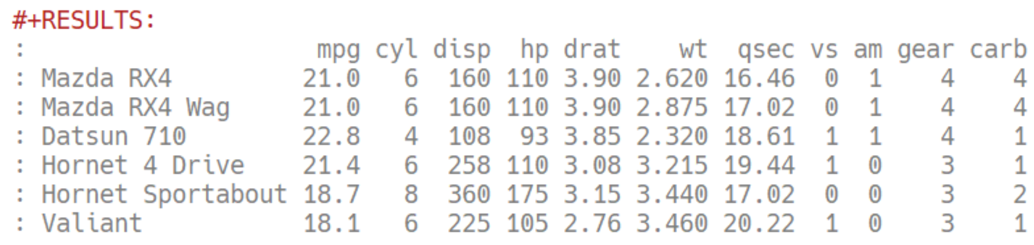
- To test your ability to combine documentation, code and output in R, create and run a code block in the Emac practice file

[1]If you need extra motivation to learn Emacs, read this or this. Alternatively (or in addition) you can take a look at this video tutorial (1 hr 12 min). I've described how to install Emacs on your PC (or Mac) in my FAQ on GitHub here. Or you can just ask me. We'll review using Emacs in class at the start but taking a good look at the tutorial will help you get started and make it more likely that you'll have fun in class!

- Create a new code block with `<s TAB R` and put two R commands into it:

```
#+begin_src R
  head(mtcars)
#+end_src
```

- You should see the output below. Otherwise you need to check your `.emacs` installation, then your R installation or your general setup.



```
#+RESULTS:
:                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
: Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
: Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
: Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
: Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
: Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
: Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Setting up the text mining environment in R

- Many R functions and packages must be installed and loaded before they can be used in an R session.

- Open a Windows CMD line terminal, enter `R` to open the R console, and then run these three commands to install string manipulation packages[2].

```
install.packages("stringi")
install.packages("stringr")
install.packages("qdap")
```

- The download results in a lengthy screen output that should complete successfully. You can check that you can work with these packages by running the following command in R:

```
library(stringi)
library(stringr)
library(qdap)
```

[2]On Linux (or Windows WSL), run `update.packages()` before to update all dependencies - since here, source code is compiled and linked.

- The output will include the information that some objects are "masked" from certain packages. This means that one and the same function name (e.g. `explain`) is used in different packages. To use `explain` from the `dplyr` package e.g., use `dplyr::explain` to resolve the ambiguity.
- To see which packages are loaded in your current R session, use `search()`:

  ```
  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"
  ```

- To access the help documentation, enter `?` followed by the name in the console window, e.g. `?stringr` for help on the `stringr` package, or `?library` for the loading function.
- Popular packages also have cheat sheets revealing their complexity:

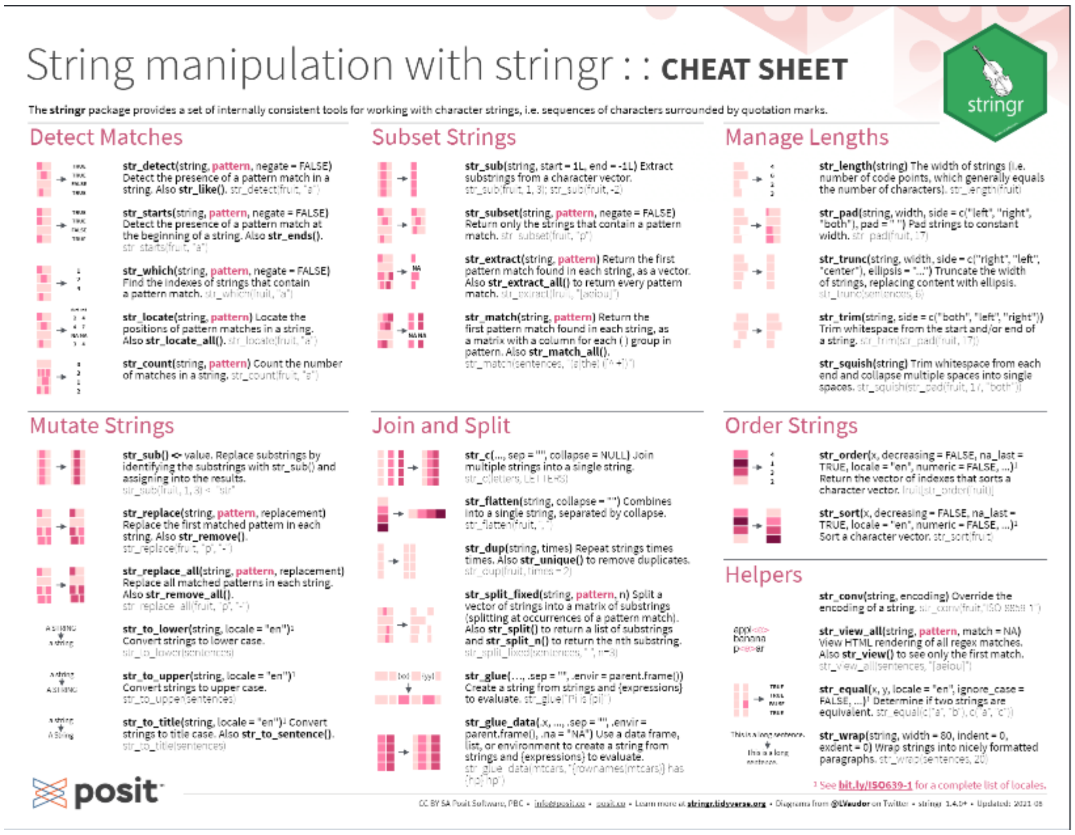
Quick taste of test mining

This is a quick example to demonstrate text mining using the `qdap` package that we just installed.

1. Load `qdap` and show all loaded packages.

   ```
   library(qdap)
   search()
   ```

   ```
    [1] ".GlobalEnv"              "package:tm"
    [3] "package:NLP"             "package:stringr"
   ```



String manipulation with stringr : : **CHEAT SHEET**

The **stringr** package provides a set of internally consistent tools for working with character strings, i.e. sequences of characters surrounded by quotation marks.

Detect Matches

str_detect(string, pattern, negate = FALSE) Detect the presence of a pattern match in a string. Also **str_like()**. str_detect(fruit, "a")

str_starts(string, pattern, negate = FALSE) Detect the presence of a pattern match at the beginning of a string. Also **str_ends()**. str_starts(fruit, "a")

str_which(string, pattern, negate = FALSE) Find the indexes of strings that contain a pattern match. str_which(fruit, "a")

str_locate(string, pattern) Locate the positions of pattern matches in a string. Also **str_locate_all()**. str_locate(fruit, "a")

str_count(string, pattern) Count the number of matches in a string. str_count(fruit, "a")

Subset Strings

str_sub(string, start = 1L, end = -1L) Extract substrings from a character vector. str_sub(fruit, 1, 3); str_sub(fruit, -2)

str_subset(string, pattern, negate = FALSE) Return only the strings that contain a pattern match. str_subset(fruit, "p")

str_extract(string, pattern) Return the first pattern match found in each string, as a vector. Also **str_extract_all()** to return every pattern match. str_extract(fruit, "[aeiou]")

str_match(string, pattern) Return the first pattern match found in each string, as a matrix with a column for each ( ) group in pattern. Also **str_match_all()**. str_match(sentences, "(a|the) ([^ +])")

Manage Lengths

str_length(string) The width of strings (i.e. number of code points, which generally equals the number of characters). str_length(fruit)

str_pad(string, width, side = c("left", "right", "both"), pad = " ") Pad strings to constant width. str_pad(fruit, 17)

str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...") Truncate the width of strings, replacing content with ellipsis. str_trunc(sentences, 6)

str_trim(string, side = c("both", "left", "right")) Trim whitespace from the start and/or end of a string. str_trim(str_pad(fruit, 17))

str_squish(string) Trim whitespace from each end and collapse multiple spaces into single spaces. str_squish(str_pad(fruit, 17, "both"))

Mutate Strings

str_sub() <- value. Replace substrings by identifying the substrings with str_sub() and assigning into the results. str_sub(fruit, 1, 3) <- "str"

str_replace(string, pattern, replacement) Replace the first matched pattern in each string. Also **str_remove()**. str_replace(fruit, "p", "-")

str_replace_all(string, pattern, replacement) Replace all matched patterns in each string. Also **str_remove_all()**. str_replace_all(fruit, "p", "-")

str_to_lower(string, locale = "en")[1] Convert strings to lower case. str_to_lower(sentences)

str_to_upper(string, locale = "en")[1] Convert strings to upper case. str_to_upper(sentences)

str_to_title(string, locale = "en")[1] Convert strings to title case. Also **str_to_sentence()**. str_to_title(sentences)

Join and Split

str_c(..., sep = "", collapse = NULL) Join multiple strings into a single string. str_c(letters, LETTERS)

str_flatten(string, collapse = "") Combines into a single string, separated by collapse. str_flatten(fruit, ", ")

str_dup(string, times) Repeat strings times times. Also **str_unique()** to remove duplicates. str_dup(fruit, times = 2)

str_split_fixed(string, pattern, n) Split a vector of strings into a matrix of substrings (splitting at occurrences of a pattern match). Also **str_split()** to return a list of substrings and **str_split_n()** to return the nth substring. str_split_fixed(sentences, " ", n=3)

str_glue(..., .sep = "", .envir = parent.frame()) Create a string from strings and {expressions} to evaluate. str_glue("Pi is {pi}")

str_glue_data(.x, ..., .sep = "", .envir = parent.frame(), .na = "NA") Use a data frame, list, or environment to create a string from strings and {expressions} to evaluate. str_glue_data(mtcars, "{rownames(mtcars)} has {hp} hp")

Order Strings

str_order(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)[1] Return the vector of indexes that sorts a character vector. fruit[str_order(fruit)]

str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)[1] Sort a character vector. str_sort(fruit)

Helpers

str_conv(string, encoding) Override the encoding of a string. str_conv(fruit,"ISO-8859-1")

str_view_all(string, pattern, match = NA) View HTML rendering of all regex matches. Also **str_view()** to see only the first match. str_view_all(sentences, "[aeiou]")

str_equal(x, y, locale = "en", ignore_case = FALSE, ...)[1] Determine if two strings are equivalent. str_equal(c("a", "b"), c("a", "c"))

str_wrap(string, width = 80, indent = 0, exdent = 0) Wrap strings into nicely formatted paragraphs. str_wrap(sentences, 20)

[1] See **bit.ly/ISO639-1** for a complete list of locales.

CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **stringr.tidyverse.org** • Diagrams from **@LVaudor** on Twitter • stringr 1.4.0+ • Updated: 2021-08

Figure 3: stringr cheat sheet at github.com/rstudio/cheatsheets

```
 [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"
```

2. Store this text in a vector `text`: "DataCamp is the first online learning platform that focuses on building the best learning experience specifically for Data Science. We have offices in New York, London, and Belgium, and to date, we trained over 11 million (aspiring) data scientists in over 150 countries. These data science enthusiasts completed more than 667 million exercises. You can take free beginner courses, or subscribe for $25/month to get access to all premium courses."

   - In Emacs, mark the start of the text with `C-SPC` (CTRL + SPACEBAR)
   - Go down to the end of the text with `C-e` (CTRL + e)
   - Copy the text with `M-w` (ALT + w)
   - Paste the text wherever you want to with `C-y` (CTRL + y)

```
text <- "DataCamp is the first online learning platform that focuses on building t
text
```

```
[1] "DataCamp is the first online learning platform that focuses on building the b
```

3. Check the data type of `text`, and print its `length` and the number of its characters.

```
class(text)
length(text)
nchar(text)
```

```
[1] "character"
[1] 1
[1] 446
```

4. Find the 10 most frequent terms and store them in `term_count`:

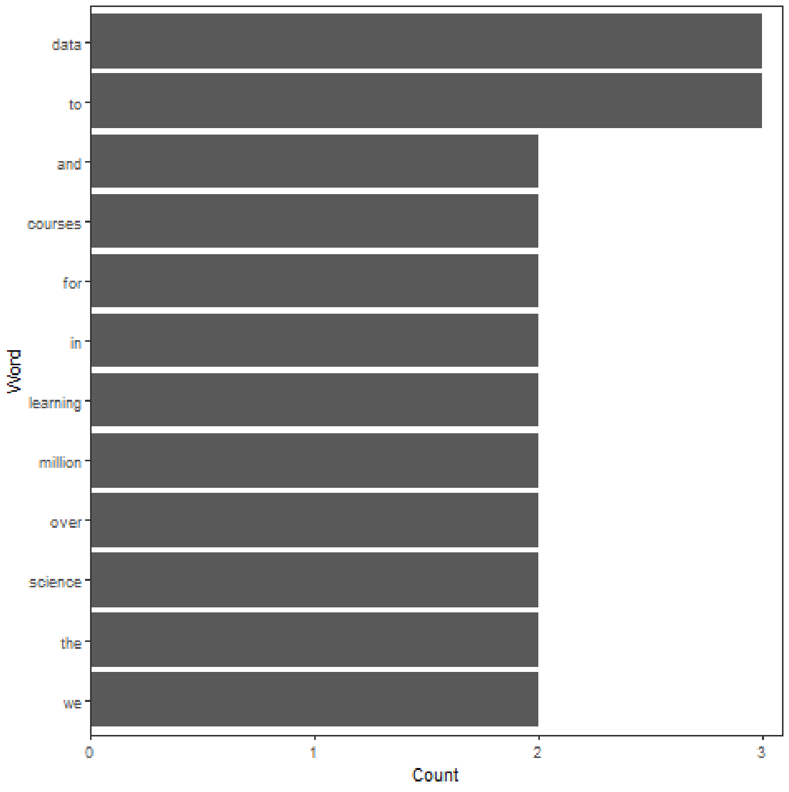
```
term_count <- freq_terms(text, 10)
term_count
```

```
   WORD     FREQ
1  data        3
2  to          3
3  and         2
4  courses     2
5  for         2
6  in          2
7  learning    2
8  million     2
9  over        2
10 science     2
11 the         2
12 we          2
```

5. If you compare with what we said above, you can see that this table is a transposed document matrix (TDM) with one feature (word frequency `FREQ`).

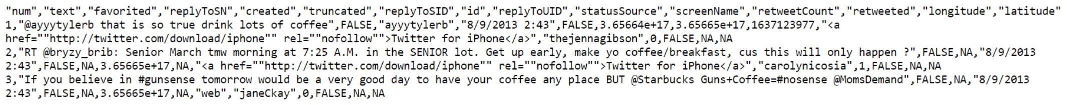
6. Plot the term count:

```
plot(term_count)
```



Getting the data

- Beginning with this section, you'll practice using your own literate programs: download the online CSV ("Comma Separated Values") dataset from this URL to your PC and name the file `coffee.csv`.



```
"num","text","favorited","replyToSN","created","truncated","replyToSID","id","replyToUID","statusSource","screenName","retweetCount","retweeted","longitude","latitude"
1,"@ayyytylerb that is so true drink lots of coffee",FALSE,"ayyytylerb","8/9/2013 2:43",FALSE,3.65664e+17,3.65665e+17,1637123977,"<a
href=""http://twitter.com/download/iphone"" rel=""nofollow"">Twitter for iPhone</a>","thejennagibson",0,FALSE,NA,NA
2,"RT @bryzy_brib: Senior March tmw morning at 7:25 A.M. in the SENIOR lot. Get up early, make yo coffee/breakfast, cus this will only happen ?",FALSE,NA,"8/9/2013
2:43",FALSE,NA,3.65665e+17,NA,"<a href=""http://twitter.com/download/iphone"" rel=""nofollow"">Twitter for iPhone</a>","carolynicosia",1,FALSE,NA,NA
3,"If you believe in #gunsense tomorrow would be a very good day to have your coffee any place BUT @Starbucks Guns+Coffee=#nosense @MomsDemand",FALSE,NA,"8/9/2013
2:43",FALSE,NA,3.65665e+17,NA,"web","janeCkay",0,FALSE,NA,NA
```

Figure 4: Top of the CSV file coffee.csvx

- We'll load the CSV data into R as a data frame - a table whose rows correspond to the individual tweets, and whose columns correspond to the features of each tweet.

```
coffee_tweets.df <- read.csv(file = "../data/coffee.csv")
```

- In the DataCamp lesson, you read that `read.csv` "treats `character` strings as factor `levels`" by default. This is **not true**. Check the `help` for `read.csv`
- Here, we assign the contents of the file as a data frame to the R object `coffee_tweets.df` using the assignment operator `<-`.
- The string `../data/coffee.csv` is a relative address to the file on your PC. Instead, you could also use the URL (in between the double apostrophs): the following command stores the file in another object `coffee_tweets.df.1` (the address comes from DataCamp):

  ```
  coffee_tweets.df.1 <- read.csv(file =
    "https://assets.datacamp.com/production/repositories/19/datasets/27a2a8587eff17a
  ```

Review: `factor` vectors

- When storing categorical variables like `male` or `female`, it is useful to treat them as a finite collection instead of as strings - e.g. to order them.
- Example: let's define a vector, `x` with 4 string values, "good", "good", "bad", "worst".

  ```
  x <- c(rep("good", times=2), "bad","worst")
  x
  ```

  ```
  [1] "good"  "good"  "bad"   "worst"
  ```

- Turn `x` into a nominal (non-ordered) `factor` named `xf`:

  ```
  xf <- factor(x)
  xf
  ```

  ```
  [1] good  good  bad   worst
  Levels: bad good worst
  ```

- The three levels of `xf` aren't semantically nominal - better to order them properly:

```
levels(xf) <- c("worst", "bad", "good")
xf
str(xf)
is.ordered(xf)
xfo <- factor(xf, ordered=TRUE)
xfo

[1] bad   bad   worst good
Levels: worst bad good
 Factor w/ 3 levels "worst","bad",..: 2 2 1 3
[1] FALSE
[1] bad   bad   worst good
Levels: worst < bad < good
```

NEXT First look at the data

- To look at the object, you could type its name - not wise because you don't know yet if it has 1 million lines. Instead, use `str` to only look at its structure (and size):

  ```
  str(coffee_tweets.df)

  '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 @bry
   $ 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=\"nofol
   $ 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 ...
  ```

- We see that `coffee.df` has 1000 rows and 15 columns or features

- You can also see that all `character` variables are intact - none have been turned into `factor` variables
- The features come in different data types: `integer`, `character`, `logical`, `numeric` (non-integer).
- Store the tweets (`text` column) in a vector `coffee_tweets` and view first 5 tweets - you can use the `$` operator for that:

```
coffee_tweets <- coffee_tweets.df$text
head(coffee_tweets, n=5)
```

```
[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
[4] "My cute coffee mug. http://t.co/2udvMU6XIG"
[5] "RT @slaredo21: I wish we had Starbucks here... Cause coffee dates in the morn
```

- The vector has a simpler structure:

```
str(coffee_tweets)
```

```
chr [1:1000] "@ayyytylerb that is so true drink lots of coffee" ...
```

TM Glossary

TERM	MEANING
Meta data	Tags stored alongside text data
OpenNLP	Apache natural language processing library
Bag of Words	No syntax just words read out as DTMs/TDMs
DTM	Document term matrix (corpus vs. words)
TDM	Transposed document matrix (words vs. corpus)
Syntactic parsing	Analyse language to discover grammar
Semantic parsing	Analyse language to discover meaning
R console	Shell application for interactive R use
`install.packages`	Install R package
`library`	Load package
`help`, `?`	Get help on package or function
`search()`	Show all loaded packages
dataframe	Table of records (rows) and features (columns)
`read.csv`	R function to read CSV data to dataframe
`str`	R function to display structure of an R object
`<-`, `->`	R assignment operator
`$`	Operator to extract vectors from dataframe
`head`	Display first few lines of an R object
`stringi`	String manipulation package
`stringr`	"Tidyverse" wrapper for `stringi`
`qdap`	Analysis package for qualitative (text) data