**Data Science: Wrangling**

**Key points**

* Many datasets are stored in spreadsheets. A spreadsheet is essentially a file version of a data frame with rows and columns.
* Spreadsheets have rows separated by returns and columns separated by a delimiter. The most common delimiters are comma, semicolon, white space and tab.
* Many spreadsheets are raw text files and can be read with any basic text editor. However, some formats are proprietary and cannot be read with a text editor, such as Microsoft Excel files (.xls).
* Most import functions assume that the first row of a spreadsheet file is a header with column names. To know if the file has a header, it helps to look at the file with a text editor before trying to import it.

**Key points**

* The working directory is where R looks for files and saves files by default.
* See your working directory with getwd(). Change your working directory with setwd().
* We suggest you create a directory for each project and keep your raw data inside that directory.
* Use the file.path() function to generate a full path from a relative path and a file name. Use file.path() instead of paste() because file.path() is aware of your operating system and will use the correct slashes to navigate your machine.
* The file.copy() function copies a file to a new path.

**Code**

# see working directory

getwd()

# change your working directory

setwd()

# set path to the location for raw data files in the dslabs package and list files

path <- system.file("extdata", package="dslabs")

list.files(path)

# generate a full path to a file

filename <- "murders.csv"

fullpath <- file.path(path, filename)

fullpath

# copy file from dslabs package to your working directory

file.copy(fullpath, getwd())

# check if the file exists

file.exists(filename)

### Key points

* **readr**is the **tidyverse** library that includes functions for reading data stored in text file spreadsheets into R. Functions in the package include read\_csv(), read\_tsv(), read\_delim() and more. These differ by the delimiter they use to split columns.
* The **readxl** package provides functions to read Microsoft Excel formatted files.
* The excel\_sheets() function gives the names of the sheets in the Excel file. These names are passed to the sheet argument for the **readxl** functions read\_excel(), read\_xls() and read\_xlsx().
* The read\_lines() function shows the first few lines of a file in R.

### Code

library(dslabs)

library(tidyverse) # includes readr

library(readxl)

# inspect the first 3 lines

read\_lines("murders.csv", n\_max = 3)

# read file in CSV format

dat <- read\_csv(filename)

#read using full path

dat <- read\_csv(fullpath)

head(dat)

#Ex：

path <- system.file("extdata", package = "dslabs")

files <- list.files(path)

files

filename <- "murders.csv"

filename1 <- "life-expectancy-and-fertility-two-countries-example.csv"

filename2 <- "fertility-two-countries-example.csv"

dat=read.csv(file.path(path, filename))

dat1=read.csv(file.path(path, filename1))

dat2=read.csv(file.path(path, filename2))

### Key points

* R-base import functions (read.csv(), read.table(), read.delim()) generate data frames rather than tibbles.
* Note that as of R 4.0, it is no longer necessary to use the argument stringsAsFactors=FALSE to prevent characters from being converted into factors.

### Code

# filename is defined in the previous video

# read.csv to import the data

dat2 <- read.csv(filename)

class(dat2$abb)

class(dat2$region)

print.dat.frame(dat2)

### Key points

* The read\_csv() function and other import functions can read a URL directly.
* If you want to have a local copy of the file, you can use download.file().
* tempdir() creates a directory with a name that is very unlikely not to be unique.
* tempfile() creates a character string that is likely to be a unique filename.

### Code

url <- "https://raw.githubusercontent.com/rafalab/dslabs/master/inst/extdata/murders.csv"

dat <- read\_csv(url)

download.file(url, "murders.csv")

tempfile()

tmp\_filename <- tempfile()

download.file(url, tmp\_filename)

dat <- read\_csv(tmp\_filename)

file.remove(tmp\_filename)

Leer desde Excel

times\_2016 <- read\_excel("times.xlsx", sheet = "2016")

### Key points

* In tidy data, each row represents an observation and each column represents a different variable.
* In wide data, each row includes several observations and one of the variables is stored in the header.

### Code

library(tidyverse)

library(dslabs)

data(gapminder)

# create and inspect a tidy data frame

tidy\_data <- gapminder %>%

filter(country %in% c("South Korea", "Germany")) %>%

select(country, year, fertility)

head(tidy\_data)

# plotting tidy data is simple

tidy\_data %>%

ggplot(aes(year, fertility, color = country)) +

geom\_point()

# import and inspect example of original Gapminder data in wide format

path <- system.file("extdata", package="dslabs")  
filename <- file.path(path, "fertility-two-countries-example.csv")  
wide\_data <- read\_csv(filename)  
select(wide\_data, country, `1960`:`1967`)

### Key points

* After importing data, a common next step is to reshape the data into a form useful for the rest of the analysis by tidying it. The **tidyr** package includes several useful functions for tidying data.
* The pivot\_longer() function converts wide data into tidy data.
* The first argument of pivot\_longer() is the data frame to be reshaped. The second argument specifies the columns containing the values to be moved into a single column.
* The new column of values is called value by default and the column containing the original names of those columns is called name by default.
* The values\_to and names\_to arguments can be used to change the default names of these columns.

### Code

# example dataset: fertility data in wide format (from previous video)

library(tidyverse)

library(dslabs)

path <- system.file("extdata", package="dslabs")

filename <- file.path(path, "fertility-two-countries-example.csv")

wide\_data <- read\_csv(filename)

# snippet of wide data

wide\_data %>% select(country, '1960':'1965')

# move the values in the columns 1960 through 2015 into a single column

wide\_data %>% pivot\_longer(`1960`:`2015`)

# another way to do this - only country isn't being pivoted

wide\_data %>% pivot\_longer(-country)

# change the default column names

new\_tidy\_data <- wide\_data %>%

pivot\_longer(-country, names\_to = "year", values\_to = "fertility")

head(new\_tidy\_data)

# compare the class from our original tidy data (year is an integer) and in the new version (year is a character)

class(tidy\_data$year)

class(new\_tidy\_data$year)

# use the names\_transform argument to change the class of the year values to numeric

new\_tidy\_data <- wide\_data %>%

pivot\_longer(-country, names\_to = "year", values\_to = "fertility",

names\_transform = list(year=as.numeric))

# plot the data as before

new\_tidy\_data %>% ggplot(aes(year, fertility, color = country)) +

geom\_point()

### Key points

* The pivot\_wider() function converts tidy data into wide data, which can be a useful intermediate step in data tidying.
* The data frame to be reshaped is the first argument in pivot\_wider().
* The argument names\_from tells pivot\_wider() which variable will be used for the column names and the argument values\_from tells pivot\_wider() which variable to use to fill in the values.
* The [tidyr cheat sheet External link](https://github.com/rstudio/cheatsheets/blob/main/tidyr.pdf) is a useful reference for these and other functions.

### Code

# still working with the same data as in the previous video

# convert the tidy data to wide data

new\_wide\_data <- new\_tidy\_data %>%

pivot\_wider(names\_from = year, values\_from = fertility)

select(new\_wide\_data, country, `1960`:`1967`)

### Key points

* The separate() function splits one column into two or more columns at a specified character that separates the variables.
* The separate() function takes three arguments (apart from the data): the name of the column to be separated, the names to be used for the new columns, and the character that separates the variables.
* When there is an extra separation, you can use extra = "merge" to merge the last two variables.

### Code

# import data

path <- system.file("extdata", package = "dslabs")

fname <- "life-expectancy-and-fertility-two-countries-example.csv"

filename <- file.path(path, fname)

raw\_dat <- read\_csv(filename)

select(raw\_dat, 1:4)

# pivot all columns except country

dat <- raw\_dat %>% pivot\_longer(-country)

head(dat)

dat$name[1:5]

# separate on underscores

dat %>% separate(name, c("year", "name"), sep = "\_")

# separate on underscores (the default), convert years to numeric

dat %>% separate(name, c("year", "name"), convert = TRUE)

# split on all underscores, pad empty cells with NA

dat %>% separate(name, c("year", "name\_1", "name\_2"),

fill = "right", convert = TRUE)

# split on first underscore but keep life\_expectancy merged

dat %>% separate(name, c("year", "name"), sep = "\_",

extra = "merge", convert = TRUE)

# separate then create a new column for each variable using pivot\_wider

dat %>% separate(name, c("year", "name"), sep = "\_",

extra = "merge", convert = TRUE) %>%

pivot\_wider()

### Key points

* The unite() function joins two columns into one.

### Code

# using the data from the previous video

# if we had used this non-optimal approach to separate

dat %>%

separate(name, c("year", "name\_1", "name\_2"),

fill = "right", convert = TRUE)

# we could unite the second and third columns using unite()

dat %>%

separate(name, c("year", "name\_1", "name\_2"),

fill = "right", convert = TRUE) %>%

unite(variable\_name, name\_1, name\_2, sep="\_")

# spread the columns

dat %>%

separate(name, c("year", "name\_1", "name\_2"),

fill = "right", convert = TRUE) %>%

unite(name, name\_1, name\_2, sep="\_") %>%

spread(name, value) %>%

rename(fertlity = fertility\_NA)

### Key points

* The join functions in the **dplyr** package combine two tables such that matching rows are together.
* left\_join() only keeps rows that have information in the first table.
* right\_join() only keeps rows that have information in the second table.
* inner\_join() only keeps rows that have information in both tables.
* full\_join() keeps all rows from both tables.
* semi\_join() keeps the part of first table for which we have information in the second.
* anti\_join() keeps the elements of the first table for which there is no information in the second.

### Code

# import US murders data

library(tidyverse)  
library(ggrepel)  
library(dslabs)  
ds\_theme\_set()  
data(murders)  
head(murders)

# import US election results data

data(polls\_us\_election\_2016)  
head(results\_us\_election\_2016)  
identical(results\_us\_election\_2016$state, murders$state)

# join the murders table and US election results table

tab <- left\_join(murders, results\_us\_election\_2016, by = "state")  
head(tab)

# plot electoral votes versus population

tab %>% ggplot(aes(population/10^6, electoral\_votes, label = abb)) +  
 geom\_point() +  
 geom\_text\_repel() +   
 scale\_x\_continuous(trans = "log2") +  
 scale\_y\_continuous(trans = "log2") +  
 geom\_smooth(method = "lm", se = FALSE)

# make two smaller tables to demonstrate joins

tab1 <- slice(murders, 1:6) %>% select(state, population)  
tab1  
tab2 <- slice(results\_us\_election\_2016, c(1:3, 5, 7:8)) %>% select(state, electoral\_votes)  
tab2

# experiment with different joins

left\_join(tab1, tab2)  
tab1 %>% left\_join(tab2)  
tab1 %>% right\_join(tab2)  
inner\_join(tab1, tab2)  
semi\_join(tab1, tab2)  
anti\_join(tab1, tab2)

### Key points

* Unlike the join functions, the binding functions do not try to match by a variable, but rather just combine datasets.
* bind\_cols() binds two objects by making them columns in a tibble. The R-base function cbind() binds columns but makes a data frame or matrix instead.
* The bind\_rows() function is similar but binds rows instead of columns. The R-base function rbind() binds rows but makes a data frame or matrix instead.

### Code

bind\_cols(a = 1:3, b = 4:6)

tab1 <- tab[, 1:3]  
tab2 <- tab[, 4:6]  
tab3 <- tab[, 7:9]  
new\_tab <- bind\_cols(tab1, tab2, tab3)  
head(new\_tab)

tab1 <- tab[1:2,]  
tab2 <- tab[3:4,]  
bind\_rows(tab1, tab2)

### Key points

* By default, the set operators in R-base work on vectors. If **tidyverse/dplyr** are loaded, they also work on data frames.
* You can take intersections of vectors using intersect(). This returns the elements common to both sets.
* You can take the union of vectors using union(). This returns the elements that are in either set.
* The set difference between a first and second argument can be obtained with setdiff(). Note that this function is not symmetric.
* The function set\_equal() tells us if two sets are the same, regardless of the order of elements.

### Code

# intersect vectors or data frames

intersect(1:10, 6:15)  
intersect(c("a","b","c"), c("b","c","d"))  
tab1 <- tab[1:5,]  
tab2 <- tab[3:7,]  
intersect(tab1, tab2)

# perform a union of vectors or data frames

union(1:10, 6:15)  
union(c("a","b","c"), c("b","c","d"))  
tab1 <- tab[1:5,]  
tab2 <- tab[3:7,]  
union(tab1, tab2)

# set difference of vectors or data frames

setdiff(1:10, 6:15)  
setdiff(6:15, 1:10)  
tab1 <- tab[1:5,]  
tab2 <- tab[3:7,]  
setdiff(tab1, tab2)

# setequal determines whether sets have the same elements, regardless of order

setequal(1:5, 1:6)  
setequal(1:5, 5:1)  
setequal(tab1, tab2)

### Key points

* Web scraping is extracting data from a website.
* The **rvest** web harvesting package includes functions to extract nodes of an HTML document: html\_nodes() extracts all nodes of different types, and html\_node() extracts the first node.
* html\_table() converts an HTML table to a data frame.

### Code

# import a webpage into R

library(rvest)  
url <- "https://en.wikipedia.org/wiki/Murder\_in\_the\_United\_States\_by\_state"  
h <- read\_html(url)  
class(h)  
h

tab <- h %>% html\_nodes("table")  
tab <- tab[[2]]

tab <- tab %>% html\_table  
class(tab)

tab <- tab %>% setNames(c("state", "population", "total", "murders", "gun\_murders", "gun\_ownership", "total\_rate", "murder\_rate", "gun\_murder\_rate"))  
head(tab)

# CSS Selectors

The default look of webpages made with the most basic HTML is quite unattractive. The aesthetically pleasing pages we see today are made using CSS. CSS is used to add style to webpages. The fact that all pages for a company have the same style is usually a result that they all use the same CSS file. The general way these CSS files work is by defining how each of the elements of a webpage will look. The title, headings, itemized lists, tables, and links, for example, each receive their own style including font, color, size, and distance from the margin, among others.

To do this CSS leverages patterns used to define these elements, referred to as selectors. An example of pattern we used in a previous video is table but there are many many more. If we want to grab data from a webpage and we happen to know a selector that is unique to the part of the page, we can use the html\_nodes() function.

However, knowing which selector to use can be quite complicated. To demonstrate this we will try to extract the recipe name, total preparation time, and list of ingredients from [this guacamole recipe External link](http://www.foodnetwork.com/recipes/alton-brown/guacamole-recipe-1940609). Looking at the code for this page, it seems that the task is impossibly complex. However, selector gadgets actually make this possible. [SelectorGadget External link](http://selectorgadget.com/) is piece of software that allows you to interactively determine what CSS selector you need to extract specific components from the webpage. If you plan on scraping data other than tables, we highly recommend you install it. A Chrome extension is available which permits you to turn on the gadget highlighting parts of the page as you click through, showing the necessary selector to extract those segments.

For the guacamole recipe page, we already have done this and determined that we need the following selectors:

h <- read\_html("http://www.foodnetwork.com/recipes/alton-brown/guacamole-recipe-1940609")  
recipe <- h %>% html\_node(".o-AssetTitle\_\_a-HeadlineText") %>% html\_text()

prep\_time <- h %>% html\_node(".m-RecipeInfo\_\_a-Description--Total") %>% html\_text()

ingredients <- h %>% html\_nodes(".o-Ingredients\_\_a-Ingredient") %>% html\_text()

You can see how complex the selectors are. In any case we are now ready to extract what we want and create a list:

guacamole <- list(recipe, prep\_time, ingredients)  
guacamole

Since recipe pages from this website follow this general layout, we can use this code to create a function that extracts this information:

get\_recipe <- function(url){

h <- read\_html(url)

recipe <- h %>% html\_node(".o-AssetTitle\_\_a-HeadlineText") %>% html\_text()

prep\_time <- h %>% html\_node(".m-RecipeInfo\_\_a-Description--Total") %>% html\_text()

ingredients <- h %>% html\_nodes(".o-Ingredients\_\_a-Ingredient") %>% html\_text()

return(list(recipe = recipe, prep\_time = prep\_time, ingredients = ingredients))

}

and then use it on any of their webpages:

get\_recipe("http://www.foodnetwork.com/recipes/food-network-kitchen/pancakes-recipe-1913844")

There are several other powerful tools provided by **rvest**. For example, the functions html\_form(), set\_values(), and submit\_form() permit you to query a webpage from R. This is a more advanced topic not covered here.

### Key points

* The most common tasks in string processing include:
  + - extracting numbers from strings
    - removing unwanted characters from text
    - finding and replacing characters
    - extracting specific parts of strings
    - converting free form text to more uniform formats
    - splitting strings into multiple values
* The **stringr** package in the **tidyverse** contains string processing functions that follow a similar naming format (str\_functionname) and are compatible with the pipe.

### Code

# read in raw murders data from Wikipedia

url <- "https://en.wikipedia.org/w/index.php?title=Gun\_violence\_in\_the\_United\_States\_by\_state&direction=prev&oldid=810166167"  
murders\_raw <- read\_html(url) %>%   
 html\_nodes("table") %>%   
 html\_table() %>%  
 .[[1]] %>%  
 setNames(c("state", "population", "total", "murder\_rate"))

# inspect data and column classes

head(murders\_raw)  
class(murders\_raw$population)  
class(murders\_raw$total)

### Key points

* Define a string by surrounding text with either single quotes or double quotes.
* To include a single quote inside a string, use double quotes on the outside. To include a double quote inside a string, use single quotes on the outside.
* The cat() function displays a string as it is represented inside R.
* To include a double quote inside of a string surrounded by double quotes, use the backslash (\) to escape the double quote. Escape a single quote to include it inside of a string defined by single quotes.
* We will see additional uses of the escape later.

### Code

s <- "Hello!" # double quotes define a string

s <- 'Hello!' # single quotes define a string

s <- `Hello` # backquotes do not

s <- "10"" # error - unclosed quotes

s <- '10"' # correct

# cat shows what the string actually looks like inside R

cat(s)

s <- "5'"  
cat(s)

# to include both single and double quotes in string, escape with \

s <- '5'10"' # error

s <- "5'10"" # error

s <- '5\'10"' # correct

cat(s)

s <- "5'10\"" # correct

cat(s)

### Key points

* The main types of string processing tasks are detecting, locating, extracting and replacing elements of strings.
* The **stringr** package from the **tidyverse** includes a variety of string processing functions that begin with str\_ and take the string as the first argument, which makes them compatible with the pipe.

### Code

# murders\_raw defined in web scraping video

# direct conversion to numeric fails because of commas

murders\_raw$population[1:3]  
as.numeric(murders\_raw$population[1:3])

library(tidyverse) # includes stringr

### Key points

* Use the str\_detect() function to determine whether a string contains a certain pattern.
* Use the str\_replace\_all() function to replace all instances of one pattern with another pattern. To remove a pattern, replace with the empty string ("").
* The parse\_number() function removes punctuation from strings and converts them to numeric.
* mutate\_at() performs the same transformation on the specified column numbers.

### Code

# murders\_raw was defined in the web scraping section

# detect whether there are commas

commas <- function(x) any(str\_detect(x, ","))  
murders\_raw %>% summarize\_all(funs(commas))

# replace commas with the empty string and convert to numeric

test\_1 <- str\_replace\_all(murders\_raw$population, ",", "")  
test\_1 <- as.numeric(test\_1)

# parse\_number also removes commas and converts to numeric

test\_2 <- parse\_number(murders\_raw$population)  
identical(test\_1, test\_2)

murders\_new <- murders\_raw %>% mutate\_at(2:3, parse\_number)  
murders\_new %>% head

### Key points

* In the raw heights data, many students did not report their height as the number of inches as requested. There are many entries with real height information but in the wrong format, which we can extract with string processing.
* When there are both text and numeric entries in a column, the column will be a character vector. Converting this column to numeric will result in NAs for some entries.
* To correct problematic entries, look for patterns that are shared across large numbers of entries, then define rules that identify those patterns and use these rules to write string processing tasks.
* Use suppressWarnings() to hide warning messages for a function.

### Code

# load raw heights data and inspect

library(dslabs)  
data(reported\_heights)  
class(reported\_heights$height)

# convert to numeric, inspect, count NAs

x <- as.numeric(reported\_heights$height)  
head(x)  
sum(is.na(x))

# keep only entries that result in NAs

reported\_heights %>% mutate(new\_height = as.numeric(height)) %>%  
 filter(is.na(new\_height)) %>%   
 head(n=10)

# calculate cutoffs that cover 99.999% of human population

alpha <- 1/10^6  
qnorm(1-alpha/2, 69.1, 2.9)  
qnorm(alpha/2, 63.7, 2.7)

# keep only entries that either result in NAs or are outside the plausible range of heights

not\_inches <- function(x, smallest = 50, tallest = 84){  
 inches <- suppressWarnings(as.numeric(x))  
 ind <- is.na(inches) | inches < smallest | inches > tallest  
 ind  
}

# number of problematic entries

problems <- reported\_heights %>%   
 filter(not\_inches(height)) %>%  
 .$height  
length(problems)

# 10 examples of x'y or x'y" or x'y\"

pattern <- "^\\d\\s\*'\\s\*\\d{1,2}\\.\*\\d\*'\*\"\*$"  
str\_subset(problems, pattern) %>% head(n=10) %>% cat

# 10 examples of x.y or x,y

pattern <- "^[4-6]\\s\*[\\.|,]\\s\*([0-9]|10|11)$"  
str\_subset(problems, pattern) %>% head(n=10) %>% cat

# 10 examples of entries in cm rather than inches

ind <- which(between(suppressWarnings(as.numeric(problems))/2.54, 54, 81) )  
ind <- ind[!is.na(ind)]  
problems[ind] %>% head(n=10) %>% cat

### Key points

* A regular expression (regex) is a way to describe a specific pattern of characters of text. A set of rules has been designed to do this specifically and efficiently.
* **stringr** functions can take a regex as a pattern.
* str\_detect() indicates whether a pattern is present in a string.
* The main difference between a regex and a regular string is that a regex can include special characters.
* The | symbol inside a regex means "or".
* Use '\\d' to represent digits. The backlash is used to distinguish it from the character 'd'. In R, you must use two backslashes for digits in regular expressions; in some other languages, you will only use one backslash for regex special characters.
* str\_view() highlights the first occurrence of a pattern, and the str\_view\_all() function highlights all occurrences of the pattern.

### Code

# load stringr through tidyverse

library(tidyverse)

# detect whether a comma is present

pattern <- ","  
str\_detect(murders\_raw$total, pattern)

# show the subset of strings including "cm"

str\_subset(reported\_heights$height, "cm")

# use the "or" symbol inside a regex (|)

yes <- c("180 cm", "70 inches")  
no <- c("180", "70''")  
s <- c(yes, no)  
str\_detect(s, "cm") | str\_detect(s, "inches")  
str\_detect(s, "cm|inches")

# highlight the first occurrence of a pattern

str\_view(s, pattern)

# highlight all instances of a pattern

str\_view\_all(s, pattern)

### Key points

* Define strings to test your regular expressions, including some elements that match and some that do not. This allows you to check for the two types of errors: failing to match and matching incorrectly.
* Square brackets define character classes: groups of characters that count as matching the pattern. You can use ranges to define character classes, such as [0-9] for digits and [a-zA-Z] for all letters.
* Anchors define patterns that must start or end at specific places. ^ and $ represent the beginning and end of the string respectively.
* Curly braces are quantifiers that state how many times a certain character can be repeated in the pattern. \\d{1,2} matches exactly 1 or 2 consecutive digits.

### Code

# s was defined in the previous video

yes <- c("5", "6", "5'10", "5 feet", "4'11")

no <- c("", ".", "Five", "six")

s <- c(yes, no)

pattern <- "\\d"

# [56] means 5 or 6

str\_view(s, "[56]")

# [4-7] means 4, 5, 6 or 7

yes <- as.character(4:7)  
no <- as.character(1:3)  
s <- c(yes, no)  
str\_detect(s, "[4-7]")

# ^ means start of string, $ means end of string

pattern <- "^\\d$"  
yes <- c("1", "5", "9")  
no <- c("12", "123", " 1", "a4", "b")  
s <- c(yes, no)  
str\_view(s, pattern)

# curly braces define quantifiers: 1 or 2 digits

pattern <- "^\\d{1,2}$"  
yes <- c("1", "5", "9", "12")  
no <- c("123", "a4", "b")  
str\_view(c(yes, no), pattern)

# combining character class, anchors and quantifier

pattern <- "^[4-7]'\\d{1,2}\"$"  
yes <- c("5'7\"", "6'2\"", "5'12\"")  
no <- c("6,2\"", "6.2\"","I am 5'11\"", "3'2\"", "64")  
str\_detect(yes, pattern)  
str\_detect(no, pattern)

### Key points

* str\_replace() replaces the first instance of the detected pattern with a specified string.
* Spaces are characters and R does not ignore them. Spaces are specified by the special character \\s.
* Additional quantifiers include \*, + and ?. \* means 0 or more instances of the previous character. ? means 0 or 1 instances. + means 1 or more instances.
* Before removing characters from strings with functions like str\_replace() and str\_replace\_all(), consider whether that replacement would have unintended effects.

### Code

The problems object is defined in the [reported heights case study introduction video](https://courses.edx.org/courses/course-v1:HarvardX+PH125.6x+2T2019/courseware/c59e9550f970406e81b8a908ce42dcc0/58fca697d7e2436187ff7059588d94e8/1?activate_block_id=block-v1%3AHarvardX%2BPH125.6x%2B2T2019%2Btype%40vertical%2Bblock%402cc7ac16174f4736aac210300b8179ba).

# number of entries matching our desired pattern

pattern <- "^[4-7]'\\d{1,2}\"$"  
sum(str\_detect(problems, pattern))

# inspect examples of entries with problems

problems[c(2, 10, 11, 12, 15)] %>% str\_view(pattern)  
str\_subset(problems, "inches")  
str\_subset(problems, "''")

# replace or remove feet/inches words before matching

pattern <- "^[4-7]'\\d{1,2}$"  
problems %>%   
 str\_replace("feet|ft|foot", "'") %>% # replace feet, ft, foot with '

str\_replace("inches|in|''|\"", "") %>% # remove all inches symbols

str\_detect(pattern) %>%   
 sum()

# R does not ignore whitespace

identical("Hi", "Hi ")

# \\s represents whitespace

pattern\_2 <- "^[4-7]'\\s\\d{1,2}\"$"

str\_subset(problems, pattern\_2)

# \* means 0 or more instances of a character

yes <- c("AB", "A1B", "A11B", "A111B", "A1111B")

no <- c("A2B", "A21B")

str\_detect(yes, "A1\*B")  
str\_detect(no, "A1\*B")

# test how \*, ? and + differ

data.frame(string = c("AB", "A1B", "A11B", "A111B", "A1111B"),

none\_or\_more = str\_detect(yes, "A1\*B"),

nore\_or\_once = str\_detect(yes, "A1?B"),

once\_or\_more = str\_detect(yes, "A1+B"))

# update pattern by adding optional spaces before and after feet symbol

pattern <- "^[4-7]\\s\*'\\s\*\\d{1,2}$"

problems %>%

str\_replace("feet|ft|foot", "'") %>% # replace feet, ft, foot with '

str\_replace("inches|in|''|\"", "") %>% # remove all inches symbols

str\_detect(pattern) %>%

sum()

### Key Points

* Groups are defined using parentheses.
* Once we define groups, we can use the function str\_match() to extract the values these groups define. str\_extract() extracts only strings that match a pattern, not the values defined by groups.
* You can refer to the ith group with \\i. For example, refer to the value in the second group with \\2.

### Code

# define regex with and without groups

pattern\_without\_groups <- "^[4-7],\\d\*$"  
pattern\_with\_groups <- "^([4-7]),(\\d\*)$"

# create examples

yes <- c("5,9", "5,11", "6,", "6,1")  
no <- c("5'9", ",", "2,8", "6.1.1")  
s <- c(yes, no)

# demonstrate the effect of groups

str\_detect(s, pattern\_without\_groups)  
str\_detect(s, pattern\_with\_groups)

# demonstrate difference between str\_match and str\_extract

str\_match(s, pattern\_with\_groups)  
str\_extract(s, pattern\_with\_groups)

# improve the pattern to recognize more events

pattern\_with\_groups <- "^([4-7]),(\\d\*)$"  
yes <- c("5,9", "5,11", "6,", "6,1")  
no <- c("5'9", ",", "2,8", "6.1.1")  
s <- c(yes, no)  
str\_replace(s, pattern\_with\_groups, "\\1'\\2")

# final pattern

pattern\_with\_groups <-"^([4-7])\\s\*[,\\.\\s+]\\s\*(\\d\*)$"

# combine stringr commands with the pipe

str\_subset(problems, pattern\_with\_groups) %>% head  
str\_subset(problems, pattern\_with\_groups) %>%   
 str\_replace(pattern\_with\_groups, "\\1'\\2") %>% head

# define regex with and without groups

pattern\_without\_groups <- "^[4-7],\\d\*$"  
pattern\_with\_groups <- "^([4-7]),(\\d\*)$"

# create examples

yes <- c("5,9", "5,11", "6,", "6,1")  
no <- c("5'9", ",", "2,8", "6.1.1")  
s <- c(yes, no)

# demonstrate the effect of groups

str\_detect(s, pattern\_without\_groups)  
str\_detect(s, pattern\_with\_groups)

# demonstrate difference between str\_match and str\_extract

str\_match(s, pattern\_with\_groups)  
str\_extract(s, pattern\_with\_groups)

# improve the pattern to recognize more events

pattern\_with\_groups <- "^([4-7]),(\\d\*)$"  
yes <- c("5,9", "5,11", "6,", "6,1")  
no <- c("5'9", ",", "2,8", "6.1.1")  
s <- c(yes, no)  
str\_replace(s, pattern\_with\_groups, "\\1'\\2")

# final pattern

pattern\_with\_groups <-"^([4-7])\\s\*[,\\.\\s+]\\s\*(\\d\*)$"

# combine stringr commands with the pipe

str\_subset(problems, pattern\_with\_groups) %>% head  
str\_subset(problems, pattern\_with\_groups) %>%   
 str\_replace(pattern\_with\_groups, "\\1'\\2") %>% head

### Key Point

* The extract() function behaves similarly to the separate() function but allows extraction of groups from regular expressions.

### Code

# first example - normally formatted heights

s <- c("5'10", "6'1")  
tab <- data.frame(x = s)

# the separate and extract functions behave similarly

tab %>% separate(x, c("feet", "inches"), sep = "'")  
tab %>% extract(x, c("feet", "inches"), regex = "(\\d)'(\\d{1,2})")

# second example - some heights with unusual formats

s <- c("5'10", "6'1\"","5'8inches")  
tab <- data.frame(x = s)

# separate fails because it leaves in extra characters, but extract keeps only the digits because of regex groups

tab %>% separate(x, c("feet","inches"), sep = "'", fill = "right")  
tab %>% extract(x, c("feet", "inches"), regex = "(\\d)'(\\d{1,2})")

# Using Groups and Quantifiers

Four clear patterns of entries have arisen along with some other minor problems:

* 1. Many students measuring exactly 5 or 6 feet did not enter any inches. For example, **6'** - our pattern requires that inches be included.
  2. Some students measuring exactly 5 or 6 feet entered just that number.
  3. Some of the inches were entered with decimal points. For example **5'7.5''**. Our pattern only looks for two digits.
  4. Some entires have spaces at the end, for example **5 ' 9**.
  5. Some entries are in meters and some of these use European decimals: **1.6, 1,7**.
  6. Two students added **cm**.
  7. One student spelled out the numbers: **Five foot eight inches.**

It is not necessarily clear that it is worth writing code to handle all these cases since they might be rare enough. However, some give us an opportunity to learn some more regex techniques so we will build a fix.

### Case 1

For case 1, if we add a '0 to, for example, convert all 6 to 6'0, then our pattern will match. This can be done using groups using the following code:

yes <- c("5", "6", "5")

no <- c("5'", "5''", "5'4")

s <- c(yes, no)

str\_replace(s, "^([4-7])$", "\\1'0")

The pattern says it has to start (**^**), be followed with a digit between 4 and 7, and then end there (**$**). The parenthesis defines the group that we pass as **\\1** to the replace regex.

### Cases 2 and 4

We can adapt this code slightly to handle case 2 as well which covers the entry **5'**. Note that the **5'** is left untouched by the code above. This is because the extra **'** makes the pattern not match since we have to end with a 5 or 6. To handle case 2, we want to permit the 5 or 6 to be followed by no or one symbol for feet. So we can simply add **'{0,1}** after the **'** to do this. We can also use the none or once special character **?**. As we saw previously, this is different from **\*** which is none or more. We now see that this code also handles the fourth case as well:

str\_replace(s, "^([56])'?$", "\\1'0")

Note that here we only permit 5 and 6 but not 4 and 7. This is because heights of exactly 5 and exactly 6 feet tall are quite common, so we assume those that typed 5 or 6 really meant either 60 or 72 inches. However, heights of exactly 4 or exactly 7 feet tall are so rare that, although we accept 84 as a valid entry, we assume that a 7 was entered in error.

### Case 3

We can use quantifiers to deal with  case 3. These entries are not matched because the inches include decimals and our pattern does not permit this. We need allow the second group to include decimals and not just digits. This means we must permit zero or one period **.** followed by zero or more digits. So we will use both **?** and **\***. Also remember that for this particular case, the period needs to be escaped since it is a special character (it means any character except a line break).

So we can adapt our pattern, currently **^[4-7]\\s\*'\\s\*\\d{1,2}$**, to permit a decimal at the end:

pattern <- "^[4-7]\\s\*'\\s\*(\\d+\\.?\\d\*)$"

### Case 5

Case 5, meters using commas, we can approach similarly to how we converted the x.y to x'y. A difference is that we require that the first digit is 1 or 2:

yes <- c("1,7", "1, 8", "2, " )

no <- c("5,8", "5,3,2", "1.7")

s <- c(yes, no)

str\_replace(s, "^([12])\\s\*,\\s\*(\\d\*)$", "\\1\\.\\2")

We will later check if the entries are meters using their numeric values.

### Trimming

In general, spaces at the start or end of the string are uninformative. These can be particularly deceptive because sometimes they can be hard to see:

s <- "Hi "

cat(s)

identical(s, "Hi")

This is a general enough problem that there is a function dedicated to removing them: str\_trim.

str\_trim("5 ' 9 ")

### To upper and to lower case

One of the entries writes out numbers as words: **Five foot eight inches**. Although not efficient, we could add 12 extra **str\_replace** to convert **zero**to **0**, **one**to **1**, and so on. To avoid having to write two separate operations for **Zero**and **zero**, **One**and **one**, etc., we can use the str\_to\_lower() function to make all words lower case first:

s <- c("Five feet eight inches")

str\_to\_lower(s)

### Putting it into a function

We are now ready to define a procedure that handles converting all the problematic cases.

We can now put all this together into a function that takes a string vector and tries to convert as many strings as possible to a single format. Below is a function that puts together the previous code replacements:

convert\_format <- function(s){

s %>%

str\_replace("feet|foot|ft", "'") %>% #convert feet symbols to '

str\_replace\_all("inches|in|''|\"|cm|and", "") %>% #remove inches and other symbols

str\_replace("^([4-7])\\s\*[,\\.\\s+]\\s\*(\\d\*)$", "\\1'\\2") %>% #change x.y, x,y x y

str\_replace("^([56])'?$", "\\1'0") %>% #add 0 when to 5 or 6

str\_replace("^([12])\\s\*,\\s\*(\\d\*)$", "\\1\\.\\2") %>% #change european decimal

str\_trim() #remove extra space

}

We can also write a function that converts words to numbers:

words\_to\_numbers <- function(s){

str\_to\_lower(s) %>%

str\_replace\_all("zero", "0") %>%

str\_replace\_all("one", "1") %>%

str\_replace\_all("two", "2") %>%

str\_replace\_all("three", "3") %>%

str\_replace\_all("four", "4") %>%

str\_replace\_all("five", "5") %>%

str\_replace\_all("six", "6") %>%

str\_replace\_all("seven", "7") %>%

str\_replace\_all("eight", "8") %>%

str\_replace\_all("nine", "9") %>%

str\_replace\_all("ten", "10") %>%

str\_replace\_all("eleven", "11")

}

Now we can see which problematic entries remain:

converted <- problems %>% words\_to\_numbers %>% convert\_format

remaining\_problems <- converted[not\_inches\_or\_cm(converted)]

pattern <- "^[4-7]\\s\*'\\s\*\\d+\\.?\\d\*$"

index <- str\_detect(remaining\_problems, pattern)

remaining\_problems[!index]

### Key Points

* The function str\_split() splits a string into a character vector on a delimiter (such as a comma, space or underscore). By default, str\_split() generates a list with one element for each original string. Use the function argument simplify=TRUE to have str\_split() return a matrix instead.
* The map() function from the **purrr** package applies the same function to each element of a list. To extract the ith entry of each element x, use map(x, i).
* map() always returns a list. Use map\_chr() to return a character vector and map\_int() to return an integer.

### Code

# read raw murders data line by line

filename <- system.file("extdata/murders.csv", package = "dslabs")  
lines <- readLines(filename)  
lines %>% head()

# split at commas with str\_split function, remove row of column names

x <- str\_split(lines, ",")   
x %>% head()  
col\_names <- x[[1]]  
x <- x[-1]

# extract first element of each list entry

library(purrr)  
map(x, function(y) y[1]) %>% head()  
map(x, 1) %>% head()

# extract columns 1-5 as characters, then convert to proper format - NOTE: DIFFERENT FROM VIDEO

dat <- data.frame(parse\_guess(map\_chr(x, 1)),

parse\_guess(map\_chr(x, 2)),

parse\_guess(map\_chr(x, 3)),

parse\_guess(map\_chr(x, 4)),

parse\_guess(map\_chr(x, 5))) %>%

setNames(col\_names)

dat %>% head

# more efficient code for the same thing

dat <- x %>%  
 transpose() %>%  
 map( ~ parse\_guess(unlist(.))) %>%  
 setNames(col\_names) %>%   
 as.data.frame()

# the simplify argument makes str\_split return a matrix instead of a list

x <- str\_split(lines, ",", simplify = TRUE)   
col\_names <- x[1,]  
x <- x[-1,]  
x %>% as\_data\_frame() %>%  
 setNames(col\_names) %>%  
 mutate\_all(parse\_guess)

**Case Study: Extracting a Table from a PDF**

library(dslabs)

data("research\_funding\_rates")

research\_funding\_rates

The data come from a [paper External link](http://www.pnas.org/content/112/40/12349.abstract) published in the prestigious journal PNAS. However, the data are not provided in a spreadsheet; they are in a table in a PDF document. We could extract the numbers by hand, but this could lead to human error. Instead we can try to wrangle the data using R.

### Downloading the data

We start by downloading the PDF document then importing it into R using the following code:

library("pdftools")

temp\_file <- tempfile()

url <- "https://www.pnas.org/action/downloadSupplement?doi=10.1073%2Fpnas.1510159112&file=pnas.201510159SI.pdf"

download.file(url, temp\_file)

txt <- pdf\_text(temp\_file)

file.remove(temp\_file)

If we examine the object text we notice that it is a character vector with an entry for each page. So we keep the page we want using the following code:

raw\_data\_research\_funding\_rates <- txt[2]

The steps above can actually be skipped because we include the raw data in the dslabs package as well:

data("raw\_data\_research\_funding\_rates")

### Looking at the download

Examining this object,

raw\_data\_research\_funding\_rates %>% head

we see that it is a long string. Each line on the page, including the table rows, is separated by the symbol for newline: \n.

We can therefore can create a list with the lines of the text as elements:

tab <- str\_split(raw\_data\_research\_funding\_rates, "\n")

Because we start off with just one element in the string, we end up with a list with just one entry:

tab <- tab[[1]]

By examining this object,

tab %>% head

we see that the information for the column names is the third and fourth entires:

the\_names\_1 <- tab[3]

the\_names\_2 <- tab[4]

In the table, the column information is spread across two lines. We want to create one vector with one name for each column. We can do this using some of the functions we have just learned.

### Extracting the table data

Let's start with the first line:

the\_names\_1

We want to remove the leading space and everything following the comma. We can use regex for the latter. Then we can obtain the elements by splitting using the space. We want to split only when there are 2 or more spaces to avoid splitting success rate. So we use the regex \\s{2,} as follows:

the\_names\_1 <- the\_names\_1 %>%

str\_trim() %>%

str\_replace\_all(",\\s.", "") %>%

str\_split("\\s{2,}", simplify = TRUE)

the\_names\_1

Now let's look at the second line:

the\_names\_2

Here we want to trim the leading space and then split by space as we did for the first line:

the\_names\_2 <- the\_names\_2 %>%

str\_trim() %>%

str\_split("\\s+", simplify = TRUE)

the\_names\_2

Now we can join these to generate one name for each column:

tmp\_names <- str\_c(rep(the\_names\_1, each = 3), the\_names\_2[-1], sep = "\_")

the\_names <- c(the\_names\_2[1], tmp\_names) %>%

str\_to\_lower() %>%

str\_replace\_all("\\s", "\_")

the\_names

Now we are ready to get the actual data. By examining the tab object, we notice that the information is in lines 6 through 14. We can use str\_split() again to achieve our goal:

new\_research\_funding\_rates <- tab[6:14] %>%

str\_trim %>%

str\_split("\\s{2,}", simplify = TRUE) %>%

data.frame(stringsAsFactors = FALSE) %>%

setNames(the\_names) %>%

mutate\_at(-1, parse\_number)

new\_research\_funding\_rates %>% head()

We can see that the objects are identical:

identical(research\_funding\_rates, new\_research\_funding\_rates)

# Recoding

### Key points

* Change long factor names with the recode() function from the **tidyverse**.
* Other similar functions include recode\_factor() and fct\_recoder() in the **forcats** package in the **tidyverse**. The same result could be obtained using the case\_when() function, but recode() is more efficient to write.

### Code

# life expectancy time series for Caribbean countries

library(dslabs)  
data("gapminder")  
gapminder %>%   
 filter(region=="Caribbean") %>%  
 ggplot(aes(year, life\_expectancy, color = country)) +  
 geom\_line()

# display long country names

gapminder %>%   
 filter(region=="Caribbean") %>%  
 filter(str\_length(country) >= 12) %>%  
 distinct(country)

# recode long country names and remake plot

gapminder %>% filter(region=="Caribbean") %>%  
 mutate(country = recode(country,   
 'Antigua and Barbuda'="Barbuda",  
 'Dominican Republic' = "DR",  
 'St. Vincent and the Grenadines' = "St. Vincent",  
 'Trinidad and Tobago' = "Trinidad")) %>%  
 ggplot(aes(year, life\_expectancy, color = country)) +  
 geom\_line()

### Key points

* Dates are a separate data type in R.The **tidyverse** includes functionality for dealing with dates through the **lubridate** package.
* Extract the year, month and day from a date object with the year(), month() and day() functions.
* Parsers convert strings into dates with the standard YYYY-MM-DD format (ISO 8601 format). Use the parser with the name corresponding to the string format of year, month and day (ymd(), ydm(), myd(), mdy(), dmy(), dym()).
* Get the current time with the Sys.time() function. Use the now() function instead to specify a time zone.
* You can extract values from time objects with the hour(), minute() and second() functions.
* Parsers convert strings into times (for example, hms()). Parsers can also create combined date-time objects (for example, mdy\_hms()).

### Code

# inspect the startdate column of 2016 polls data, a Date type

library(tidyverse)  
library(dslabs)  
data("polls\_us\_election\_2016")  
polls\_us\_election\_2016$startdate %>% head  
class(polls\_us\_election\_2016$startdate)  
as.numeric(polls\_us\_election\_2016$startdate) %>% head

# ggplot is aware of dates

polls\_us\_election\_2016 %>% filter(pollster == "Ipsos" & state =="U.S.") %>%  
 ggplot(aes(startdate, rawpoll\_trump)) +  
 geom\_line()

# lubridate: the tidyverse date package

library(lubridate)

# select some random dates from polls

set.seed(2)  
dates <- sample(polls\_us\_election\_2016$startdate, 10) %>% sort  
dates

# extract month, day, year from date strings

data.frame(date = dates,   
 month = month(dates),  
 day = day(dates),  
 year = year(dates))

month(dates, label = TRUE) # extract month label

# ymd works on mixed date styles

x <- c(20090101, "2009-01-02", "2009 01 03", "2009-1-4",  
 "2009-1, 5", "Created on 2009 1 6", "200901 !!! 07")  
ymd(x)

# different parsers extract year, month and day in different orders

x <- "09/01/02"  
ymd(x)  
mdy(x)  
ydm(x)  
myd(x)  
dmy(x)  
dym(x)

now() # current time in your time zone

now("GMT") # current time in GMT

now() %>% hour() # current hour

now() %>% minute() # current minute

now() %>% second() # current second

# parse time

x <- c("12:34:56")  
hms(x)

#parse datetime

x <- "Nov/2/2012 12:34:56"  
mdy\_hms(x)

# Text Mining

### Key points

* The **tidytext** package helps us convert free form text into a tidy table.
* Use unnest\_tokens() to extract individual words and other meaningful chunks of text.
* Sentiment analysis assigns emotions or a positive/negative score to tokens. You can extract sentiments using get\_sentiments(). Common lexicons for sentiment analysis are "bing", "afinn", "nrc" and "loughran".

With the exception of labels used to represent categorical data, we have focused on numerical data, but in many applications data starts as text. Well known examples are spam filtering, cyber-crime prevention, counter-terrorism and sentiment analysis.

In all these examples, the raw data is composed of free form texts. Our task is to extract insights from these data. In this section, we learn how to generate useful numerical summaries from text data to which we can apply some of the powerful data visualization and analysis techniques we have learned.

### Case study: Trump Tweets

During the 2016 US presidential election, then-candidate Donald J. Trump used his Twitter account as a way to communicate with potential voters. On August 6, 2016 Todd Vaziri [tweeted External link](https://twitter.com/tvaziri/status/762005541388378112/photo/1) about Trump that "Every non-hyperbolic tweet is from iPhone (his staff). Every hyperbolic tweet is from Android (from him)." Data scientist David Robinson conducted an [analysis External link](http://varianceexplained.org/r/trump-tweets/) to determine if data supported this assertion. Here we go through David's analysis to learn some of the basics of text mining. To learn more about text mining in R we recommend [this book External link](https://www.tidytextmining.com/).

We will use the following libraries

library(tidyverse)

library(ggplot2)

library(lubridate)

library(tidyr)

library(scales)

set.seed(1)

In general, we can extract data directly from Twitter using the **rtweet** package. However, in this case, a group has already compiled data for us and made it available at [https://www.thetrumparchive.com/ External link](https://www.thetrumparchive.com/).

url <- 'https://drive.google.com/file/d/16wm-2NTKohhcA26w-kaWfhLIGwl\_oX95/view'

trump\_tweets <- map(2009:2017, ~sprintf(url, .x)) %>%

map\_df(jsonlite::fromJSON, simplifyDataFrame = TRUE) %>%

filter(!is\_retweet & !str\_detect(text, '^"')) %>%

mutate(created\_at = parse\_date\_time(created\_at, orders = "a b! d! H!:M!:S! z!\* Y!", tz="EST"))

For convenience we include the result of the code above in the **dslabs** package:

library(dslabs)

data("trump\_tweets")

This is data frame with information about the tweet:

head(trump\_tweets)

The variables that are included are:

names(trump\_tweets)

The help file ?trump\_tweets provides details on what each variable represents. The tweets are represented by the text variable:

trump\_tweets %>% select(text) %>% head

and the source variable tells us the device that was used to compose and upload each tweet:

trump\_tweets %>% count(source) %>% arrange(desc(n))

We can use extract to remove the Twitter for part of the source and filter out retweets.

trump\_tweets %>%

extract(source, "source", "Twitter for (.\*)") %>%

count(source)

We are interested in what happened during the campaign, so for the analysis here we will focus on what was tweeted between the day Trump announced his campaign and election day. So we define the following table:

campaign\_tweets <- trump\_tweets %>%

extract(source, "source", "Twitter for (.\*)") %>%

filter(source %in% c("Android", "iPhone") &

created\_at >= ymd("2015-06-17") &

created\_at < ymd("2016-11-08")) %>%

filter(!is\_retweet) %>%

arrange(created\_at)

We can now use data visualization to explore the possibility that two different groups were tweeting from these devices. For each tweet, we will extract the hour, in the east coast (EST), it was tweeted then compute the proportion of tweets tweeted at each hour for each device.

ds\_theme\_set()

campaign\_tweets %>%

mutate(hour = hour(with\_tz(created\_at, "EST"))) %>%

count(source, hour) %>%

group\_by(source) %>%

mutate(percent = n / sum(n)) %>%

ungroup %>%

ggplot(aes(hour, percent, color = source)) +

geom\_line() +

geom\_point() +

scale\_y\_continuous(labels = percent\_format()) +

labs(x = "Hour of day (EST)",

y = "% of tweets",

color = "")

We notice a big peak for the Android in early hours of the morning, between 6 and 8 AM. There seems to be a clear different in these patterns. We will therefore assume that two different entities are using these two devices. Now we will study how their tweets differ. To do this we introduce the **tidytext** package.

### Text as data

The **tidytext** package helps us convert free from text into a tidy table. Having the data in this format greatly facilitates data visualization and applying statistical techniques.

library(tidytext)

The main function needed to achieve this is unnest\_tokens(). A token refers to the units that we are considering to be a data point. The most common tokens will be words, but they can also be single characters, ngrams, sentences, lines or a pattern defined by a regex. The functions will take a vector of strings and extract the tokens so that each one gets a row in the new table. Here is a simple example:

example <- data\_frame(line = c(1, 2, 3, 4),

text = c("Roses are red,", "Violets are blue,", "Sugar is sweet,", "And so are you."))

example

example %>% unnest\_tokens(word, text)

Now let's look at a quick example with a tweet number 3008:

i <- 3008

campaign\_tweets$text[i]

campaign\_tweets[i,] %>%

unnest\_tokens(word, text) %>%

select(word)

Note that the function tries to convert tokens into words and strips characters important to twitter such as # and @. A token in twitter is not the same as in regular English. For this reason, instead of using the default token, words, we define a regex that captures twitter character. The pattern appears complex but all we are defining is a patter that starts with @, # or neither and is followed by any combination of letters or digits:

pattern <- "([^A-Za-z\\d#@']|'(?![A-Za-z\\d#@]))"

We can now use the unnest\_tokens() function with the regex option and appropriately extract the hashtags and mentions:

campaign\_tweets[i,] %>%

unnest\_tokens(word, text, token = "regex", pattern = pattern) %>%

select(word)

Another minor adjustment we want to make is remove the links to pictures:

campaign\_tweets[i,] %>%

mutate(text = str\_replace\_all(text, "https://t.co/[A-Za-z\\d]+|&amp;", "")) %>%

unnest\_tokens(word, text, token = "regex", pattern = pattern) %>%

select(word)

Now we are ready to extract the words for all our tweets.

tweet\_words <- campaign\_tweets %>%

mutate(text = str\_replace\_all(text, "https://t.co/[A-Za-z\\d]+|&amp;", "")) %>%

unnest\_tokens(word, text, token = "regex", pattern = pattern)

And we can now answer questions such as "what are the most commonly used words?"

tweet\_words %>%

count(word) %>%

arrange(desc(n))

It is not surprising that these are the top words. The top words are not informative. The tidytext package has database of these commonly used words, referred to as stop words, in text mining:

stop\_words

If we filter out rows representing stop words with filter(!word %in% stop\_words$word):

tweet\_words <- campaign\_tweets %>%

mutate(text = str\_replace\_all(text, "https://t.co/[A-Za-z\\d]+|&amp;", "")) %>%

unnest\_tokens(word, text, token = "regex", pattern = pattern) %>%

filter(!word %in% stop\_words$word )

We end up with a much more informative set of top 10 tweeted words:

tweet\_words %>%

count(word) %>%

top\_n(10, n) %>%

mutate(word = reorder(word, n)) %>%

arrange(desc(n))

Some exploration of the resulting words (not show here) reveals a couple of unwanted characteristics in our tokens. First, some of our tokens are just numbers (years for example). We want to remove these and we can find them using the regex ^\d+$. Second, some of our tokens come from a quote and they start with '. We want to remove the ' when it's at the start of a word, so we will use str\_replace(). We add these two lines to the code above to generate our final table:

tweet\_words <- campaign\_tweets %>%

mutate(text = str\_replace\_all(text, "https://t.co/[A-Za-z\\d]+|&amp;", "")) %>%

unnest\_tokens(word, text, token = "regex", pattern = pattern) %>%

filter(!word %in% stop\_words$word &

!str\_detect(word, "^\\d+$")) %>%

mutate(word = str\_replace(word, "^'", ""))

Now that we have all our words in a table, along with information about what device was used to compose the tweet they came from, we can start exploring which words are more common when comparing Android to iPhone.

For each word we want to know if it is more likely to come from an Android tweet or an iPhone tweet. We previously introduced the odds ratio, a summary statistic useful for quantifying these differences. For each device and a given word, let's call it y, we compute the odds or the ratio between the proportion of words that are y and not y and compute the ratio of those odds. Here we will have many proportions that are 0 so we use the 0.5 correction.

android\_iphone\_or <- tweet\_words %>%

count(word, source) %>%

spread(source, n, fill = 0) %>%

mutate(or = (Android + 0.5) / (sum(Android) - Android + 0.5) /

( (iPhone + 0.5) / (sum(iPhone) - iPhone + 0.5)))

android\_iphone\_or %>% arrange(desc(or))

android\_iphone\_or %>% arrange(or)

Given that several of these words are overall low frequency words we can impose a filter based on the total frequency like this:

android\_iphone\_or %>% filter(Android+iPhone > 100) %>%

arrange(desc(or))

android\_iphone\_or %>% filter(Android+iPhone > 100) %>%

arrange(or)

We already see somewhat of a pattern in the types of words that are being tweeted more in one device versus the other. However, we are not interested in specific words but rather in the tone. Vaziri's assertion is that the Android tweets are more hyperbolic. So how can we check this with data? Hyperbolic is a hard sentiment to extract from words as it relies on interpreting phrases. However, words can be associated to more basic sentiment such as as anger, fear, joy and surprise. In the next section we demonstrate basic sentiment analysis.

### Sentiment Analysis

In sentiment analysis we assign a word to one or more "sentiment". Although this approach will miss context dependent sentiments, such as sarcasm, when performed on large numbers of words, summaries can provide insights.

The first step in sentiment analysis is to assign a sentiment to each word. The tidytext package includes several maps or lexicons in the object sentiments:

sentiments

There are several lexicons in the tidytext package that give different sentiments. For example, the bing lexicon divides words into positive and negative. We can see this using the **tidytext** function get\_sentiments():

get\_sentiments("bing")

The AFINN lexicon assigns a score between -5 and 5, with -5 the most negative and 5 the most positive.

get\_sentiments("afinn")

The loughran and nrc lexicons provide several different sentiments:

get\_sentiments("loughran") %>% count(sentiment)

get\_sentiments("nrc") %>% count(sentiment)

To start learning about how these lexicons were developed, read this help file: ?sentiments.

For the analysis here we are interested in exploring the different sentiments of each tweet, so we will use the nrc lexicon:

nrc <- get\_sentiments("nrc") %>%

select(word, sentiment)

We can combine the words and sentiments using inner\_join(), which will only keep words associated with a sentiment. Here are 10 random words extracted from the tweets:

tweet\_words %>% inner\_join(nrc, by = "word") %>%

select(source, word, sentiment) %>% sample\_n(10)

Now we are ready to perform a quantitative analysis comparing Android and iPhone by comparing the sentiments of the tweets posted from each device. Here we could perform a tweet by tweet analysis, assigning a sentiment to each tweet. However, this somewhat complex since each tweet will have several sentiments attached to it, one for each word appearing in the lexicon. For illustrative purposes, we will perform a much simpler analysis: we will count and compare the frequencies of each sentiment appears for each device.

sentiment\_counts <- tweet\_words %>%

left\_join(nrc, by = "word") %>%

count(source, sentiment) %>%

spread(source, n) %>%

mutate(sentiment = replace\_na(sentiment, replace = "none"))

sentiment\_counts

Because more words were used on the Android than on the phone:

tweet\_words %>% group\_by(source) %>% summarize(n = n())

for each sentiment we can compute the odds of being in the device: proportion of words with sentiment versus proportion of words without and then compute the odds ratio comparing the two devices:

sentiment\_counts %>%

mutate(Android = Android / (sum(Android) - Android) ,

iPhone = iPhone / (sum(iPhone) - iPhone),

or = Android/iPhone) %>%

arrange(desc(or))

So we do see some difference and the order is interesting: the largest three sentiments are disgust, anger, and negative! But are they statistically significant? How does this compare if we are just assigning sentiments at random?

To answer that question we can compute, for each sentiment, an odds ratio and confidence interval. We will add the two values we need to form a two-by-two table and the odds ratio:

library(broom)

log\_or <- sentiment\_counts %>%

mutate( log\_or = log( (Android / (sum(Android) - Android)) / (iPhone / (sum(iPhone) - iPhone))),

se = sqrt( 1/Android + 1/(sum(Android) - Android) + 1/iPhone + 1/(sum(iPhone) - iPhone)),

conf.low = log\_or - qnorm(0.975)\*se,

conf.high = log\_or + qnorm(0.975)\*se) %>%

arrange(desc(log\_or))

log\_or

A graphical visualization shows some sentiments that are clearly overrepresented:

log\_or %>%

mutate(sentiment = reorder(sentiment, log\_or),) %>%

ggplot(aes(x = sentiment, ymin = conf.low, ymax = conf.high)) +

geom\_errorbar() +

geom\_point(aes(sentiment, log\_or)) +

ylab("Log odds ratio for association between Android and sentiment") +

coord\_flip()

We see that the disgust, anger, negative sadness and fear sentiments are associated with the Android in a way that is hard to explain by chance alone. Words not associated to a sentiment were strongly associated with the iPhone source, which is in agreement with the original claim about hyperbolic tweets.

If we are interested in exploring which specific words are driving these differences, we can back to our android\_iphone\_or object:

android\_iphone\_or %>% inner\_join(nrc) %>%

filter(sentiment == "disgust" & Android + iPhone > 10) %>%

arrange(desc(or))

We can make a graph:

android\_iphone\_or %>% inner\_join(nrc, by = "word") %>%

mutate(sentiment = factor(sentiment, levels = log\_or$sentiment)) %>%

mutate(log\_or = log(or)) %>%

filter(Android + iPhone > 10 & abs(log\_or)>1) %>%

mutate(word = reorder(word, log\_or)) %>%

ggplot(aes(word, log\_or, fill = log\_or < 0)) +

facet\_wrap(~sentiment, scales = "free\_x", nrow = 2) +

geom\_bar(stat="identity", show.legend = FALSE) +

theme(axis.text.x = element\_text(angle = 90, hjust = 1))