TEXT AS DATA: WEEK 5

MATTHIAS HABER

06 OCTOBER 2021

GOALS FOR TODAY

GOALS

- Review homework
- String manipulation and regular expressions

HOMEWORK FROM LAST WEEK

CONNECTING TO AN API

```
library(jsonlite)
data <- fromJSON(paste0(
  "https://www.abgeordnetenwatch.de/api/v2/",
  "candidacies-mandates?",
  "politician=175456"))
df <- data$data %>%
  filter(type == "candidacy") %>%
  as_tibble()
```

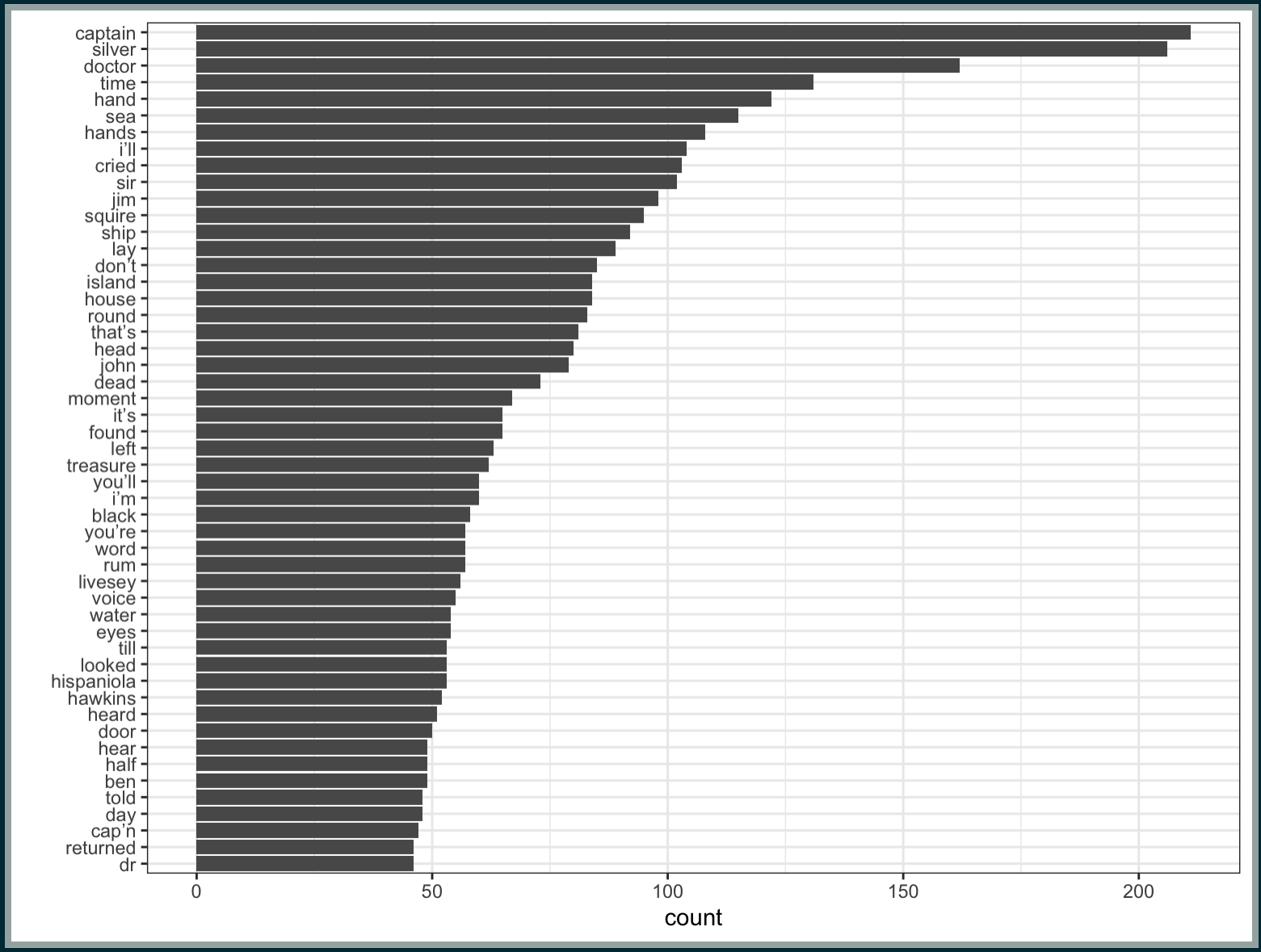
TOP 50 WORDS IN TREASURE ISLAND

```
library(gutenbergr)
library(tidytext)
data("stop_words")
treasure_island <- gutenberg_download(120, mirror =
        "http://mirrors.xmission.com/gutenberg/")
treasure_island_top <- treasure_island %>%
  unnest_tokens(word, text) %>%
  anti_join(stop_words) %>%
  count(word, sort = TRUE) %>%
  top_n(50)
```

PLOT TOP 50 WORDS

```
library(ggplot2)
treasure_island_top %>%
  ggplot(aes(y = reorder(word, n), x = n)) +
  geom_col() +
  labs(y = "", x = "count") +
  theme_bw() +
  theme(rect = element_rect(fill = "transparent"))
```

PLOT TOP 50 WORDS



STRING MANIPULATION WITH STRINGR

PACKAGES

```
library(tidyverse)
library(stringr) #install.packages("stringr")
library(htmlwidgets) #install.packages("htmlwidgets")
```

CREATING STRINGS

Strings are wrapped in ' or " quotes:

```
string1 <- "This is a string"
string2 <- 'If I want to include a "quote" inside
a string, I use single quotes'
```

You can use \ to “escape” single or double quotes inside a string:

```
double_quote <- "\"" # or '"'
single_quote <- '\'' # or "'"
```

SPECIAL CHARACTERS

- `\n` newline
- `\r` carriage return
- `\t` tab
- `\b` backspace
- `\a` alert (bell)
- `\f` form feed
- `\v` vertical tab
- `\\` backslash \

FUNCTIONS FOR STRINGS

```
# Length
str_length("Text as Data")
```

```
## [1] 12
```

```
# Combining strings
str_c("Text","as Data", sep = " ")
```

```
## [1] "Text as Data"
```

FUNCTIONS FOR STRINGS

```
#Subsetting strings
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
```

```
## [1] "App" "Ban" "Pea"
```

```
str_sub(x, -3, -1)
```

```
## [1] "ple" "ana" "ear"
```

FUNCTIONS FOR STRINGS

```
# Changing case
str_to_upper(c("a", "b"))
```

```
## [1] "A" "B"
```

```
str_to_lower(c("A", "B"))
```

```
## [1] "a" "b"
```

FUNCTIONS FOR STRINGS

`stringr` has 43 functions to manipulate strings. If you need more, then use `stringi`, which has 232 functions. The main difference between the functions in both packages is the prefix: `str_` vs. `stri_`.

REGULAR EXPRESSIONS

REGULAR EXPRESSIONS

Regular Expressions (regex) are a language or syntax to search in texts. Regex are used by most search engines in one form or another and are part of almost any programming language. In R, many string functions in `base` R as well as in `stringr` package use regular expressions, even Rstudio's search and replace allows regular expression.

REGULAR EXPRESSION SYNTAX

Regular expressions typically specify characters to seek out, possibly with information about repeats and location within the string. This is accomplished with the help of meta characters that have specific meaning:

- `$ * + . ? [ ] ^ { } | ( ) \.`

STRING FUNCTIONS RELATED TO REGULAR EXPRESSION

- To identify match to a pattern
  - `grep(..., value = FALSE)`, `grepl()`, `stringr::str_detect()`
- To extract match to a pattern
  - `grep(..., value = TRUE)`, `stringr::str_extract()`, `stringr::str_extract_all()`
- To locate pattern within a string
  - `regexpr()`, `gregexpr()`, `stringr::str_locate()`, `stringr::str_locate_all()`

MORE STRING FUNCTIONS RELATED TO REGULAR EXPRESSION

- To replace a pattern
  - `sub()`, `gsub()`, `stringr::str_replace()`, `stringr::str_replace_all()`
- To split a string using a pattern
  - `strsplit()`, `stringr::str_split()`

PATTERN MATCHING

The simplest patterns match exact strings:

```
x <- c("apple", "banana", "pear")
str_view(x, "an")
```

PATTERN MATCHING

`.` matches any character (except a newline):

```
str_view(x, ".a.")
```

apple

banana

pear

POSITION OF PATTERN WITHIN THE STRING

^ matches the start of the string.

```
x <- c("apple", "banana", "pear")
str_view(x, "^a")
```

POSITION OF PATTERN WITHIN THE STRING

$ matches the end of the string.

```
str_view(x, "a$")
```

apple

banana

pear

\b matches the empty string at either edge of a *word*.
\B matches the empty string provided it is not at an edge of a word.

QUANTIFIERS

Quantifiers specify the number of repetitions of the pattern.

- `*`: matches at least 0 times.
- `+`: matches at least 1 times.
- `?`: matches at most 1 times.
- `{n}`: matches exactly n times.
- `{n,}`: matches at least n times.
- `{n,m}`: matches between n and m times.

QUANTIFIERS

```
strings <- c("a", "ab", "acb", "accb", "acccb", "accccb")
grep("ac*b", strings, value = TRUE)
```

```
## [1] "ab"     "acb"    "accb"   "acccb"  "accccb"
```

QUANTIFIERS

```
grep("ac+b", strings, value = TRUE)
```

```
## [1] "acb"    "accb"   "acccb"  "accccb"
```

QUANTIFIERS

```
grep("ac?b", strings, value = TRUE)
```

```
## [1] "ab"  "acb"
```

QUANTIFIERS

```
grep("ac{2}b", strings, value = TRUE)
```

```
## [1] "accb"
```

QUANTIFIERS

```
grep("ac{2,}b", strings, value = TRUE)
```

```
## [1] "accb"   "acccb"  "accccb"
```

QUANTIFIERS

```
grep("ac{2,3}b", strings, value = TRUE)
```

```
## [1] "accb"  "acccb"
```

MORE OPERATORS

- `\`: suppress the special meaning of meta characters in regular expression,
  i.e. `$ * + . ? [ ] ^ { } | ( ) \`, similar to its usage in escape sequences. Since `\` itself needs to be escaped in R, we need to escape these meta characters with double backslash like `\\$`.
- `[...]`: a character list, matches any one of the characters inside the square brackets. We can also use `–` inside the brackets to specify a range of characters.
- `[^...]`: an inverted character list, similar to `[...]`, but matches any characters **except** those inside the square brackets.

MORE OPERATORS

- `|`: an “or” operator, matches patterns on either side of the `|`.
- `(...)`: grouping in regular expressions which allows to retrieve the bits that matched various parts of your regular expression. Each group can than be refer using `\\N`, with N being the No. of `(...)` used. This is called **backreference**.

OPERATORS

```
strings <- c("^ab", "ab", "abc", "abd", "abe", "ab 12")
grep("ab[c-e]", strings, value = TRUE)
```

```
## [1] "abc" "abd" "abe"
```

OPERATORS

```
grep("ab[^c]", strings, value = TRUE)
```

```
## [1] "abd"   "abe"   "ab 12"
```

OPERATORS

```
grep("^ab", strings, value = TRUE)
```

```
## [1] "ab"    "abc"   "abd"   "abe"   "ab 12"
```

OPERATORS

```
grep("\\^ab", strings, value = TRUE)
```

```
## [1] "^ab"
```

OPERATORS

```
grep("abc|abd", strings, value = TRUE)
```

```
## [1] "abc" "abd"
```

OPERATORS

```
gsub("(ab) 12", "\\1 34", strings)
```

```
## [1] "^ab"   "ab"    "abc"   "abd"   "abe"   "ab 34"
```

CHARACTER CLASSES

Character classes allow to specify classes such as numbers, letters, etc. There are two flavors of character classes, one uses `[:` and `:]` around a predefined name and the other uses `\` and a special character:

- `[:digit:]` or `\d`: digits, 0 1 2 3 4 5 6 7 8 9, equivalent to `[0-9]`.
- `\D`: non-digits, equivalent to `[^0-9]`.
- `[:lower:]`: lower-case letters, equivalent to `[a-z]`.
- `[:upper:]`: upper-case letters, equivalent to `[A-Z]`.
- `[:alpha:]`: alphabetic characters, equivalent to `[[:lower:][:upper:]]` or `[A-z]`.
- `[:alnum:]`: alphanumeric characters, equivalent to `[[:alpha:][:digit:]]` or `[A-z0-9]`.

COMMON CHARACTER CLASSES

- `\w`: word characters, equivalent to `[[:alnum:]_]` or `[A-z0-9_]`.
- `\W`: not word, equivalent to `[^A-z0-9_]`.
- `[:xdigit:]`: hexadecimal digits (base 16), 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f, equivalent to `[0-9A-Fa-f]`.
- `[:blank:]`: blank characters, i.e. space and tab.
- `[:space:]`: space characters: tab, newline, vertical tab, form feed, carriage return, space.
- `\s`: space, .
- `\S`: not space.
- `[:punct:]`: punctuation characters, ! " # $ % & ’ ( ) - + , - . / : ; < = > ? @ [ ] ^ _ ` { | } ~.

GENERAL MODES FOR PATTERNS

There are different syntax standards for regular expressions, and R offers two: POSIX extended regular expressions (default) and Perl-like regular expressions.

You can easily switch between by specifying `perl = FALSE/TRUE` in `base` R functions, such as `grep()` and `sub()`. For functions in the `stringr` package, wrap the pattern with `perl()`.

GENERAL MODES FOR PATTERNS

By default, pattern matching is case sensitive in R, but you can turn it off with `ignore.case = TRUE` (base R functions). In `stringr`, you need to pass the `ignore.case = TRUE` inside a modifier function like `str_replace(string, regex(pattern, ignore.case=`

Alternatively, you can use `tolower()` and `toupper()` functions to convert everything to lower or upper case.

REGULAR EXPRESSION VS SHELL GLOBBING

The term globbing refers to pattern matching based on wildcard characters. A wildcard character can be used to substitute for any other character or characters in a string. Globbing is commonly used for matching file names or paths, and has a much simpler syntax. Below is a list of globbing syntax and their comparisons to regular expression:

- `*`: matches any number of unknown characters, same as `.*` in regular expression.
- `?`: matches one unknown character, same as `.` in regular expression.

DETECT MATCHES

To determine if a character vector matches a pattern, use `str_detect()`. It returns a logical vector the same length as the input:

```
x <- c("apple", "banana", "pear")
str_detect(x, "e")
```

```
## [1]  TRUE FALSE  TRUE
```

DETECT MATCHES

```
# How many common words start with b?
sum(str_detect(words, "^b"))
```

```
## [1] 58
```

```
# What proportion of common words end with a vowel?
mean(str_detect(words, "[aeiou]$"))
```

```
## [1] 0.2765306
```

DETECT MATCHES

A variation on `str_detect()` is `str_count()`: rather than a simple yes or no, it tells you how many matches there are in a string:

```
x <- c("apple", "banana", "pear")
str_count(x, "a")
```

```
## [1] 1 3 1
```

```
# On average, how many vowels per word?
mean(str_count(words, "[aeiou]"))
```

```
## [1] 1.991837
```

REPLACE MATCHES

str_replace()andstr_replace_all()` allow you to replace matches with new strings. The simplest use is to replace a pattern with a fixed string:

```
x <- c("apple", "pear", "banana")
str_replace(x, "[aeiou]", "-")
```

```
## [1] "-pple"  "p-ar"   "b-nana"
```

```
str_replace_all(x, "[aeiou]", "-")
```

```
## [1] "-ppl-"  "p--r"   "b-n-n-"
```

REPLACE MATCHES

With `str_replace_all()` you can perform multiple replacements by supplying a named vector:

```
x <- c("1 house", "2 cars", "3 people")
str_replace_all(x, c("1" = "one", "2" = "two", "3" = "three"))
```

```
## [1] "one house"    "two cars"     "three people"
```

STRING SPLITTING

Use `str_split()` to split a string up into pieces.

```
"a|b|c|d" %>%
  str_split("\\|")
```

```
## [[1]]
## [1] "a" "b" "c" "d"
```

RESOURCES

- Regular expression in R official document.
- Perl-like regular expression: regular expression in perl manual.
- `qdapRegex` package: a collection of handy regular expression tools
- On these websites, you can simply paste your test data and write regular expression, and matches will be highlighted.
  - regexpal
  - RegExr

Test your skills solving regexp crosswords at
https://regexcrossword.com/challenges/beginner

EXERCISES

EXERCISES

1. Given the corpus of common words in `stringr::words`, create regular expressions that find all words that:
    1. Start with “y”.
    2. End with “x”
    3. Start with a vowel.
    4. That only contain consonants.
    5. End with `ed`, but not with `eed`.
    6. Start with three consonants.
    7. Have three or more vowels in a row.
    8. Have two or more vowel-consonant pairs in a row.

WRAPPING UP

QUESTIONS?

OUTLOOK FOR OUR NEXT SESSION

- Next week we will look at how to prepare text for actual analysis

THAT'S IT FOR TODAY

Thanks for your attention!