Intro to Text Mining: Strings and regular expressions

Filippo Chiarello, Ph.D.

Tet Mining in R

This chapter introduces you to string manipulation in R. You'll learn the basics of how strings work and how to create them by hand, but the focus of this chapter will be on **regular expressions**, or **regex** for short.

Load packages

```
library(stringr)
```

https://www.rdocumentation.org/packages/stringr/versions/1.4.0

String basics

```
string = "Hello world!"
writeLines(string)

## Hello world!

# escape character "
string = "Hello \"world\"!"
writeLines(string)

## Hello "world"!
```

String basics

```
# a vector of strings
s = c("Hello", "world", "!")
s
```

```
## [1] "Hello" "world" "!"
```

```
# string length
str_length("Hello")
```

```
## [1] 5
```

```
str_length(s)
```

```
## [1] 5 5 1
```

String basics

```
# combining strings
str_c("Hello", "world!")

## [1] "Helloworld!"

str_c("Hello", "world!", sep=" ")

## [1] "Hello world!"

# vectorized
str_c(c("1", "2", "3"), c("a", "b", "c"))

## [1] "1a" "2b" "3c"

str_c("prefix-", c("a", "b", "c"), "-suffix")

## [1] "prefix-a-suffix" "prefix-b-suffix" "prefix-c-suffix"
```

Regular expressions

- Regular expressions are useful because strings usually contain unstructured or semi-structured data.
- Regexps are a concise language for describing **patterns in strings**.
- When you first look at a regexp, you'll think a *cat walked across your keyboard*, but as your understanding improves they will soon start to make sense.

Basics

Basic regular expressions are build sequences of characters, including some special characters like:

- `.`: matches any character.
- `\d`: matches any digit.
- `\s`: matches any whitespace (e.g. space, tab, newline).
- `[abc]`: matches a, b, or c.
- `[^abc]`: matches anything except a, b, or c.
- `^` matches the start of the string.
- `$` matches the end of the string.

Remember, to create a regular expression containing `\d` or `\s`, you'll need to escape the `\` for the string, so you'll type `"\\d"` or `"\\s"`.

Regex

```
# a famous sentence
Gregory <- "To move is to stir, and to be valiant is to stand: therefore, if thou art moved, th

# exact match
str_view_all(Gregory, "move")
```

```
To move is to stir, and to be valiant is to stand:
therefore, if thou art moved, thou runn'st away.
```

Notice that matches never overlap

```
str_view_all("abababa", "aba")
```

any character

```
# any character
str_view_all(Gregory, "m.v.")
```

```
To move is to stir, and to be valiant is to stand:
therefored, thou runn'st away.
```

any digit

```
str_view_all("1 is not prime, but 2 is a prime!", "\\d")
```

```
1 is not prime, but 2 is a prime!
```

any whitespace

```
str_view_all(Gregory, "u\\s")
```

```
To move is to stir, and to be valiant is to stand:
therefore, if thou art moved, thou runn'st away.
```

any character in a set

```
str_view_all(Gregory, "[aeiou]\\s")
```

```
To move is to stir, and to be valiant is to stand:
therefore, if thou art moved, thou runn'st away.
```

any character not in a set

```
str_view_all(Gregory, "[^aeiou]\\s")
```

```
To move is to stir, and to be valiant is to stand:
therefore, if thou art moved, thou runn's t away.
```

start of the string

```
str_view(Gregory, "^..")
```

```
To move is to stir, and to be valiant is to stand:
therefore, if thou art moved, thou runn'st away.
```

end of the string

```
str_view_all(Gregory, "..$")
```

```
To move is to stir, and to be valiant is to stand:
therefore, if thou art moved, thou runn'st away.
```

Union and repetition

More complicated regular expressions are build from basic expressions using union and repetition operators:

- $|$: unioun
- $?$: 0 or 1
- $+$: 1 or more
- $*$: 0 or more

```
# union
str_view_all(Gregory, "move|stand")
```

To move is to stir, and to be valiant is to stand:
therefore, if thou art moved, thou runn'st away.

zero or many

Notice that quantifiers are greedy because they match as many characters as possible

```
CCCP <- c("P", "CP", "CCP", "CCCP")
str_view(CCCP, "CC*")
```

```
P
CP
CCP
CCCP
```

one or many

```
str_view(CCCP, "CC+")
```

```
P

CP

CCP

CCCP
```

zero or one

```
str_view(CCCP, "CC?")
```

```
P
CP
CCP
CCCP
```

exactly n

```
str_view(CCCP, "C{2}")
```

```
P
```

```
CP
```

```
CCP
```

```
CCCP
```

at least n

```
str_view(CCCP, "C{2,}")
```

```
P
```

```
CP
```

```
CCP
```

```
CCCP
```

between n and m

```
str_view(CCCP, "C{2,3}")
```

```
P
CP
CCP
CCCP
```

Tools

Now that you've learned the basics of regular expressions, it's time to learn how to apply them to real problems. In this section you'll learn a wide array of stringr functions that let you:

- **detect** which strings match a pattern
- **count** matches
- **extract** the content of matches
- **replace** matches with new values
- **split** a string based on a match

Detect and count

```
str_detect("apple", "p")
```

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

```
str_count("apple", "p")
```

```
## [1] 2
```

```
str_detect(c("apple", "banana", "pear"), "e")
```

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

```
str_count(c("apple", "banana", "pear"), "a")
```

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

Extract

```
str_extract(Gregory, "mov.?")
```

```
## [1] "move"
```

```
str_extract_all(Gregory, "mov.?")
```

```
## [[1]]
## [1] "move" "move"
```

Replace

```
# replace the first match
str_replace("apple", "[aeiou]", "-")

## [1] "-pple"

# replace all matches
str_replace_all("apple", "[aeiou]", "-")

## [1] "-ppl-"
```

Split

```
str_split("banana", "n")

## [[1]]
## [1] "ba" "a"  "a"

str_split(Gregory, "\\s")

## [[1]]
##  [1] "To"         "move"       "is"         "to"
##  [5] "stir,"      "and"        "to"         "be"
##  [9] "valiant"    "is"         "to"         "stand:"
## [13] "therefore," "if"         "thou"       "art"
## [17] "moved,"     "thou"       "runn'st"    "away."
```