Text Mining

Hope Hennessy

2025-09-15

Text Mining with `stringr` and Regular Expressions

Text mining is the process of extracting useful patterns, insights, and information from text data. This is essential because text is unstructured, inconsistent, and often noisy. Examples of text mining tasks include:

- Cleaning survey responses (`Yes`, `yes!`, `Y` → standardized to `Yes`).
- Extracting emails, hashtags, phone numbers, or IDs from a dataset.
- Preparing text for Natural Language Processing (NLP) tasks like sentiment analysis or topic modeling.

Text analysis generally involves two steps:

1. Preprocessing & Cleaning: Standardizing, trimming, splitting, and extracting relevant text.
2. Pattern Matching & Feature Engineering: Using regex and string functions to identify useful features for modeling.

This notebook introduces:

1. The basics of string manipulation with the `stringr` package.
2. Regular expressions (regex) for pattern matching.
3. Practical examples of cleaning and preparing real-world text data.

1. Why Use `stringr`?

R has many built-in string manipulation functions, but they can be inconsistent in syntax. The `stringr` package (part of the tidyverse) provides:

- A consistent set of functions, all starting with `str_`.
- Vectorized functions (they work on entire columns of data).
- Functions designed to integrate smoothly with data frames, `dplyr`, and other tidyverse tools.

```r
library(tidyverse)
```

2. String Basics

Strings are sequences of characters stored as text. They are enclosed in:

- Single quotes: `'This is a string'`
- Double quotes: `"This is also a string"`

```r
string <- "This is a string"
string
```

```
## [1] "This is a string"
```

```r
str(string)    # Displays structure of the object
```

```
##  chr "This is a string"
```

```r
length(string) # Number of elements (here, 1)
```

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

```r
nchar(string)  # Base R: number of characters in the string
```

```
## [1] 16
```

```r
str_length(string) # stringr equivalent
```

```
## [1] 16
```

Important distinction:

- `length()` counts the number of elements (like rows in a vector).
- `str_length()` counts the number of characters in each string.

```r
string <- "He said, \"Hello!\""

print(string)      # Shows quotes
```

```
## [1] "He said, \"Hello!\""
```

```r
cat(string)        # No quotes, no newline
```

```
## He said, "Hello!"
```

```r
writeLines(string) # No quotes, adds newline
```

```
## He said, "Hello!"
```

Escaping Characters

Some characters have special meaning in strings or regex and must be escaped using a backslash `\`.

Example:

```r
string <- "He said, \"Hello!\""
writeLines(string)
```

```
## He said, "Hello!"
```

To include a backslash itself, escape it twice:

```r
writeLines("C:\\Users\\Documents")
```

```
## C:\Users\Documents
```

Special Meta-Characters

These characters represent actions like tabs or newlines:

Character	Meaning
`\t`	Tab
`\n`	Newline
`\r`	Carriage return

```r
writeLines("Column1\tColumn2\nValue1\tValue2")
```

```
## Column1  Column2
## Value1   Value2
```

Combining & Extracting Substrings

Combine multiple strings: `str_c()`

```r
str_c("Data", "Science", sep = " ")
```

```
## [1] "Data Science"
```

```r
str_c("Hello", "World", sep = ", ")
```

```
## [1] "Hello, World"
```

Extract parts of strings by position: `str_sub()`

```r
url <- "http://example.com/index.html"
str_sub(url, 1, 4)   # First four characters
```

```
## [1] "http"
```

```r
str_sub(url, -5, -1) # Last five characters
```

```
## [1] ".html"
```

Negative indices count from the end of the string.

Case Conversion

```r
str_to_upper("data science")
```

```
## [1] "DATA SCIENCE"
```

```r
str_to_lower("DATA Science")
```

```
## [1] "data science"
```

Trimming Whitespace

Extra whitespace is common in messy datasets:

```r
str_trim("   messy text   ")
```

```
## [1] "messy text"
```

Tip: Always trim text before comparing or grouping values, especially in survey data.

Vectorization

Most `stringr` functions automatically process vectors element-wise:

```r
str_to_upper(c("data", "science", "rocks"))
```

```
## [1] "DATA"    "SCIENCE" "ROCKS"
```

This makes them ideal for working with data frame columns.

3. Regular Expressions (Regex)

Regular expressions are a mini-language for pattern matching within text. They allow you to:

- Detect whether text fits a certain format.
- Extract specific sequences like emails, IDs, or phone numbers.
- Clean or standardize messy text efficiently.

Basic Matching

```r
x <- c("apple", "banana", "pear", "28.50", "Probability 0", "test123")
```

Match exact patterns:

```r
str_view(x, "ap")
```

```
## [1] | <ap>ple
```

Special Character Classes

Pattern	Matches
`\\d`	Any digit `[0-9]`
`\\w`	Any word character `[A-Z & a-z & 0-9 & _]`
`\\s`	Any whitespace (space, tab, newline)

Negations (capitalized):

Pattern	Meaning
`\\D`	Non-digit
`\\W`	Non-word character
`\\S`	Non-whitespace

Why double backslashes?

- The first \ escapes for R.
- The second \ is read by the regex engine.

Example:

```r
str_view(x, "\\d")
```

```
## [4] | <2><8>.<5><0>
## [5] | Probability <0>
## [6] | test<1><2><3>
```

```r
str_view(x, "\\D")
```

```
## [1] | <a><p><p><l><e>
## [2] | <b><a><n><a><n><a>
## [3] | <p><e><a><r>
## [4] | 28<.>50
## [5] | <P><r><o><b><a><b><i><l><i><t><y>< >0
## [6] | <t><e><s><t>123
```

```r
z <- c("Hello", "World123", "user_id", "no-spaces!", "123")

# Detect strings containing word characters
str_detect(z, "\\w")
```

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

Character Sets and Wildcards

Syntax	Description	Example
`[abc]`	Match a single character: a OR b OR c	`str_view(x, "[ap]")`
`[^abc]`	Match anything EXCEPT a, b, or c	`str_view(x, "[^b]a")`
`.`	Match any single character	`str_view(x, "a.")`
`[A-Z]`	Any uppercase letter	`str_view(x, "[A-Z]")`
`[0-9]`	Any digit	`str_view(x, "[0-9]")`

```
str_view(x, "[ap]")
```

```
## [1] | <a><p><p>le
## [2] | b<a>n<a>n<a>
## [3] | <p>e<a>r
## [5] | Prob<a>bility 0
```

```
str_view(x, "[ap][pe]")
```

```
## [1] | <ap>ple
## [3] | <pe>ar
```

```
str_view(x, "[^b]a")
```

```
## [2] | ba<na><na>
## [3] | p<ea>r
```

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

```
## [1] | <ap>ple
## [2] | b<an><an>a
## [3] | pe<ar>
## [5] | Prob<ab>ility 0
```

```
str_view(x, "[A-Z]")
```

```
## [5] | <P>robability 0
```

Repetitions (Quantifiers)

Symbol	Meaning	Example
`+`	1 or more	`"a+"` matches `"a"`, `"aa"`
`*`	0 or more	`"ba*"` matches `"b"`, `"baaa"`
`?`	0 or 1 (optional)	`"colou?r"` matches `"color"` and `"colour"`
`{n}`	Exactly n	`"\\d{3}"` matches 123
`{n,}`	n or more	`"\\d{2,}"` matches 12, 12345
`{n,m}`	Between n and m	`"\\d{2,4}"` matches 12, 123, 1234

Example:

```
str_view(x, '\\w{2,3}') # any alphanum between 2 and 3 times
```

```
## [1] | <app><le>
## [2] | <ban><ana>
## [3] | <pea>r
## [4] | <28>.<50>
## [5] | <Pro><bab><ili><ty> 0
## [6] | <tes><t12>3
```

```r
str_view(x, '\\w{2,}') # any alphanum at least 2
```

```
## [1] | <apple>
## [2] | <banana>
## [3] | <pear>
## [4] | <28>.<50>
## [5] | <Probability> 0
## [6] | <test123>
```

```r
str_view(x, '[an]{4}') # four consecutive characters, each of which can be an a or n
```

```
## [2] | b<anan>a
```

```r
str_view(x, 'a[na]+')
```

```
## [2] | b<anana>
```

```r
str_view(x, 'a[na]*')
```

```
## [1] | <a>pple
## [2] | b<anana>
## [3] | pe<a>r
## [5] | Prob<a>bility 0
```

```r
str_view('apple', 'ap?p')
```

```
## [1] | <app>le
```

```r
str_view('apple', 'an?p')
```

```
## [1] | <ap>ple
```

Anchors

Anchor	Description
^	Start of string
$	End of string

Examples:

```r
str_view(x, '^a')
```

```
## [1] | <a>pple
```

```r
str_view(x, 'a$')
```

```
## [2] | banan<a>
```

```r
str_view(x, '^a.*e$')
```

```
## [1] | <apple>
```

```r
str_view(x, "\\d$") # ends with a digit
```

```
## [4] | 28.5<0>
## [5] | Probability <0>
## [6] | test12<3>
```

Grouping

Parentheses create logical groupings and precedence:

```r
str_view(c("I love cats", "I love dogs"), "I love (cats|dogs)")
```

```
## [1] | <I love cats>
## [2] | <I love dogs>
```

```r
str_view('I love cats', 'I love (cats|dogs)')
```

```
## [1] | <I love cats>
```

```r
str_view('I love birds', 'I love (cats|dogs)')
str_view('I love dogs', 'I love cats|dogs')
```

```
## [1] | I love <dogs>
```

4. Common stringr Functions

Pattern Detection

The `str_detect()` function can be used to determine whether an input string contains a specified pattern. Where the input is a vector, `str_detect()` checks whether each element in turn contains the pattern i.e. it returns a logical vector of the same length as the input.

Determine whether a string matches a pattern:

```r
str_detect(x, "\\d")
```

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

Filter matching values:

```r
# Base R
x[str_detect(x, "\\d")] # elements of x containing a digit
```

```
## [1] "28.50"         "Probability 0" "test123"
```

```r
x[!str_detect(x, '\\d')] # elements of x not containing a digit
```

```
## [1] "apple"  "banana" "pear"
```

```r
str_subset(x, "\\d")    # tidyverse shortcut
```

```
## [1] "28.50"         "Probability 0" "test123"
```

```r
str_which(x, '\\d')     # extracts indices of elements matching a pattern
```

```
## [1] 4 5 6
```

Counting Matches

Count how many times a pattern appears:

```r
str_count(x, "\\d")
```

```
## [1] 0 0 0 4 1 3
```

Example: count spaces in sentences:

```r
sentences <- c("Hello world", "R is fun")
str_count(sentences, " ")
```

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

Extracting Matches

First match: `str_extract()` All matches: `str_extract_all()`

```
str_extract(x, "\\d+")
```

```
## [1] NA    NA    NA    "28"  "0"   "123"
```

```
str_extract_all(x, "\\d+")
```

```
## [[1]]
## character(0)
##
## [[2]]
## character(0)
##
## [[3]]
## character(0)
##
## [[4]]
## [1] "28" "50"
##
## [[5]]
## [1] "0"
##
## [[6]]
## [1] "123"
```

```
str_extract(x, 'a[nb]')
```

```
## [1] NA   "an" NA   NA   "ab" NA
```

```
str_extract_all(x, 'a[nb]')
```

```
## [[1]]
## character(0)
##
## [[2]]
## [1] "an" "an"
##
## [[3]]
## character(0)
##
## [[4]]
## character(0)
##
## [[5]]
## [1] "ab"
##
## [[6]]
## character(0)
```

Replacing Text

Replace matched patterns with new text:

```r
str_replace(x, "[aeiou]", "-")
```

```
## [1] "-pple"          "b-nana"         "p-ar"           "28.50"
## [5] "Pr-bability 0" "t-st123"
```

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

```
## [1] "-ppl-"          "b-n-n-"         "p--r"           "28.50"
## [5] "Pr-b-b-l-ty 0" "t-st123"
```

Use case: Clean punctuation or standardize inconsistent text.

Splitting Strings

Divide text into parts based on a delimiter:

```r
str_split("Data Science with R", " ")
```

```
## [[1]]
## [1] "Data"    "Science" "with"    "R"
```

```r
str_split("2025-09-15", "-")
```

```
## [[1]]
## [1] "2025" "09"   "15"
```

Example: splitting CSV-like data:

```r
str_split("A,B,C,D", ",")
```

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

5. Real-World Use Cases

Cleaning Survey Data

Messy responses often contain extra spaces or symbols:

```r
responses <- c("Yes ", " yes", "YES!", "no", "No ", "N/A")

clean_responses <- responses %>%
  str_to_lower() %>%
  str_trim() %>%
  str_replace_all("[^a-z]", "") # remove non-letters

clean_responses
```

```
## [1] "yes" "yes" "yes" "no"  "no"  "na"
```

Extracting Email Addresses

```r
emails <- c("Contact: test@domain.com", "support@company.org", "no-email")
str_extract(emails, "[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}")
```

```
## [1] "test@domain.com"     "support@company.org" NA
```

Detecting Hashtags

```r
tweets <- c("Loving #datascience", "#rstats is amazing", "no tags here")
str_subset(tweets, "#\\w+")
```

```
## [1] "Loving #datascience" "#rstats is amazing"
```

Tokenizing Text

Splitting sentences into individual words:

```r
text <- "Text mining is fun!"
tokens <- str_split(text, "\\s+")[[1]]
tokens
```

```
## [1] "Text"   "mining" "is"     "fun!"
```

Validating IDs or Codes

Ensure entries follow a strict pattern:

```r
ids <- c("AB123", "A1", "XYZ789")
str_detect(ids, "^[A-Z]{2,3}\\d{3}$")
```

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

Advanced Data Cleaning Example

```r
# Messy customer data
messy_customer_data <- c(
  "John Doe (555) 123-4567 john@email.com",
  "JANE SMITH 555.987.6543 jane@company.org",
  "Bob Johnson  (555)456-7890  bob123@test.net")

# Extract names (everything before the first parenthesis or digit)
names <- str_extract(messy_customer_data, "^[A-Za-z\\s]+")
names <- str_trim(names) %>% str_to_title()

# Extract phone numbers
phones <- str_extract(messy_customer_data, "\\(?\\d{3}\\)?[.-]?\\s?\\d{3}[.-]?\\d{4}")

# Extract emails
emails <- str_extract(messy_customer_data, "[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}")

# Create clean data frame
clean_customer_data <- data.frame(
  name = names,
  phone = phones,
  email = emails)

print(clean_customer_data)
```

```
##          name          phone            email
## 1    John Doe (555) 123-4567   john@email.com
## 2  Jane Smith   555.987.6543 jane@company.org
## 3 Bob Johnson  (555)456-7890  bob123@test.net
```

Log File Processing

```r
# Sample log entries
log_entries <- c(
  "2025-09-15 10:30:15 ERROR Database connection failed",
  "2025-09-15 10:31:22 INFO User logged in successfully",
  "2025-09-15 10:32:08 WARNING Low disk space detected")

# Extract components using regex
dates <- str_extract(log_entries, "\\d{4}-\\d{2}-\\d{2}")
times <- str_extract(log_entries, "\\d{2}:\\d{2}:\\d{2}")
levels <- str_extract(log_entries, "(ERROR|INFO|WARNING)")
messages <- str_extract(log_entries, "(?<=ERROR |INFO |WARNING ).*")

# Create structured log data
log_data <- data.frame(
  date = dates,
  time = times,
  level = levels,
  message = messages)

print(log_data)
```

```
##         date     time   level                     message
## 1 2025-09-15 10:30:15   ERROR  Database connection failed
## 2 2025-09-15 10:31:22    INFO User logged in successfully
## 3 2025-09-15 10:32:08 WARNING     Low disk space detected
```

Text Validation Functions

```r
# Create validation functions for common data types

# Email validation
validate_email <- function(email) {
  pattern <- "^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}$"
  str_detect(email, pattern)
}

# Phone validation (US format)
validate_phone <- function(phone) {
  pattern <- "\\(?\\d{3}\\)?[.-]?\\s?\\d{3}[.-]?\\d{4}"
  str_detect(phone, pattern)
}

# Date validation (YYYY-MM-DD format)
validate_date <- function(date) {
  pattern <- "^\\d{4}-\\d{2}-\\d{2}$"
  str_detect(date, pattern)
}

# Test the validation functions
test_data <- c(
  "john@email.com",
  "invalid-email",
  "(555) 123-4567",
  "555-invalid",
  "2025-09-15",
  "invalid-date"
)

tibble(
  text = test_data,
  is_email = validate_email(test_data),
  is_phone = validate_phone(test_data),
  is_date = validate_date(test_data))
```

```
## # A tibble: 6 x 4
##   text           is_email is_phone is_date
##   <chr>          <lgl>    <lgl>    <lgl>
## 1 john@email.com TRUE     FALSE    FALSE
## 2 invalid-email  FALSE    FALSE    FALSE
## 3 (555) 123-4567 FALSE    TRUE     FALSE
## 4 555-invalid    FALSE    FALSE    FALSE
## 5 2025-09-15     FALSE    FALSE    TRUE
## 6 invalid-date   FALSE    FALSE    FALSE
```

6. Performance Tips and Best Practices

```
# For repeated operations, compile regex patterns
email_pattern <- regex("[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}",
                       ignore_case = TRUE)

# Use vectorized operations instead of loops
large_text <- rep(messy_customer_data, 100)

# Efficient approach
system.time({
  results <- str_extract(large_text, email_pattern)
})
```

```
##    user  system elapsed
##    0.00    0.00    0.02
```

```
# Handle missing values gracefully
text_with_na <- c("john@email.com", NA, "jane@company.org", "")

safe_extract <- function(text, pattern) {
  if_else(is.na(text) | text == "",
          NA_character_,
          str_extract(text, pattern))
}

safe_extract(text_with_na, "[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}")
```

```
## [1] "john@email.com"   NA                 "jane@company.org" NA
```

7. Summary Table

Task	Function	Example
Detect pattern	`str_detect()`	`str_detect(x, "\\d")`
Filter matches	`str_subset()`	`str_subset(x, "\\d")`
Count matches	`str_count()`	`str_count(x, "\\d")`
Extract text	`str_extract()`, `str_extract_all()`	`str_extract(x, "\\d+")`
Replace text	`str_replace()`, `str_replace_all()`	`str_replace(x, "a", "-")`
Split text	`str_split()`	`str_split("a,b,c", ",")`
Trim whitespace	`str_trim()`, `str_squish()`	`str_trim("  text  ")`
Change case	`str_to_upper()`, `str_to_lower()`	`str_to_upper("text")`

8. Common Regex Patterns Quick Reference

```
# Common patterns for validation and extraction
patterns <- tribble(
  ~Description, ~Pattern, ~Example,
  "Email", "[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}", "user@domain.com",
  "Phone (US)", "\\(?\\d{3}\\)?[.-]?\\s?\\d{3}[.-]?\\d{4}", "(555) 123-4567",
  "Date (YYYY-MM-DD)", "\\d{4}-\\d{2}-\\d{2}", "2025-09-15",
  "URL", "https?://[A-Za-z0-9.-]+\\.[A-Za-z]{2,}", "https://example.com",
  "Hashtag", "#\\w+", "#datascience",
  "Mention", "@\\w+", "@username",
  "Currency", "\\$\\d+\\.?\\d*", "$19.99",
  "ZIP Code", "\\d{5}(-\\d{4})?", "12345-6789")

print(patterns)
```

```
## # A tibble: 8 x 3
##   Description       Pattern                                          Example
##   <chr>             <chr>                                            <chr>
## 1 Email             "[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}" user@doma~
## 2 Phone (US)        "\\(?\\d{3}\\)?[.-]?\\s?\\d{3}[.-]?\\d{4}"       (555) 123~
## 3 Date (YYYY-MM-DD) "\\d{4}-\\d{2}-\\d{2}"                           2025-09-15
## 4 URL               "https?://[A-Za-z0-9.-]+\\.[A-Za-z]{2,}"         https://e~
## 5 Hashtag           "#\\w+"                                          #datascie~
## 6 Mention           "@\\w+"                                          @username
## 7 Currency          "\\$\\d+\\.?\\d*"                                $19.99
## 8 ZIP Code          "\\d{5}(-\\d{4})?"                               12345-6789
```

9. Exercises

1. Clean a column of messy text (extra spaces, inconsistent capitalization, and punctuation).
2. Write a regex to extract:
   - Phone numbers like `(123) 456-7890`.
   - Twitter handles like `@username`.
3. Detect and extract dates in the format `YYYY-MM-DD`.
4. Count how many hashtags appear in each tweet.
5. Use `str_detect()` to filter rows that contain two consecutive digits.
6. Create a function that standardizes phone numbers to the format `(XXX) XXX-XXXX`.
7. Parse a log file and extract IP addresses, timestamps, and error codes.

```
# Exercise 6 solution: Phone number standardization
standardize_phone <- function(phone) {
```

```r
  # Remove all non-digits
  digits_only <- str_replace_all(phone, "[^\\d]", "")

  # Check if we have exactly 10 digits
  if (str_length(digits_only) == 10) {
    # Format as (XXX) XXX-XXXX
    formatted <- str_replace(digits_only, "(\\d{3})(\\d{3})(\\d{4})", "(\\1) \\2-\\3")
    return(formatted)
  } else {
    return(phone)  # Return original if not valid
  }
}

# Test the function
test_phones <- c("5551234567", "(555) 123-4567", "555.123.4567", "invalid")
standardized <- map_chr(test_phones, standardize_phone)

tibble(original = test_phones, standardized = standardized)
```

```
## # A tibble: 4 x 2
##   original       standardized
##   <chr>          <chr>
## 1 5551234567     (555) 123-4567
## 2 (555) 123-4567 (555) 123-4567
## 3 555.123.4567   (555) 123-4567
## 4 invalid        invalid
```