Text Mining and Sentiment Analysis

Prof. Annamaria Bianchi
A.Y. 2024/2025

Lecture 3

24 February 2025

UNIVERSITÀ DEGLI STUDI DI BERGAMO | Dipartimento di Scienze Economiche

Outline

Encoding

Regular expressions

Package: stringr

Functions: `stringr::str_view()`,

Encoding

A computer cannot store "letters", "numbers", "pictures" or anything else. The only thing it can store and work with are *bits*. A bit can only have two values: yes or no, true or false, 1 or 0. The same representation (byte) can be reported in octal, decimal, or hexadecimal numbers.

In R we can obtain the underlying representation of a string using `charToRaw()`:

```
> charToRaw("string")
[1] 73 74 72 69 6e 67
```

Each of these six hexadecimal numbers represents one letter. The mapping from hexadecimal number to character is the encoding, which in this case is CP-1252. This works well for English.

Things aren't so easy for Languages other than English. In the early days of computing, there were many competing standards for encoding non-English characters (ASCII – American Standard Code for Information Interchange, Latin1 used for Western European languages, Latin2 for Central European languages.

b1 → Latin1 ± → Latin2 ą

Encoding

Today there is one standard that is supported almost everywhere: **UTF-8** (Unicode Transformation Format, 8 bit), which can encode about every character used by humans today and many extra symbols (e.g., emoji)

readr uses UTF-8 everywhere. This is a good default but it will fail for data produced by older systems that don't use UTF-8. If this happens your stings will look weird when you print them.

How do you find the correct encoding? If you are lucky, it will be included somewhere in the data documentation. However, that's rarely the case. **readr** provides `guess_encoing()` to help you figure it out.

Regular expressions

Regular expressions are a tool that allows us to work with text by describing text patterns.

A **regular expression** is a pattern that describes a set of strings

Regular expressions are an essential tool in text cleaning and text analysis.

With regular expressions you can look for specific patterns. They are used to string detecting, locating, extracting, matching, replacing, and splitting.

Because the term “regular expression” is rather long, most people use the word **regex** as a shortcut term.

What are regular expressions used for?

- We use regular expressions to work with text.
- Some of its common uses involve:
  - test if a phone number has the correct number of digits,
  - if a date follows a specific format (e.g. mm/dd/yy),
  - if an email address is in a valid format, or if a password has numbers and special characters.
  - search a document for *gray* spelt either as “gray” or “grey”.
  - search a document and replace all occurrences of “Will”, “Bill”, or “W.” with William.
  - count the number of times in a document that the word “analysis” is immediately preceded by the words “data”, “computer” or “statistical” only in those cases.
  - convert a comma-delimited file into a tab-delimited file or find duplicate words in a text.
- In each of these cases, you are going to use a regular expression to write a description of what you are looking for using symbols. Once you have defined a pattern then the regex processor will use our description to return matching results, or in the case of the test, to return true or false for whether or not it matched.

Regular expressions basics

- The **main purpose** of working with regular expressions is to describe patterns that are used to match against text strings. Simply put, working with regular expressions is nothing more than **pattern matching**. The result of a match is either successful or not.

- The **simplest version** of pattern matching is to search for one occurrence (or all occurrences) of some specific characters in a string. For example, we might want to search for the word "programming" in a large text document, or we might want to search for all occurrences of the string "apply" in a series of files containing R scripts.

- Typically, regular expression patterns consist of a **combination of alphanumeric characters as well as special characters**. A regex pattern can be as simple as a single character, or it can be formed by several characters with a more complex structure. In all cases we construct regular expressions much in the same form in which we construct arithmetic expressions, by using various operators to combine smaller expressions.

Function `str_view()`

To have a **visual representation** of the actual pattern that is matched, we will use the function `str_view()` from the package **stringr**.

This function takes a character vector and a regular expression and shows how they match. This is shown in the **Viewer window**, with HTML rendering of regular expression match.

```
str_view(string, pattern = NULL, match = T, html = F,…)
```

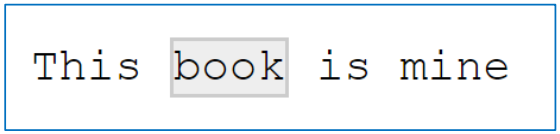
`string`	Input vector
`pattern`	Pattern to look for.
`match`	Which elements should be shown? TRUE, the default, shows only elements that match the pattern. NA shows all elements. FALSE shows only elements that don't match the pattern.
`html`	If TRUE will create an HTML widget; if FALSE will style using ANSI escapes. Package htmlwidgets necessary

Exact match

```
> library(tidyverse)
> install.packages("htmlwidgets")
> library(htmlwidgets)
```

The simplest patterns match exact strings:

```
> sentence1 = 'This book is mine'
> str_view(sentence1, 'book')
[1] | This <book> is mine
> str_view(sentence1, 'book', html = T)
```



Exact match

Regex searches are **case sensitive** by default. This means that the pattern "Book" would not match *book* in *sentence1*

```
> str_view(sentence1, 'Book')
```

By default, it shows all occurrences

```
> sentence2 <- 'This book is mine. I wrote this book with word.'
> str_view(sentence2, "book")
[1] | This <book> is mine. I wrote this <book> with word.
```

Exact match

Option `match`. By default, only matching strings are shwon

```
> books = c('Book', 'book')
> str_view(books, "book")
[2] | <book>
```

It is possible to show all the elements

```
> str_view(books, "book", match = NA)
[1] | Book [2] | <book>
```

It is also possible to show only non-matching elements

```
> str_view(books, "book", match = F)
[1] | Book
```

Exact match

`str_view()` is a vectorial function

```
> x = c("apple", "banana", "pear")
> str_view(x, "an", html = T, match = NA)
```

apple

banana

pear

```
> str_view(x, "a")
[1] | <a>pple
[2] | b<a>n<a>n<a>
[3] | pe<a>r
```

Match any character

In order to build more complex patterns, there are a number of **metacharacters** that have specific meaning.

The simplest example of a metacharacter is the **full stop**. The full stop character matches any single character of any sort (apart from a newline).

```
> str_view(x, ".a.")
[2] | <ban>ana
[3] | p<ear>
nn <- c("not", "note", "knot", "nut")
> str_view(nn, "n.t")
[1] | <not>
[2] | <not>e
[3] | k<not>
[4] | <nut>
```

```
> str_view(nn, "no.")
[1] | <not>
[2] | <not>e
[3] | k<not>
```

Metacharacters

List of metacharacters

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

- the dot `.`
- the backslash `\`
- the bar `|`
- left or opening parenthesis `(`
- right or closing parenthesis `)`
- left or opening bracket `[`
- right or closing bracket `]`
- left or opening brace `{`
- right or closing brace `}`
- the dollar sign `$`
- the dash, hyphen or minus sign `-`
- the caret or hat `^`
- the star or asterisk `*`
- the plus sign `+`
- the question mark `?`

Match any character

In order to build more complex patterns, there are a number of **metacharacters** that have specific meaning.

The simplest example of a metacharacter is the **full stop**. The full stop character matches any single character of any sort (apart from a newline).

```
> str_view(x, ".a.")
[2] | <ban>ana
[3] | p<ear>
nn <- c("not", "note", "knot", "nut")
> str_view(nn, "n.t")
[1] | <not>
[2] | <not>e
[3] | k<not>
[4] | <nut>
```

```
> str_view(nn, "no.")
[1] | <not>
[2] | <not>e
[3] | k<not>
```

Escaping metacharacters

Question: If '.' matches any character, how do you match the character '.'?

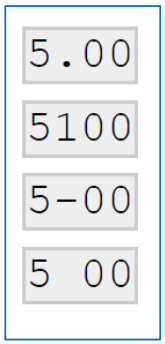
You need to use an 'escape' to tell the regular expression you want to match it exactly and not use its special behaviour.

Suppose we want to match ''5.00'' in the following vector

```
> fives <- c("5.00", "5100", "5-00", "5 00")
```

First we try

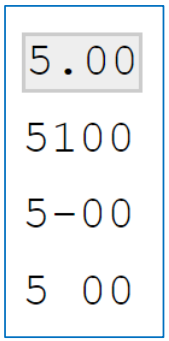
```
> str_view(fives, "5.00", html = T)
```



Escaping metacharacters

To actually match the dot character, what you need to do is **escape** the metacharacter.
In regular expressions in R, the way to escape a metacharacter is by adding **two backslash characters ** in front of the metacharacter: "\\.".
When you use two backslashes in front of a metacharacter you are "escaping" the character, this means that the character no longer has a special meaning, and it will match itself.

```
> str_view(fives, "5\\.00", html = T, match = NA)
```



Escaping metacharacters

Exercise. Create the vector containing the elements etc., e.g., i.e., that, etcetera, (etc).

1. Create a regular expression to match all strings containing at least one dot.
2. Create a regular expression to match all strings containing a parenthesis.

Anchors

By default, regular expressions will match any part of a string. It is often useful to anchor the regular expression so that it matches from the **start** or end of the **string**.

Anchors are metacharacters that help us assert the position, say, the beginning or end of the string:

- ^ matches the start of the string
- $ matches the end of the string.

```
> x [1] "apple" "banana" "pear"
> str_view(x, "^a", html = T, match = NA)
```

apple
banana
pear

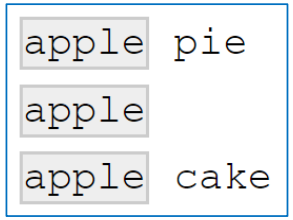
```
> str_view(x, "a$", html = T, match = NA)
```

apple
banana
pear

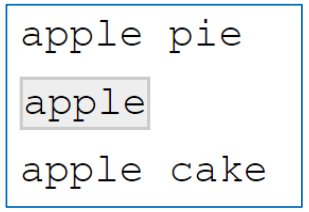
Anchors

To force a regular expression to only match a complete string, anchor it with both ^ and $:

```
> xx = c("apple pie", "apple", "apple cake")
> str_view(xx, "apple")
```




```
> str_view(xx, "^apple$")
```



Anchors

You can also match the boundary between words (i.e. the start or end of a word) with `\b`.

For example, to find all uses of sum(), you can use the regex \bsum\b to avoid matching summarise, summary, rowsum, etc.:

```
> y <- c("summary(x)", "summarise(x)", "rowsum(x)", "sum(x)")
> str_view(y, "sum")
[1] | <sum>mary(x)
[2] | <sum>marise(x)
[3] | row<sum>(x)
[4] | <sum>(x)
> str_view(y, "\\bsum\\b") [4] | <sum>(x)
```

Alternation

The metacharacter | can be used to pick between one or more alternative patterns. For example, ab|de will match either ‘ab’ or ‘de’. In some cases it might be useful to use parentheses.

Assume I want to identify all the strings containg ‘grey’ or ‘gray’

```
> str_view(c("grey", "gray"), "gr(e|a)y")
```

grey

gray

Character classes

A character class allows to match any character in a set. A character set will match only one character. The order of the characters inside the set does not matter; what matter is just the presence of the characters inside the brackets.

You can construct your own sets with `[ ]`. For example, `[abc]` matches a, b, or c. There are three characters that have special meaning inside `[]`:

- - defines a range; e.g., `[a-z]` matches any lowercase letter, and [0-9] matches any number;
- ^ allows to invert the match, e.g. `[^abc]` matches any character except a, b, or c;
- \ escapes special characters; e.g. `[\^\-\]]` matches ^, -, or].

Character classes

```
> pns <- c("pan", "pen", "pin", "pon", "pun")
> str_view(pns, "p[aeiou]n")
```

```
pan
pen
pin
pon
pun
```

```
> pnx <- c("pan", "pen", "pin", "p0n", "p.n", "p1n", "paun")
> str_view(pnx, "p[aeiou]n")
```

```
pan
pen
pin
p0n
p.n
p1n
paun
```

Character classes

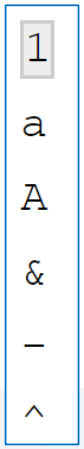
If you are interested in matching all capital letters in English, you can define a set formed as:

```
[ABCDEFGHIJKLMNOPQRSTUVWXYZ]
```

Character ranges give a convenient shortcut based on the dash metacharacter "-" to indicate a range of characters. A character range consists of a character set with two characters separated by a dash or minus "-" sign.
For example, the set of all capital letters can be expressed as a character range using the following pattern:

```
[A-Z]
```

```
> basic <- c("1", "a", "A", "&", "-", "^")
> basic
[1] "1" "a" "A" "&" "-" "^"
> # digits
> str_view(basic, "[0-9]")
```



Character classes

```
> # lower case letters
> str_view(basic, "[a-z]")
```

```
1
a
A
&
–
^
```

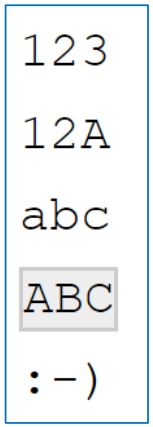
```
> # upper case letters
> str_view(basic, "[A-Z]")
```

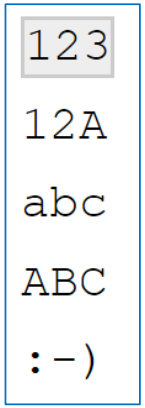
```
1
a
A
&
–
^
```

Character classes

You can use a series of character ranges to match various occurrences of a certain type of character. For example, to match three consecutive digits you can define a pattern `"[0-9][0-9][0-9]"`; to match three consecutive lower case letters you can use the pattern `"[a-z][a-z][a-z]"`; and the same idea applies to a pattern that matches three consecutive upper case letters `"[A-Z][A-Z][A-Z]"`.

```
> triplets <- c("123", "12A", "abc", "ABC", ":-)")
> str_view(triplets, "[0-9][0-9][0-9]")
> str_view(triplets, "[A-Z][A-Z][A-Z]")
```





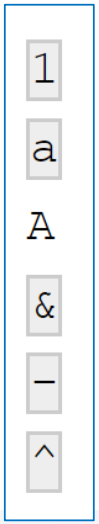
Character classes

A common situation when working with regular expressions consists of matching characters that are **NOT** part of a certain set. This type of matching can be done using a **negative character set**: by matching any one character that is not in the set. To define this type of sets you are going to use the metacharacter caret "^"

For example, `[^abc]` matches anything except a, b, or c.

To match those elements that are NOT upper case letters, you define a negative character range `[^A-Z]`

```
> str_view(basic, '[^A-Z]')
```



Character classes

Some character classes are used so often that they get their own shortcut. There are three particularly useful pairs:

- \d matches any digit

  \D matches anything that is not a digit
- \s matches any whitespace

  \S matches anything that is not whitespace
- \w matches any «word» character, i.e. letters and numbers

  \W matches any «non-word» character

Character classes

```
> pnx
[1] "pan" "pen" "pin" "p0n" "p.n"
[6] "p1n" "paun"
> str_view(pnx, "\\w", html = T, match = NA)
```

pan
pen
pin
p0n
p.n
p1n
paun

```
> str_view(pnx, "\\d", html = T, match = NA)
```

pan
pen
pin
p0n
p.n
p1n
paun

Class exercise

Consider the character vector `stringr::words` and take a look at it.
Create regular expressions to find all words in the `stringr::words` sample vector

1. containing an «x» sorrounded by vowels
2. containing a «y» sorrounded by consonants.

Quantifiers

Quantifiers **control how many times a pattern matches**. Quantifiers can refer to characters, groups or character classes and should be placed **after** the element that is being quantified.

- ?: the preceding item is optional and will be matched at most once (0 or 1)
- +: the preceding item will be matched one or more times
- *: the preceding item will be matched zero or more times
- {n}: the preceding item is matched exactly *n* times
- {n,}: the preceding item is matched *n* or more times
- {,m}: the preceding item is matched at most *m* times
- {n,m}: the preceding item is matched at least *n* times, but not more than *m* times (between *n* and *m*)

Quantifiers

```
> gg = c("Gooooal", "Goooooal")
> str_view(gg, "Go{4}al")
[1] | <Gooooal>
> str_view(gg, "Go+al")
[1] | <Gooooal>
[2] | <Goooooal>
```

Class exercise

1) Create a vector of strings (named student_names) containing the names: Francesca, Luca, Andrea, Tommaso, Lucia, Mara, Anna Chiara.
2) Identify the names that contain more than 4 characters and less than 7 characters.
3) Identify the names that contain e or u

Exercise for you

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. Are exactly three letters long.
4. Have seven letters or more.
5. Start with a vowel.
6. Only contain consonants.
7. End with `ed`, but not with `eed`.
8. Start with three consonants.
9. Have three or more vowels
10. Have two or more vowel-consonant pairs in a row.

Since the `word` list is long, you might want to use the `match` argument to `str_view()` to show only the matching or non-matching words.