Regular expressions

Max Turgeon

SCI 2000–Introduction to Data Science

Lecture Objectives

- Understand the definition of regular expressions
- Recognize and use the different metacharacters
- Use regular expressions to filter and edit text data

Motivation

- As we have seen, data is often encoded using text.
    - `M/F` or `minivan/suv/pickup`
- When we are lucky, this data is free of errors and coded exactly the way we need it for data analysis.
- More often than not, there are typos or inconsistencies in the data that need to be addressed.

Regular expression–Definition

- A **regular expression** (or **regex**) is a sequence of characters that specify a search pattern.
- In other words, a regex is a pattern that we want to find in a string of text.
- Common applications of regexes include:
    - finding observations where a certain word appears
    - replacing a string by another one
    - splitting a string according to a certain pattern

Examples i

```r
library(stringr)
# Detect a pattern
str_detect(c("apple", "orange", "pineapple"),
           pattern = "apple")
```

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

```r
# Careful: This is case-sensitive
str_detect(c("Apple", "Orange", "Pineapple"),
           pattern = "apple")
```

Examples ii

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

```
# To ignore case
str_detect(c("Apple", "Orange", "Pineapple"),
           pattern = regex("apple",
                           ignore_case = TRUE))
```

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

```
# Match one of the patterns
str_detect(c("color", "colour", "coulour"),
           pattern = "color|colour")
```

Examples iii

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

```
# Or more compact
str_detect(c("color", "colour", "coulour"),
           pattern = "col(o|ou)r")
```

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

Examples iv

```
# Replace patterns
str_replace_all(c("Male", "male", "MaLe"),
                pattern = regex("male",
                                ignore_case = TRUE),
                replacement = "male")
```

```
## [1] "male" "male" "male"
```

Examples v

```
# Split a string using a pattern
str_split(c("it is a sentence", "it is another one"),
          pattern = " ")
```

```
## [[1]]
## [1] "it"       "is"       "a"        "sentence"
##
## [[2]]
## [1] "it"      "is"      "another" "one"
```

Anchors

- **Anchors** are special characters (i.e. metacharacters) that can be used to specify *where* we want to find a match.
- There are two main anchors:
    - `^pattern` will match any string that *starts* with `pattern`
    - `pattern$` will match any string that *ends* with `pattern`
- You can combine them:
    - `^pattern$` will *only* match the string `pattern`
- If you want to match on a metacharacter (e.g. `$`), you need to escape it (see example below).

Example i

```r
# This doesn't work...
str_detect(c("$15.99", "$3.75", "1.99$"),
           pattern = "^$")
```

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

```r
# But this does!
str_detect(c("$15.99", "$3.75", "1.99$"),
           pattern = "^\\$")
```

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

Example ii

```r
# Matching on prices that start or end with dollar sign
# Use logical operator
str_detect(c("$15.99", "$3.75", "1.99$"),
           pattern = "^\\$") |
  str_detect(c("$15.99", "$3.75", "1.99$"),
           pattern = "\\$$")
```

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

Example iii

```r
# Or more compact
str_detect(c("$15.99", "$3.75", "1.99$"),
           pattern = "^\\$|\\$$")
```

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

Quantifiers

- **Quantifiers** are ways to specify how many times a certain pattern should appear.
    - At least once? Exactly three times?
- There are four important metacharacters to remember.
    - `.` will match any single character, except a new line.
    - `?` will match the item on its left at most once.
    - `*` will match the item on its left zero or more times.
    - `+` will match the item on its left once or more times.
- Key distinction between `*` and `+`
    - the latter requires at least one match.

Example i

```
# Revisiting an earlier example
str_detect(c("color", "colour", "coulour"),
           pattern = "colou?r")
```

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

```
# Matching strings that end with a bunch of periods
str_detect(c("str", "str.", "str..", "str..."),
           pattern = "\\.+$")
```

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

Quantifiers cont'd

- You can also control the number of matches more precisely.
  - `{n}` will match the item on its left exactly n times.
  - `{n,}` will match the item on its left at least n times.
  - `{n,m}` will match the item on its left at least n times, but no more than m times.

Exercise

Find a regular expression that matches string ending with an ellipsis (i.e. three dots).

Solution

```r
str_detect(c("string.", "string..", "string..."),
           pattern = "\\.{3}$")
```

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

```r
# Be careful: a string with 4 dots will also match
str_detect("string....", pattern = "\\.{3}$")
```

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

Character classes i

- When discussing quantifiers, I used “item on the left” instead of “character on the left”.
- This was intentional: these items could also be **character classes**.
- We can create them using square brackets.
    - E.g. `p[ao]rt` will match both `part` and `port`.
- Character classes can also be created using *sequences*.
    - `[a-z]` will match all lower case letters
    - `[a-zA-Z]` will match all lower and upper case letters
    - `[0-9]` will match all ten digits

Character classes ii

- There are also built-in character classes:
  - `\\d` matches any digit character (equivalent to `[0-9]`)
  - `\\s` matches any space character (including tabs, new lines, etc.)
  - `\\w` matches any word character (equivalent to `[A-Za-z0-9_]`)
  - `\\b` matches word boundaries
- Finally, you can negate character classes to get non-matches.
  - `p[^ao]rt` matches `purt` and `pert`
  - The negation of `\\d`, `\\s`, `\\w`, `\\b` are `\\D`, `\\S`, `\\W`, `\\B` respectively.

Example i

```
# Split a sentence into words
str_split("The fox ate a berry.", "\\b")
```

```
## [[1]]
## [1] "" "The" " " "fox" " " "ate" " " "a" " "
## [10] "berry" "."
```

```
str_split("The fox ate a berry.", "\\s")
```

```
## [[1]]
## [1] "The" "fox" "ate" "a" "berry."
```

Example ii

```
# Trim white space
str_replace_all(”Is  this     enough?”,
                pattern = ”\\s+”,
                replacement = ” ”)
```

```
## [1] ”Is this enough?”
```

Exercise

Find a regular expression that matches white space at the beginning and the end of a string.

Solution

```r
str_replace_all(" Is this enough?   ",
                pattern = "(^\\s+|\\s+$)",
                replacement = "")
```

```
## [1] "Is this enough?"
```

Summary

- Regular expressions are patterns that we want to search within a string.
- Anchors and quantifiers are metacharacters that allow us to be quite specific about the type of matches we want.
    - Remember: to match on a metacharacter literally, you need to escape it!
- Most modern implementations of regular expressions also have **lookaround operators**, which we won't cover. But look it up if you're interested!

Example i

```
library(tidyverse)
library(dslabs)
glimpse(movielens)
```

```
## Rows: 100,004
## Columns: 7
## $ movieId <int> 31, 1029, 1061, 1129, 1172,
1263, 1287, 1293, 1339, 1343, 13~
## $ title <chr> "Dangerous Minds", "Dumbo",
"Sleepers", "Escape from New Yor~
## $ year <int> 1995, 1941, 1996, 1981, 1989,
```

Example ii

```
1978, 1959, 1982, 1992, 1991, ~
## $ genres <fct> Drama,
Animation|Children|Drama|Musical, Thriller,
Action|Ad~
## $ userId <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
## $ rating <dbl> 2.5, 3.0, 3.0, 2.0, 4.0, 2.0,
2.0, 2.0, 3.5, 2.0, 2.5, 1.0, ~
## $ timestamp <int> 1260759144, 1260759179,
1260759182, 1260759185, 1260759205, ~
```

Example iii

```
# How many horror movie reviews
movielens %>%
  filter(str_detect(genres, "Horror")) %>%
  nrow()
```

```
## [1] 6790
```

```r
# What proportion are about thrillers?
movielens %>%
  mutate(thriller = str_detect(genres,
                               "Thriller")) %>%
  summarise(prop = mean(thriller))
```

```
##        prop
## 1 0.2523899
```

```
# What genre is Forrest Gump?
movielens %>%
  filter(str_detect(title, "Gump")) %>%
  pull(genres) %>%
  unique %>%
  str_split(pattern = "\\|")
```

```
## [[1]]
## [1] "Comedy"  "Drama"   "Romance" "War"
```

Exercise

The dataset `reported_heights` in the `dslabs` package contains self-reported heights in no specific format. Clean up the data by making all heights comparable.

This is a challenging exercise. Try to do as much as you can.

(Partial) solution i

- The first thing to do is to decide which units we will use.
- Looking at the data, it seems that most people reported their height in inches.
    - Therefore, let's use inches as our unit of measurement.
- Next, we need to look at the data and find heights that don't seem to be in inches.
- For example, some heights are a single digit, which is probably the height in feet.
    - To find these heights, we can use `^\\d$` as our regex.
    - Once found, we need to convert to an integer and multiply by 12.

(Partial) solution ii

```
library(tidyverse)
library(dslabs)
library(stringr)

reported_heights %>%
  filter(str_detect(height, "^\\d$")) %>%
  count(height)
```

(Partial) solution iii

```
##   height  n
## 1      0  1
## 2      1  3
## 3      2  1
## 4      5  4
## 5      6 19
## 6      7  1
```

- We can see some errors already.
    - A height of 0 or 1 is probably a mistake.
    - A height of 2 is either a mistake or 2 meters. We will assume the latter.

(Partial) solution iv

```
data_clean1 <- reported_heights %>%
  filter(!height %in% c(0, 1)) %>% # Remove mistakes
  filter(str_detect(height, "^\\d$")) %>%
  mutate(height_in = if_else(height == "2",
                             78.75,
                             12 * as.numeric(height)))
nrow(data_clean1)
```

```
## [1] 25
```

(Partial) solution v

- We can look for other patterns by looking at heights that are not written as one or two digits.
    - The regex we want to use is `^\\d{1,2}$`

```
reported_heights %>%
    filter(!str_detect(height, "^\\d{1,2}$")) %>%
  head()
```

(Partial) solution vi

```
##               time_stamp    sex height
## 1 2014-09-02 15:16:28   Male  5' 4”
## 2 2014-09-02 15:16:31 Female  66.75
## 3 2014-09-02 15:16:32 Female    5.3
## 4 2014-09-02 15:16:37   Male   70.5
## 5 2014-09-02 15:16:37 Female  165cm
## 6 2014-09-02 15:16:41   Male    511
```

- For a complete-ish solution, see UM Learn.