R para Ciência de Dados 2

Manipulando texto (stingr)

agosto de 2020

Motivação

Bases com colunas em texto já são *extremamente* comuns hoje em dia, então saber lidar com strings se torna essencial na caixa de ferramentas do cientista de dados

Além de ajudar em análise de dados, tratar strings ajuda com programação porque grande parte das linguagens modernas funcionam da mesma maneira que o R nesse quesito

O conhecimento de expressões regulares vale para a vida, é impossível descrever com poucas palavras todas as coisas que são implementáveis via regex

Normalmente os textos são bagunçados, independentemente do quão cuidadosa foi a coleta de dados, então precisamos arrumá-los; podemos fazer isso do jeito fácil (`{stringr}` e regex) ou do jeito difícil (`{base}` e lágrimas)

Introdução

- Strings não passam sequências de caracteres ("cadeias" em português)
- No R podemos criar uma string com um par de aspas (simples ou duplas)
- O `print()` mostra a estrutura da string, enquanto `cat()` mostra o texto

```
print("こんにちは! Está 10\u00BAC na lá fora")
```

```
#> [1] "こんにちは! Está 10ºC na lá fora"
```

- Para colocar aspas dentro de uma string, podemos **escapar** o caractere

```
cat("Ele disse \"escapar\"")
```

```
#> Ele disse "escapar"
```

O pacote {stringr}

- O pacote `{stringr}` é a forma mais simples de trabalhar com strings no R

```
library(stringr)

abc <- c("a", "b", "c")
str_c("prefixo-", abc, "-sufixo")
```

```
#> [1] "prefixo-a-sufixo" "prefixo-b-sufixo" "prefixo-c-sufixo"
```

- Todas as funções relevantes começam com `str_` e funcionam bem juntas

```
abc %>%
  str_c("-sufixo") %>%
  str_length()
```

```
#> [1] 8 8 8
```

Principais funções

Função(ões)	Significado
`str_c`	Colar strings
`str_length`	Contagem de caracteres na string
`str_detect`	O padrão existe na string?
`str_extract[_all]`	Extrair o padrão da string
`str_replace[_all]`	Substituir um padrão por outro na string
`str_remove[_all]`	Remover um padrão da string
`str_split`	Quebrar a string em pedaços
`str_squish`	Remover espaços extras da string
`str_sub`	Extrair um pedaço da string
`str_to_[lower/upper]`	Converter a string para caixa baixa/alta

Exemplos

```
str_detect("Colando Strings", pattern = "ando")
```

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

```
str_extract("Colando Strings", pattern = "ando")
```

```
#> [1] "ando"
```

```
str_replace("Colando Strings", pattern = "ando", replacement = "ei")
```

```
#> [1] "Colei Strings"
```

```
str_remove("Colando Strings", pattern = " Strings")
```

```
#> [1] "Colando"
```

Exemplos (cont.)

```
str_split("Colando Strings", pattern = " ")
```

```
#> [[1]]
#> [1] "Colando" "Strings"
```

```
str_squish("   Colando   Strings   ")
```

```
#> [1] "Colando Strings"
```

```
str_sub("Colando Strings", start = 1, end = 7)
```

```
#> [1] "Colando"
```

```
str_to_lower("Colando Strings")
```

```
#> [1] "colando strings"
```

Regex

- **Expressões regulares** são "programação para strings", permitindo extrair padrões bastante complexos com comandos simples
- Elas giram em torno de padrões "normais" de texto, mas com alguns símbolos especiais com significados específicos

```
frutas <- c("banana", "TANGERINA", "maçã", "lima")
str_detect(frutas, pattern = "na")
```

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

- Exemplos: `.` (qualquer caractere), ^ (início da string) e $ (fim da string)

```
str_detect(frutas, pattern = "^ma")
```

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

Mais regex

- Podemos contar as ocorrências de um padrão: `+` (1 ou mais vezes), `*` (0 ou mais vezes), `{m,n}` (entre `m` e `n` vezes), `?` (0 ou 1 vez)

```
ois <- c("oi", "oii", "oiii!", "oioioi!")
str_extract(ois, pattern = "i+")
```

```
#> [1] "i"   "ii"  "iii" "i"
```

- `[]` é um conjunto e `()` é um conjunto "inquebrável"

```
str_extract(ois, pattern = "[i!]$")
```

```
#> [1] "i" "i" "!" "!"
```

```
str_extract(ois, pattern = "(oi)+")
```

```
#> [1] "oi"     "oi"     "oi"     "oioioi"
```

Ainda mais regex

- Se de fato precisarmos encontrar um dos **caracteres reservados** descritos anteriormente, precisamos escapá-los da mesma forma como vimos antes

```
str_replace("Bom dia.", pattern = ".", replacement = "!")
```

```
#> [1] "!om dia."
```

```
str_replace("Bom dia.", pattern = "\\.", replacement = "!")
```

```
#> [1] "Bom dia!"
```

- Não esquecer que algumas funções do `{stringr}` possuem variações

```
str_replace_all("Bom. Dia.", pattern = "\\.", replacement = "!")
```

```
#> [1] "Bom! Dia!"
```

Exemplos intermináveis

```
str_subset(c("banana", "TANGERINA", "maçã", "lima"), "NA")  # Maiúscula
```

```
#> [1] "TANGERINA"
```

```
str_subset(c("banana", "TANGERINA", "maçã", "lima"), "^ma") # Início
```

```
#> [1] "maçã"
```

```
str_subset(c("banana", "TANGERINA", "maçã", "lima"), "ma$") # Final
```

```
#> [1] "lima"
```

```
str_subset(c("banana", "TANGERINA", "maçã", "lima"), ".m")  # Qualquer
```

```
#> [1] "lima"
```

Exemplos intermináveis (cont.)

```
str_extract(c("oii", "oiii!", "oiii!!!", "oioioi!"), "i+!")    # 1 ou mais
```

```
#> [1] NA     "iii!" "iii!" "i!"
```

```
str_extract(c("oii", "oiii!", "oiii!!!", "oioioi!"), "i+!?")   # 0 ou 1
```

```
#> [1] "ii"   "iii!" "iii!" "i"
```

```
str_extract(c("oii", "oiii!", "oiii!!!", "oioioi!"), "i+!*")   # 0 ou mais
```

```
#> [1] "ii"     "iii!"   "iii!!!" "i"
```

```
str_extract(c("oii", "oiii!", "oiii!!!", "oioioi!"), "i{1,2}") # Entre m e n
```

```
#> [1] "ii" "ii" "ii" "i"
```

Exemplos intermináveis (cont.)

```
str_extract(c("oii", "oiii!", "oiii!!!", "oioioi!"), "[i!]+")  # Algum
```

```
#> [1] "ii"     "iii!"   "iii!!!" "i"
```

```
str_extract(c("banana", "TANGERINA", "maçã", "lima"), "[a-z]") # Conjuntos
```

```
#> [1] "b" NA  "m" "l"
```

```
str_extract(c("oii", "oiii!", "oiii!!!", "oioioi!"), "(oi)+")  # Tudo
```

```
#> [1] "oi"     "oi"     "oi"     "oioioi"
```

```
str_extract(c("oii", "oiii!", "ola!!!", "oioioi!"), "(i+|!+)") # Ou
```

```
#> [1] "ii"  "iii" "!!!" "i"
```

Exemplos intermináveis (cont.)

```
str_replace("Bom dia.", "\\.", "!")      # Escapando
```

```
#> [1] "Bom dia!"
```

```
str_replace("Bom. Dia.", "\\.", "!")     # Primeira ocorrência
```

```
#> [1] "Bom! Dia."
```

```
str_replace_all("Bom. Dia.", "\\.", "!") # Lembrar do _all
```

```
#> [1] "Bom! Dia!"
```

```
str_remove_all("Bom \"dia\"", "\\\"")    # Escapando escape
```

```
#> [1] "Bom dia"
```

Exemplos intermináveis (cont.)

```
stringi::stri_trans_general("Váríös àçêntõs", "Latin-ASCII") # Remover acentos
```

```
#> [1] "Varios acentos"
```

```
str_extract_all("Número: (11) 91234-1234", "[0-9]+")         # Números
```

```
#> [[1]]
#> [1] "11"    "91234" "1234"
```

```
str_extract("Número: (11) 91234-1234", "[A-Za-z]+")          # Conjuntos juntos
```

```
#> [1] "N"
```

```
str_extract("Número: (11) 91234-1234", "[:alpha:]+")         # Acentos
```

```
#> [1] "Número"
```

R para Ciência de Dados 2

Manipulando datas (lubridate)

agosto de 2020

Motivação

É difícil encontrar um tipo de dado mais delicado do que datas (e horas): diferentemente de textos e erros de *encoding*, erros de *locale* podem passar desapercebidos e estragar uma análise inteira.

Operações com tempo são complicadas, pois envolvem precisão e diversos fatores que variam de um lugar para o outro (fuso horário, horário de verão, anos bissextos, formato da data, etc.).

Além das variações normais de como cada país escreve suas datas, cada computador tem seu jeito de interpretá-las e cada programa tem seu jeito de salvá-las.

Entender como é a representação de tempo dentro de linguagens de programação é muito valioso porque isso é um problema relevante independentemente da ferramenta sendo utilizada.

Introdução

- Como representar datas em um universo cheio de fusos e calendários diferentes? Estabelecendo um momento universal e contando os segundos que se passaram desde lá

```
library(lubridate)
now()
```

```
#> [1] "2020-08-27 22:10:25 -03"
```

```
as.numeric(now())
```

```
#> [1] 1598577026
```

- O formato padrão é denominado "Era UNIX" e conta o número de segundos desde o ano novo de 1970 em Londres (01/01/1970 00:00:00 UTC)

O pacote {lubridate}

- O pacote `{lubridate}` vai nos possibilitar trabalhar com datas e data-horas fora do ISO 8601 (`ano-mês-dia hora:minuto:segundo`)
- Para converter uma data-hora do formato brasileiro para o padrão universal, pensamos na ordem das unidades em inglês: *day*, *month*, *year*, *hour*, *minute*, *second*

```
dmy_hms("27/08/2020 02:25:00")
```

```
#> [1] "2020-08-27 02:25:00 UTC"
```

- Também é possível trabalhar só com datas usando a mesma lógica das unidades

```
dmy("27/08/2020")
```

```
#> [1] "2020-08-27"
```

Você acredita em mágica?

- O `{lubridate}` é tão poderoso que pode parecer mágica

```
dmy("27 de agosto de 2020", locale = "pt_BR.UTF-8") # No Win: Portuguese_Brazil.1252
```

```
#> [1] "2020-08-27"
```

```
mdy("August 27th 2020", locale = "en_US.UTF-8") # Mês-dia-ano
```

```
#> [1] "2020-08-27"
```

- Às vezes o Excel salva datas como o número de dias desde 01/01/1970, mas nem isso pode vencer o `{lubridate}`

```
as_date(18501)
```

```
#> [1] "2020-08-27"
```

Fusos

- É mais raro precisar lidar com fusos horários porque normalmente trabalhamos com data-horas de um mesmo fuso, mas o `{lubridate}` permite lidar com isso também

```
now(tzone = "Europe/London")
```

```
#> [1] "2020-08-28 02:10:25 BST"
```

- Nem o horário de verão consegue atrapalhar um cálculo preciso: com a função `dst()` é possível saber se em um dado dia aquele lugar estava no horário de verão

```
dst(now(tzone = "Europe/London"))
```

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

Componentes

- As funções `year()`, `month()`, `day()`... (**no singular**) podem extrair os componentes de uma data

```
month("2020-08-27")
```

```
#> [1] 8
```

- Obs.: Note como não foi necessário converter a string para data porque ela já está no formato esperado pelo `{lubridate}`

- As funções `years()`, `months()`, `days()`... (**no plural**) permitem fazer contas com datas e data-horas

```
now() + days(5)
```

```
#> [1] "2020-09-01 22:10:25 -03"
```

Operações

- Com os operadores matemáticos normais também somos capazes de calcular distâncias entre datas e horas

```
dif <- dmy("27/08/2020") - dmy("25/08/2020")
dif
```

```
#> Time difference of 2 days
```

- Podemos transformar um objeto de diferença temporal em qualquer unidade que queiramos usando as funções no plural

```
as.period(dif) / minutes(1)
```

```
#> [1] 2880
```

- Para diferenças entre data-horas pode ser importante usar os fusos

Exemplos intermináveis

```
dmy_hms("27/08/2020 02:25:30") # Data-hora
```

```
#> [1] "2020-08-27 02:25:30 UTC"
```

```
dmy_hm("27/08/2020 02:25")     # Sem segundo
```

```
#> [1] "2020-08-27 02:25:00 UTC"
```

```
dmy_h("27/08/2020 02")         # Sem minuto
```

```
#> [1] "2020-08-27 02:00:00 UTC"
```

```
as_datetime(1598539497)        # Numérico
```

```
#> [1] "2020-08-27 14:44:57 UTC"
```

Exemplos intermináveis (cont.)

```
mdy_hms("8/27/20 2:25:30 PM")                                    # Americano
```

```
#> [1] "2020-08-27 14:25:30 UTC"
```

```
dmy_hms("27/08/2020 02:25:30", tz = "Europe/London")             # Com fuso
```

```
#> [1] "2020-08-27 02:25:30 BST"
```

```
now() - dmy_hms("27/08/2020 02:25:30")                           # Diferença
```

```
#> Time difference of 22.74881 hours
```

```
now() - dmy_hms("27/08/2020 02:25:30", tz = "Europe/London") # Com fuso
```

```
#> Time difference of 23.74881 hours
```

Exemplos intermináveis (cont.)

```
minute("2020-08-27 02:25:30")                                    # Minuto
```

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

```
year("2020-08-27")                                               # Ano
```

```
#> [1] 2020
```

```
wday("2020-08-27")                                               # Dia da semana
```

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

```
month("2020-08-27", label = TRUE, abbr = FALSE, locale = "pt_BR.UTF-8") # Mês (sem abrev.)
```

```
#> [1] agosto
#> 12 Levels: janeiro < fevereiro < março < abril < maio < junho < julho < ... < dezembro
```

Exemplos intermináveis (cont.)

```
today() + months(5)                                  # Dia
```

```
#> [1] "2021-01-27"
```

```
now() + seconds(5)                                   # Segundo
```

```
#> [1] "2020-08-27 22:10:30 -03"
```

```
now() + days(5)                                      # Dia
```

```
#> [1] "2020-09-01 22:10:25 -03"
```

```
as.period(today() - dmy("01/01/2020")) / days(1) # Dia - dia
```

```
#> [1] 239
```

Exemplos intermináveis (cont.)

```
t1 <- dmy_hms("27/08/2020 02:25:00", tz = "America/Sao_Paulo")
t2 <- dmy_hms("27/08/2020 02:25:00")
t1 - t2
```

```
#> Time difference of 3 hours
```

```
t1 <- dmy_hms("27/08/2020 02:25:00", tz = "America/Sao_Paulo")
t2 <- dmy_hms("27/08/2020 02:25:00", tz = "Europe/London")
t1 - t2
```

```
#> Time difference of 4 hours
```

```
head(OlsonNames())
```

```
#> [1] "Africa/Abidjan"     "Africa/Accra"       "Africa/Addis_Ababa"
#> [4] "Africa/Algiers"     "Africa/Asmara"      "Africa/Asmera"
```

R para Ciência de Dados 2

Manipulando listas (purrr)

agosto de 2020

Motivação

Além de lidar com listas, o `{purrr}` lida com iterações, um dos padrões mais comuns em qualquer tarefa de programação (independentemente da linguagem).

Apesar de não ser tão evidente, todos os data frames do R não passam de listas com algumas propriedades específicas, então saber lidar com elas pode ser útil em diversos lugares.

Ainda pode ser difícil entender como isso funciona, mas, além de números, strings e datas, é possível colocar listas nas colunas de um data frame.

A sintaxe do `{purrr}` é capaz de mudar para sempre o modo como você funciona, ensinando padrões robustos de programação (sem efeitos colaterais e sem repetição de código).

Introdução

- Listas são como vetores, com a diferença de que elas não precisam ser homogêneas (seus elementos podem ter qualquer tipo)

```
l <- list(
  um_numero = 123,
  um_vetor = c(TRUE, FALSE, TRUE),
  uma_string = "abc",
  uma_lista = list(1, 2, 3)
)
str(l)
```

```
#> List of 4
#>  $ um_numero : num 123
#>  $ um_vetor  : logi [1:3] TRUE FALSE TRUE
#>  $ uma_string: chr "abc"
#>  $ uma_lista :List of 3
#>   ..$ : num 1
#>   ..$ : num 2
#>   ..$ : num 3
```

Estrutura

```
l
```

```
#> $um_numero
#> [1] 123
#>
#> $um_vetor
#> [1]  TRUE FALSE  TRUE
#>
#> $uma_string
#> [1] "abc"
#>
#> $uma_lista
#> $uma_lista[[1]]
#> [1] 1
#>
#> $uma_lista[[2]]
#> [1] 2
#>
#> $uma_lista[[3]]
#> [1] 3
```

Indexação

- Para acessar os elementos de uma lista precisamos tomar cuidado com a diferença entre `[]` e `[[]]` (ou `purrr::pluck()`): o primeiro acesa uma posição, enquanto o segundo acessa um elemento

```
l[3]
```

```
#> $uma_string
#> [1] "abc"
```

```
l[[3]]
```

```
#> [1] "abc"
```

```
library(purrr)
pluck(l, 4, 2) # Indexação profunda
```

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

Iterações

- Iteração não é nada mais do que a repetição de um trecho de código várias vezes, normalmente associada a um *loop* (laço)

```
vec <- 1:5
for (i in seq_along(vec)) {
  vec[i] <- vec[i] + 10
}
vec
```

```
#> [1] 11 12 13 14 15
```

- Note como a única coisa que fazemos é aplicar uma operação em cada elemento do vetor (`vec[i] + 10`)
- Identificamos algumas estruturas: **entrada** (vetor de 1 a 5) e **função** (somar 10 a cada elemento)

Simplificando

- O pacote `{purrr}` nos permite simplificar iterações e integrá-las a pipelines do `{tidyverse}`: a função `map()` realiza uma iteração recebendo apenas uma entrada e uma função

```
vec <- 1:5
soma_dez <- function(x) {
  x + 10
}

l <- map(vec, soma_dez)
str(l)
```

```
#> List of 5
#>  $ : num 11
#>  $ : num 12
#>  $ : num 13
#>  $ : num 14
#>  $ : num 15
```

Achatamento

- `map()` sempre retorna uma lista independente do objeto recebido porque ela não pode assumir nada sobre o resultado
- Se quisermos achatar os resultados só precisamos chamar uma função da família `map_***()` (onde `***` é a abreviação do tipo do objeto que deve ser retornado)

```
map_dbl(vec, soma_dez)
```

```
#> [1] 11 12 13 14 15
```

- Os tipos possíveis são: `dbl` (números), `chr` (strings), `dfc` (*data frame columns*), `dfr` (*data frame rows*), `int` (inteiros) e `lgl` (lógicos)

```
map_chr(vec, soma_dez)
```

```
#> [1] "11.000000" "12.000000" "13.000000" "14.000000" "15.000000"
```

Funções

- Para passar outros argumentos **fixos** a uma função, basta adicioná-los ao final da chamada de `map()`

```
soma_n <- function(x, n) {
  x + n
}
map_dbl(vec, soma_n, n = 3)
```

```
#> [1] 4 5 6 7 8
```

- Para simplificar funções curtas, podemos usar uma notação **lambda** na qual `.x` representa onde deve ser inserido o elemento atual da iteração (o valor "iterante") e ~ indica a declaração da função

```
map_dbl(vec, ~3+.x)
```

```
#> [1] 4 5 6 7 8
```

Duas entradas

- Se for necessário iterar em duas listas ou vetores, basta usar `map2()`

```
strings <- c("oiii", "como vai", "tchau")
padroes <- c("i+",   "(.o){2}",  "[au]+$")
map2_chr(strings, padroes, stringr::str_extract)
```

```
#> [1] "iii"  "como" "au"
```

- A notação lambda funciona exatamente do mesmo modo, com `.x` e `.y` representando o primeiro e o segundo elementos da iteração

```
map2_chr(strings, padroes, ~stringr::str_c(.y, " | ", .x))
```

```
#> [1] "i+ | oiii"          "(.o){2} | como vai" "[au]+$ | tchau"
```

List-columns

- *List-columns* são colunas nas quais cada elemento é uma lista (ou até mesmo uma tabela completa)
- A função `str_split()`, por exemplo, retorna uma lista contendo os pedaços da string original quebrada com um regex

```
imdb %>%
  mutate(split_generos = str_split(generos, "\\|")) %>%
  select(titulo, generos, split_generos)
```

```
#> # A tibble: 3,713 x 3
#>   titulo                                   generos                        split_generos
#>   <chr>                                    <chr>                          <list>
#> 1 Avatar                                   Action|Adventure|Fantasy|Sci… <chr [4]>
#> 2 Pirates of the Caribbean: At World's E… Action|Adventure|Fantasy       <chr [3]>
#> 3 The Dark Knight Rises                    Action|Thriller                <chr [2]>
#> 4 John Carter                              Action|Adventure|Sci-Fi        <chr [3]>
#> # … with 3,709 more rows
```

Unnest

- Usando `tidyr::unnest()` é possível "abrir" a list-column de modo que cada linha fique com um de seus elementos (neste caso, o título do filme vai ser repetido uma vez para cada gênero ao qual o filme pertence)

```
library(tidyr)
imdb %>%
  mutate(split_generos = str_split(generos, "\\|")) %>%
  select(titulo, split_generos) %>%
  unnest(split_generos)
```

```
#> # A tibble: 10,612 x 2
#>   titulo split_generos
#>   <chr>  <chr>
#> 1 Avatar Action
#> 2 Avatar Adventure
#> 3 Avatar Fantasy
#> 4 Avatar Sci-Fi
#> # … with 10,608 more rows
```

Nest

- A operação inversa do `unnest()` é o `nest()`, que transforma um grupo de linhas em uma list-column

```
imdb %>%
  mutate(split_generos = str_split(generos, "\\|")) %>%
  select(titulo, split_generos) %>%
  unnest(split_generos) %>%
  group_by(titulo) %>%
  nest(generos = c(split_generos))
```

```
#> # A tibble: 3,711 x 2
#> # Groups:   titulo [3,711]
#>   titulo                                    generos
#>   <chr>                                     <list>
#> 1 Avatar                                    <tibble [4 × 1]>
#> 2 Pirates of the Caribbean: At World's End <tibble [3 × 1]>
#> 3 The Dark Knight Rises                     <tibble [2 × 1]>
#> 4 John Carter                               <tibble [3 × 1]>
#> # … with 3,707 more rows
```

Voltando ao {purrr}

- Trazendo o assunto de volta para o {purrr}, o pacote nos permite operar com facilidade em list-columns justamente pela sua capacidade de tratar listas

```
imdb %>%
  mutate(split_generos = str_split(generos, "\\|")) %>%
  select(titulo, split_generos) %>%
  unnest(split_generos) %>%
  group_by(titulo) %>%
  nest(generos = c(split_generos)) %>%
  ungroup() %>%
  mutate(n_generos = map_dbl(generos, nrow))
```

```
#> # A tibble: 3,711 x 3
#>   titulo                                    generos          n_generos
#>   <chr>                                     <list>               <dbl>
#> 1 Avatar                                    <tibble [4 × 1]>         4
#> 2 Pirates of the Caribbean: At World's End  <tibble [3 × 1]>         3
#> 3 The Dark Knight Rises                     <tibble [2 × 1]>         2
#> 4 John Carter                               <tibble [3 × 1]>         3
#> # … with 3,707 more rows
```

Para saber mais

O `{purrr}` simplifica loops, impede efeitos colaterais e ainda deixa seu código mais bonito! Para entender melhor como esse pacote incrível funciona, veja mais nos links abaixo:

- A Magia de Purrr
- Webinar de purrr avançado
- Exemplos com purrr
- Purrr cheat sheet
- Purrr tutorial
- R for Data Science: Iteration