R para Ciência de Dados 2

Trabalhando com strings

Iniciar gravação!

stringr

Motivação

Até agora já aprendemos muito sobre manipulação de dados, mas na verdade só aprendemos a manipular *tabelas*. Existem inúmeros tipos de dados dentro das tabelas e as funções que vimos até agora lidam praticamente só com números.

Bases com colunas textuais são *extremamente* comuns hoje em dia, então saber lidar com strings se torna uma parte essencial do processo da ciência de dados.

Muitas pessoas acham que trabalhar com textos é igual a trabalhar com variáveis categóricas; basta converter cada string para um número e seguir com a modelagem. Entretanto, isso não faz sentido para grande parte das variáveis que veremos aqui.

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

Mas o que há de tão diferente nas strings? Vamos entender isso na prática...

Criando strings

Strings não passam de sequências de caracteres. No R é muito fácil criar uma string, basta escrever qualquer coisa entre parênteses:

```
"Isto é uma string"
```

```
#> [1] "Isto é uma string"
```

Se precisarmos colocar aspas duplas dentro da nossa string, podemos usar aspas simples para criá-la.

```
'Isto é uma "string"'
```

```
#> [1] "Isto é uma \"string\""
```

O R sempre usa aspas duplas para exibir strings, até aí ok. Mas de onde veio essa \"?

Escapando caracteres

Como o R atribui uma função primária à aspa (criar strings), precisamos de um mecanismo para acessar sua função secundária (ser um caractere em uma string). A **barra invertida** indica para o R que estamos usando a função secundária do caractere seguinte.

```
c("'simples'", '\'simples\'', '"dupla"', "\"dupla\"", "\\barra\\")
```

```
#> [1] "'simples'" "'simples'" "\"dupla\"" "\"dupla\"" "\\barra\\"
```

Se quisermos ver uma string sem os escapes, podemos usar a função `cat()`. Ela retorna qualquer texto da forma como ele seria exibido fora do R.

```
cat(c("'simples'", '\'simples\'', '"dupla"', "\"dupla\"", "\\barra\\")
```

```
#> 'simples' 'simples' "dupla" "dupla" \barra\
```

Outros caracteres especiais

Vários caracteres têm funções secundárias. O `\n`, por exemplo, indica que esta não é a letra `n` e sim uma **n**ova linha; o `\t`, por sua vez, é um **t**ab.

```
cat("Cachorro:\n\t- Dexter")
```

```
#> Cachorro:
#>     - Dexter
```

O `\u` é extremamente poderoso. Ele indica que os próximos *quatro* caracteres serão um código Unicode, que é a lista internacional de todos os caracteres possíveis.

```
cat("Temperatura: 10\u00ba C")
```

```
#> Temperatura: 10º C
```

Agora que já sabemos criar strings, como podemos trabalhar com elas?

Ilustração por @allison_horst

O pacote stringr

O pacote que nos permite trabalhar com strings de forma intuitiva e eficiente é o stringr. Praticamente todas as suas funções começam com `str_`, o que facilita muito na hora de achar o que você precisa.

```
# Já carrega o stringr
library(tidyverse)

cachorros <- tibble(cachorro = c("Bacon", "Dexter", "Zip"))
cachorros
```

```
#> # A tibble: 3 × 1
#>   cachorro
#>   <chr>
#> 1 Bacon
#> 2 Dexter
#> 3 Zip
```

str_c()

A primeira (e talvez mais importante) função do stringr é a `str_c()`. Assim como `c()` serve para juntar elementos, a `str_c()` serve para juntar strings.

```
cachorros |>
  mutate(oi = str_c("Oi, ", cachorro, "!"))
```

```
#> # A tibble: 3 × 2
#>   cachorro oi
#>   <chr>    <chr>
#> 1 Bacon    Oi, Bacon!
#> 2 Dexter   Oi, Dexter!
#> 3 Zip      Oi, Zip!
```

A `str_c()` aceita qualquer número de argumentos e, se algum deles for um vetor de strings, ela faz a junção iterativamente.

str_flatten()

A `str_flatten()` também junta strings, mas ela sempre **achata** todas e retorna sempre uma única string. Ela funciona perfeitamente com a `summarise()`:

```
cachorros |>
  summarise(todos = str_flatten(cachorro, ", ", last = " e "))
```

```
#> # A tibble: 1 × 1
#>   todos
#>   <chr>
#> 1 Bacon, Dexter e Zip
```

O seu segundo argumento especifica um separador a ser colocado entre cada string do vetor de entrada. Já o argumento `last` especifica uma variação desse separador a ser utilizada somente na última string.

str_length() e str_sub()

Duas funções trabalham com caracteres individuais: `str_length()` e `str_sub()`. A primeira faz a contagem e a segunda extrai os caracteres entre duas posições.

```
cachorros |>
  mutate(
    letras = str_length(cachorro),
    apelido = str_sub(cachorro, 1, 3)
  )
```

```
#> # A tibble: 3 × 3
#>   cachorro letras apelido
#>   <chr>     <int> <chr>
#> 1 Bacon         5 Bac
#> 2 Dexter        6 Dex
#> 3 Zip           3 Zip
```

str_squish() e str_pad()

A `str_squish()` elimina espaços desnecessários em uma string. Ela remove todos os espaços no início e no fim, além de encurtar espaços excessivos no meio.

```
str_squish("  Nenhum   espaço   sobrando    ")
```

```
#> [1] "Nenhum espaço sobrando"
```

Já a `str_pad()` adiciona caracteres até que a string fique com o comprimento desejado. O argumento `side` permite até escolher de qual lado vai o enchimento.

```
str_pad("7", width = 3, side = "left", pad = "0")
```

```
#> [1] "007"
```

str_to_*()

```
str_to_upper("caixa alta e baixa. título e frase.")
```

```
#> [1] "CAIXA ALTA E BAIXA. TÍTULO E FRASE."
```

```
str_to_lower("caixa alta e baixa. título e frase.")
```

```
#> [1] "caixa alta e baixa. título e frase."
```

```
str_to_title("caixa alta e baixa. título e frase.")
```

```
#> [1] "Caixa Alta E Baixa. Título E Frase."
```

```
str_to_sentence("caixa alta e baixa. título e frase.")
```

```
#> [1] "Caixa alta e baixa. título e frase."
```

Expressões regulares

Motivação

Por enquanto vimos como manipular strings, mas não estamos extraindo nenhuma informação. Podemos juntar, encurtar, colar e resumir strings, mas não interagir com seus conteúdos.

Uma das partes mais importantes (e difíceis) de se aprender nesta área são as **expressões regulares**. O resto da aula vai ser praticamente todo dedicado a ensinar regex e mostrar como essa tecnologia funciona, e ainda assim não vamos ter nem arranhado a ponta desse iceberg.

A parte boa é que regex funciona praticamente da mesma forma em qualquer linguagem de programação, então esse conhecimento é altamente replicável mesmo que você precise trabalhar com alguma outra ferramenta.

Não exitem em tirar dúvidas! No início tudo vai parecer chato e óbvio, mas eu garanto que temos muita coisa para vocês colocarem em suas caixas de ferramentas. Vamos entender o que é e como funcionam as expressões regulares...

Regex

Expressões regulares (ou "regex") são uma linguagem para descrever padrões de strings. Elas são extremamente poderosas e muito usadas pelo stringr.

```
frutas <- c("laranja", "melancia", "maçã", "mamão")
```

Para aprender sobre o assunto, vamos usar a função `str_extract()`. Ela extrai a parte de uma string que bate com o regex do segundo argumento.

```
str_extract(string = frutas, pattern = "ma")
```

```
#> [1] NA   NA   "ma" "ma"
```

Note que ela retorna `NA` se não houve nenhum *match*.

Literais

Os **caracteres literais** são majoritariamente letras e numerais. Eles não têm significado especial em regex e, portanto, casam com si próprios.

```
str_extract(c("laranja", "MELANCIA", "maçã", "mamão"), "la")
```

```
#> [1] "la" NA   NA   NA
```

Os literais são sensíveis a maiúsculas e minúsculas, por isso não houve *match* com `"MELANCIA"`. Abaixo vemos a situação inversa.

```
str_extract(c("laranja", "MELANCIA", "maçã", "mamão"), "LA")
```

```
#> [1] NA   "LA" NA   NA
```

Nunca esqueça que o **espaço em branco** é um literal válido!

Meta-caracteres

A maior parte das pontuações têm significados especiais, sendo chamadas de **meta-caracteres**. O `.`, por exemplo, é o coringa e casa com qualquer caractere.

```
str_extract(c("laranja", "MELANCIA", "maçã", "mamão"), "a.")
```

```
#> [1] "ar" NA   "aç" "am"
```

Para fazer uma regex que dê *match* com um ponto-final literal, precisamos acessar a função secundária do `.` usando `\\.`:

```
str_extract("strings.", "s\\.") # "s." daria match com "st"
```

```
#> [1] "s."
```

Obs.: Nosso livro fala de um método mais simples, mas mais limitado que o `\\`.

Escapando meta-caracteres

Em regex, a função primária do `.` é ser o coringa e a secundária é ser o ponto-final literal. Já no R, a função primária do `.` é ser o ponto-final literal e não há nenhuma função secundária.

Se usarmos `\.`, o R vai tentar achar uma função secundária do `.` e falhar no processo (`unrecognized escape`). Logo, precisamos **escapar a barra invertida**! `\\.` vai passar `\.` ao regex, que consegue achar uma função secundária para o ponto-final.

Quando as barras invertidas começam a atrapalhar a leitura, a melhor saída é usar **raw strings**. Ao invés de apas, raw strings são criadas com sequências incomuns de caracteres como `r"()"`, `r"[]"` ou `r"{}"`.

```
cat(c("\\.", "\\\\", r"(\.)", r"(\\)"))
```

```
#> \. \\ \. \\
```

Meta-caracteres: âncoras

^ casa com o início de uma string e $ casa com o final. É como se toda string tivesse 2 caracteres invisíveis que só podem ser acessados com as âncoras.

```
str_extract(c("laranja", "melancia"), "^la")
```

```
#> [1] "la" NA
```

```
str_extract(c("laranja", "jaca"), "ja$")
```

```
#> [1] "ja" NA
```

```
str_extract(c("jaca", " jaca "), "^jaca$")
```

```
#> [1] "jaca" NA
```

Meta-caracteres: conjuntos

`[]` permite casar com qualquer caractere em um conjunto, enquanto `[^]` faz exatamente o contrário. Neste contexto, `-` pode definir um intervalo.

```
str_extract(c("romã", "laranja", "cereja"), "r[ao]")
```

```
#> [1] "ro" "ra" NA
```

```
str_extract(c("romã", "laranja", "cereja"), "r[^ao]")
```

```
#> [1] NA   NA   "re"
```

```
str_extract(c("coc0", "carambola", "melanc1a"), "c[0-9]")
```

```
#> [1] "c0" NA   "c1"
```

Meta-caracteres: grupos

`()` permite agrupar expressões, com `|` separando uma da outra. Na prática, funciona como `[]`, mas se aplica a padrões com mais de um caractere.

```
str_extract(c("laranja", "cereja"), "(an|re)ja")
```

```
#> [1] "anja" "reja"
```

`()` também pode ser usado com **referências retroativas**, onde `\\1` copia o conteúdo capturado pelo primeiro `()`, `\\2` pelo segundo e assim por diante.

```
str_extract(c("banana", "cacau", "coco"), "(..)\\1")
```

```
#> [1] "anan" "caca" "coco"
```

Meta-caracteres: quantificadores

? torna um padrão opcional (0 ou 1 vez), + permite que um padrão repita (1 ou mais vezes) e * permite que um padrão seja opcional ou repita (0 ou mais vezes).

```
str_extract(c("a", "ab", "abb"), "ab?")
```

```
#> [1] "a"  "ab" "ab"
```

```
str_extract(c("a", "ab", "abb"), "ab+")
```

```
#> [1] NA    "ab"  "abb"
```

```
str_extract(c("a", "ab", "abb"), "ab*")
```

```
#> [1] "a"   "ab"  "abb"
```

Meta-caracteres: quantificadores específicos

`{m,n}` permite casar entre `m` e `n` vezes. Podemos omitir um dos dois para casar exatamente `n` ou pelo menos `n` vezes.

```
str_extract(c("a", "aa", "aaa", "aaaa"), "a{2,3}") # Entre m e n
```

```
#> [1] NA    "aa"  "aaa" "aaa"
```

```
str_extract(c("a", "aa", "aaa", "aaaa"), "a{3}") # Exatamente n
```

```
#> [1] NA    NA    "aaa" "aaa"
```

```
str_extract(c("a", "aa", "aaa", "aaaa"), "a{3,}") # n ou mais
```

```
#> [1] NA     NA     "aaa"  "aaaa"
```

Meta-caracteres: quantificadores com [] e ()

Quantificadores são muito utilizados com `[]` e `()`. Neste caso, eles vão se aplicar ao conteúdo completo do conjunto ou grupo.

```
str_extract("Telefone: 91234-5678", "[0-9]+")
```

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

```
str_extract("Telefone: 91234-5678", "[0-9]{5}-[0-9]{4}")
```

```
#> [1] "91234-5678"
```

```
str_extract(c("oi olá oi", "olá oi olá"), "(oi |olá )+")
```

```
#> [1] "oi olá " "olá oi "
```

str_detect()

Podemos usar regex na função `str_detect()` para verificar se um padrão existe. Isso é muito útil na hora de filtrar tabelas com colunas textuais.

```
# Usando a tabela da Billboard da aula passada
musicas |>
  filter(str_detect(track, "[Bb]aby"))
```

```
#> # A tibble: 6 × 7
#>   track                    wk1   wk2   wk3   wk4   wk5   wk6
#>   <chr>                  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 Baby Don't Cry (Keep...    87    82    72    77    87    94
#> 2 Come On Over Baby (A...    57    47    45    29    23    18
#> 3 My Baby You                82    76    76    70    82    81
#> 4 Crybaby                    28    34    48    62    77    90
#> 5 Baby U Are                 96    89    92    96    96    NA
#> # ℹ 1 more row
```

str_replace() e str_remove()

A `str_replace()` aceita um regex e substitui o *match* usando o argumento `replacement` (que pode conter referências retroativas). Se quisermos só remover o *match*, podemos usar `str_remove()`.

```
str_replace(
  string = c("/user/pasta/arq1.py", "/user/pasta/arq2.py"),
  pattern = "pasta/(arq[0-9])\\.py",
  replacement = "nova_pasta/\\1.R"
)
```

```
#> [1] "/user/nova_pasta/arq1.R" "/user/nova_pasta/arq2.R"
```

```
str_remove("/user/pasta/arq1.py", "\\.py")
```

```
#> [1] "/user/pasta/arq1"
```

str_*_all()

Várias funções que vimos até agora só atuam no *primeiro match* da string. Se quisermos atuar em todos os *matches*, precisamos das suas versões com `_all()`.

```
str_extract_all("Telefone: 91234-5678", "[0-9]+")
```

```
#> [[1]]
#> [1] "91234" "5678"
```

```
str_replace_all("Bom dia. Boa tarde. Boa noite.", "\\.", "!")
```

```
#> [1] "Bom dia! Boa tarde! Boa noite!"
```

```
str_remove_all('Bom. dia, turma.', '\\.')
```

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

Rodada bônus!

O tidyr também tem uma função para lidar com regex: `separate_wider_regex()`.

```
tibble(dados = "Nome: Caio / Sexo: M / Idade: 27") |>
  separate_wider_regex(dados, patterns = c(
    "Nome: ",
    nome = "[A-Za-z]+", # Dois intervalos juntos
    " / Sexo: ",
    sexo = "[A-Z]+",
    " / Idade: ",
    idade = "[0-9]+"
  ))
```

```
#> # A tibble: 1 × 3
#>   nome  sexo  idade
#>   <chr> <chr> <chr>
#> 1 Caio  M     27
```

Miscelânea

Problemas com acentos

Para casar com todas as letras com acento, precisamos usar `[:alpha:]`.

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

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

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

```
#> [1] "Número"
```

Se quisermos remover os acentos, precisamos da `stringi::stri_trans_general()`

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

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

Problemas com formatos

Este assunto foge um pouco do escopo do curso, mas é importante que vocês estejam cientes de que ele existe. Para que uma string seja interpretada corretamente, ela depende de um **locale** (e do seu **encoding**).

Veja o código abaixo. A segunda linha teve um erro de leitura porque ela foi armazenada usando o encoding `Latin1`, o padrão do Excel brasileiro.

```
arquivo |>
  read_csv() |>
  arrange(texto)
```

```
#> # A tibble: 2 × 1
#>   texto
#>   <chr>
#> 1 "Z"
#> 2 "\xc1"
```

Encoding

Para corrigir o problema anterior, basta especificar um encoding com a função `locale()`. A `guess_encoding()` pode te ajudar neste processo.

```
arquivo |>
  read_csv(locale = locale(encoding = "Latin1")) |>
  arrange(texto)
```

```
#> # A tibble: 2 × 1
#>   texto
#>   <chr>
#> 1 Z
#> 2 Á
```

Mas agora podemos ver que a `arrange()` não está ordenando as linhas corretamente! Isso acontece quando o seu `.locale` não bate com o idioma.

Locale

Para corrigir este segundo problema, precisamos especificar o idioma da tabela na `arrange()`. No geral, vamos sempre usar `"pt_BR"`.

```
arquivo |>
  read_csv(locale = locale(encoding = "Latin1")) |>
  arrange(texto, .locale = "pt_BR")
```

```
#> # A tibble: 2 × 1
#>   texto
#>   <chr>
#> 1 Á
#> 2 Z
```

Para saber mais sobre esse tópico, recomendo a seção no nosso livro sobre encoding e este post sobre locale.

Fim