

R para Ciência de Dados 2

dplyr++

Rafael Vetromille

10/09/2020

O pacote dplyr

```
# Carregando o pacote
library(dplyr)

##
## Attaching package: 'dplyr'
##
## The following objects are masked from 'package:stats':
##
##   filter, lag
##
## The following objects are masked from 'package:base':
##
##   intersect, setdiff, setequal, union
```

A função bind_rows()

Vamos usar a função `bind_rows()` para juntarmos duas bases com as mesmas colunas. Neste caso, a função `bind_rows()` é equivalente à função `rbind()`.

```
# Juntando duas bases.
imdb_2015 <- readr::read_rds("./data/imdb_por_ano/imdb_2015.rds")
imdb_2016 <- readr::read_rds("./data/imdb_por_ano/imdb_2016.rds")

# A função bind_rows().
bind_rows(imdb_2015, imdb_2016) %>% head()

## # A tibble: 6 x 15
##   titulo    ano diretor duracao cor   generos pais classificacao orcamento
##   <chr>   <int> <chr>    <int> <chr> <chr>   <chr>   <chr>          <int>
## 1 Aveng~   2015 Joss W~    141 Color Action~ USA   A partir de ~ 250000000
## 2 Juras~   2015 Colin ~    124 Color Action~ USA   A partir de ~ 150000000
## 3 Furio~   2015 James ~    140 Color Action~ USA   A partir de ~ 190000000
## 4 The G~   2015 Peter ~     93 Color Advent~ USA   Livre          NA
## 5 Jupit~   2015 Lana W~    127 Color Action~ USA   A partir de ~ 176000000
## 6 Insid~   2015 Pete D~     95 Color Advent~ USA   Livre          175000000
## # ... with 6 more variables: receita <int>, nota_imdb <dbl>,
## #   likes_facebook <int>, ator_1 <chr>, ator_2 <chr>, ator_3 <chr>
```

Observação: tanto para a `bind_rows()` quanto para a `rbind()`, a ordem das colunas em ambas as bases pode ser diferente. As colunas são empilhadas pelo nome.

Podemos também usar a função `bind_rows()` para juntar várias tabelas. Aqui, todas as tabelas continuam tendo as mesmas colunas. A função `list.files` produz um vetor de caracteres dos nomes dos arquivos ou diretórios no diretório nomeado.

```
## Vetor de arquivos na pasta especificada.
arquivos <- list.files("./data/imdb_por_ano/", full.names = TRUE)

## Importar dados de uma única vez.
df <- arquivos %>%
  purrr::map(readr::read_rds) %>%
  bind_rows()
```

ou, simplesmente,

```
## Importando com a função purrr::map_dfr().
df <- arquivos %>%
  purrr::map_dfr(readr::read_rds)
```

onde o `r` da função `purrr::map_dfr()` indica *rows*. Além disso, a função `bind_rows()` também funciona para empilhar bases com colunas diferentes.

```
# Tabela 1
tab1 <- tibble::tibble(
  var1 = c(1, 2, 3),
  var2 = c("a", "b", "c"),
  var3 = c(10, 20, 30)
)

# Tabela 2
tab2 <- tibble::tibble(
  var2 = c("d", "e", "f"),
  var1 = c(4, 5, 6)
)

# Juntar as linhas com a tab1 por cima.
dplyr::bind_rows(tab1, tab2)
```

```
## # A tibble: 6 x 3
##   var1 var2  var3
##   <dbl> <chr> <dbl>
## 1     1  a      10
## 2     2  b      20
## 3     3  c      30
## 4     4  d       NA
## 5     5  e       NA
## 6     6  f       NA
```

Além da função `dplyr::bind_rows()` o `{dplyr}` também possui a função `dplyr::bind_cols()` que junta duas bases colando suas colunas lado a lado.

A função `case_when()`

A função `dplyr::case_when()` é uma generalização da função `base::ifelse()`. Ela permite trabalharmos com quantas condições forem necessárias.

```
## Vetor x transformado em tibble
x <- sample(-10:10, size = 10) %>%
  as_tibble()

## Aplicando a função case_when()
x %>%
  dplyr::mutate(sinal = case_when(x < 0 ~ "negativo",
                                x == 0 ~ "zero",
                                x > 0 ~ "positivo"))
```

```
## # A tibble: 10 x 2
##   value sinal
##   <int> <chr>
## 1     7 positivo
## 2    -5 negativo
## 3    -8 negativo
## 4     1 positivo
## 5    -3 negativo
## 6    -9 negativo
## 7     0 zero
## 8    -1 negativo
## 9     2 positivo
## 10   -6 negativo
```

Se fossemos utilizar a função `base::ifelse()`, precisaríamos usar a função duas vezes, assim como é feito no Excel um `se` dentro de outro:

```
## A função ifelse() em contraste com a case_when()
x %>%
  dplyr::mutate(sinal = ifelse(x < 0, "negativo",
                              ifelse(x == 0, "zero", "positivo")))
```

```
## # A tibble: 10 x 2
##   value sinal[, "value"]
##   <int> <chr>
## 1     7 positivo
## 2    -5 negativo
## 3    -8 negativo
## 4     1 positivo
## 5    -3 negativo
## 6    -9 negativo
## 7     0 zero
## 8    -1 negativo
## 9     2 positivo
## 10   -6 negativo
```

A ordem das condições é importante na função `dplyr::case_when()`, pois os testes são realizados na ordem em que aparecem e o próximo teste não substitui o anterior. Por exemplo,

```
## A ordem importa na função case_when().
mtcars %>%
  mutate(
```

```

mpg_cat = case_when(
  mpg <= 15 ~ "economico",
  mpg < 22 ~ "regular",
  mpg >= 22 ~ "bebe bem"
)
) %>%
head(15)

```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	mpg_cat
## 1	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4	regular
## 2	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4	regular
## 3	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1	bebe bem
## 4	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1	regular
## 5	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2	regular
## 6	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1	regular
## 7	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4	economico
## 8	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2	bebe bem
## 9	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2	bebe bem
## 10	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4	regular
## 11	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4	regular
## 12	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3	regular
## 13	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3	regular
## 14	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3	regular
## 15	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4	economico

Nesse caso, os carros com `mpg` menor ou igual a 15 são considerados *econômicos*, os caros com `mpg` maior (estrito) que 15 e menor (estrito) que 22 são *regulares* e, por fim, os carros com `mpg` maior ou igual à 22 são considerados como *bebe bem*. Como a última condição é complementar, isto é, as observações que não entram nas condições anteriores com certeza entrarão na última condição, podemos substituí-la por um simples `TRUE`.

```

## Um facilitador nem tão facilitante (TRUE)
mtcars %>%
  mutate(
    mpg_cat = case_when(
      mpg < 15 ~ "economico",
      mpg < 22 ~ "regular",
      TRUE ~ "bebe bem"
    )
  ) %>%
  head(15)

```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	mpg_cat
## 1	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4	regular
## 2	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4	regular
## 3	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1	bebe bem
## 4	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1	regular
## 5	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2	regular
## 6	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1	regular
## 7	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4	economico
## 8	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2	bebe bem
## 9	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2	bebe bem
## 10	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4	regular
## 11	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4	regular
## 12	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3	regular
## 13	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3	regular

##	14	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3	regular
##	15	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4	economico

Como pudemos ver, a função `dplyr::case_when()` é extremamente útil associada à função `dplyr::mutate()` pois assim, conseguimos criar variáveis novas que são obtidas através de testes de outras variáveis. Um caso bastante usado é quando queremos fazer gráficos em que os valores negativos ficam em vermelho enquanto que os valores positivos ficam em verde.

```
mtcars %>%
```

```
  mutate(
    mpg_cat = case_when(
      mpg < 15 ~ "economico",
      mpg < 22 ~ "regular",
      TRUE ~ "bebe bem"
    )
  ) %>%
  head(14)
```

```
##      mpg cyl  disp  hp drat   wt  qsec vs am gear carb  mpg_cat
## 1  21.0   6  160.0 110 3.90 2.620 16.46  0  1   4    4   regular
## 2  21.0   6  160.0 110 3.90 2.875 17.02  0  1   4    4   regular
## 3  22.8   4  108.0  93 3.85 2.320 18.61  1  1   4    1  bebe bem
## 4  21.4   6  258.0 110 3.08 3.215 19.44  1  0   3    1   regular
## 5  18.7   8  360.0 175 3.15 3.440 17.02  0  0   3    2   regular
## 6  18.1   6  225.0 105 2.76 3.460 20.22  1  0   3    1   regular
## 7  14.3   8  360.0 245 3.21 3.570 15.84  0  0   3    4 economico
## 8  24.4   4  146.7  62 3.69 3.190 20.00  1  0   4    2  bebe bem
## 9  22.8   4  140.8  95 3.92 3.150 22.90  1  0   4    2  bebe bem
## 10 19.2   6  167.6 123 3.92 3.440 18.30  1  0   4    4   regular
## 11 17.8   6  167.6 123 3.92 3.440 18.90  1  0   4    4   regular
## 12 16.4   8  275.8 180 3.07 4.070 17.40  0  0   3    3   regular
## 13 17.3   8  275.8 180 3.07 3.730 17.60  0  0   3    3   regular
## 14 15.2   8  275.8 180 3.07 3.780 18.00  0  0   3    3   regular
```

As funções `first()` and `last()`

Como o próprio nome já indica, essas funções retornam o primeiro e último valor de vetor/coluna. Por exemplo,

```
x <- c(1, 12, 30, 41, 15)
```

```
first(x)
```

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

```
last(x)
```

```
## [1] 15
```

São funções úteis quando temos algum tipo de ordem, por exemplo:

```
tab <- tibble::tibble(
  tempo = c(1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4),
  var = c(1, 4, 10, 33, 1, 3, 0, 21, 12, 7, 9, 17),
  grupo = c(rep("a", 4), rep("b", 4), rep("c", 4))
)
```

```
tab %>%
```

```
  group_by(grupo) %>%
```

```

arrange(tempo, .by_group = TRUE) %>%
mutate(inicio = first(var),
       fim = last(var))

```

```

## # A tibble: 12 x 5
## # Groups:   grupo [3]
##   tempo  var grupo inicio  fim
##   <dbl> <dbl> <chr>  <dbl> <dbl>
## 1     1     1     a      1     33
## 2     2     4     a      1     33
## 3     3    10     a      1     33
## 4     4    33     a      1     33
## 5     1     1     b      1     21
## 6     2     3     b      1     21
## 7     3     0     b      1     21
## 8     4    21     b      1     21
## 9     1    12     c     12     17
## 10    2     7     c     12     17
## 11    3     9     c     12     17
## 12    4    17     c     12     17

```

A função na_if()

Existem tabelas ou planilhas em que o valor NA não está escrito por padrão como NA. Dessa forma, a função `na_if()` transforma um padrão em NA. Por exemplo,

```

tab <- tibble::tibble(
  var = c(1, 10, 2, -99, 10, -99)
)

tab %>% mutate(var = na_if(var, -99))

```

```

## # A tibble: 6 x 1
##   var
##   <dbl>
## 1     1
## 2    10
## 3     2
## 4    NA
## 5    10
## 6    NA

```

A função `coalesce()`

A função `coalesce()` substitui os NAs de uma coluna pelos valores equivalentes de uma segunda coluna. No exemplo abaixo, substituímos os NAs da coluna `var1` pelos valores equivalentes da coluna `var2` (criamos uma nova coluna `var3` com o resultado para visualizarmos melhor). Repare que, no caso em que as duas colunas apresentavam NA, a coluna `var3` permaneceu com o NA.

```
tab <- tibble::tibble(  
  var1 = c(1, 2, NA, 10, NA, NA),  
  var2 = c(NA, 2, 2, 3, 0, NA)  
)  
  
tab %>%  
  mutate(var3 = coalesce(var1, var2))
```

```
## # A tibble: 6 x 3  
##   var1  var2  var3  
##   <dbl> <dbl> <dbl>  
## 1     1    NA     1  
## 2     2     2     2  
## 3    NA     2     2  
## 4    10     3    10  
## 5    NA     0     0  
## 6    NA    NA    NA
```

Você também pode usar para substituir os valores NA de uma variável por um valor específico, por exemplo:

```
tab %>%  
  mutate(var3 = coalesce(var1, 33))
```

```
## # A tibble: 6 x 3  
##   var1  var2  var3  
##   <dbl> <dbl> <dbl>  
## 1     1    NA     1  
## 2     2     2     2  
## 3    NA     2    33  
## 4    10     3    10  
## 5    NA     0    33  
## 6    NA    NA    33
```

Além disso, existe a função `dplyr::replace_na()` que faz a mesma coisa, por exemplo:

```
tab %>%  
  tidyr::replace_na(replace = list(var1 = 33, var2 = 66))
```

```
## # A tibble: 6 x 2  
##   var1  var2  
##   <dbl> <dbl>  
## 1     1    66  
## 2     2     2  
## 3    33     2  
## 4    10     3  
## 5    33     0  
## 6    33    66
```


As funções `lag()` e `lead()`

Essas funções devolvem o valor defasado e valor futuro.

```
tab <- tibble::tibble(  
  tempo = c(1, 2, 3, 4, 5),  
  var = c(1, 4, 10, 33, 20)  
)  
  
tab %>%  
  dplyr::mutate(  
    var_lag1 = lag(var, n = 1L),  
    var_lead1 = lead(var, n = 1L)  
  )
```

```
## # A tibble: 5 x 4  
##   tempo  var var_lag1 var_lead1  
##   <dbl> <dbl>   <dbl>   <dbl>  
## 1     1     1      NA        4  
## 2     2     4       1       10  
## 3     3    10       4       33  
## 4     4    33      10       20  
## 5     5    20      33      NA
```

A função `pull()`

Devolve uma coluna da base como vetor.

```
mtcars %>% pull(mpg)
```

```
## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4  
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7  
## [31] 15.0 21.4
```

A função `slice_sample()`

Essa função pode ser utilizada para pegarmos uma amostra de linhas da nossa base de forma aleatória. No exemplo abaixo, pegamos uma amostra aleatória de tamanho 10 da base `mtcars`.

```
slice_sample(mtcars, n = 10L)
```

```
##           mpg cyl  disp  hp drat   wt  qsec vs am gear carb  
## Duster 360   14.3   8 360.0 245 3.21 3.570 15.84 0 0   3    4  
## Merc 450SL   17.3   8 275.8 180 3.07 3.730 17.60 0 0   3    3  
## Merc 450SE   16.4   8 275.8 180 3.07 4.070 17.40 0 0   3    3  
## Volvo 142E   21.4   4 121.0 109 4.11 2.780 18.60 1 1   4    2  
## AMC Javelin  15.2   8 304.0 150 3.15 3.435 17.30 0 0   3    2  
## Ferrari Dino  19.7   6 145.0 175 3.62 2.770 15.50 0 1   5    6  
## Honda Civic  30.4   4  75.7  52 4.93 1.615 18.52 1 1   4    2  
## Cadillac Fleetwood 10.4  8 472.0 205 2.93 5.250 17.98 0 0   3    4  
## Fiat X1-9     27.3   4  79.0  66 4.08 1.935 18.90 1 1   4    1  
## Porsche 914-2 26.0   4 120.3  91 4.43 2.140 16.70 0 1   5    2
```

Agora, pegamos 50% da base `mtcars` de forma aleatória.

```
slice_sample(mtcars, prop = 0.5)
```

##	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
## Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
## Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
## Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
## Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2
## Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
## Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
## Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
## Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
## Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
## Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
## Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
## Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
## Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
## Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
## Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
## Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1

Além dessa função, existem as funções:

- `slice_head()` and `slice_tail()` select the first or last rows.
- `slice_min()` and `slice_max()` select rows with highest or lowest values of a variable.
- `slice()` lets you index rows by their (integer) locations.

O novo dplyr

A versão 1.0.0 do pacote {dplyr} foi oficialmente lançada em junho de 2020 e contou com diversas novidades. Vamos falar das principais mudanças:

- A nova função `across()`, que facilita aplicar uma mesma operação em várias colunas.
- A repaginada da função `rowwise()` que objetiva fazer operações por linha.
- As novas funcionalidades das funções `select()` e `rename()`.
- A nova função `relocate()` que facilita a mudança de posição de colunas.

Motivação

Base de dados de venda de casas na cidade de Ames, nos Estados Unidos.

- 2930 linhas e 77 colunas.
- Cada linha corresponde a uma casa vendida e cada coluna a uma característica da casa ou da venda.

```
# Base de dados

ames <- readr::read_rds("./data/ames.rds")

# Pegando apenas 5 colunas por uma questão de espaço.

ames %>%
  select(1:5) %>%
  head()

## # A tibble: 6 x 5
##   lote_fachada lote_area lote_formato lote_config terreno_contorno
##   <int>      <int> <chr>          <chr>      <chr>
## 1      141     31770 IR1          Corner    Lvl
## 2       80     11622 Reg          Inside    Lvl
## 3       81     14267 IR1          Corner    Lvl
## 4       93     11160 Reg          Corner    Lvl
## 5       74     13830 IR1          Inside    Lvl
## 6       78      9978 IR1          Inside    Lvl
```

A função `across()`

A função `across()` substitui a família de verbos `verbo_all()`, `verbo_if` e `verbo_at()`. A ideia é facilitar a aplicação de uma operação a diversas colunas da base. Para sumarizar a base para mais de uma variável, antigamente fazíamos

Como era antigamente ...

```
ames %>%
  group_by(geral_qualidade) %>%
  summarise(
    lote_area_medio = mean(lote_area, na.rm = TRUE),
    venda_valor_medio = mean(venda_valor, na.rm = TRUE)
  )

## `summarise()` ungrouping output (override with `.groups` argument)

## # A tibble: 10 x 3
##   geral_qualidade lote_area_medio venda_valor_medio
##         <int>         <dbl>         <dbl>
## 1             1         15214.          48725
## 2             2          9326.          52325.
## 3             3          9439.          83186.
## 4             4          8464.         106485.
## 5             5          9995.         134753.
## 6             6          9788.         162130.
## 7             7         10309.         205026.
## 8             8         10618.         270914.
## 9             9         12777.         368337.
## 10            10         18071.         450217.
```

Ou, ainda,

```
ames %>%
  group_by(geral_qualidade) %>%
  summarise_at(
    .vars = vars(lote_area, venda_valor),
    .funs = ~ mean(.x, na.rm = TRUE)
  )

## # A tibble: 10 x 3
##   geral_qualidade lote_area venda_valor
##         <int>     <dbl>     <dbl>
## 1             1    15214.    48725
## 2             2     9326.    52325.
## 3             3     9439.    83186.
## 4             4     8464.   106485.
## 5             5     9995.   134753.
## 6             6     9788.   162130.
## 7             7    10309.   205026.
## 8             8    10618.   270914.
## 9             9    12777.   368337.
## 10            10    18071.   450217.
```

Agora, com a nova função `across()`, podemos fazer a mesma sumarização da seguinte forma:

```
ames %>%
  group_by(geral_qualidade) %>%
  summarise(across(
    .cols = c(lote_area, venda_valor),
    .fns = mean, na.rm = TRUE
  ))

## `summarise()` ungrouping output (override with `.groups` argument)

## # A tibble: 10 x 3
##   geral_qualidade lote_area venda_valor
##   <int>         <dbl>         <dbl>
## 1             1    15214.         48725
## 2             2     9326.         52325.
## 3             3     9439.         83186.
## 4             4     8464.        106485.
## 5             5     9995.        134753.
## 6             6     9788.        162130.
## 7             7    10309.        205026.
## 8             8    10618.        270914.
## 9             9    12777.        368337.
## 10          10    18071.        450217.
```

A sintaxe é parecida com a função `summarise_at()`, mas agora não precisamos mais usar a função `vars()` e nem usar `list(nome_da_funcao)` ou `~nome_da_funcao(.x)` para definir a função aplicada nas colunas.

Usando `across()`, podemos facilmente aplicar uma função em todas as colunas da nossa base. Abaixo, calculamos o número de valores distintos para todas as variáveis da base `ames`.

```
# A função across() e summarise

ames %>%
  summarise(across(
    .cols = everything(), # default, não era necessário
    .fns = n_distinct, na.rm = TRUE
  )) %>%
  select(1:5)

## # A tibble: 1 x 5
##   lote_fachada lote_area lote_formato lote_config terreno_contorno
##   <int>         <int>         <int>         <int>         <int>
## 1      128      1960             4             5             4

# A purrr-style formula for across() - more intuitive

ames %>%
  summarise(across(
    .cols = everything(), # default, não era necessário
    .fns = ~ n_distinct(.x, na.rm = TRUE)
  )) %>%
  select(1:5)

## # A tibble: 1 x 5
##   lote_fachada lote_area lote_formato lote_config terreno_contorno
##   <int>         <int>         <int>         <int>         <int>
## 1      128      1960             4             5             4
```

Se quisermos selecionar as colunas a serem modificadas a partir de um teste lógico, utilizamos o ajudante `where()`. No exemplo abaixo, calculamos o número de valores distintos das colunas do tibble que são texto (character).

As função across(), com o auxiliar where() e summarise()

```
ames %>%
  summarise(across(
    .cols = where(is.character),
    .fns = n_distinct, na.rm = TRUE
  )) %>%
  select(1:5)
```

```
## # A tibble: 1 x 5
##   lote_formato lote_config terreno_contorno terreno_declive rua_tipo
##         <int>         <int>         <int>         <int>     <int>
## 1             4             5             4             3         2
```

A purrr-style formula for across() - more intuitive

```
ames %>%
  summarise(across(
    .cols = where(is.character),
    .fns = ~ n_distinct(.x, na.rm = T)
  )) %>%
  select(1:5)
```

```
## # A tibble: 1 x 5
##   lote_formato lote_config terreno_contorno terreno_declive rua_tipo
##         <int>         <int>         <int>         <int>     <int>
## 1             4             5             4             3         2
```

Todas as colunas da base resultante eram colunas com classe `character` na base `ames`. Antes, utilizávamos a função `summarise_if()`, no entanto, com o ajudante `where()` não há mais necessidade.

A função (antiga) summarise_if()

```
ames %>%
  summarise_if(.predicate = is.character,
    .funs = n_distinct, na.rm = TRUE) %>%
  select(1:5)
```

```
## # A tibble: 1 x 5
##   lote_formato lote_config terreno_contorno terreno_declive rua_tipo
##         <int>         <int>         <int>         <int>     <int>
## 1             4             5             4             3         2
```

A purrr-style formula for across() - more intuitive

```
ames %>%
  summarise_if(.predicate = is.character,
    .funs = ~ n_distinct(.x, na.rm = TRUE)) %>%
  select(1:5)
```

```
## # A tibble: 1 x 5
##   lote_formato lote_config terreno_contorno terreno_declive rua_tipo
##         <int>         <int>         <int>         <int>     <int>
## 1             4             5             4             3         2
```

Você também pode combinar as ações do `summarise_if()` e `summarise_at()` em um único `across()`. Calculamos as áreas médias, garantindo que pegamos apenas variáveis numéricas.

```
ames %>%
  summarise(across(
    .cols = where(is.numeric) & contains("area"),
    .fns = mean, na.rm = TRUE
  )) %>%
  select(1:5)
```

```
## # A tibble: 1 x 5
##   lote_area alvenaria_area porao_area_com_ac~ porao_area_com_a~ porao_area_sem_~
##   <dbl>         <dbl>         <dbl>         <dbl>         <dbl>
## 1   10148.          102.          443.          49.7          559.
```

Além disso, com a função `across()`, podemos fazer sumarizações bastante complexas. Por exemplo,

summarise function with complex summarizations.

```
ames %>%
  group_by(fundacao_tipo) %>%
  summarise(
    across(contains("area"), mean, na.rm = TRUE),
    across(where(is.character), ~sum(is.na(.x))),
    n_obs = n(),
  ) %>%
  select(1:4, n_obs)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
## # A tibble: 6 x 5
##   fundacao_tipo lote_area alvenaria_area porao_area_com_acabamento_1 n_obs
##   <chr>         <dbl>         <dbl>         <dbl> <int>
## 1 BrkTil          8712.          10.2          151.   311
## 2 CBlock         10616.          85.0          468.  1244
## 3 PConc          10054.          144.          506.  1310
## 4 Slab           10250.          35.2           0     49
## 5 Stone           8659.           0           43.9   11
## 6 Wood           9838.          16           812.    5
```

A purrr-style formula for across() - more intuitive

```
ames %>%
  group_by(fundacao_tipo) %>%
  summarise(
    across(
      .cols = contains("area"),
      .fns = ~ mean(.x, na.rm = TRUE)
    ),
    across(.cols = where(is.character),
      .fns = ~ sum(is.na(.x))),
    n_obs = n(),
  ) %>%
  select(1:4, n_obs)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
## # A tibble: 6 x 5
```

```
##   fundacao_tipo lote_area alvenaria_area porao_area_com_acabamento_1 n_obs
##   <chr>          <dbl>          <dbl>          <dbl> <int>
## 1 BrkTil         8712.           10.2           151.   311
## 2 CBlock         10616.          85.0           468.  1244
## 3 PConc          10054.          144.           506.  1310
## 4 Slab           10250.           35.2            0    49
## 5 Stone           8659.            0            43.9   11
## 6 Wood           9838.            16            812.    5
```

As função across(), com o auxiliar where() e summarise()

```
ames %>%
  summarise(across(
    .cols = where(is.numeric),
    .fns = mean, na.rm = T
  )) %>%
  select(1:5)
```

```
## # A tibble: 1 x 5
##   lote_fachada lote_area construcao_ano remodelacao_ano geral_qualidade
##   <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
## 1    69.2    10148.      1971.      1984.      6.09
```

A purrr-style formula for across() - more intuitive

```
ames %>%
  summarise(across(
    .cols = where(is.numeric),
    .fns = ~ mean(.x, na.rm = T)
  )) %>%
  select(1:5)
```

```
## # A tibble: 1 x 5
##   lote_fachada lote_area construcao_ano remodelacao_ano geral_qualidade
##   <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
## 1    69.2    10148.      1971.      1984.      6.09
```

```
ames %>%
  select(where(is.character)) %>%
  dim()
```

```
## [1] 2930  41
```