Café com estatística e R

Treinamento 2 - Importação e manipulação de dados no R e estatística descritiva: parte 1

Marcelo Teixeira Paiva

2025-10-14

Abstract

Relatório do segundo treinamento onde foi apresentado como importar dados e manipulá-los no R, bem como as principais estatísticas descritivas univariadas e multivariadas.

Índice

Lista de Figuras

Lista de Tabelas


```
# Pacotes
library(tidyverse)
library(gridExtra)
library(plotly)
library(gt)
library(tidyverse)
library(kableExtra)

# Tema personalizado para gráficos
tema_didatico <- theme_minimal() +
  theme(
    plot.title = element_text(face = "bold", size = 14),
    plot.subtitle = element_text(face = "italic", size = 11),
    axis.title = element_text(size = 12),
    legend.position = "top",
    panel.grid.minor = element_blank()
  )

cores <- c("#FF6B6B", "#4ECDC4", "#45B7D1", "#96CEB4", "#FFEAA7")
```

Chapter 1

Importação de dados no R

1.1 Pacotes necessários

Pacotes (**package**) são coleções de funções, dados e documentação que estendem as capacidades do R base (aquele que você recebe na instalação padrão). São como "caixas de ferramentas" especializadas que você adiciona ao R para realizar tarefas específicas, então você tem pacotes para elaboração de gráficos, para certos tipos de análises, para manipulação de dados, para leitura (importação) de dados. Em https://cran.r-project.org/web/views/ há uma "breve" lista de pacotes conforme a sua finalidade.

```
# funções no R base
length(ls("package:base"))
```

```
[1] 1270
```

```
# funções especializadas no pacote dplyr
length(ls("package:dplyr"))
```

```
[1] 297
```

```
# um pacote possui um conjunto de arquivos associados
system.file(package = "ggplot2") %>% list.files()
```

```
 [1] "CITATION"    "data"        "DESCRIPTION" "doc"         "help"
 [6] "html"        "INDEX"       "LICENSE"     "Meta"        "NAMESPACE"
[11] "NEWS.md"     "R"
```

Por padrão, ao iniciar uma sessão no R, serão carregados os pacotes e funções associados ao R base. Os demais devem ser instalados primeiramente, e depois carregados na seção para serem usados.

```
# pacotes carregados no seu ambiente
search()
```

```
 [1] ".GlobalEnv"         "package:kableExtra" "package:gt"
 [4] "package:plotly"     "package:gridExtra"  "package:lubridate"
 [7] "package:forcats"    "package:stringr"    "package:dplyr"
[10] "package:purrr"      "package:readr"      "package:tidyr"
[13] "package:tibble"     "package:ggplot2"    "package:tidyverse"
[16] "package:stats"      "package:graphics"   "package:grDevices"
[19] "package:utils"      "package:datasets"   "package:methods"
```

```
[22] "Autoloads"          "package:base"
```

```
# pacotes instalados
instalados <- installed.packages()[, "Package"]
instalados[1:4]
```

```
      abind      ARTool     askpass   backports
    "abind"    "ARTool"   "askpass" "backports"
```

```
length(instalados)
```

```
[1] 336
```

```
# verificando se um pacote já está instalado
sum(installed.packages()[, "Package"] == 'dplyr')
```

```
[1] 1
```

```
any(installed.packages()[, "Package"] == 'dplyr')
```

```
[1] TRUE
```

```
"ggplot2" %in% rownames(installed.packages())
```

```
[1] TRUE
```

A instalação de pacotes no R é feita usando a função `install.packages` ou `devtools::install_github` para pacotes que estão no github e não em um repositório de pacotes.

```
# pelo repositório oficial (na web)
install.packages("ggplot2")
install.packages(c("dplyr", "tidyr", "readr")) # instalando vários pacotes de uma vez

# Instalar o pacote e todas dependências relacionadas a ele
install.packages("ggplot2", dependencies = TRUE)

# instalar de um arquivo local
install.packages("caminho/para/pacote.tar.gz", repos = NULL, type = "source")

# Instalar pacote mantido no GitHub
install.packages("devtools")
devtools::install_github("tidyverse/ggplot2")

# Usar outros repositórios para instalação
install.packages("ggplot2", repos = "https://cloud.r-project.org/")
```

Para carregar um pacote em uma sessão usamos `library()` ou `require()`. A diferença entre os dois é que, na ausência do pacote que você pretende carregar, `library` gera um erro, enquanto o `require` retorna um valor `FALSE` invisível, o qual pode ser usado, por exemplo, para criar uma lógica em seu script para instalar o pacote caso o mesmo não possa ser carregado ou, então, para gerar uma mensagem no terminal indicando essa ausência do pacote.

```
library(ggplot2)

# Não exibir mensagens de carregamento do pacote
suppressPackageStartupMessages(library(ggplot2))
```

```r
# criando uma lógica simples com require para instalar pacotes que
# não possam ser carregados
if(!require(ggplot2)) {
  install.packages("ggplot2")
  require(ggplot2)
}

# usando uma função do pacote sem o carregar (namespace qualification)
head(dplyr::filter(mtcars, mpg > 20), 2)
```

```
              mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4
```

```r
# carregando vários pacotes de uma lista de nomes
pacotes <- c("ggplot2", "dplyr", "tidyr")
x <- lapply(pacotes, library, character.only = TRUE, quietly = TRUE)
```

Além dessas funções para instalação e carregamento de pacotes, também outras funções que devem ser conhecidas na rotina são as de atualização (`update.packages()`) e remoção (`remove.packages()`) de pacotes, descrição (`packageDescription()`), versão (`packageVersion()`) e forma recomendada pelo seus autores de citação (`citation()`) quando usada em uma publicação.

```r
# Atualização de pacotes
update.packages()  # todos
update.packages(ask = FALSE)  # todos, mas exige confirmação

# apagar um pacote
remove.packages("nome_pacote")
```

```r
# descrição e versão
packageDescription("ggplot2")
```

```
Package: ggplot2
Title: Create Elegant Data Visualisations Using the Grammar of Graphics
Version: 4.0.0
Authors@R: c( person("Hadley", "Wickham", , "hadley@posit.co", role =
        "aut", comment = c(ORCID = "0000-0003-4757-117X")),
        person("Winston", "Chang", role = "aut", comment = c(ORCID =
        "0000-0002-1576-2126")), person("Lionel", "Henry", role =
        "aut"), person("Thomas Lin", "Pedersen", ,
        "thomas.pedersen@posit.co", role = c("aut", "cre"), comment =
        c(ORCID = "0000-0002-5147-4711")), person("Kohske",
        "Takahashi", role = "aut"), person("Claus", "Wilke", role =
        "aut", comment = c(ORCID = "0000-0002-7470-9261")),
        person("Kara", "Woo", role = "aut", comment = c(ORCID =
        "0000-0002-5125-4188")), person("Hiroaki", "Yutani", role =
        "aut", comment = c(ORCID = "0000-0002-3385-7233")),
        person("Dewey", "Dunnington", role = "aut", comment = c(ORCID =
        "0000-0002-9415-4582")), person("Teun", "van den Brand", role =
        "aut", comment = c(ORCID = "0000-0002-9335-7468")),
        person("Posit, PBC", role = c("cph", "fnd"), comment = c(ROR =
```

```
        "03wc8by49")) )
Description: A system for 'declaratively' creating graphics, based on
        "The Grammar of Graphics". You provide the data, tell 'ggplot2'
        how to map variables to aesthetics, what graphical primitives
        to use, and it takes care of the details.
License: MIT + file LICENSE
URL: https://ggplot2.tidyverse.org,
        https://github.com/tidyverse/ggplot2
BugReports: https://github.com/tidyverse/ggplot2/issues
Depends: R (>= 4.1)
Imports: cli, grDevices, grid, gtable (>= 0.3.6), isoband, lifecycle (>
        1.0.1), rlang (>= 1.1.0), S7, scales (>= 1.4.0), stats, vctrs
        (>= 0.6.0), withr (>= 2.5.0)
Suggests: broom, covr, dplyr, ggplot2movies, hexbin, Hmisc, knitr,
        mapproj, maps, MASS, mgcv, multcomp, munsell, nlme, profvis,
        quantreg, ragg (>= 1.2.6), RColorBrewer, rmarkdown, roxygen2,
        rpart, sf (>= 0.7-3), svglite (>= 2.1.2), testthat (>= 3.1.5),
        tibble, vdiffr (>= 1.0.6), xml2
Enhances: sp
VignetteBuilder: knitr
Config/Needs/website: ggtext, tidyr, forcats, tidyverse/tidytemplate
Config/testthat/edition: 3
Config/usethis/last-upkeep: 2025-04-23
Encoding: UTF-8
LazyData: true
RoxygenNote: 7.3.2
Collate: 'ggproto.R' 'ggplot-global.R' 'aaa-.R'
        'aes-colour-fill-alpha.R' .....
NeedsCompilation: no
Packaged: 2025-08-19 08:21:45 UTC; thomas
Author: Hadley Wickham [aut] (ORCID:
        <https://orcid.org/0000-0003-4757-117X>), Winston Chang [aut]
        (ORCID: <https://orcid.org/0000-0002-1576-2126>), Lionel Henry
        [aut], Thomas Lin Pedersen [aut, cre] (ORCID:
        <https://orcid.org/0000-0002-5147-4711>), Kohske Takahashi
        [aut], Claus Wilke [aut] (ORCID:
        <https://orcid.org/0000-0002-7470-9261>), Kara Woo [aut]
        (ORCID: <https://orcid.org/0000-0002-5125-4188>), Hiroaki
        Yutani [aut] (ORCID: <https://orcid.org/0000-0002-3385-7233>),
        Dewey Dunnington [aut] (ORCID:
        <https://orcid.org/0000-0002-9415-4582>), Teun van den Brand
        [aut] (ORCID: <https://orcid.org/0000-0002-9335-7468>), Posit,
        PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>)
Maintainer: Thomas Lin Pedersen <thomas.pedersen@posit.co>
Repository: CRAN
Date/Publication: 2025-09-11 07:10:02 UTC
Built: R 4.3.3; ; 2025-10-07 18:16:38 UTC; unix

-- File: /home/marcelo/R/x86_64-pc-linux-gnu-library/4.3/ggplot2/Meta/package.rds
```

```r
packageVersion("ggplot2")
```

```
[1] '4.0.0'
```

```r
# forma de citação
citation("ggplot2")
```

```
To cite ggplot2 in publications, please use

  H. Wickham. ggplot2: Elegant Graphics for Data Analysis.
  Springer-Verlag New York, 2016.

A BibTeX entry for LaTeX users is

  @Book{,
    author = {Hadley Wickham},
    title = {ggplot2: Elegant Graphics for Data Analysis},
    publisher = {Springer-Verlag New York},
    year = {2016},
    isbn = {978-3-319-24277-4},
    url = {https://ggplot2.tidyverse.org},
  }
```

Algo a se ter em mente é que nada impede de vários pacotes terem o mesmo nome para funções com finalidades diferentes. Nesse caso, ao carregar esses pacotes, o último a ser carregado irá mascarar o nome da anterior no seu ambiente. Assim, para evitar conflitos, ou o uso da função errada, recomenda-se usar a função seguindo o padrão `nome_do_pacote::nome_da_função`.

1.2 Leitura de datasets externos ao R

A importação de dados é o primeiro passo em qualquer análise. O R oferece múltiplos pacotes especializados para diferentes formatos de arquivos, mas iremos focar nos pacotes de leitura dos arquivos provenientes dos softwares Excel, SAS, Stata e SPSS. Para isso, utilizaremos os pacotes `readxl` e `haven`.

```r
# mini rotina para instalar um pacote se ainda não estiver instalado
instala_se_nao_existe <- function(nome_do_pacote){
  if(nome_do_pacote %in% rownames(installed.packages())) return()
  install.packages(nome_do_pacote)
}
lapply(c("readxl", "haven"), instala_se_nao_existe)

# Carregar pacotes
library(readxl)    # Excel
library(haven)     # SAS, SPSS, STATA
```

1.2.1 Importando dados do Excel

Para leitura de arquivos do Excel nos formatos .xls e .xlsx usaremos o pacote `readxl`, o qual faz parte do conjunto de pacotes do `tidyverse`. Dele podemos usar as funções `read_excel()`, `read_xls()` ou `read_xlsx()`, os quais recebem argumentos semelhantes, com a diferença que os dois últimos são específicos ao formato do arquivo.

O primeiro e mais importante argumento a ser fornecido para essa função é o `path`, o local onde o arquivo se encontra no seu computador. Esse caminho pode ser absoluto (desde a raiz, normalmente `/` no linux ou `C:` no windows, até o local) ou relativo ao diretório de trabalho (que pode ser verificado usando a função `getwd()`).

Como os arquivos do Excel aceitam múltiplas planilhas (em diferentes abas), o argumento de `sheet` do `read_excel()` permite escolher qual aba se pretende carregar. Caso seja necessário verificar primeiro o nome das abas disponíveis no arquivo, use `excel_sheets(path)`.

Outro problema comum em arquivos do Excel são planilhas que não iniciam na linha 1 ou que apresentam um conjunto de colunas que não pretendemos usar (sem conteúdo ou preenchido com informações que não fazem parte do dateset). Para contornar esses obstáculos, podemos usar o argumento `skip` com o número de linhas iniciais que não devem ser lidas, ou usar o `range` com um `character` indicando a primeira e última células que delimitam seus dados (por exemplo, `range = "B2:D20"` indica que devem ser lidas as colunas B, C e D, das linhas 2 até a 20).

Por padrão, essas funções buscam adivinhar o tipo de dados presente em cada coluna da planilha, mas é possível declarar o tipo usando o argumento col_types com um vetor com comprimento igual ao número de colunas que irá importar. Esse vetor deve, para cada coluna, usar uma das opções:

- "skip": remove a coluna do dataset
- "guess": deixa para a função escolher o tipo
- "logical": booleano
- "numeric": numérico
- "date": data
- "text": character
- "list": lista

Também por padrão, a primeira linha é usada para obter os nomes de cada coluna. Se você não possui nomes das colunas na sua planilha use `col_names = FALSE` na função ou passe um vetor dos nomes das colunas para o argumento `col_names`.

Um aspecto importante de qualquer conjunto de dados é saber como foram codificados os dados ausentes. O argumento `na` permite passar um vetor de `character` com os códigos usados na planilha para declarar um dado ausente, o qual será convertido para `NA` no R.

```
excel_sheets("../datasets/excel/ap2.xlsx")
```

```
[1] "Data"
```

```
dados_excel <- read_excel("../datasets/excel/ap2.xlsx")
head(dados_excel)
```

```
# A tibble: 6 x 21
  farm_id batch_id litt_id pig_id parity vacc_mp seas_fin age_t w_age_t age_t6
    <dbl>    <dbl>   <dbl>  <dbl>  <dbl>   <dbl>    <dbl> <dbl>   <dbl>  <dbl>
1       1        1       1      1      8       1        1    70    33.8    116
2       1        1       1      2      8       1        1    70    32.9    116
3       1        1       1      3      8       1        1    70    29.4    116
4       1        1       2      4      8       1        1    60    19.8    106
5       1        1       2      5      8       1        1    60    20.4    106
6       1        1       2      6      8       1        1    60    20.3    106
# i 11 more variables: w_age_t6 <dbl>, dwg_fin <dbl>, ap2_t <dbl>, mp_t <dbl>,
#   infl_t <dbl>, prrs_t <dbl>, ap2_t6 <dbl>, mp_t6 <dbl>, infl_t6 <dbl>,
#   prrs_t6 <dbl>, ap2_sc <dbl>
```

```r
# definir a planilha por nome ou índice
dados_pela_aba <- read_excel("../datasets/excel/ap2.xlsx", sheet = "Data")
dados_pela_aba <- read_excel("../datasets/excel/ap2.xlsx", sheet = 1)

# carregar somente um intervalo de células, em que a linha 1 não é header
dados_pelo_range <- read_excel(
  "../datasets/excel/ap2.xlsx",
  range = "A2:B100",
  sheet = "Data",
  col_names = FALSE
)
```

```
New names:
* `` -> `...1`
* `` -> `...2`
```

```r
head(dados_pelo_range)
```

```
# A tibble: 6 x 2
   ...1  ...2
  <dbl> <dbl>
1     1     1
2     1     1
3     1     1
4     1     1
5     1     1
6     1     1
```

```r
# mesmo exemplo, mas definindo os nomes das colunas
dados_pelo_range <- read_excel(
  "../datasets/excel/ap2.xlsx",
  range = "A2:B100",
  sheet = "Data",
  col_names = c('fazenda', 'lote')
)
head(dados_pelo_range)
```

```
# A tibble: 6 x 2
  fazenda  lote
    <dbl> <dbl>
1       1     1
2       1     1
3       1     1
4       1     1
5       1     1
6       1     1
```

```r
# declarando os tipos de colunas
dados_tipos <- read_excel(
  "../datasets/excel/ap2.xlsx",
  col_types = c("text", "numeric", rep("text", 19))
)
```

```r
head(dados_tipos)
```

```
# A tibble: 6 x 21
  farm_id batch_id litt_id pig_id parity vacc_mp seas_fin age_t w_age_t age_t6
  <chr>      <dbl> <chr>   <chr>  <chr>  <chr>   <chr>    <chr> <chr>   <chr>
1 1              1 1       1      8      1       1        70    33.8    116
2 1              1 1       2      8      1       1        70    32.9    116
3 1              1 1       3      8      1       1        70    29.4    116
4 1              1 2       4      8      1       1        60    19.8    106
5 1              1 2       5      8      1       1        60    20.4    106
6 1              1 2       6      8      1       1        60    20.3    106
# i 11 more variables: w_age_t6 <chr>, dwg_fin <chr>, ap2_t <chr>, mp_t <chr>,
#   infl_t <chr>, prrs_t <chr>, ap2_t6 <chr>, mp_t6 <chr>, infl_t6 <chr>,
#   prrs_t6 <chr>, ap2_sc <chr>
```

```r
# definir os códigos usados na planilha para dados ausentes
dados_na <- read_excel(
  "../datasets/excel/ap2.xlsx",
  na = c("", "NA", "N/A", "-")
)
```

1.2.2 Importando dados do Stata

Para leitura de arquivos do Stata no formato .dta usaremos o pacote `heaven`, o qual possui funções para leitura de arquivos do Stata, SPSS e SAS. Nesse treinamento vamos focar na função `read_dta()` para leitura dos arquivos do Stata (superiores a versão 13.0).

Assim como no `read_excel()`, o primeiro argumento de `read_dta()` deve ser a localização do arquivo. Além disso, a função aceita como argumentos `encoding`, a codificação de carácteres usada, `skip` para remover um certo número de linhas, `col_select` para definir quais colunas serão selecionadas e `n_max` para declarar o número máximo de linhas que devem ser importadas.

Um diferença importante entre arquivos do Excel e do Stata é que no segundo o dataset e as suas variáveis podem conter metadados ("notes" e "labels") com informações sobre esses dados. Essas informações podem ser acessadas na função `attr()`.

```r
dados_stata <- read_dta("../datasets/stata/ap2.dta")
head(dados_stata)
```

```
# A tibble: 6 x 21
  farm_id batch_id litt_id pig_id parity vacc_mp   seas_fin age_t w_age_t age_t6
    <dbl>    <dbl>   <dbl>  <dbl>  <dbl> <dbl+lbl> <dbl+lb> <dbl>   <dbl>  <dbl>
1       1        1       1      1      8 1 [vac]   1 [wint~    70    33.8    116
2       1        1       1      2      8 1 [vac]   1 [wint~    70    32.9    116
3       1        1       1      3      8 1 [vac]   1 [wint~    70    29.4    116
4       1        1       2      4      8 1 [vac]   1 [wint~    60    19.8    106
5       1        1       2      5      8 1 [vac]   1 [wint~    60    20.4    106
6       1        1       2      6      8 1 [vac]   1 [wint~    60    20.3    106
# i 11 more variables: w_age_t6 <dbl>, dwg_fin <dbl>, ap2_t <dbl+lbl>,
#   mp_t <dbl+lbl>, infl_t <dbl+lbl>, prrs_t <dbl+lbl>, ap2_t6 <dbl+lbl>,
#   mp_t6 <dbl+lbl>, infl_t6 <dbl+lbl>, prrs_t6 <dbl+lbl>, ap2_sc <dbl+lbl>
```

```r
dados_stata_com_encoding <- read_dta(
  "../datasets/stata/ap2.dta",
  encoding = "UTF-8"
)

# tranformar colunas labelled em factor
dados_stata_como_factor <- read_dta(
  "../datasets/stata/ap2.dta",
  encoding = "UTF-8"
) |> as_factor()
head(dados_stata_como_factor)
```

```
# A tibble: 6 x 21
  farm_id batch_id litt_id pig_id parity vacc_mp seas_fin age_t w_age_t age_t6
    <dbl>    <dbl>   <dbl>  <dbl>  <dbl> <fct>   <fct>    <dbl>   <dbl>  <dbl>
1       1        1       1      1      8 vac     winter      70    33.8    116
2       1        1       1      2      8 vac     winter      70    32.9    116
3       1        1       1      3      8 vac     winter      70    29.4    116
4       1        1       2      4      8 vac     winter      60    19.8    106
5       1        1       2      5      8 vac     winter      60    20.4    106
6       1        1       2      6      8 vac     winter      60    20.3    106
# i 11 more variables: w_age_t6 <dbl>, dwg_fin <dbl>, ap2_t <fct>, mp_t <fct>,
#   infl_t <fct>, prrs_t <fct>, ap2_t6 <fct>, mp_t6 <fct>, infl_t6 <fct>,
#   prrs_t6 <fct>, ap2_sc <fct>
```

```r
# Notas do Stata
attr(dados_stata, "notes")
```

```
[1] "5 Aug 2002 16:24 data provided by Dr. Haakan Vigre, Denmark"
[2] "1"
```

```r
# Labels das variáveis
labels <- sapply(dados_stata, function(x) attr(x, "label"))
kable(
  tibble(var=names(labels), metadata=labels),
  col.names = c("Variável", "Label")
)
```

Variável	Label
farm_id	farm identification
batch_id	batch identifiaction number
litt_id	litter identification number
pig_id	pig identification
parity	the farrowing no. of the sow
vacc_mp	the batch vaccinated against M.hyop yes=1
seas_fin	prod. season in finishing unit: winther=1
age_t	pig-age transfer from weaning to finishing unit
w_age_t	weight in kg. at age_t
age_t6	age_tra plus approx. 6 weeks
w_age_t6	weight in kg. at age_t6
dwg_fin	dwg in g. between age_t and age_t6

Variável	Label
ap2_t	serological reac. against A.pleuropneumoniae serotype 2 at age_t
mp_t	serological reac. against M.hyopneumoniae at age_t
infl_t	serological reac. against Influenza virus at age_t
prrs_t	serological reac. against PRRS virus at age_t
ap2_t6	serological reac. against A.pleuropneumoniae serotype 2 at age_t6
mp_t6	serological reac. against M.hyopneumoniae at age_t6
infl_t6	serological reac. against Influenza virus at age_t6
prrs_t6	serological reac. against PRRS virus at age_t6
ap2_sc	seroconversion to ap2 during the finishing period

1.2.3 Verificação e diagnóstico dos dados importados

Uma vez carregados os dados, é importante avaliar a estrutura desse conjunto de dados importado. Para uma exploração inicial, será interessante avaliar, no mínimo, as dimensões desses dados (número de observações e variáveis), quais os tipos das variáveis no R, resumos estatísticos simples, quantidade de valores ausentes por variável.

```
verificar_dados <- function(dados) {
  cat("Dimensões:", dim(dados), "\n")
  cat("Tipos de variáveis:\n")
  print(sapply(dados, class))
  cat("\nPrimeiras linhas:\n")
  print(head(dados, 3))
  cat("\nResumo estatístico:\n")
  print(summary(dados))
  cat("\nValores missing por coluna:\n")
  print(colSums(is.na(dados)))
  cat("\nStructura dos dados:\n")
  str(dados)
}

# Aplicar a qualquer dataset importado
verificar_dados(dados_stata)
```

Por fim, em grandes datasets é comum que os dados sejam registrados em múltiplos arquivos (principalmente no Excel, por causa do limite de linhas). Nesse caso, para não ser necessário carregar cada um desses arquivos e depois construir um data.frame que uni todos, podemos usar recursos de programação funcional do pacote `purrr` para importar diretamente todos os arquivos em um único data.frame.

```
library(purrr)
# obter uma lista dos arquivos que serão importados e
# mapear todos os arquivos para um unico data.frame
dados <- list.files("datasets/csv", pattern = "\\.csv$", full.names = TRUE) |> map_df(read_csv2)
```

Quadro Resumo das funções que podem ser usadas na importação de arquivos externos ao R

Formato	Pacote	Função
CSV	`readr`	`read_delim()`, `read_csv()`, `read_csv2()`
Excel	`readxl`	`read_excel()`, `read_xls()`, `read_xlsx()`

Formato	Pacote	Função
SAS	`haven`	`read_sas()`
SPSS	`haven`	`read_sav()`
Stata	`haven`	`read_stata()`, `read_dta()`
Múltiplos	`rio`	`import()`

Chapter 2

Manipulação de dados com os pacotes do `tidyverse`

O **tidyverse** é uma coleção de pacotes R voltados para a ciência de dados, que compatilham uma mesma filosofia, gramática e estruturas de dados. Ele é composto dos seguintes pacotes:

- tibble: extensão do `data.frame`;
- dplyr: funções na forma de verbos que fornece a gramática para a manipulação dos dados;
- tidyr: funções para obtenção dos dados que seguem a filosia dos "dados arrumados";
- readr: importação de dados tabulares (csv, tsv, fwf);
- purrr: programação funcional;
- stringr: manipulação de strings (`character`);
- forcats: manipulação de fatores (`factor`);
- lubridate: manipulação de datas (`date`);
- ggplot2: criação de gráficos.

```
# Carregar todo o conjunto de pacotes
library(tidyverse)

# Ou carregar pacotes individuais
library(dplyr)
library(tidyr)
library(readr)
```

Como já mencionado, o `tidyverse` segue a filosofia de "dados arrumados" (*tidy data*), o que basicamente significa que:

- **cada variável forma uma coluna;**
- **cada unidade observacional (unidade amostral) forma uma linha;**
- **cada célula é uma observação e um único valor**

Esse padrão facilita análises posteriores:

```
set.seed(42)
dados_tidy <- tibble(
  animal_id = 1:6,
  especie = rep(c("cão", "gato"), 3),
```

```
  peso_kg = round(rnorm(6, mean = 15, sd = 5), 1),
  idade_anos = sample(1:10, 6, replace = TRUE)
)
dados_tidy
```

```
# A tibble: 6 x 4
  animal_id especie peso_kg idade_anos
      <int> <chr>     <dbl>      <int>
1         1 cão        21.9          7
2         2 gato       12.2          4
3         3 cão        16.8          9
4         4 gato       18.2          5
5         5 cão        17            4
6         6 gato       14.5         10
```

O `tibble` facilita a compreensão dos seus dados, uma vez que sua impressão (com `print`) apresenta o tipo de cada variável, não imprime o conjunto completo (somente as primeiras linhas) e não faz conversões automáticas de variáveis `character` para `factor`. Além disso, ele aceita nomes não sintáticos do R para as variáveis (usando “).

Uma diferença importante entre `tibble` e `data.frame` está na forma como você extrai uma variável do conjunto. No `data.frame` usamos os padrões `nome_do_dataframe["nome_da_coluna"]`, `nome_do_dataframe[indice_da_coluna]` ou `nome_do_dataframe$nome_da_coluna`. No `tibble` usamos os padrões `nome_do_tibble$nome_da_coluna`, `nome_do_tibble[[indice_da_coluna]]`, `nome_do_tibble[["nome_da_coluna"]]` ou, ainda, extrair por meio do pipe com `nome_do_tibble %>% .$nome_da_coluna`, `nome_do_tibble %>% .[["nome_da_coluna"]]` ou `nome_do_tibble |> pull("nome_da_coluna")`.

```
# extraindo uma variável do tibble
dados_tidy %>% .$idade_anos
```

```
[1]  7  4  9  5  4 10
```

```
dados_tidy %>% .[["idade_anos"]]
```

```
[1]  7  4  9  5  4 10
```

```
dados_tidy |> pull("idade_anos")
```

```
[1]  7  4  9  5  4 10
```

2.1 O pipe (%>% e |>)

O pipe permite agrupar em um código que parece ser uma única operação múltiplas operações (chamadas de funções), em que o resultado de uma operação e fornecido como o primeiro argumento da subsequente. Isso torna o código mais legível.

```
# Sem pipe, com funções aninhadas
resultado <- summarise(
  group_by(
    filter(dados_tidy, peso_kg > 10),
    especie
  ),
```

```
  peso_medio = mean(peso_kg)
)
# ou criando várias etapas
dados_filtrados <- filter(dados_tidy, peso_kg > 10)
dados_agrupados <- group_by(dados_filtrados, especie)
resultado <- summarise(dados_agrupados, peso_medio = mean(peso_kg))

# Com pipe do tidyverse (%>%)
resultado <- dados_tidy %>%
  filter(peso_kg > 10) %>%
  group_by(especie) %>%
  summarise(peso_medio = mean(peso_kg))

# com pipe nativo do R 4.1+ (|>)
resultado <- dados_tidy |>
  filter(peso_kg > 10) |>
  group_by(especie) |>
  summarise(peso_medio = mean(peso_kg))
```

2.2 Manipulação dos dados com o `dplyr`

Usamos o `select()` para obter um subconjunto do nosso dataset com somente as variáveis de interesse.

```
df_exemplo <- tibble(
  id = 1:100,
  especie = sample(c("cão", "gato", "coelho"), 100, replace = TRUE),
  idade = round(runif(100, 1, 15), 1),
  peso = round(rnorm(100, 15, 5), 2),
  vacinado = sample(c(TRUE, FALSE), 100, replace = TRUE),
  data_consulta = seq(as.Date("2023-01-01"), by = "day", length.out = 100),
  temperatura = round(rnorm(100, 38.5, 0.5), 1)
)
```

```
# seleção de colunas pelo nome ou com vetor de caractéres
df_exemplo %>%
  select(id, especie, peso)
df_exemplo %>%
  select(c("id", "especie", "peso"))

# Seleção com funções helpers
# starts_with para as colunas que iniciam com certo valor
df_exemplo %>%
  select(starts_with("data"))
# ends_with para as colunas que terminam com certo valor
df_exemplo %>%
  select(ends_with("do"))
# contains para as colunas que possuem um certo valor
df_exemplo %>%
  select(contains("ac"))
# where para colunas que correspondem a TRUE para alguma função de retorno lógico
```

```r
df_exemplo %>%
  select(where(is.numeric))
# usando padrão de fórmula para múltiplas condições
df_exemplo %>%
  select(where(~ is.numeric(.x) && min(.x) > 10))

# Seleção pela exclusão de determinadas colunas
df_exemplo %>%
  select(-id, -data_consulta)
df_exemplo %>%
  select(-c("id", "data_consulta"))

# Seleção com renomeação de determinadas colunas
df_exemplo %>%
  select(
    identificador = id,
    tipo_animal = especie,
    everything()
  )
```

Usamos o `filter()` para obter um subconjunto do nosso dataset com somente as observações que atendem a uma determinada condição.

```r
# Filtro básico
df_exemplo %>%
  filter(especie == "cão")

# Múltiplas condições
# AND - , ou &
df_exemplo %>%
  filter(especie == "gato", peso > 10, vacinado == TRUE)
df_exemplo %>%
  filter(especie == "gato" & peso > 10 & vacinado == TRUE)

# OR - |
df_exemplo %>%
  filter(especie == "cão" | especie == "gato")

# agrupando os OR de == com %in%
df_exemplo %>%
  filter(especie %in% c("cão", "gato"))

# Filtros com funções
# between para min <= x <= max
df_exemplo %>%
  filter(between(idade, 5, 10))

df_exemplo %>%
  filter(!is.na(temperatura))
```

Usamos o `mutate()` para criar ou modificar variáveis.

```r
# criar colunas
# case_when para construir uma variável baseado em condições das demais
df_exemplo %>%
  mutate(
    score_inventado = peso / (idade ^ 0.5),
    categoria_idade = case_when(
      idade < 1 ~ "Filhote",
      idade < 7 ~ "Adulto",
      TRUE ~ "Idoso"
    )
  ) %>%
  select(id, especie, idade, categoria_idade, score_inventado)

# modificar colunas
df_exemplo %>%
  mutate(
    peso = round(peso, 0),
    temperatura = temperatura * 9/5 + 32
  )

# transformações em várias colunas
# scale centraliza a variável (desvio / desvio-padrão)
# cuidado para multiplos across no mutate, a ordem importa
df_exemplo %>%
  mutate(
    across(where(is.numeric), ~round(.x, 1)),
    across(c(peso, temperatura), ~scale(.x)[,1], .names = "{.col}_z")
  )
df_exemplo %>%
  mutate(
    across(c(peso, temperatura), ~scale(.x)[,1], .names = "{.col}_z"),
    across(where(is.numeric), ~round(.x, 1))
  )

# transmute() - mantém apenas as colunas criadas
df_exemplo %>%
  transmute(
    id,
    peso_libras = peso * 2.205,
    idade_meses = idade * 12
  )
```

Usamos o `arrange()` para ordenar as variáveis por uma ou mais variáveis.

```r
# Ordenação crescente por uma variável
df_exemplo %>%
  arrange(peso)

# Ordenação decrescente por uma variável
df_exemplo %>%
  arrange(desc(peso))
```

```r
# Ordenação múltipla
df_exemplo %>%
  arrange(especie, desc(idade), peso)

# Ordenação com NA
df_exemplo %>%
  arrange(desc(is.na(temperatura)), temperatura)
```

Para criar agregações (resumos estatíticos) usamos `summarise()` e o `group_by()` caso esse resumo deva ser calculado para cada categoria de determinada variável.

```r
# Resumo simples
df_exemplo %>%
  summarise(
    n_animais = n(),
    peso_medio = mean(peso, na.rm = TRUE),
    peso_dp = sd(peso, na.rm = TRUE),
    peso_mediana = median(peso, na.rm = TRUE),
    peso_min = min(peso, na.rm = TRUE),
    peso_max = max(peso, na.rm = TRUE)
  )

# Agrupamento e resumo
df_exemplo %>%
  group_by(especie) %>%
  summarise(
    n = n(),
    idade_media = mean(idade, na.rm = TRUE),
    prop_vacinados = mean(vacinado, na.rm = TRUE),
    .groups = "drop"
  )

# Agrupamento por múltiplas variáveis
df_exemplo %>%
  group_by(especie, vacinado) %>%
  summarise(
    n = n(),
    peso_medio = mean(peso, na.rm = TRUE),
    temp_media = mean(temperatura, na.rm = TRUE),
    .groups = "drop"
  ) %>%
  arrange(especie, vacinado)

# Criando colunas baseado nas informações do grupo (categoria)
df_exemplo %>%
  group_by(especie) %>%
  transmute(
    especie,
    peso,
    peso_padronizado = (peso - mean(peso)) / sd(peso),
```

```
    peso_centralizado = scale(peso)[,1],
    rank_peso = rank(peso)
  ) %>%
  ungroup() %>%
  arrange(rank_peso)
```

Para obter um subconjunto de observações também podemos usar funções da família `slice_*`.

```
# Primeiras ou últimas n observações
df_exemplo %>% slice_head(n = 5)
df_exemplo %>% slice_tail(n = 5)

# Linhas específicas
df_exemplo %>% slice(c(1, 5, 10))

# Amostragem "aleatória"
df_exemplo %>% slice_sample(n = 10)
df_exemplo %>% slice_sample(prop = 0.1)

# Extremos por grupo
df_exemplo %>%
  group_by(especie) %>%
  slice_max(peso, n = 3) # 3 maiores valores
df_exemplo %>%
  group_by(especie) %>%
  slice_min(idade, n = 2) # 2 menores valores
# retornando exatamente o valor n
df_exemplo %>%
  group_by(especie) %>%
  slice_min(idade, n = 2, with_ties = FALSE)
```

Para obter os valores únicos de uma ou mais variáveis usamos `distinct()`.

```
# Valores únicos de uma coluna
df_exemplo %>%
  distinct(especie)

# Combinações únicas
# .keep_all = TRUE para manter as outras colunas
df_exemplo %>%
  distinct(especie, vacinado, .keep_all = TRUE)

# Remover duplicatas
df_exemplo %>%
  distinct()
```

Para contagem de ocorrências usamos `count()` e `add_count()`.

```
# quantidade de observações por categoria
df_exemplo %>%
  count(especie, sort = TRUE, name = "Amostra")
```

```r
# Contagem com peso
df_exemplo %>%
  count(especie, wt = peso, name = "peso_total")
# mesmo que agrupar e agregar para a soma
df_exemplo %>%
  group_by(especie) %>%
  summarise(
    peso_total = sum(peso)
  )

# Adicionar contagem sem agregar
df_exemplo %>%
  add_count(especie, name = "n_por_especie") %>%
  add_count(especie, wt = peso, name = "peso_por_especie") %>%
  select(id, especie, n_por_especie, peso_por_especie)
```

2.3 Reestruturação dos dados com o `tidyr`

Algumas vezes os dados importados apresentam um conjunto de colunas que precisam ser transformadas em uma única com seu valor e outra com a categoria, ou o contrário. Para conseguir isso usamos as funções `pivot_longer()` e `pivot_wider()`. `pivot_longer()` retorna um dataset com mais observações e menos colunas, usando os nomes das colunas alvo para construir uma variável e o valor de cada coluna para construir outra. `pivot_wider()` retorna um dataset com mais colunas e menos observações, usando os valores de uma ou mais colunas alvo para criar novas variáveis e outra coluna para extrair os valores.

```r
dados_wide <- tibble(
  id = 1:100,
  especie = sample(c("cão", "gato"), 100, replace = TRUE),
  peso_2021 = round(rnorm(100, 13, 5), 2),
  peso_2022 = round(rnorm(100, 15.1, 2), 2),
  peso_2023 = round(rnorm(100, 11.3, 5), 2),
)

# Converter para long
dados_long <- dados_wide %>%
  pivot_longer(
    cols = starts_with("peso"),
    names_to = "ano",
    values_to = "peso",
    names_prefix = "peso_",
    names_transform = list(ano = as.integer)
  )

dados_long %>% slice_sample(n = 4)
```

```
# A tibble: 4 x 4
     id especie   ano  peso
  <int> <chr>   <int> <dbl>
1    30 gato     2021 10.8
2    19 gato     2023 11.6
```

```
3    17 gato     2021  8.12
4    27 gato     2023 17.6
```

```r
dados_wide_novo <- dados_long %>%
  pivot_wider(
    names_from = ano,
    values_from = peso,
    names_prefix = "peso_"
  )

# exemplo mais complexo
medidas_long <- tibble(
  animal_id = rep(1:3, each = 6),
  momento = rep(c("antes", "depois"), each = 3, times = 3),
  parametro = rep(c("peso", "temperatura", "frequencia"), times = 6),
  valor = round(runif(18, 10, 40), 1)
)
medidas_wide <- medidas_long %>%
  pivot_wider(
    names_from = c(parametro, momento),
    values_from = valor,
    names_sep = "_"
  ) %>%
  select(animal_id, starts_with("peso"), starts_with("temperatura"), starts_with("frequencia"))
nomes_tabela <- c("Animal", "Peso (antes)", "Peso (depois)", "Temp (antes)", "Temp (depois)", "Freq (antes
kable(medidas_wide, col.names = nomes_tabela)
```

Animal	Peso (antes)	Peso (depois)	Temp (antes)	Temp (depois)	Freq (antes)	Freq (depois)
1	17.1	25.0	26.7	13.5	29.9	29.3
2	28.8	20.6	17.7	19.2	17.1	33.1
3	16.9	17.7	23.8	23.8	35.1	35.0

Também é possível separar os valores de uma variável em novas variáveis usando o `separate()` ou unir os valores de diferentes variáveis em uma única usando o `unite()`.

```r
df_exemplo_2 <- tibble(
  id = 1:5,
  info = c(
    "cão_macho_5anos",
    "gato_femea_3anos",
    "cão_femea_8anos",
    "gato_macho_2anos",
    "coelho_macho_1anos"
  )
)

# Separar em múltiplas colunas
dados_separados <- df_exemplo_2 %>%
  separate(
    info,
```

```
    into = c("especie", "sexo", "idade"),
    sep = "_",
    convert = TRUE
  ) %>%
  mutate(idade = as.integer(str_remove(idade, "anos")))

# Unir colunas
dados_unidos <- dados_separados %>%
  unite("descricao", especie, sexo, sep = ", ", remove = FALSE)

# separate_rows() separa em múltiplas linhas
dados_unidos <- tibble(
    veterinario = c("Dr. Fulano", "Dra. Sicrano"),
    especialidades = c("cirurgia,clinica", "clinica,dermatologia,cardiologia")
  ) %>%
  separate_rows(especialidades, sep = ",")
```

Quando temos dados ausentes (`NA`) em variáveis do nosso conjunto de dados, normalmente, algum tratamento será necessário para lidar com essas observações, seja removendo elas ou substituindo por outro valor. `fill()` substitui os valores `NA` das variáveis alvo por valores anteriores e posteriores àquela observação no conjunto de dados, enquanto `replace_na()` substitui os valores `NA` por um valor padrão. Já o `drop_na()` remove a unidade observacional inteira que apresente um `NA` nas variáveis alvo.

```
dados_na <- tibble(
  dia = 1:7,
  temperatura = c(38.5, NA, 38.7, NA, NA, 39.0, 38.6),
  medicacao = c("A", NA, NA, "B", NA, NA, "C")
)

# fill() - mais útil para dados temporais
dados_na %>%
  fill(temperatura, .direction = "down")
dados_na %>%
  fill(everything(), .direction = "up")
dados_na %>%
  fill(everything(), .direction = "updown")

# replace_na()
dados_na %>%
  replace_na(list(
    temperatura = 38.5,
    medicacao = "Nenhuma"
  ))

# drop_na()
dados_na %>%
  drop_na()
dados_na %>%
  drop_na(temperatura)
```

Por vezes em nossa análise chegamos em um ponto onde é necessário dividir nosso dataset em multiplos

conjuntos, baseado em alguma categoria, seja para aplicar algum teste ou ajustar um modelo. Esse processo, normalmente, envolveria criar várias variáveis cada uma com o dataset filtrado para a categoria de interesse e depois aplicar o teste a cada uma desses subconjuntos. Entretanto, o `tidyr` oferece uma estratégia mais elegante para esse processo por meio do aninhamento usando o `nest()`. O `nest()` permite criar um novo `tibble` em que, em uma coluna do tipo lista, cada observação se torna um `tibble` filtrado para a categoria de interesse, assim, podemos aplicar o teste de forma iterativa em cada um, por meio de funções de programação funcional do `purrr`, mantendo também o resultado em formato de lista para cada observação.

```
dados_aninhados <- df_exemplo %>%
  group_by(especie) %>%
  nest() %>%
  mutate(
    n_observacoes = map_int(data, nrow),
    modelo = map(data, ~lm(peso ~ idade, data = .x)),
    teste_media_peso_vacinado = map(data, ~t.test(peso ~ vacinado, data = .x))
  )

dados_desaninhados <- dados_aninhados %>%
  select(-modelo) %>%
  unnest(data) %>%
  ungroup()

# resultados de gato
summary(dados_aninhados$modelo[[1]])
```

```

Call:
lm(formula = peso ~ idade, data = .x)

Residuals:
     Min       1Q   Median       3Q      Max
-10.5686  -3.9560   0.4428   2.6257  12.5723

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  12.1732     1.9516   6.237    3e-07 ***
idade         0.3133     0.2401   1.305      0.2
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 5.364 on 37 degrees of freedom
Multiple R-squared:  0.04399,   Adjusted R-squared:  0.01816
F-statistic: 1.703 on 1 and 37 DF,  p-value: 0.2
```

```
dados_aninhados$teste_media_peso_vacinado[[1]]
```

```

    Welch Two Sample t-test

data:  peso by vacinado
t = 0.0048531, df = 34.736, p-value = 0.9962
alternative hypothesis: true difference in means between group FALSE and group TRUE is not equal to 0
```

```
95 percent confidence interval:
 -3.539016  3.555972
sample estimates:
mean in group FALSE  mean in group TRUE
           14.46500            14.45652
```

2.4 Unindo diferentes tabelas com `dplyr`

Outra situação com que se depara na análise de dados é a necessidade de importar dados de diferentes planilhas (tabelas), as quais depois precisamos unir em um único dataset, baseado na informação de alguma variável que ocorre em ambas as tabelas. Isso é particularmente comum quando trabalhamos com dados importados de bancos de dados relacionais, onde temos uma tabela que possui uma coluna de chave primária (valores únicos) e outra com chave estrangeira que indica que aquele registro se refere à observação única daquela outra tabela.

Para unir essas tabelas usamos `joins`, uma operação derivada do SQL, uma linguagem de consulta de banco de dados relacionais. `joins` guardam uma semelhança com a teoria de conjuntos (união, diferença, intersercção).

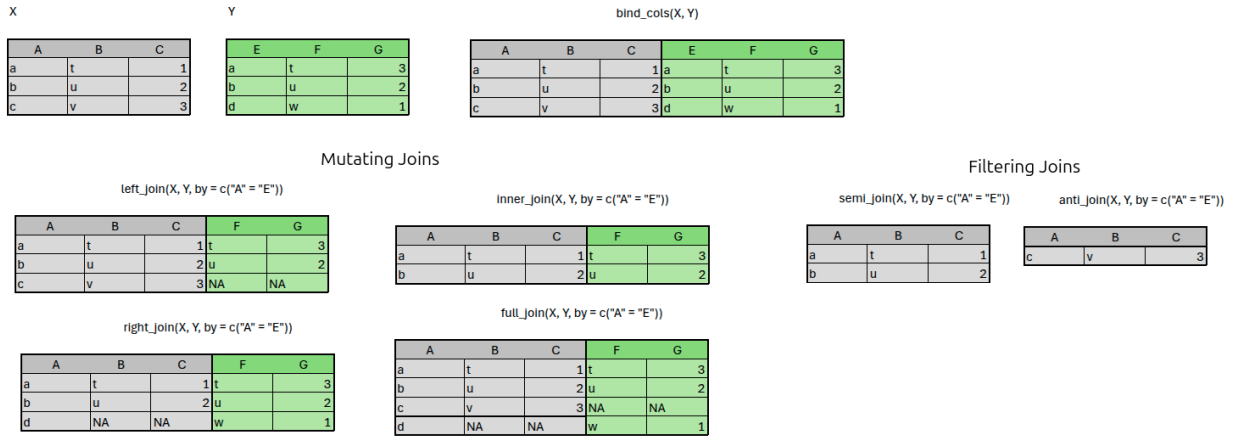


Figura 2.1: Tipos de joins do `dplyr`

```r
animais <- tibble(
  id = 1:5,
  nome = c("Rex", "Mia", "Bob", "Luna", "Max"),
  especie = c("cão", "gato", "cão", "gato", "coelho")
)

consultas <- tibble(
  id_animal = c(1, 2, 1, 3, 6),  # id 6 não existe no tibble animais
  data = as.Date(c(
    "2023-01-10", "2023-01-15", "2023-02-01",
    "2023-02-10", "2023-02-15"
  )),
  motivo = c("vacina", "checkup", "checkup", "vacina", "emergência")
```

```
)

# inner_join - mantém apenas registros com correspondência
inner <- animais %>%
  inner_join(consultas, by = c("id" = "id_animal"))

# left_join - mantém todos da esquerda
left <- animais %>%
  left_join(consultas, by = c("id" = "id_animal"))

# right_join - mantém todos da direita
right <- animais %>%
  right_join(consultas, by = c("id" = "id_animal"))

# full_join - mantém todos
full <- animais %>%
  full_join(consultas, by = c("id" = "id_animal"))

# semi_join - mantém linhas de L que têm match em R
animais_com_consulta <- animais %>%
  semi_join(consultas, by = c("id" = "id_animal"))

# anti_join - mantém linhas de L que NÃO têm match em R
animais_sem_consulta <- animais %>%
  anti_join(consultas, by = c("id" = "id_animal"))
```

2.5 Manipulando texto, datas e fatores com `stringr`, `lubridate` e `forcats`

Quando nosso dataset possui variáveis que representam um texto, por exemplo, uma resposta aberta, é normal ter que manipular essa variável para padronizá-la antes de uma análise. O `stringr` oferece um conjunto de funções que permitem manipular dados do tipo `character`, que inclui meios de tranformar o texto em maiúsculas ou minúsculas, detectar certas palavras, contar o número de ocorrências de uma palavra, remover ou substituir uma palavra por outra ou, ainda, usar expressões regulares (*REGEX*) para tranformar seus dados iniciais.

```
exemplo <- tibble(
  id = 1:5,
  descricao = c(
    "  Infecção respiratória AGUDA  ",
    "dermatite alérgica crônica",
    "FRATURA do fêmur esquerdo",
    "gastroenterite viral",
    "Otite média bilateral"
  )
)

# Funções básicas
diagnosticos_limpos <- exemplo %>%
```

```
  mutate(
    # Remover espaços extras no inicio e fim do texto
    descricao_limpa = str_trim(descricao),

    # Converter para minúsculas
    descricao_lower = str_to_lower(descricao_limpa),

    # Converter para título ("Um Texto Dessa Forma")
    descricao_title = str_to_title(descricao_limpa),

    # Detectar padrões
    tem_infeccao = str_detect(descricao_lower, "infec"),

    # Extrair palavras
    primeira_palavra = str_extract(descricao_limpa, "^\\w+"),

    # Substituir
    descricao_mod = str_replace(descricao_lower, "aguda|crônica", "***"),

    # Contar palavras
    n_palavras = str_count(descricao_limpa, "\\w+")
  )

diagnosticos_limpos

# Expressões regulares
telefones <- c("(11) 1234-5678", "11-98765.4321", "1112345678", "11 1234 5678")
telefones_limpos <- telefones %>%
  str_remove_all("[^0-9]") %>%  # Remove tudo exceto números
  str_replace("^(\\d{2})(\\d{4,5})(\\d{4})$", "(\\1) \\2-\\3")

telefones_limpos
```

O `lubridate` por sua vez permite trabalhar com dados temporais (data e tempo). Com ele você converter dados para os tipos `Date` e `POSIXct` (*date-time*), usar aritméticas entre datas, adcionar ou remover *timezones*, além de criar períodos, durações e intervalos.

```
datas_texto <- c("01/03/2023", "15-06-2023", "2023-12-25")

datas <- tibble(
  texto = datas_texto,
  data_dmy = dmy(c("01/03/2023", "15/06/2023", "25/12/2023")),
  data_dmy_hm = dmy_hm(c("01/03/2023 15:20", "15/06/2023 08:12", "25/12/2023 17:30")),
  data_ymd = ymd("2023-12-25")
)

class(df_exemplo$data_consulta)

# obter os componentes da data
consultas_datas <- df_exemplo %>%
  mutate(
```

```
    ano = year(data_consulta),
    mes = month(data_consulta, label = TRUE),
    dia = day(data_consulta),
    dia_semana = wday(data_consulta, label = TRUE),
    semana_epidemio = epiweek(data_consulta),
    trimestre = quarter(data_consulta),
    dia_ano = yday(data_consulta)
  )

intervencoes <- tibble(
  inicio = ymd(c("2023-01-01", "2023-03-15", "2023-06-01")),
  fim = ymd(c("2023-01-15", "2023-04-01", "2023-06-30"))
) %>%
  mutate(
    duracao_dias = as.numeric(fim - inicio),
    duracao_semanas = as.numeric(difftime(fim, inicio, units = "weeks")),
    meio_periodo = inicio + days(as.integer(duracao_dias / 2))
  )

# criando periodos
intervencoes$inicio + years(1)
# criando durações
intervencoes$inicio + dyears(1)
# criando intervalos
intevalo_um_ano <- interval(
  intervencoes$inicio, intervencoes$inicio + years(1)
)
# testando se uma data está dentro de um intervalo
(intervencoes$inicio[1] + days(20)) %within% intevalo_um_ano

# Sequências de datas
calendario_vacinacao <- tibble(
  data = seq(ymd("2023-01-01"), ymd("2023-12-31"), by = "month"),
  tipo = "Vacinação mensal"
)

# Arredondar datas
agora <- now()
tibble(
  original = agora,
  hora = floor_date(agora, "hour"), # floor arredonda para baixo
  dia = round_date(agora, "day"), # round arredonda para o mais próximo
  semana = ceiling_date(agora, "week"), # ceiling arredonda para cima
  mes = floor_date(agora, "month")
)
```

Para finalizar nossa seção sobre o `tidyverse` temos o pacote `forcats` para manipulação de fatores. Ele possui ferramentas para alterar ou reordenar os fatores de uma variável de forma simples.

```
dados_fatores <- df_exemplo %>%
  mutate(
```

```
    especie_fator = factor(especie),
    especie_freq = fct_infreq(especie_fator), # Reordenar níveis por frequência
    especie_peso = fct_reorder(especie_fator, peso, mean), # Reordenar por outra variável

    # Recodificar níveis
    especie_pt = fct_recode(
      especie_fator,
      "Canino" = "cão", # padrão "Novo nome" = "Antigo nome"
      "Felino" = "gato",
      "Lagomorfo" = "coelho"
    ),

    # Agrupar níveis raros
    especie_agrup = fct_lump_min(
      especie_fator, min = 30, other_level = "Outros"
    )
  )

levels(dados_fatores$especie_fator)
levels(dados_fatores$especie_freq)

dados_fatores %>%
  group_by(especie) %>%
  summarise(media = mean(peso)) %>%
  arrange(media)
levels(dados_fatores$especie_peso)

levels(dados_fatores$especie_pt)
levels(dados_fatores$especie_agrup)
```

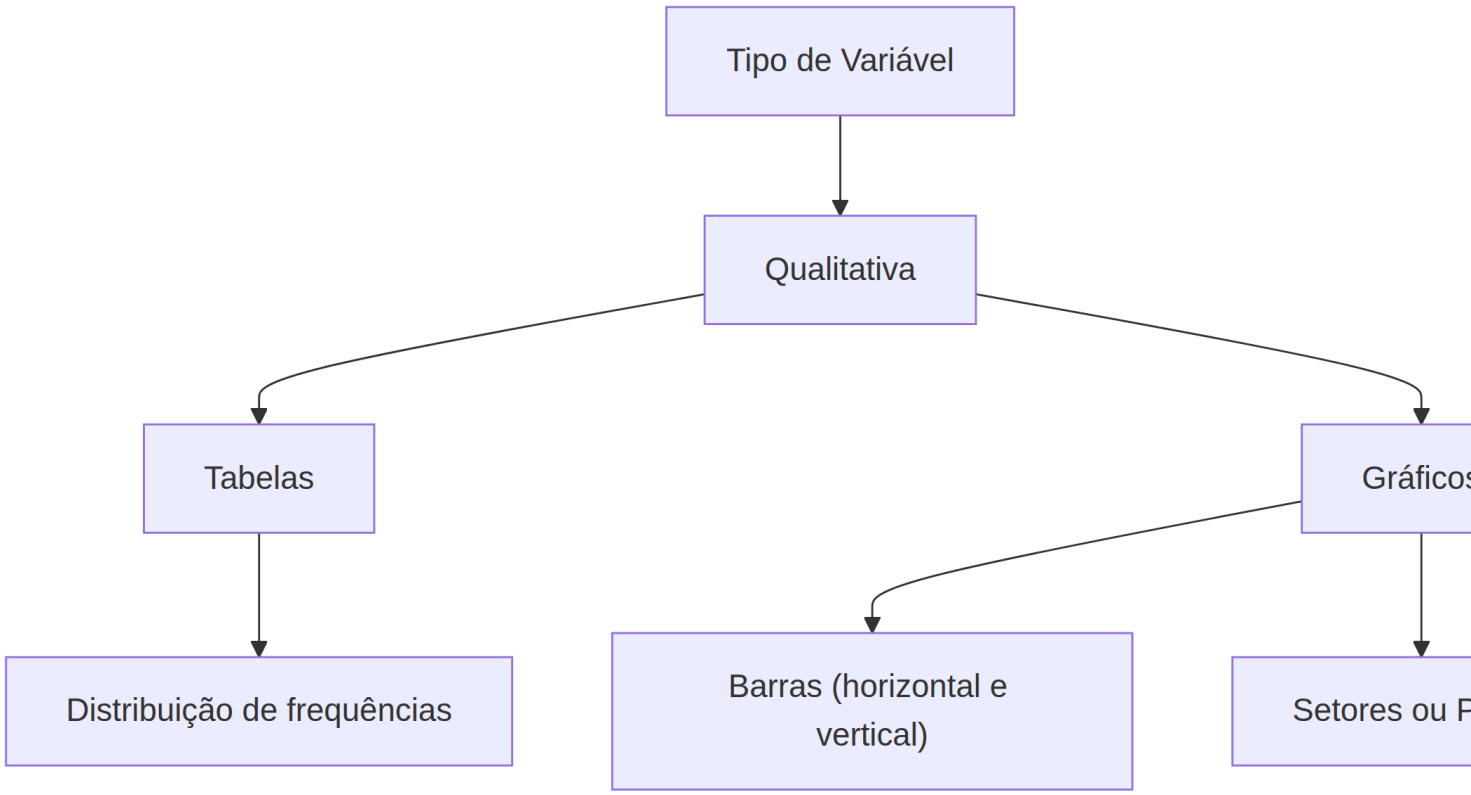
Os pacotes apresentados possuem uma grande gama de ferramentas, e nem todas serão apresentadas no nosso treinamento. Caso queira aprender mais sobre elas, recomendo o livro R para Ciência de Dados, o qual possui uma versão online de acesso aberto ou a documentação dos pacotes do `tidyverse` que podem ser encontrados em https://www.tidyverse.org/packages/.

Chapter 3

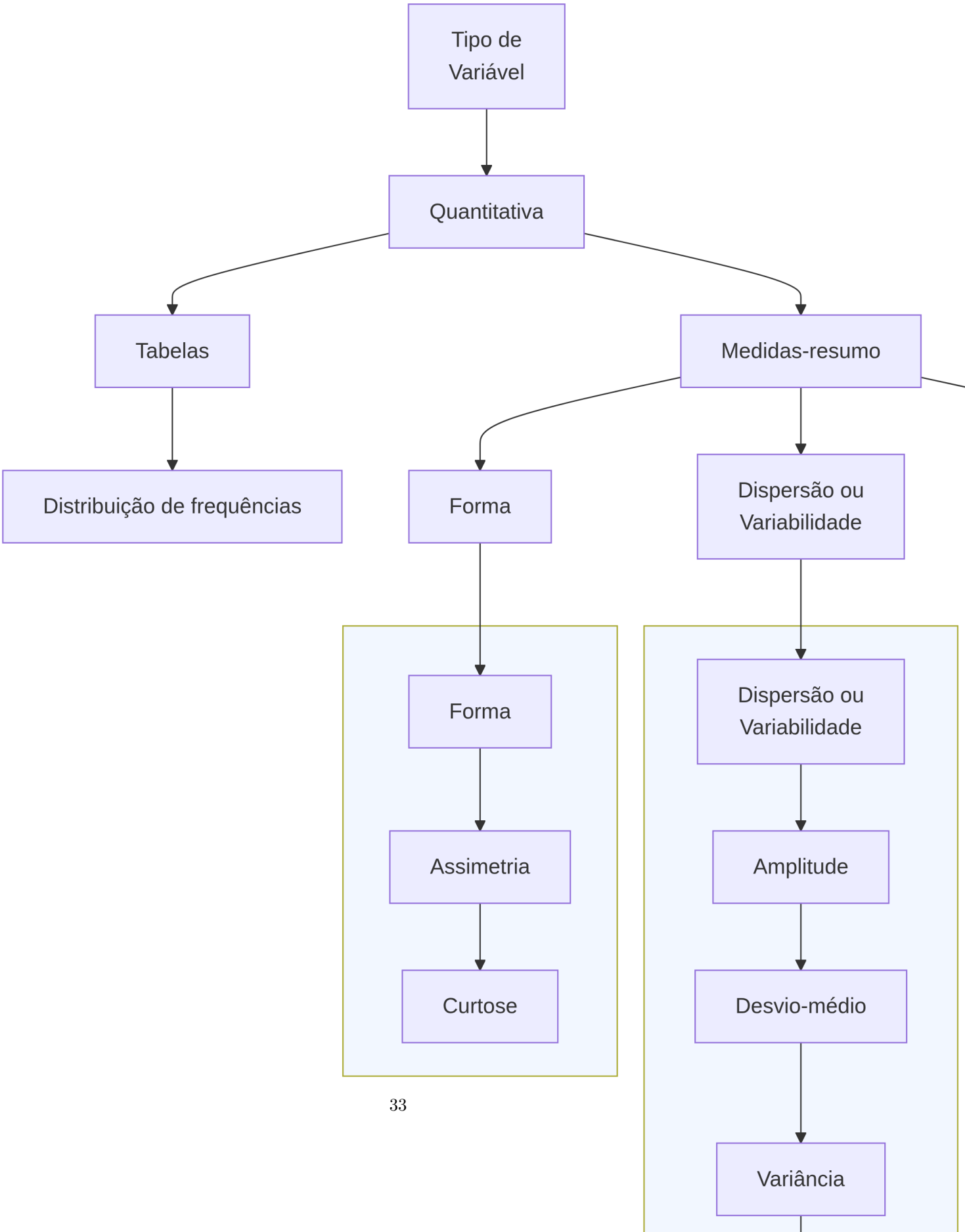
Estatística descritiva

Estatísticas descritivas fazem parte de todo trabalho de análise de dados. Podemos ver ela como um passo inicial, em que você utiliza técnicas de análise exploratória para entender melhor o comportamento dos seus dados considerando o todo, por meio de tabelas, gráficos e medidas-resumo. Na estatística descritiva univariada buscamos o comportamento de uma variável isolada das demais do conjunto de dados, enquanto nas estatísticas bivariada ou multivariada exploramos o comportamento em conjunto de duas ou mais variáveis.

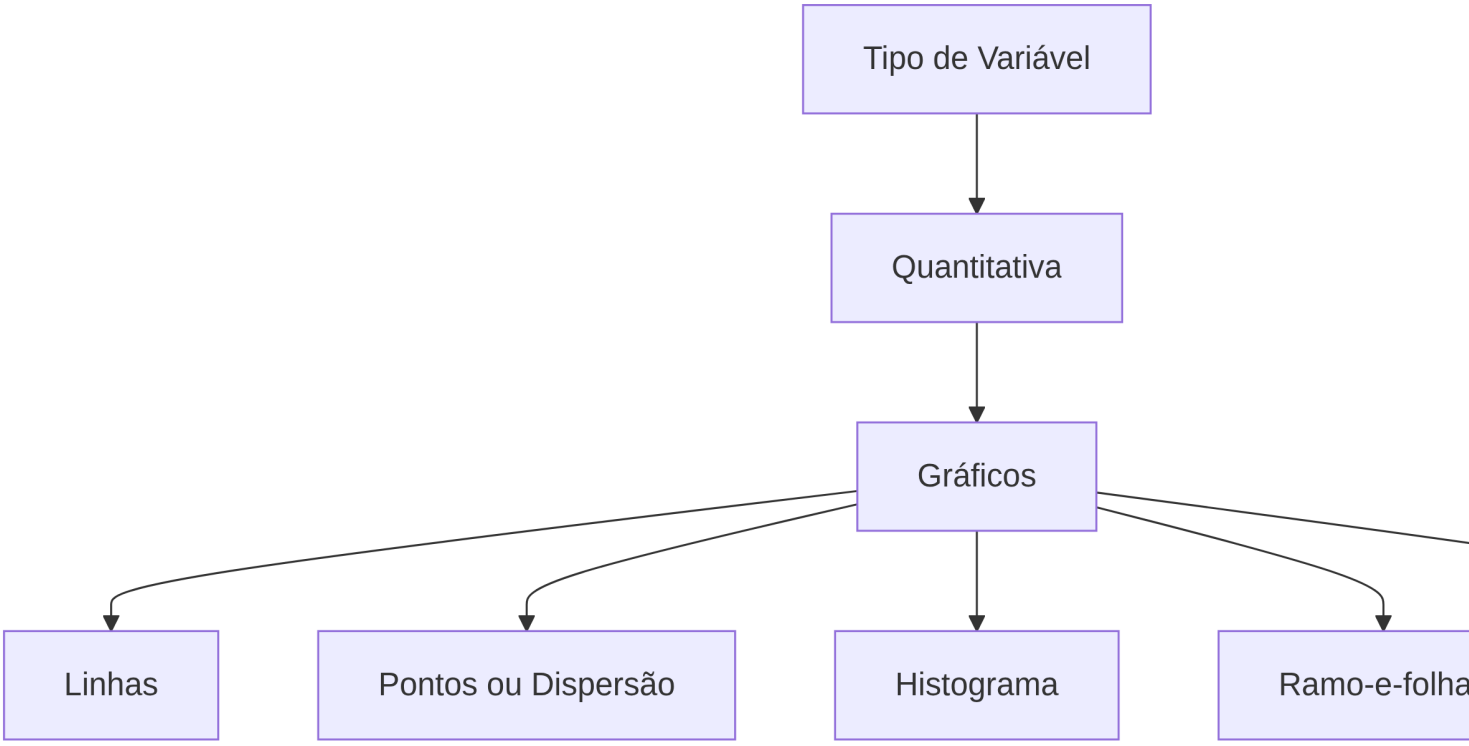
O tipo de variável em estudo irá influenciar na técnica que você escolhe para analisá-la, assim, podemos sintetizar as possibilidades de análise da seguinte forma para estatísticas univariadas:



Estatísticas descritivas univariadas para variáveis qualitativas. Fonte: adaptado de Favaro *et al.* (2017)


Tipo de Variável
Quantitativa
Tabelas
Distribuição de frequências
Medidas-resumo
Forma
Dispersão ou Variabilidade
Forma
Assimetria
Curtose
Dispersão ou Variabilidade
Amplitude
Desvio-médio
Variância

Estatísticas descritivas univariadas para variáveis quantitativas (tabelas e medidas-resumo). Fonte: adaptado de Favaro *et al.* (2017)



Estatísticas descritivas univariadas para variáveis quantitativas (gráficos). Fonte: adaptado de Favaro *et al.* (2017)

3.1 Tabela de distribuição de frequências

Tabelas de frequências representam o número absoluto ou relativo de ocorrências de uma categoria (variáveis qualitativas), valor (variáveis quantitativas discretas) ou intervalo de valores (variáveis quantitativas contínuas).

Assim, a tabela será composta dos possíveis valores:

- Frequência absoluta (F_i): contagem das observações na classe i;
- Frequência relativa (Fr_i): contagem das observações na classe i em relação ao total de observações da variável;
- Frequência acumulada (F_{ac_i}): soma das observações na classe i e nas anteriores a ela (só faz sentido se há noção de ordem nos dados);
- Frequência acumulada relativa (Fr_{ac_i}): soma das frequências relativas até a classe i (inclusive);

No caso de variáveis contínuas, para criarmos tabelas de frequências delas, primeiro criamos classes representado intervalos de valores, depois contamos o número de observações em cada intervalo. O número de classes e o intervalo de cada classe é arbitrário, entretanto, Bussab e Morettin (2011) sugerem o seguinte algoritmo para construção da tabela:

1. Ordenar os dados do menor ao maior;
2. Calcular o número de classes (k) por

- $k = 1 + 1,33 \times \log_{10} n$ (Equação de Sturges)
- $k = \sqrt{n}$
- n é o número de observações e k deve ser arredondado para o inteiro mais próximo;

3. Calcular o intervalo das classes, Δ_h, como $\Delta_h = \frac{A}{k}$, em que A é a amplitude (máximo valor menos o mínimo);
4. Construa os intervalos iniciando pelo menor valor como primeiro limite inferior e somando Δ_h ao limite inferior de cada classe para obter os limites superiores. Cada intervalo, exceto o primeiro, será aberto (exclui) no limite inferior e fechado (inclui) no superior, e o limite inferior das classes após a primeira serão o limite superior da anterior (ex.: {[1, 11],]11, 21],]21, 31],]31, 41]});
5. Conte o número de observações que possuem valores dentro de cada intervalo para construir a tabela de frequências.

No R, as tabelas de frequências podem ser obtidas da seguinte forma:

```
ap2 <- read_dta(
  "../datasets/stata/ap2.dta",
  encoding = "UTF-8"
  ) %>%
  as_factor() %>%
  mutate(
    vacc_mp = fct_recode(
      vacc_mp,
      "Vacinado" = "vac",
      "Não vacinado" = "not vac."
    )
  )

# Qualitativa
tab_frequencias_qualitativa <- ap2 %>%
  count(vacc_mp, name = "Fi") %>%
  mutate(
    Fri = scales::percent(Fi / sum(Fi), accuracy = .1),
    Fac = cumsum(Fi),
    Frac = scales::percent(cumsum(Fi / sum(Fi)), accuracy = .1)
  )

# Quantitativa discreta
tab_frequencias_discreta <- ap2 %>%
  count(parity, name = "Fi") %>%
  mutate(
    Fri = scales::percent(Fi / sum(Fi), accuracy = .1),
    Fac = cumsum(Fi),
    Frac = scales::percent(cumsum(Fi / sum(Fi)), accuracy = .1)
  )

# Quantitativa contínua
classes_k <- round(1 + 3.322 * log10(length(ap2$w_age_t)))
breaks <- floor(seq(min(ap2$w_age_t), max(ap2$w_age_t), length.out = classes_k + 1))
breaks[length(breaks)] <- ceiling(max(ap2$w_age_t))

tab_frequencias_continua <- ap2 %>%
```

```r
  mutate(
    w_age_t_classificado = cut(w_age_t, breaks = breaks, right = TRUE, include.lowest = TRUE)
  ) %>%
  count(w_age_t_classificado, name = "Fi") %>%
  mutate(
    Fri = scales::percent(Fi / sum(Fi), accuracy = .1),
    Fac = cumsum(Fi),
    Frac = scales::percent(cumsum(Fi / sum(Fi)), accuracy = .1)
  )
```

```r
tabela_headers <- c("Classe", "$F_i$", "$Fr_i$", "$F_{ac_i}$", "$Fr_{ac_i}$")

kable(tab_frequencias_qualitativa, col.names = tabela_headers)
```

Tabela 3.1: Distribuição de frequências de uma variável qualitativa

Classe	F_i	Fr_i	F_{ac_i}	Fr_{ac_i}
Não vacinado	419	37.6%	419	37.6%
Vacinado	695	62.4%	1114	100.0%

```r
kable(tab_frequencias_discreta, col.names = tabela_headers)
```

Tabela 3.2: Distribuição de frequências de uma variável quantitativa discreta

Classe	F_i	Fr_i	F_{ac_i}	Fr_{ac_i}
1	199	17.9%	199	17.9%
2	194	17.4%	393	35.3%
3	158	14.2%	551	49.5%
4	157	14.1%	708	63.6%
5	162	14.5%	870	78.1%
6	127	11.4%	997	89.5%
7	50	4.5%	1047	94.0%
8	36	3.2%	1083	97.2%
9	16	1.4%	1099	98.7%
10	9	0.8%	1108	99.5%
11	6	0.5%	1114	100.0%

```r
kable(tab_frequencias_continua, col.names = tabela_headers)
```

Tabela 3.3: Distribuição de frequências de uma variável quantitativa contínua

Classe	F_i	Fr_i	F_{ac_i}	Fr_{ac_i}
[12,15]	8	0.7%	8	0.7%
(15,18]	62	5.6%	70	6.3%
(18,21]	125	11.2%	195	17.5%
(21,24]	186	16.7%	381	34.2%
(24,28]	244	21.9%	625	56.1%

Classe	F_i	Fr_i	F_{ac_i}	Fr_{ac_i}
(28,31]	157	14.1%	782	70.2%
(31,34]	132	11.8%	914	82.0%
(34,37]	99	8.9%	1013	90.9%
(37,40]	68	6.1%	1081	97.0%
(40,44]	28	2.5%	1109	99.6%
(44,48]	5	0.4%	1114	100.0%

3.2 Representação gráfica dos dados

Os gráficos hoje são insdispensáveis na estatística e análise de dados, servindo como uma ponte entre os dados brutos e seu comportamento em conjunto. Normalmente os usamos para detectar padrões, tendências, anomalias e relações que não seriam percebidos em tabelas ou medidas resumo.

Na análise exploratória de dados (EDA), histogramas, boxplots e gráficos de dispersão permitem identificar a forma da distribuição, detectar outliers e avaliar a simetria. Também usamos gráficos para auxiliar na verificação de suposições de modelos estatísticos, por exemplo, quando criamos um gráfico Q-Q para identificar desvios da normalidade.

Além disso, os gráficos são ferramentas para a comunicação estatística, os quais usamos para traduzir conceitos complexos em representações visuais mais intuitivas e didáticas que facilitem a compreensão por um público que não necessariamente possui um conhecimento formal de estatística, mas que podem ser o alvo dos nossos resultados, por exemplo, um gestor público ou de uma empresa cujas nossas análises poderiam (ou deveriam) trazer uma informações relevantes para aumentar a eficácia de sua administração.

Em contextos como epidemiologia veterinária, um gráfico de série temporal pode sugerir surtos epidêmicos ou sazonalidade de doenças, mapas de calor espaciais podem revelar clusters de casos, indicando os locais para intensificar as ações de defesa sanitária.

Em nossos treinamentos, a maioria dos gráficos serão construídos usando o pacote `ggplot2` do `tidyverse`. Existem muitas fontes de informação para o aprendizado do `ggplot2` e da criação de gráficos no R, porém, deixo aqui duas recomendações que considero como um "guia de bolso" para criação de gráficos no R, que são o livro *R Graphics Cookbook* do Winston Chang (2025), um livro online de acesso aberto com várias explicações sobre a criação de gráficos com o `ggplot2`, e o site The R Graph Gallery onde você encontra explicações e exemplos de código para a criação dos mais variados gráficos no R. Agora, se você quer um conteúdo avançado sobre o `ggplot2`, o livro online ggplot2: Elegant Graphics for Data Analysis é o que você procura.

3.2.1 Criação de gráficos no `ggplot2`

O ggplot2 implementa os conceitos da *Grammar of Graphics* de Leland Wilkinson, onde gráficos são construídos através de **camadas semânticas**, as quais são combinadas de forma modular e sistemática. Seus principais aspectos são:

- **Decomposição**: Todo gráfico pode ser decomposto em componentes independentes;
- **Composição**: Componentes são combinados usando o operador `+`;
- **Declarativo**: Você descreve O **QUE** quer, não **COMO** desenhar.

Todo gráfico criado com o `ggplot2` vai depender da seguinte estrutura mínima:

```
ggplot(
  data = <DATA>,            # 1. Dados (fonte da informação para o gráfico)
```

```
  mapping = aes(<MAPPINGS>) # 2. Mapeamentos estéticos (quais variáveis serão usadas e onde)
  ) +
  <GEOM_FUNCTION>()         # 3. Geometria (o que será desenhado)
```

Assim, `ggplot()` inicia a construção do gráfico, indicando às camadas posteriores de construção qual o dataset fonte das variáveis (argumento `data`) e, em geral, como serão mapeadas as variáveis para o gráfico (quem será o eixo das ordenadas? e das abscissas? exitem variáveis categóricas que gostaríamos de usar como fonte para as cores do que será apresentado, ou seja, criar subconjuntos?). Iniciado o `ggplot()`, que podemos pensar como um quadro em branco, passamos a de fato desenhar nosso gráfico com `geom_*()` e outras camadas de estilização.

Além desses componentes primários, outros que costumam ser usados na construção de gráficos são:

- **Facets**: Divisão em subgráficos
- **Statistics (stat)**: Transformações estatísticas
- **Coordinates (coord)**: Sistema de coordenadas
- **Themes**: Aparência não relacionada aos dados
- **Scales**: Controle de mapeamentos

Vamos então observar na prática como é construção do gráfico no `ggplot2`. Iniciamos com `ggplot(data=dados)`, o que gera somente o nosso painel, sem qualquer escala ainda.

```r
set.seed(42)

dados <- data.frame(
  x = rnorm(100, mean = 10, sd = 2),
  y = rnorm(100, mean = 15, sd = 3),
  grupo = sample(c("A", "B", "C"), 100, replace = TRUE),
  tamanho = runif(100, 1, 10)
)

p_base <- ggplot(dados)
p_base  # Produz apenas o painel vazio
```

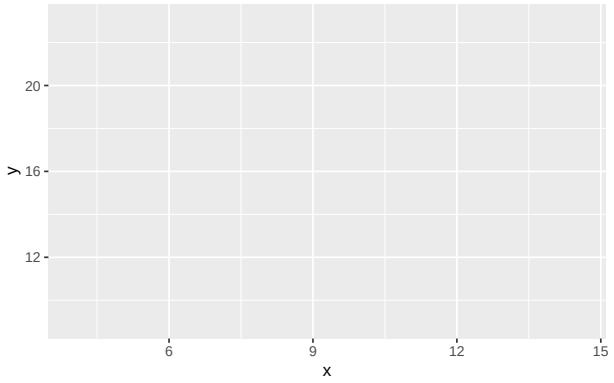
Figura 3.1: Chamada de `ggplot` somente com argumento data

O `aes()` pode ser informado no próprio `ggplot`, caso esse mapeamento vá ser usado da mesma forma por todas as demais camadas, ou pode ser informado a cada `geom_*()`, caso seja algo específico daquela geometria. Os principais elementos estéticos definidos no `aes()` são: `x` (eixo das abscissas), `y` (eixo das ordenadas), `color` (cor de delimitação), `fill` (cor de preenchimento), `size` (tamanho), `alpha` (transparência), `shape` (forma), `linetype` (tipo de linha).

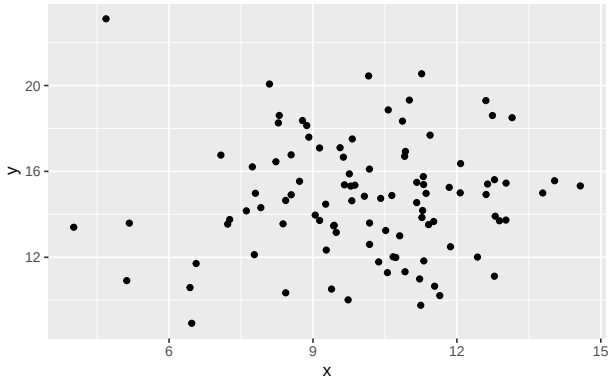
```
p_base <- ggplot(dados, aes(x = x, y = y))

# Mapeamento geral
p1 <- ggplot(dados, aes(x = x, y = y)) +
  geom_point()

# Mapeamento local (na geometria)
p2 <- ggplot(dados) +
  geom_point(aes(x = x, y = y))

# Mapeamentos múltiplos
p3 <- ggplot(dados, aes(x = x, y = y)) +
  geom_point(aes(color = grupo, size = tamanho), alpha = 0.6)  # alpha fixo (não mapeado)
```

```
p_base
p1
p2
p3
```

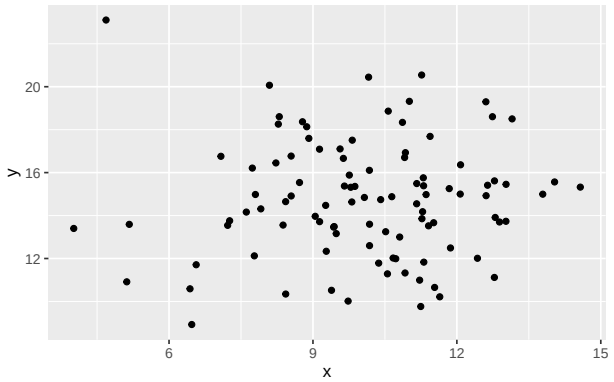
Como já falado, o `ggplot2` segue um sistema de camadas, então cada operação após o `+` adiciona uma nova feição sobre o gráfico e pode inclusive sobrescrever geometrias definidas em operações anteriores, então é



(a) Chamada de `ggplot(dados, aes(x = x, y = y))` sem uma geometria



(a) Chamada de `ggplot(dados, aes(x = x, y = y))` com uma geometria de pontos



(a) `aes()` definidos na geometria

(a) `aes()` em diferentes camadas

Uso do `aes()`

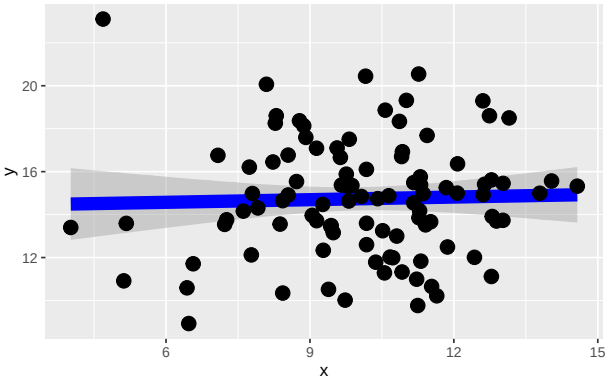
importante verificar a ordem em que são declarados os aspectos visuais do gráfico. Observe abaixo, como alterar a ordem de declaração das geometrias "esconde" certos pontos no gráfico da esquerda em relação ao da direita.

```
p_ordem1 <- ggplot(dados, aes(x, y)) +
  geom_smooth(method = "lm", se = TRUE, color = "blue", linewidth = 4) +
  geom_point(size = 4, color = "black")  # Pontos sobre a linha

p_ordem2 <- ggplot(dados, aes(x, y)) +
  geom_point(size = 4, color = "black") +
  geom_smooth(method = "lm", se = TRUE, color = "blue", linewidth = 4)  # Linha sobre pontos

p_ordem1
```

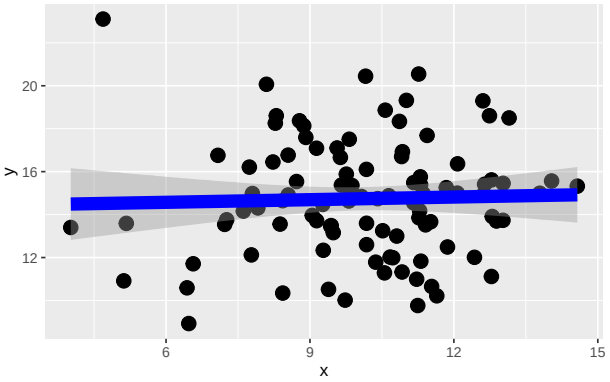
```
`geom_smooth()` using formula = 'y ~ x'
```

```
p_ordem2
```

```
`geom_smooth()` using formula = 'y ~ x'
```



(a) Chamada de `ggplot() + geom_smooth() + geom_point()`



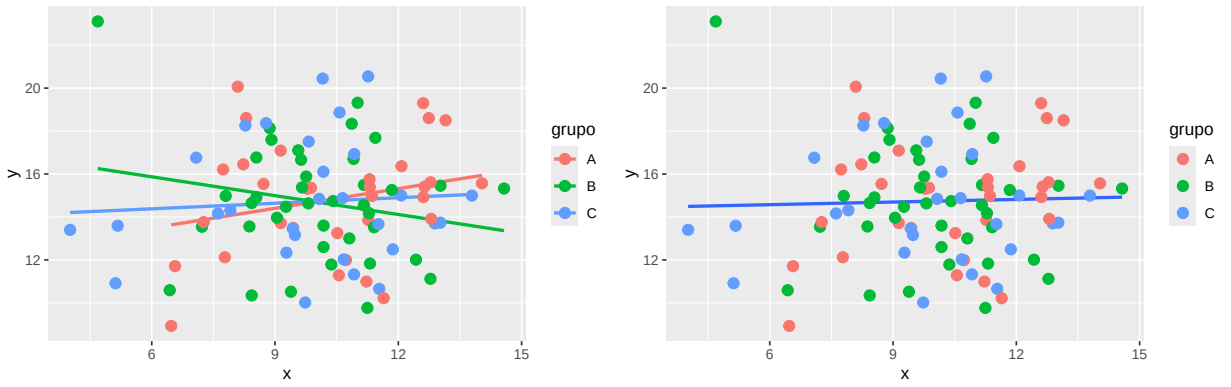
(a) Chamada de `ggplot() + geom_point() + geom_smooth()`

Importância de definir a ordem correta das geometrias no seu gráfico.

Quando definimos um `aes()` na chamada de `ggplot()`, ele será herdado pelas demais camadas, porém, ao usarmos ele em uma geometria, ele só será usado nela.

```
p_heranca <- ggplot(dados, aes(x, y, color = grupo)) +  # color global
  geom_smooth(method = "lm", se = FALSE) +
  geom_point(size = 3)

p_sem_heranca <- ggplot(dados, aes(x, y)) +
  geom_smooth(method = "lm", se = FALSE) +
  geom_point(size = 3, aes(color = grupo))

p_heranca
p_sem_heranca
```

Quando queremos criar múltiplos gráficos da mesma geometria, segundo uma variável alvo, podemos usar



(a) `ggplot() + geom_smooth() + geom_point()` com herança de `aes()` do `ggplot()`

(a) Chamada de `ggplot() + geom_point() + geom_smooth()` com `aes()` específico para `geom_point()`

Herança de `aes()` no `ggplot2`.

o `facet_*()` para criação de vários subpainéis.

Com `facet_wrap(~ variavel_alvo)` os painéis são dispostos horizontalmente considerando o número de categorias. Por padrão, os painéis serão dispostos em uma tabela quadrada, mas essa disposição pode ser alterada usando os argumentos `nrow` para o número de linhas e `ncol` para o número de colunas.

```
p_wrap <- ggplot(dados, aes(x, y)) +
  geom_point(color = "steelblue") +
  geom_smooth(method = "lm", se = FALSE, color = "red") +
  facet_wrap(~ grupo, ncol = 2)

p_wrap
```

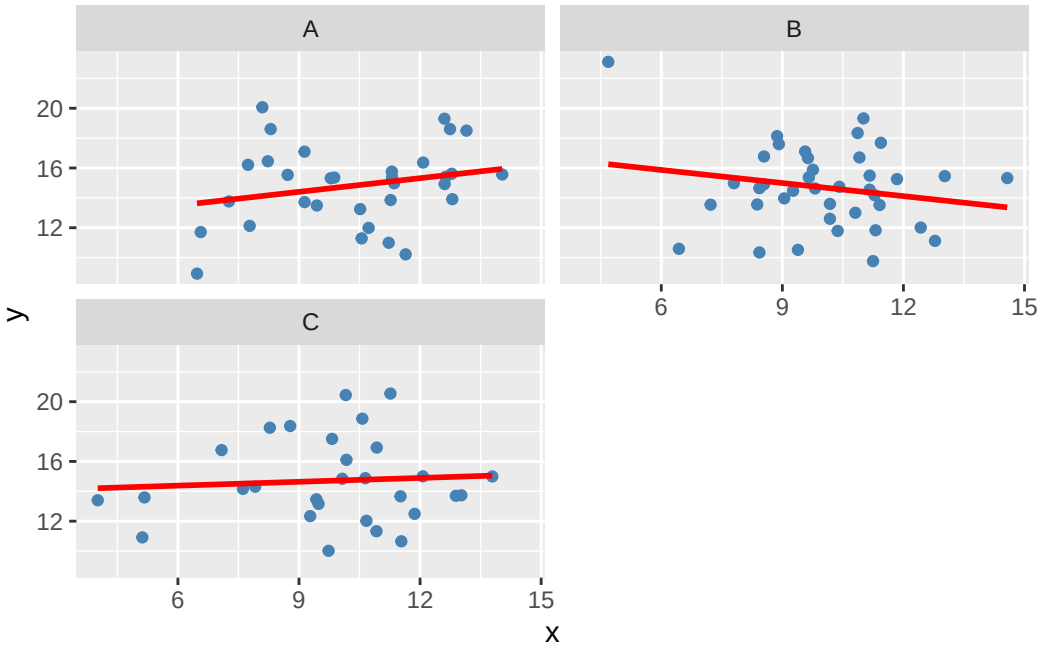


Figura 3.10: Múltiplos painéis no `ggplot2` com `facet_wrap`.

Observe na figura acima que o nome das categorias são exibidas como *headers* de cada painel, assim, se queremos alterar o título da categoria de cada painel, seria necessário alterar a própria variável (usando por exemplo o `recode()` se for um dado do tipo `factor`). O argumento `labeller = label_both` faz com que o título do painel também apresente o nome da variável. Aspectos visuais do título serão alterados na camada de `theme()` do gráfico no `ggplot2`, a qual será apresentada mais adiante no treinamento, mas podemos alterar tanto características do texto (`strip.text`) quanto do fundo do título (`strip.background`).

```
p_wrap <- dados %>%
  mutate(rcd_grupo = recode(grupo, "A" = "Tratamento 1", "B" = "Tratamento 2", "C" = "Tratamento 3")) %>%
  ggplot(aes(x, y)) +
  geom_point(color = "steelblue") +
  geom_smooth(method = "lm", se = FALSE, color = "red") +
  facet_wrap(~ rcd_grupo, labeller = label_both) +
  theme(
    strip.text = element_text(face = "italic", size = rel(1.2)),
    strip.background = element_rect(fill = "#7688d6bb", colour = "black", linewidth = .7)
  )

p_wrap
```

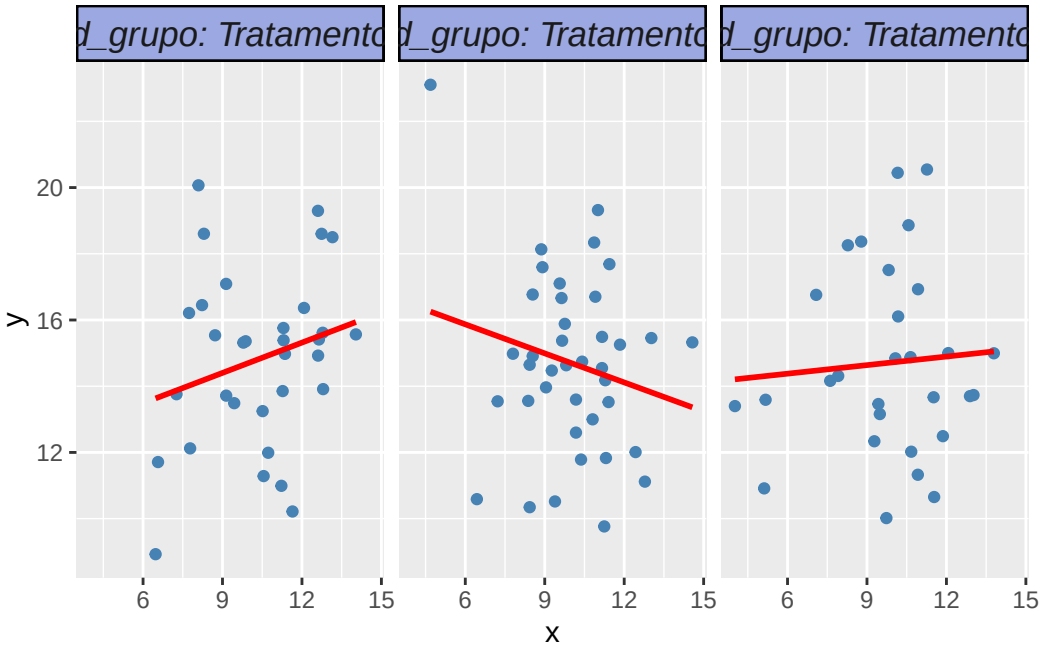


Figura 3.11: Alterando os *headers* de painéis do `facet_*()`.

Já o `facet_grid(variavel_alvo_linha ~ variavel_alvo_coluna)` permite criar painéis a partir de subconjuntos de uma variável para as colunas e outra para as linhas do gráfico. As mesmas técnicas de estilização do `facet_wrap` se aplicam ao `facet_grid`.

```
# facet_grid: duas variáveis
dados$categoria <- sample(c("Categoria 1", "Categoria 2"), 100, replace = TRUE)

p_grid <- ggplot(dados, aes(x, y)) +
  geom_point(aes(color = tamanho)) +
  facet_grid(categoria ~ grupo) +
  theme(
    strip.text = element_text(face = "italic", size = rel(1.2)),
    strip.background = element_rect(fill = "#7688d6bb", colour = "black", linewidth = .7)
  )

p_grid
```

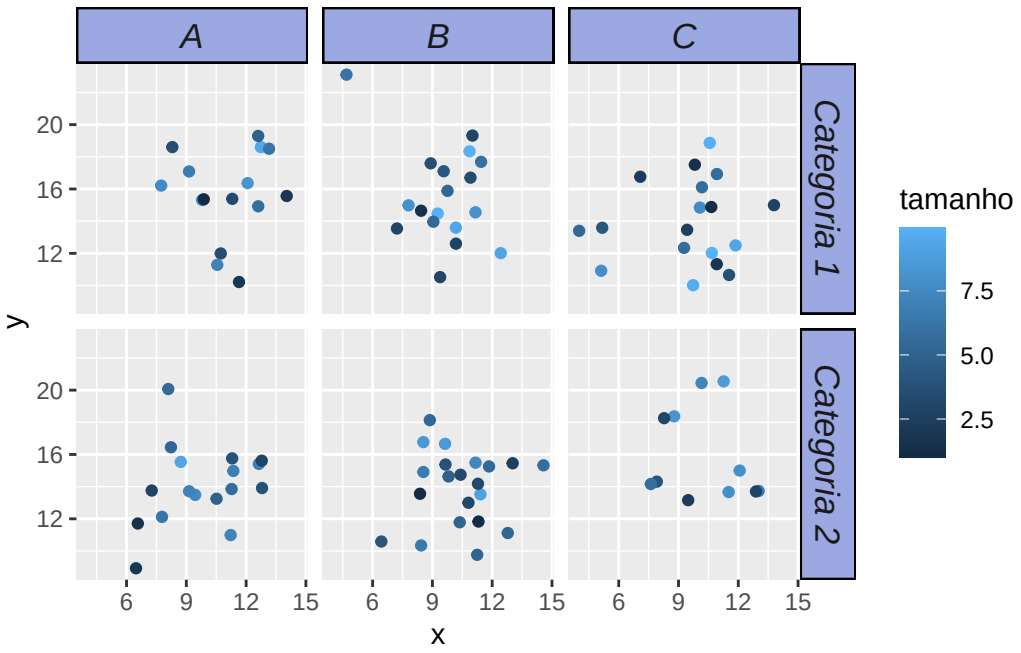


Figura 3.12: Múltiplos painéis no `ggplot2` com `facet_grid`.

Para controlar a escala das estéticas mapeadas no gráfico do `ggplot2` usamos as funções da família `scale_*`. `scale_*` segue um padrão de nome onde primeiro declaramos qual estética queremos controlar (x, y, size...) e em seguida a forma ou transformação que consideramos para a variável mapeada.

Para escalas de posição (eixos x e y), usamos `scale_*_continuous` para controlar a escala que varia em um intervalo contínuo e `scale_*_discrete` se queremos que cada valor único da variável mapeada seja tratada como uma categoria.

Com `scale_*_continuous` podemos controlar os limites do eixo alvo com `limits = c(minimo, maximo)` e suas marcações com `breaks` e `minor_breaks`. `breaks` e `minor_breaks` recebe um vetor das posições das marcações ou uma função que, a partir dos limites do eixo, calculará as posições das marcações. Além disso, podemos usar o argumento `transform` para transformações dos dados naquela escala, por exemplo, aplicando a raiz quadrada ou o log, e podemos usar `labels` para alterar o texto exibido em cada marcação do eixo.

```
p_scales <- ggplot(dados, aes(x, y, color = tamanho)) +
  geom_point(size = 3) +
  scale_x_continuous(
    name = "X",
    limits = c(5, 15),
    breaks = seq(5, 15, length.out = 3),
    labels = c("pequeno", "grande", "muito grande")
  ) +
  scale_y_continuous(
    name = "sqrt(Y)",
    transform = "sqrt"
  )
```

```
p_scales
```

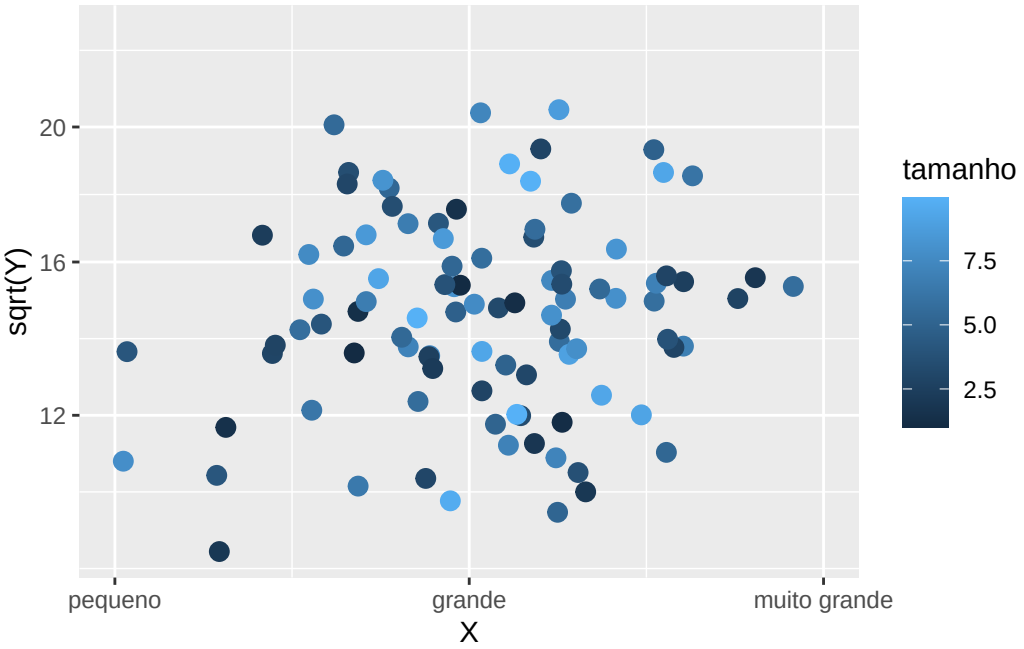


Figura 3.13: Manipulando escalas no `ggplot2` com `scale_*_continuous`.

Com `scale_*_discrete` podemos também definir o nome do eixo, o texto das marcações e outras características como no `scale_*_continuous`.

```
ggplot(dados, aes(x = grupo)) +
  geom_bar(aes(y = after_stat(prop), group = 1), fill = "#ac565693", color= "black") +
  scale_y_continuous(
    name = "Frequência relativa (%)",
    labels = scales::label_percent(),
    limits = c(.0, .65),
    breaks = scales::breaks_width(width = .05),
    minor_breaks =  scales::breaks_width(width = .025),
  ) +
  scale_x_discrete(
    name = "Grupos",
    labels = c("Tratamento 1", "Tratamento 2", "Controle")
  )
```

Figura 3.14: Manipulando escalas no `ggplot2` com `scale_*_discrete`.

Em ambos, `limits` define o próprio limite para os dados mapeados, então, devemos tomar cuidado, principalmente em gráficos que calculam estatísticas para sua construção, já que dados fora desses limites serão desconsiderados

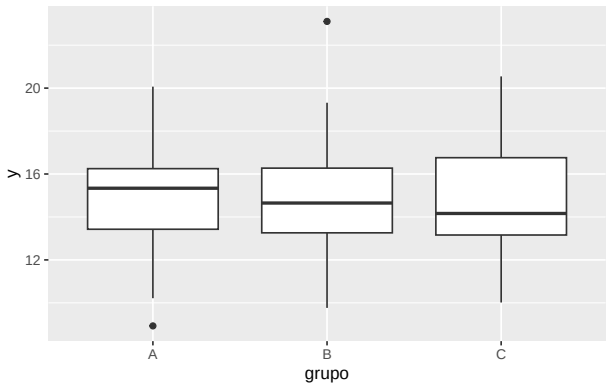
```
p_cat <- ggplot(dados, aes(grupo, y)) +
  geom_boxplot()

p_cat_scales <- ggplot(dados, aes(grupo, y)) +
  geom_boxplot() +
  scale_y_continuous(limits = c(NA, 20)) +
  scale_x_discrete(limits = c("A", "B"))

p_cat
p_cat_scales
```

```
Warning: Removed 29 rows containing missing values or values outside the scale range
(`stat_boxplot()`).
```

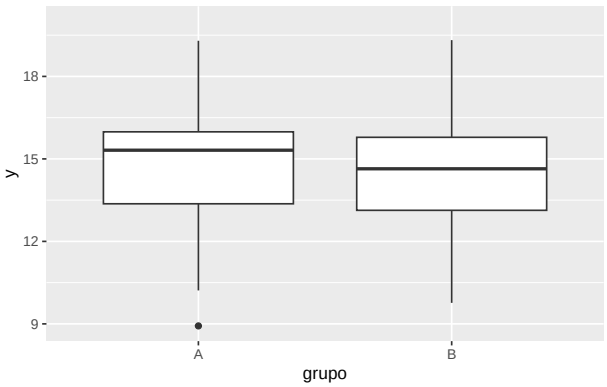
```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_boxplot()`).
```

Se queremos limitar nossa escala no gráfico, sem alterar esse mapeamento e, consequentemente, haver perda de dados, podemos usar `coord_cartesian` e definir nossos limites para as ordenadas com `ylim` e as abscissas com `xlim`.

```
p_cat <- ggplot(dados, aes(grupo, y)) +
  geom_boxplot() +
  coord_cartesian(ylim = c(11, 20), xlim = c(1, 2))
```



(a) Original sem perda



(a) Usando limites nas ordenadas e nas abscissas

Perda de dados ao usar `limits` no `scale_*_`.

```
p_cat
```

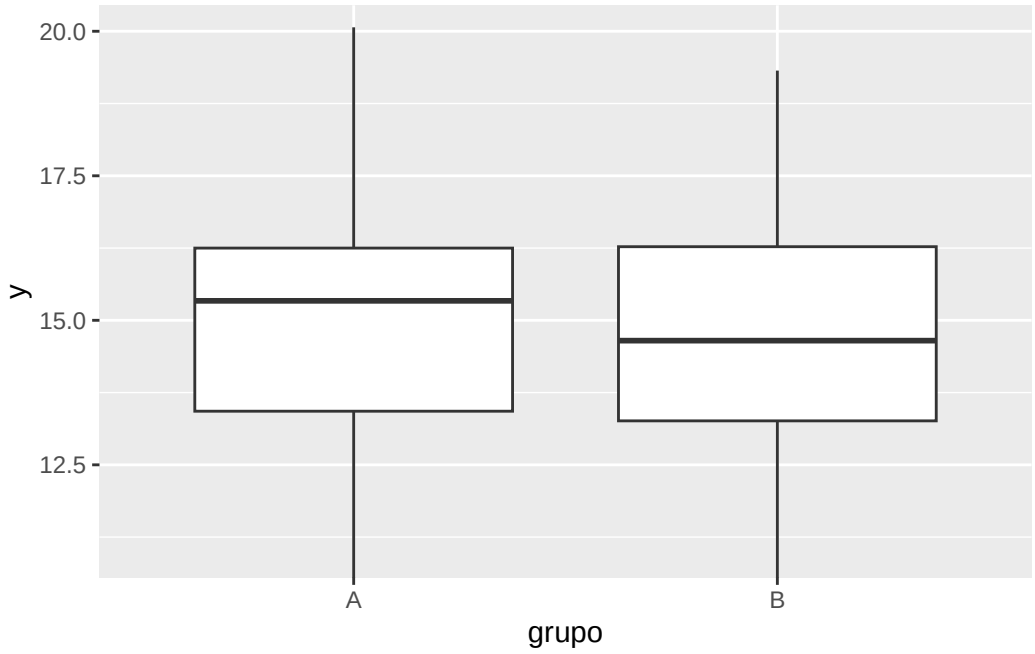


Figura 3.17: Alterando as escalas dos eixos com `coord_cartesian`.

Para alterar escalas de cores e preenchimentos usamos `scale_color_*` e `scale_fill_*`. Para criar um gradiente contínuo podemos usar `scale_*_gradient`, o qual aceita os parâmetros `low` e `high` como os valores das cores a usar para criar seu gradiente de cores. Então, se usados ambos, será criado uma escala com um gradiente de cores variando entre a cor em `low` (menor valor dos dados e maior intensidade da cor declara em `low`) e a cor em `high` (maior valor dos dados e maior intensidade da cor declara em `high`). Se declarado somente `low`, teremos um gradiente criado a partir de uma única cor, com a maior intensidade

no menor valor.

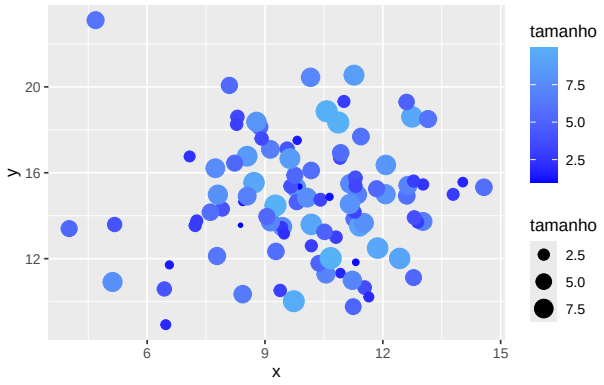
Para criar uma escala de cor divergente (gradiente com três cores, com um delas indicando um ponto central), podemos usar `scale_*_gradient2`. Além disso, podemos criar um gradiente com n cores com `scale_*_gradientn` (nesse caso, deve ser fornecido um vetor dos nomes de cores a usar no gradiente).

```
ggplot(dados, aes(x, y, color = tamanho, size = tamanho)) +
  geom_point() +
  scale_color_gradient(low = "blue")

ggplot(dados, aes(x, y, color = tamanho, size = tamanho)) +
  geom_point() +
  scale_color_gradient(low = "blue", high = "red")

ggplot(dados, aes(x, y, color = tamanho, size = tamanho)) +
  geom_point() +
  scale_color_gradient2(
    low = "blue",
    mid = "white",
    high = "red",
    midpoint = mean(dados$tamanho)
  )

ggplot(dados, aes(x, y, color = tamanho, size = tamanho)) +
  geom_point() +
  scale_color_gradientn(colors = c("blue", "red", "grey", "green"))
```

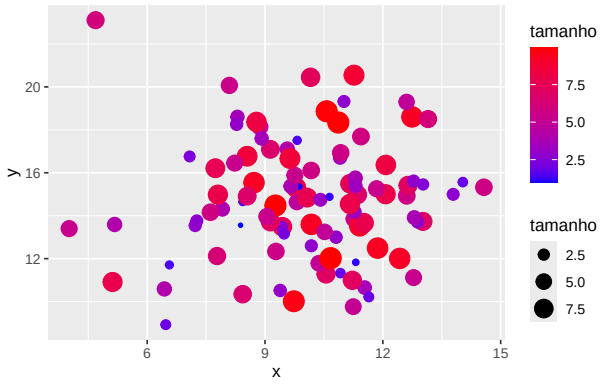
Para escalas de cores discretas, por exemplo, para diferenciar categorias, podemos usar `scale_*_manual` com o vetor de cores de cada categoria.

Para escalas de tamanho (`size`) podemos alterar o comportamento com `scale_radius()` se queremos uma escala contínua de variação, ou `scale_size_binned()` se queremos criar quebras.

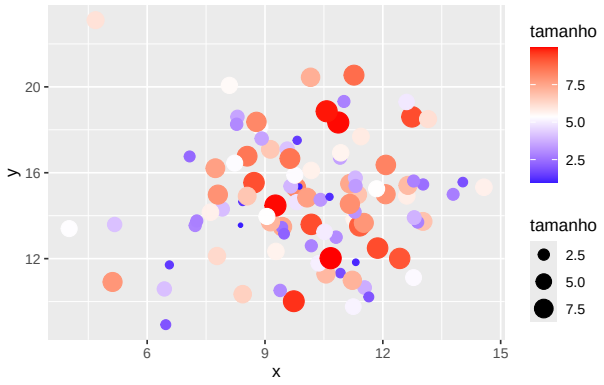
```
ggplot(dados, aes(x, y, color = grupo, size = tamanho)) +
  geom_point() +
  scale_color_manual(values = c("red", "blue", "black")) +
  scale_radius(limits = c(0, NA))

ggplot(dados, aes(x, y, color = grupo, size = tamanho)) +
  geom_point() +
  scale_color_manual(values = c("red", "blue", "black")) +
  scale_size_binned() +
  guides(
    size = guide_bins(
      show.limits = TRUE,
      title = "Tamanho",
      axis.colour = "blue",
      axis.arrow = arrow(
        length = unit(.05, "inches"),
        ends = "first",
        type = "closed"
      )
    )
```



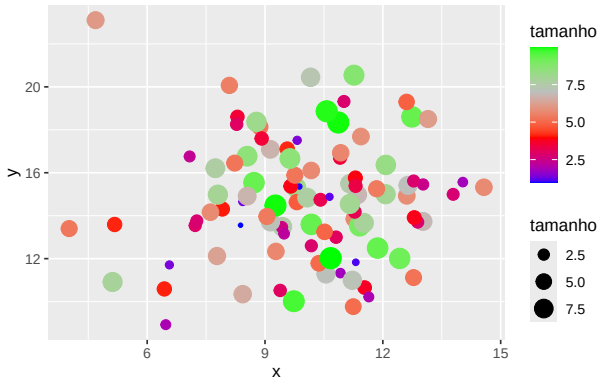
(a) `scale_color_gradient` definindo somente `low`



(a) `scale_color_gradient` definindo `low` e `high`



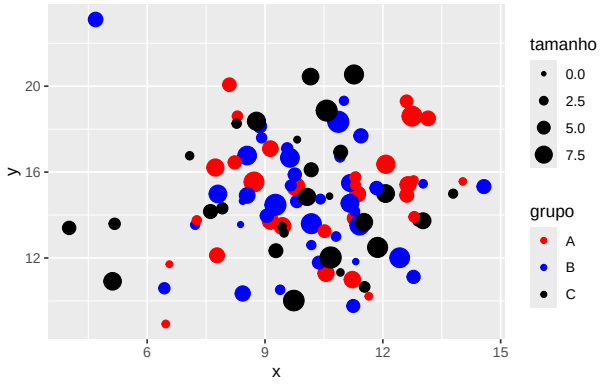
(a) `scale_color_gradient2`



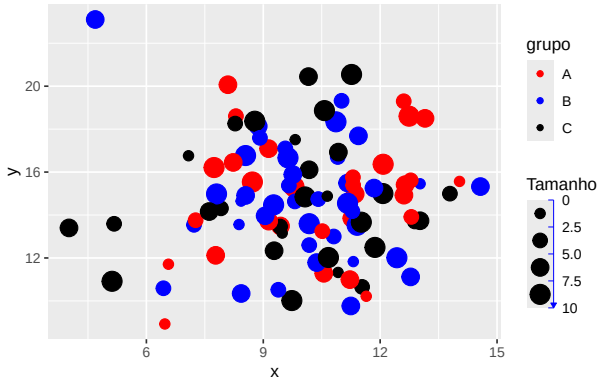
(a) `scale_color_gradientn` com 4 cores

Alterando as escalas de cores com `scale_*_gradient`.

```
  )
```



(a) `scale_radius`



(a) `scale_size_binned`

Alterando as escalas de cores discretas com `scale_*_discrete` e tamanhos com `scale_radius` e `scale_size_binned`.

Embora seja possível definir manualmente as cores que queremos usar nas escalas, a escolha de cores interfere em muito em como o público conseguirá identificar padrões no gráfico. Por isso, recomendo usar funções que já fornecem paletas de cores criadas com uma finalidade específica como as do ColorBrewer com `scale_*_brewer` ou do pacote `paletteer`. ColorBrewer possui um site onde é possível testar a paleta de cores que queremos usar. Usando o `scale_*_brewer`, definimos o tipo de paleta (seq" para sequencial, "div" para divergente ou "qual" para qualitativo) e o nome da paleta.

```
ggplot(dados, aes(grupo, fill = grupo)) +
  geom_bar() +
  scale_fill_brewer(type = "qual")

ggplot(dados, aes(grupo, fill = grupo)) +
  geom_bar() +
  scale_fill_brewer(palette = "Pastel1")
```

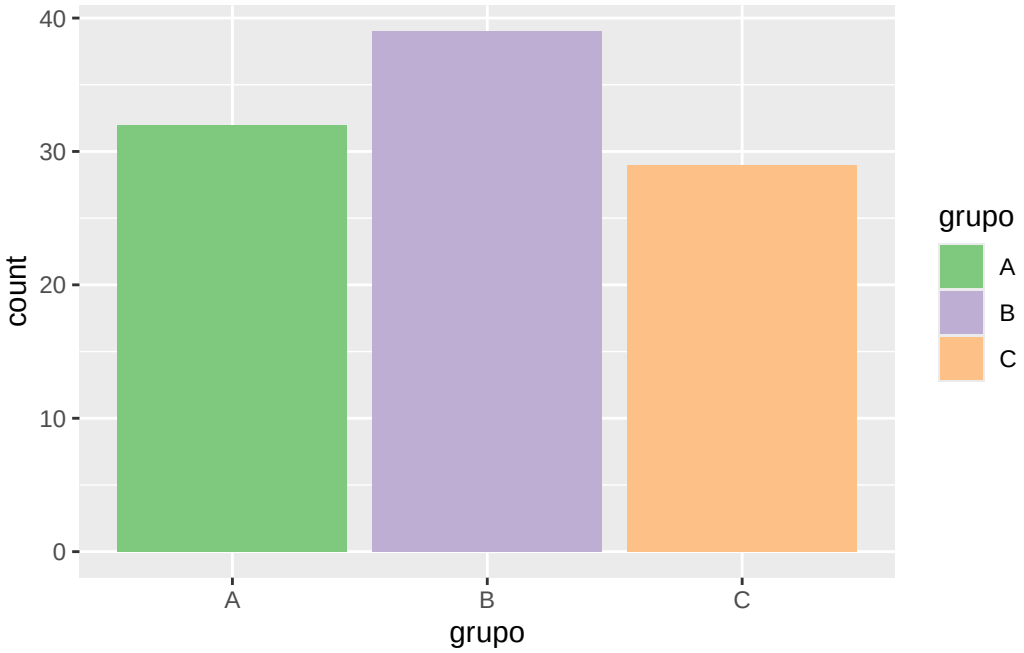


Figura 3.24: `scale_fill_brewer(type = "qual")`

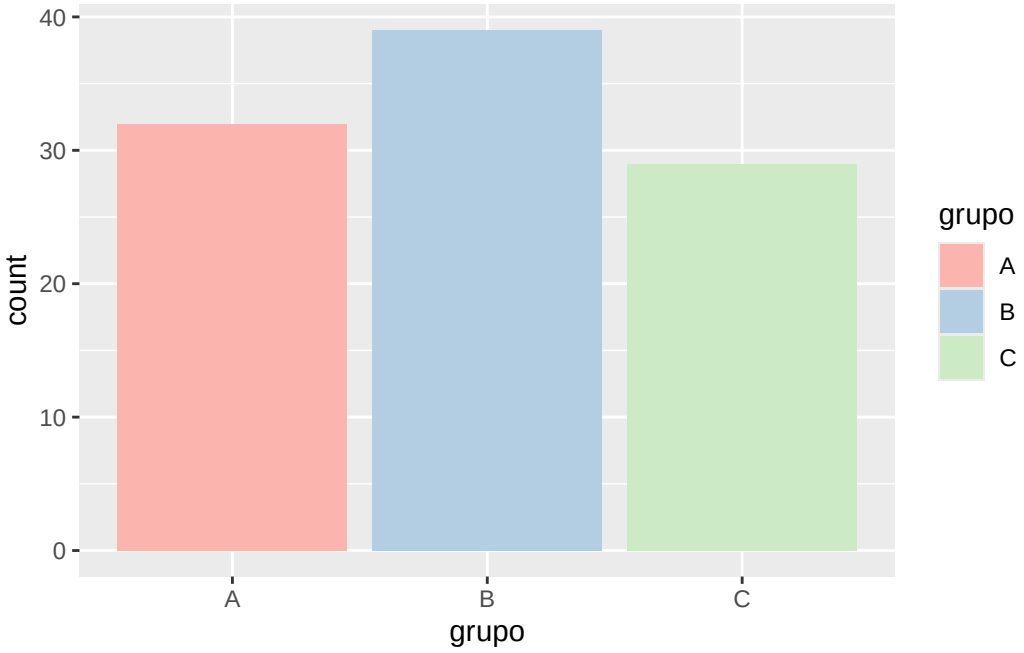
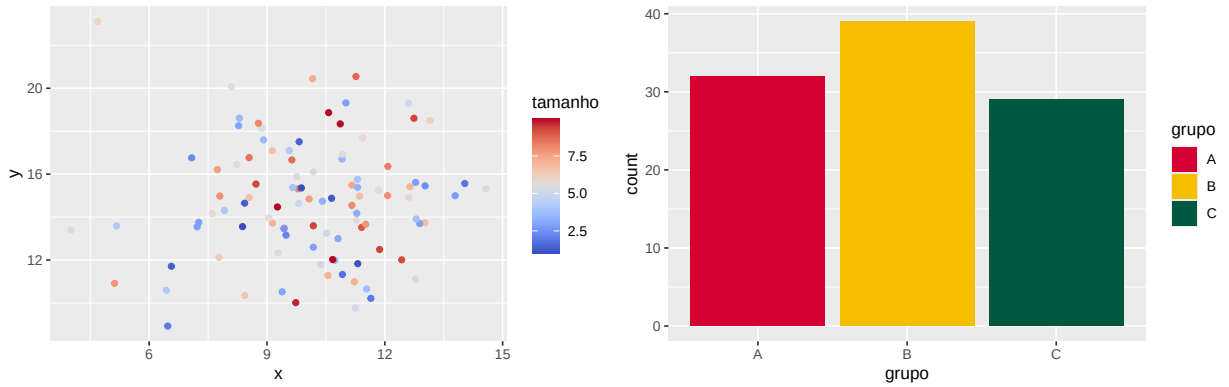


Figura 3.25: `scale_fill_brewer(palette = "Pastel1")`

Alterando as escalas de cores com `scale_*_brewer`.

O pacote `paletter` também possui um site para escolher sua paleta. Para usá-lo precisamos instalar o pacote e depois utilizar as funções adequadas ao tipo de dados (`scale_*_paletteer_c` para contínuas e `scale_*_paletteer_d` para discretas).

```
require(paletteer)

ggplot(dados, aes(x, y, color = tamanho)) +
  geom_point() +
  scale_color_paletteer_c("pals::coolwarm")

ggplot(dados, aes(grupo, fill = grupo)) +
  geom_bar() +
  scale_fill_paletteer_d("nbapalettes::supersonics_holiday")
```



(a) `scale_color_paletteer_c` para paletas contínuas (a) `scale_fill_paletteer_d` para paletas discretas Alterando as escalas de cores com `scale_*_paletteer`.

Para finalizar nossa introdução à criação de gráficos com o `ggplot2` vamos ver como personalizar nosso gráfico. A camada de legendas e títulos do gráfico podem ser alteradas com `labs()`. Os argumentos `title`, `subtitle` e `caption` criam textos para o título, subtítulo e nota de rodapé do gráfico, respectivamente. Outros elementos como as legendas e nomes dos eixos são representados pelo nome do argumento usado no `aes()`, por exemplo, o argumento `color` declara o título da legenda que representa as categorias dessa estética no gráfico. Caso queira remover o título de uma legenda ou eixo atribua o valor `element_blank()`, que indica ao `ggplot2` para não desenhar nada para aquele elemento.

```
p_base <- ggplot(
  dados,
  aes(x, y, color = grupo, size = tamanho, shape = categoria)
  ) +
  geom_point() +
  labs(
    title = "Gráfico de dispersão de X por Y",
    subtitle = "Um subtítulo para esse gráfico",
    x = "Título do eixo X",
    y = "Título do eixo Y",
    shape = "Legenda de shape",
    color = "Legenda de color",
```

```
    size = "Legenda de size",
    caption = "Uma nota explicativa ao conteúdo do gráfico"
  )
p_base
```

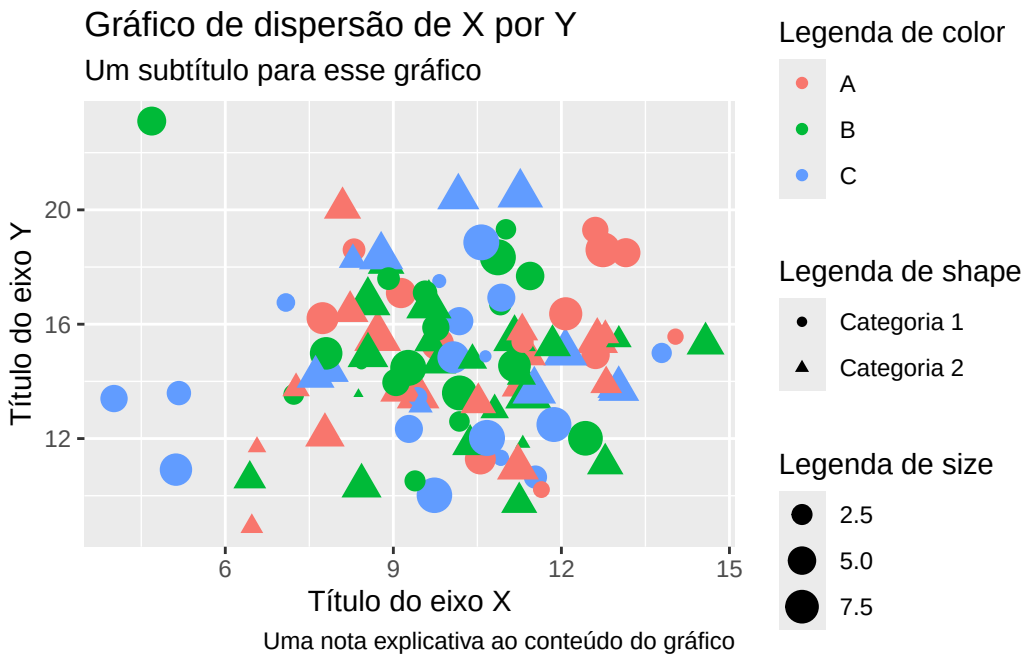
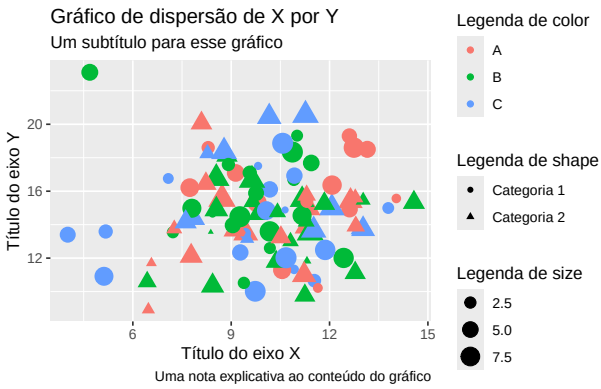


Figura 3.28: Alterando títulos e legendas em `labs()`.

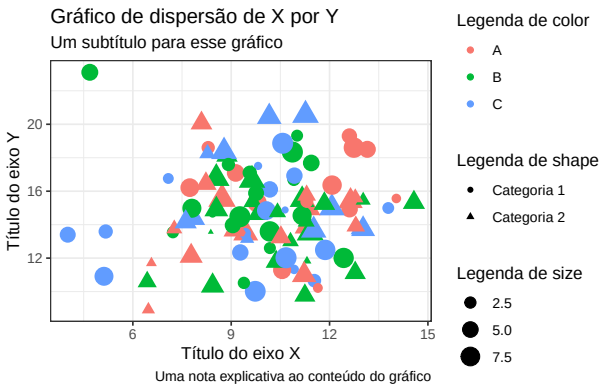
Além do padrão temático inicial contruído no `ggplot2`, o pacote fornece alguns temas já contruídos que podemos usar em nosso gráfico como modelo de estilo e ir adicionando novas personalizações conforme a necessidade. Os seguintes temas estão disponíveis: `theme_gray`, `theme_bw`, `theme_linedraw`, `theme_light`, `theme_dark`, `theme_minimal`, `theme_classic` e `theme_void`.

```
p_base + theme_gray()
p_base + theme_bw()
p_base + theme_linedraw()
p_base + theme_light()
p_base + theme_dark()
p_base + theme_minimal()
p_base + theme_classic()
p_base + theme_void()
```

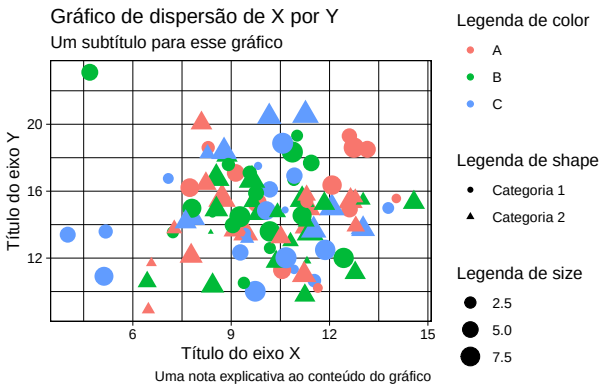
Cada um desses temas predefinidos possuem argumentos para certo controle de sua aparência (verifique os argumentos na documentação deles com `help(nome_do_tema)`), porém, quando queremos alterar algum elemento de aparência do gráfico, normalmente usaremos `theme()`. `theme()` permite personalizar todos componentes não definidos pelos seus dados. Sua utilização envolve declarar um argumento com o nome do componente que queremos modificar e fornecer um objeto do tipo `element_*`. `element_*` representam elementos temáticos: `element_blank` para declarar que o componente não deve ser desenhado, `element_rect` para bordas e planos de fundo, `element_line` para linhas, `element_text` para componentes



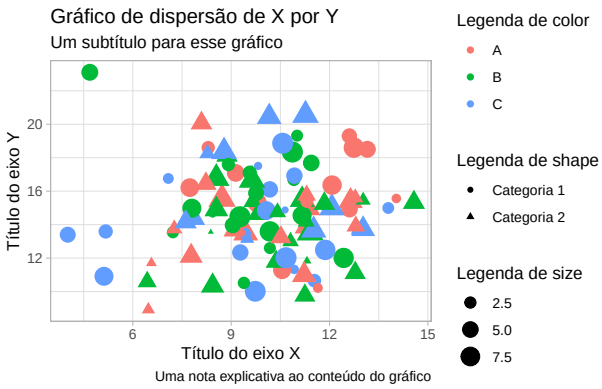
(a) `theme_gray`



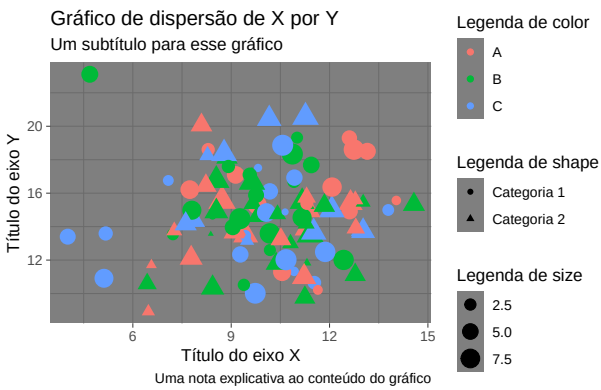
(a) `theme_bw`



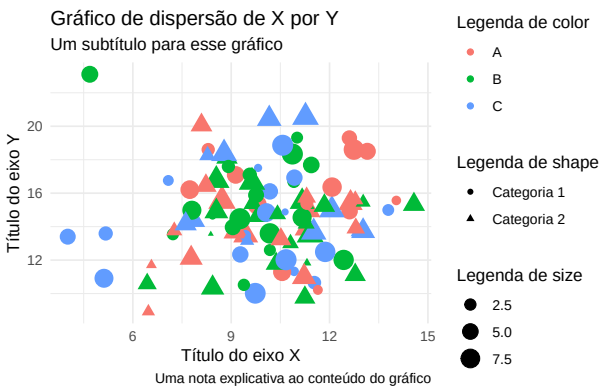
(a) `theme_linedraw`



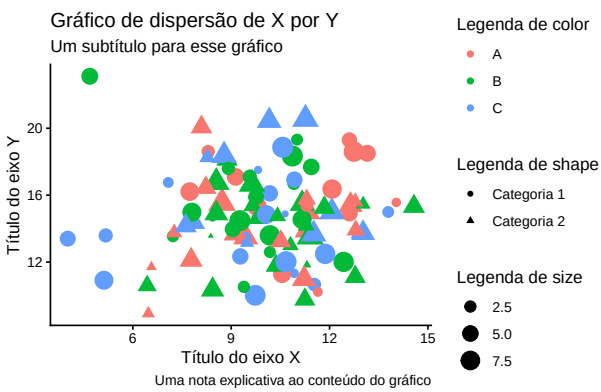
(a) `theme_light`



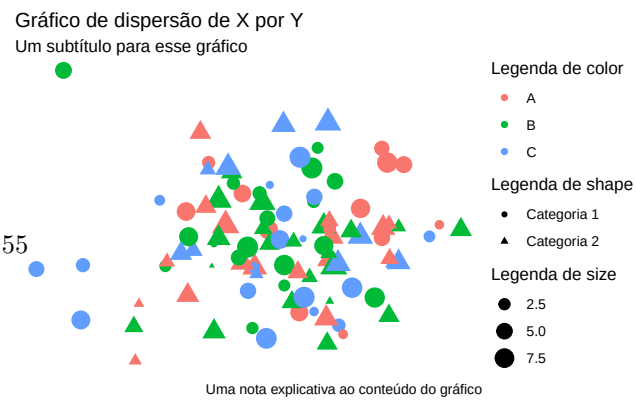
(a) `theme_dark`



(a) `theme_minimal`



(a) `theme_classic`



(a) `theme_void`

de texto, `element_polygon` para polígonos e `element_point` para pontos.

Então, por exemplo, para personalizar o texto do título e subtítulo podemos usar em `theme()` o `plot.title = element_text(args)` e `plot.subtitle = element_text(args)`, para alterar a legenda dos eixos usar o `axis.title = element_text(args)`, para alterar as linhas de guia no gráfico usar `panel.grid.major = element_rect()` e `panel.grid.minor = element_rect()`. Além disso, alguns argumentos de `theme` não recebem `element_`, como o `legend.position`, no qual você declara onde as legendas serão alocadas (topo, abaixo, esquerda ou direita).

```
require(showtext)
font_add_google("Alegreya")
font_add_google("IBM Plex Serif")
showtext_auto()

p_base +
  theme_minimal() +
  theme(
    plot.title = element_text(
      family = "Alegreya", face = "bold", size = 12
    ),
    plot.subtitle = element_text(
      family = "Alegreya", face = "italic", size = 9
    ),
    axis.title = element_text(
      family = "IBM Plex Serif", color = "#e60e3dff", size = 9
    ),
    panel.grid.major = element_line(
      color = "black", linetype = "solid", linewidth = .5
    ),
    panel.grid.minor = element_line(
      color = "black", linetype = "dashed", linewidth = .2
    ),
    legend.position = "top"
  )
```

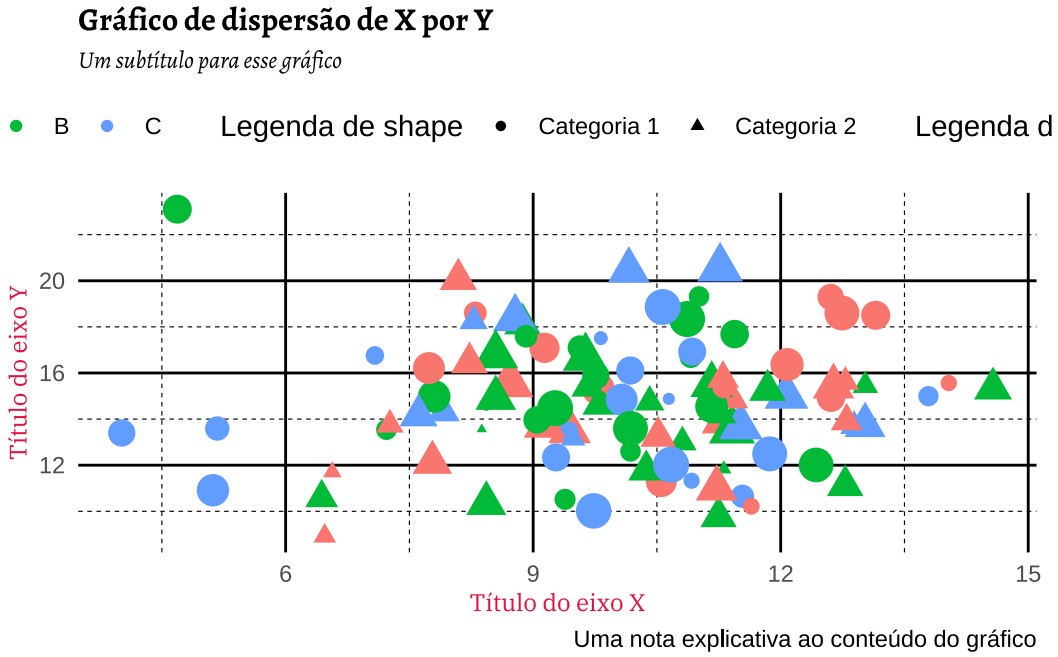


Figura 3.37: Personalizando temas em `theme()`.

Embora o `ggplot2` traga muitas possibilidades de criação de gráficos, há sempre espaço para inovações e especializações na comunidade do R, então recomendo o site ggplot2 extensions que traz uma série de pacotes baseados no `ggplot2`, mas com um "temperinho a mais" para determinados objetivos que poderiam ser particularmente difíceis ou tediosos de alcançar usando somente o `ggplot2` ("não reinvente a roda se ela já foi criada").

- Estatística descritiva univariada
  - Representação gráfica dos dados
  - Representação gráfica para variáveis qualitativas
    - Gráfico de barras
    - Gráfico de setores ou pizza
    - Diagrama de Pareto
  - Representação gráfica para variáveis quantitativas
    - Gráfico de linhas
    - Gráfico de pontos ou dispersão
    - Histograma
    - Gráfico de ramo-e-folhas
    - Boxplot ou diagrama de caixa
  - Medidas-resumo
    - Medidas de posição ou localização
      - Medidas de tendência central
      - Medidas separatrizes
      - Identificação de existência de outliers univariados
    - Medidas de dispersão ou variabilidade
      - Amplitude
      - Desvio-médio
      - Variância

            - Desvio-padrão
            - Erro-padrão
            - Coeficiente de variação
    - Medidas de forma
        * Medidas de assimetria
        * Medidas de curtose
- Estatística descritiva bivariada
    - Associação entre duas variáveis qualitativas
        * Tabelas de distribuição conjunta de frequências
    - Medidas de associação
        * Estatística qui-quadrado
        * Outras medidas de associação baseadas no qui-quadrado
        * coeficiente de Spearman
    - Correlação entre duas variáveis quantitativas
        * Tabelas de distribuição conjunta de frequências
        * Representação gráfica por meio de um diagrama de dispersão
    - Medidas de correlação
        * Covariância
        * Coeficiente de correlação de Pearson