Usando R

Uma introdução para pesquisadores em Humanidades Digitais

Fernando Meireles Denisson Silva

Índice

Prefácio

Este é um manuscrito em desenvolvimento. A versão em *website* deste livro pode ser encontrada em: www.fmeireles.com/livro

Para ver a proposta geral do livro (*book prospect*), clique aqui.

Introdução

A pesquisa quantitativa nas Ciências Sociais cresceu imensamente nas últimas décadas. Com o aumento da disponibilidade de dados e de novos *softwares*, muitas pessoas passaram a ter acesso a um arsenal de ferramentas para testar hipóteses que antes sequer eram imagináveis. Com o mínimo de treinamento, um simples *notebook* pode ser usado para carregar e manipular bases de dados com centenas de milhões de entradas, como aquelas encontradas em microdados censitários.

Se você está lendo este livro, é provável que tenha interesse em aprender a fazer justamente este tipo de análise. É o que esse livro oferece: uma porta de entrada à pesquisa quantitativa utilizando R – uma das linguagens de programação mais populares para análise de dados, tanto na academia quanto fora dela.

Nossa abordagem é prática e direta. Acreditamos que a melhor maneira de aprender a fazer pesquisa quantitativa é por meio da prática. Por isso, em vez de nos aprofundarmos em conceitos abstratos de programação, preferimos usar exemplos aplicados, demonstrando como usar o R para resolver problemas concretos de pesquisa. Este livro, em outras palavras, é um guia de uso. Ao longo dele, você aprenderá a usar R para:

- Realizar tarefas básicas: criar objetos, utilizar funções e instalar pacotes.
- Importar dados, desde planilhas do Excel até bases de dados complexas e/ou imensas.
- Realizar as principais operações de manipulação de dados.
- Criar gráficos informativos e totalmente customizáveis.
- Realizar análises estatísticas, o que inclui calcular estatísticas descritivas, estimar modelos de regressão e testar hipóteses.
- Apresentar resultados de forma replicável em diferentes formatos.

Antes de prosseguir, no entanto, vamos preparar o terreno: o R é um ambiente de programação, o que significa que não abriremos um banco de dados utilizando um menu de tarefas, nem calcularemos estatísticas clicando em um botão. Em vez disso, precisaremos programar, isto é, escrever código de forma ordenada para que o computador o execute sequencialmente. Aprender a programar, especialmente no início, pode ser um pouco difícil, mas acho que não precisamos reforçar o quanto todo o esforço envolvido valerá à pena.

Nossa filosofia

O mote do livro: ensinar a usar R para resolver problemas aplicados de pesquisa social e de análise de dados, e não necessariamente para aprender lógica de programação. No fundo, acreditamos que ir o mais rapidamente possível para a prática é a melhor forma de entender o potencial do R. É por essa razão que não cobrimos de forma aprofundada tópicos considerados essenciais em livros introdutórios de programação, como estruturas de repetição e condicionais e princípios de programação orientada a objetos.

Além da opção geral por um livro prático, procuramos seguir alguns princípios menores na escrita desse livro. São eles:

- Priorizamos código fácil de ler, mesmo que ele seja um pouco mais extenso;
- Preferimos usar ferramentas simples e versáteis para resolver problemas, e não necessariamente as mais eficientes e especializadas;
- Organizamos tarefas de análise de dados em módulos independentes, como importação de dados e manipulação, de forma que cada parte possa ser reutilizada em outros projetos;
- Partimos do pressuposto de que, sempre que possível, análises devem ser replicáveis – qualquer pessoa familiarizada com o R deve ser capaz de reproduzir nossos códigos.

Para quem este livro é indicado?

Recomendamos este livro sobretudo para cientistas sociais, economistas e pessoas de áreas próximas que estão dando seus primeiros passos no mundo da metodologia quantitativa e da análise de dados. Nossa ideia é que ele seja um atalho para o aprendizado de R e que, a partir dele, a leitura de livros e manuais mais avançados, como os Wickham, Çetinkaya-Rundel, e Grolemund (2023) ou o de Aquino (2014), seja mais fácil e proveitosa.

Vamos enfatizar: este não é um livro de programação, pelo menos não em sentido estrito. Antes, ele é um guia introdutório para o uso aplicado do R em pesquisa social quantitativa. Respondendo à pergunta do sub-título, este livro é indicado principalmente para quem quer aprender rapidamente a usar o R para resolver problemas reais de pesquisa.

Como usar o livro

Pensamos este livro como um complemento para um curso de R de curta duração, com cinco aulas. Dessa forma, cada capítulo corresponde a um dia de trabalho: antes dos encontros, alunas e alunos idealmente lerão um capítulo para cobrir o conteúdo exposto em aula e,

no turno oposto, praticam o que foi visto com os exercícios disponíveis ao final do mesmo capítulo.

Os capítulos do livro podem ser lidos ou consultados de forma independente, mas seguem um percurso planejado: começamos com o básico sobre como instalar e usar o R e o RStudio e concluímos com modelos de regressão linear simples e multivariados. Por conta disso, para estudo individual sugerimos que cada capítulo seja lido seguindo a sequência em que são apresentados.

1 Básico

1.1 Introdução ao R

Este capítulo é o nosso primeiro encontro com o R. Nele, veremos alguns dos principais conceitos necessários para poder usá-lo para análise de dados – o que, afinal de contas, é o nosso principal objetivo.

Esta não é uma introdução formal ao R. Antes, este capítulo cobre o fundamental para saltarmos diretamente para o uso de ferramentas mais avançadas, que nos ajudarão a fazer análise de dados e pesquisa acadêmica.

Antes de começar, no entanto, precisaremos instalar o R. Na verdade, precisaremos instalar dois *softwares*: o R e o RStudio. O primeiro é de fato o *software* por detrás da linguagem de programação, mas ele não possui um interface, como o Excel ou outros *softwares* de armazenamento e análise de dados. É por isso que usaremos o R por meio do segundo *software*, o RStudio, que é um interface com um conjunto de funcionalidades que nos ajudará a trabalhar com o R. Depois disso, o restante do capítulo focará em como escrever código em R, como salvar e manipular informações na memória, como usar funções e como instalar pacotes.

1.2 Instalando o R e RStudio

O R é um programa de código aberto[1] que pode ser baixado gratuitamente em https://cran.r-project.org, o site oficial do projeto que mantém o R. Uma vez no site, basta buscar pela opção *download* e seguir as instruções específicas para o seu sistema operacional.[2] A instalação deverá criar um atalho para o R no seu computador que, uma vez acessado, provavelmente mostrará algo como indica a Figura 1.1.

Sem uma interface, o R nada mais é do que um console, uma tela textual onde podemos ler e digitar código. Para termos uma interface melhor, podemos agora baixar o RStudio em https://posit.co/download/rstudio-desktop/, site da empresa que o mantém – apesar de desenvolverem também outras versões, o RStudio *Desktop* – *Open Source License* também é

[1] Código aberto é uma expressão usada normalmente para se referir a programas com um tipo de licença que permite que qualquer pessoa os usem, modifiquem e compartilhem. O R um desses programas e, portanto, é gratuito.

[2] Para quem usa Linux (Debian), é possível instalar o R diretamente pelo terminal, usando o comando `sudo apt-get install r-base`, por exemplo.

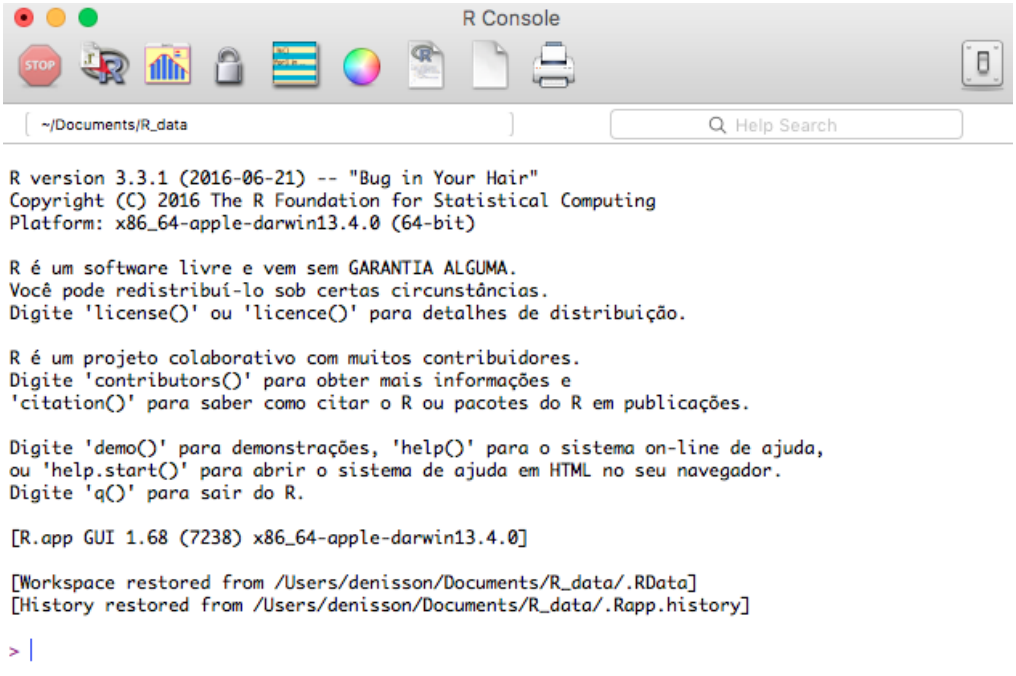


Figura 1.1: Console do R

gratuito. Novamente, basta buscar a opção mais adequada para o seu sistema operacional e seguir as intruções de instalação. Abrindo o atalho do RStudio, a visão deverá ser bem melhor.

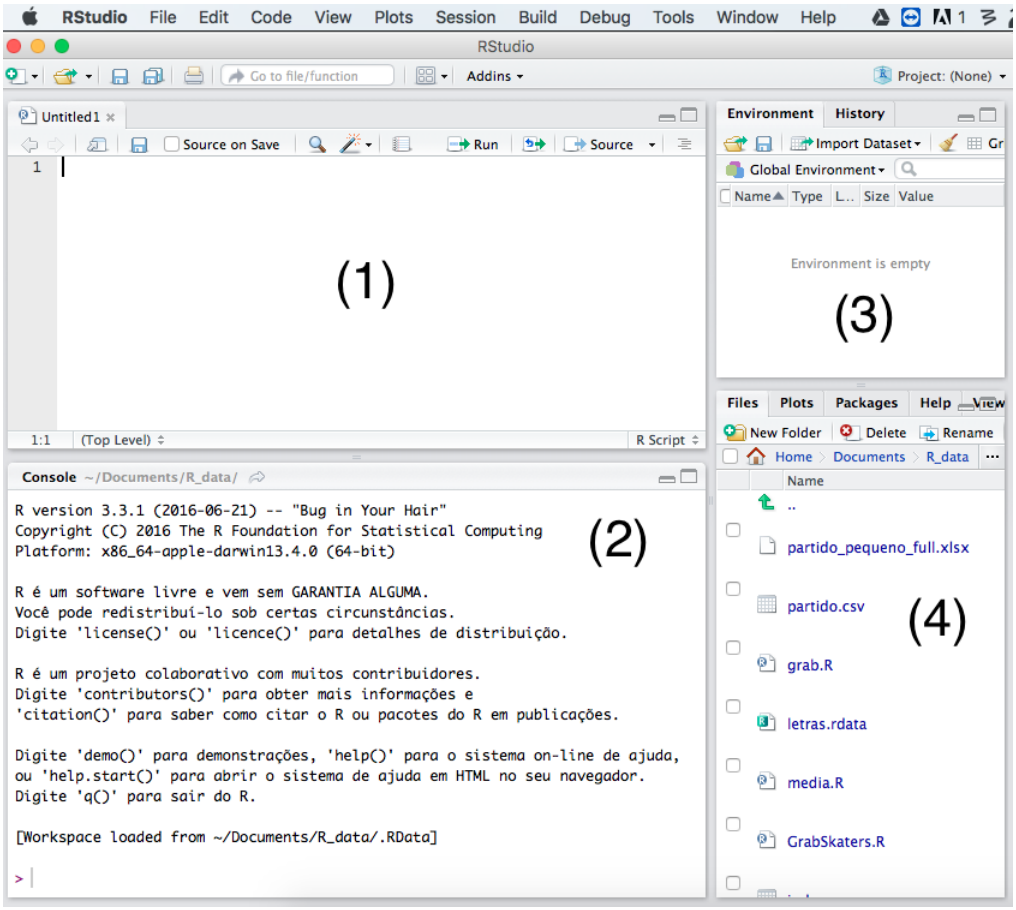


Figura 1.2: RStudio

1.2.1 O RStudio

No RStudio, temos 4 sub-janelas por padrão, isto é, a janela do *software* é dividida em quatro áreas diferentes, como ilustra a Figura. Um resumo da utilidade de cada uma:

1. Janela de *script* na qual escrevemos e documentamos nossos códigos;
2. Console do R, onde podemos executar código e, também, ter retorno de mensagens de erro e avisos;
3. Nesta sub-janela tempo duas abas principais:

i. *Environment*, na qual visualizamos quais objetos estão na memória do R (e.g., vetores, banco de dados, listas);
ii. *History*, na qual podemos ver o histórico dos códigos que já executamos;

4. Aqui temos cinco abas principais:

i. *Files*, onde é possível visualizar a lista de arquivo da pasta (área de trabalho, no jargão do R) em que você está trabalhando no seu computador;
ii. *Plots*, na qual podemos visualizar gráficos criado no R;
iii. *Packages*, que exibi pacotes de funções instalados no R; e, (d) *Help*; onde será visualizadas as ajudas solicitadas dentro do próprio programa;
iv. *View*, usada para visualizar o resultado da execução de certas funções.

De início, o mais importante é pensar no RStudio como uma espécie de pacote Office, mas para o R: é nele que escreveremos nossos códigos, executaremos e visualizaremos os seus resultados.

1.2.2 Usando o R e o RStudio pelo navegador

Para quem tem problemas ao instalar o R, ou não pode instalá-lo por qualquer razão, há uma alternativa simples pela nuvem: o Posit Cloud, uma plataforma mantida pela mesma empresa do RStudio que permite o seu uso diretamente pelo navegador, sem a necessidade de instalação. Para usá-lo, basta criar uma conta no site, selecionar o plano gratuito (*Free forever*) e começar a usar o R de lá. A tela do seu navegador deverá mostrar algo como na Figura 1.3.

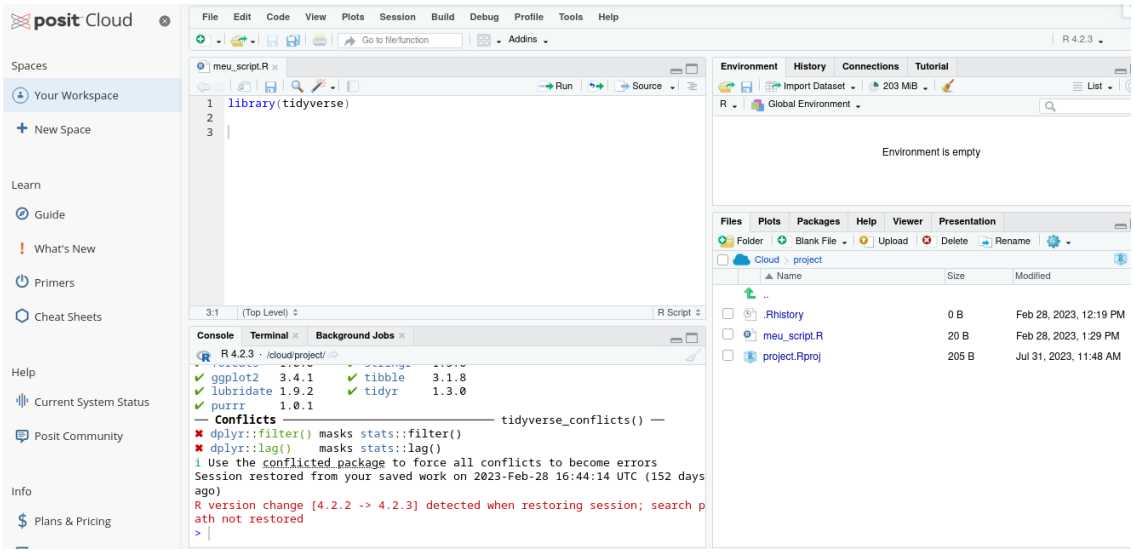


Figura 1.3: Usando o RStudio do navegador

1.3 Sintaxe básica do R

A partir de agora, começaremos a aprender R do jeito mais direto possível: escrevendo e executando códigos. Para tanto, as próximas seções começarão a introduzir exemplos de código que, a princípio, podem parecer confusas. Mas não se preocupe: o objetivo é aprendermos R de forma prática, sem memorizações, entendendo o que cada parte de um código faz.

Daqui até o final do livro, o seguinte se aplicará:

- Tudo o que estiver em caixa cinza, com texto destacado por cores, é código e pode ser executado no R (basta copiar e colar o código no console do RStudio, janela (2) na Figura 1.1, e apertar *enter*);
- Tudo o que estiver logo após precedido de um `[1]` ou algo do tipo é *output* do R, isto é, o resultado da execução de um código;
- Alguns códigos dependem de códigos anteriores; caso encontre algum erro ao rodar um código de exemplo neste livro, tente voltar atrás e rodar os códigos anteriores, pois estamos escrevendo uma sequência de tarefas para chegar em um objetivo.

1.3.1 R como uma calculadora

Assim como em outras linguagens de programação, podemos usar o R como uma calculadora. Experimento digitar `2 + 2` no console do RStudio e apertar *enter*:

```
2 + 2
```

```
[1] 4
```

O R reproduzirá o resultado da soma antecedido por `[1]`. Aproveitando a deixa, `#` indica um comentário: tudo o que vem sucedido de `#` o R não executará.

```
# 2 + 2
```

Nada acontece. Comentários são úteis para documentar nossos códigos, algo que veremos em seguida. Por enquanto, experimente usar o console como uma calculadora (logo veremos usos mais interessantes do R):

```
8 + 7 # Adição (depois do #, nada é executado)
```

```
[1] 15
```

```
8 - 7 # Subtração (depois do #, nada é executado)
```

```
[1] 1
```

Para resolver expressões numéricas, usamos `()`.[3]

```
2 / (3 + 5)
```

```
[1] 0.25
```

```
4 * ((2 ^ 5) / 3)
```

```
[1] 42.66667
```

1.3.2 Operadores

Anteriormente, usamos operadores aritméticos, como `+` e `*` (você deve ter percebido que `*` é o operador de multiplicação no R, e não `x`). No R, existem vários outros (tente adaptar os exemplos):

```
3^2
```

```
[1] 9
```

```
11 / 5
```

```
[1] 2.2
```

```
11 %/% 5
```

```
[1] 2
```

```
11 %% 5
```

```
[1] 1
```

[3] No R, `[]` e `{}` são reservados para outros usos, como veremos ao longo deste livro.

Caso você não tenha entendido algum apenas pelo seu uso, a Tabela 1.1 apresenta uma descrição dos principais operadores matemáticos comuns em R.

Tabela 1.1: Operadores matemáticos no R

Operação	Símbolo
Adição	+
Subtração	-
Divisão	/
Multiplicação	*
Exponenciação	^
Divisão inteira	%/%
Resto da divisão	%%

Além dos operandos matemáticos, existem também operadores lógicos, que usamos para saber se algo é verdadeiro ou falso. Para sermos mais concretos, podemos usar `==` (dois `=`) para testar se um número é igual a outro:

```
1 == 1
```

```
[1] TRUE
```

O que o código anterior faz é testar se `1` é igual a `1`, retornando `TRUE`. Um exemplo falso:

```
2 == 1
```

```
[1] FALSE
```

Testes lógicos também nos permitem fazer operações mais complexas. Por exemplo, podemos testar se um número é maior ou menor que outro:

```
10 > 5
```

```
[1] TRUE
```

```
3 > 1
```

```
[1] TRUE
```

E, indo além, podemos combinar dois testes usando o operador `&` (que significa E, em inglês AND):

```
(10 > 5) & (3 > 1)
```

```
[1] TRUE
```

```
(10 > 5) & (5 < 2)
```

```
[1] FALSE
```

No caso acima, o resultado de cada expressão só será `TRUE` se ambos os testes forem verdadeiros. Se quisermos que o resultado seja `TRUE` se pelo menos um dos testes for verdadeiro, usamos o operador `|` (que significa ou, em inglês OR):

```
(10 > 5) | (5 < 2)
```

```
[1] TRUE
```

E se quisermos testar se um número ou valor pertence a um determinado conjunto? Usamos o operador `%in%`:

```
1 %in% c(1, 2, 3)
```

```
[1] TRUE
```

```
5 %in% c(1, 2, 3)
```

```
[1] FALSE
```

```
1:5 %in% c(1, 2, 3)
```

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

> **Testes lógicos**
>
> Testes lógicos sempre retornam `TRUE` ou `FALSE`, em maiúsculo. Em R, algumas vezes é possível usar `T` e `F` para representar esses valores, mas não é algo recomendado.

A Tabela 1.2 apresenta os operadores lógicos mais comuns:

Tabela 1.2: Operadores lógicos comuns no R

Operação	Símbolo	Exemplo
Igualdade	==	1 == 1
Diferença	!=	1 != 1
Maior que	>	1 > 1
Menor que	<	1 < 1
Maior ou igual	>=	1 >= 1
Menor ou igual	<=	1 <= 1
E	&	(1 == 1) & (2 == 2)
OU	|	(1 == 1) | (2 == 2)
NÃO	!	!(1 == 1)
Pertence	%in%	1 %in% c(1, 2, 3)

Todos esses operadores são úteis – mas certamente não é por causa deles que o R é tão utilizado.

1.4 Funções

Parte da potencialidade do R advém do fato dele conter uma série de funções nativas para realizar as mais diversas tarefas de pesquisa. É por isso que ele é considerado um ambiente, e não apenas uma linguagem de programação.[4] Dito de forma simples, funções são códigos que executam uma tarefa específica. A função `sqrt()`, por exemplo, calcula a raiz quadrada de um número:

```
sqrt(4) # Raiz quadrada do número 4
```

```
[1] 2
```

[4] Embora ele também seja uma linguagem de programação.

Em R, funções têm uma anatomia específica: o nome da função, seguido de parênteses, dentro dos quais estão os argumentos da função – o *input* que a função recebe e processa. No caso da função `sqrt()`, o argumento é o número cuja raiz quadrada queremos calcular.[5] Vale memorizar: uma função nada mais é do que uma espécie de ferramenta que recebe uma determinada informação e a transforma em outra.[6]

1.4.1 Usando funções

No R, as informações que passamos para determinada função vão dentro de parêntesis. A função `sum`, por exemplo, recebe e soma dois ou mais números, todos separados por vírgula. Se esquecermos de fazer essa separação, obtemos um erro.

```
sum(2 2) # retorna erro
```

```
Error: <text>:1:7: unexpected numeric constant
1: sum(2 2
          ^
```

> **Erros**
>
> Quando executamos um código que o R não consegue interpretar, ele retorna um erro no console.

Para corrigir o código anterior, basta separar os números por vírgula:

```
sum(2, 2) # retorna 4
```

```
[1] 4
```

Há algumas outras funções que podemos usar para trabalhar com números. A função `abs()`, por exemplo, retorna o valor absoluto de um número, isto é, o número sem o sinal negativo:

```
abs(-5) # retorna 5
```

```
[1] 5
```

[5] Em R, argumentos são os valores que uma função recebe para executar uma tarefa e, como veremos em seguida, há funções que recebem vários argumentos, alguns deles nomeados.

[6] Há funções que não recebem *inputs*, assim como outros que não retornam *outputs*, mas esses não são os usos mais comuns de funções.

A função `round()` arredonda um número com casa decimal para o número inteiro mais próximo:

```
round(3.14) # retorna 3
```

```
[1] 3
```

Algumas funções em R, no entanto, não recebem e transformam números, mas sim textos. Diferentemente de números, textos devem estar contidos entre aspas para o R o reconhecer como tal. Experimente, por exemplo, digitar o seguinte no console:

```
"Um texto"
```

```
[1] "Um texto"
```

E, agora, experimente digitar o mesmo texto sem as aspas:

```
Um texto
```

```
Error: <text>:1:4: unexpected symbol
1: Um texto
       ^
```

Esse é um erro comum para quem está começando a programar em R. O R não reconhece `Um texto` como um texto, mas sim como um objeto que deveria ser uma função ou um objeto já criado. Tomado esse cuidado, podemos usar funções próprias para manipulação de textos. A função `toupper()`, por exemplo, transforma um texto com letras minúsculas para outro com letras maiúsculas:

```
toupper("Um texto")
```

```
[1] "UM TEXTO"
```

De forma similar, o R nos oferece uma função conveniente, `nchar()`, para sabermos quantos caracteres têm um texto ou palavra:

```
nchar("Um texto maior")
```

```
[1] 14
```

Algo que é extremamente útil, e que veremos de relance agora, é como combinar textos. A função `paste()` faz exatamente isso (o exemplo, assim, parece trivial, mas logo veremos que não é):

```
paste("Um texto", "maior")
```

```
[1] "Um texto maior"
```

Note que, no exemplo anterior, temos dois textos separados por uma vírgula, mas a função `paste()` os combina em um único texto que é retornado no console.

1.5 Objetos

Para além de executar código, o R nos permite salvar informações na memória do programa. Essas informações são armazenadas em objetos, que podem ser usados posteriormente. De forma bem simples, objetos são como locais na memória do programa que armazenam quaisquer valores. No R esses valores podem ser: números, textos, um vetor de números (isto é, uma sequência de números), um banco de dados e, até mesmo, uma função.

Podemos armazenar objetos no R com o operador `<-` (menor que, seguido de hífen). Basicamente, ele diz ao R para armazenar um valor em um objeto para podermos acessá-lo posteriormente. Exemplo: vamos salvar o número 2 em um objeto chamado `x`.

```
x <- 2
```

Tocamos em algo extremamente importante: agora, podemos digitar `x` no lugar de 2 para realizar outras operações.

```
x
```

```
[1] 2
```

```
x + 1
```

```
[1] 3
```

```
x / 2
```

```
[1] 1
```

E como fazemos para salvar o resultado de uma nova operação, como `x + 10`, por exemplo? Simples: basta criar um novo objeto.

```
y <- x + 10
y
```

```
[1] 12
```

Também é possível armazenar texto em um objeto. Note que, para o R reconhecer algo como texto, precisamos colocá-lo entre aspas:

```
texto <- "um texto"
texto
```

```
[1] "um texto"
```

No R elementos entre aspas, simples ou duplas, são considerados textos.

> **Criação de objetos**
>
> No R também é possível criar objetos usando o símbolo de igualdade, `=`, como em `x = 1`. No entanto, não usaremos essa sintaxe neste livro e, por razões de consistência de código, também não recomendamos seu uso.

1.5.1 Tipos de objeto

Números são diferentes de textos e, em R, essa diferença também existe: ela é dada pelas classes de objetos. Classes são como categorias de objetos, isto é, grupos de objetos que compartilham de uma mesma estrutura e, portanto, podem ser manipulados de forma semelhante. O número 1 é um objeto da classe `integer` (inteiro), assim como os números 2 e 10, que também são inteiros. O número 1,5, direfentemente, é um objeto da classe `numeric`, por que não é um número inteiro (por conta da casa decimal). Para saber a classe de um objeto, usamos a função `class()`:

```
class(1)
```

```
[1] "numeric"
```

```r
class(1.5)
```

```
[1] "numeric"
```

Diferentes funções podem exigir diferentes classes de objetos. Por exemplo, a função `sum()` exige que os objetos que ela soma sejam da classe `numeric` ou `integer`. Se tentarmos somar um objeto da classe `character`, o R retornará um erro:

```r
sum("1", "2")
```

```
Error in sum("1", "2"): 'type' inválido (character) do argumento
```

Para resumir, classes determinam o tipo de informação que diferentes objetos armazenam e o que podemos fazer com elas. Entendido isso, podemos começar a aprender sobre as classes mais comuns no R: `integer`, `numeric`, `character`, `factor`, `matrix`, `data.frame` e `list`.

1.5.1.1 Números, textos e categorias

1.5.1.1.1 integer

`integer` é uma classe de objeto específica para números inteiros.

```r
exemplo_inteiro <- 20
class(exemplo_inteiro)
```

```
[1] "numeric"
```

Até agora, só criamos objetos com um elemento, mas, quando estamos analisando muitos dados, podemos combiná-los em vetores, ou seja, objetos com mais de um elementos (mais de um caso). Uma forma elementar de criar um vetor é por meio da função *combine*, `c`:

```r
# Cria um vetor de números
x <- c(18, 20, 19, 25, 21)
x
```

```
[1] 18 20 19 25 21
```

1.5.1.1.2 numeric

A classe `numeric` também é composta por números, mas, diferentemente de `integer`, armazenam números decimais.

```
exemplo_decimal <- 20.5
class(exemplo_decimal)
```

```
[1] "numeric"
```

Por padrão, o R já atribui classe aos objetos quando os criamos, deduzindo o tipo adequado a partir do nosso código. No caso de `integer` ou `numeric`, a escolha está atrelada à quantidade de memória reservada no programa para armazenar as informações: quando temos números decimais, a classe sempre será `numeric` pois é necessário mais espaço para guardar informações das casas decimais, e todos os números do vetor passaram a ter uma decimal, mesmo aqueles que foram declarados (inseridos) sem decimal:[7]

```
y <- c(50, 65.5, 55.8, 70, 85.6)
class(y)
```

```
[1] "numeric"
```

> ⚠ Decimal
>
> O R adota o sistema de casas decimais americano, com ponto. Por isso, ao declarar um número decimal no R, usamos o ponto, e não a vírgula.

1.5.1.1.3 character

Como já dito, `character` é a classe usada no R para armazenar informações textuais, que devem estar contidas entre aspas.

```
w <- c("superior", "médio", "fundamental", "superior")
class(w)
```

```
[1] "character"
```

[7] Na verdade, em R, vetores de inteiros são armazenados como `numeric`, o que você pode ver por conta própria rodando `class(c(1, 2, 3))`.

Assim como vimos com informações números, há várias funções no R para trabalharmos com texto. A função `nchar()`, por exemplo, nos diz quantos caracteres têm um determinado texto:

```
nchar("um texto") # retorna 8
```

```
[1] 8
```

A função `toupper()`, por sua vez, transforma um texto em letras maiúsculas:

```
toupper("um texto") # retorna "UM TEXTO"
```

```
[1] "UM TEXTO"
```

O importante a fixar é que algumas funções servem para trabalhar com números, outras com textos – e, às vezes, com ambos os tipos de informação. Mas isso não é tudo.

1.5.1.1.4 factor

Similar a `character`, `factor` é uma classe que guarda simulneamente uma informação textual com uma númerica associada – o que costumamos chamar de variável categórica nas Ciências Sociais e similares.

```
z <- factor(c("Feminino", "Masculino", "Feminino", "Masculino", "Feminino"))
class(z)
```

```
[1] "factor"
```

```
z
```

```
[1] Feminino  Masculino Feminino  Masculino Feminino
Levels: Feminino Masculino
```

Como podemos ver pelo retorno do R anterior, um vetor da classe `factor` nos mostra seus *levels*, ou seja, as categorias da nossa variavel: `Feminino` e `Masculino`. Mas, como podemos ver, o R não nos mostra os valores numéricos associados a cada categoria. Para isso, podemos usar a função `as.numeric()`, que converte objetos de outras classes para `numeric` (quando essa conversão for possível ou necessária):

```r
as.numeric(z)
```

```
[1] 1 2 1 2 1
```

1.5.1.2 Matrizes e bancos de dados

1.5.1.2.1 matrix

A classe `matrix` é um tipo de objeto bidimensional utilizada principalmente para representar linhas e colunas. De forma geral, matrizes são espécies de tabelas ou planilhas como as que vemos no Excel, mas com uma diferença essencial: todos os elementos devem ser do mesmo tipo, isto é, todos `numeric`, `integer`, `character`, e assim por diante.

Podemos criar uma matriz com a função `matrix`, declarando argumentos que indicam quantas linhas e quantas colunas essa matriz deverá ter. Um exemplo de matriz:

```r
matriz <- matrix(c(1, 3, 4, 5, 6, 7), nrow = 2,  ncol = 3)
matriz
```

```
     [,1] [,2] [,3]
[1,]    1    4    6
[2,]    3    5    7
```

```r
class(matriz)
```

```
[1] "matrix" "array"
```

Note que, no exemplo anterior, criamos uma matriz com 2 linhas e 3 coluna e passamos a ela um vetor com os elementos `c(1, 3, 4, 5, 6, 7)`. Em outras palavras, os argumentos `nrow` (i.e., número de linhas) e `ncol` (i.e., número de colunas) que determinam como o conteúdo da matriz será dividido entre linhas e colunas.

1.5.1.2.2 data.frame

Já que matrizes salvam apenas informações da mesma classe, naturalmente precisamos de outra classe se quisermos analisar variáveis, ou colunas, de classes diferentes. `data.frame` é exatamente a classe que nos permite fazer isso. Especificamente, `data.frame` também é bidimensional e tabular, como a `matrix`, mas é mais versátil.

Vamos criar aqui um banco de dados a partir de vetores com a função `data.frame`:

```r
x <- c("Superior", "Médio", "Médio")
y <- c(23, 45, 63)
z <- c("Feminino", "Masculino", "Masculino")
banco <- data.frame(escolaridade = x, idade = y, sexo = z)
class(banco)
```

```
[1] "data.frame"
```

Com o banco criado, podemos ver suas informações com a função `print`, que serve para mostrar no console o conteúdo de um objeto:

```r
print(banco)
```

```
  escolaridade idade      sexo
1     Superior    23  Feminino
2        Médio    45 Masculino
3        Médio    63 Masculino
```

Para o caso de bancos maiores, podemos usar a função `View()`, que abrirá uma nova janela no RStudio com o conteúdo do banco de dados.[8]

> ⚠ data.frames
>
> Para criar matrizes e bancos de dados a partir de vetores, todos eles precisam ter o mesmo número de elementos, caso contrário o R retornará um erro.

1.5.1.3 Listas

Finalmente, os objetos da classe `list` são um dos mais complexos que veremos – eles são multimensionais. Em particular, com eles armazenamos objetos de diferentes classes, mas não só vetores do mesmo tamanho como em um `data.frame`. Ou seja, em um objeto tipo `list` podemos armazenar vetores de diversos tamanhos, `matrix` e `data.frame`, ou mesmo outras listas. Vejamos um exemplo:

```r
# Cria uma lista chamada 'guarda_trecos'
guarda_trecos <- list(x, y, z, banco)
class(guarda_trecos)
```

[8] Note que, para usar a função `View()` adequadamente, precisamos que o RStudio esteja instalado no computador.

```
[1] "list"
```

```
print(guarda_trecos)
```

```
[[1]]
[1] "Superior" "Médio"    "Médio"

[[2]]
[1] 23 45 63

[[3]]
[1] "Feminino"  "Masculino" "Masculino"

[[4]]
  escolaridade idade      sexo
1     Superior    23  Feminino
2        Médio    45 Masculino
3        Médio    63 Masculino
```

Como podemos ver cada item (objeto) foi armazendos na lista `guarda_trecos`, na ordem em que foram colocado dentro da função list().

1.5.2 Manipulando objetos

Criamos alguns objetos de distintas classes e exibimos eles por completo no console. Mas e se quisermos apresentar no console apenas um elemento de um objeto? Para isso precisamos nos mover pelos objetos usandos índices. Ao exibir elementos de um objeto no console, o R há nos dá uma dica de como fazer isso: o `[1]` sempre indica o conteúdo do primeiro elemento. Se quisermos acessá-lo, basta executar:

```
x <- c(1, 2, 3, 4, 5)
x[1]
```

```
[1] 1
```

De forma geral, em objetos unidimensional basta usar `objeto[índice]`, com a posição que desejada selecionar entre colchetes, para acessar determinado elemento, como o quarto e o quinto, digamos:

```r
x[4]
```

```
[1] 4
```

```r
x[5]
```

```
[1] 5
```

```r
x[c(4, 5)] # Podemos outro vetor para acessar mais de um elemento
```

```
[1] 4 5
```

Pode ser mais útil do que acessar elementos pela sua posição é acessá-los com base em alguma condição (e.g., apenas números maiores que 2). Para isso, podemos usar operadores lógicos dentro dos colchetes de indexação. Por exemplo, para acessar apenas os números maiores que 2, usamos:

```r
x[x > 2]
```

```
[1] 3 4 5
```

Caso você tenha tido dificuldade em entender o código acima, basta pensar no seguinte: `x > 2` retorna um vetor de valores lógicos, `TRUE` ou `FALSE`, que indicam se cada elemento de `x` é maior que 2. Se o resultado do teste lógico for `TRUE` para o primeiro elemento de `x`, ele é mantido; se for `FALSE`, é descartado. Para cada elemento de `x`, o resultado do teste instruirá o R a descartar os dois primeiros elementos de `x` (que são `FALSE`) e a manter os três últimos:

```r
x > 2
```

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

Em objetos multidimensionais como um `data.frame` o modo de acesso de um elemento é um pouco diferente. Por exemplo, no nosso objeto `banco` criado anteriormente precisamos indexar linhas e colunas, `objeto[linhas, colunas]`. Para acessar a célula da primeira linha e da terceira coluna, usamos:

```r
banco[1, 3]
```

```
[1] "Feminino"
```

No exemplo acima, estamo selecionando o elemento (caso) numero 1 que estar na coluna (variável) 3 que é o sexo. É importante fixar: em objeto bidimensional como um `data.frame`, antes da virgula nos colchetes temos as linhas e, só depois da virgula, as colunas. Outro caso:

```r
banco[, 3]
```

```
[1] "Feminino"  "Masculino" "Masculino"
```

Quando deixamos o do lado esquerdo do colchete vazio, estamos dizendo ao R que retorne um vetor com todas as linhas (casos) da coluna (variável) identificada no lado direito da virgula. Nesse exemplo, temos o sexo de todas as pessoas no `banco`. Já aqui, pegamos todas as informações da pessoa na segunda linha do banco:

```r
banco[2, ]
```

```
  escolaridade idade      sexo
2        Médio    45 Masculino
```

Se quizermos selecionar mais um caso ou variável podemos usar um vetor, também podemos usar vetores usando a função `c` ou dois pontos, para criar uma sequência de inteiros entre dois números:

```r
banco[3:5, c(1, 3)]
```

```
     escolaridade      sexo
3           Médio Masculino
NA           <NA>      <NA>
NA.1         <NA>      <NA>
```

No exemplo acima estamos selecionando os casos de 3 a 5 (o código `3:5` cria uma sequência de inteiros de 3 a 5) da base de dados e as variáveis 1 e 3. Assim como em vetores, podemos usar operadores lógicos para selecionar linhas ou variáveis de um `data.frame`. Por exemplo, para selecionar apenas as pessoas com idade maior que 30 anos, precisamos apenas usar o operador lógico `>` dentro dos colchetes de indexação na posição das linhas, isto é, antes da vírgula:

```
banco[banco$idade > 30, ]
```

```
  escolaridade idade      sexo
2        Médio    45 Masculino
3        Médio    63 Masculino
```

Indexadores também funcionam em listas, mas com uma diferença: como listas são objetos multimensionais, precisamos usar dois conjuntos de colchetes para acessar elementos. Por exemplo, para acessar o primeiro elemento da lista `guarda_trecos`, usamos:

```
guarda_trecos[[1]]
```

```
[1] "Superior" "Médio"    "Médio"
```

O primeiro conjunto de colchetes indica que queremos acessar um elemento da lista, enquanto o segundo indica qual elemento queremos acessar. Se quisermos acessar um valor dentro do primeiro elemento da lista, basta adicionar um colchete simples logo depois dos colchetes duplos indicando o índice do elemento desejado:

```
guarda_trecos[[1]][2]
```

```
[1] "Médio"
```

Com isso, selecionamos o segundo elemento do vetor amarzenado na sublista 1. E se o conteúdo da sublista for um `data.frame`, como aceso um valor dentro dele? Assim:

```
guarda_trecos[[4]][1, 3]
```

```
[1] "Feminino"
```

Para `data.frames`, há um jeito mais simples de se acessar o conteúdo inteiro de uma variável: por meio do cifrão (`$`). Por exemplo, para acessar a variável `sexo` do `banco`, basta executar:

```
banco$sexo
```

```
[1] "Feminino"  "Masculino" "Masculino"
```

Como dá para notar, é preciso saber o nome da coluna que queremos acessar para usar esse meio de indexação. Um jeito simples de fazer isso é usando a função `names()`, que retorna os nomes das colunas de um `data.frame`:

```
names(banco)
```

```
[1] "escolaridade" "idade"        "sexo"
```

Assim sabemos que a primeira variável se chama "escolaridade", a segunda "idade", e assim por diante.

Combinando o `$` com os indexadores que vimos há pouco, é fácil obter, por exemplo, o terceiro elemento da variável `sexo` no objeto `banco`:

```
banco$sexo[3]
```

```
[1] "Masculino"
```

Manipular objetos no R pode parecer bastante complicado, mas, com o devido tempo e prática, tudo se torna mais simples. Ao final deste capítulo, sugerimos alguns exercícios que ajudarão no processo.

1.6 *Pipes*

Criar objetos e manipulá-los pode ser algo que rapidamente foge de controle. Por exemplo, imagine que queremos calcular a média da variável `idade` do objeto `banco` e, depois, calcular a sua raiz quadrada com a função `sqrt`. Para isso, executaríamos:

```
media_idade <- mean(banco$idade)
sqrt(media_idade)
```

```
[1] 6.608076
```

Para evitar ter que criar um objeto intermediário para salvar a média, podemos usar *pipes*, que são representados por `|>`.[9] Eles servem para encadear resultados de funções, isto é, executar uma função e, em seguida, executar outra função com o resultado da primeira. No exemplo anterior, poderíamos usar o *pipe* para calcular a média e, em seguida, calcular a raiz quadrada do resultado com o seguinte código:

[9] O *pipe* `|>`, chamado de *pipe* nativo, foi introduzido na versão 4.1.0 do R. Anteriormente, o pacote `magrittr`, parte do `tidyverse`, era a única fonte de *pipe*, com o operador `%>%`. Para saber mais sobre as diferenças, ver um resumo do blog do `tidyverse` em https://www.tidyverse.org/blog/2023/04/base-vs-magrittr-pipe/.

```
banco$idade |>
    mean() |>
    sqrt()
```

```
[1] 6.608076
```

O código acima é muito mais legível. Podemos, inclusive, ler o código como se fosse uma frase: "pegue a variável `idade` do objeto `banco` e *jogue* ela dentro da função que calcula a média; depois, pegue esse resultado e *jogue ele* dentro da função que calcula a raiz quadrada".[10] Com *pipes*, podemos criar complexas sequências:

```
banco$idade |>
    mean() |>
    sqrt() |>
    round() |>
    print()
```

```
[1] 7
```

Talvez esse tópico pareça um pouco confuso agora, mas, quando começarmos a cobrir a manipulação de bases de dados, no Capítulo 3, *pipes* serão uma ferramenta essencial.

1.7 Pacotes

O R já vem com uma série de funcionalidades embutidas nele – como as funções `sqrt` e `sum`, que já vimos. Mas, como já dito, uma das grandes vantagens do R é a sua comunidade, que desenvolve novas funcionalidades para a linguagem e, norlamente, as disponibilizam por meio de pacotes, ou bibliotecas. Estes são como extensões do R, que adicionam novas funcionalidades ao programa – pense em um pacote como um livro de receitas ou um manual de instruções, que ensina o R como fazer coisas novas.

Em R, a principal fonte de pacotes o *CRAN (The Comprehensive R Archive Network)*, que é uma comunidade de desenvolvedores que mantém o código base do R e os seus pacotes oficiais, aqueles que passaram por uma série de testes e que seguem uma série de protocolos que garantem o seu funcionamento estável e harmônico com outras ferramentas no R.[11]

[10] Note que, para usar *pipes*, precisamos usar a função `mean()` sem argumentos, isto é, sem o nome da variável que queremos calcular a média. Isso porque, com *pipes*, o resultado da função anterior é passado para a próxima função, e não precisamos mais especificar o objeto que queremos usar.

[11] Enquanto escrevemos este livro, o CRAN se aproxima de 20 mil pacotes oficiais mantidos em seu *website*. Informação disponível em: https://cran.r-project.org/web/packages/.

1.7.1 Instalando Pacotes

Para instalar pacotes que está no CRAN, basta sabermos o seu nome e usar a função `install.packages`:

```
install.packages("electionsBR")
```

No exemplo acima, instalamos o pacote `electionsBR` e, com ele, damos ao R a capacidade de se conectar ao repositório de dados eleitorais do TSE (Tribunal Superior Eleitoral) para obter informações eleitorais.

1.7.2 Instalação de pacotes do GitHub

Apesar da imensidade de pacotes no CRAN, encontramos outro grande volume de pacotes em outros repositórios não-oficiais, a maioria em densevolvimento. A principal fonte destes pacotes, depois do CRAN, é o GitHub.[12]

Para instalar pacotes do GitHub, precisamos instalar outro pacote antes, o `remotes`:

```
install.packages("remotes")
```

Este pacote contém uma função, `install_github`, que permite ao R se conectar ao GitHub, obter de lá o código fonte de um pacote e realizar a sua instalação. Para usar esta função, precisamos antes carregar o pacote `remotes`, isto é, tornar ela acessível ao R, o que fazemos por meio da função `library`:

```
library(remotes) # Carrega o pacote
install_github("silvadenisson/electionsBR") # Instala o pacote
```

Instalar e carregar pacotes são duas tarefas similares, mas suas diferenças são importantes: no primeiro caso, estamos incorporando novas funções no nosso R, assim como instalar o Office no computador nos permite usar o processador de texto Word; no segundo caso, estamos carregando o pacote instalado, assim como quando abrimos o Word pelo seu atalho no computador.

[12] GitHub é uma plataforma de armazenamento e versionamento de *software* criado em cima do Git, um *software* de código aberto de controle de versões. Atualmente, o GitHub é um dos maiores repositórios de código aberto no mundo. Para acessá-lo, visite https://github.com.

> **Pacotes**
>
> Pacotes só precisam ser instalados uma vez, mas precisam ser carregados (abertos) no R em cada seção em que precisarmos de suas funções.

No exemplo anterior, usamos a função `library` para carregar o pacote `remotes`; com este, usamos a função `install_github` para instalarmos a versão de desenvolvimento do pacote `electionsBR`.

1.8 Scripts

Trabalhar no console, digitando e executando código diretamente de lá, é algo rápido para tarefas simples, mas inviável para análises mais complexas. Pior que isso, sem poder salvar nossos códigos em algum lugar, não temos como reproduzir uma análise, ou compartilhar nossos passos com outras pessoas. Justamente para evitar isso, usamos *scripts*, documentos de texto que servem para documentar e armazenar códigos.

No Rstudio, podemos criar um *script* clicando, no menu superior esquerdo, em *File* > *New File* > *R Script*, ou, também no canto superior esquerdo, no símbolo de uma folha em branco acompanhada de um símbolo de mais em verde. Feito isso, uma nova janela será aberta, na qual podemos escrever nosso códigos. Para salvá-los, basta clicar em *File* > *Save* e escolher um nome para o arquivo, ou clicar no ícone de disquete ligeiramente acima, ou, ainda, teclar *ctrl/command + s*. O *script* salvo aparecerá na sub-janela de gestão de arquivos do RStudio, indicada no item 4 da Figura 1.2.

Para acompanhar o restante deste capítulo e os próximos, acostume-se a criar *scripts* e use comentários para descrever o que cada linha faz – isso será muito útil para documentar o que estamos aprendendo. A título de exemplo, um *script* de acompanhamento deste capítulo poderia ter o seguinte início:

```
# Capítulo 1: Introdução ao R

# Este é um comentário. Ele não é executado pelo R, mas serve para documentar o que estamos
# Para executar um código, basta clicar na linha e teclar ctrl/command + enter.
# Para executar várias linhas, basta selecioná-las e teclar ctrl/command + enter.

# Criando objetos
x <- 2
y <- x + 10

# ...
```

Documentar o script é uma das tarefas mais importantes do densolvimento do seu código. Primeiro porque podes voltar em um outro momento e saber o que exatamente estas tentendo fazer com seu script. Isso pode parecer tolice, mas tenha certeza que não é, principalmente quando chegamos no nível de trabalhar com muitos scripts.

Esse motivo acima já sería suficiente, no entanto, há outro mais importante para o desenvolvimento de pesquisas científicas que é a replicabilidade. Pois, quando documenta teu código aumenta a capacidade replicativa dele. E replicabilidade é a plavra que chave na ciência, porque não fazemos ciência para ficar na gaveta, ou melhor, em pasta perdida dentro do computador, e sim para que outra pessoa saibam o que fizemos e possam replicar, vamos abordar mais sobre replicação no capítulo 8.

1.9 Recomendações

Podemos criar objetos e realizar operações no R de forma simples, como vimos. No entanto, algumas coisas devem ser evitadas quando escrevemos nosso código, seja para evitar erros ou para facilitar a leitura dele por outras pessoas.

A recomendação mais básica neste sentido é: evite criar objetos com nomes que comecem com números, caracteres especiais ou nomes de funções. Algumas destas coisas produzirão erros imediatos; outras, podem complicar códigos inteiros. Alguns exemplos.

```
# Exemplos de nomes de objetos que produzem erros
2x <- 1
_x <- 1
&x <- 1
```

```
Error: <text>:2:2: unexpected symbol
1: # Exemplos de nomes de objetos que produzem erros
2: 2x
    ^
```

```
# Exemplos de nomes de objetos que não produzem erros imediatos
sum <- 1
sqrt <- 1
```

Também note que o R é *case sensitive*, `A` (maiúsculo) não é a mesma coisa que `a` (minúsculo).

```
A <- 1
print(a)
```

```
Error in eval(expr, envir, enclos): objeto 'a' não encontrado
```

Sempre que criar um objeto armazenando texto, não esqueça das aspas (outra forma de cometer erros no R bastante comum).

```
x <- Texto
```

```
Error in eval(expr, envir, enclos): objeto 'Texto' não encontrado
```

Por fim, quando abrir parênteses, não esqueça de fechá-los (caso contrário, aparecerá um + no console, indicando que o R espera mais conteúdo). Caso esteja executando um código e não saiba porque apareceu um + no cosole, opte por cancelar a operação e volte ao código para ver o que há de errado.[13]

1.9.1 Estilo

Não é algo obrigatóro, mas algumas noções de estilo nos ajudam a compreender e partilhar códigos, tanto nossos quanto os de outras pessoas. Resumidamente, as principais considerações aqui são:

- Use espaços entre objetos, operadores e chamadas a funções;
- Use quebra de linhas para separar blocos de códigos;
- Sempre que possível, crie objetos apenas com letras minúsculas;
- Se precisar separar o nome de um objeto, use _ (underscore);
- Prefira nomes curtos para objetos;
- Prefira o atribuidor <- a = (eles fazem a mesma coisa).

```
y<-1 # Ruim
y <- 1 # OK

y+y+y+y # Ruim
```

```
[1] 4
```

```
y + y + y + y # OK
```

```
[1] 4
```

[13] Estas e outras recomendações comuns para evitar erros podem ser vistas em: http://www.alex-singleton.com/R-Tutorial-Materials/common-error-msg.pdf (em inglês).

```
print (y) # Ruim
```

```
[1] 1
```

```
print(y) # OK
```

```
[1] 1
```

```
y = 1 # Ruim
y <- 1 # OK

OBJETO <- 1 # Ruim
objeto <- 1 # OK

meu.objeto <- 1 # Ruim
meu_objeto <- 1 # OK

objeto_com_nome_excessivamente_grande <- 1 # Ruim
objeto <- 1 # OK
```

Para uma lista mais completa de recomendações, pessoas desenvolvedoras por trás do RStudio criaram um *website* com um guia completo de estilo em R – feito especificamente para pessoas que usam seus pacotes. O guia pode ser visto em: https://style.tidyverse.org/.

1.10 Obtendo ajuda

Para a nossa sorte, a comunidade em torno do R cresceu muito nos últimos anos e, com ela, a quantidade de material disponível na internet. Sempre é bom dizer: dúvidas e erros podem e devem ser buscados no Google ou, mais recentemente, no ChatGPT[14] ou Google Bard[15], ótimas fontes para resolução de dúvidas. De toda forma, a maneira mais simples de se obter ajuda no R sobre alguma função ou operador é consultando a sua documentação – em geral, muito boa. Para isso, podemos usar a função `help`:

```
help(sum)
```

[14] Para quem eventualmente o desconheça, o ChatGPT é uma interface para o modelo generativo de texto (*large language model*) GPT-4, desenvolvido pela Openai. Para saber mais, acesse <chat.openai.com/>.
[15] Bard é o *large language model* do Google, que pode ser acessado em https://bard.google.com/.

Esse recurso só é útil, entretanto, quando sabemos o nome exato da função que queremos consultar (e quando a temos instalado e carregado o pacote que a função pertence). Outra forma de consultar documentação é usando um ponto de interrogação antes do nome de uma função:

```
?sum
```

Quando não sabemos o nome da função que queremos usar, ou até mesmo para saber se existe no R uma função específica para uma determinada tarefa que queremos executar, precisamos recorrer a outras fontes. Antes mesmo de ir para o Google, contudo, há no próprio R um pacote que faz uma busca por palavras-chave nos repositórios oficiais do R, o `sos`. Para usá-lo, precisamos instalá-lo e, depois, carregá-lo:

```
install.packages("sos")
```

```
library(sos)
```

E, então, usar a função `findFn`, que tem como argumento principal um texto (*string*) que será pesquisado. Exemplo:

```
findFn('regresion')
```

Quando executada, a função irá abrir seu navegador em uma pagina com os resultado, como a Figura 1.4 ilustra.

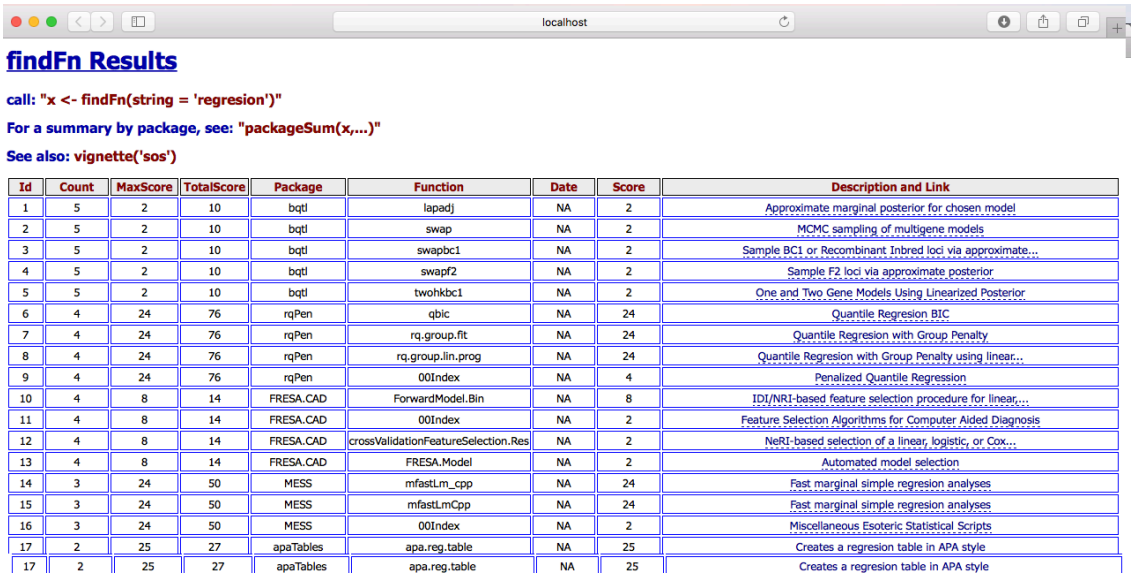
Como é possível ver, usando o pacote `sos` obtemos uma lista com nome de pacotes, o nome da função específica que tem algum relação com o termo pesquisado e uma breve descrição e página que podemos acessar para ver mais detalhes.

1.10.1 Como pedir ajuda ao ChatGPT

Pela nossa experiência recente oferecendo treinamento em R, muitas pessoas ou não usam, ou usam inadequadamente, soluções como o ChatGPT para obter ajudar. Por isso, algumas dicas para obter recursos de forma mais eficiente e eficaz:[16]

- Não use o ChatGPT para gerar código do zero – esse é o pior uso possível dele; sempre escreva algum código antes de pedir ajuda – caso contrário, ele poderá usar pacotes ou funções, ou mesmo seguir uma lógica, que você não conhece;

[16] Um recurso importante para qualquer usuário do ChatGPT é o guia de prompt engineering da OpenAI que fornece uma série de dicas práticas sobre como fazer perguntas ao modelo.



findFn Results

call: "x <- findFn(string = 'regresion')"

For a summary by package, see: "packageSum(x,...)"

See also: vignette('sos')

Id	Count	MaxScore	TotalScore	Package	Function	Date	Score	Description and Link
1	5	2	10	bqtl	lapadj	NA	2	Approximate marginal posterior for chosen model
2	5	2	10	bqtl	swap	NA	2	MCMC sampling of multigene models
3	5	2	10	bqtl	swapbc1	NA	2	Sample BC1 or Recombinant Inbred loci via approximate...
4	5	2	10	bqtl	swapf2	NA	2	Sample F2 loci via approximate posterior
5	5	2	10	bqtl	twohkbc1	NA	2	One and Two Gene Models Using Linearized Posterior
6	4	24	76	rqPen	qbic	NA	24	Quantile Regresion BIC
7	4	24	76	rqPen	rq.group.fit	NA	24	Quantile Regresion with Group Penalty
8	4	24	76	rqPen	rq.group.lin.prog	NA	24	Quantile Regresion with Group Penalty using linear...
9	4	24	76	rqPen	00Index	NA	4	Penalized Quantile Regression
10	4	8	14	FRESA.CAD	ForwardModel.Bin	NA	8	IDI/NRI-based feature selection procedure for linear,...
11	4	8	14	FRESA.CAD	00Index	NA	2	Feature Selection Algorithms for Computer Aided Diagnosis
12	4	8	14	FRESA.CAD	crossValidationFeatureSelection.Res	NA	2	NeRI-based selection of a linear, logistic, or Cox...
13	4	8	14	FRESA.CAD	FRESA.Model	NA	2	Automated model selection
14	3	24	50	MESS	mfastLm_cpp	NA	24	Fast marginal simple regresion analyses
15	3	24	50	MESS	mfastLmCpp	NA	24	Fast marginal simple regresion analyses
16	3	24	50	MESS	00Index	NA	2	Miscellaneous Esoteric Statistical Scripts
17	2	25	27	apaTables	apa.reg.table	NA	25	Creates a regresion table in APA style
17	2	25	27	apaTables	apa.reg.table	NA	25	Creates a regresion table in APA style

Figura 1.4: Pacote sos

- Não use o ChatGPT para obter ajuda sobre pacotes ou funções que você não conhece – use o `help` ou o `?` para isso; modelos de linguagem não são tão bons para recuperar informações factuais, pelo menos não sem contexto ou conexão com fontes de informação;
- Tente seguir um modelo de ajuda específico: descreva em palavras o que está tentando fazer; cole o código que estava usando; e, ao final, cole o erro ou a mensagem obtida no console. Um exemplo:

  Estou tentando criar um objeto chamado `x` com o valor 2 no R, mas obtive um erro ao tentar somar `x+x`. O que estou fazendo de errado? Segue o código que usei e o erro que obtive:

  ```
  x <- "2"
  x + x
  ```

  Error in x + x: argumento não-numérico para operador binário

Um exemplo de como ele poderia responder:

> O erro ocorreu porque você atribuiu o valor 2 a x como uma string (texto), e não como um número. No R, os valores entre aspas (" ") são tratados como texto (ou caracteres), e não como números. Por isso, quando você tentou somar x + x, o R não conseguiu realizar a operação, pois estava tentando somar textos, não números. Para corrigir o erro e fazer a soma funcionar, você deve atribuir o valor 2 a x sem as aspas, assim:

```r
x <- 2
x + x
```

E, dessa forma, em vez de apenas pedir um código e não entender o que ele faz, ganhamos uma explicação didática sobre o que estava errado.

1.10.2 Outros recursos

Alem das opções de ajuda dentro do próprio R, a internete está cheia de material sobre. Por exemplo, nos últimos anos aumentou largamente a quantia de tutoriais no YouTube ensinando as mais diversas tarefas em R. Mais útil, há várias fontes ricas para buscar sobre os mais diferentes tópicos, como o Stackoverflow, R-bloggers, R Brasil - Programadores (Facebook), rbloggersbr (twitter), entre outros.

O primeiro deles, o Stackoverflow, é um fórum onde programadores de todos os níveis e linguagens publicam suas dúvidas e soluções a elas. Originalmente em inglês, conta também com uma versão em português: https://pt.stackoverflow.com. Para refinar as busca dentro do fórum é necessario, antes do termo buscado, inserir o nome da linguagem dentro de colchetes. Por exemplo: `[R] data.frame`.

O R-bloggers é outro site famoso na comunidade de R por reunir postagens de vários blogs sobre R. Em certo sentido, ele é um agregador de tutoriais (em inglês). Seu endereço é https://www.r-bloggers.com/tag/rblogs/.

Em português, finalmente, há também uma iniciativa no Twitter para agregar as postagens do blogs brasileiros cadastrados, https://twitter.com/rbloggersbr. Para quem costuma usar a rede social, basta postar sobre R usando a *hashtag* `#rstats` para rapidamente se conectar a outras pessoas interessadas pela linguagem – no fim das contas, como sugerimos ao longo deste capítulo, o R é também uma comunidade, e não apenas uma linguagem de programação.

1.11 Resumo do capítulo

Neste capítulo, aprendemos os conceitos básicos do R, como instalar e carregar pacotes, criar objetos, usar funções e obter ajuda. Também vimos algumas recomendações para escrever códigos mais legíveis e eficientes. Com o que vimos aqui, ainda não conseguimos fazer análises em R, mas já aprendemos a usar alguns dos ingredientes que precisaremos para isso.

1.12 Indo além

No início, não há alternativa: a melhor forma de aprender R é escrever código em R. Por isso, para quem deseja ir além do que vimos, recomendamos fortemente a realização dos exercícios deste capítulo – mesmo que você já tenha feito algum curso de R antes. Para além destes, para quem deseja complementar a leitura com vídeos, sugerimos a série vídeos introdutórios feitos pelo *R Ladies Belo Horizonte*, capítulo local do *R Ladies Global*[17], que está disponível no YouTube.

Por ser também uma linguagem de programação, o R conta com recursos, que não vimos neste capítulo, como estruturas de controle de fluxo e funções, que se conectam a tópicos mais gerais de programação, como programação funcional e orientada a objetos. Em um curso de introdução à linguagem, ou de introdução à programação de forma mais geral, alguns desses tópicos são abordados já no início. Nesse ponto, sugerimos a leitura do livro de Aquino (2014), que avança por alguns deles.

Exercícios

Para realizar estes exercícios, crie um *script* no R e salve-o com um nome de fácil identificação, como `exercicios_cap1.R`. Use comentários para descrever o que cada linha do código faz. Um exemplo de como organizar o seu arquivo:

```
# Capítulo 1: Exercícios

# 1) Primeiros passos com R
meu_ano_nascimento <- 1990
# ...
```

1. Primeiros passos com R

Crie um objeto chamado `meu_ano_nascimento` e salve nele o ano do seu nascimento. Em seguida, crie um objeto chamado `ano_atual` e salve nele o ano atual. Por fim, crie um objeto chamado `minha_idade` e atribua a ele a diferença entre `ano_atual` e `meu_ano_nascimento`. Use o console para visualizar o valor de `minha_idade`.

[17] O R Ladies é uma iniciativa voltada a promover a diversidade de gênero na comunidade R. Para saber mais, acesse https://rladies.org/.

2. Trabalhando com textos

Crie um objeto chamado `meu_nome` e salve nele o seu nome como um texto (lembre-se de usar aspas). Em seguida, use a função a `paste()` para criar uma frase que diga "Meu nome é [meu_nome]", substituindo [meu_nome] pelo objeto recém criado.

3. Usando funções básicas

Calcule a raiz quadrada do número de letras no seu nome (use o objeto `meu_nome` e a função `nchar()` para contar as letras). Salve o resultado em um objeto chamado `raiz_nome`.

4. Criando e usando vetores

Crie um vetor chamado `notas` com cinco valores que representam notas que você recebeu em algum curso ou disciplina (use valores de 0 a 10). Calcule a média das notas usando a função `mean()` e salve o resultado em um objeto chamado `media_notas`.

5. Usando lógica condicional

Teste se a média das suas notas, salvas no objeto `notas`, é maior que 8. Crie um objeto chamado `aprovado` para guardar os resultados desse teste – eles devem indicar com o valor `TRUE` notas maiores que 8, caso contrário, `FALSE`. Use um comentário, `#`, para explicar o que o seu código faz.

6. Trabalhando com textos (*strings*)

Existe uma função em R chamada `abbreviate()` que abrevia palavras. Use essa função para abreviar o seu nome, salvo em `meu_nome`, e salve o novo resultado em um objeto chamado `nome_abreviado`. Use um comentário para explicar como a função abreviou seu nome, isto é, para tentar explicar a lógica por detrás da abreviação.

7. Operações com vetores I

Crie um vetor chamado `anos` que contenha os últimos cinco anos, incluindo 2024. Com esse objeto, faça o seguinte:

- Usando o objeto `meu_ano_nascimento`, calcule quantos anos você tinha em cada um dos cinco anos no vetor `anos`. Salve o resultado em um objeto chamado `minhas_idades`.

- Descubra uma forma de calcular a média das idades que você tinha nos anos do vetor `anos`. Salve o resultado em um objeto chamado `media_idades`.
- Subtraia a média das idades que você tinha nos anos do vetor `anos` da sua idade atual.

8. Operações com vetores II

Usando o vetor `notas` e o objeto `media_notas` criados anteriormente, identifique quais notas são menores que a sua média. Crie um vetor chamado `notas_abaixo_media` com apenas as notas que são menores que a média. Use comentários para explicar o seu código.

9. Explorando data.frames

Crie um `data.frame`, isto é, uma base de dados, chamado `dados_pessoais` com três colunas: `ano` e `idade` usando os vetores `anos` e `minhas_idades` criados anteriormente. Com esse `data.frame`, use nele as seguintes funções e descreva, usando comentários, o que cada uma faz:

- `names()`
- `nrow()`
- `ncol()`
- `View()`

10. Manipulando data.frames I

Copie e cole o seguinte código para criar um `data.frame` com algumas informações sobre as capitais do Sudeste brasileiro:

```
capitais_sudeste <- data.frame(
  capital = c("Belo Horizonte", "São Paulo", "Rio de Janeiro", "Vitória"),
  estado = c("MG", "SP", "RJ", "ES"),
  populacao_por_mil = c(2315, 11451, 6211, 322)
)
```

Crie um novo `data.frame` que mantenha apenas as cidades com mais de 5 milhões de habitantes (`5000` milhares, na escala do exemplo). Use qualquer forma que achar para resolver o problema e, ao final, descreva o que fez usando comentários.

11. Manipulando data.frames II

Usando o `data.frame` `capitais_sudeste` criado anteriormente, altere os conteúdos da variáveis `estado` para indicar o nome completo dos estados. Salve o resultado em um novo `data.frame` chamado `capitais_uf`.

12. Instalação e uso de pacotes

Instale o pacote `ggplot2` (que estudaremos no Capítulo 4). Com ele instalado, apenas execute o seguinte código (certifique-se de ter criado o `data.frame` `dados_pessoais` no exercício 9):

```
ggplot(data = dados_pessoais, aes(x = anos, y = idade)) +
    geom_line() +
    geom_point()
```

2 Importação

No capítulo anterior, aprendemos a criar e a usar objetos e bases de dados R – mas isso nem de longe cobre as habilidades necessárias para realizar uma análise. Na verdade, ainda não aprendemos algo essencial: carregar nossas próprias bases de dados. Certamente existem outras coisas úteis para se aprender no R, mas, para os nossos objetivos, esta é quase obrigatória.

Neste capítulo, veremos como carregar os mais diversos tipos de dados no R, desde planilhas Excel até formatos mais modernos, como arquivos `parquet` com dezenas ou centenas de *gibabytes*. Com este conhecimento, dominaremos um pequeno conjunto de ferramentas para trazer ao R os mais diversos tipos de informação para análise: textos, bancos de dados criados em outros *softwares*, microdados censitários ou administrativos, arquivos com formatos específicos, entre outros.

Para ilustrar o conteúdo, carregaremos alguns arquivos que estão disponíveis na página de materiais complementares deste livro. O que veremos em seguida também pressupõe que você já saiba o que é um `data.frame` no R; caso tenha algumas dúvidas sobre isto, o Capítulo 1 é o melhor lugar para começar.

2.1 Importando dados no R

Só há um segredo para se aprender quando o assunto é carregamento de dados no R: cada tipo de arquivo geralmente requer uma solução específica de importação (mas, neste capítulo, veremos uma bastante geral). Além disso, também precisamos considerar dois maiores problemas. O primeiro deles é o limite de memória do computador, já que, no R, podemos carregar dados até o limite da memória RAM disponível.[1] Já o segundo diz respeito a lidar com erros de acentuação e de reconhecimento de caracteres em cada base que formos trabalhar, o que pode resultar em bases carregadas de forma inadequada – ou, até mesmo, erro no carregamento. O R oferece soluções simples para contornar estes problemas, que veremos na parte final do capítulo.

[1] Assim como outras linguagens de programação, o R precisa carregar informações na memória RAM para poder trabalhar com eles, daí o limite de armazenamento de bancos grandes de dados.

Antes de seguirmos, precisaremos instalar alguns pacotes que nos ajudarão a carregar dados[2]. Alguns destes pacotes são:

- `readxl`, para carregar planilhas do *Excel*;
- `readODS`, para carregar planilhas *Open Document*;
- `haven`, para importar dados do *SPSS* e *Stata*; e
- `rio`, para importar diversos tipos de dados.

Para instalar estes pacotes, use `install.packages("nome_do_pacote")`:

```
install.packages("readODS")
install.packages("readxl")
install.packages("haven")
install.packages("rio")
```

2.2 tidyverse

Reservamos um espaço especial para um pacote que é o centro deste livro: o `tidyverse`. Este é, na verdade, uma espécie de meta-pacote que abriga um conjunto de outros pacotes menores, específicos para diferentes tarefas. Em particular, o desenho do `tidyverse` segue princípios gerais, isto é, suas ferramentas são feitas com uma preocupação de consistência e de integração.

Teremos a chance de ver várias das funções do `tidyverse` durante o nosso percurso, mas, no que diz respeito a carregamento de dados, ele oferece duas funções que nos ajudarão bastante: `read_csv()` e `read_delim()`, ambas pertencentes ao pacote `readr`. É por esta razão que também usaremos e instalaremos o `tidyverse` antes de prosseguir (executar a linha a seguir pode levar vários minutos dado que, por baixo dos panos, vários pacotes serão instalados):

```
install.packages("tidyverse")
```

2.3 A mecânica da importação de arquivos

Temos várias formas de salvar informações em um computador. Podemos, por exemplo, escrever um texto no *Word* ou *Libre Office* e salvá-lo em um arquivo chamado `Meu texto.doc`, assim como podemos criar uma planilha no *Excel* e salvá-la no arquivo `Minha planilha.xls`.

[2] As ferramentas que usaremos aqui não são nem de longe as únicas, nem necessariamente as melhores, para carregar dados – na verdade, o próprio R já vem com algumas funções nativas para importação de dados. Nossa escolha aqui reflete mais nossa experiência trabalhando com o R e a filosofia mais geral deste livro: as funções que usamos são simples, flexíveis e, em geral, as mais rápidas.

O importante aqui é que da mesma forma que cada um destes programas serve para trabalhar com um tipo específico de arquivo, no R também precisaremos de ferramentas específicas para abrir diferentes tipos de arquivo. Às vezes, faremos isto usando funções diferentes. Em outros casos, apenas precisaremos dizer para o R como ele deve proceder – qual *encoding* ele deve usar, onde ficarão os nomes das variáveis, qual é o tipo de delimitar de texto que deverá ser usado, entre outros.

A primeira coisa que precisamos saber, portanto, é qual solução usar para cada tipo de arquivo. Há formas simples de identificar isso, mas elas pressupõe saber a extensão do arquivo que queremos abrir (as letras depois do ponto ao final do nome do arquivo, e.g., `.doc`, `.xlsx`, etc.), que indicam qual é o seu formato. No *Windows*, podemos descobrir a extensão de um arquivo simplesmente clicando com o botão direito do *mouse* em cima dele e, depois, na opção "Propriedades" no menu que será aberto; feito isto, a extensão do arquivo será exibida logo acima (no campo de texto destacado em azul).

Para orientação geral, a Tabela 2.1 exibe um resumo dos principais tipos de arquivos de dados, geralmente usados em análises, que aprenderemos a abrir neste capítulo com suas respectivas extensões – e funções e pacotes que usaremos para carregá-los no R.

Tabela 2.1: Tipos de arquivos, suas extensões e funções usadas para carregá-los no R

Arquivo	Extensão	Pacote	Função
Texto delimitado	`.txt`	`readr`	`read_delim`
Texto delimitado	`.csv`	`readr`	`read_delim`, `read_csv`
Planilha do Excel	`xls`, `xlsx`, `.ods`	`readxl`, `openxlsx`, `readODS`	`read_excel`, `read.xlsx`, `read.ods`
Banco de dados do SPSS	`.sav`, `.por`	`haven`	`read_sav`, `read_por`
Banco de dados do Stata	`.dta`	`haven`	`read_dta`
Banco de dados do SAS	`.sas7bdat`	`haven`	`read_sas`
R Data	`.Rda`	-	`load`
Apache Parquet	`.parquet`	`duckdb` e `DBI`	`dbConnect` e `tbl`

Apesar de parecer muita coisa, a mecânica geral de carregar dados é mais ou menos a mesma para qualquer tipo de arquivo: se aprendermos a usar uma solução, provavelmente saberemos usar as demais. A ideia básica, detalhada em seguida, é:

```
objeto <- nome_da_funcao("nome_do_arquivo.extensao", outros_argumentos...)
```

2.4 Importando arquivos

2.4.1 Arquivos de texto delimitado

Começaremos carregando um dos tipos de arquivos mais comuns no R: o `.csv`, de *comma-separated values*, ou valores separados por vírgulas. Além de simples, este formato é flexível (pode ser salvo também em arquivo com extensão `.tab`, `.txt`, etc.) e intuitivo: cada observação no banco (linha) é separada por quebra de parágrafo (nova linha) e cada variável (coluna) é separada por um caractere fixo (como ponto e vígula ou vírgula)[3]. É possível abrir diretamente estes arquivos com algum editor de texto simples para ver como eles são organizados, como mostra a Figura 2.1.

```
"Var1","Var2"
1,100
2,99
3,98
4,97
5,96
6,95
7,94
8,93
9,92
10,91
11,90
12,89
13,88
14,87
15,86
16,85
17,84
18,83
```

Figura 2.1: Exemplo de arquivo de texto delimitado

Como é possível notar, temos duas variáveis neste arquivo: "Var1" e "Var2". Cada linha é uma observação, e os valores de cada variáveis estão separados por uma vírgula. De forma geral, esta é a forma como dados são salvos neste tipo de arquivo – precisamos apenas saber qual é o separador das colunas (no caso, vírgula).

Para carregar este arquivo, podemos usar a função `read_delim` do pacote `readr` – parte do pacote `tidyverse`. Como seu nome sugere, a função serve para ler arquivos delimitados. O

[3]Uma opção interessante para arquivos grandes delimitados é a função *fread* do pacote data.table (Dowle e Srinivasan 2023), que não vamos abordar aqui.

procedimento é simples: passamos para a função o nome do arquivo, que deverá estar no diretório corrente de trabalho do `R` (ou passar o endereço do arquivo no computador), e o delimitador de colunas para o argumento `delim =`.

```
# Carrega o pacote tidyverse
library(tidyverse)

# Carrega os dados do arquivo "exemplo.csv"
meu_banco <- read_delim("exemplo.csv", delim = ",")
```

> ⚠ Diretório de trabalho
>
> O R só consegue carregar arquivos que estão no diretório de trabalho (para saber qual é este diretório, basta executar a função `getwd()` no console). Uma boa prática é criar um projeto com o RStudio na pasta onde estão os seus dados. Para tanto, basta ir em *File > New Project* e escolher a opção *Existing Directory* e clicar em *Create Project*.

O código acima já salva os dados do arquivo no objeto chamado `meu_banco`, que é um `data.frame`. Com isto, podemos usar a função `glimpse` do pacote `dplyr` (parte do `tidyverse`) para visualizar a estrutura do banco:

```
glimpse(meu_banco)
```

```
Rows: 100
Columns: 2
$ Var1 <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19~
$ Var2 <dbl> 100, 99, 98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, ~
```

Ou podemos usar a função `View` para visualizar os dados do banco.

```
View(meu_banco)
```

O que deverá abrir uma nova aba no RStudio semelhante a essa:

A função `read_delim` ainda pode ser adaptada para outros tipos de arquivos de texto delimitado, como `.txt` ou `.tab`; ou para abrir arquivos com outros delimitadores de colunas, como ponto e vírgula (`delim = ";"`) ou TAB (`delim = "\t"`, o que indica à função que as colunas são separadas por dois espaços simples). Os exemplos abaixo fazem exatamente isto.

```
banco1 <- read_delim("exemplo_ponto_virgula.csv", delim = ";")
banco2 <- read_delim("exemplo_texto.txt", delim = ",")
banco3 <- read_delim("exemplo_tabular.tab", delim = ",")
banco4 <- read_delim("exemplo_espacos.csv", delim = "\tab")
```

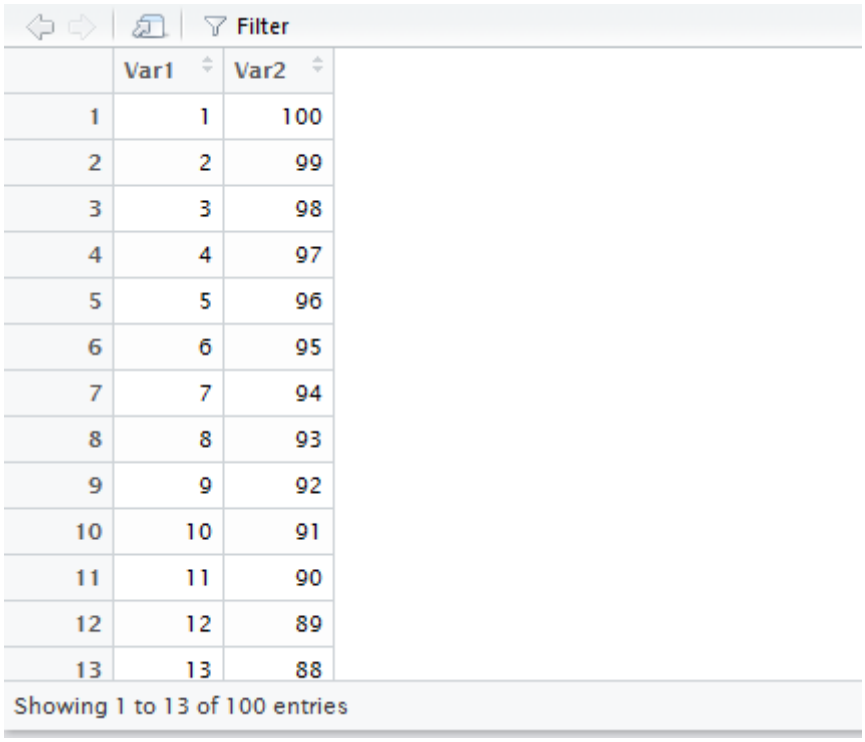


Figura 2.2: Usando a função View

Além de arquivos armazenados no computador, também podemos carregar arquivos na internet: no lugar do nome do arquivo, é só passar para a função o *link* de onde o arquivo está hospedado.[4].

```
banco5 <- read_delim("https://github.com/tidyverse/readr/raw/master/inst/extdata/mtcars.csv"
```

Além destas extensões e do argumento `delim`, a função `read_delim` também nos permite passar outras instruções para o `R` carregar um arquivo. Dentre estas, a mais útil é `skip`, que serve para indicar a partir de qual linha queremos iniciar o carregamento dos dados (que pode ser usada para pular linhas que não estão formatadas corretamente).

Na pasta de materiais complementares deste capítulo, temos um arquivo chamado `pessoas.csv`, que contém os nomes e as idades, salvas em duas variáveis, de algumas pessoas fictícias. Abrindo este arquivo com um editor de texto simples, veremos que o conteúdo dele está organizado de uma forma um pouco diferente do que já vimos anteriormente:

```
"Exemplo"

"NOME";"IDADE"
"Maria";45
"João";23
"Antônio";14
"Ana";9
"José";60
"Rosa";36
```

Figura 2.3: Arquivo pessoas.csv

Pela Figura 2.3, é possível notar que existe a palavra "Exemplo" em uma linha acima do restante do conteúdo do arquivo e que, além disso, esta linha tem apenas um campo – o texto "Exemplo". Vamos tentar carregar este arquivo com a função `read_delim`, que já vimos, para ver como o `R` lerá estes dados.

```
# Carrega o arquivo "pessoas.csv"
pessoas <- read_delim("pessoas.csv", delim = ";")
View(pessoas)
```

P resultado, como fica evidente, não é o que queríamos. Para corrigir isso, precisamos usar o argumento `skip` da função `read_delim` para pedir que ela carregue os dados pulando algumas

[4]Nem todas as funções de carregamento de arquivos que veremos suportam importação de arquivos da internet. Outro aspecto a notar é que `read_delim` e similares carregam apenas arquivos hospedados em servidores que não exigem autenticação – ou seja, que não exigem que você faça *login* para acessar um dado arquivo.

Figura 2.4: Arquivo pessoas.csv lido com read_delim

linhas (1, 2, 3, etc., linhas) – exatamente para pular aquele "Exemplo" e aquela linha em branco depois disso. Fazemos isto assim:

```
# Carrega o arquivo "pessoas.csv" pulando tres linhas
pessoas <- read_delim("pessoas.csv", delim = ";", skip = 3)
View(pessoas)
```

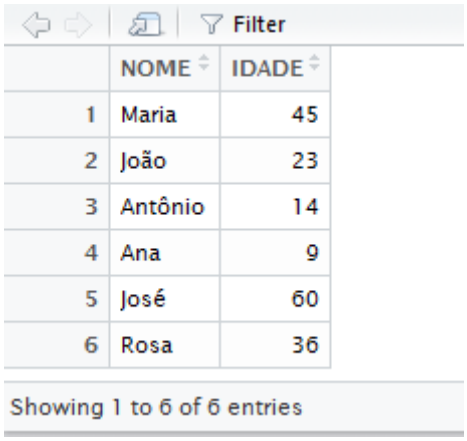


Figura 2.5: Arquivo pessoas.csv lido corretamente

> **!** Erros na importação de arquivos
>
> Como o exemplo do arquivos `pessoas.csv` ilustra, uma das principais fontes de erro na leitura de arquivos ocorre por especificação correta de como ler dados delimitados. Para evitar este tipo de problema, vale sempre abrir o arquivo que queremos carregar com um editor de texto para ver como ele está organizado.

`skip` não esgota as possibilidades da função `read_delim`. Ao contrário, ela possui diversos argumentos adicionais úteis para contornar problemas. Na Tabela 2.2, segue uma descrição de alguns deles (para ver outros, digite no console `help(read_delim)`).

Tabela 2.2: Argumentos da função `read_delim`

Argumento	Descrição	Uso
`quote`	Delimitador de campos textuais	`quote = "\""`
`col_names`	Passa novos nomes para as variáveis carregadas	`col_names = c("Nome1", "Nome2", ...`
`locale`	Muda as configurações de horário e acentuação	Veremos adiante.
`comment`	Carrega apenas linhas que não começam com o caractere especificado	`comment = "#"`
`trim_ws`	Remove espaços em branco no início e no fim de cada campo	`trim_ws = TRUE`
`col_types`	Especifica o tipo de cada variável	`col_types = "ccdi"`

2.4.2 Outros formatos

Uma vez que aprendemos como carregar arquivos com extensão `.csv`, é fácil carregar qualquer outro arquivo. O que veremos a seguir, portanto, são as funções e os pacotes mais comumente usados para carregar outros formatos de arquivo. De forma complementar, nas duas últimas seções aprendermos a lidar com os erros mais frequentes quando tentamos carregar algum arquivo e a exportar dados para arquivos dos mais diversos formatos.

2.4.3 Planilhas

Para abrir planilhas do Excel, com extensões `.xls` ou `.xlsx`, usamos a função `read_excel` do pacote `readxl`, que é semelhante à função `read_delim`. Exemplo:

```
# Carrega o pacote readxl
library(readxl)

# Carrega a planilha 'populacao_brasil.xls' na pasta do livro
dados <- read_excel("populacao_brasil.xls")
```

Novamente, a primeira coisa que passamos para a função é o nome do arquivo (ou, aqui também, o *link* de onde o arquivo está hospedado na internet) – na maioria dos casos, apenas isto é suficiente.

Também podemos passar argumentos opcionais para a função `read_excel`, tais como: `sheet`, que indica o número da planilha dentro do arquivo (`1` para a primeira, `2` para a segunda, e assim por diante); e `skip`, que diz quantas linhas a função deve pular para começar a ler o conteúdo do arquivo, exatamente como na função `read_delim`.

No exemplo a seguir, carregamos a segunda planilha do mesmo arquivo, pedindo também para a função começar a ler os dados a partir da primeira linha.

```
# Carrega a primeira planilha do arquivo pulando a primeira linha
dados <- read_excel("populacao_brasil.xls", sheet = 1, skip = 1)
```

O pacote `readxl`, entretanto, não serve para abrir planilhas feitas pelo *Open Office* (*OpenDocument Spreadsheet*, extensão `.ods`). Para carregar dados neste formato, existe um pacote específico: `readODS`. Basicamente, precisamos apenas carregá-lo e usar a função `read.ods` para carregar arquivos `.ods`:

```
# Carrega o pacote
library(readODS)

# Carrega a planilha 'populacao_brasil.ods' na pasta do livro
dados <- read.ods("populacao_brasil.ods", sheet = 1)
```

2.4.4 SPSS, Stata e SAS

Outros *softwares* de análise de dados possuem arquivos próprios para armazenar dados. Estes são os casos do *SPSS* (arquivos `.sav` e `.por`), *Stata* (arquivos `.dta`) e *SAS* (arquivos `.sas7bdat`), todos os três muito populares na academia e no mercado.

Para importar dados criados pelos *softwares* mencionados, recorremos ao pacote `haven`, que usa o código de outro pacote desenvolvido em `C`, o `ReadStat`, para fazer o trabalho. A título de

exemplo, vamos carregar um banco de dados de um *survey* realizado na Austrália para avaliar o impacto de privações de sono.[5] O uso do pacote é auto-explicativo.

```
# Carrega o pacote haven
library(haven)

# Carrega o arquivo
dados <- read_sav("sleep.sav")
```

Nos três casos, o pacote faz o trabalho de manter as informações originais dos arquivos ao máximo possível. Em arquivos do *SPSS*, isso inclui manter os *labels* originais das variáveis e os seus tipos (a função `read_sav` converte variáveis numéricas e categóricas para seus tipos respectivos no `R`).

Podemos verificar isto abrindo o objeto onde salvamos o arquivo do `nome_do_arquivo`, como na imagem abaixo – os *labels* aparecem logo abaixo do nome das variáveis.[6] Quando possível, para arquivos do *Stata* e do *SAS* o mesmo também ocorre.[7]

Às vezes, pode ser útil usar os *labels* diretamente, em vez de usar a codificação numérica das variáveis (e.g., "Masculino" em vez de 1, "Feminino" em vez de 2; isso é chamado também de máscara). Para isto, podemos usar a função `as_factor` do pacote `haven` para converter as variáveis para fatores, o que preserva os *labels* originais. O exemplo a seguir faz exatamente isto.

```
# Carrega o arquivo
dados <- read_sav("sleep.sav")

# Converte as variáveis para fatores
dados <- as_factor(dados)
```

2.4.5 JSON

Outro formato popular, ainda que pouco utilizado na academia, é o *JSON* (*JavaScript Object Notation*) – que pode ser encontrado cada vez mais em sites e API's, como a de Dados Abertos do Governo Federal.[8] O formato é bastante simples: chaves armazenam valores separados

[5] Detalhes do *survey* podem ser vistos em: http://spss.allenandunwin.com.s3-website-ap-southeast-2.amazonaws.com/data-files.html.

[6] A depender da versão de *software* proprietário utilizada no salvamento do arquivo e de sua estrutura, *labels* podem não ser carregados por padrão. Nesse caso, é possível passar o `data.frame` carregado para a função `as_factor` do pacote `haven` que, se possível, os incluirá explicitamente no objeto.

[7] As funções `read_dta`e `read_sas` possuem alguns argumentos adicionais, que podem ser úteis para corrigir acentuação e especificar outros detalhes.

[8] Disponível em https://dados.gov.br/.

Filter

	id Identification Number	sex sex	age age	marital marital status	edlevel highest education level achieved
1	83	0	42	2	2
2	294	0	54	2	5
3	425	1	NaN	2	2
4	64	0	41	2	5
5	536	0	39	2	5
6	57	0	66	2	4
7	251	0	36	1	3
8	255	0	35	2	5
9	265	1	NaN	2	5
10	290	1	41	2	5
11	418	1	NaN	2	5

Showing 1 to 12 of 271 entries

Figura 2.6: Base de dados do survey

por dois pontos (e.g., `{'valor' : 10, 23, 44}`). A estrutura pode conter muitos valores separados por vírgula e, também, chaves dentro de chaves, ou ainda chaves dentro de `[]` (*arrays*), o que dá flexibilidade para aramanzenar diferentes tipos de informação. Um exemplo fictício para armazenar informações de preferências partidárias de algumas pessoas:

```
[
  {
    "nome": "João",
    "idade": 35,
    "partido": "PT",
    "partidos_preferidos": ["PT", "PSB", "MDB"]
  },
  {
    "nome": "Maria",
    "idade": 32,
    "partido": "MDB",
    "partidos_preferidos": ["MDB", "PSDB", "DEM"]
  }
]
```

Para importar este tipo de arquivo no R, podemos usar a função `import` do pacote `rio` (abreviação de *R Imput/Output*; veremos outras utilidades dele adiante). O arquivo de exemplo vem da página da Transparência Internacional, da pesquisa *Corruption Perceptions Index 2015*.[9]

```
# Carrega o pacote rio
library(rio)

# Carrega o banco de dados do CPI 2015
cpi <- import("cpi-data.json")
```

2.4.6 R Data

Por fim, temos o formato nativo do R, o *R Data*, para salvar dados. O deixamos por último por um motivo especial: ele é o formato mais adequado para salvar dados no R, tanto por simplicidade quanto por eficiência. Em primeiro lugar, e diferentemente de formatos de texto como `.csv` e `.tab`, arquivos `.Rda` são binários – o que, traduzindo, permite que se guarde muito mais informação em menos espaço, inclusive forçando a compressão dos dados. Em segundo lugar, ler e salvar estes arquivos pelo R é geralmente mais rápido, e isto apesar da compressão.

[9] A página da pesquisa, bem como os dados e outros recursos, estão disponíveis em: https://www.transparency.org/cpi2015/.

Por fim, o formato salva e carrega objetos de um jeito mais intuitivo, como mostra o exemplo a seguir.

```
# Carrega os mesmos dados da CPI 2015, agora em formato .Rda
load("cpi2015.Rda")
```

Não é preciso carregar nenhum pacote, nem realizar nenhuma configuração: o objeto carregado vai direto para a memória do R, onde pode ser visto na aba `Environment` do *RStudio*. Outra vantagem do formato é que ele pode armazenar, de uma só vez, vários `data.frames` ou objetos quaisquer, facilitando a transposição de um projeto inteiro de um computador para outro – como quando temos precisamos analisar mais de uma base de dados.

> **RDS**
>
> Outro formato nativo no R é o RDS, que permite salvar e carregar arquivos usando um objeto para atribuição (`dados <- readRDS("dados.Rda")`). A diferença deste para o `Rdata` é que o RDS não permite salvar mais de um objeto, mas é igualmente rápido e atinge os mesmos níveis de compresão.

2.4.7 Outros formatos

Embora tenhamos visto como abrir os tipos de arquivos mais comuns – delimitados por texto, planilhas e de outros *softwares*, entre outros – existe uma infinidade de formas de se armazenar dados em arquivos e, muitas vezes, precisaremos recorrer a alguma ferramenta diferente das que estudamos. Quando isso acontecer, no entanto, há uma opção mais simples: o pacote `rio`.

Resumidamente, o `rio` funciona como uma espécie de canivete suíço para a importação e exportação de dados: basta passar para a função `import` o nome do arquivo que queremos abrir. A partir disto, o `rio` identifica o formato do arquivo que estamos tentando abrir e chama internamente a função e especificações mais adequadas para tanto. Entre outros, os arquivos suportados pelo pacote incluem: `.csv`, `.tsv`, `.fst`, `.psv`, `.fwf`, `.Rda`, `.Rds`, `.json`, `.dta`, `.sav`, `.xls`, `.mpt`, `.dif`, entre outros[10]. Exemplo de funcionamento da função `import`:

```
# Carrega o pacote rio
library(rio)

# Importa alguns dados
dados <- import("exemplo.csv")
dados2 <- import("sleep.sav")
```

[10]Para ver a lista completa de arquivos suportados ver https://github.com/leeper/rio

2.5 Exportando dados

Se importar dados para o `R` é algo fácil, como vimos, exportá-los é ainda mais. Tendo já alguns dados armazenados na memória do `R`, usamos funções semelhantes as de carregamento para exportá-los. Dentre estas, as principais são:

Tabela 2.3: Funções de exportação de dados {#tab-tabela33}

Arquivo	Extensão	Pacote	Função
Texto delimitado	`.txt`	`readr`	`write_delim`
Texto delimitado	`.csv`	`readr`	`write_delim`
Planilha do Excel	`.xlsx`	`openxlsx`	`write.xlsx`
SPSS	`.sav`	`haven`	`write_sav`
Stata	`.dta`	`haven`	`write_dta`
SAS	`.sas7bdat`	`haven`	`write_sas`
Outros	-	`rio`	`export`
Outros	-	`rio`	`convert`

Para exportar um `data.frame` qualquer, o procedimento básico é mais ou menos esse: o primeiro argumento que passamos para a função é o nome do objeto seguido do nome do arquivo que queremos criar entre aspas (não podemos esquecer de incluir a extensão do arquivo, que, no exemplo a seguir, é `.txt`).

```
# Carrega o pacote readr
library(readr)

# Cria um data.frame com duas variaveis
banco <- data.frame(x = 1:10, y = 1:10)

# Exporta ele para um arquivo .txt
write_delim(banco, "banco.txt")
```

Exemplos das outras funções de exportação:

```
# Outros pacotes
library(haven)
library(rio)

# Exporta para .sav
write_sav(banco, "banco.sav")
```

```
# Exporta para .dta
write_dta(banco, "banco.dta")

# Exporta para .json (e' preciso declarar 'file =')
export(banco, "banco.json")
```

Ainda usando o pacote `rio`, também podemos converter diretamente um arquivo de um formato para outro, o que nos poupa o trabalho de, primeiro, ler o arquivo para, então, exportá-lo. Como exemplo, vamos converter o arquivo `exemplo.csv`, que está na pasta de materiais complementares deste livro, para `.sav`, formato do SPSS:

```
# Converte o arquivo 'exemplo.csv' para .sav
convert("exemplo.csv", "exemplo.sav")
```

Para esta função, tudo o que precisamos fazer é passar o nome, ou o endereço com o nome, do arquivo que queremos converter e, como segundo argumento, o nome do arquivo que queremos criar – com a nova extensão. A depender do tamanho do arquivo, em poucos segundos a conversão é concluída. Mais tipos de conversão que a função `convert` executa podem ser vistos digitando `?convert` no console.

2.6 Lidando com erros

Aprender a usar funções adequadas para importar diferentes tipos de arquivo cobre boa parte do que precisamos para trabalhar com nossos dados no R, mas não tudo. Com frequência, usamos a ferramenta adequada e, mesmo assim, obtemos algum erro: o arquivo não abre, o R trava, ou ainda os dados abrem desconfigurados. Este tipo de coisa raramente é coberto em materiais didáticos, apesar de ser importante termos algumas noções básicas de como identificar – e de como contornar – erros na importação de dados. É justamente isso o que abordamos nesta seção.

2.6.1 Especificação do delimitador

Em arquivos delimitados de texto, talvez o erro mais comum é o de especificar de forma errada o delimitador: passar uma vírgula quando ele é, na verdade, ponto e vírgula; ou passar ponto e vírgula quando ele é outra coisa. Aqui o truque é quase banal: tentar abrir o arquivo com um editor de texto simples[11] para olhar os dados. Na maioria das vezes, isto já permite localizar o identificador adequado. O problema desta solução é que isto pode não dar certo se o arquivo for muito grande (e o editor de texto não conseguir abri-lo).

[11] por exemplo: bloco notas, notepad++, sublime e outros

Outra solução é ir na tentativa e erro. Por exemplo:

```
# Se isto nao der certo...
banco <- read_delim("exemplo_ponto_virgula.csv", delim = ",")

# Tentamos isto...
banco <- read_delim("exemplo_ponto_virgula.csv", delim = "\tab")

# E se tambem nao der, tentamos isto
banco <- read_delim("exemplo_ponto_virgula.csv", delim = ";")
```

2.6.2 Células vazias

Alguns arquivos às vezes vêm com células vazias, isto é, com informações não preenchidas (como *missings*), e isto pode resultar em erros. Em geral, isto ocorre mais em arquivos de texto delimitados, mas as funções que mostramos aqui para abri-los (`read_delim`, principalmente) nos dão notificações sobre estes erros. Os dados são carregados normalmente, mas ficamos sabendo onde procurar lacunas na base.

2.6.3 Problemas de acentuação

Outro problema comum para quem trabalha com bancos de dados que contêm informações textuais (nomes, endereços, etc.) é a acentuação. Volta e meia importamos um arquivo com `Ã` ou `Â` que são exibidos como `¢` e `Ă` no lugar.

Explicar por que isto acontece foge muito do escopo deste livro, mas é útil entender que cada sistema possui um conjunto de caracteres válidos para se escrever texto: em português, temos alguns acentos; em inglês, não. Assim, quando informações escritas usando um conjunto de caracteres particular, que chamamos de *encoding*, é trasposto para outro conjunto, coisas como estas ocorrem. E trocar de sistema operacional, abrir arquivos criados por um *software* em outro, entre outros, são situações onde isto pode acontecer.

Em português, usamos principalmente os *encodings* `UTF-8` e `latin1` (mas existem outros, alguns mais específicos) e, portanto, nossa primeira tentativa de corrigir estes erros é passando estes *encodings* para as funções que usamos para carregar dados que possam conter acentos usados em português. No caso da função `read_delim`, isto seria feito da seguinte forma:

```
# Caso o arquivo 'exemplo.csv' tivesse erro de encoding, tentariamos...
dados <- read_delim("exemplo.csv", delim = ",", locale = locale(encoding = "UTF-8"))

# Ou tentariamos...
dados <- read_delim("exemplo.csv", delim = ",", locale = locale(encoding = "latin1"))
```

Às vezes, isto não resolve: o *encoding* do arquivo não é nenhum dos dois. Para a nossa sorte, o pacote `readr` possui uma função chamada `guess_encoding` que tenta descobrir o *encoding* de um arquivo. Caso `UTF-8` e `latin1` não sirvam, portanto, tente o seguinte:

```
library(readr)
guess_encoding("exemplo.csv")
```

```
# A tibble: 1 x 2
  encoding confidence
  <chr>         <dbl>
1 ASCII             1
```

E aqui vemos que o *encoding* do arquivo `exemplo.csv`, que já carregamos antes, é provavelmente ASCII (um tipo de *encoding* com suporte para inglês, sem acentos).

2.6.4 Erros humanos

Neste ponto, precisamos falar de erros humanos: digitar errado o nome de um arquivo, passar o local errado de onde o arquivo está, usar uma função que abre um tipo de arquivo para tentar abrir arquivos de outro formato, entre outros. Mesmo parecendo algo trivial, tanto pessoas aprendendo R quanto outras experientes cometem este tipo de erro toda hora. Nosso alerta final, portanto, é: certifique-se de ter usado a função correta, de não ter digitado nada errado e de garantir de que o endereço do arquivo (ou o diretório corrente do R) existe.

2.7 Bases muito grandes

O R possui uma grande limitação em relação ao carregamento de dados: por armazenar informações na memória RAM do computador, e não no disco rígido, ele não suporta dados muito pesados, isto é, bases mais pesadas do que a capacidade de memória do seu computador. Por isso, a placa de RAM do seu computador (8gb, ou 16gb, etc.) é quem dita o tamanho dos arquivos que podemos carregar.[12]

Caso você tenha uma base de dados muito grande, que excede em tamanho a memória RAM do seu computador, será necessário usar outras soluções para importá-la. Há pacotes no R que fornecem algumas soluções alternativas de importação, mas não os abordaremos aqui – são pouco utilizadas e têm limitações de integração com outras ferramentas que vimos ou

[12]Nas versões mais recentes do *RStudio*, é possível ver a memória RAM disponível e já usada pela sua sessão do R no canto superior direito da tela, na aba *envorinment*. Para saber mais sobre as abas do *RStudio*, ver o [capítulo -#sec-cap1].

que ainda veremos.[13] Em vez disso, seguiremos o mote geral deste livro: veremos um par de ferramentas, o pacote `DBI` e o pacote `duckdb`, que nos dá uma solução simples e versátil para carregar e manipular dados de qualquer tamanho.

2.7.1 Pacote DBI

O `DBI` é uma interface para conectar o R a bancos de dados relacionais como o MySQL, o Postgres, o SQLite, entre outros.[14] Podemos pensar no `DBI` da seguinte forma: em vez de carregar e manipular dados que não cabem na memória do computador, o `DBI` tira essa tarefa do R e a delega para um banco de dados relacional, que é capaz de lidar com arquivos muito grandes.

Podemos instalar o `DBI` com o nosso conhecido `install.packages` e, depois, carregá-lo com `library`:

```
install.packages("DBI")
library(DBI)
```

No lugar de usar alguma função `read_`, o carro-chefe do `DBI` é a função `dbConnect`, que serve para conectar o R a um banco de relacional. A razão de usarmos esse procedimento é simples: bancos de dados, no mais das vezes, não são arquivos que existem localmente, como uma planilha de Excel; antes, são servidores que armazenam e gerenciam informações – o que queremos fazer, portanto, é nos conectarmos a esse servidores para poder passar a ele instruções, via R, de como manipular os dados que estão armazenados nele. Um exemplo genérico de como usaríamos `dbConnect`, que será detalhado na sequência:

```
# Conecta o R a um banco de dados relacional
con <- dbConnect(duckdb::duckdb())
```

2.7.2 DuckDB

A maioria dos sistemas de gerenciamento de bancos de dados, como MySQL e Postgres, rodam em servidores na internet e, além disso, dependem que instalemos *softwares* específicos, chamados de *drivers*, para que o R se conecte a eles. Há alguma exceções a esta regra geral, no entanto. Uma delas é o DuckDB, um sistema de gerenciamento de banco de dados que roda

[13] Entre outros, vale checar o `ff` (https://CRAN.R-project.org/package=ff) e o `bigmemory` (https://CRAN.R-project.org/package=bigmemory), que permitem manipular dados diretamente do disco, criando apenas atalhos na memória.

[14] Fugiria muito do escopo do livro abordar bases relacionais, tópico que, por si só, é complexo e que antecede em muito o desenvolvimento do próprio R. Recomendados, no entanto, a leitura do capítulo 21 do livro de Wickham, Çetinkaya-Rundel, e Grolemund (2023) para quem quiser uma introdução geral e intuitiva ao tema.

localmente, isto é, no seu computador, e que não precisa de nenhum *driver* adicional para ser usado no R.

O DuckDB é um banco de dados relativamente novo, mas que tem ganhado popularidade por ser rápido para tarefas típicas de análise de dados, como a leitura e manipulação de colunas com até mesmo centenas de milhões de linhas. Especialmente útil, o DuckDB contém funcionalidades para importação de grandes arquivos, como arquivos de texto delimitados; arquivos de Excel; e arquivos no formato `parquet`, outro formato que discutiremos em seguida. Por todas essas razões é que, neste livro, sugerimos o uso do DuckDB para fazer o carregamento e manipulação de arquivos muito grandes, que não poderiam ser carregados diretamente na memória do computador via R.

Para instalar o DuckDB, não precisamos de nada além de `install.packages`:

```
install.packages("duckdb")
library(duckdb)
```

Com o pacote instalado, para criar e nos conectarmos a um banco de dados DuckDB, que fará o carregamento propriamente de arquivos muitos grandes, usamos a linha que já vimos:

```
con <- dbConnect(duckdb::duckdb())
```

Neste código, o argumento `duckdb::duckdb()` serve para estabelecer que a função `dbConnect` deverá criar e se conectar a um banco de dados DuckDB – com isso, já temos a infraestrutura necessária em ação para carregar arquivos muito grandes. Imagine, por exemplo, que tenhamos um arquivo `CSV` com 8gb de tamanho chamado `exemplo.csv`. Para carregá-lo, passamos o objeto `con` criado há pouco para a função `tbl` do pacote `dplyr` (parte do `tidyverse`), que usaremos para importar os dados:

```
# Carregamos o pacote tidyverse
library(tidyverse)

df <- tbl(con, "exemplo.csv")
```

O código acima é similar ao que usamos para carregar outros tipos de arquivos: passamos o endereço do arquivo que queremos carregar para a função `tbl`, que serve para ler uma tabela a partir de um banco de dados relacional ao qual nos conectamos, e criamos o objeto `df`, que armazenará o resultado dessa tabela. Na maioria das vezes, sequer precisamos especificar o delimitador de colunas, pois o DuckDB é capaz de identificá-lo automaticamente.[15]

[15] Alternativamente, é possível usar a função `duckdb_read_csv` para importar o arquivo. Para mais detalhes, vale consultar a documentação do pacote (Muhleisen, Raasveldt, e DuckDB Contributors 2020).

O processo de importação de dados com `DBI` e `duckdb` é mais ou menos esse, exceto por um detalhe: o arquivo `exemplo.csv` não foi efetivamente carregado na memória do computador; em vez disso, o que temos é um atalho para o arquivo que será manipulado pelo banco de dados relacional criado com o DuckDB. Desse modo, acionamos o DuckDB para que ele carregue o arquivo `exemplo.csv` e nos dê um atalho para manipulá-lo a partir do disco rígido. Se quisermos pré-visualizar o conteúdo do arquivo importado via DuckDB, basta executar o objeto `df` no console:

```
df
```

```
# Source:   table<pessoas> [10 x 2]
# Database: DuckDB v1.0.1-dev1922 [fmeireles@Linux 6.9.5-200.fc40.x86_64:R 4.4.0/:memory:]
   nome     idade
   <chr>    <dbl>
 1 João        35
 2 Maria       32
 3 José        28
 4 Ana         31
 5 Pedro       29
 6 Mariana     27
 7 Carlos      33
 8 Juliana     30
 9 Fernando    26
10 Luana       34
```

O resultado dessa execução exibe um sumário, com as primeiras linhas e algumas colunas do arquivo, para facilitar a nossa consulta. Vale notar também algo importante: logo na segunda linha do *output* do R, há o trecho `Database: DuckDB ...`, que indica que a base que estamos lendo está em um banco relacional DuckDB que roda a partir do nosso computador.

2.7.3 Arquivos `parquet`

Além de carregar arquivos delimitados, o DuckDB também é capaz de carregar arquivos no formato `parquet`, um formato estruturado de armazenamento de dados orientado por colunas, que é especialmente útil para tarefas de análises de dados.[16]

Como exemplo da potencialidade dos pacotes `DBI` e `duckdb`, carregaremos como exemplo a base de microdados de pessoas do Censo de 2010, disponibilizada na internet em formato `parquet` no repositório do pacote de R censobr (Pereira e Barbosa 2023).[17]

[16] Para mais detalhes sobre o formato `Apache parquet`, ver: https://parquet.apache.org/.

[17] Todos os arquivos em `parquet` do Censos realizados pelo IBGE divulgados pelo pacote `censobr` podem ser encontrados no seguinte endereço: https://github.com/ipeaGIT/censobr/releases/tag/v0.2.0.

```
censo <- tbl(con, "'2010_population_v0.2.0.parquet'")
```

Isso feito, podemos pré-visualizar as informações do objeto como fizemos antes[18]:

```
censo
```

```
# Source:   SQL [?? x 251]
# Database: DuckDB v1.0.1-dev1922 [fmeireles@Linux 6.9.5-200.fc40.x86_64:R 4.4.0/:memory:]
   code_muni code_state abbrev_state name_state code_region name_region
   <chr>     <chr>      <chr>        <chr>      <chr>       <chr>
 1 1100015   11         RO           Rondônia   1           Norte
 2 1100015   11         RO           Rondônia   1           Norte
 3 1100015   11         RO           Rondônia   1           Norte
 4 1100015   11         RO           Rondônia   1           Norte
 5 1100015   11         RO           Rondônia   1           Norte
 6 1100015   11         RO           Rondônia   1           Norte
 7 1100015   11         RO           Rondônia   1           Norte
 8 1100015   11         RO           Rondônia   1           Norte
 9 1100015   11         RO           Rondônia   1           Norte
10 1100015   11         RO           Rondônia   1           Norte
# i more rows
# i 245 more variables: code_weighting <chr>, V0001 <chr>, V0002 <chr>,
#   V0011 <chr>, V0300 <dbl>, V0010 <dbl>, V1001 <chr>, V1002 <chr>,
#   V1003 <chr>, V1004 <chr>, V1006 <chr>, V0502 <chr>, V0504 <chr>,
#   V0601 <chr>, V6033 <dbl>, V6036 <dbl>, V6037 <dbl>, V6040 <chr>,
#   V0606 <chr>, V0613 <chr>, V0614 <chr>, V0615 <chr>, V0616 <chr>,
#   V0617 <chr>, V0618 <chr>, V0619 <chr>, V0620 <chr>, V0621 <chr>, ...
```

Neste exemplo, dá para notar que a base, muito grande, não é carregada inteiramente, o que é indicado logo na primeira linha do *output* em `Source: table<data/2010_population_v0.2.0.parquet> [?? x 251]`, o que indica que a base tem 251 colunas e um número indeterminado de linhas.

2.7.4 Outros bancos relacionais

Bancos de dados relacionais são comuns em diferentes áreas, e há diferentes alternativas específicas para análise de dados. Para quem já tem alguma experiência com eles, o `DBI` oferece uma interface unificada para integrar o R a outros bancos relacionais, como o MySQL, o Postgres, o SQLite, para ficar apenas entre alguns mais populares.[19] O procedimento é similar ao que

[18] Cabe notar apenas que usamos `'` dentro das aspas com o nome do arquivo. Esse é um detalhe de como o pacote `dplyr` funciona para esse tipo de arquivo.

[19] Para mais detalhes sobre como usar o `DBI` com outros bancos relacionais, ver: https://db.rstudio.com/.

vimos para o DuckDB: precisamos instalar o *driver* do banco de dados que queremos usar e, depois, nos conectarmos a ele com a função `dbConnect`.

Talvez a segunda alternativa mais fácil, e de menor custo de configuração depois do DuckDB, seja o SQLite, um banco de dados relacional que roda localmente, no seu computador, e que não precisa de nenhum *driver* adicional para ser usado no R.[20] Para usá-lo, precisamos instalar o pacote `RSQLite` e, depois, nos conectarmos a ele com a função `dbConnect`:

```
install.packages("RSQLite")
library(RSQLite)

con <- dbConnect(RSQLite::SQLite())
```

> **Drivers**
>
> Para usar o `DBI` com outros bancos relacionais, precisamos instalar o *driver* do banco de dados específico que queremos usar. Por exemplo, para usar o `DBI` com o MySQL, precisamos instalar o pacote `MariaDB` antes; para usar com o Postgres, precisamos instalar o pacote `RPostgres`; e assim por diante.

Esse é apenas mais uma das inúmeras possibilidades de uso do `DBI`. Para além da função `dbConnect`, o pacote também possui funções para criar tabelas, inserir e atualizar dados, entre outras, e existem várias opções de integração com outros tipos de bancos relacionais e formatos – dos mais tradicionais, como o MySQL, a soluções como o Google Big Query, que permite o armazenamento e a análise de grandes volumes de dados na nuvem.[21]. Em todo o caso, a dupla `DBI + duckdb` é não só suficiente como, com frequência, uma das mais indicadas para resolver a maioria dos problemas de carregamento de dados encontrados na prática.

2.8 Resumo do capítulo

Neste capítulo, cobrimos a mecânica básica da importação de dados no R. Começamos com os tipos de arquivos mais comuns, como arquivos de texto delimitados, planilhas e arquivos de outros *softwares* de análise de dados, como o SPSS e o Stata. O truque geral é: cada formato de arquivo demanda um tipo específico de solução, e o R possui funções específicas para cada um deles. Depois, vimos como carregar arquivos menos comuns, como arquivos `JSON` e arquivos `parquet`. Por fim, vimos como carregar arquivos muito grandes, que não cabem na memória do computador, usando o pacote `DBI` e o sistema de gerenciamento de bancos relacionais DuckDB disponível via `duckdb`.

[20] Para mais detalhes sobre o SQLite, ver: https://www.sqlite.org/index.html.
[21] Para mais detalhes sobre como usar o `DBI` com outros bancos relacionais, ver: https://db.rstudio.com/.

2.9 Indo além

Importar dados é uma tarefa complexa, e muitas vezes uma receita pronta, como as que vimos aqui, não servirá. Conforme você aprenda mais sobre o R e comece a trabalhar em projetos específicos, é possível que se depare com a necessidade de buscar outras soluções, ou mesmo de ter que criar ou adaptar alguma para uso próprio. Neste sentido, vale a pena conhecer alguns pacotes que podem ser úteis para importar dados de formatos mais específicos.

...

Há diversos pacotes em R que servem para justamente importar dados de diferentes fontes diretamente no R. Alguns deles, que podem ajudar principalmente a obter dados sobre o Brasil, são:

- `censobr`
- `congressbr`
- `electionsBR`
- `PNADcIBGE`
- `sidrar`

Exercícios

> Arquivos necessários
>
> Para realizar estes exercícios, será necessário baixar os arquivos que estão na pasta de materiais complementares deste livro e salvá-los na pasta de trabalho do R.

1. Carregando arquivos simples I

Carregue o arquivo `pessoas.csv`, que contém informações de algumas pessoas fictícias, e salve seu conteúdo no objeto `pessoas`. Depois de carregado os dados, use a função `head` para pré-visualizar as primeiras linhas do `data.frame` `pessoas`. Use comentários para explicar como você descobriu a forma correta de carrergar o arquivo.

2. Carregando arquivos simples II

Carregue um arquivo `pesquisa_satisfacao.txt`, que contém os resultados de uma pesquisa de satisfação de cinco clientes de uma loja (as notas variam de 1 a 5). Salve o conteúdo do arquivo no objeto `satisfacao`. Depois de carregado os dados, use a função `head` para pré-visualizar as primeiras linhas do `data.frame` `satisfacao`. Use comentários para explicar como você descobriu a forma correta de carrergar o arquivo.

3. Carregando arquivos simples III

Carregue o arquivo `casos_registrados.csv`, que contém um relatório de casos de uma patologia específica em bairros da zona norte do Rio de Janeiro, e salve or esultado no objeto `casos`. Certifique-se de carregar apenas o conteúdo das colunas `Bairro`, `Casos`, e `Internações`. Use comentários para explicar qual dificuldade você encontrou ao carregar o arquivo e como a contornou.

4. Carregando arquivos delimitados

Para este exercício, carregue dados do Censo de 1872, o primeiro realizado no Brasil, disponíveis no seguinte link:[22]

- https://raw.githubusercontent.com/izabelflores/Censo_1872/main/Censo_1872_dados_tidy_versao2.csv

Salve o resultado no objeto `censo`. Depois de carregado o arquivo, use a função `head` para pré-visualizar as primeiras linhas do `data.frame censo`.

5. Carregando arquivos de outros formatos I

É comum que dados de pesquisas de opinião, i.e. *surveys*, sejam armazenados em arquivos `SPSS` ou `Stata`, criados pelos *softwares* de mesmo nome. Neste exercício, sua tarefa será carregar um arquivo `SPSS` e um `Stata` com dados da pesquisa do *World Values Survey*, principal fonte de dados sobre valores, crenças e comportamentos das populações de diferentes países. Os dados do Brasil estão no arquivo `wvs.sav` e `wvs.dta`. Carregue ambos os arquivos e salve os resultados nos objetos `wvs_spss` e `wvs_stata`, respectivamente.[23]

6. Carregando arquivos de outros formatos II

Algo para a prática de pesquisas replicáveis – isto é, pesquisas cujos resultados podem ser reproduzidos por outras pessoas – é a disponibilização de dados brutos. Internacionalmente, a principal plataforma para a disponibilização de dados para replicação é o Harvard Dataverse, uma plataforma de repositório de dados de pesquisa que permite que pesquisadores publiquem, compartilhem, documentem e citem dados.[24]

Neste exercício, sua tarefa será obter e carregar no R um arquivo de dados brutos de um repositório do Harvard Dataverse. Em particular, baixaremos dados de anúncios do Airbnb,

[22] Esses dados foram organizados e disponibilizados por Izabel Flores e estão disponíveis em (baixe e salve o arquivo para a pasta de trabalho do R): https://github.com/izabelflores/Censo_1872.
[23] Os dados originais podem ser obtidos em https://www.worldvaluessurvey.org/WVSDocumentationWV7.jsp.
[24] Para mais detalhes sobre o Harvard Dataverse, ver: https://dataverse.harvard.edu/.

serviço de hospedagem que permite que pessoas anunciem e reservem acomodações em todo o mundo, em pequenas cidades do Brasil. Os dados estão neste repositório, com uma planilha de Excel para cada uma das cidades listadas. Escolha uma das cidades, baixe seu arquivo correspondente e carregue-o no R. Salve o resultado no objeto `airbnb` e use a função `head` para pré-visualizar as primeiras linhas do `data.frame`.

7. Carregando microdados administrativos

Diferentes órgãos públicos no Brasil disponibilizam seus microdados administrativos para acesso. Um deles é o INEP, que tem uma página dedicada para *download* de bases de dados com informações sobre os exames e pesquisas que conduz.[25]

Neste exercício, sua tarefa será obter e carregar no R alguns microdados do Censo da Educação Superior de 2022, levantamento anual do INEP que coleta informações sobre as instituições de ensino superior, cursos de graduação e estudantes no Brasil todo. Os dados que deverão ser baixados são *os referentes ao ano de 2022*, que poderão ser baixados desta página do INEP. Os dados estão em formato compactado (zipado) e deverão ser descompactados no diretório local do R antes de serem carregados. Uma vez descompactados, carregue o arquivo `MICRODADOS_ED_SUP_IES_2022.CSV` e salve o resultado no objeto `alunos`. Feito isso, reporte o seguinte:

- O nome das colunas do `data.frame alunos`;
- O número de linhas e de colunas do `data.frame alunos`.

[25] A página de *download* de dados do INEP está disponível em https://www.gov.br/inep/pt-br/acesso-a-informacao/dados-abertos/microdados.

3 Manipulação

É raro trabalharmos numa base de dados que já esteja, digamos, pronta. Ter de transformar valores de uma variável, excluir outros, remover ou incluir observações é algo quase compulsório em uma análise de dados. Apesar disso, esse processo essencial em qualquer pesquisa acadêmica ou análise de dados é algo quase ausente em livros de metodologia – normalmente já pressupondo que alguma base de dados existe e que não é necessário modificá-la.

Nesse sentido a nossa abordagem é diferente. Assumimos que manipular e limpar bases de dados são etapas fundamentais em uma análise. Também acreditamos que essas etapas não precisam consumir tanto tempo, tampouco exigir esforço manual repetitivo. Mostraremos neste capítulo que é possível realizar diferentes operações de transformação em uma base de dados com poucas ferramentas; e que, como vantagem desse método de manipulação, qualquer pessoa poderá replicar nossos procedimentos desde os menores detalhes – normalmente não documentados.

O trajeto que faremos até o fim deste capítulo também envolverá aprender a pensar numa base de dados de forma um pouco diferente do que estamos habituados. Simplesmente empilhar células numa planilha, ou criar tabelas estruturadas de forma arbitrária, não será suficiente. Isso é o que cobrimos na Seção 3.1. A partir daí, aprenderemos na Seção 3.2 a usar quatro das mais comuns operações de manipulação de dados: *filtrar*, *selecionar*, *modificar* e *agrupar/resumir*, aplicáveis até mesmo em bases com milhões de linhas ou que não cabem na memória RAM do computador. Finalmente, cobriremos como combinar e cruzar bases de dados na seção Seção 3.3.

Neste capítulo, trabalharemos com coisas como `tibbles`, importação de dados e manipulação de vetores; caso não tenha familiaridade com estes tópicos, veja o Capítulo 1 e o Capítulo 2. Para acompanhar o material a seguir, também será necessário usar os pacotes `dplyr` e `tidyr`, ambos partes do `tidyverse` voltados para a manipulação de dados, que precisaremos carregar com o comando `library(tidyverse)`:

```
library(tidyverse)
```

3.1 Tidy data

Qualquer pessoa minimamente familiarizadas com metodologia de pesquisa sabe que bases com problemas podem invalidar uma análise: esquecer de deflacionar séries de preços em análises

Tabela 3.1: Exemplos de bases de dados com votos de partidos políticos

(a) Tidy

Partido	Votos
A	234
B	451
C	200

(b) Não-tidy

A	B	C
234	451	200

históricas, por exemplo, é um problema porque sabemos que R$ 100,00 de hoje não vale o mesmo que R$ 100,00 em 1998. Ainda assim, duas ou mais pessoas minimamente familiarizadas com análise de dados podem divergir sobre como estruturar uma base. Imagine, por exemplo, que tenhamos um pequeno banco de dados com a quantidade de votos de partidos políticos fictícios. Uma forma razoável de organizar estes dados seria assim:

Se repararmos bem, ambas as formas de disposição dos dados são consistentes. Cada linha ou coluna contém apenas o mesmo tipo de informação – partido ou votos – e é fácil identificar a estrutura de cada base: na primeira, cada linha indica os atributos de um único partido; já na segunda, cada coluna indica a votação do respectivo partido. Temos exemplos destes dois métodos inclusive em bases que já carregamos no Capítulo 2: o arquivo `exemplo.csv` está organizado da primeira forma, enquanto que o arquivo `populacao_brasil.xls` está organizado da segunda.

Ainda que a segunda forma seja útil em determinadas aplicações, daqui até o final do livro trabalharemos com a primeira forma, popularizada pelo estatístico e desenvolvedor de `R` Hadley como *tidy data*, ou *dados arrumados* (Wickham 2014). Nesta estrutura, basicamente três regras são seguidas:

1) Cada variável é uma coluna;
2) Cada observação é uma linha;
3) Cada valor está numa única célula.

Exposto dessa forma, não é óbvio o significado de cada uma dessas regras. Em primeiro lugar, o conjunto desses princípios nos indica que as colunas são compostas por variáveis, que são simplesmente atributos de cada observação. Voltando à nossa tabela inicial, "votos" é uma variável e, portanto, está em uma coluna, pois representa a votação de cada partido. O mesmo vale para o nome do partido, na coluna "partido", e também para outros atributos do partido, como número de filiados, número de deputados, entre outros. Em outras palavras, uma variável é uma característica de uma observação e, portanto, deve estar em uma coluna.

Em segundo lugar, as linhas indicam os indivíduos ou observações que temos – cada um dos partidos na nossa base, como na tabela anterior. Podemos pensar na observação como a nossa *unidade de análise*. Poderíamos ter observações repetidas de um mesmo indivíduo ao longo do tempo, como o partido A em 2003 e o partido A em 2004; neste caso, o ano seria uma

nova variável na nossa base. Por fim, cada atributo de cada unidade de análise – partidos no nosso caso – está em uma única célula, o que significa que não temos informações repetidas ou ausentes. É o que vemos na tabela anterior: cada partido tem apenas um nome, uma votação e uma sigla.

A Tabela 3.2, a seguir, resume a ideia por detrás das regras de *tidy data*:

Tabela 3.2: Princípios de *tidy data*

Regra	Significado	Exemplo
Cada variável é uma coluna	Cada coluna deve armazenar apenas informações de um mesmo atributo	Votação dos partidos
Cada observação é uma linha	Cada linha representa uma unidade de análise ou um indivíduo	Partido A
Cada valor está numa única célula	Cada célula contém apenas um valor	234

3.1.1 Espalhar e reunir

Duas operação resumem tudo o que precisamos fazer para estruturar uma base no formato *tidy*: alongar e reunir. No `R`, essas operações podem ser feitas usando o pacote `tidyr`, com a função `pivot_longer` servindo para alongar valores de uma coluna e, por sua vez, a função `pivot_wider` para reunir numa única coluna valores dispersos em várias.

Para aprendermos a usar ambas as funções, trabalharemos com um banco de dados contendo informações sobre o número de homicídios ocorridos anualmente nos estados brasileiros entre 2000 e 2009, conforme disponibilizado pelo Ipeadata[1] com base nos dados originais do Datasus[2]. Os dados estão na pasta de materiais complementares deste livro em uma planilha de Excel chamada `homicidios_uf.xls`. Para carregá-la (ver o Capítulo 2), usaremos a função `read_excel` do pacote `readxl`:

```
library(readxl)
homic <- read_excel("homicidios_uf.xls")
```

Com a função `head`, podemos visualizar as primeiras observações deste banco.

[1] Estes dados podem ser obtidos diretamente pelo *website* do Ipeadata: http://www.ipeadata.gov.br/.
[2] O *website* do Datasus, do qual estes dados também podem ser obtidos, é: http://datasus.saude.gov.br/.

```
head(homic)
```

```
# A tibble: 6 x 13
  Sigla Codigo Estado   `2000` `2001` `2002` `2003` `2004` `2005` `2006` `2007`
  <chr> <chr>  <chr>     <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 AC    12     Acre        108    122    151    135    115    125    155    133
2 AL    27     Alagoas     724    836    989   1041   1034   1211   1617   1839
3 AM    13     Amazonas    559    478    512    561    523    598    697    711
4 AP    16     Amapá       155    184    181    190    173    196    203    171
5 BA    29     Bahia      1223   1573   1735   2155   2255   2823   3276   3608
6 CE    23     Ceará      1229   1298   1443   1560   1538   1692   1793   1936
# i 2 more variables: `2008` <dbl>, `2009` <dbl>
```

É possível perceber que a estrutura do banco `homic` não é *tidy*: temos várias colunas com alguns anos (que são atributos do momento em que um indivíduo é observado), em vez de uma coluna com número de homicídios e uma coluna para anos. O que gostaríamos de ter, portanto, seria algo mais ou menos assim:

```
# A tibble: 6 x 5
  Sigla Codigo Estado Ano   Homicidios
  <chr> <chr>  <chr>  <chr>      <dbl>
1 AC    12     Acre   2000         108
2 AC    12     Acre   2001         122
3 AC    12     Acre   2002         151
4 AC    12     Acre   2003         135
5 AC    12     Acre   2004         115
6 AC    12     Acre   2005         125
```

Como fazer essa transformação? Usamos a função `pivot_longer`, para alongar a base `homic` que é originalmente larga. Seu uso é simples e requer apenas que passemos a ela o nome do objeto onde está o banco que queremos modificar; e o nome das variáveis que queremos alongar ou preservar como estão no banco. Aplicamos `pivot_longer` da seguinte forma:

```
# Carrega o pacote tidyverse
library(tidyverse)

# Reune as variaveis de ano espalhadas pela base 'homic'
homic2 <- pivot_longer(homic, -c(Sigla, Codigo, Estado), names_to = "Ano", values_to = "Homi

# Verifica as primeiras observacoes do novo banco
head(homic2)
```

```
# A tibble: 6 x 5
  Sigla Codigo Estado Ano   Homicidios
  <chr> <chr>  <chr>  <chr>      <dbl>
1 AC    12     Acre   2000         108
2 AC    12     Acre   2001         122
3 AC    12     Acre   2002         151
4 AC    12     Acre   2003         135
5 AC    12     Acre   2004         115
6 AC    12     Acre   2005         125
```

O resultado da aplicação de `pivot_longer` é uma base *tidy*. De forma mais detalhada, a função `pivot_longer` possui dois argumentos obrigatórios: `data`, que é o nome do objeto onde está o banco; e `cols`, que indica quais colunas devem ser alongadas (no nosso exemplo, as colunas `2000`, `2001`, etc.). O mais importante do código anterior é que declaramos as variáveis a manter como estavam (pois já estavam no formato *tidy*) com o uso de `-c()`. Podemos fazer o inverso: indicar quais variáveis devem ser alongadas, em vez de quais devem ser mantidas:

```
# Reune as variaveis de ano espalhadas pela base 'homic'
homic3 <- pivot_longer(homic, `2000`:`2009`, names_to = "Ano", values_to = "Homicidios")
head(homic3)
```

```
# A tibble: 6 x 5
  Sigla Codigo Estado Ano   Homicidios
  <chr> <chr>  <chr>  <chr>      <dbl>
1 AC    12     Acre   2000         108
2 AC    12     Acre   2001         122
3 AC    12     Acre   2002         151
4 AC    12     Acre   2003         135
5 AC    12     Acre   2004         115
6 AC    12     Acre   2005         125
```

No exemplo, usamos `2000`:`2009` para indicar o nome de todas as variáveis que queríamos alongar, o que deve ser lido como "selecione todas as variáveis entre `2000` e `2009`" (não se preocupe se esse uso de `:` para selecionar variáveis não fez sentido agora, veremos isso adiante).

A função `pivot_longer` também possui dois argumentos opcionais: `names_to`, que é o nome que iremos dar à variável que armazenará o nome das variáveis reunidas (neste caso, "Ano"); e `values_to`, que é o nome da variável que armazenará os valores das variáveis reunidas. Note que não é necessário declarar os argumentos `names_to` e `values_to`, caso no qual a função `pivot_longer` atribui às novas variáveis os nomes `name` e `value`, respectivamente.

```r
# Reune as variaveis de ano espalhadas pela base 'homic'
homic4 <- pivot_longer(homic, -c(Sigla, Codigo, Estado))
head(homic4)
```

```
# A tibble: 6 x 5
  Sigla Codigo Estado name  value
  <chr> <chr>  <chr>  <chr> <dbl>
1 AC    12     Acre   2000    108
2 AC    12     Acre   2001    122
3 AC    12     Acre   2002    151
4 AC    12     Acre   2003    135
5 AC    12     Acre   2004    115
6 AC    12     Acre   2005    125
```

> ⚠ Sobrescrevendo objetos
>
> Ao restruturar a base `homic`, criamos novos objetos `homic2`, `homic3` e `homic4` para não sobrescrever o conteúdo do `tibble homic` original. Dito de outra maneira, executar `homic <- pivot_longer(homic, -c(Sigla, Codigo, Estado))` faria com que o objeto `homic` original fosse substituído pelo resultado de `pivot_longer`.

Para desfazer a operação de alongamento, usamos a função inversa, que é `pivot_wider`. Precisamos passar para ela apenas o nome das variáveis que queremos espalhar em diferentes colunas (vamos usar o `tibble homic2` aqui, criado algumas linhas atrás):

```r
# Espalha as variaveis de ano reunidas pela base 'homic'
homic5 <- pivot_wider(homic2, names_from = Ano, values_from = Homicidios)
head(homic5)
```

```
# A tibble: 6 x 13
  Sigla Codigo Estado   `2000` `2001` `2002` `2003` `2004` `2005` `2006` `2007`
  <chr> <chr>  <chr>     <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 AC    12     Acre        108    122    151    135    115    125    155    133
2 AL    27     Alagoas     724    836    989   1041   1034   1211   1617   1839
3 AM    13     Amazonas    559    478    512    561    523    598    697    711
4 AP    16     Amapá       155    184    181    190    173    196    203    171
5 BA    29     Bahia      1223   1573   1735   2155   2255   2823   3276   3608
6 CE    23     Ceará      1229   1298   1443   1560   1538   1692   1793   1936
# i 2 more variables: `2008` <dbl>, `2009` <dbl>
```

Diferentemente de `pivot_longer`, com `pivot_wider` temos que passar dois argumentos obrigatórios: `names_from`, que é o nome da variável que armazena os nomes das variáveis que queremos espalhar; e `values_from`, que é o nome da variável que armazena os valores das variáveis que queremos espalhar, sem aspas.[3]

Com estas duas funções podemos tanto colocar em várias colunas valores que estavam agrupados numa única (último exemplo) quanto colocar numa mesma coluna valores que estavam espalhados por várias outras (primeiro exemplo). Na sequência, começaremos a usar o pacote `dplyr` para manipular bases de dados, mas, quando necessário, recorreremos às funções `pivot_` para estruturar inicialmente elas em formato *tidy*.

3.2 Operacões básicas de manipulação de dados

Com uma base estruturada de forma adequada, podemos realizar outras operações nela (na verdade, frequentemente precisamos realizar operações *tidy* em várias etapas de limpeza de dados). Neste capítulo, vamos nos concentrar em quatro operações, que chamaremos de verbos:

- *filtrar*, para escolher observações (linhas) para manter ou excluir com base em algum critério;
- *selecionar*, para escolher colunas a manter, reordenar ou remover;
- *modificar*, para criar ou alterar variáveis e observações; e
- *agrupar*, para realizar modificações ou resumos de informações por grupo.

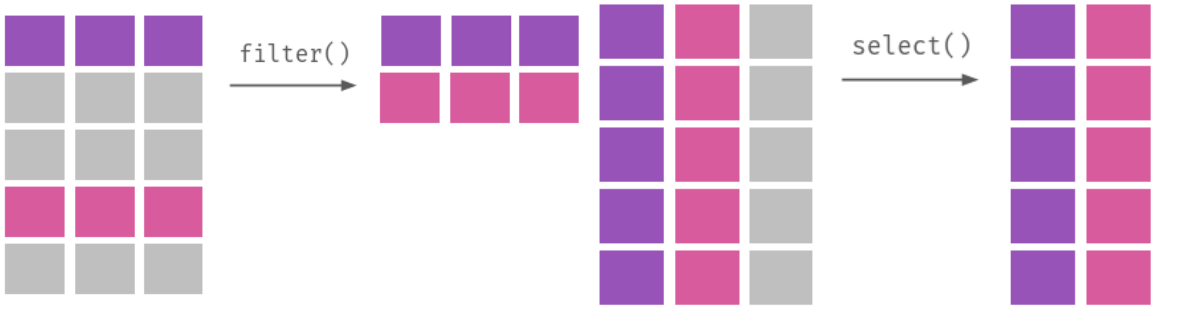
Melhor do que explicar, a Figura 3.1 ilustra visualmente o que cada um desses verbos faz em uma base de dados.

Enquanto que `slice` e `filter` fazem operações horizontais (elas cortam linhas de um banco de dados), `select` faz operações verticais (ela corta, ou reordena, colunas); `mutate`, por sua vez, faz operações dos dois tipos, já que podemos usá-la para modificar apenas algumas observações de uma variável quanto adicionar, ou remover, colunas a uma base. Por fim, `group_by` serve para agrupar observações em um banco, algo útil para calcular estatísticas de um grupo, algo que fazemos com `summarise`.

Para exemplificar esses principais verbos de manipulação, trabalharemos com alguns dados sobre as despesas realizadas por todas as capitais brasileiras no ano de 2012, disponibilizados pela Secretaria do Tesouro Nacional em seu *website*.[4] A base está no arquivo `capitais.Rda`

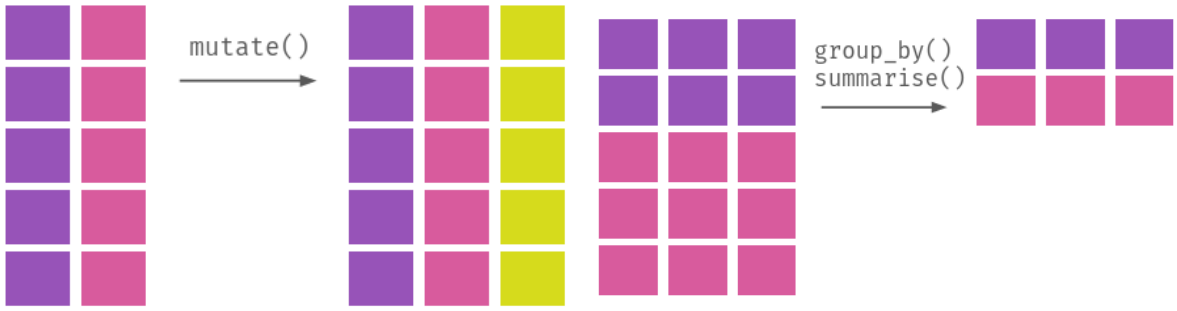
[3] Muitas vezes é confuso o uso ou não de aspas ao passar o nome de variáveis para alguma função em R. Via de regra, funções do `tidyverse` dispensam as aspas, algo chamado de *tidy evaluation* e que tem como objetivo facilitar a escrita de código.

[4] Os dados completos podem ser obtidos no endereço: http://www.tesouro.fazenda.gov.br/pt_PT/contas-anuais.



(a) Filtrar linhas

(b) Selecionar colunas



(c) Criar/modificar colunas

(d) Resumir por grupo

Figura 3.1: Principais verbos de manipulação de dados do pacote `dplyr`

nos materiais complementares do livro e pode ser carregada com `load`. Feito isto, ela ficará salva na memória no objeto `capitais` no formato `tibble`.

```
load("capitais.Rda")
```

3.2.1 Filtrar linhas

A base `capitais` tem 26 observações e 8 variáveis (lembre-se: você pode usar as funções `View`, `nrow` e `ncol` para checar isso). Começaremos a usar o pacote `dplyr` para selecionar e filtrar observações. Para fazer isso indicando apenas a posição das linhas, usamos a função `slice` - passamos um vetor para a função indicando a posição das linhas que queremos remover. Veja alguns exemplos.

```
# Filtra apenas as cinco primeiras observações do banco capitais
slice(capitais, 1:5)
```

```
# A tibble: 5 x 8
  regiao       uf    capital      populacao despesa_total despesa_assistencia_~1
  <chr>        <chr> <chr>            <dbl>         <dbl>                  <dbl>
1 Nordeste     SE    ARACAJU         587701   1150364953.              31383656.
2 Norte        PA    BELEM          1410430   2110549149               58105215
3 Sudeste      MG    BELO HORIZO~   2395785   6917817946.             172347581.
4 Norte        RR    BOA VISTA       296959    491953689.              14018664.
5 Centro-Oeste MS    CAMPO GRANDE    805397   2290844087.              39604187.
# i abbreviated name: 1: despesa_assistencia_social
# i 2 more variables: despesa_saude <dbl>, despesa_educacao <dbl>
```

```
# Filtra apenas a primeira e a quinta observações do banco capitais
slice(capitais, c(1, 5))
```

```
# A tibble: 2 x 8
  regiao       uf    capital      populacao despesa_total despesa_assistencia_~1
  <chr>        <chr> <chr>            <dbl>         <dbl>                  <dbl>
1 Nordeste     SE    ARACAJU         587701   1150364953.              31383656.
2 Centro-Oeste MS    CAMPO GRANDE    805397   2290844087.              39604187.
# i abbreviated name: 1: despesa_assistencia_social
# i 2 more variables: despesa_saude <dbl>, despesa_educacao <dbl>
```

```r
# Remove as 10 primeiras observacoes do banco capitais
# e salva o resultado no objeto 'cap'
cap <- slice(capitais, -c(1:10))
head(cap)
```

```
# A tibble: 6 x 8
  regiao   uf    capital     populacao despesa_total despesa_assistencia_social
  <chr>    <chr> <chr>           <dbl>         <dbl>                       <dbl>
1 Nordeste PB    JOAO PESSOA    742478   1535075866.                   27579724.
2 Norte    AP    MACAPA         415554    506401565.                    6743489.
3 Nordeste AL    MACEIO         953393   1530192466.                   22125734.
4 Norte    AM    MANAUS        1861838   2962009189.                  101830120.
5 Nordeste RN    NATAL          817590   1325168010.                   42486957.
6 Norte    TO    PALMAS         242070    584961477.                   20624102.
# i 2 more variables: despesa_saude <dbl>, despesa_educacao <dbl>
```

Enquanto que `slice` remove linhas baseadas nas suas posições[5], a outra função para cortar horizontalmente, `filter`, é muito mais flexível. Com ela, podemos especificar condições para remover observações (e.g., remover observações de `capitais` cuja despesa total no ano de 2012 seja maior ou menor que algum valor) ou combinações de várias condições. Ela é útil, portanto, para filtrar observações com base em um ou mais critérios, como mostram os exemplos a seguir.

```r
# Filtra observacoes com populacao maior que 2 milhoes habitantes
filter(capitais, populacao > 2000000)
```

```
# A tibble: 5 x 8
  regiao   uf    capital        populacao despesa_total despesa_assistencia_so~1
  <chr>    <chr> <chr>              <dbl>         <dbl>                      <dbl>
1 Sudeste  MG    BELO HORIZONTE   2395785   6917817946.                172347581.
2 Nordeste CE    FORTALEZA        2500194   4137588203.                 78034074.
3 Sudeste  RJ    RIO DE JANEIRO   6390290  18702324296.                565052833.
4 Nordeste BA    SALVADOR         2710968   3618049094.                 40981531.
5 Sudeste  SP    SAO PAULO       11376685  36400104976.                919021471.
# i abbreviated name: 1: despesa_assistencia_social
# i 2 more variables: despesa_saude <dbl>, despesa_educacao <dbl>
```

[5]Essas operações simples fazemos com R-base determinando as posições indexadas com conchetes: capitais[-c(1:10),], como vimos no Capítulo 1

```
# Filtra observacoes da regiao sul e cria um novo objeto
sul <- filter(capitais, regiao == "Sul")
sul
```

```
# A tibble: 3 x 8
  regiao uf    capital       populacao despesa_total despesa_assistencia_social
  <chr>  <chr> <chr>             <dbl>         <dbl>                       <dbl>
1 Sul    PR    CURITIBA        1776761   5115609915.                 117962804.
2 Sul    SC    FLORIANOPOLIS    433158   1080743166.                  33082667.
3 Sul    RS    PORTO ALEGRE    1416714   4122115448.                 146233833.
# i 2 more variables: despesa_saude <dbl>, despesa_educacao <dbl>
```

O primeiro argumento recebido por `filter`, assim como `slice`, é o nome do objeto com a base de dados (um `data.frame` ou `tibble`) e, depois, os critérios usados para filtragem. Algo essencial aqui é que podemos usar qualquer operador lógico (como `==`, `>`, `<`, `>=`, `<=`, `!=`) para criar condições de filtragem, incluindo combinações de condições, o que é feito com o operador `&` (e) ou `,`. Por exemplo, para filtrar observações com população maior que 500 mil e menor que 1 milhão, podemos usar qualquer uma das duas alternativas a seguir:

```
# Filtra observacoes com populacao maior que 500 mil e menor que 1 milhao
filter(capitais, populacao > 500000 & populacao < 1000000) # ou
filter(capitais, populacao > 500000, populacao < 1000000)
```

Um uso mais comum de `filter` é o de combinar critérios com base em diferentes variáveis. Imagine, por exemplo, que queremos filtrar apenas capitais que gastaram mais de R$ 250 milhões em saúde e mais de R$ 300 milhões em educação em 2012. Para isso, usamos `filter` da seguinte forma:

```
filter(capitais,
       despesa_saude > 250000000,
       despesa_educacao > 300000000)
```

Note que, por fazer testes lógicos (que retornam `TRUE` ou `FALSE`, geralmente feitos com os operadores lógicos vistos no Capítulo 1), `filter` pode ser usada para realizar tarefas como remover observações com *missings* (função `is.na()`) ou valores extremos. Por exemplo, para remover observações com *missings* em `despesa_saude`, podemos usar:

```
filter(capitais, !is.na(despesa_saude))
```

Finalmente, um uso de `filter` que não podemos deixar de mencionar é o de manter apenas observações cujos valores de uma variável pertencem a um conjunto de valores – usando, para isso, o operador `%in%` (visto no Capítulo 1). Por exemplo, para manter apenas capitais que pertencem às regiões Sul ou Sudeste, podemos usar:

```
filter(capitais, regiao %in% c("Sul", "Sudeste"))
```

Combinando diferentes operadores lógicos e variáveis, podemos realizar uma série de operações de filtragem. Teste as seguintes operações de filtragem para entender melhor algumas dessas possibilidades:

```
filter(capitais, !uf %in% c("RS", "SP", "MG"))
filter(capitais, regiao == "Sul" | regiao == "Sudeste")
filter(capitais, regiao == "Nordeste" & populacao < 1000000)
filter(capitais, !(regiao == "Nordeste" & populacao < 1000000))
```

3.2.2 Selecionar colunas

Selecionar colunas é algo que usamos com frequência para manter, reordenar ou remover variáveis em uma análise. Para realizar este tipo de operação, usamos a função `select`[6] do pacote `dplyr` (que é parte do `tidyverse` e, portanto, é carregado automaticamente quando executamos `library(tidyverse)`). Um exemplo de como usar `select`:

```
# Seleciona apenas as variaveis uf, capital e populacao do banco
cap1 <- select(capitais, uf, capital, populacao)
head(cap1)
```

```
# A tibble: 6 x 3
  uf    capital         populacao
  <chr> <chr>               <dbl>
1 SE    ARACAJU            587701
2 PA    BELEM             1410430
3 MG    BELO HORIZONTE    2395785
4 RR    BOA VISTA          296959
5 MS    CAMPO GRANDE       805397
6 MT    CUIABA             561329
```

[6] Não raro podemos ter alguns problemas ao usar a função `select` quando estamos com o pacote `MASS` carregado: ambos possuem uma função chamada `select`, o que pode gerrar erros como `Error in select(...) : unused argument (...)`. Nestes casos, temos duas opções: (1) descarregar o pacote `MASS` com `detach("package:MASS", unload = T)`, ou, (2), usar a função `select` com `dplyr::select`.

```r
# Remove a variavel populacao
cap2 <- select(capitais, -populacao)
head(cap2)
```

```
# A tibble: 6 x 7
  regiao       uf    capital  despesa_total despesa_assistencia_~1 despesa_saude
  <chr>        <chr> <chr>            <dbl>                  <dbl>         <dbl>
1 Nordeste     SE    ARACAJU    1150364953.              31383656.    391263344.
2 Norte        PA    BELEM      2110549149               58105215     595930546
3 Sudeste      MG    BELO HO~   6917817946.             172347581.   2029533813.
4 Norte        RR    BOA VIS~    491953689.              14018664.    105492562.
5 Centro-Oeste MS    CAMPO G~   2290844087.              39604187.    734214086.
6 Centro-Oeste MT    CUIABA     1302650057.              32016290.    366936045.
# i abbreviated name: 1: despesa_assistencia_social
# i 1 more variable: despesa_educacao <dbl>
```

O primeiro argumento da função `select` é o banco de dados que queremos manipular, seguido do nome das variáveis que queremos manter, sem aspas e separadas por vírgula; se quisermos excluir uma variável, colocamos um sinal de subtração, `-`, antes do seu nome. Além destes usos, também podemos selecionar colunas com `select` com base na posição delas.

```r
# Mantem apenas a 1a e a 3a colunas
select(capitais, 1, 3)

# Exclui a 1a e a 3a colunas
select(capitais, -1, -3)
select(capitais, -c(1, 3)) # Mesmo resultado
```

Para diminuir a quantidade de código que precisamos escrever, também podemos usar dois pontos, `:`, como em vetores, para selecionar colunas, o que deve ser lido como "selecione todas as colunas contidas entre a variável A e B (A:B)":

```r
# Mantem as colunas entre uf e despesa_total
select(capitais, uf:despesa_total)

# Mantem as colunas entre uf e populacao e a coluna despesa_saude
select(capitais, uf:populacao, despesa_saude)
```

Dado que podemos selecionar colunas com `select`, é fácil perceber que podemos reordenar, ou mesmo duplicar, colunas com ela. Para isso, basta passar para `select` as colunas na ordem desejada:

```r
# Reordena as colunas do banco capitais
select(capitais, populacao, uf, capital)

# Duplica a variavel populacao
select(capitais, populacao, uf, capital, populacao)

# Inverte a ordem das colunas
select(capitais, 8:1)
```

3.2.2.1 Funções auxiliares a select

E se tivermos uma base de dados muito grande, com centenas de variáveis? Como selecionar as que queremos manter sem ter que escrever o nome ou a posição de cada uma? Para casos assim, o `dplyr` nos fornece funções auxiliares para selecionar colunas. Destas, as principais são:

- `starts_with()` e `ends_with()`, para selecionar apenas variáveis cujos nomes contenham algum prefixo ou sufixo.

```r
# Seleciona apenas variaveis que comecem com 'despesa'
cap1 <- select(capitais, starts_with("despesa"))
head(cap1)
```

```
# A tibble: 6 x 4
  despesa_total despesa_assistencia_social despesa_saude despesa_educacao
          <dbl>                      <dbl>         <dbl>            <dbl>
1   1150364953.                   31383656.    391263344.       156571174.
2   2110549149                    58105215     595930546        360221999
3   6917817946.                  172347581.   2029533813.      1169885015.
4    491953689.                   14018664.    105492562.       110284325.
5   2290844087.                   39604187.    734214086.       484548412.
6   1302650057.                   32016290.    366936045.       277989807.
```

- `contains()`, para selecionar apenas variáveis cujos nomes contenham alguma palavra ou caracteres.

```r
# Seleciona apenas variaveis que contenham 'acao'
cap1 <- select(capitais, contains("acao"))
head(cap1)
```

```
# A tibble: 6 x 2
  populacao despesa_educacao
      <dbl>            <dbl>
1    587701       156571174.
2   1410430       360221999
3   2395785      1169885015.
4    296959       110284325.
5    805397       484548412.
6    561329       277989807.
```

- `where()`, para selecionar variáveis com base em alguma condição (e.g., manter apenas variáveis numéricas).[7]

```
# Seleciona apenas variaveis numericas
select(capitais, where(is.numeric))

# Seleciona variaveis numericas e 'capital'
select(capitais, capital, where(is.numeric))
```

3.2.3 Criar e modificar variáveis

O `dplyr` não serve apenas para filtrar e selecionar observações e variáveis. Com `mutate`, podemos alterar variáveis ou adicionar novas a um banco (elas são incluídas no fim do banco, logo após todas as demais). Podemos usá-la, por exemplo, para calcular a despesa total per capita das capitais brasileiras em 2012 – que é igual a despesas total dividida pelo número de habitantes de cada capital.

```
# Cria a variavel despesa_per_capita
cap1 <- mutate(capitais, despesa_per_capita = despesa_total / populacao)
select(cap1, capital, despesa_total, populacao, despesa_per_capita)
```

```
# A tibble: 26 x 4
   capital        despesa_total populacao despesa_per_capita
   <chr>                  <dbl>     <dbl>              <dbl>
 1 ARACAJU          1150364953.    587701              1957.
 2 BELEM            2110549149    1410430              1496.
 3 BELO HORIZONTE   6917817946.   2395785              2887.
 4 BOA VISTA         491953689.    296959              1657.
```

[7] Em versões anteriores do `dplyr`, a função `select_if` era usada como padrão para selecionar colunas com base em alguma condição. A partir da versão 1.0.0, no entanto, a função `where` passou a ser o padrão recomendado para esse tipo de uso.

```
 5 CAMPO GRANDE     2290844087.    805397             2844.
 6 CUIABA           1302650057.    561329             2321.
 7 CURITIBA         5115609915.   1776761             2879.
 8 FLORIANOPOLIS    1080743166.    433158             2495.
 9 FORTALEZA        4137588203.   2500194             1655.
10 GOIANIA          2952160894.   1333767             2213.
# i 16 more rows
```

Como mostra o exemplo anterior, só precisamos passar o nome do banco para a função, o nome da nova variável a ser criada, sem aspas, e o conteúdo dela – que pode ser o resultado de alguma operação aritmética em cima de uma das variáveis do banco. Além da economia de caracteres, a função `mutate` consegue usar as variáveis já existentes do banco para criar uma nova. Intuitivamente, o que ela faz é nos dar acesso às demais variáveis de forma vetorizada: se quisermos criar uma nova variável que seja igual a o logaritmo natural da variável população, portanto, `mutate` aplicará o `log` a cada elemento da variável `populacao`.

```
# Cria uma variavel que e' igual ao log de populacao
cap_log <- mutate(capitais, log_populacao = log(populacao))
select(cap_log, capital, log_populacao)
```

```
# A tibble: 26 x 2
   capital        log_populacao
   <chr>                  <dbl>
 1 ARACAJU                 13.3
 2 BELEM                   14.2
 3 BELO HORIZONTE          14.7
 4 BOA VISTA               12.6
 5 CAMPO GRANDE            13.6
 6 CUIABA                  13.2
 7 CURITIBA                14.4
 8 FLORIANOPOLIS           13.0
 9 FORTALEZA               14.7
10 GOIANIA                 14.1
# i 16 more rows
```

Com `mutate`, também podemos criar mais de uma variável por vez:

```
# Cria tres variaveis de uma so vez
mutate(capitais,
       despesa_saude_per_capita = despesa_saude / populacao,
       despesa_educacao_per_capita = despesa_educacao / populacao,
```

```
        despesa_assistencia_social = despesa_assistencia_social / populacao
        )
```

Além de criar, podemos modificar variáveis que já temos:

```
# Substitui a variavel de populacao
mutate(capitais, populacao = populacao / 1000)
```

> ⚠ Sobrescrevendo variáveis
>
> Ao criar uma variável com o mesmo nome de outra que já existe em uma base de dados, `mutate` sobrescreve a variável original (desde que o resultado seja salvo no mesmo objeto). Para evitar isso, podemos salvar o resultado de `mutate` em um novo objeto.

Para além desses usos, `mutate` também pode ser usada para criar variáveis que tenham algum valor único, isto é, que se repete para todas as observações. Imagine, por exemplo, que queremos criar uma variável que indique o ano à nossa base `capitais`. Para isso, podemos usar `mutate` da seguinte forma:

```
# Cria uma variavel indicando o ano
mutate(capitais, ano = 2012)
```

Como você já deve ter notado, `mutate` sempre retorna todas as variáveis da base original e adiciona as recém-criadas no final do banco. Podemos alterar esse comportamento por meio de dois argumentos: `.keep`, que indica quais variáveis queremos manter (o padrão é `all`); e `.before` ou `.after`. Para manter apenas as variáveis criadas, por exemplo, podemos usar `.keep = "none"`:

```
mutate(capitais,
    ano = 2012,
    populacao = populacao / 1000,
    .keep = "none"
)
```

```
# A tibble: 26 x 2
   populacao   ano
       <dbl> <dbl>
 1      588.  2012
 2     1410.  2012
 3     2396.  2012
 4      297.  2012
```

```
 5      805.  2012
 6      561.  2012
 7     1777.  2012
 8      433.  2012
 9     2500.  2012
10     1334.  2012
# i 16 more rows
```

E, para posicionar as novas variáveis antes do nome de alguma variável, usamos `.before` (`.after` funciona de maneira similar):

```
mutate(capitais, ano = 2012, .before = regiao)
```

```
# A tibble: 26 x 9
     ano regiao     uf    capital populacao despesa_total despesa_assistencia_~1
   <dbl> <chr>      <chr> <chr>       <dbl>         <dbl>                  <dbl>
 1  2012 Nordeste   SE    ARACAJU    587701   1150364953.              31383656.
 2  2012 Norte      PA    BELEM     1410430   2110549149               58105215
 3  2012 Sudeste    MG    BELO H~   2395785   6917817946.             172347581.
 4  2012 Norte      RR    BOA VI~    296959    491953689.              14018664.
 5  2012 Centro-Oe~ MS    CAMPO ~    805397   2290844087.              39604187.
 6  2012 Centro-Oe~ MT    CUIABA     561329   1302650057.              32016290.
 7  2012 Sul        PR    CURITI~   1776761   5115609915.             117962804.
 8  2012 Sul        SC    FLORIA~    433158   1080743166.              33082667.
 9  2012 Nordeste   CE    FORTAL~   2500194   4137588203.              78034074.
10  2012 Centro-Oe~ GO    GOIANIA   1333767   2952160894.              15382878.
# i 16 more rows
# i abbreviated name: 1: despesa_assistencia_social
# i 2 more variables: despesa_saude <dbl>, despesa_educacao <dbl>
```

3.2.3.1 Criando variáveis condicionalmente

Outro uso importante de `mutate` é em operações condicionais, como quando queremos criar uma variável que assume um determinado valor se uma condição for verdadeira (e.g., a população do município no banco `capitais` é maior que 500 mil habitantes) e outro valor, caso esta condicação seja falsa. Para tanto, usamos `mutate` em conjunto com `if_else`[8], que também é uma função do pacote `dplyr`. Um exemplo:

[8]Existe outra função, `ifelse`, no `R-base` que cumpre a mesma função de `if_else`, mas é de forma geralmente mais lenta que esta. Para uma dicussão deste e de outros problemas de `ifelse`, ver Spector (2008).

```
# Cria uma variavel que indica municipios com mais de 500 mil habitantes
mutate(capitais, capitais_grandes = if_else(populacao > 500000, "Capital de grande porte", "
```

Para os casos em que temos múltiplas condições para testar – imagine, por exemplo, termos de criar uma variável que indique o porte das capitais em cinco faixas –, podemos usar `case_when`, que é uma espécie de combinação de vários `if_else`. Para usá-la, passamos para ela uma sequência de condições e valores, separados por vírgula, que são testadas em sequência – note que o primeiro valor que satisfizer a condição será atribuído à nova variável. Um exemplo:

```
# Cria uma variavel que indica o porte das capitais
porte <- mutate(capitais, porte = case_when(
  populacao < 500000 ~ "Capital de menor porte",
  populacao < 1000000 ~ "Capital de porte intermediario",
  populacao < 2000000 ~ "Capital de grande porte",
  populacao < 5000000 ~ "Capital de grande porte II",
  .default = "Capital de grande porte III"
))

select(porte, capital, populacao, porte)
```

```
# A tibble: 26 x 3
   capital        populacao porte
   <chr>              <dbl> <chr>
 1 ARACAJU           587701 Capital de porte intermediario
 2 BELEM            1410430 Capital de grande porte
 3 BELO HORIZONTE   2395785 Capital de grande porte II
 4 BOA VISTA         296959 Capital de menor porte
 5 CAMPO GRANDE      805397 Capital de porte intermediario
 6 CUIABA            561329 Capital de porte intermediario
 7 CURITIBA         1776761 Capital de grande porte
 8 FLORIANOPOLIS     433158 Capital de menor porte
 9 FORTALEZA        2500194 Capital de grande porte II
10 GOIANIA          1333767 Capital de grande porte
# i 16 more rows
```

Duas coisas a notar: para cada condição declarada em `case_when`, usamos o operador `~` para indicar o valor que a variável deve assumir caso a condição seja verdadeira, ou seja, `TRUE`; e usamos `.default` para indicar o valor que a variável deve assumir caso nenhuma das condições declaradas seja verdadeira.

3.2.4 Agrupar e resumir

`summarise`, assim como `mutate`, é usada para modificar variáveis num banco. Mas, diferentemente desta última, ela agrega as informações, retornando um resumo dos dados numa única observação. Um exemplo: calcular a população total das capitais estaduais brasileiras em 2012:

```
# Calcula a populacao total das capitais
summarise(capitais, populacao_total = sum(populacao))
```

```
# A tibble: 1 x 1
  populacao_total
            <dbl>
1        43578158
```

A sintaxe desta função é semelhante a da função `mutate`: como de praxe, passamos o nome do banco de dados para a função e, depois, o nome da variável que queremos criar seguida do seu conteúdo. Deve ficar nítido, contudo, que `summarise` colapsa informações em uma única linha – esse é a sua utilidade. Podemos somar valores de variáveis, calcular estatísticas descritivas (média e desvio-padrão, por exemplo), entre outros:

```
# Calcula estatisticas da populacao das capitais estaduais em 2012
summarise(capitais,
          media_populacao = mean(populacao),
          mediana_populacao = median(populacao),
          desvio_populacao = sd(populacao)
          )
```

```
# A tibble: 1 x 3
  media_populacao mediana_populacao desvio_populacao
            <dbl>             <dbl>            <dbl>
1         1676083            891812         2345416.
```

3.2.4.1 Operações dentro de grupos

Com frequência, em vez de resumir todas as informações de um banco em uma única linha, queremos resumir as informações por grupos. Para ilustrar esse uso, vamos calcular agora a população total das capitais estaduais por região do país, isto é, somaremos a população de todas as capitais que pertencem à mesma região (i.e., o mesmo grupo). Para tanto, usamos `group_by`, que é uma função do `dplyr` que agrupa as observações de um banco de dados com base em alguma variável:

```r
# Agrupa as observacoes por regiao
capitais_regiao <- group_by(capitais, regiao)

# Calcula a populacao total por regiao
summarise(capitais_regiao, populacao_total = sum(populacao))
```

```
# A tibble: 5 x 2
  regiao       populacao_total
  <chr>                  <dbl>
1 Centro-Oeste         2700493
2 Nordeste            11737204
3 Norte                5017906
4 Sudeste             20495922
5 Sul                  3626633
```

A base retornada é bem menor, e preserva apenas uma observação por região (para quem usa outros *softwares* de análise de dados, isto é o equivalente a agregar informações). Fundamental nesse exemplo, para calcular a população total por região usamos `summarise` em conjunto com `group_by`. Isso é necessário porque, se usássemos apenas `summarise(capitais, populacao_total = sum(populacao))`, o `R` somaria a população de todas as capitais, sem considerar a região a que elas pertencem. Dizendo de outra forma, `group_by` indica para a função `summarise` que qualquer operação de resumo de variáveis devem ser feitas *dentro dos grupos* indicados por ela.

Por si só, `group_by` não altera nada na base de dados usada, isto é, nenhuma observação ou coluna é alterada. Ao contrário, o que ela faz é um tipo de modificação interna em uma base: é como se ela dividisse um banco em vários sub-bancos especificados por uma ou mais variáveis. Para ver se uma base foi agrupada, basta executar o seu objeto:

```r
capitais_agrupadas_regiao <- group_by(capitais, regiao)
capitais_agrupadas_regiao
```

```
# A tibble: 26 x 8
# Groups:   regiao [5]
   regiao       uf    capital     populacao despesa_total despesa_assistencia_~1
   <chr>        <chr> <chr>           <dbl>         <dbl>                  <dbl>
 1 Nordeste     SE    ARACAJU        587701   1150364953.              31383656.
 2 Norte        PA    BELEM         1410430   2110549149                58105215
 3 Sudeste      MG    BELO HORIZ~   2395785   6917817946.             172347581.
 4 Norte        RR    BOA VISTA      296959    491953689.              14018664.
 5 Centro-Oeste MS    CAMPO GRAN~    805397   2290844087.              39604187.
```

```
 6 Centro-Oeste MT    CUIABA         561329   1302650057.              32016290.
 7 Sul          PR    CURITIBA      1776761   5115609915.             117962804.
 8 Sul          SC    FLORIANOPO~    433158   1080743166.              33082667.
 9 Nordeste     CE    FORTALEZA     2500194   4137588203.              78034074.
10 Centro-Oeste GO    GOIANIA       1333767   2952160894.              15382878.
# i 16 more rows
# i abbreviated name: 1: despesa_assistencia_social
# i 2 more variables: despesa_saude <dbl>, despesa_educacao <dbl>
```

A segunda linha do *output* indica que a base está agrupada pela variável `regiao` (`Groups: regiao [5]`), o que significa que qualquer operação de resumo retornará uma linha para cada região do país:

```
summarise(capitais_agrupadas_regiao,
          media_populacao = mean(populacao),
          mediana_populacao = median(populacao),
          desvio_populacao = sd(populacao)
          )
```

```
# A tibble: 5 x 4
  regiao       media_populacao mediana_populacao desvio_populacao
  <chr>                  <dbl>             <dbl>            <dbl>
1 Centro-Oeste         900164.           805397           394843.
2 Nordeste            1304134.           953393           787062.
3 Norte                716844.           415554           644916.
4 Sudeste             5123980.          4393038.         4868088.
5 Sul                 1208878.          1416714           695496.
```

Embora `group_by` seja mais frequentemente usada em conjunto com `summarise`, podemos combiná-la com `mutate`. Imagine, por exemplo, que seja necessário adicionar uma variável à base `capitais` que seja igual à população das capitais de cada região, isto é, uma variável que some a população de todas as capitais de cada região (e.g., população de Porto Alegre + população de Curitiba + população de Florianópolis para a região Sul). Como fazemos isso? Usamos `group_by` junto de `mutate`:

```
# Agrupa o banco capitais por regiao
cap_regiao <- group_by(capitais, regiao)

# Soma a populacao das capitais
cap_regiao <- mutate(cap_regiao, pop_regiao = sum(populacao))
```

```r
# Resultado (usando select para selecionar algumas variaveis)
select(cap_regiao, regiao, capital, pop_regiao)
```

```
# A tibble: 26 x 3
# Groups:   regiao [5]
   regiao       capital        pop_regiao
   <chr>        <chr>               <dbl>
 1 Nordeste     ARACAJU          11737204
 2 Norte        BELEM             5017906
 3 Sudeste      BELO HORIZONTE   20495922
 4 Norte        BOA VISTA         5017906
 5 Centro-Oeste CAMPO GRANDE      2700493
 6 Centro-Oeste CUIABA            2700493
 7 Sul          CURITIBA          3626633
 8 Sul          FLORIANOPOLIS     3626633
 9 Nordeste     FORTALEZA        11737204
10 Centro-Oeste GOIANIA           2700493
# i 16 more rows
```

Enquanto uma base estiver agrupada, todas as operações que realizarmos nela serão feitas nos grupos. Para evitar isto, usamos `ungroup`.

```r
# Agrupa o banco capitais por regiao
cap_regiao <- group_by(capitais, regiao)

# Soma a populacao das capitais
cap_regiao <- mutate(cap_regiao, pop_regiao = sum(populacao))

# Desagrupa o banco
cap_regiao <- ungroup(cap_regiao)
```

> **Desagrupando bases resumidas**
>
> Se usarmos `summarise` para criar uma base resumida por grupo, podemos usar o argumento `.groups = "drop"` para desagrupar a base resultante, sem a necessidade de usar `ungroup`. Exemplo:
>
> ```r
> cap_regiao <- group_by(capitais, regiao)
> summarise(cap_regiao, pop_regiao = sum(populacao), .groups = "drop")
> ```

3.2.5 Modificando múltiplas variáveis com `mutate` e `summarise`

Os exemplos anteriores mostram como alterar ou resumir variáveis, agrupando elas ou não, de forma individual. No mais das vezes, é isso o que precisamos: transformar apenas uma ou duas variáveis, ou ainda resumir múltiplas informações usando `group_by`. Em outros casos, porém, precisaremos alterar inúmeras variáveis ao mesmo tempo: imagine, por exemplo, ter de transformar em logaritmo 50 variáveis; com o que vimos anteriormente, isso equivaleria a repetir essa transformação também 50 vezes, uma para cada variável. Algo mais ou menos assim:

```
# Processo para transformar 50 variaveis (exemplo hipotetico)
df <- mutate(df, var1 = log(var1),
             var2 = log(var2),
             var3 = log(var3),
             ... # O codigo continuaria ate chegarmos em var50
             )
```

Para evitar as repetições de código, o `dplyr` oferece uma função auxiliar chamada `across` para aplicar uma operação a múltiplas variáveis. Seu uso é ligeiramente diferente do que vimos até agora:

```
# Transforma em logaritmo todas as variaveis numericas
cap <- mutate(capitais, across(where(is.numeric), log))
select(cap, where(is.numeric))
```

```
# A tibble: 26 x 5
   populacao despesa_total despesa_assistencia_~1 despesa_saude despesa_educacao
       <dbl>         <dbl>                  <dbl>         <dbl>            <dbl>
 1      13.3          20.9                   17.3          19.8             18.9
 2      14.2          21.5                   17.9          20.2             19.7
 3      14.7          22.7                   19.0          21.4             20.9
 4      12.6          20.0                   16.5          18.5             18.5
 5      13.6          21.6                   17.5          20.4             20.0
 6      13.2          21.0                   17.3          19.7             19.4
 7      14.4          22.4                   18.6          20.8             20.5
 8      13.0          20.8                   17.3          19.2             19.3
 9      14.7          22.1                   18.2          21.0             20.4
10      14.1          21.8                   16.5          20.7             20.2
# i 16 more rows
# i abbreviated name: 1: despesa_assistencia_social
```

Em vez de especificar o nome de cada variável que será criada ou modificada, `across` aplica uma operação a todas as variáveis que satisfaçam uma determinada condição. De forma esquemática, toda chamada da função `across` contém duas partes: primeiro, indicamos quais variáveis serão modificadas (no exemplo, usamos `where(is.numeric)` para selecionar todas as variáveis numéricas da base); segundo, indicamos qual operação faremos nas variáveis selecionadas (no caso, usamos `log`).

`across` é flexível o suficiente para permitir, também, usá-la para resumir variáveis de um banco. Podemos, por exemplo, calcular a média de todas as variáveis numérica da base `capitais` com:

```
# Calcula a media de todas as variaveis numericas
summarise(capitais, across(where(is.numeric), mean))
```

```
# A tibble: 1 x 5
  populacao despesa_total despesa_assistencia_s~1 despesa_saude despesa_educacao
      <dbl>         <dbl>                   <dbl>         <dbl>            <dbl>
1   1676083   4176143826.              102628661.    949270809.       798449762.
# i abbreviated name: 1: despesa_assistencia_social
```

`across`, como dá para imaginar, também pode ser usada junto de `group_by` para resumir variáveis por grupo. Para calcular a média de todas as variáveis numéricas da base `capitais` por região, por exemplo, usamos:

```
cap <- group_by(capitais, regiao)
summarise(cap, across(where(is.numeric), mean))
```

```
# A tibble: 5 x 6
  regiao       populacao despesa_total despesa_assistencia_social despesa_saude
  <chr>            <dbl>         <dbl>                      <dbl>         <dbl>
1 Centro-Oeste   900164.   2181885013.                  29001118.    693170960.
2 Nordeste      1304134.   2235820825.                  37825580.    664463187.
3 Norte          716844.   1159226954.                  34821304.    264852902.
4 Sudeste       5123980.  15869659949.                 424970795.   3077616245.
5 Sul           1208878.   3439489509.                  99093101.    818974725.
# i 1 more variable: despesa_educacao <dbl>
```

Indo além, `across` não está limitada a usarmos `where` para selecionar variáveis – é possível usar qualquer lógica de seleção válida para a função `select`, que vimos anteriormente. Desse modo, todos os seguintes usos são válidos (teste cada um deles):

```
# Transforma em logaritmo todas as variaveis que comecem com 'despesa'
mutate(capitais, across(starts_with("despesa"), log))

# Resume apenas variaveis selecionadas pelo nome
summarise(capitais, across(c(populacao, despesa_total), median))

# Calcula valores per capita das variaveis de despesa
mutate(capitais, across(contains("despesa"), \(x) x / populacao))
```

No último exemplo, usamos o que é chamado de *função anônima* para aplicar uma operação a todas as variáveis que satisfazem uma determinada condição, usando `x` como uma espécie de coringa – o que o R interpreta como sendo cada uma das variáveis selecionadas por `contains("despesa")`.[9] Aqui, o `\(x)` sempre deve ser colocado para indicar que queremos aplicar uma operação a cada uma das variáveis selecionadas. Para entender melhor como isso funciona, teste o seguinte código:

3.2.6 Encadeando operações com *pipes*

Como vimos no Capítulo 1, o *pipe* (`|>`) serve para encadearmos operações em R, isto é, passar o resultado de uma função como argumento para outra função – o que, em muitos casos, tem a vantagem de tornar nossos códigos mais legíveis. Quando trabalhando manipulando bases de dados, esse uso dos *pipes* se destaca especialmente. O código que rodamos anteriormente em mais de uma linha para agrupar a base de capitais e calcular resumos dela, por exemplo, com *pipes* pode ser escrito assim:

```
capitais |>
    group_by(regiao) |>
    summarise(populacao_total = sum(populacao))
```

```
# A tibble: 5 x 2
  regiao       populacao_total
  <chr>                  <dbl>
1 Centro-Oeste         2700493
2 Nordeste            11737204
3 Norte                5017906
4 Sudeste             20495922
5 Sul                  3626633
```

[9] Entender como `across` e `mutate` funcionam por meio de funções anônimas envolve estudar tópicos como controle de fluxo e programação funcional, algo que extrapola os objetivos deste livro. Para quem quiser aprender sobre, o melhor lugar para começar é o capítulo 26 da segunda edição do livro *R for Data Science*, de Wickham, Çetinkaya-Rundel, e Grolemund (2023).

O código com *pipes* funciona exatamente da mesma forma, exceto pelo seguinte: o `data.frame` `capitais` é passado para `group_by` por meio do *pipe*, isto é, ele é jogado da linha de cima para dentro dos parênteses da função `group_by`, na linha seguinte; por sua vez, o resultado de `group_by` é passado para `summarise` da mesma forma, e assim por diante. O *pipe* é, portanto, uma forma de escrever códigos que, de certa forma, imita a forma como lemos um texto: da esquerda para a direita, de cima para baixo.

Quando temos que realizar diferentes operações de manipulação em uma base, usar *pipes* evita muitas repetições e a necessidade de termos de criar objetos intermediários, que guardam resultados parciais de operações. Por exemplo, imagine que queremos calcular a média da população das capitais brasileiras em 2012 por região, mas antes queremos remover as capitais com populações menores que 500 mil habitantes e, também, transformar a variável da população em população por mil habitantes (i.e., dividi-la por 1000). Sem usar *pipes*, faríamos algo assim:

```
# Remove as capitais com populacao menor que 500 mil habitantes
cap1 <- filter(capitais, populacao > 500000)

# Transforma a variavel populacao
cap2 <- mutate(cap1, populacao = populacao / 1000)

# Calcula a media da populacao por regiao
cap3 <- group_by(cap2, regiao)
cap4 <- summarise(cap3, media_populacao = mean(populacao))
```

Com *pipes*, esse amontado de código pode ser escrito de forma bem mais sucinta:

```
# Calcula a media da populacao por regiao
capitais |>
    filter(populacao > 500000) |>
    mutate(populacao = populacao / 1000) |>
    group_by(regiao) |>
    summarise(media_populacao = mean(populacao))
```

A moral da história: sempre que possível, use *pipes* para encadear operações em R. Além de tornar seu código mais legível, ele evita a criação de objetos intermediários – o que pode economizar memória RAM.

3.2.7 Outras operações úteis: ordernar, renomear e sortear

Embora não seja o objetivo do capítulo cobrir todas as possibilidades de manipulação de bancos de dados, o `dplyr` contém algumas funções que podem ser extremamente úteis em

determinadas situações. Reordenar observações de acordo com os valores de uma variável (por exemplo, ordenando o banco `capitais` pelo tamanho de suas populações) poderia ser uma delas. Selecionar aleatoriamente apenas algumas observações de um banco para ter uma amostra, outra. Longe de esgotar as possibilidades do pacote, listamos aqui algumas funções que nos ajudam a resolver estes e outros problemas específicos (é recomendável reproduzir esses exemplos para fixar as funções de interesse). Seguem:

```
# arrange() serve para ordenar as observacoes de um banco
head(arrange(capitais, populacao))

# rename() serve para renomear uma variavel (nome atual vem na frente, seguido do nome antig
names(capitais) # nomes atuais

capitais |>
    rename(populacao_novo = populacao)
names(capitais) # novos nomes

capitais |>
    rename(populacao = populacao_novo, SAUDE = despesa_saude)
names(capitais) # nomes novos 2

# sample_n() sorteia apenas algumas obsvervacoes de um banco
sample_frac(capitais, 2) # sorteia duas capitais
sample_frac(capitais, 3) # sorteia tres capitais
```

3.2.8 Manipulando bases muito grandes

E se quisermos manipular grandes bases de dados, grandes o suficiente para não caberem na memória RAM do computador? No Capítulo 2, vimos que podemos usar o pacote `DBI` junto do `duckdb` para ler essas bases em instâncias intermediárias – bancos de dados relacionais gerenciados pelo DuckDB. A grande vantagem dessa solução é que ela é totalmente integrada ao `tidyverse`: uma vez importando via DuckDB, podemos manipular uma base usando todas as funções[10] do `dplyr`. Para ilustrar, vamos importar novamente a base de microdados de pessoas do Censo de 2010 (Pereira e Barbosa 2023) usando `DBI` e `duckdb`:

```
library(duckdb)
library(DBI)
```

[10] Algumas funções específicas do `dplyr`, na verdade, não possuem equivalentes que podem ser traduzidos para SQL, a linguagem de consulta utilizada pela maioria dos sistemas de gerenciamento de bancos relacionais. Para uma visão geral sobre os casos em que é possível usar `dplyr` com bancos de dados, ver a documentação do pacote dbplyr, responsável por traduzir código em R para SQL.

```
con <- dbConnect(duckdb::duckdb())
censo <- tbl(con, "2010_population_v0.2.0.parquet")
```

Com o banco importado para o DuckDB, podemos manipulá-lo normalmente com `dplyr` – que traduzirá nosso código em R para algo que o DuckDB entenda. Para selecionar apenas as três primeiras colunas da base de microdados, por exemplo, podemos usar `select` assim:

```
select(censo, 1:3)
```

```
# Source:   SQL [?? x 3]
# Database: DuckDB v1.0.1-dev1922 [fmeireles@Linux 6.9.5-200.fc40.x86_64:R 4.4.0/:memory:]
   code_muni code_state abbrev_state
   <chr>     <chr>      <chr>
 1 1100015   11         RO
 2 1100015   11         RO
 3 1100015   11         RO
 4 1100015   11         RO
 5 1100015   11         RO
 6 1100015   11         RO
 7 1100015   11         RO
 8 1100015   11         RO
 9 1100015   11         RO
10 1100015   11         RO
# i more rows
```

Ou, outro exemplo, podemos usar `summarise` para calcular o tamanho da população brasileira em 2010 de acordo com os dados do Censo de 2010 (usando a variável `V0010` da base, que contém os pesos amostrais, isto é, o quanto cada linha representa em termos de habitantes):

```
summarise(censo, populacao = sum(V0010))
```

```
# Source:   SQL [1 x 1]
# Database: DuckDB v1.0.1-dev1922 [fmeireles@Linux 6.9.5-200.fc40.x86_64:R 4.4.0/:memory:]
   populacao
       <dbl>
1 190755799.
```

Se quisermos saber a população de cada região, basta incluir antes uma chamada à função `group_by`, para agrupar a base:

```
censo |>
    group_by(name_region) |>
    summarise(populacao = sum(V0010))
```

```
# Source:   SQL [5 x 2]
# Database: DuckDB v1.0.1-dev1922 [fmeireles@Linux 6.9.5-200.fc40.x86_64:R 4.4.0/:memory:]
  name_region  populacao
  <chr>            <dbl>
1 Centro-oeste 14058094.
2 Norte        15864454.
3 Sudeste      80364410.
4 Nordeste     53081950.
5 Sul          27386891.
```

Em todos os casos, se quisermos salvar o resultado das operações em um objeto é necessário usar a função `collect`. A razão disso decorre do fato de que, ao usar `dplyr` com bancos de dados, o resultado das operações não é salvo na memória RAM, mas sim no DuckDB. Usando `collect` para trazer o resultado para a memória RAM, o código anterior ficaria assim:

```
consulta <- censo |>
    group_by(name_region) |>
    summarise(populacao = sum(V0010)) |>
    collect()

consulta
```

```
# A tibble: 5 x 2
  name_region  populacao
  <chr>            <dbl>
1 Centro-oeste 14058094.
2 Norte        15864454.
3 Sudeste      80364410.
4 Nordeste     53081950.
5 Sul          27386891.
```

Como se percepe pelo *output* do R, o objeto exibido agora é um `tibble` com o resultado da operação.

3.3 Cruzar e combinar dados

Até aqui, cobrimos algumas das principais operações de manipulação de dados: sabemos como chegar ao formato `tidy` e como filtrar linhas e selecionar, criar, modificar e resumir colunas de uma base. O que ainda não vimos é como cruzar dados de diferentes bases – algo geralmente necessário em pesquisas reais.

Para aprendermos a fazer cruzamentos, usaremos duas bases de dados com informações sobre as cinco regiões do país. Para carregar as duas bases, chamadas de `regioes` e `territorio` e que estão no arquivo `regioes.Rda`[11], pode usar a função `load`:

```
load("regioes.Rda")
```

Executando os objetos no console, você verá que cada `tibble` contém uma coluna chamada `regiao`, que indica o nome de cada uma das regiões do país. Importante para os nossos exemplos, a base `territorio` tem uma linha a menos, pois não contém a região Sul. Além disso, a grafia da região Centro-Oeste está diferente nas duas bases: na base `regioes`, usa-se hífen; na `territorio`, não.

```
regioes
territorio
```

```
# A tibble: 5 x 2
  regiao       populacao
  <chr>            <dbl>
1 Norte          15864454
2 Nordeste       53081950
3 Centro-Oeste   14058094
4 Sudeste        80364410
5 Sul            27386891
```

```
# A tibble: 4 x 2
  regiao           km2
  <chr>          <dbl>
1 Norte        3853840
2 Nordeste     1554291
3 Centro Oeste 1606234
4 Sudeste       924608
```

3.3.1 Cruzamentos e colunas-chave

Para cruzar dados de diferentes bases, usamos as funções `_join` do pacote `dplyr`, como `left_join`, `inner_join`, `full_join` e `right_join`. A diferença entre elas é a forma de combinar as informações das bases. Para entender melhor, vamos cruzar as duas bases de dados que temos usando `left_join`:

[11] É possível salvar mais de um objeto ou base de dados em um arquivo `.Rda`. Para isso, basta separar os objetos por vírgula na hora de salvá-los (e.g., `save(df1, df2, df3, file = "dados.Rda")`).

```
left_join(regioes, territorio, by = join_by(regiao == regiao))
```

```
# A tibble: 5 x 3
  regiao       populacao     km2
  <chr>            <dbl>   <dbl>
1 Norte          15864454 3853840
2 Nordeste       53081950 1554291
3 Centro-Oeste   14058094      NA
4 Sudeste        80364410  924608
5 Sul            27386891      NA
```

A explicação do código é a seguinte: `left_join` combina as informações de duas bases de dados, mantendo todas as linhas da base à esquerda (no caso, `regioes`) e adicionando as informações da base à direita (no caso, `territorio`) quando houver correspondência entre as variáveis usadas para cruzar as bases (no caso, usamos `regiao == regiao`, passado no argumento `by = join_by(regiao == regiao)`, para dizer que a informação contida na coluna `regiao` de uma base tem correspondência na coluna `regiao` da outra). Em outras palavras, para cada região do país na base `regiões`, a sua correspondente foi buscada na base `territorio` e, se encontrada, as informações da coluna `km2` foram adicionadas à base `regioes` como uma nova coluna.[12]

O resultado do cruzamento das bases `regioes` e `territorio` também ilustra o que acontece quando falta correspondência entre valores. Como a base `territorio` não contém a região Sul, a linha correspondente a essa região na base `regioes` ficou com valores faltantes na coluna `km2`; adicionalmente, dado que a grafia da região Centro-Oeste estava diferente nas duas bases, `left_join` não encontrou um valor de `km2` na base `territorios` para preencher.

O que podemos tirar de lição geral do exemplo de `left_join` é que para adicionarmos variáveis a um banco de dados a partir de informações específicas (Sudeste com Sudeste, Norte com Norte, etc.) precisamos de variáveis que possuam valores comuns em duas bases. Isso significa que não podemos combinar duas bases que não partilhem informações comuns, chamadas convencionalmente de *colunas-chave*: só foi possível cominar as bases `regioes` e `territorio` porque ambas têm uma variável com o nome das macrorregiões do Brasil, e é a partir delas que junção das bases foi feita: como `Sudeste` em uma das bases é igual a `Sudeste` na outra, o `R` entende que as linhas que contêm essa informação devem ser mantidas lado a lado. O que a função `left_join` faz, portanto, é combinar bases a partir de valores comuns de variáveis de ambas.

[12] Um *insight* útil da documentação do pacote `dplyr` é o de que as funções `_join` são chamadas de *mutate join* justamente porque funcionam como `mutate()`, adicionando novas colunas a uma base. Ver, por exemplo, a documentação de `left_join` com `?left_join`.

> ⚠ Colunas-chave e cruzamentos
>
> Para cruzar duas bases de dados, precisamos de variáveis que possuam valores comuns em ambas, as *colunas-chave*, que podem ser nomeadas de formas diferentes nas duas bases – mas precisam ser da mesma classe.

3.3.2 Funções `_join`

De forma prática, a função `left_join` (e as demais funções `_join`) possui três argumentos principais. O primeiro indica a primeira base de dados usada na junção dos dados; a segunda, a outra base que será unida à primeira; por fim, usamos o argumento `by` para indicar quais variáveis são comuns nas duas bases. Esse último argumento é o principal da função, já que é com ele que informamos ao `R` como unir as variáveis.

`left_join`, contudo, é apenas uma das funções contidas no pacote `dplyr`. Além disso, ela realiza essa operação de forma específica: como o `left_` indica, ela cruza valores da segunda base passada à função para a primeira. A consequência desse procedimento, desse modo, é que todas as observações do primeiro banco (`regioes`) são preservadas, e as da segunda base são usadas para preencher os casos comuns.

E se quisermos manter todas as observações da base `territorio` e usar as da base `regioes` para preencher valores de população? Para além da solução mais óbvia (trocar `territorio` e `regioes` de lugar), podemos usar `right_join`:

```
right_join(regioes, territorio, by = join_by(regiao == regiao))
```

```
# A tibble: 4 x 3
  regiao       populacao     km2
  <chr>            <dbl>   <dbl>
1 Norte          15864454 3853840
2 Nordeste       53081950 1554291
3 Sudeste        80364410  924608
4 Centro Oeste         NA 1606234
```

O resultado é autoexplicativo. O `dplyr` também possui outras variantes de `_join` úteis. Considere essas duas:

```
inner_join(regioes, territorio, by = join_by(regiao == regiao))
```

```
# A tibble: 3 x 3
  regiao   populacao     km2
  <chr>        <dbl>   <dbl>
1 Norte     15864454 3853840
2 Nordeste  53081950 1554291
3 Sudeste   80364410  924608
```

```
full_join(regioes, territorio, by = join_by(regiao == regiao))
```

```
# A tibble: 6 x 3
  regiao       populacao     km2
  <chr>            <dbl>   <dbl>
1 Norte          15864454 3853840
2 Nordeste       53081950 1554291
3 Centro-Oeste   14058094      NA
4 Sudeste        80364410  924608
5 Sul            27386891      NA
6 Centro Oeste         NA 1606234
```

Como é possível depreender, `inner_join` e `full_join` cruzam dados de formas inteiramente diferentes: no primeiro caso, apenas linhas que possuem correspondência nas duas bases são mantidas; no segundo, todas as linhas são mantidas, mesmo que não haja correspondência entre as bases.

Em vez de termos de memorizar cada nome de função, uma forma mais intuitiva de entender suas diferentes utilidades é por meio de um diagrama de Venn como o que segue. Nele, cada círculo representa uma base de dados, e as interseções entre eles indicam as observações que serão mantidas em cada função. O que fica patente é que `left_join` e `right_join` são complementares: o que uma função mantém, a outra exclui, assim como `inner_join` e `full_join`. A depender do caso – e do que queremos manter –, uma ou outra função será mais útil.

3.3.3 Controlando o comportamento das funções `_join`

Na maioria das vezes, o que cobrimos é o suficiente para cruzar duas bases. Algo que não vimos em maior detalhe é que, quando os nomes das colunas-chave em duas bases são diferentes, precisamos usar o argumento `by` de forma diferente. Para entender como, imagine que temos agora uma base chamada `regioes2` que contém as mesmas informações de `regioes`, mas com o nome da coluna-chave diferente:

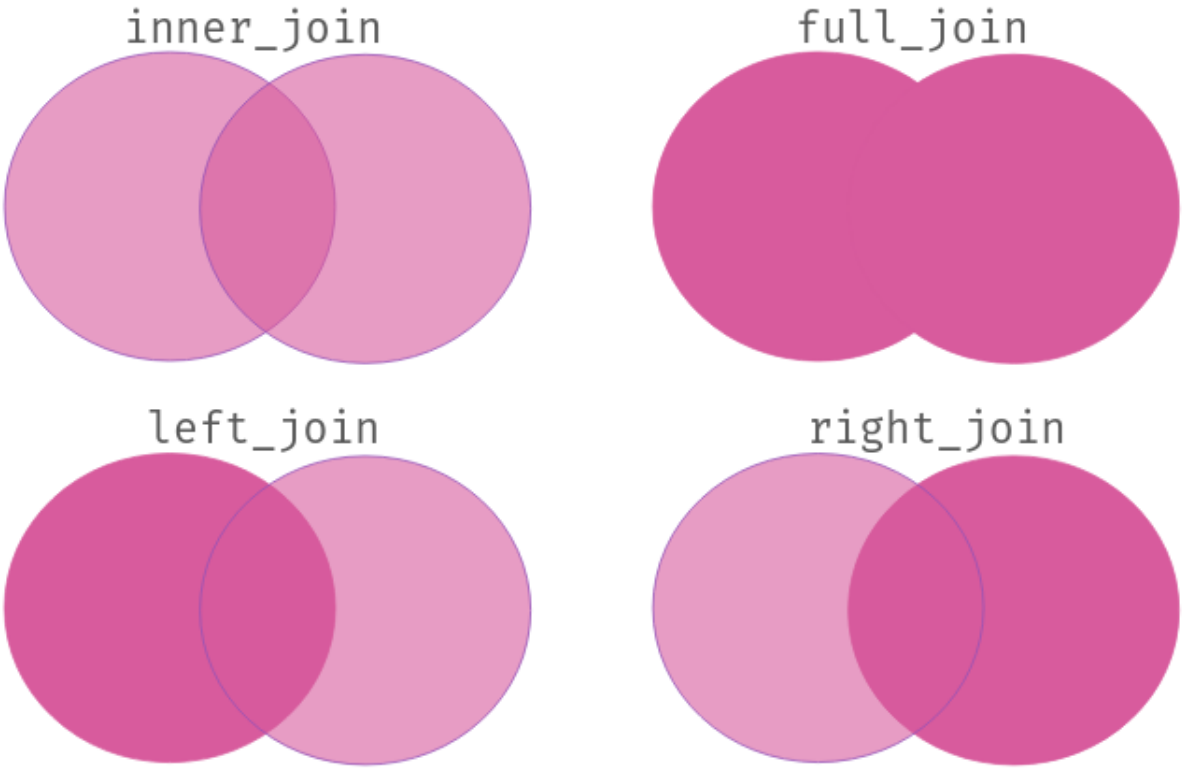


Figura 3.2: Usos das funções `_join`

```
regioes2 <- regioes |>
    rename(regiao2 = regiao)
```

Se tentarmos cruzar `regioes2` com `territorio` usando o código que já vimos anteriormente, teremos um erro:

```
left_join(regioes2, territorio, by = join_by(regiao == regiao))
```

```
Error in `left_join()`:
! Join columns in `x` must be present in the data.
x Problem with `regiao`.
```

Nestes casos, precisamos usar o argumento `by` de forma diferente. Em vez de passar uma expressão `regiao == regiao`, vamos precisar indicar que a coluna `regiao2` da base `regioes2` é igual à coluna `regiao` da base `territorio`:

```
left_join(regioes2, territorio, by = join_by(regiao2 == regiao))
```

```
# A tibble: 5 x 3
  regiao2      populacao     km2
  <chr>            <dbl>   <dbl>
1 Norte          15864454 3853840
2 Nordeste       53081950 1554291
3 Centro-Oeste   14058094      NA
4 Sudeste        80364410  924608
5 Sul            27386891      NA
```

O resultado, agora, não retorna erro – exatamente porque especifica quais colunas têm correspondência entre as bases.

Outro alerta comum ao usar funções `_join` é o de *multiple matches*, que ocorrem quando há mais de uma correspondência entre as bases, isto é, quando uma mesma observação da base à esquerda tem mais de uma correspondência na base à direita. Para entender melhor o ponto, vamos criar uma base de dados chamada `regioes3` que contém duas observações para a região Sudeste (transformando a região Sul em Sudeste com `if_else`):

```
regioes3 <- regioes |>
    mutate(regiao = if_else(regiao == "Sul", "Sudeste", regiao))
```

Se tentarmos cruzar `regioes3` com `territorio`, não teremos erro:

```
left_join(territorio, regioes3, by = join_by(regiao == regiao))
```

```
# A tibble: 5 x 3
  regiao           km2 populacao
  <chr>          <dbl>     <dbl>
1 Norte        3853840  15864454
2 Nordeste     1554291  53081950
3 Centro Oeste 1606234        NA
4 Sudeste       924608  80364410
5 Sudeste       924608  27386891
```

O código roda sem problemas e, como resultado, as duas linhas da região Sudeste são mantidas – mas, se você rodou o código localmente, verá que o `R` emite uma mensagem avisando que há múltiplas correspondências. Caso esse seja o resultado que você espera da operação, basta usar o argumento `relationship = "many-to-many"`, ou `relationship = "one-to-many"`, para indicar que todas as correspondências devem ser mantidas:

```
left_join(regioes3, territorio, by = join_by(regiao == regiao), relationship = "many-to-many"
```

```
# A tibble: 5 x 3
  regiao       populacao     km2
  <chr>            <dbl>   <dbl>
1 Norte          15864454 3853840
2 Nordeste       53081950 1554291
3 Centro-Oeste   14058094      NA
4 Sudeste        80364410  924608
5 Sudeste        27386891  924608
```

Caso isso não seja o que você espera, é necessário investigar o que está acontecendo – em geral, avisos de múltiplas correspondências podem indicar linhas duplicadas ou outros problemas em uma base quando não esperamos o aviso.

3.3.4 Empilhando bases

Para fechar esta seção sobre cruzamentos, vamos falar sobre outro tipo comum de combinar bases: o empilhamento, isto é, quando queremos combinar duas bases que possuem as mesmas variáveis, mas com observações diferentes. Em casos assim, usamos a função `bind_rows`:

```r
bind_rows(regioes, regioes3)
```

```
# A tibble: 10 x 2
   regiao       populacao
   <chr>            <dbl>
 1 Norte          15864454
 2 Nordeste       53081950
 3 Centro-Oeste   14058094
 4 Sudeste        80364410
 5 Sul            27386891
 6 Norte          15864454
 7 Nordeste       53081950
 8 Centro-Oeste   14058094
 9 Sudeste        80364410
10 Sudeste        27386891
```

O resultado é uma base de dados empilhada, que contém todas as observações de `regioes` e `regioes2` – algo útil quando queremos, por exemplo, combinar dados de diferentes anos, ou de diferentes estados, em uma única base.

3.4 Resumo do capítulo

Este capítulo sobre manipulação de dados começou com uma breve introdução sobre o formato `tidy`, que é o formato ideal para a maioria das operações de manipulação de dados. Em seguida, vimos como filtrar linhas de uma base de dados com `filter`, selecionar colunas com `select`, criar e modificar variáveis com `mutate` e resumir informações com `summarise`. Também vimos como agrupar e resumir informações com `group_by` e `summarise`, e como modificar múltiplas variáveis com `across`. Por fim, vimos como cruzar dados de diferentes bases com `left_join`, `right_join`, `inner_join` e `full_join`, e como empilhar bases com `bind_rows`.

Exercícios

> **Arquivos necessários**
>
> Para realizar estes exercícios, será necessário baixar os arquivos que estão na pasta de materiais complementares deste livro e salvá-los na pasta de trabalho do R.

1. Filtrando bases de dados I

O arquivo `pop_quilombola.csv` contém informações sobre a população quilombola nos municípios do Brasil, conforme documentado recentemente pelos primeiros resultados do universo do Censo de 2022 (Geografia e Estatística (IBGE) 2023). Carregue a base e a salve em um objeto (dê a ele o nome que preferir). Em seguida, crie um novo objeto que mantenha apenas as 100 primeiras observações da base.

2. Filtrando bases de dados II

Ainda com o banco disponível no arquivo `pop_quilombola.csv`, mantenha na base apenas as observações que tenham pelo menos uma pessoa auto-identificada como quilombola (se necessário, salve o resultado da operação em um novo objeto). Use comentários para documentar quantos municípios têm populações quilombolas no país.

3. Seleção de variáveis, filtragem e ordenamento

Na pasta de materiais do livro, há um arquivo chamado `pop_total.xlsx` com a informação da população total dos municípios segundo os resultados do universo do Censo de 2022. Carregue esse arquivo no objeto `populacao` e, a partir dele, crie um novo objeto chamado `maiores_populacoes` contendo apenas os 50 municípios mais populados do país, ordenados de forma decrescente. Use comentários para indicar qual é o 50º município mais populoso do país.

4. Criação de variáveis

Usando o objeto `populacao`, criado no exercício anterior, adicione a ele duas novas variáveis: uma que contenha a população dos municípios em mil habitantes e a outra que tenha como valores `Pequeno porte`, para municípios com menos de 50 mil habitantes, e `Outros` para os demais municípios (não é necessário salvar o resultado da operação em um objeto).

5. Resumo de variáveis

Use as duas bases que carregamos (`pop_quilombola.csv` e `pop_total.xlsx`), calcule agora o tamanho da população quilombola e a população total do Brasil.

6. Cruzando bases de dados I

Como visto nos exercícios anteriores, as duas bases nos arquivos `pop_quilombola.csv` e `pop_total.xlsx` têm uma variável chamada `cod_ibge`, que indica o identificador único (ID) atribuído pelo IBGE a cada um dos 5570 municípios do país. Use essa variável para cruzar as duas bases de forma que a base resultante contenha informações sobre a população total e a população quilombola de cada município. Salve o resultado dessa operação no objeto `municipios`.

7. Criação de variáveis, filtragem e ordenamento

Usando a base salva em `municipios` criada há pouco, crie uma nova variável que indique a proporção de pessoas quilombolas em relação ao total da população de cada município e salve o resultado no objeto `municipios_pop_quilombola`. Reporte os 3 municípios com as maiores proporções de pessoas quilombolas nos comentários.

8. Cruzamento de bases de dados II

Na pasta de materiais, há um arquivo chamado `uf_municipios.csv` com três variáveis: `regiao`, `cod_ibge`, `sigla_uf` e `cod_ibge`. Carregue e salve essa base no objeto `uf_municipios` e use-o para adicionar as variáveis com informações de região e estado à base `municipios` – ao final, a base `municipios` deverá ter 6 variáveis: `regiao`, `sigla_uf`, `municipio`, `cod_ibge`, `pop_total` e `pop_quilombola`.

9. Agrupamento e resumo de variáveis I

Usando a nova base `municipios` criada no exercício anterior, calcule a população quilombola e a população total de cada uma das regiões do país. Feito isso, calcule a proporção de pessoas quilombolas em relação à população total de cada região (não é necessário salvar os resultados em um novo objeto).

10. Agrupamento e resumo de variáveis II

Com a base `municipios`, descubra quais são os 5 estados com as maiores proporções de pessoas quilombolas em relação à população total. Reporte os resultados usando comentários.

11. Agrupamento e criação de variáveis

Ainda com a base `municipios`, crie uma nova base chamada `taxa_pop_quilombola` que tenha apenas duas variáveis: `sigla_uf`, com a sigla de cada estado, e `taxa_pop_quilombola`, com o número de pessoas quilombolas por 100 mil habitantes. A base final deve ter apenas 27 linhas, uma para cada unidade da federação. Use essa base para descobrir qual é o estado com a maior taxa de pessoas quilombolas por 100 mil habitantes no país.

4 Visualização

Parte importante do trabalho de análise é a visualização dos nossos dados. Ela pode ser utilizada não só para resumir resultados de pesquisa como, também, para fazer análise exploratória e descobrir padrões. Nesse capítulo, veremos justamente como usar R gerar visualizações que nos ajudem a explorar dados, documentar e exportar resultados de análises.

É possivel gerar visualizações apenas usando R-base, isto é, funcionalidade que já estão disponíveis por padrão no R. No entanto, seguindo que já fizemos em outros capítulos, aqui também adotaremos o `tidyverse`, particularmente o seu pacote `ggplot2`, desenvolvido por Wickham (2016) a partir da gramática de grafícos de Wilkinson (2012). A ideia por detrás dessa gramática, adotada pelo `ggplot2`, é a de que todo gráfico pode ser decomposto em camadas, cada uma delas representando um elemento do gráfico. Abordaremos este tema no capítulo, ainda que brevemente, para facilitar a compreensão da lógica de funcionamento do `ggplot2` – pacote que revolucionou o modo de produzir gráficos usando a linguagem R.

Para poder executar os códigos deste capítulo, você precisará carregar o `tidyverse`:

```
library(tidyverse)
```

4.1 Por que usar visualizações?

Dados podem ser apresentados em uma tabela, seja com valores brutos ou resumidos em alguma métrica, como veremos no próximo capítulo. No entanto, como argumentam Kastellec e Leoni (2007), em geral a análise gráfica é mais intuitiva. Compare, por exemplo, a tabela a seguir, que mostra um indicador fictício que varia ao longo dos anos, de 2010 a 2018, com a sua versão gráfica, ao lado:

Quando bem feito, um gráfico é capaz mostrar dados de forma *precisa*, *nítida* e *eficiente* suas informações (Healy 2018). Em outras palavras, ele passa a sua mensagem, ou a sua história, sem a necessidade ou o auxílio de um texto.

Para que tenhamos um visual gráfico claro, preciso e eficiente, vale levar em conta algumas observações de Tufte (1983). Para ele, bons gráficos colocam os dados acima de tudo, ou seja, todo e qualquer artifício de *design* só deve ser feito exclusivamente para ajudar deixar os dados mais nítidos, e não mais obtusos. A ideia de fundo é que devemos nos concentrar na substância

Tabela 4.1: Indicador fictício ao longo do tempo

ano	indicador1
2010	10
2011	12
2012	15
2013	18
2014	17
2015	22
2016	27
2017	28
2018	29

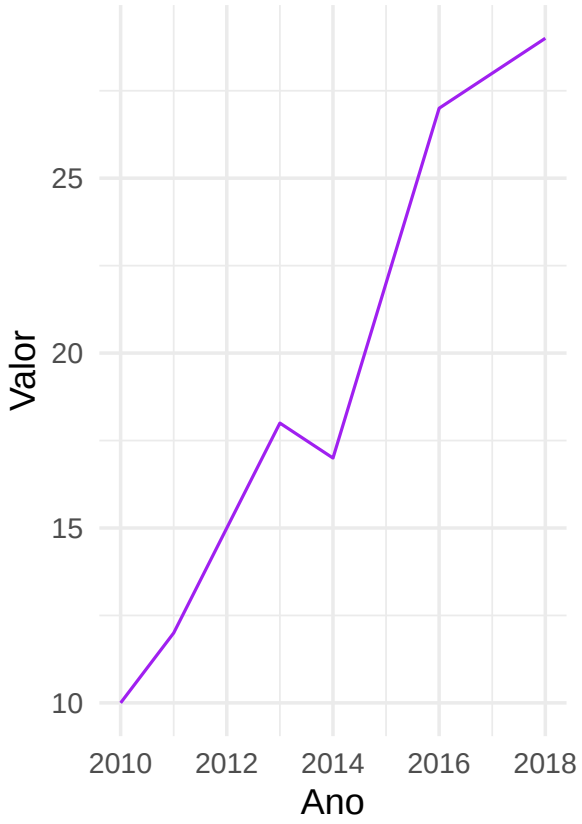


Figura 4.1: Gráfico de um indicador fictício

dos dados, e não em qualquer outro elemento, como adereços desnecessários ou metodologia usada na produção de um gráfico. Sempre que possível, simplifique.

Outro ponto importante é que uma boa visualização deve conseguir levar aos olhos comparações de diferentes pedaços de dados disponibilizados – sem, contudo, gerar distorções destes mesmos dados, levando em consideração o propósito geral da análise, seja ela exploração ou descrição. Tudo isso considerado, usar visualizações é uma forma de comunicar resultados de forma global, com economia de espaço e, se bem feita, sem perda de precisão.

4.2 Fundamentos do `ggplot2`

De longe o pacote mais utilizado para produzir visualizações em R, o `ggplot2`[1] é baseado na gramática de gráficos, que é uma forma de descrever visualizações por meio de camadas. A ideia é que cada camada representa um elemento do gráfico: os dados usados para produzir formas e determinar escalas dos eixos, geometria, coordenadas, facetas e temas. Essa lógica separa, portanto, diferentes partes da produção de um gráfico, o que abre um universo de possibilidades de combinações entre essas camadas.

O que precisamos para fazer um gráfico no ggplot2? Basicamente, penas determinar quais são os nosso dados e a geometria que vamos usar. Para dar um exemplo inicial, usaremos dados de uma das mais tradicionais pesquisas de opinião do Brasil, o Estudo Eleitoral Brasileiro (ESEB), que é realizada periodicamente após eleições nacionais pelo Centro de Estudos de Opinião Pública (CESOP), da Universidade Estadual de Campinas (UNICAMP).[2] A base de dados que usaremos é a do ESEB 2022, disponível no formato SPSS no arquivo `eseb2022.sav` na pasta de materiais complementares deste livro. Para carregar o arquivo, podemos usar a função `read_sav` do pacote `haven`:

```
# Carregando pacotes
library(haven)

# Carregando dados
eseb <- read_sav("eseb2022.sav", encoding = "latin1")
```

Se inspecionarmos a base (com `View(eseb)`), veremos que os *labels* das variáveis não estão disponíveis. Isso acontece porque a função `read_sav` carrega apenas os códigos das categorias das variáveis, e não os *labels* que as descrevem, como vimos no Capítulo 2. Para usar os *labels*, podemos usar a função `as_factor`, também do pacote `haven`:

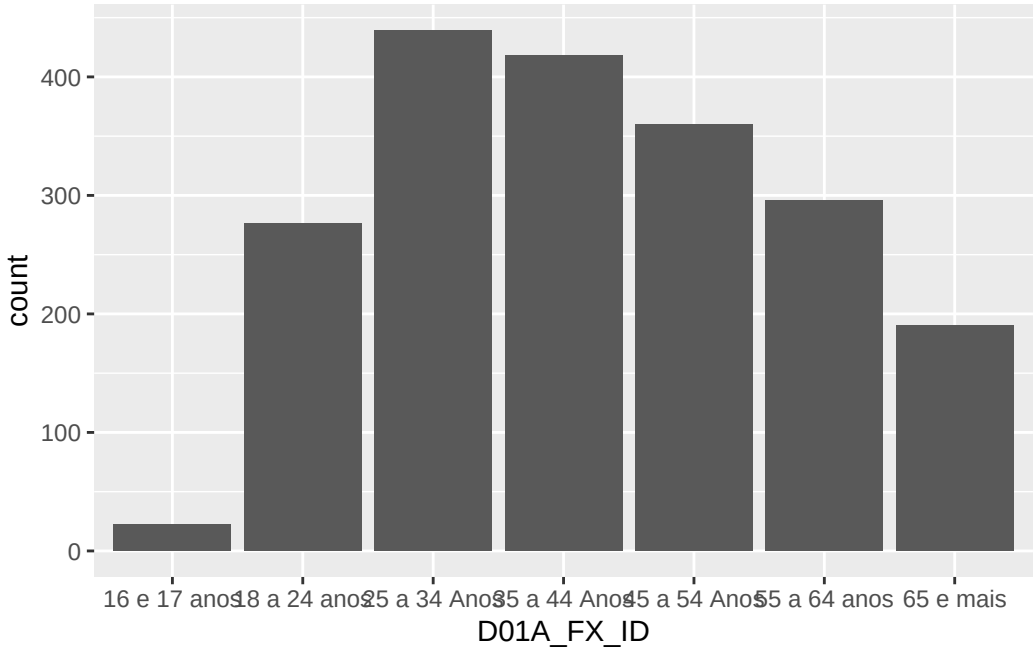
[1] Normalmente mencionamos `ggplot` quando estamos falando do pacote `ggplot2`, utilizado para produzir gráficos.

[2] O ESEB é conduzido desde 2002 e é vinculado ao *Comparative Study of Electoral Systems Project (CSES)*, projeto global de pesquisas pós-eleitorais. Para saber mais sobre o ESEB, ver https://www.cesop.unicamp.br/por/eseb.

```
eseb <- as_factor(eseb)
```

Com os dados carregados e com *labels* atribuídos, podemos fazer o seguinte gráfico usando o ggplot:

```
# Um grafico com ggplot
ggplot(eseb, aes(x = D01A_FX_ID)) +
  geom_bar()
```

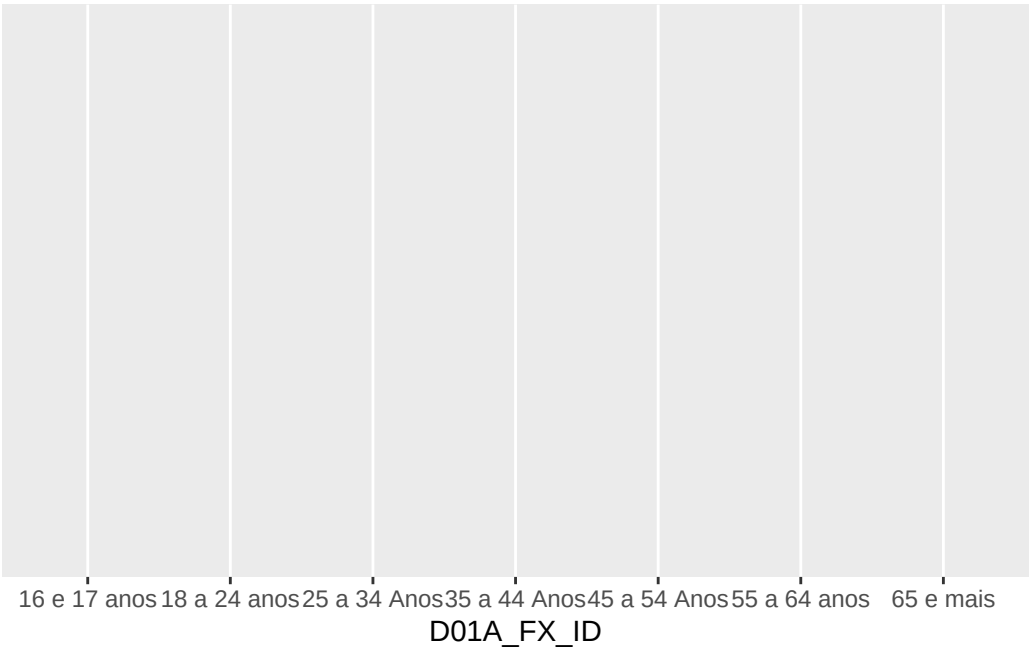


O código anterior cria um gráfico de barras indicando o número de pessoas entrevistadas pelo ESEB em diferentes faixas de idade. Apesar de simples e compacto, no entanto, há várias coisas que merecem ser destacadas nestas linhas. A primeira e mais importante delas: todo gráfico criado com o pacote `ggplot2` depende de execurtamos, logo no início, a função `ggplot` e, no mais das vezes, passarmos para ele nossos dados. Essa é a primeira camada de qualquer gráfico em `ggplot2` – se executarmos apenas esta linha, o R cria um painel em branco, sem qualquer informação, isto é, o fundo no qual todas as demais camadas serão adicionadas. Veja:

```
ggplot(eseb)
```

No gráfico de barras feito anteriormente, passamos outro argumento para a função `ggplot` que é a estética (*aesthetic*), `aes(x = D01A_FX_ID)`. Esta é a parte da primeira camada do gráfico que mapeia as variáveis dos nossos dados para os atributos visuais do gráfico, gerando eixos, cores, formas e tamanhos. No nosso caso, mapeamos a variável `D01A_FX_ID` para o eixo X, ou eixo horizontal. Em termos práticos, passar a estética `aes(x = D01A_FX_ID)` produz o seguinte resultado:

```
ggplot(eseb, aes(x = D01A_FX_ID))
```



Nosso gráfico, agora, não é apenas uma tela em branco – `aes()` passou a informação de que queremos usar a variável `D01A_FX_ID` para o eixo X e, automaticamente, o `ggplot2` adicionou um eixo X com escalas e rótulos. Com esse código, portanto, criamos a primeira camada da nossa visualização Finalmente, a segunda camada que usamos no exemplo inicial foi uma geometria, introduzida por meio da função `+ geom_bar()`. Como o nome sugere, esta serve para criar barras, fazendo a contagem de quantas pessoas entrevistadas temos em cada faixa de idade na base do ESEB.

> **!** Combinação de camadas
>
> Para criar um gráfico no `ggplot2`, conectando camadas por meio do operador `+` (não confundir com o *pipe*, `|>`.).

E se quisermos fazer um gráfico de pontos (*scatterplot*) em de um gráfico de barras? E se quisermos mudar as escalas dos eixos do nosso gráfico? É possível mudar cores, tamanhos e formas? A resposta para todas essas perguntas é sim, e a lógica para fazer isso é sempre a mesma: adicionamos camadas ao gráfico, combinando-as com o operador `+`, como veremos na sequência.

4.3 Camadas de uma visualização

Podemos melhorar muito a capacidade informativa do nosso gráfico. Para isso, podemos explorar mais os elementos que consitituem a lógica da criação de gráficos em R com `ggplot` – ou, melhor, a gramática de gráficos (Wilkinson 2012). Já usamos os dois primeiros elementos desta lógica:

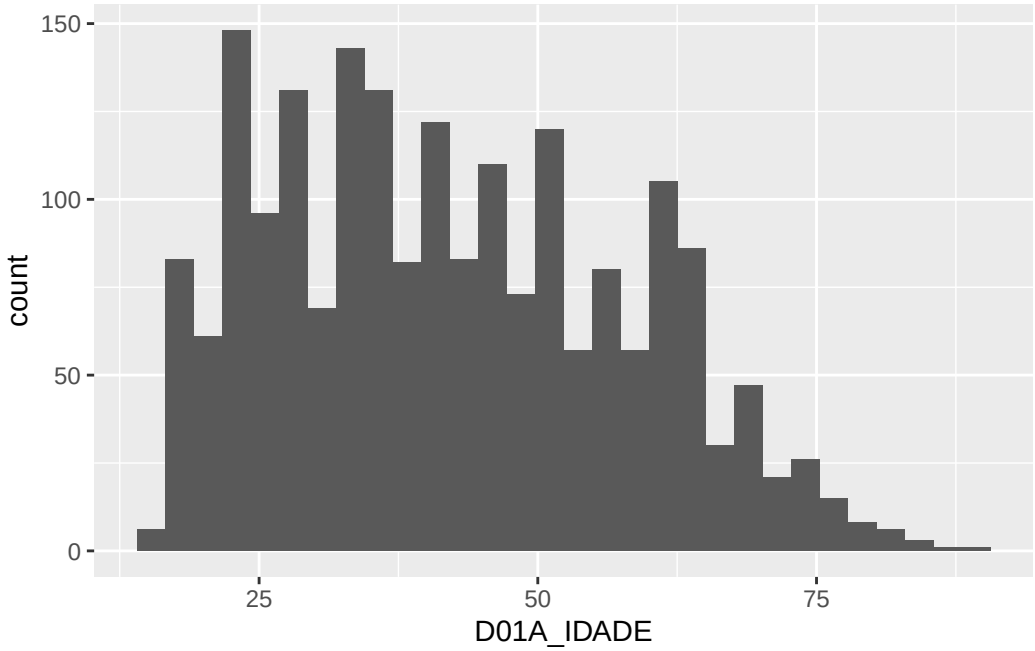
1. **Dados:** conjutos de informações que queremos visualizar e variáveis que serão mapeadas em atributos estéticos;
2. **Geometria:** camadas que contém elementos geoméricos e transformações estatísticas; no ggplot e extensões, essas funções começam com o prefixo `geom_`;
3. **Escalas:** valores em um espaço, incluindo cor, tamanhos ou formas de atributos do gráfico, além de eixos e legendas; funções começam com o prefixo `scale_`;
4. **Coordenadas:** o sistema de coordenadas utilizada, por padrão a coordenada cartesiana, indicando um X e um Y (eixo vertical); estão disponíveis também outros sistemas de coordenadas, como a polar e projeção de mapas; funções têm o prefixo `coord_`;
5. **Faceta:** para fazer gráficos em subgrupos do mesmo conjunto de dados; funções usam o prefixo `facet_`;
6. **Temas**: controle de vários recursos de exibição, como tamanho da fonte e cor do plano de fundo, rotação de texto nos eixos, grade, entre outros; há uma função genérica chama `theme`, que permite a customização de diferentes detalhes de um gráfico, e outras funções usam o prefixo `theme_`.

Dito isto, vamos editar nosso gráfico.

4.3.1 Geometrias

Vamos começar com camadas de geometria. No exemplo anterior, usamos a função `geom_bar()` para criar um gráfico de barras. No entanto, o `ggplot2` tem várias outras geometrias que podem ser usadas para criar diferentes tipos de gráficos. Na base do ESEB, temos uma variável `D01A_IDADE` que indica a idade das pessoas entrevistas, em números inteiros. Podemos usar essa variável para criar, por exemplo, um histograma, que nada mais é do que um gráfico de barras que mostra a distribuição de uma variável numérica. Para isso, usamos a função `geom_histogram()`:

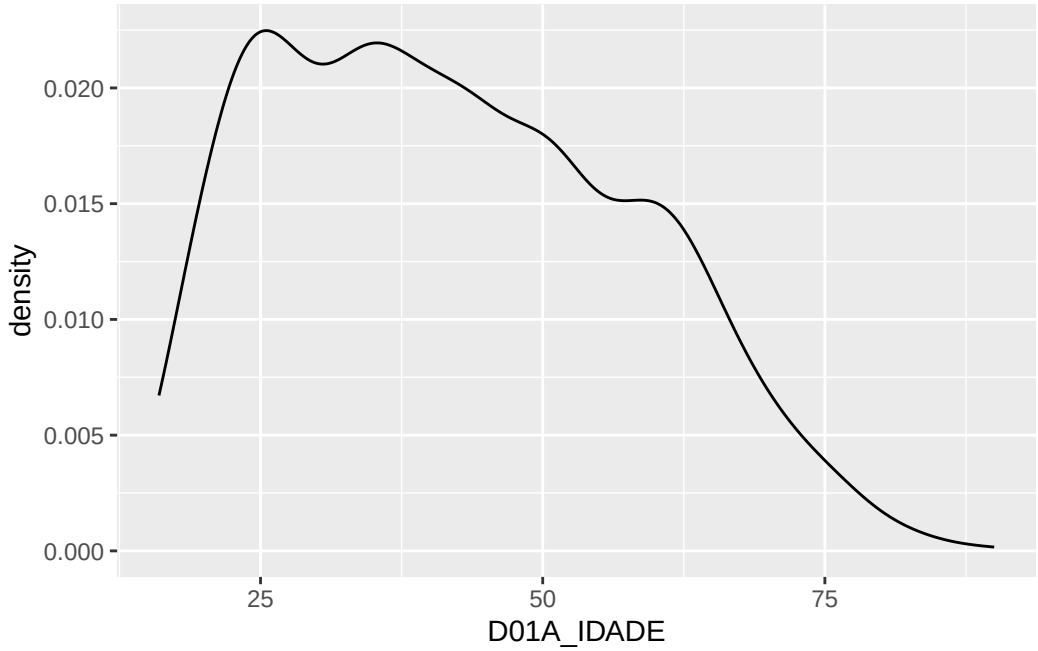
```
ggplot(eseb, aes(x = D01A_IDADE)) +
  geom_histogram()
```



Como podemos ver, o gráfico anterior mostra o número de pessoas com determinada idade. Outra forma de visualizar a mesma informação é usando um gráfico de densidade, que mostra a distribuição de uma variável numérica de forma suave. Para isso, usamos a função `geom_density()`:[3]

```
ggplot(eseb, aes(x = D01A_IDADE)) +
  geom_density()
```

[3]Densidade aqui se refere a uma função de densidade de probabilidade, que é uma função que descreve a probabilidade de uma variável aleatória cair em um intervalo particular de valores.



Cada função `geom_`, portanto, cria um tipo diferente de gráfico. No entanto, é importante lembrar que cada uma delas tem suas próprias particularidades – `geom_density` e `geom_histogram`, por exemplo, não servem para fazer gráficos de variáveis categóricas, enquanto que `geom_bar` não serve para fazer gráficos de variáveis numéricas. A tabela abaixo apresenta um resumo das principais funções `geom_`, quais tipos de variáveis demandam e o tipo de gráfico que produzem:

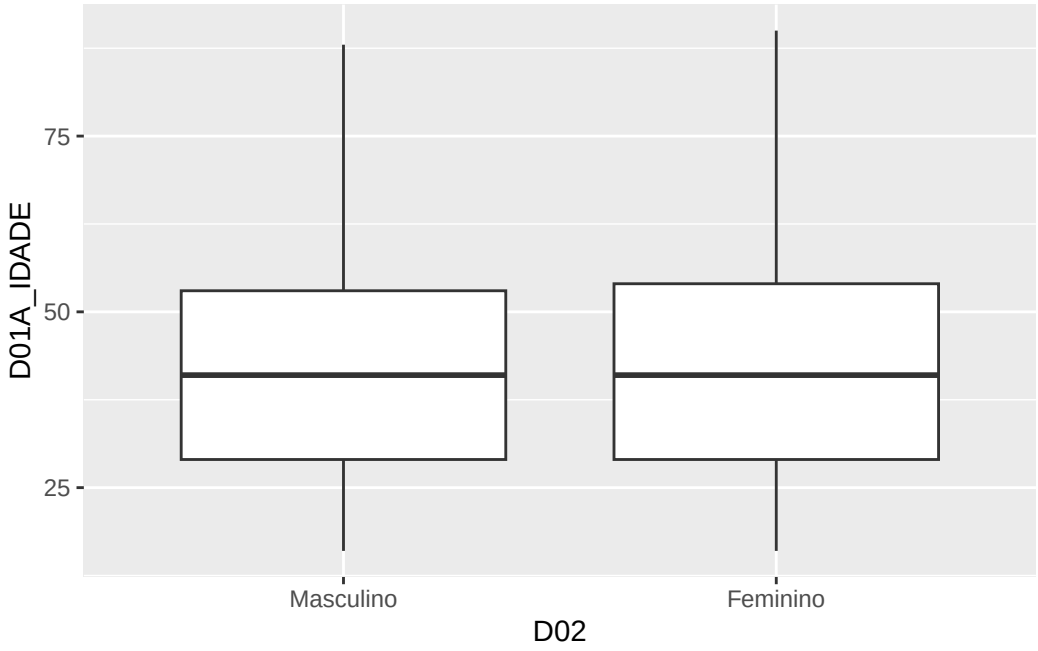
Tabela 4.2: Funções `geom_` mais comuns

Função	Variáveis necessárias	Visualização
`geom_bar()`	`x` categórica	Gráfico de barras
`geom_col()`	`x` categórica e `y` numérica	Gráfico de barras
`geom_histogram()`	`x` numérica	Histograma
`geom_density()`	`x` numérica	Gráfico de densidade
`geom_line()`	`x` e `y` numéricas	Gráfico de linhas
`geom_point()`	`x` e `y` numéricas	Gráfico de pontos
`geom_boxplot()`	`x` categórica e `y` numérica	Boxplot
`geom_violin()`	`x` categórica e `y` numérica	Violin plot

Apesar de diferentes, geometrias como `geom_bar` e `geom_density` precisam apenas de uma variável para criar visualizações, outras como `geom_boxplot` e `geom_violin` precisam de duas variáveis, uma categórica, que informará o eixo X, e, a outra, numérica. Para exemplificar

ambas, podemos fazer um gráfico que mostra a distribuição da idade das pessoas entrevistadas pelo ESEB por sexo (variável `D02`), usando um boxplot:

```
ggplot(eseb, aes(x = D02, y = D01A_IDADE)) +
  geom_boxplot()
```



E um gráfico que mostra a distribuição da idade das pessoas entrevistadas pelo ESEB por sexo (variável `D02`), usando gráfico de violino – um tipo de gráfico que combina boxplot e gráfico de densidade:

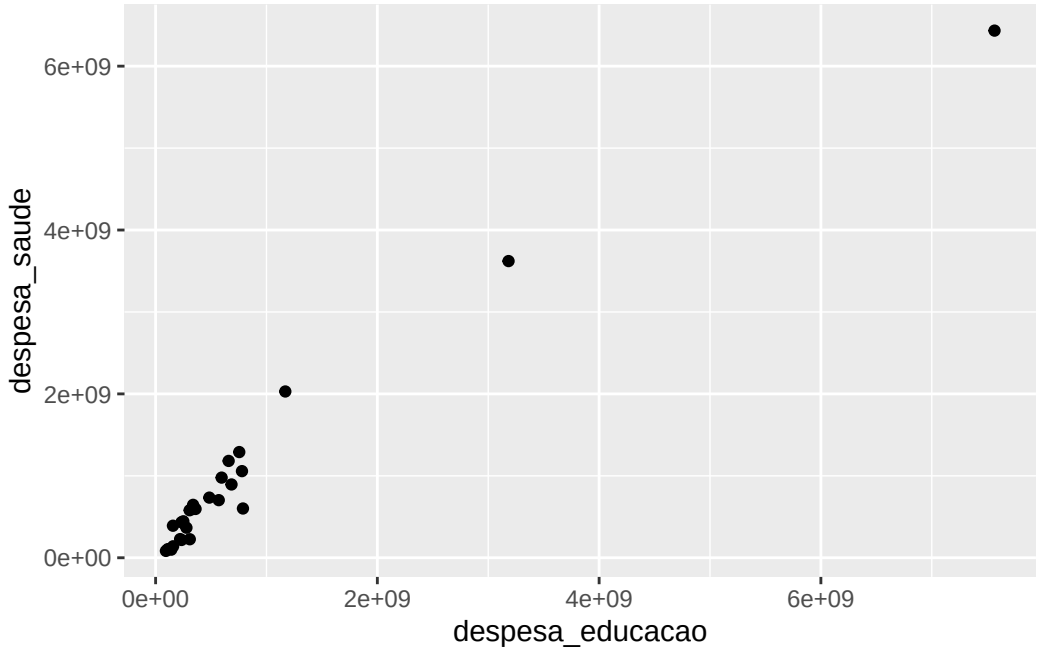
```
ggplot(eseb, aes(x = D02, y = D01A_IDADE)) +
  geom_violin()
```

Para outros tipos de gráficos, por sua vez, às vezes é necessário ter duas variáveis numéricas, como no caso de gráficos de pontos. Para exemplificar essa geometria, vamos carregar novamente a base sobre capitais do país, que vimos no Capítulo 3:

```
load("capitais.Rda")
```

Com ela, podemos fazer, por exemplo, um gráfico de pontos que mostra a relação entre despesas em educação e despesas em saúde desses municípios usando `geom_point`:

```
ggplot(capitais, aes(x = despesa_educacao, y = despesa_saude)) +
  geom_point()
```

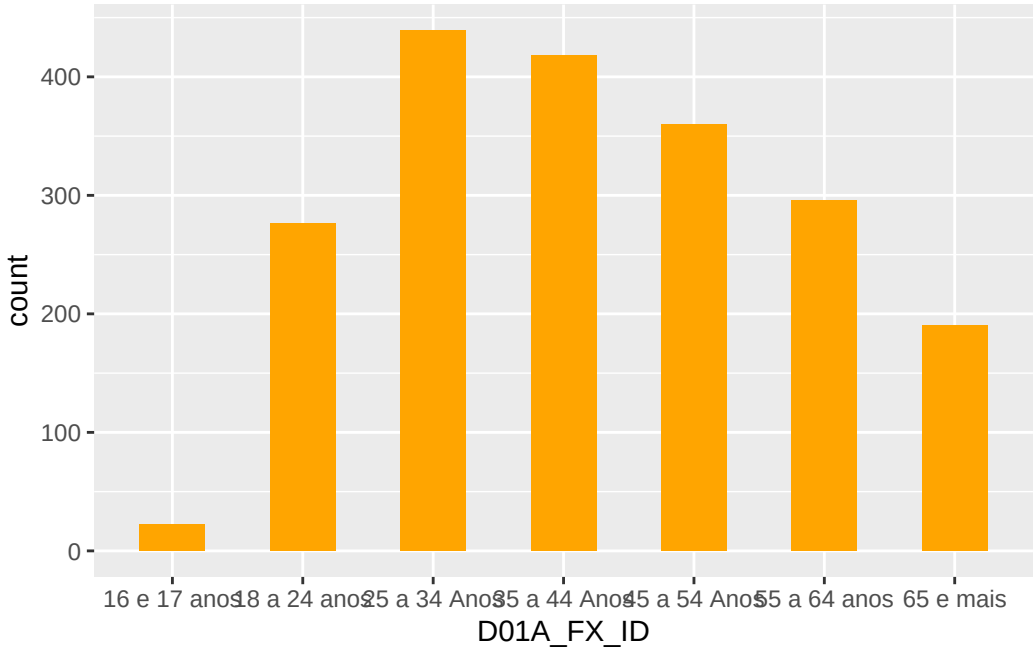


O conteúdo do gráfico, obviamente, não nos diz muito, mas o código exemplifica o ponto subjacente: para fazer um gráfico de pontos, precisamos de duas variáveis numéricas, uma para o eixo X e, a outra, para o eixo Y.

4.3.1.1 Customizando geometrias

Além de criar diferentes tipos de gráficos, as funções `geom_` também permitem customizações por meio dos seus diferentes argumentos. Em um gráfico de barras criado com `geom_bar`, por exemplo, podemos mudar a cor e a largura das barras; a cor e o tamanho dos contornos das barras; etc. No exemplo a seguir, usamos a função `geom_bar` para criar um gráfico de barras que mostra o número de pessoas entrevistadas pelo ESEB em diferentes faixas de idade, mas com barras em laranja e mais finas:

```
ggplot(eseb, aes(x = D01A_FX_ID)) +
  geom_bar(fill = "orange", width = 0.5)
```



Tudo o que fizemos no código anterior foi declarar dois argumentos dentro da função `geom_bar`: `fill`, que muda a cor das barras, e `width`, que muda a largura das barras. Quanto à primeira modificação, quando mudamos cores podemos recorrer a nomes de cores em inglês (como *gray*, *blue*, *darkblue*), ou códigos hexadecimais, também chamados de cores *hex*[4], como `#FFA500` para laranja. Quanto à segunda modificação, a largura das barras é dada em proporção ao espaço disponível no eixo X, que vai de 0 a 1 – número mais próximos de 1 criam barras mais largas.

Cada `geom_` têm seus próprios argumentos, e a melhor forma de saber quais são eles é consultando a documentação do `ggplot2` ou usando a função `?` seguida do nome da função. Via de regra, entretanto, vale lembrar que a maioria das funções `geom_` têm argumentos que controlam cor, tamanho e forma dos elementos que criam. Para referência, reportamos em seguida alguns dos argumentos mais comuns dessas `geom_`, o que fazem e valores de referência:

[4]Cores podem ser representadas por códigos de seis dígitos que representam a intensidade de vermelho, verde e azul (RGB) de uma cor. Por exemplo, o código hexadecimal para a cor laranja é `#FFA500` e, para vermelho, `#FF0000`. Há diversos *websites* dedicados para consultar códigos *hex* de cores específicas, como o color-hex.com.

Tabela 4.3: Argumentos comuns em funções `geom_`

Geometria	Argumento	Descrição	Valores
`geom_bar()`, `geom_col()`, `geom_histogram()`, `geom_density()`, entre outras	`fill`	Muda a cor de preenchimento de barras e áreas	Nome de cor ou código *hex* (e.g., "orange")
`geom_bar()`, `geom_col()`, `geom_histogram()`, `geom_density()`, entre outras	`color`	Muda a cor do contorno de barras e áreas	Nome de cor ou código *hex* (e.g., "orange")
`geom_bar()`, `geom_col()`, `geom_histogram()`, `geom_density()`, entre outras	`alpha`	Muda a transparência de barras e áreas	Número entre 0 e 1
`geom_bar()`, `geom_col()`, `geom_histogram()`, `geom_density()`, entre outras	`width`	Muda a largura de barras e áreas	Número entre 0 e 1
`geom_line()`, `geom_path()`, entre outras	`color`	Muda a cor de linhas	Nome de cor ou código *hex* (e.g., "orange")
`geom_line()`, `geom_path()`, entre outras	`linewidth`	Muda a espessura de linhas	Número
`geom_point()`, `geom_jitter()`, entre outras	`color`	Muda a cor de pontos	Nome de cor ou código *hex* (e.g., "orange")

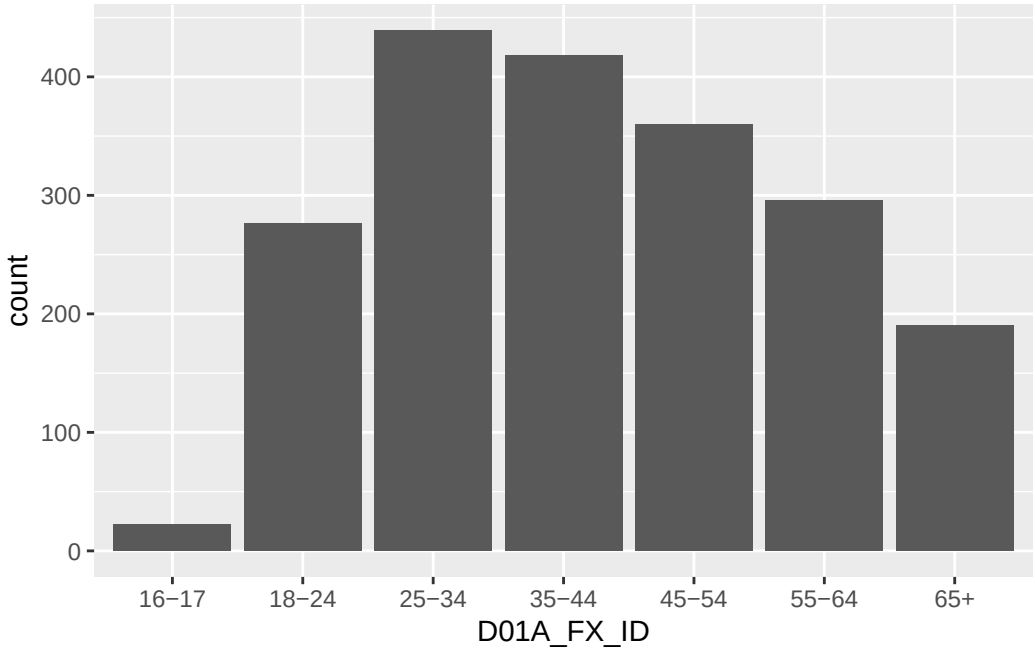
Geometria	Argumento	Descrição	Valores
`geom_point()`, `geom_jitter()`, entre outras	`size`	Muda o tamanho de pontos	Número
`geom_point()`, `geom_jitter()`, entre outras	`shape`	Muda a forma de pontos	Número entre 0 e 25

Ainda neste capítulo, veremos mais exemplos de aplicação destes argumentos.

4.3.2 Escalas

A próxima camada que vamos explorar são as escalas. No exemplo do gráfico de barras que fizemos anteriormente, por exemplo, usamos a função `geom_bar` para criar um gráfico de barras que mostra o número de pessoas entrevistadas pelo ESEB em diferentes faixas de idade. No entanto, o eixo X do gráfico mostra categorias fechadas. Se quisermos alterar isso, podemos usar a função `scale_x_discrete` (ou `scale_y_discrete`) para mapear os códigos das faixas de idade para as faixas de idade em si:

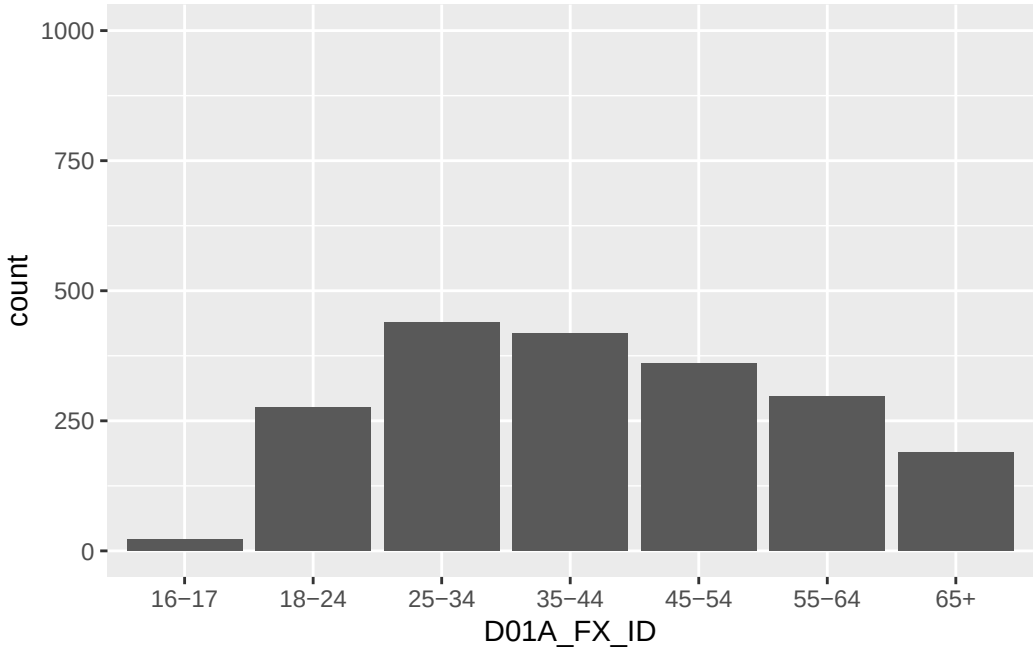
```
ggplot(eseb, aes(x = D01A_FX_ID)) +
  geom_bar() +
  scale_x_discrete(labels = c("16 e 17 anos" = "16-17",
                              "18 a 24 anos" = "18-24",
                              "25 a 34 Anos" = "25-34",
                              "35 a 44 Anos" = "35-44",
                              "45 a 54 Anos" = "45-54",
                              "55 a 64 anos" = "55-64",
                              "65 e mais" = "65+"))
```



Com `scale_`, passamos o argumento `labels =` que mapeia os códigos das faixas de idade para outros textos quaisquer. O resultado é um gráfico de barras que mostra o número de pessoas entrevistadas pelo ESEB em diferentes faixas de idade, mas com rótulos mais concisos.

A função correspondente para eixos numéricos, isto é, `scale_x_continuous` (ou `scale_y_discrete`, se for para o eixo Y), também permite mudar a escala do eixo Y de um gráfico. Um dos seus usos mais comuns é o de alterar os valores máximos e mínimos a serem exibidos no eixo desejado. Um exemplo alterando o valor máximo do eixo Y para 1000:

```
ggplot(eseb, aes(x = D01A_FX_ID)) +
  geom_bar() +
   scale_x_discrete(labels = c("16 e 17 anos" = "16-17",
                               "18 a 24 anos" = "18-24",
                               "25 a 34 Anos" = "25-34",
                               "35 a 44 Anos" = "35-44",
                               "45 a 54 Anos" = "45-54",
                               "55 a 64 anos" = "55-64",
                               "65 e mais" = "65+")) +
    scale_y_continuous(limits = c(0, 1000))
```



Para alterar nomes dos eixos, títulos e subtítulos, podemos usar a função `labs()` em mais uma camada:

```
ggplot(eseb, aes(x = D01A_FX_ID)) +
  geom_bar() +
   scale_x_discrete(labels = c("16 e 17 anos" = "16-17",
                               "18 a 24 anos" = "18-24",
                               "25 a 34 Anos" = "25-34",
                               "35 a 44 Anos" = "35-44",
                               "45 a 54 Anos" = "45-54",
                               "55 a 64 anos" = "55-64",
                               "65 e mais" = "65+")) +
    labs(title = "Entrevistados por faixa de idade",
         subtitle = "ESEB 2022",
         x = "Faixa de Idade",
         y = "N")
```

A função, como dá para notar pelo código anterior, tem argumentos que permitem mudar o título do gráfico, o subtítulo e os rótulos dos eixos X e Y, além de outros como `caption`, para adicionar nota de pé de gráfico.

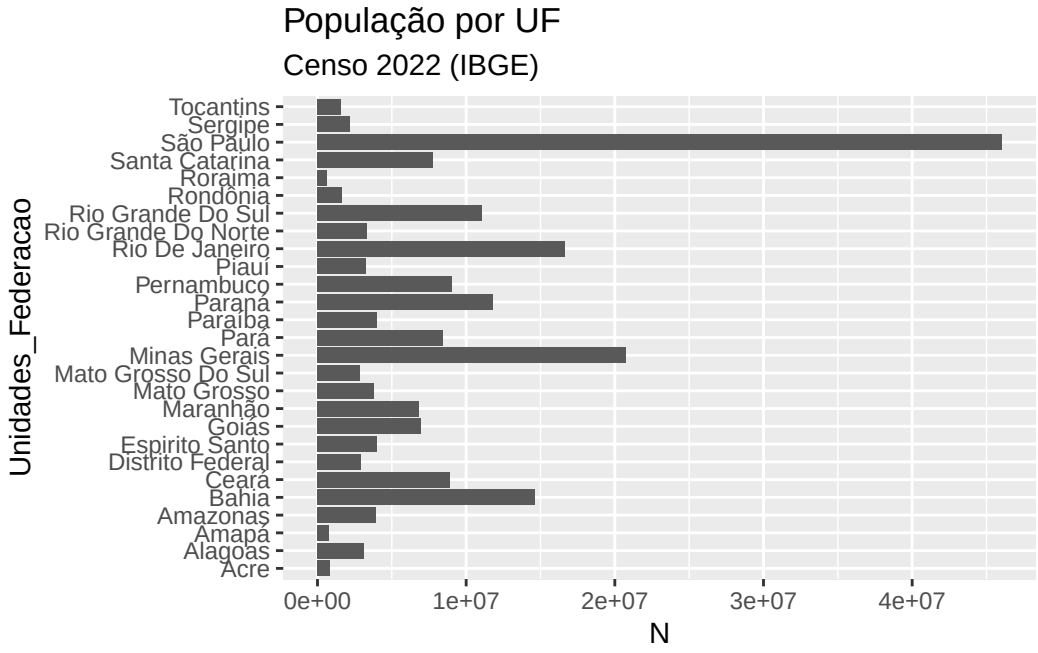
4.3.3 Coordenadas

Vamos agora fazer um novo gráfico com a mesma geometria, a barra, dessa vez com dados sobre o tamanho da população do Brasil por Unidade da Feração, conforme apurado pelo Censo de 2022. O arquivo com esses dados está na pasta de materiais complementares deste livro com o nome de `POP2022_Brasil.csv`. Podemos carregá-lo assim:

```
pop_uf <- read_delim("POP2022_Brasil.csv", delim = ";")
```

Isto feito, podemos fazer um gráfico de barras que mostra a população de cada Unidade da Federação do Brasil com uma camada nova, `coord_flip()`, que inverte os eixos X e Y do gráfico:

```
ggplot(pop_uf, aes(x = Unidades_Federacao, y=  POPULACAO)) +
  geom_col() +
  labs(title = "População por UF",
       subtitle = "Censo 2022 (IBGE)", y = "N") +
  coord_flip()
```



Repetimos basicamente o mesmo código do gráfico com os dados do ESEB. As principais alterações foram: 1) na definição da estética, com `aes()`, incluímos um eixo Y indicando o tamanho de cada barra representando as categorias do eixo X, e 2) na geometria, usamos`geom_col` justamente porque passamos os valores de tamanho das barras em `y`. A maior diferença, no

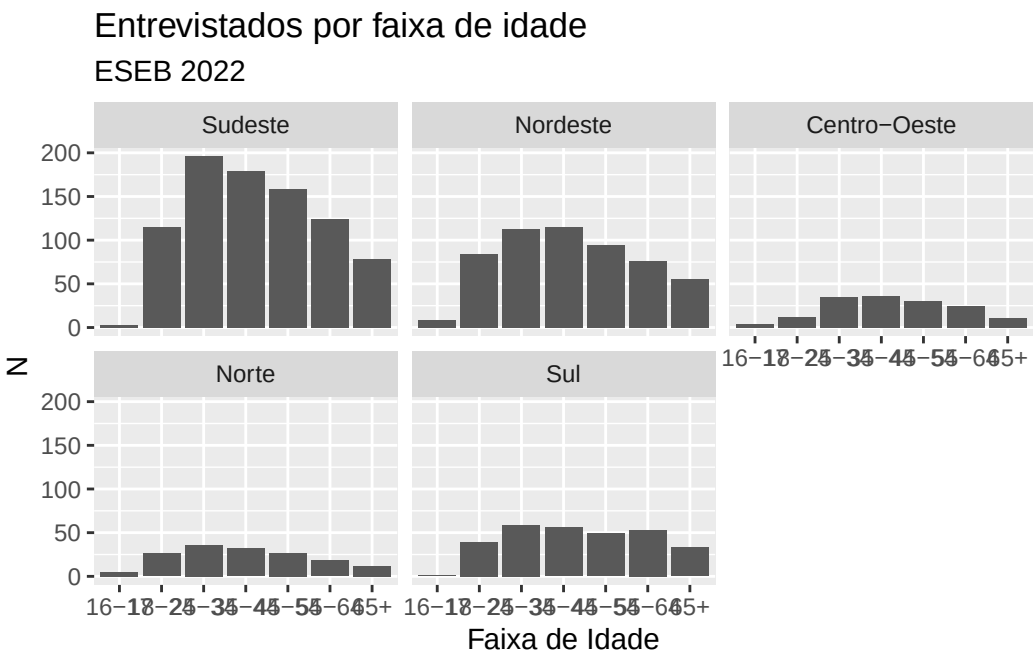
entanto, é a adição da camada `coord_flip()`, que inverte os eixos X e Y do gráfico, o que produz uma visualização na qual as barras aparecem deitadas, isto é, na horizontal.

Embora o `ggplot2` tenha outras funções `coord_`, `cood_flip` é, de longe, a mais comum. No entanto, há outras funções `coord_` que podem ser úteis em diferentes situações. A função `coord_polar`, por exemplo, é usada para criar gráficos circulares, enquanto `coord_map` é usada às vezes para criar mapas, ajustando coordenadas para representar adequadamente a superfície da Terra.

4.3.4 Facetas

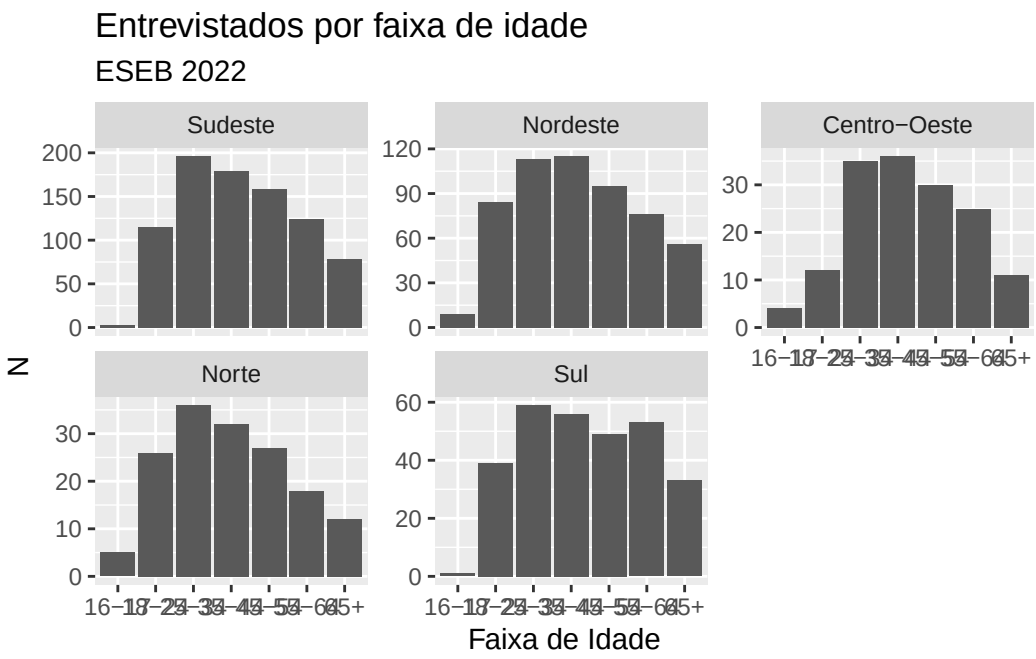
Imagine o seguinte problema: temos os dados de faixa etária da população do país em um gráfico de barras, mas queremos ver essa distribuição para cada uma das cinco regiões do país. Como fazer isso sem precisar criar cinco gráficos individualmente? Esse é o problema que as facetas resolvem. Especificamente, estas permitem que façamos gráficos em subgrupos do mesmo conjunto de dados indicando apenas qual variável deve ser usada para dividir os gráficos. No exemplo a seguir, usaremos uma dessas funções, `facet_wrap`, para refazer o nosso gráfico de barras com dados do ESEB, dessa vez quebrando ele por região do país (a variável que indica as regiões no banco `eseb` é `REG`):

```
ggplot(eseb, aes(x = D01A_FX_ID)) +
  geom_bar() +
   scale_x_discrete(labels = c("16 e 17 anos" = "16-17",
                               "18 a 24 anos" = "18-24",
                               "25 a 34 Anos" = "25-34",
                               "35 a 44 Anos" = "35-44",
                               "45 a 54 Anos" = "45-54",
                               "55 a 64 anos" = "55-64",
                               "65 e mais" = "65+")) +
    facet_wrap(~ REG) +
    labs(title = "Entrevistados por faixa de idade",
         subtitle = "ESEB 2022",
         x = "Faixa de Idade",
         y = "N")
```



Note que apenas uma linha de código foi adicionada ao gráfico anterior, `facet_wrap(~ REG)`, mas o resultado, agora, são cinco painéis, um para cada região do país. O detalhe de termos de usar um til, `~ REG`, pode ser lido como um indicativo de que queremos fazer um gráfico para cada nível da variável `REG` (é possível combinar variáveis usando `+`, embora seja um uso raro de facetas). Outro detalhe é que, por padrão, `facet_wrap` usa a mesma escala dos eixos X e Y para todos os painéis, mas é possível mudar isso com o argumento `scales`, que podem assumir os valores de "fixed", "free", "free_x" e "free_y". Para deixar o eixo Y de cada painel variar livremente, por exemplo, podemos fazer o seguinte:

```
ggplot(eseb, aes(x = D01A_FX_ID)) +
  geom_bar() +
   scale_x_discrete(labels = c("16 e 17 anos" = "16-17",
                               "18 a 24 anos" = "18-24",
                               "25 a 34 Anos" = "25-34",
                               "35 a 44 Anos" = "35-44",
                               "45 a 54 Anos" = "45-54",
                               "55 a 64 anos" = "55-64",
                               "65 e mais" = "65+")) +
    facet_wrap(~ REG, scales = "free_y") +
  labs(title = "Entrevistados por faixa de idade",
       subtitle = "ESEB 2022",
       x = "Faixa de Idade",
       y = "N")
```



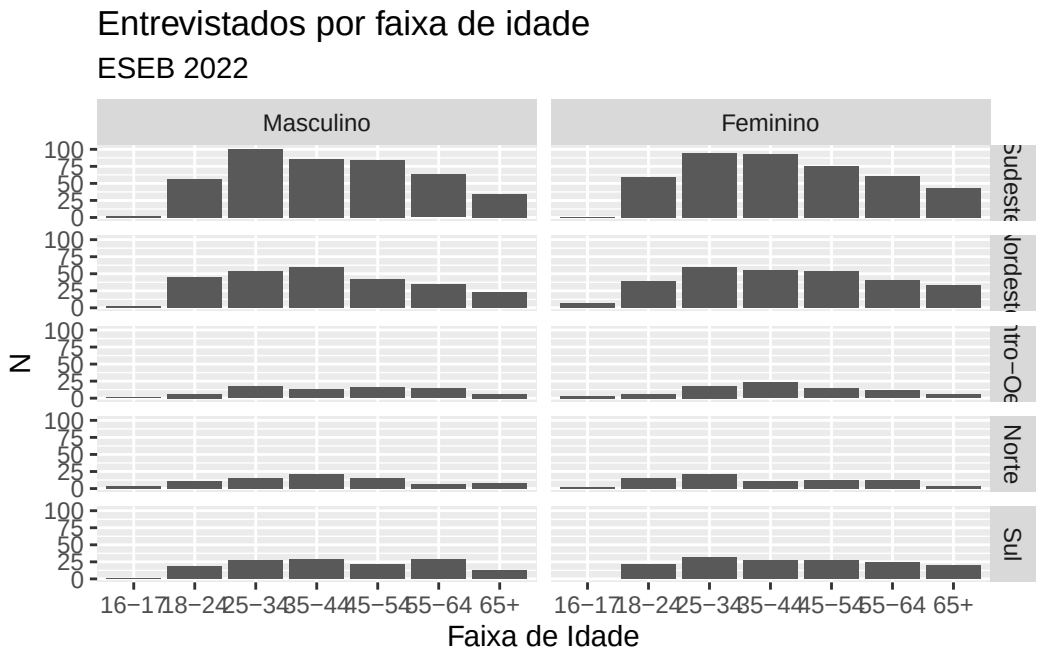
Por fim, outro argumento útil da função `facet_wrap` é `ncol`, que permite especificar o número de colunas que os painéis devem ter. No exemplo a seguir, usamos `ncol = 2` para fazer um gráfico com duas colunas de painéis:

```
ggplot(eseb, aes(x = D01A_FX_ID)) +
  geom_bar() +
   scale_x_discrete(labels = c("16 e 17 anos" = "16-17",
                               "18 a 24 anos" = "18-24",
                               "25 a 34 Anos" = "25-34",
                               "35 a 44 Anos" = "35-44",
                               "45 a 54 Anos" = "45-54",
                               "55 a 64 anos" = "55-64",
                               "65 e mais" = "65+")) +
    facet_wrap(~ REG, ncol = 2) +
  labs(title = "Entrevistados por faixa de idade",
       subtitle = "ESEB 2022",
       x = "Faixa de Idade",
       y = "N")
```

Facetas não é algo exclusivo de `facet_wrap`. Há outras funções `facet_` que podem ser usadas para fazer gráficos em subgrupos do mesmo conjunto de dados: `facet_grid`, por exemplo, é usada para fazer gráficos em subgrupos de duas variáveis, uma para o eixo X e outra para o

eixo Y. A função `facet_grid` é especialmente útil para fazer gráficos em subgrupos de duas variáveis categóricas, como no exemplo a seguir:

```
ggplot(eseb, aes(x = D01A_FX_ID)) +
  geom_bar() +
   scale_x_discrete(labels = c("16 e 17 anos" = "16-17",
                               "18 a 24 anos" = "18-24",
                               "25 a 34 Anos" = "25-34",
                               "35 a 44 Anos" = "35-44",
                               "45 a 54 Anos" = "45-54",
                               "55 a 64 anos" = "55-64",
                               "65 e mais" = "65+")) +  facet_grid(REG ~ D02) +
  labs(title = "Entrevistados por faixa de idade",
       subtitle = "ESEB 2022",
       x = "Faixa de Idade",
       y = "N")
```

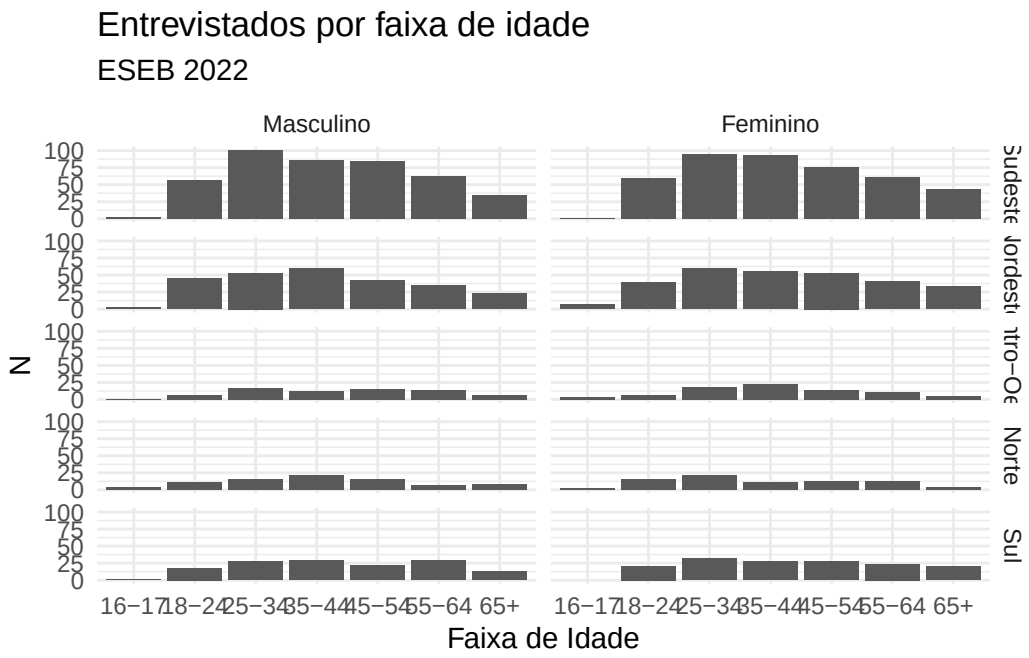


4.3.5 Temas

A última camada que abordaremos é, em geral, também a mais especial no `ggplot2`, aquela que torna o pacote especialmente popular entre cientistas de dados: a camada de temas.

Resumidamente, temas são funções que controlam vários recursos de exibição de uma visualização, como tamanho da fonte e cor do plano de fundo, rotação e cor de textos, grades e linhas de eixos, entre inúmeras outras coisas. Para fazer essas modificações, o `ggplot2` tem uma função genérica chama `theme()` que permite a customização de diferentes detalhes de um gráfico, além de outras funções que usam o prefixo `theme_` e aplicam customizações globais pré-estabelecidas em um gráficos. Começando por estas últimas funções, vamos usar a função `theme_minimal()` para criar um gráfico de barras com um tema minimalista (e, portanto, menos distrativo):

```
ggplot(eseb, aes(x = D01A_FX_ID)) +
  geom_bar() +
   scale_x_discrete(labels = c("16 e 17 anos" = "16-17",
                               "18 a 24 anos" = "18-24",
                               "25 a 34 Anos" = "25-34",
                               "35 a 44 Anos" = "35-44",
                               "45 a 54 Anos" = "45-54",
                               "55 a 64 anos" = "55-64",
                               "65 e mais" = "65+")) +  facet_grid(REG ~ D02) +
  labs(title = "Entrevistados por faixa de idade",
       subtitle = "ESEB 2022",
       x = "Faixa de Idade",
       y = "N") +
    theme_minimal()
```



Além desta, o `ggplot2` também tem outras funções `theme_` comuns: `theme_bw`, por exemplo, cria um gráfico com um tema em preto e branco, enquanto `theme_classic` cria um gráfico com um tema clássico. A função `theme_void`, por sua vez, cria um gráfico sem nenhum tema, isto é, sem grade, sem fundo, sem nada, o que pode ser útil em mapas, como veremos adiante.

Suponha agora que, em seu projeto de visualização de dados, as informações do eixo Y sejam desnecessárias. Podemos usar nossa função genérica `theme` para fazer essa edição. Como ja estamos usando um tema, a camada com a função genérica vai logo após o tema em uso.

```
ggplot(dados, aes(as.character(D01A_FX_ID))) +
  geom_bar() +
  labs(title = "Número de entrevistados por faixa de idade",
       subtitle = "ESEB - 2022",
       x = "Faixa de Idade",
       y = "N") +
  theme_classic() +
  theme(axis.line.y = element_blank(),
        axis.text.y = element_blank(),
        axis.ticks.y = element_blank(),
        axis.title.y = element_blank())
```

O código anterior é menos intuitivo e, por isso mesmo, merece maior atenção. A função `theme` é usada para fazer customizações em um gráfico, e a função `element_blank()` é usada para remover elementos de um gráfico. No caso, determinamos que os argumentos da função `theme` `axis.line.y`, `axis.text.y` e `axis.ticks.y` (que são usados para customizar a linha, o texto, os traços e o título do eixo Y, respectivamente), deverão ser removidos do gráfico. Mas e se, em vez de remove a linha do eixo Y do gráfico, quisermos mudar a cor dela? Podemos fazer isso com a função `element_line`, que é usada para customizar linhas de um gráfico:

```
ggplot(dados, aes(as.character(D01A_FX_ID))) +
  geom_bar() +
  labs(title = "Número de entrevistados por faixa de idade",
       subtitle = "ESEB 2022",
       x = "Faixa de Idade",
       y = "N") +
  theme_classic() +
  theme(axis.line.y = element_line(color = "red"))
```

De forma geral, para cada mínimo componente imaginável de uma tema em `ggplot2` há um argumento da função `theme` que pode ser usado para customizá-lo. A melhor forma de saber quais são esses argumentos é consultando a documentação do `ggplot2` ou usando a função `?` seguida do nome da função que você deseja usar. Assim como geometrias e outros, no entanto,

vale lembrar que a maioria das funções `theme_` têm argumentos que controlam cor, tamanho e forma dos elementos que criam. Avançar neste tópico é uma forma de aprimorar a estética de suas visualizações, mas também de torná-las mais eficientes e legíveis.

> Extensões
>
> Além das geometria nativas do `ggplot2`, é possível criar suas próprias extensões ou instalar a que outras pessoas criaram. O `tidyverse` organiza e disponiliza uma coleção bastatante rica dessas extenções: https://exts.ggplot2.tidyverse.org/gallery/.

4.4 Resumo

Neste capítulo, estudamos os principais conceitos para produção de gráficos no R usando o pacote ggplot. Utilizamos alguns dados de exemplo para compreender o funcionamento do pacote `ggplot` e o processo de adição de camadas para criar nossos projetos gráficos. Embora não tenhamos explorado todas as possibilidades aqui, abrimos os principais caminhos seguidos para produzirmos visualizações simples e eficientes.

Exercícios

> Arquivos necessários
>
> Para realizar estes exercícios, será necessário baixar os arquivos que estão na pasta de materiais complementares deste livro e salvá-los na pasta de trabalho do R.

1. Gráficos de barras

A base de dados do ESEB de 2022, disponível no arquivo `eseb2022.sav`, tem uma variavel chamada de `Q10P2b` que indica o voto das pessoas entrevistadas para Presidente no segundo turno das eleições de 2022 disputada entre Lula e Bolsonaro. Usando essa variável, crie um gráfico de barras que o total de votos de cada candidato na amostra do Eseb.

2. Gráficos de barras com facetas

Usando a mesma variável `Q10P2b` e a variável `REG`, que indica região, da base do Eseb, crie agora um gráfico de barras que mostra o total de votos de cada candidato na amostra do Eseb, mas com painéis para cada região do país. Use duas colunas para organizar os painéis.

3. Gráficos de histograma

Para este exercício, carregue novamente dados do Censo de 1872, que carregamos no Capítulo 2, disponíveis no seguinte link:

- https://raw.githubusercontent.com/izabelflores/Censo_1872/main/Censo_1872_dados_tidy_versao2.csv

Com os dados carregados, crie um gráfico que mostre a distribuição das pessoas consideradas de raça preta, segundo a denominação do Censo, indicada pela variável `Raças_Preto`.

4. Gráficos de barras com totais calculados

Usando a mesma base do Censo de 1872, use funções de manipulação de dados para calcular o total de pessoas (variável `Total_Almas`) por província no Brasil da época (a variável `PrimeiroDeProvincia` indica as províncias). Com esses totais calculados, crie um gráfico de barras que mostra o total de pessoas por província (dica: este gráfico deve indicar em `aes` um eixo X e um eixo Y).

5. Temas e customizações

Com o mesmo gráfico do exercício anterior, aplique um tema de sua escolha e faça customizações no gráfico, como mudar cores, tamanhos e formas de elementos, e remover ou adicionar elementos do gráfico. Use comentários para explicar as modificações realizadas.

5 Análise

Depois de carregar, manipular e visualizar as informações de uma base de dados, temos uma boa noção do que ela nos tem a dizer: sabemos quais são as suas variáveis, como se distribuem e como se relacionam. Em pesquisas acadêmicas, entretanto, isso raramente basta. Frequentemente, precisamos quantificar as características de nossos dados e testar hipóteses sobre as relações entre as variáveis observadas. Produzir esses resumos quantitativos usando R é o que cobriremos neste capítulo final.

Começaremos o nosso percurso passando brevemente pelo básico sobre estatísticas descritivas, isto é, medidas numéricas que servem para resumir as características de uma variável. Em seguida, veremos como estimar modelos de regressão linear simples em R para testar hipóteses sobre a relação entre duas variáveis. Por fim, mostraremos algumas ferramentas que facilitam a exibição e exportação dessas e de outras análises. Seguindo nossa abordagem geral, focaremos no uso de prático de algumas poucas ferramentas que, por serem versáteis, podem ser usadas em uma grande variedade de projetos de análise de dados.

Antes de avançar, certifique-se de ter os pacotes e bases de dados necessários. Para criar e exportar tabelas em formato adequado para publicações, usaremos os pacotes `gt` e `modelsummary`, que podem ser instalados com o seguinte código:

```
install.packages("modelsummary")
install.packages("gt")
```

Para carregar os pacotes que usaremos, basta executar:

```
library(gt)
library(tidyverse)
library(modelsummary)
```

Como forma de aplicar os conceitos que veremos neste capítulo, usaremos uma pequena base de dados que contém os votos válidos e o percentual de gastos de campanha de candidaturas aos governos estaduais em 2022 no primeiro turno em alguns estados específicos (GO, MG, RJ e PR). Esta base está disponível no repositório de dados deste livro em formato delimitado por ponto e vírgula (`governadores.csv`). Para carregá-la, podemos usar a função `read_csv2()` do pacote:

```
gov <- read_csv2("governadores.csv")
```

Executado o código, a base carregada salva no objeto `gov` tem 14 linhas com as seguintes variáveis:

- `uf`: unidade da federação;
- `candidatura`: nome da(o) candidata(o);
- `partido`: partido da candidatura;
- `pct_gastos`: percentual gasto na campanha da candidatura em relação às demais candidaturas;
- `pct_votos`: percentual de votos válidos no 1º turno da eleição para governo do estado.

5.1 Estatísticas descritivas

Como vimos no capítulo anterior, uma das primeiras coisas que fazemos ao analisar um conjunto de dados é inspecionar suas características, o que pode ser feito por meio de gráficos. Há um problema, no entanto: gráficos não nos oferecem descrições precisas das características de uma variável – o que normalmente é demandado em publicações acadêmicas. Embora gráficos sejam ferramentas essenciais em qualquer projeto, saber como resumir as características de uma variável de forma precisa é um complemento necessário.

A chave para descrever de forma precisa nossas variáveis é usarmos algumas medidas numéricas que resumam a distribuição dos valores que elas assumem. Pense, por exemplo, em uma variável que meça a altura de um grupo de pessoas. Os valores dessa variável poderiam ser estes:

Tabela 5.1: Altura fictícia de 6 pessoas (em metros)

Pessoa	Altura (metros)
1	1.60
2	1.70
3	1.75
4	1.80
5	1.85
6	1.90

Uma forma de descrever essa informação seria, naturalmente, listar cada uma das alturas individuais, isto é, dizer que a pessoa 1 tem 1,60, a 2 tem 1,70, e assim por diante. Caso tenhamos muitas observações, contudo, a descrição deixa de fazer sentido e, no lugar, precisamos de algo mais sintético. Um exemplo? Note que a altura máxima que vimos acima é 1.9 e a mínima é

1.6. Essas duas medidas já nos dão uma ideia geral de que todas as informações estão contidas entre estes limites mínimo e máximo. Mais, podemos calcular a média das alturas, que é 1.8, para ter uma ideia de qual é altura "típica" de uma pessoa da turma; e, finalmente, podemos calcular o desvio padrão, que é 0.11, para ter uma ideia do quanto as alturas individuais se distanciam da média. Munido dessas informações, temos uma boa ideia da distribuição das alturas do grupo de pessoas, tenha ele 6 ou 600 integrantes.

Na Estatística, tais medidas são conhecidas como *estatísticas descritivas* e são usadas tanto para descrever tendências centrais (e.g., média, mediana, moda) quanto para descrever a dispersão dos valores (e.g., desvio padrão, variância, mínimo, máximo, etc.). A depender do tipo de variável que estamos analisando, algumas dessas medidas são mais adequadas que outras: para variáveis numéricas (i.e., contínuas), por exemplo, a média e o desvio padrão são medidas geralmente são utilizadas; já para variáveis categóricas (e.g., sexo), a moda, isto é, o valor mais frequente, é uma medida útil.

> **Estatísticas descritivas**
>
> Uma *estatística descritiva* é um número único que condensa uma propriedade de uma variável (Kellstedt e Whitten 2018, cap. 6). Estatísticas descritivas comuns incluem a média, a mediana, a moda, o desvio padrão, a variância, o valor mínimo, o valor máximo, entre outros.

Em R, podemos calcular essas e outras estatísticas descritivas usando funções específicas. Para exemplificar, vamos calcular algumas estatísticas descritivas para a variável `pct_gastos` da base `gov`. Para calcular a média, que já vimos no Capítulo 3, usamos a função `mean()`:[1]

```
mean(gov$pct_gastos)
```

```
[1] 27.10714
```

Outras estatísticas descritivas podem ser calculadas de forma similar. Seguem algumas das mais comuns:

- `median()`: mediana, que é o valor que divide a distribuição em duas partes iguais;
- `var()`: variância, que é a média dos quadrados dos desvios em relação à média;
- `sd()`: desvio padrão, que é a raiz quadrada da variância;
- `min()`: mínimo;
- `max()`: máximo;
- `range()`: intervalo, que é a diferença entre o máximo e o mínimo;

[1] Usamos o indexador `$` para acessar a variável `pct_gastos` da base `gov`. Caso tenha dúvidas sobre esse operador, veja a seção correspondente no Capítulo 1.

Por exemplo:

```
median(gov$pct_gastos)
```

```
[1] 27.84
```

```
sd(gov$pct_gastos)
```

```
[1] 15.68703
```

```
min(gov$pct_gastos)
```

```
[1] 2.62
```

```
max(gov$pct_gastos)
```

```
[1] 61.72
```

Com esse conjunto de estatísticas calculadas, já temos uma boa ideia da distribuição da variável `pct_gastos` da base `gov`. Para termos uma ideia mais completa, podemos usar a função `summary()`, que calcula várias estatísticas descritivas de uma variável de uma só vez:

```
summary(gov$pct_gastos)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2.62   16.25   27.84   27.11   36.35   61.72
```

Os números anteriores formam uma espécie de retrato: média e mediana indicam que o centro da distribuição está em torno de 27, isto é, candidaturas típicas tendem a gastar em torno desse valor percentual; valores mínimo e máximo são de 2.62 e 61.72, respectivamente, o que nos mostra os limites de gastos registrados nos nossos dados; e, finalmente, os primeiro e terceiro quantis mostram que a maioria dos gastos percentuais registrados na nossa base está situada entre os valores de 16.245 e 36.355.

> **Quantis**
>
> *Quantis*, ou *quartis*, são estatísticas descritivas que dividem uma distribuição em quatro partes. O *primeiro quartil*, às vezes chamado de Q1, divide todos os valores de uma variável em duas partes iguais, sendo que 25% dos valores estão abaixo dele e, 75%, acima; Q3, por sua vez, é o valor que divide a distribuição com 75% dos valores abaixo dele e 25% acima; e, finalmente, o Q2 é a mediana, que divide a distribuição com 50% dos valores acima, e 50% abaixo, dela.

5.1.1 Calculando múltiplas estatísticas descritivas

Calcular várias estatísticas de uma só vez é algo normal em pesquisas. Por exemplo, suponha que queremos calcular a média e o desvio padrão das variáveis `pct_gastos` e `pct_votos` da base `gov`. Para tanto, podemos usar a função `summarise()` do pacote `dplyr`, que já vimos no Capítulo 3:

```
gov |>
    summarise(media_gastos = mean(pct_gastos),
              desvio_gastos = sd(pct_gastos),
              media_votos = mean(pct_votos),
              desvio_votos = sd(pct_votos))
```

```
# A tibble: 1 x 4
  media_gastos desvio_gastos media_votos desvio_votos
         <dbl>         <dbl>       <dbl>        <dbl>
1         27.1          15.7        28.2         22.8
```

5.1.2 Estatísticas descritivas por grupo

Outra tarefa comum é calcular estatísticas descritivas de uma variável para grupos específicos. Imagine, por exemplo, que queremos calcular a média e desvio padrão das variáveis `pct_gastos` e `pct_votos` para cada um dos três estados incluídos na base `gov`. Como fazemos isso? Simples: por meio das funções `group_by()` e `summarise()` do pacote `dplyr`, que já vimos no Capítulo 3. Depois de agruparmos a base por `uf`, calculamos as estatísticas para cada grupo com `summarise`:

```
gov |>
    group_by(uf) |>
    summarise(media_gastos = mean(pct_gastos),
              desvio_gastos = sd(pct_gastos),
```

```
              media_votos = mean(pct_votos),
              desvio_votos = sd(pct_votos))
```

```
# A tibble: 4 x 5
  uf    media_gastos desvio_gastos media_votos desvio_votos
  <chr>        <dbl>         <dbl>       <dbl>        <dbl>
1 GO            24.6          14.6        24.7         19.6
2 MG            28.9          11.6        32.8         24.6
3 PR            31.7          26.0        32.6         34.2
4 RJ            24.8          16.5        24.8         24.6
```

Note que a principal diferença aqui foi o uso de `group_by(uf)` para dizer ao R que operações de resumo deveriam ser feitas para cada um dos grupos definidos pela variável `uf`.

5.1.3 Transformando tabelas de estatísticas descritivas

Quando calculamos estatísticas descritivas para grupos, o resultado é uma tabela com uma linha para cada grupo e uma coluna para cada estatística calculada. Se quisermos alterar essa disposição de informações, podemos usar os princípios *tidy* que vimos no Capítulo 3. Por exemplo, para obtermos uma tabela com uma linha para cada estatística calculada e uma coluna para cada grupo, podemos usar as funções `pivot_longer()` e `pivot_wider()` do pacote `tidyr` em duas etapas. Primeiro, usamos `pivot_longer()` para alongar as colunas com estatísticas:

```
tab_longa <- gov |>
    group_by(uf) |>
    summarise(media_gastos = mean(pct_gastos),
              desvio_gastos = sd(pct_gastos),
              media_votos = mean(pct_votos),
              desvio_votos = sd(pct_votos)) |>
    pivot_longer(cols = -uf, names_to = "estatistica", values_to = "valor")

tab_longa
```

```
# A tibble: 16 x 3
   uf    estatistica   valor
   <chr> <chr>         <dbl>
 1 GO    media_gastos   24.6
 2 GO    desvio_gastos  14.6
 3 GO    media_votos    24.7
```

```
 4 GO    desvio_votos   19.6
 5 MG    media_gastos   28.9
 6 MG    desvio_gastos  11.6
 7 MG    media_votos    32.8
 8 MG    desvio_votos   24.6
 9 PR    media_gastos   31.7
10 PR    desvio_gastos  26.0
11 PR    media_votos    32.6
12 PR    desvio_votos   34.2
13 RJ    media_gastos   24.8
14 RJ    desvio_gastos  16.5
15 RJ    media_votos    24.8
16 RJ    desvio_votos   24.6
```

Tudo o que fizemos aqui foi indicar que queremos manter a coluna `uf` e alongar as demais (`cols = -uf`) para que cada uma delas vire uma linha (`names_to = "estatistica"`) e que seus valores sejam posicionados em uma nova coluna (`values_to = "valor"`). Isso feito, podemos usar `pivot_wider()` para transformar a coluna `uf` em múltiplas colunas, uma para cada estado:

```
tab_final <- tab_longa |>
    pivot_wider(names_from = uf, values_from = valor)

tab_final
```

```
# A tibble: 4 x 5
  estatistica      GO    MG    PR    RJ
  <chr>         <dbl> <dbl> <dbl> <dbl>
1 media_gastos   24.6  28.9  31.7  24.8
2 desvio_gastos  14.6  11.6  26.0  16.5
3 media_votos    24.7  32.8  32.6  24.8
4 desvio_votos   19.6  24.6  34.2  24.6
```

Nem sempre é intuitivo saber quando usar `pivot_longer()` e `pivot_wider()`, e em qual ordem, mas, com um pouco de prática, é fácil pegar os padrões mais recorrentes de transformação de acordo com os princípios *tidy* – caso queira praticar mais um pouco, os exercícios do Capítulo 3 são um bom lugar para começar.

5.1.4 Exportando resultados

Uma vez calculadas algumas estatísticas descritivas, a sequência natural é exportá-las para um arquivo de texto ou planilha para uso posterior. Uma forma fácil de fazer isso é por meio

da função `write_csv()` do pacote `readr`, que já vimos no Capítulo 2:

```
tab_final |>
    write_csv("minha_tabela.csv")
```

O ponto negativo dessa abordagem é que, por padrão, a função `write_csv()` não aplica nenhuma formatação ao resultado. É por essa razão que sugerimos usar o pacote `gt` – um pacote que facilita a criação modular de tabelas em HTML, LaTeX ou documentos de texto – para salvar estatísticas descritivas. Exportar a tabela anterior com `gt` é questão de aplicar a função `gt()`, para criar um objeto `gt`, e, em seguida, usar a função `gtsave()` para exportá-lo em formato HTML:

```
tab_final |>
    gt() |>
    gtsave("minha_tabela.html")
```

Abrindo o arquivo `minha_tabela.html` em um navegador de internet (clicando no arquivo com o botão direito do *mouse* e selecionando a opção "Abrir com..."), obtemos o seguinte resultado:

estatistica	GO	MG	PR	RJ
media_gastos	24.56250	28.87333	31.74667	24.84750
desvio_gastos	14.62260	11.61861	25.97810	16.50908
media_votos	24.70000	32.83000	32.65000	24.85000
desvio_votos	19.55345	24.55244	34.23450	24.60989

A tabela exportada tem uma boa formatação, ainda que falte ajustar detalhes como o excesso de casas decimais e a ausência de título e fonte. Para esses e outros ajustes finos, o pacote `gt` oferece uma série de funções auxiliares. Por exemplo, para manter apenas uma casa decimal e adicionar títulos, podemos usar as funções `fmt_number()`, `tab_header()` e `tab_source_note()` da seguinte forma:

```
tab_final |>
    gt() |>
    fmt_number(decimals = 1) |>
    tab_header(title = "Estatísticas descritivas de gastos e votos por estado") |>
    tab_source_note(source_note = "Fonte: TSE.")
```

Estatísticas descritivas de gastos e votos por estado

estatistica	GO	MG	PR	RJ
media_gastos	24.6	28.9	31.7	24.8
desvio_gastos	14.6	11.6	26.0	16.5
media_votos	24.7	32.8	32.6	24.9
desvio_votos	19.6	24.6	34.2	24.6

Fonte: TSE.

E, usando um pouco de manipulação de dados, conseguimos renomear a coluna de `estatisticas` e seus valores para algo mais adequado:

```
tab_formatada <- tab_final |>
    rename(Estatística = estatistica) |>
        mutate(Estatística = case_when(Estatística == "media_gastos" ~ "Média de gastos",
                                       Estatística == "desvio_gastos" ~ "Desvio de gastos",
                                       Estatística == "media_votos" ~ "Média de votos",
                                       Estatística == "desvio_votos" ~ "Desvio de votos")) |>
    gt() |>
    fmt_number(decimals = 1) |>
    tab_header(title = "Estatísticas descritivas de gastos e votos por estado") |>
    tab_source_note(source_note = "Fonte: TSE.")

tab_formatada
```

Estatísticas descritivas de gastos e votos por estado

Estatística	GO	MG	PR	RJ
Média de gastos	24.6	28.9	31.7	24.8
Desvio de gastos	14.6	11.6	26.0	16.5
Média de votos	24.7	32.8	32.6	24.9
Desvio de votos	19.6	24.6	34.2	24.6

Fonte: TSE.

A tabela resultante tem uma formatação muito melhor. Podemos agora exportá-la para um documento de texto, no formato RTF:

```
tab_formatada |>
    gtsave("minha_tabela.rtf")
```

O arquivo salvo adapta a formatação que vimos acima para o formato de texto que pode ser aberto em qualquer editor – o que facilita o trabalho de incluir resultados do R em um documento de Word, por exemplo.

5.1.5 Criando tabelas automaticamente com `modelsummary`

O pacote `gt`, visto anteriormente, permite inúmeras customizações em uma tabela, várias delas que sequer mostramos[2] – o custo é ter de aprender a usar uma série de funções e argumentos. Para quem não quer se preocupar com isso, uma alternativa econômica é usar o pacote `modelsummary`, que calcula estatísticas descritivas e cria tabelas automaticamente a partir de uma base de dados. Para criar uma tabela descritiva da base `gov`, por exemplo, basta usar a função `datasummary_skim()` como fizemos abaixo:

```
datasummary_skim(gov)
```

Como é possível notar, `datasummary_skim()` automaticamente detecta quais colunas na nossa base são numéricas e, partir disso, calcula e reporta uma série de estatísticas úteis, como número de linhas; números de *missings*; média; entre outas. Para termos mais controle sobre o resultado, podemos usar a função `datasummary()` especificando quais variáveis e quais estatísticas queremos calcular:

```
datasummary(pct_gastos + pct_votos ~ Mean + Median, data = gov)
```

No código anterior, usamos `pct_gastos + pct_votos` para indicar quais variáveis da base `gov` deveriam ser incluídas na tabela, e `Mean + Median` para indicar quais estatísticas deveriam ser calculadas para cada uma delas. Adaptando um pouco essa fórmula, podemos criar uma tabela com estatísticas descritivas para cada um dos estados da base `gov`:

```
datasummary(pct_gastos + pct_votos ~ Mean * uf, data = gov)
```

O resultado, agora, é uma tabela com uma linha para cada estado e uma coluna para cada estatística calculada. Para exportar essa tabela para um arquivo de texto, usamos apenas um argumento adicional, `output`:

```
datasummary(pct_gastos + pct_votos ~ Mean * uf, data = gov, output = "tabela.txt") # Texto
datasummary(pct_gastos + pct_votos ~ Mean * uf, data = gov, output = "tabela.rtf") # RTF
datasummary(pct_gastos + pct_votos ~ Mean * uf, data = gov, output = "tabela.html") # HTML
```

[2] Para saber mais sobre todas as funcionalidades do `gt`, nossa recomendação é consultar a documentação oficial do pacote em https://gt.rstudio.com.

	Unique	Missing Pct.	Mean	SD	Min	Media
pct_gastos	14	0	27.1	15.7	2.6	27.8
pct_votos	14	0	28.2	22.8	2.1	25.7
		N	%			
uf	GO	4	28.6			
	MG	3	21.4			
	PR	3	21.4			
	RJ	4	28.6			
candidatura	ALEXANDRE KALIL	1	7.1			
	CARLOS ALBERTO DIAS VIANA	1	7.1			
	CARLOS ROBERTO MASSA JUNIOR	1	7.1			
	CLÁUDIO BOMFIM DE CASTRO E SILVA	1	7.1			
	GUSTAVO MENDANHA MELO	1	7.1			
	MARCELO RIBEIRO FREIXO	1	7.1			
	PAULO GUSTAVO GANIME ALVES TEIXEIRA	1	7.1			
	RICARDO CRACHINESKI GOMYDE	1	7.1			
	ROBERTO REQUIÃO DE MELLO E SILVA	1	7.1			
	RODRIGO NEVES BARRETO	1	7.1			
	ROMEU ZEMA NETO	1	7.1			
	RONALDO RAMOS CAIADO	1	7.1			
	VITOR HUGO DE ARAUJO ALMEIDA	1	7.1			
	WOLMIR THEREZIO AMADO	1	7.1			
partido	NOVO	2	14.3			
	PATRIOTA	1	7.1			
	PDT	2	14.3			
	PL	3	21.4			
	PSB	1	7.1			
	PSD	2	14.3			
	PT	2	14.3			
	UNIÃO	1	7.1			

	Mean	Median
pct_gastos	27.11	27.84
pct_votos	28.19	25.72

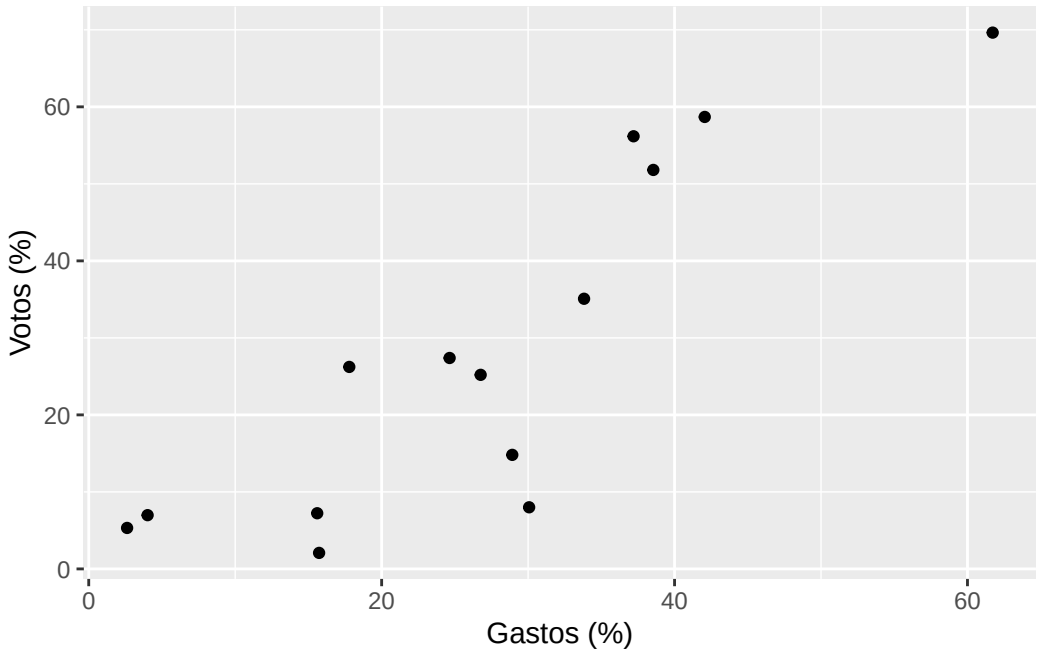
	GO	MG	PR	RJ
pct_gastos	24.56	28.87	31.75	24.85
pct_votos	24.70	32.83	32.65	24.85

Uma nota final: ainda que tenha opções de customização que não cobrimos aqui, `modelsummary` não produz resultados iniciais tão bem acabados quanto o `gt`, mas é uma boa ferramenta para criar tabelas descritivas de forma rápida. A depender do uso, uma opção é começar a exploração de dados usando `datasummary()` para, se preciso, refinar resultados para exportação com `gt()` e suas funções auxiliares.

5.2 Modelos de regressão linear simples

Em análises de dados, é algo comum querermos investigar se existe relação entre duas ou mais variáveis numéricas. Um exemplo: será que quem gasta mais em sua campanha eleitoral faz mais votos? Com os dados das eleições para governos de estado na base `gov`, podemos ilustrar essa relação com um gráfico de dispersão:

```
gov |>
    ggplot(aes(x = pct_gastos, y = pct_votos)) +
    geom_point() +
    labs(x = "Gastos (%)", y = "Votos (%)")
```



Como se depreende do gráfico, parece haver uma associação positiva entre gasto de campanha e votos: pontos com maiores valores na variável `pct_gastos` têm maiores valores em `pct_votos`, no geral. Essa é uma relação sugestiva, mas quão forte ela é? Será que é algo que aconteceu por acaso?

Ainda que gráficos sejam úteis para nos ajudar a detectar padrões em meio a pontos dispersos, o fato é que eles não servem para responder a questões como as feitas acima. Em vez disso, precisamos de uma ferramenta para estimar, de forma precisa, o efeito predito de gastar mais ou menos na campanha no número de votos obtidos pelas candidaturas. Dito de outro modo, precisamos de um modelo.

Antes de entrarmos em maiores detalhes, vamos supor que um bom modelo, nesse caso, seja uma reta. Podemos começar traçando uma arbitrariamente nos dados para ver o resultado.

este exemplo, usamos a geometria `geom_abline` para desenhar uma reta a partir dos argumentos `slope` e `intercept`, que indicam, respectivamente, a inclinação e o ponto em que a reta cruza o valor de 0 no eixo X. Em termos mais simples: `intercept` indica qual é o valor de Y (`pct_votos`) quando X é zero (`pct_gastos = 0`); `slope` indica o quanto Y aumenta ou diminui quando X aumenta em uma unidade. No nosso exemplo, a reta que desenhamos tem inclinação de 2.3 e intercepta o eixo Y em -3, indicando que o valor de Y quando X é zero é -3 e que, a cada unidade que X aumenta, Y aumenta 2.3 unidades. Assim, podemos dizer, com base nesse modelo, que candidaturas que gastaram 20% de todas as receitas de campanha no seu estado esperariam, em média, obter 40% dos votos – basta acompanhar a reta para encontrar esses valores, ou fazer a conta:

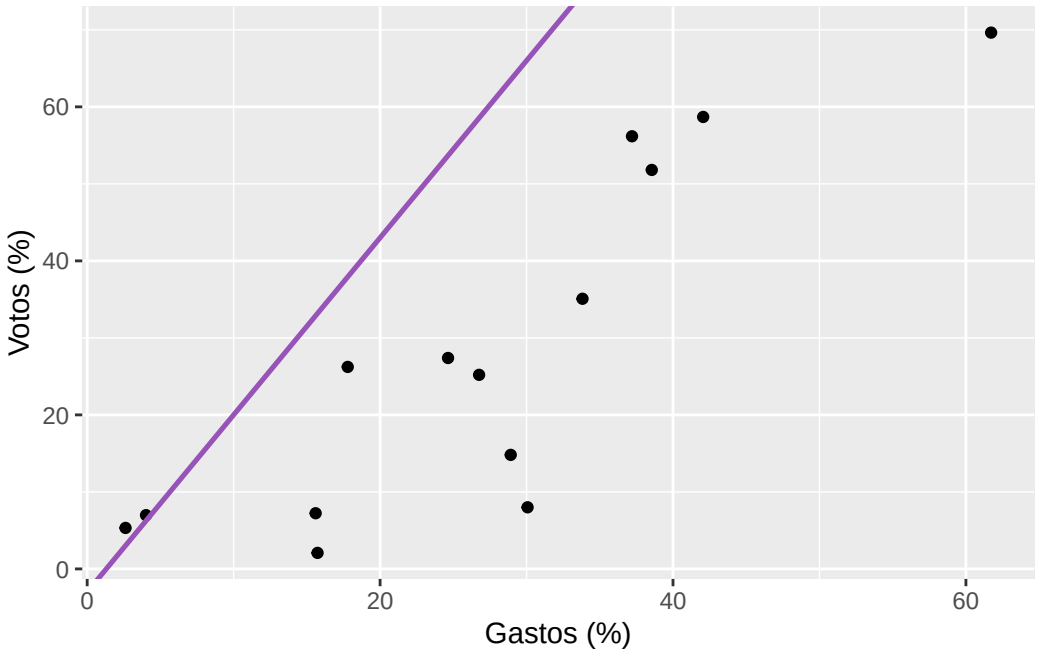


Figura 5.1: Relação entre gastos de campanha e votos

```
(-3) + 2.3 * 20
```

```
[1] 43
```

A reta que desenhamos parece acompanhar a relação entre gastos de campanha e votos. Candidaturas que gastaram mais tiveram mais votos, no geral, e a inclinação ascendente da reta que traçamos captura essa tendência. De qualquer forma, muitos pontos estão distantes dessa reta arbitrária que criamos e, mais que isso, a disposição dos pontos no gráficos sugere que uma menor inclinação seria melhor. Não é necessário treinamento quantitativo para perceber que não temos um bom modelo.

5.2.1 Mínimos quadrados ordinários

Para conseguir um modelo melhor, precisamos de uma reta que esteja mais próxima da maioria dos pontos, ou seja, que minimize a distância entre os pontos no gráfico e a reta. Podemos pensar, por exemplo, em um critério simples: calcular a distância vertical entre os pontos e a reta, isto é, a distância de cada ponto no eixo Y em relação à reta que traçamos. Em termos visuais, esse critério equivaleria a avaliar a nossa reta arbitrária da seguinte forma.

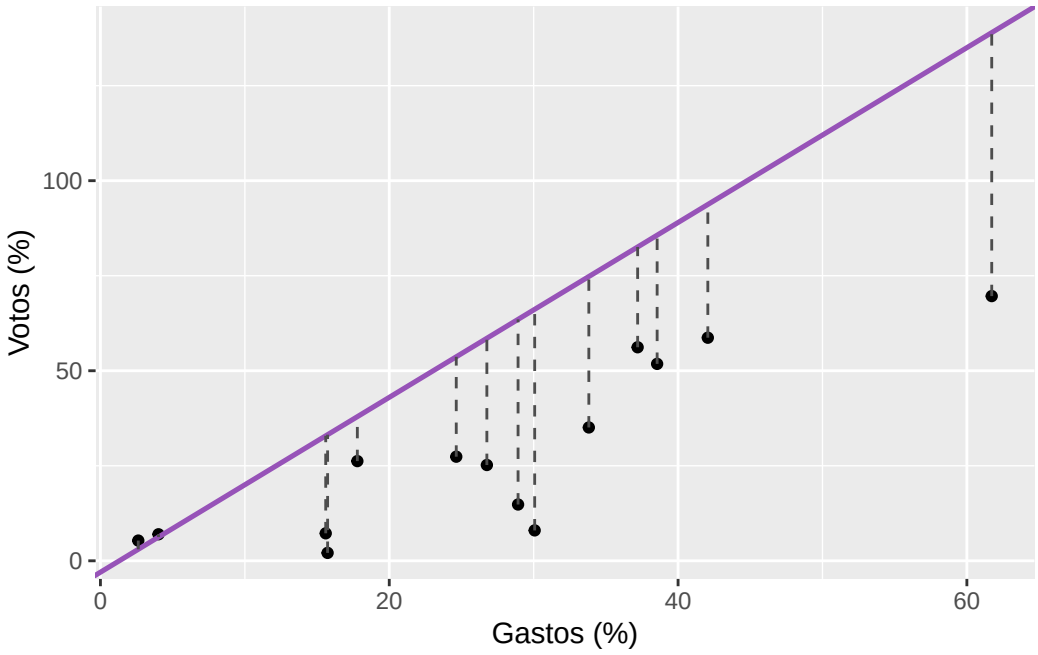


Figura 5.2: Distância vertical entre pontos e reta

Com esse gráfico, vemos a distância de cada ponto em relação à reta que usamos como nosso modelo, reforçando a visão de que ele não resume adequadamente a relação entre gastos de campanha e votos. Precisamos, portanto, de um método para minimizar essas distâncias. É aqui que entra a regressão por *Mínimos Quadrados Ordinários* (MQO), que relaciona duas variáveis de forma a encontrar uma reta que minimize a distância entre pontos e a reta encontrada.[3]

A ideia central aqui é encontrarmos um intercepto (o valor de Y quando X é igual a 0) e uma inclinação (quanto ela aumenta ou diminui quando andamos uma casa no eixo X) que minimize a distância que vimos entre pontos e reta – na verdade, como o nome do modelo indica (Mínimos Quadrados), ele faz isso calculando a distância de cada ponto no eixo Y em relação à reta ao quadrado, penalizando observações mais distantes na hora de estimar o melhor modelo.

Em R, a principal função que usaremos para estimar modelos lineares é a `lm`, de *linear model*. Ela já vem por padrão em qualquer instalação do R e pode ser usada com apenas dois argumentos: a fórmula do nosso modelo, estipulando qual variável é a dependente (e.g., qual variável queremos predizer) e qual, ou quais, variáveis usaremos para predizer a variação na dependente (às vezes essas variáveis também são chamadas de preditores); e o banco de dados os estão essas variáveis. Vamos ver mais detidamente cada argumento.

[3]Para uma explicação intuitiva de como funciona o método de MQO, ver Kellstedt e Whitten (2018).

Uma fórmula, em primeiro lugar, é o meio que usamos para especificar a relação entre variáveis. No nosso exemplo hipotético da relação entre gastos de campanha e votos, uma fórmula que explicita nosso modelo seria o seguinte:

```
pct_votos ~ pct_gastos
```

O que pode ser lido como: votos válidos de cada candidato ou candidata é predita pelo quanto ele(a) gastou em sua campanha. Para dizer de outro modo, tudo o que vem depois de `~` serve para explicar o que vem antes. Em R, expressões como estas, que já vimos em outros lugares deste livro, são chamadas de *fórmulas* e são usadas para especificar modelos de regressão linear simples e múltipla (experimente rodar `?formula` para saber mais sobre elas).

Esse jeito de expressar fórmulas é direto: `pct_votos` é nossa variável dependente porque ela vem antes do `~`; dado que `pct_gastos` segue depois disso, ela é nossa variável independente. Declarar essa relação dessa maneira tem uma implicação: assumimos uma ordem, com uma variável antecedendo a outra. Com um modelo estimado, dessa forma, conseguimos predizer quantos votos uma candidatura hipotética teria se soubermos quanto ela gastou.

Para estimar um modelo linear simples usando a função `lm`, basta passar a fórmula e o banco de dados para a função:

```
mod <- lm(pct_votos ~ pct_gastos, data = gov)
mod
```

```
Call:
lm(formula = pct_votos ~ pct_gastos, data = gov)

Coefficients:
(Intercept)   pct_gastos
     -5.832        1.255
```

O *output* da função, salvo no objeto `mod`, pode parecer confuso à primeira vista, mas é fácil interpretá-lo. Em primeiro lugar, temos abaixo de `Call:` uma cópia do código que usamos para estimar nosso modelo, onde podemos ver a fórmula empregada, `pct_votos ~ pct_gastos`. Em segundo lugar, temos na linha seguinte temos, abaixo de `Coefficients:`, os valores estimados dos parâmetros do nosso modelo – que é tudo o que precisamos saber para traçarmos uma reta. Nessa parte, podemos ver que o valor indicado por `(Intercept)` é igual a -5.8317262, sugerindo que o valor predito de votos por um candidato ou candidata que gastou zero reais em campanha é de cerca de -6. De forma similar, o valor estimado do efeito predito de `pct_gastos` sugere que cada ponto percentual gasto em campanha prediz um retorno médio de cerca de 1 votos válidos.

A título de ilustração, podemos comparar as retas dos modelos que trabalhamos até aqui: o nosso modelo arbitrário e o estimado pela função `lm`, em vermelho:

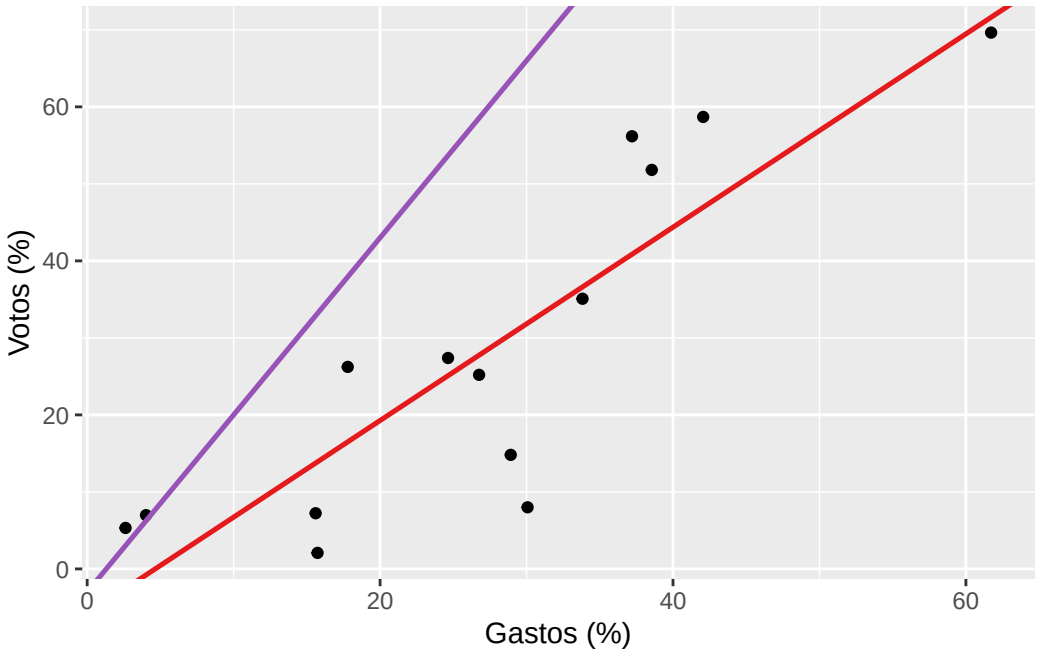


Figura 5.3: Comparação entre modelos

5.2.2 Modelo linear simples

O modelo do exemplo anterior, com apenas duas variáveis – uma dependente e, a outra, independente – normalmente é chamado de *modelo linear simples*. Apesar do nome, ele é extremamente flexível e muito utilizado não só em pesquisas, para resumir relações entre duas variáveis, mas também em diversas outras aplicações. Formalmente, estes podem ser descritos por equações da seguinte forma geral:

$Y_i = \alpha + \beta X_i + \epsilon_i$

onde:

- i indexa cada observação no banco de dados, $i = 1, 2, 3, ...N$ (onde N indica o total de observações no banco);
- Y_i indica nossa variável dependente;
- X_i indica nossa variável independente;
- ϵ_i é um termo de erro (a distância entre o valor predito e o valor real de Y_i);
- α é o intercepto do modelo; e
- β é o coeficiente de inclinação, ou *slope*.

Usando MQO, nosso objetivo é estimar valores para os parâmetros α e β para encontrar uma reta que melhor se ajuste aos dados. O critério, como vimos, é o de minimizar a distância ao

quadrado entre valores preditos e reais de Y_i. Com isso, achamos estimativas dos parâmetros do modelo, $\hat{\alpha}$ e $\hat{\beta}$.[4] A seguir, veremos alguns desses elementos de um modelo em maior detalhe.

5.2.2.1 Coeficientes

A primeira informação relevante de um modelo linear, e normalmente a mais usada em pesquisas, são as estimativas de α e β, também chamados de *coeficientes*. Com elas, identificamos se a relação entre variável independente (e.g., gasto de campanha) e a variável dependente (e.g., votos) é positiva ou negativa e, além disso, sua magnitude. Como salvamos os resultados do nosso modelo no objeto `mod`, coeficientes podem ser acessados diretamente (o objeto `mod` é uma espécie de lista, com vários elementos, e `coefficients` é um deles):

```
mod$coefficients
```

```
(Intercept)  pct_gastos
  -5.831726    1.255031
```

O que esses números significam? Novamente, que o aumento de uma unidade de `pct_gastos` prediz um acréscimo médio de 1.26 unidades de `pct_votos`. Por sua vez, o intercepto indica que o valor predito de `pct_votos` quando `pct_gastos` é zero é de -5.83 – o que não faz sentido, neste caso, dado que não é possível ter um percentual negativo de votos. Isso é algo a se ter em mente: interceptos podem ser interpretados como valores preditos de Y_i quando X_i é zero, mas nem sempre indicam valores reais.

5.2.2.2 Inferência

Nossas estimativas salvas em `mod` contêm informações adicionais: com elas, podemos fazer inferência, isto é, testar hipóteses sobre a relação entre variáveis. Por exemplo, podemos testar se o efeito predito de `pct_gastos` é estatisticamente diferente de zero. Podemos obter um resumo dessas e outras informações usando a função `summary` da seguinte maneira:

```
summary(mod)
```

```
Call:
lm(formula = pct_votos ~ pct_gastos, data = gov)

Residuals:
```

[4] A notação com ^ é reservada, neste contexto, para falar de estimativas já feitas.

```
     Min       1Q   Median       3Q      Max
-23.9070  -5.5258   0.3822   8.9111  15.3246

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   -5.832      6.517  -0.895    0.388
pct_gastos     1.255      0.210   5.977 6.44e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 11.88 on 12 degrees of freedom
Multiple R-squared:  0.7486,    Adjusted R-squared:  0.7276
F-statistic: 35.73 on 1 and 12 DF,  p-value: 6.436e-05
```

A primeira informação que nos interessa, agora, é `Std. Error`, que indica o erro-padrão de cada coeficiente do nosso modelo. Essa informação pode ser entendida como a variação de nossa estimativa – e quão maior ele for em relação à escala dos nossos coeficientes, menos precisa é nossa estimativa. Por essa razão, estimativas de incerteza são reportadas em pesquisas acadêmicas para que outras pessoas possam avaliar nossas inferências. Nas colunas à direita, temos também os T-values e os p-values de cada coeficiente: o `t value` é calculado dividindo o valor do coeficiente pelo seu erro-padrão, e o `p-value` é a probabilidade de obtermos um T-valor igual ou mais extremo do que o observado se a hipótese nula for verdadeira.[5] No nosso caso, o p-value de `pct_gastos` é 6.4360651×10^{-5}, o que indica que a probabilidade de obtermos um valor igual ou mais extremo do que o observado se a hipótese nula for verdadeira é irrisória. Como esse valor é muito menor que 0.05, normalmente usado como limiar para rejeitar a hipótese nula, não podemos rejeitar a hipótese de que o efeito de `pct_gastos` é igual a zero. Dizendo de outra forma, dificilmente veríamos a relação entre `pct_gastos` e `pct_votos` por fruto de mero acaso.

Com essas informações do nosso modelo estimado, podemos também calcular intervalos de confiança para nossas estimativas. Obtemos estes intervalos facilmente usando a função `confint`:[6]

```
confint(mod, , level = 0.95)
```

```
                2.5 %   97.5 %
(Intercept) -20.0301902 8.366738
pct_gastos    0.7975635 1.712498
```

[5] Para mais detalhes sobre o cálculo de T-values e p-values, ver Kellstedt e Whitten (2018).

[6] Como não é nosso objetivo cobrir em profundidade o estimador por MQO, não nos deteremos sobre como são calculadas algumas estatísticas que implementaremos no R daqui para frente. Para mais detalhes sobre modelos lineares, ver Wooldridge (2010), por exemplo.

No exemplo acima, `level = 0.95` estipula que o intervalo de confiança calculado é de 95%, isto é, com probabilidade de 95% de conter o valor real dos nossos parâmetros. Uma vez mais, os resultados desse exercício reforça o que já sabíamos: `pct_gastos` possui relação com `pct_votos`, uma vez que seu efeito predito varia de um mínimo de 0.7975635 a um máximo de 1.712498. Desse modo, ainda que possa ser maior ou menor, ele certamente é positivo.

5.2.2.3 Predições

Tendo estimado os coeficientes de um modelo, podemos usá-los para fazer predições. Vamos voltar ao nosso modelo estimado, salvo em `mod`. Para extrair apenas as estimativas dos coeficientes dele, vamos usar agora a função `coef`, que é uma outra maneira simples de fazer isso.

```
coef(mod)
```

```
(Intercept)  pct_gastos
  -5.831726    1.255031
```

Com essas estimativas, podemos fazer predições para qualquer observação. Aqui, o método consiste no seguinte: precisamos "plugar" as estimativas com valores de `x`, `pct_gastos`. Em primeiro lugar, sabemos que quando `pct_gastos` é zero o valor predito de `pct_votos` para qualquer observação é igual a -5.8317262, que é o valor do intercepto. Segundo, sabemos que uma unidade a mais de `pct_gastos` prediz um aumento de 1.2550307 na variável `pct_Votos`. Dessa forma, plugar esses resultados é uma questão de somar o valor do intercepto e multiplicar o valor de $\hat{\beta}$ pelo valor `pct_gastos` que quisermos predizer. Um exemplo: para predizer quantos votos válidos uma candidatura que gastou 20% de todas as receitas de campanha no seu estado teria, basta fazer a conta:

```
-5.831726 + 1.255031 * 20
```

```
[1] 19.26889
```

O R tem uma função que faz isso automaticamente, `predict`, que pode ser usada da seguinte forma:

```
dados_ficticios <- data.frame(pct_gastos = 20)
predict(mod, newdata = dados_ficticios)
```

```
       1
19.26889
```

Aqui, o essencial é que criamos um novo banco de dados, salvo em `dados_ficticios`, com apenas uma observação, `pct_gastos = 20` – note que esse banco precisa ter o nome das variáveis que usamos para estimar o modelo, `pct_gastos`. Com essa base fictícia, passamos o modelo estimado para a função `predict` e, em seguida, o banco criado para o argumento `newdata`. O resultado é o mesmo que obtivemos acima, mas a função `predict` é útil para fazer predições para várias observações de uma só vez. Veja:

```
dados_ficticios2 <- data.frame(pct_gastos = seq(20, 80, 10))
predict(mod, newdata = dados_ficticios2)
```

```
       1        2        3        4        5        6        7
19.26889 31.81920 44.36950 56.91981 69.47012 82.02043 94.57073
```

Como é possível ver, com apenas uma chamada da função conseguimos usar nosso modelo para predizer quantos votos válidos candidaturas com diferentes percentuais de gastos de campanha teriam.

5.2.2.4 Ajuste do modelo

Alguns modelos são melhores que outros ao explicar a variação de nossa variável dependente, o que afeta diretamente nossa capacidade de fazer predições. Para avaliarmos o desempenho de um modelo isso, temos algumas alternativas. Em primeiro lugar, o objeto do resultado de `summary` contém uma estatística útil para fazermos isso: o R-quadrado.

```
resultados <- summary(mod)
resultados$r.squared
```

```
[1] 0.7485839
```

O código anterior retorna uma métrica que varia de 0 a 1, onde 1 indica que nossas variáveis independentes explicam perfeitamente a variação de Y.[7] No nosso exemplo, o R-quadrado retornou um número de cerca de 0.7, o que indica que nosso modelo explica boa parcela da variação de `pct_votos`. Embora não seja uma métrica definitiva, nos dá uma boa ideia geral de desempenho.

Além dessa métrica, também podemos usar o resultado do teste F do nosso modelo para avaliá-lo. O que ele faz? Em síntese, ele teste a hipótese nula de que nosso modelo completo, com

[7] Particularmente, o R-quadrado é uma razão entre a soma dos quadrados dos resíduos do modelo (a distância entre os valores preditos e reais de Y) e a soma total dos quadrados (a distância entre os valores reais de Y e a média de Y).Na prática, isso nos dá uma ideia de quanto um modelo linear é melhor do que uma simples média para predizer valores de Y.

variáveis independentes, explica a mesma variância do que um modelo modelo sem variáveis (apenas com o intercepto, α), que seria o mesmo que usar a média de `y` para predizer seus valores. O P-valor desse teste pode ser acessado na última linha do resultado de `summary`:

```
summary(mod)
```

```
Call:
lm(formula = pct_votos ~ pct_gastos, data = gov)

Residuals:
     Min       1Q   Median       3Q      Max
-23.9070  -5.5258   0.3822   8.9111  15.3246

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   -5.832      6.517  -0.895    0.388
pct_gastos     1.255      0.210   5.977 6.44e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 11.88 on 12 degrees of freedom
Multiple R-squared:  0.7486,    Adjusted R-squared:  0.7276
F-statistic: 35.73 on 1 and 12 DF,  p-value: 6.436e-05
```

No caso, como ele é menor que 0.05, rejeitamos a hipótese nula: nosso modelo com `pct_gastos` é melhor que um modelo sem essa informação.

5.2.3 Reportando resultados de modelos

Uma vez estimado um modelo, o próximo passo é reportar seus resultados em um artigo ou documento. Para isso, podemos usar a função `modelsummary`, do pacote de mesmo nome, que já vimos nas seções anteriores. Para usá-lo, basta passar os modelos que queremos reportar para a função:

```
modelsummary(mod)
```

Por padrão, `modelsummary()` reporta modelos como colunas e uma tabela nas quais as linhas indicam os coeficientes, como o do intercepto e da variável independente `pct_gastos`, no caso do nosso exemplo. Além disso, a tabela exportada também indica o erro-padrão de

	(1)
(Intercept)	−5.832
	(6.517)
pct_gastos	1.255
	(0.210)
Num.Obs.	14
R2	0.749
R2 Adj.	0.728
AIC	112.9
BIC	114.8
Log.Lik.	−53.429
RMSE	10.99

	(1)
(Intercept)	−5.832
	(6.517)
pct_gastos	1.255
	(0.210)
Num.Obs.	14
R2	0.749

cada coeficiente em parenteses, logo abaixo de cada estimativa. Finalmente, para modelos lineares simples por mínimos quadrados, como o nosso, a tabela uma série de estatísticas, como o R-quadrado, o R-quadrado ajustado (uma medida que penaliza modelos com muitas variáveis independentes), a estatística F e algumas outras medidas de ajustes que podem ser úteis para determinados casos. Para deixar a tabela menor, mantendo apenas informações mais utilizadas, `modelsummary()` reserva o argumento `got_map`, que pode ser usado para indicar quais estatísticas queremos reportar. Para manter apenas o R-quadrado e o número de observações utilizadas na estimação do modelo, podemos usar o seguinte:

```
modelsummary(mod, gof_map = c("nobs", "r.squared"))
```

Para facilitar a leitura da tabela, `modelsummary()` também permite que renomeemos os coeficientes e estatísticas de um modelo. Para isso, basta passar um vetor com os nomes que queremos usar no lugar dos originais para o argumento `coef_map`:

	(1)
Intercepto	−5.832
	(6.517)
Gastos (%)	1.255
	(0.210)
Num.Obs.	14
R2	0.749

```
nomes <- c("(Intercept)" = "Intercepto",
           "pct_gastos" = "Gastos (%)",
           "Num.Obs." = "Observações")
modelsummary(mod, coef_map = nomes, gof_map = c("nobs", "r.squared"))
```

Assim como fizemos com a função `datasummary()`, exportar o resultado de `modelsummary()` pode ser feito com o argumento `output`:

```
modelsummary(mod, coef_map = nomes, gof_map = c("nobs", "r.squared"), output = "tabela.txt")
modelsummary(mod, coef_map = nomes, gof_map = c("nobs", "r.squared"), output = "tabela.rtf")
modelsummary(mod, coef_map = nomes, gof_map = c("nobs", "r.squared"), output = "tabela.html")
```

5.3 Resumo do capítulo

Este capítulo final introduziu alguns conceitos básicos para análise de dados quantitativos. Começamos com uma breve introdução sobre o que são estatísticas descritivas e vimos como calcular algumas das mais comuns usando tanto funções do `tidyverse` quanto com `modelsummary`. Em seguida, introduzimos modelos de regressão linear simples e, usando R, mostramos como obter e interpretar alguns de seus resultados. Por fim, vimos como reportar resultados de modelos de regressão linear simples usando a função `modelsummary`.

Exercícios

> Arquivos necessários
>
> Para realizar estes exercícios, será necessário baixar os arquivos que estão na pasta de materiais complementares deste livro e salvá-los na pasta de trabalho do R.

1. Medidas de tendência central

Na pasta de materiais complementares deste livro, você encontrará um arquivo chamado `distribuicao_renda_rfb.txt`.[8] Esta contém informações sobre a distribuição da renda, apurada pela Receita Federal, de contribuintes de todo o país, entre 2006 e 2020, a partir de suas declarações de renda; em particular, os dados reportam informações de renda agregadas por centil da distribuição de renda – isto é, informações de renda do 1% mais rico, do 2% mais rico, e assim por diante.

Carregue o arquivo em R e o salve no objeto `renda`. Em seguida, use operações de manipulação de dados para calcular o total (soma), a média e a mediana da renda tributável reportada (variável `rendimentos_tributaveis_milhoes`) para cada centil de renda. O resultado ser uma base nova com uma linha apenas por centil. A variável que indica os centis de renda chama-se `centil`.

2. Dispersão

Usando a mesma base de dados agrupada por ano (variável `ano`), calcule o desvio-padrão, os valores máximo e mínimo das variáveis `bens_milhoes` e `rendimentos_isentos_milhoes`.

3. Comparando grupos

Ainda com a base `renda`, crie uma nova base que contenha apenas os centis de renda de 1 a 50 e o 100 (i.e., exclua os centis de renda de 51 a 99). Feito isso, use a função `if_else` e adicione nesta base uma variável que indique "Rico", quanto o centil for igual a 100; e "Pobre", para os centis de renda de 1 a 50. Em seguida, calcule a média dos `rendimentos_tributaveis_milhoes`, `bens_milhoes` e `rendimentos_isentos_milhoes` para cada um destes dois grupos. Comente sobre as diferenças de rendimentos observadas entre os grupos.

[8] Dados disponíveis no Portal de Dados Abertos do Governo Federal.

4. Tabelas formatadas

Crie um novo `data.frame` que mantenha apenas os anos de 2015 a 2020 da base `renda` e apenas o centil 100 (i.e., o centil do 1% mais rico). Em seguida, use o pacote `gt` para criar uma tabela que reporte os valores de `bens_milhoes` e `rendimentos_isentos_milhoes` (colunas) para cada ano (linhas). Personalize a tabela adicionando um título e uma nota de rodapé indicando a fonte dos dados.

5. Tabelas de estatísticas descritivas

Usando a base `renda`, crie uma tabela de estatísticas descritivas para as variáveis `rendimentos_tributaveis_milhoes`, `bens_milhoes` e `rendimentos_isentos_milhoes` utilizando o pacote `modelsummary` para formatar a tabela. Inclua na tabela a média, mediana, desvio-padrão, mínimo e máximo. Adicione um título à tabela e a exporte para um arquivo HTML.

6. Modelo de regressão linear simples

Estime um modelo de regressão linear simples que use a variável `rendimentos_isentos_milhoes` para predizer a variável `rendimentos_tributaveis_milhoes`. Salve o modelo estimado em um objeto chamado `modelo`.

7. Exportando resultados do modelo

Use o pacote `modelsummary` para criar uma tabela bem formatada dos resultados do seu modelo de regressão linear estimado anteriormente. Exporte a tabela para um arquivo HTML.

Referências

Aquino, Jakson Alves de. 2014. «R para cientistas sociais». EDITUS-Editora da UESC.

Dowle, Matt, e Arun Srinivasan. 2023. *data.table: Extension of 'data.frame'*. https://CRAN.R-project.org/package=data.table.

Geografia e Estatística (IBGE), Instituto Brasileiro de. 2023. *Censo Demográfico 2022: Quilombolas: Primeiros Resultados do Universo*. Rio de Janeiro: IBGE.

Healy, Kieran. 2018. *Data visualization: a practical introduction*. Princeton University Press.

Kastellec, Jonathan P, e Eduardo L Leoni. 2007. «Using graphs instead of tables in political science». *Perspectives on politics* 5 (4): 755–71.

Kellstedt, Paul M, e Guy D Whitten. 2018. *The fundamentals of political science research*. Cambridge University Press.

Muhleisen, Hannes, Mark Raasveldt, e DuckDB Contributors. 2020. *duckdb: DBI Package for the DuckDB Database Management System*. https://CRAN.R-project.org/package=duckdb.

Pereira, Rafael H. M., e Rogério J. Barbosa. 2023. *censobr: Download Data from Brazil's Population Census* (versão v0.2.0). https://CRAN.R-project.org/package=censobr.

Spector, Phil. 2008. *Data manipulation with R*. Springer Science & Business Media.

Tufte, Edward R. 1983. «The visual display of». *Quantitative Information*, 13.

Wickham, Hadley. 2014. «Tidy data». *Journal of Statistical Software* 59 (10): 1–23.

———. 2016. *ggplot2: Elegant Graphics for Data Analysis*. Springer.

Wickham, Hadley, Mine Çetinkaya-Rundel, e Garrett Grolemund. 2023. *R for data science*. " O'Reilly Media, Inc.".

Wilkinson, Leland. 2012. *The grammar of graphics*. Springer.

Wooldridge, Jeffrey M. 2010. *Econometric analysis of cross section and panel data*. MIT press.