Jakson Alves de Aquino

R para cientistas sociais

Editora da UESC

Jakson Alves de Aquino

R para cientistas sociais

Ilhéus – Bahia

Editora da UESC

2014

Universidade Estadual de Santa Cruz

GOVERNO DO ESTADO DA BAHIA
JAQUES WAGNER – GOVERNADOR

SECRETARIA DE EDUCAÇÃO
OSVALDO BARRETO FILHO – SECRETÁRIO

UNIVERSIDADE ESTADUAL DE SANTA CRUZ
ADÉLIA MARIA CARVALHO DE MELO PINHEIRO – REITORA
EVANDRO SENA FREIRE – VICE-REITOR

DIRETORA DA EDITUS
RITA VIRGINIA ALVES SANTOS ARGOLLO

Depósito legal na Biblioteca Nacional,
conforme Lei no 10.994, de 14 de dezembro de 2004.

CAPA
Alencar Júnior, com imagem de Jenny Rollo (www.freeimages.com)

REVISÃO
Maria Luiza Nora de Andrade

Dados Internacionais de Catalogação na Publicação (CIP)

A657 Aquino, Jakson Alves de
R para cientistas sociais/ Jakson Alves de Aquino.
– Ilhéus, BA: EDITUS, 2014. 157 p.: il.

ISBN: 978-85-7455-369-6

Referências: p. 156-157.
Inclui glossário.
Inclui índice.

1. Programas de computador. 2. Linguagem de programação (Computadores). 3. Ciências sociais – Métodos estatísticos. 4. Cientistas sociais. I. Título.

CDD 005.133

EDITUS - EDITORA DA UESC
Universidade Estadual de Santa Cruz
Rodovia Jorge Amado, km 16 - 45662-900
Ilhéus, Bahia, Brasil
Tel.: (73) 3680-5028
www.uesc.br/editora
editus@uesc.br
EDITORA FILIADA À

Associação Brasileira
das Editoras Universitárias

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

Apresentação

O R é um software livre de estatística que funciona em diversos sistemas operacionais: GNU Linux, Microsoft Windows, Mac OS X e outros. O aprendizado do R é difícil no início devido à necessidade de se adaptar à sua lógica de funcionamento, se acostumar com a estrutura dos seus documentos de ajuda e memorizar alguns comandos básicos. É preciso bastante perseverança e motivação para aprender os comandos básicos, e disposição para ler as páginas de ajuda e os manuais. Entretanto, depois de um certo tempo, ele possibilita que se trabalhe com grande produtividade e, o que é mais importante, eficácia.

Este não é um livro de estatística e, apenas de forma muito introdutória, pode ser considerado um manual de análise de dados. O objetivo deste livro é introduzir o leitor no uso do R, com foco nas principais tarefas que um cientista social precisa executar quando começa a manusear um banco de dados. Quando comecei a escrevê-lo, não encontrei textos que tivessem propósito semelhante em português e nem mesmo em inglês disponíveis livremente na internet. Havia apenas um livro, em francês, disponível na internet, *R pour les sociologues*, de Barnier (2008). Hoje, já existem mais opções, inclusive em português (LANDEIRO, 2011).

Este livro deve ser lido de uma maneira que não é habitual para cientistas sociais. Não basta fazer uma ou duas leituras, refletir e discutir sobre o assunto com os colegas. É preciso fazer uma leitura não apenas reproduzindo os comandos no R, mas imaginando e testando

alterações nos comandos para comparar com os resultados originais. Em seguida, é preciso enfrentar alguma situação real de pesquisa e reler as seções que trazem soluções para os problemas que forem surgindo. Muitas vezes, entretanto, a solução estará no uso criativo dos comandos vistos ao longo do livro e não numa seção específica. Além disso, o leitor deve sempre ler o documento de ajuda de cada função porque a maioria delas possui argumentos adicionais que não utilizaremos neste livro, mas que são úteis em situações diversas. A minha pretensão com este livro não era de escrever um manual completo de uso do R, mas apenas, em meio à multiplicidade de pacotes e funções disponíveis para o R, indicar caminhos a seguir.

Ao longo do texto, faço referência a programas de escritório do LibreOffice, mas as informações, com alguns ajustes, também são válidas para os programas da Microsoft, que, atualmente, possuem uma maior fatia do mercado. Assim, instruções dirigidas a usuários do LibreOffice Writer podem ser seguidas, sem muitas dificuldades, por usuários do Microsoft Word, e as informações referentes ao LibreOffice Calc são geralmente válidas para o Microsoft Excel.

O livro foi escrito e o código foi testado em ambiente Linux, mas o funcionamento do R é praticamente igual em todas as plataformas e, portanto, isso não deverá ser motivo de dificuldade adicional para usuários de outros sistemas operacionais. Os dados necessários para replicar os exemplos apresentados neste livro podem ser encontrados em: `http://www.lepem.ufc.br/jaa/RparaCS.php`

Sou grato aos alunos de Ciências Sociais do Departamento de Ciências que, no segundo semestre de 2009, embarcaram na aventura de fazer uma disciplina cujo livro didático estava começando a ser escrito, em especial à monitora da disciplina, Gabriella Maria Lima Bezerra, que encontrou várias passagens pouco claras ao longo do texto. Sou grato também a Milton (milton ruser) por várias sugestões de melhoria nos gráficos, e a José Cláudio Faria e Enio Jelihovschi por contribuições que permitiram melhoria no estilo da diagramação do texto. Gostaria, finalmente, de parabenizar à Editora da Universidade Estadual de Santa Cruz — Editus — por, mais do que permitir, incentivar a livre distribuição deste trabalho.

CAPÍTULO 2

PRIMEIROS PASSOS

2.1 APRESENTAÇÃO DO R

O R possui uma enorme quantidade de procedimentos estatísticos em milhares de pacotes livremente disponíveis na internet e que podem ser carregados opcionalmente. Softwares comerciais com interface gráfica são usualmente mais fáceis de usar, mas possuem apenas algumas dezenas de funções acessíveis com o mouse, sendo preciso comprar módulos adicionais para executar funções extras.

Com o R, é possível criar e manter disponíveis, na área de trabalho, vários tipos de objetos. Isso permite grande flexibilidade e rapidez, mas tem um preço: todos os objetos ficam carregados na memória e algumas operações realizam a criação automática de vários objetos, tornando mais complexa a tarefa de se trabalhar com bancos de dados muito grandes.

Existem dezenas de interfaces para o R. Algumas exigem a memorização de numerosos comandos; outras oferecem uma interface com vários botões e itens de menu clicáveis. Algumas funcionam apenas no Linux, no Windows ou no OS X; outras funcionam nos três sistemas operacionais. Algumas funcionam bem num dos sistemas operacionais e são problemáticas nos demais. Algumas são fáceis de instalar e não exigem nenhuma configuração especial; outras, para funcionar, precisam ser cuidadosamente configuradas. Na maioria das interfaces, o modo básico de uso consiste em editar o código no editor e enviar os

comandos para o console do R. Caberá ao leitor experimentar várias opções e escolher a que melhor se adequar ao seu estilo de trabalho e às suas necessidades. No momento em que este livro era escrito, algumas opções de editores de texto ou ambiente integrado de desenvolvimento eram: RStudio, Tinn-R, Eclipse com plugin StatET, RKward, GEdit com plugin rgedit, JGR, Vim com o plugin Vim-R-plugin, e Emacs com ESS. Os dois últimos, Vim e Emacs, são de uso mais complexo, sendo preciso dedicar algum tempo à aprendizagem de seus comandos, mas, em compensação, dominados os comandos básicos, a edição de código poderá ser mais produtiva do que se realizada com outros editores.

2.2 INICIANDO E FINALIZANDO O R

No Linux, o R pode ser iniciado num terminal: basta digitar a letra R e pressionar Enter. O R é um programa de linha de comando, ou seja, após abri-lo, precisamos digitar algum texto no console e pressionar a tecla Enter para enviar a linha para o seu interpretador. O símbolo > no início da última linha exibida na tela chama-se *prompt* e indica que o R está pronto para receber comandos.

Todos as funções terminam com `()`. Quando quiser fechar o R, utilize a função `quit()`. O programa lhe perguntará se quer salvar a área de trabalho. Escolha "Sim" se quiser continuar o trabalho da próxima vez que abrir o R na mesma pasta. Os objetos criados serão salvos no arquivo `.RData` e o histórico de todos os comandos digitados no arquivo `.Rhistory`. Se quiser sair sem salvar a área de trabalho e o histórico, escolha `não`. O comando `quit()` possui uma versão mais curta `q()`:

```
q()
```

A maioria das funções pode receber um ou mais argumentos para execução. A função `q()`, por exemplo, pode receber o argumento *save* de modo que pode-se executar:

```
q(save = "no")
```

Com o comando executado desta forma, estamos dizendo ao R que ele deve sair e que o valor do argumento *save* é *no*, ou seja, estamos dizendo que não queremos salvar os objetos da área de trabalho.

Como o leitor certamente já percebeu, neste livro, os comandos a serem digitados no console do R estão coloridos de forma a facilitar a sua interpretação. Os resultados dos comandos impressos no console do R são exibidos aqui precedidos do símbolo #. Observe que ao digitar ou colar comandos longos no console, ocupando mais de uma linha, o R adiciona o símbolo + à primeira coluna da tela para indicar que a linha é a continuação do comando anterior. Por exemplo, ao digitar no console do R

```
c(1, 2, 3, 4, 5, 6,
  7, 8, 9, 10, 11)
```

será exibido na tela

```
> c(1, 2, 3, 4, 5, 6,
+   7, 8, 9, 10, 11)
```

2.3 OBTENDO AJUDA

O manual completo do R e dos pacotes adicionais instalados pode ser acessado com a função `help.start()`. Mas há várias outras formas de se obter ajuda no uso do R. Uma delas é pela chamada à função `args()`, que lista os argumentos recebidos por uma função, como nos exemplos abaixo:

```
args(setwd)
# function (dir)
# NULL

args(log)
# function (x, base = exp(1))
# NULL

args(head)
# function (x, ...)
# NULL
```

O comando `args(setwd)` nos informa que `setwd()` é uma função que recebe um único argumento, *dir*, o diretório no qual o R deve ler e salvar arquivos. O comando `args(log)` informa que a função `log()` recebe dois argumentos, *x*, o valor cujo logaritmo se pretende calcular, e *base*, a base a ser utilizada no cálculo. Cada argumento é separado do outro por uma vírgula. A presença de "..." na lista de argumentos de algumas funções significa que a função poderá receber um número indefinido de argumentos. Muitas vezes, esses argumentos são repassados para outras funções chamadas internamente.

Alguns comandos possuem argumentos opcionais; outros são obrigatórios e, se não forem fornecidos, os comandos não funcionarão. Não há nenhuma diferença formal entre argumentos obrigatórios e opcionais e, portanto, somente tentando executar a função, ou lendo a sua documentação, saberemos se um determinado argumento é obrigatório ou não. Por exemplo, ao tentarmos executar a função `setwd()` sem nenhum argumento, ocorre um erro, ficando claro que o argumento é obrigatório:

```
setwd()
# Error: argumento "dir" ausente, sem padrão
```

Para a função ser executada sem erros, é preciso fornecer, como argumento, o nome de um diretório entre aspas, e esse diretório deverá existir. O primeiro dos dois exemplos abaixo irá falhar porque o diretório `"/lugar/nenhum"` não existe:

```
setwd("/lugar/nenhum")
# Error: não é possível mudar o diretório de trabalho

setwd("/tmp")
```

Em alguns casos, os argumentos possuem valores pré-definidos que serão utilizados se não indicarmos explicitamente algo diferente; em outros, os argumentos não possuem valores pré-definidos. Podemos identificar argumentos com valores pré-definidos pela presença do símbolo =, como na função `log()` que, por padrão, usa a base 2,718

— o resultado de `exp(1)`. Por isso, os três comandos a seguir produzem o mesmo resultado:

```
log(10, base = exp(1))
# [1] 2.303

log(10, base = 2.718282)
# [1] 2.303

log(10)
# [1] 2.303
```

O valor de um argumento pode ser definido de duas formas: pela inclusão explícita do nome do argumento ou por sua posição. Por exemplo, todos os procedimentos abaixo realizam o cálculo do logaritmo de 10 na base 5:

```
log(10, 5)
# [1] 1.431

log(x = 10, base = 5)
# [1] 1.431

log(10, base = 5)
# [1] 1.431

log(base = 5, x = 10)
# [1] 1.431
```

No segundo comando acima, os nomes dos dois argumentos (*x* e *base*) foram fornecidos à função, mas, como no primeiro comando, poderiam ser dispensados porque os argumentos estão na ordem esperada pela função. Pelo mesmo motivo, o argumento *base* era desnecessário no terceiro comando. No quarto comando, entretanto, os nomes dos argumentos foram realmente necessários porque a função espera que o primeiro valor fornecido seja *x*, e o segundo, *base*.

A função `args()` é útil quando já conhecemos a função e precisamos apenas de uma ajuda para lembrar dos seus argumentos. Se precisarmos saber o significado de cada argumento, o tipo de objeto retornado pela função, e maiores detalhes sobre o seu uso, usamos a função `help()` ou, na sua versão mais sucinta, `?`. Exemplos:

```
help(demo)
?quit
```

Quando não lembramos do nome exato de uma função, podemos obter uma lista de todas as funções existentes que tenham um determinado texto como parte de seu nome com a função `apropos()`:

```
apropos("csv")
# [1] "read.csv"   "read.csv2"  "write.csv"  "write.csv2"
```

Se realmente não conhecemos a função de que precisamos, podemos fazer uma busca de texto mais completa, no nome da função e na sua descrição, usando `help.search()` ou sua forma abreviada, `??`:

```
help.search("install")
??network
```

Se isso não for suficiente para localizar o comando de que precisamos, uma opção será fazer uma busca na internet. Isso pode ser feito a partir do próprio R se o computador estiver conectado à internet:

```
RSiteSearch("social network analysis")
```

O arquivo de ajuda de algumas funções inclui uma seção de exemplos. Para que esses exemplos sejam executados automaticamente, utilize a função `example()`, fornecendo como argumento o nome da função cujos exemplos se deseja ver, como abaixo:

```
example("ls")
```

Por fim, alguns pacotes incluem códigos de demonstração, como a própria mensagem de saudação do R informa. Digite `demo()` para obter uma lista dos códigos de demonstração disponíveis. Se, por exemplo, quiser ver demonstrações de gráficos, digite:

```
demo("graphics")
```

Como o leitor já percebeu, uma limitação do sistema de ajuda do R é que todos os termos de busca devem ser digitados em inglês.

A última opção será pedir ajuda em algum fórum ou lista de discussão sobre R. Se resolver fazer isso, procure fornecer algum código ilustrando o seu problema e algumas informações sobre o seu sistema se desconfiar que o problema pode ser específico da sua versão do R ou do sistema operacional. Para tanto, o comando `sessionInfo()` poderá ser útil:[1]

```
sessionInfo()
# R version 3.1.0 (2014-04-10)
# Platform: x86_64-unknown-linux-gnu (64-bit)
#
# locale:
#  [1] LC_CTYPE=pt_BR.UTF-8       LC_NUMERIC=C
#  [3] LC_TIME=pt_BR.UTF-8        LC_COLLATE=pt_BR.UTF-8
#  [5] LC_MONETARY=pt_BR.UTF-8    LC_MESSAGES=pt_BR.UTF-8
#  [7] LC_PAPER=pt_BR.UTF-8       LC_NAME=C
#  [9] LC_ADDRESS=C               LC_TELEPHONE=C
# [11] LC_MEASUREMENT=pt_BR.UTF-8 LC_IDENTIFICATION=C
#
# attached base packages:
# [1] stats     graphics  grDevices utils     datasets  methods
# [7] base
#
# other attached packages:
# [1] knitr_1.5         vimcom.plus_1.0-0 setwidth_1.0-3
# [4] colorout_1.0-3
#
# loaded via a namespace (and not attached):
# [1] evaluate_0.5.5 formatR_0.10   highr_0.3      stringr_0.6.2
# [5] tools_3.1.0
```

2.4 CRIANDO, DESTRUINDO, SALVANDO OBJETOS

Objetos são criados no R por meio do símbolo de atribuição `<-`. Por exemplo, para criar o objeto `x` com valor próximo ao de π, devemos digitar:

[1] Os interessados em assinar a *Lista Brasileira do* R devem acessar https://listas.inf.ufpr.br/cgi-bin/mailman/listinfo/r-br.

```
x <- 3.1415926
```

Observe que o separador de decimais é o ponto, mesmo que o R esteja sendo executado num ambiente em português. Para listar os objetos existentes na área de trabalho do R, usamos o comando `ls()`:

```
ls()
# [1] "n" "x"
```

O comando `print()` imprime o objeto que lhe for passado como parâmetro, mas se simplesmente digitarmos o nome do objeto e pressionarmos Enter, obteremos o mesmo resultado, porque o comportamento padrão do R é chamar a função `print()` quando lhe é passado o nome de um objeto pela linha de comando:

```
print(x)
# [1] 3.142

x
# [1] 3.142
```

A vantagem de usar o comando `print()` explicitamente é a possibilidade de personalizar o resultado, usando, por exemplo, o parâmetro *digits* para imprimir um valor numérico com uma quantidade especificada de dígitos:

```
print(x, digits = 3)
# [1] 3.14
```

Observe que o resultado do comando `print()` acima, chamado implícita ou explicitamente, tem sempre adicionado `[1]` antes do valor de `x`. Isso ocorre porque o R sempre cria vetores e o objeto `x` é um vetor de comprimento 1. O número 1 entre colchetes indica o índice inicial do vetor numérico `x`. Para criar uma sequência de números, podemos usar o operador "`:`", como abaixo:

```
x <- 5:60
x
```

```
#  [1]  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
# [22] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
# [43] 47 48 49 50 51 52 53 54 55 56 57 58 59 60
```

O antigo objeto `x` foi destruído e o novo objeto é um vetor com 60 algarismos. A coluna de números entre colchetes indica qual o índice do primeiro número da linha.

Uma dica importante é a de sempre usar a tecla Tab para completar o nome de objetos e de arquivos existentes no disco. Este procedimento acelera e evita erros na digitação dos comandos. Experimente digitar `let` Tab Enter. O resultado deverá ser:

```
letters
#  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"
# [17] "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

Outra dica importante é usar as setas ↓ e ↑ sempre que precisar corrigir e reenviar comandos já digitados.

Para salvar um objeto, usamos a função `save()`, fornecendo como parâmetros os objetos a serem salvos e o nome do arquivo. Por convenção, objetos do R são salvos com a extensão `RData`. Assim, usaremos o comando abaixo para salvar o nosso objeto `x`, mas note que o R não adiciona a extensão automaticamente. Precisamos digitá-la explicitamente:

```
save(x, file = "x.RData")
```

Para carregar na área de trabalho um objeto salvo anteriormente com o comando `save()`, usamos `load()`:

```
load("x.RData")
```

Como em muitos comandos do R, se não houver nenhum erro na execução, nenhuma mensagem é impressa na tela, mas a função `load()` retorna invisivelmente um vetor de caracteres com os nomes dos objetos presentes na área de trabalho carregada. Para vermos essa

lista, podemos usar a função `print()` para imprimir na tela o resultado da função `load()`:

```
print(load("x.RData"))
# [1] "x"
```

Os objetos serão salvos na pasta de trabalho atual. Para saber em qual pasta o R está atualmente procurando e salvando arquivos, e para saber quais os seus arquivos e pastas que estão nesta pasta, digite:

```
getwd()
dir()
```

`getwd()` é uma forma abreviada de dizer *get working directory* (me diga qual a pasta de trabalho atual). Para configurar uma pasta diferente deve-se usar `setwd()`. As tabelas 2.1 e 2.2 listam três importantes pastas do Linux e de versões mais recentes do Windows, o que deve facilitar o uso de `setwd()` por aqueles pouco familiarizados com a visualização do sistema de arquivos em modo texto. Nas duas tabelas, supõe-se a existência de um usuário chamado `estudante`.

Tabela 2.1: Localização de pastas no Linux

Nome fantasia	**Localização real**
Pasta pessoal	`/home/estudante` ou `~/`
Área de trabalho	`/home/estudante/Desktop`
Pendrive	`/media/RotuloDoPendrive`

Tabela 2.2: Localização de pastas em versão recente do Windows

Nome fantasia	**Localização real**
Pasta pessoal	`C:/Users/estudante/Meus documentos` ou `~/`
Área de trabalho	`C:/Users/estudante/Desktop` ou `~/../Desktop`
Pendrive	`F:/` (ou outra letra)

Tanto no Linux quanto no Windows, a pasta pessoal pode ser abreviada pelo símbolo ~. Para saber qual pasta o R considera como pasta pessoal no seu sistema operacional, digite:

```
setwd("~")
getwd()
```

Embora no Windows seja geralmente usada uma barra invertida, "\", para delimitar pastas e arquivos, no R o uso da barra normal, "/", é recomendado porque com ela é possível usar a tecla Tab para completar nomes de arquivos, e os scripts ficarão mais facilmente portáveis de um sistema operacional para outro. Note, entretanto, que os nomes de arquivos e pastas poderão não ser mais completados após a ocorrência de algum espaço em branco no nome de uma pasta. Por isso, é recomendável que se evite o uso de espaços em branco ao nomear arquivos e pastas a serem usados com o R.

Ao baixar da internet arquivos compactados (como os que têm extensão `.zip`, `.tar.gz`, `tar.bz2`, `.rar` ou outra), não dê um duplo clique sobre eles. Geralmente, a melhor opção é clicar sobre o arquivo com o botão direito do mouse e escolher a opção "Extrair tudo...", "Extrair aqui..." ou outra semelhante, com atenção para o local onde os arquivos serão descompactados.

Para este livro, sugiro que o arquivo com os bancos de dados a serem usados neste texto seja descompactado na pasta pessoal, `"~/"`. Assim, em qualquer sistema operacional, o primeiro comando a ser digitado ao iniciar o R seria:

```
setwd("~/RparaCS-dados")
```

2.5 TIPOS DE VARIÁVEIS

Durante uma análise de dados, precisamos lidar com diferentes tipos de variáveis. A seguir estão listados os tipos mais comuns, com o nome utilizado no R para se referir a eles entre parênteses:

- Numéricas (`numeric`): Números inteiros ou reais, como idade, renda, número de filhos.

- Datas (`Date`): São um tipo especial de variável numérica.
- Categóricas (`factor`): Variáveis qualitativas, ou seja, características dos indivíduos para as quais não é possível atribuir um valor numérico, como sexo, religião, estado civil, opinião sobre algum tema. É possível agrupar os indivíduos em categorias e contar quantos indivíduos pertencem a cada categoria, mas se, por exemplo, um indivíduo afirma ser católico, e outro, protestante, não podemos, com base nessas afirmações, considerar um mais religioso do que o outro.
- Categóricas ordenáveis (`ordered`): Tipo de variável categórica cujas categorias podem ser hierarquizáveis, como grau de escolaridade, alguns tipos de respostas a perguntas de questionário. Se à pergunta "Qual o papel do governo?", as opções de resposta forem "O governo deve mandar em tudo", "O governo deve controlar algumas coisas" e "Não precisamos de governo", poderíamos considerar aqueles que optaram pela primeira opção adeptos de uma ideologia *mais* estatizante do que aqueles que escolheram a terceira opção.
- Texto (`character`): Características puramente individuais que não podem ser utilizadas para categorizar os indivíduos. Geralmente aparecem nos bancos de dados apenas para ajudar em análises qualitativas e não estatísticas. Exemplo: o nome dos candidatos num banco de dados de resultados eleitorais. Em alguns casos, os textos são passíveis de categorização, como as respostas a uma pergunta aberta. Neste caso, seria preciso manualmente recodificar as respostas abertas numa nova variável contendo um número limitado de categorias.
- Booleanas (`logical`): Variáveis cujos valores podem ser `VERDADEIRO` ou `FALSO`; no R, `TRUE` ou `FALSE`.

2.6 OPERADORES MATEMÁTICOS E LÓGICOS

Um software de estatística não poderia deixar de ser capaz de fazer operações matemáticas e lógicas. A Tabela 2.3 apresenta os principais operadores matemáticos e lógicos usados no R. Aprendemos o significado dos operadores lógicos > e < no ensino fundamental, mas alguns

operadores lógicos do R devem ser novidade para muitos leitores. Observe que o símbolo == tem significado diferente do símbolo = utilizado para atribuir valores a argumentos de funções.

Tabela 2.3: Operadores matemáticos e lógicos

Operador	Significado	Operador	Significado
+	soma	>=	MAIOR OU IGUAL A
-	subtração	<=	MENOR OU IGUAL A
/	divisão	&	E
*	multiplicação	|	OU
^	exponenciação	==	IGUAL A
>	MAIOR QUE	!	NÃO
<	MENOR QUE	!=	DIFERENTE DE

O código abaixo apresenta os resultados de algumas operações matemáticas para ilustrar o uso dos operadores. Os operadores * e / têm precedência sobre os operadores + e -, ou seja, normalmente, multiplicações e divisões são realizadas antes de somas e subtrações. Para alterar esse comportamento, é preciso escrever as operações de menor prioridade entre parênteses:

```
9 / 3
# [1] 3

2 * 3
# [1] 6

4 ^ 2
# [1] 16

(3 + 1) * (6 - 1) ^ 2
# [1] 100

3 + 1 * 6 - 1 ^ 2
# [1] 8

(1 + (2 * 3)) * 5
# [1] 35
```

Note que, ao contrário dos cálculos que aprendemos a fazer manualmente no Ensino Fundamental, não se usa colchetes e chaves nas

operações matemáticas. Quando necessário, usa-se vários grupos de parênteses aninhados. Colchetes e chaves têm outros usos na linguagem R. Os colchetes são usados para selecionar índices de vetores e matrizes, tabelas e data.frames e as chaves são usadas para delimitar blocos de programação.

Nas operações lógicas, também utilizamos parênteses para isolar operações ou para tornar o código mais legível. Observe nos exemplos abaixo que o resultado de um teste lógico é sempre uma variável booleana:

```
3 > 2
# [1] TRUE

5 < 2
# [1] FALSE

2 == 2
# [1] TRUE

2 != 2
# [1] FALSE

(6 > 5) & (7 > 8)
# [1] FALSE

(6 > 5) | (7 > 8)
# [1] TRUE
```

Para a extração da raiz quadrada e cálculo do logaritmo natural utilizamos, respectivamente, as funções `sqrt()` e `log()`:

```
sqrt(16) + log(1)
# [1] 4
```

2.7 USANDO UM EDITOR DE TEXTOS

Usualmente, se trabalha no R produzindo scripts, ou seja, arquivos de texto plano contendo uma sequência de comandos. Os scripts do R são convencionalmente salvos com a extensão `.R`, mas algumas

das interfaces gráficas para o R não acrescentam a extensão automaticamente, ficando a cargo do usuário fazê-lo manualmente. A grande vantagem do uso de scripts é a possibilidade de repetir toda a análise dos dados em poucos segundos após a descoberta e correção de algum erro ou esquecimento no procedimento seguido durante a análise. A maneira recomendada de se trabalhar com o R é escrever os comandos não diretamente no console do R, mas sim no Editor, enviando-os para o console com o atalho de teclado próprio para isso.

É recomendável que três janelas, do console do R, do Editor e dos Gráficos, sejam arranjadas de modo a permanecerem visíveis simultaneamente. Isso facilita a visualização dos resultados dos comandos enviados do Editor, o que é fundamental para perceber e corrigir erros no código. Alguns editores, como o RStudio, fazem esse arranjo automaticamente.

Um script deve conter todo o código necessário para atingir o objetivo desejado, ou seja, a sua execução, linha por linha, deverá ocorrer sem erros. Mas nem todos os comandos executados durante uma sessão de uso do R precisam estar presentes no script. Em alguns momentos, é útil escrever comandos diretamente no console, principalmente aqueles utilizados para explorar os dados antes de analisá-los. O console pode ser visto como um local para rascunhos e experimentações, enquanto não se chega à forma definitiva dos comandos que produzirão um relatório de pesquisa. Se quiser ver uma lista dos comandos digitados (para copiar e colar no editor) utilize o comando `history()`. Ao digitar alguns comandos diretamente no console, o script fica mais conciso e fácil de ler. Por exemplo, as funções `ls()`, `summary()`, `str()`, `names()`, `getwd()`, `dir()`, `levels()` e `dput()` comumente são mais úteis no console do que no script. Os significados dessas funções serão esclarecidos ao longo do livro.

Observe que qualquer texto após o caractere # é ignorado pelo R e esta característica é utilizada para inserir comentários no script que ajudarão o pesquisador a entender o próprio código se precisar retornar a ele alguns meses após tê-lo escrito.

Atenção: Dependendo do editor de textos que estiver sendo utilizado e da forma como os scripts são abertos, pode ser recomendável ter a função `setwd()` como primeiro comando do script, configurando

o diretório de trabalho como aquele em que estão os arquivos (bases de dados, mapas) que serão carregados durante a execução do script. Isso facilitará o trabalho ao tornar desnecessária a digitação do caminho completo para a localização dos arquivos contidos no diretório. Exemplo:

```
setwd("/diretorio/onde/estao/os/arquivos")
```

Antes de considerar que seu script está concluído ou de enviá-lo para alguém, é preciso reiniciar o R e testar o script do início ao fim. O correto funcionamento do script pode depender de algum objeto criado por código não incluído ou já eliminado do script, ou depender do carregamento de algum pacote que não está sendo carregado pelo script ou de algum outro erro. Por isso, a forma mais segura de conferir se o script possui erros é:

1. Salvar o script.
2. Fechar o R sem salvar a área de trabalho.
3. Abrir o R novamente, com atenção para que nenhuma área de trabalho anterior tenha sido carregada.
4. Executar todo o código do script, do início ao fim.

2.8 MILHARES DE FUNÇÕES EM MILHARES DE PACOTES

O R é iniciado com pouco mais de 2000 funções e outros objetos na memória. Milhares de outras funções, para as mais diversas tarefas, podem ser adicionadas por meio de pacotes (*packages*) disponíveis livremente na internet.[2] É claro que ninguém precisa de todas as funções do R; neste livro, utilizaremos em torno de 120.

Para instalar um novo pacote, utilize a função `install.packages()` e para carregar na memória um pacote já instalado, `library()`. No Linux, e em versões recentes do Windows, o R deverá ser executado

[2] Acesse o sítio `http://www.r-project.org/` para ver a lista de pacotes disponíveis, todos acompanhados de uma descrição e do manual de referência.

com privilégios de administrador para que os pacotes fiquem disponíveis para todos os usuários do sistema. Se o R não estiver sendo executado com privilégios de administrador, os pacotes instalados ficarão disponíveis apenas para o usuário que os instalou. No exemplo abaixo, o pacote `descr` é instalado e, em seguida, suas funções são disponibilizadas:

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

O comportamento padrão da função `install.packages()` é obter a última versão do pacote na internet e prosseguir com a instalação. Se você quiser instalar um pacote a partir do arquivo `zip` (Windows) ou `tar.gz` (Linux), terá que fornecer o nome completo do arquivo, como no exemplo:

```
install.packages("~/R/nome_do_pacote_1.0.tar.gz")
```

O comando a seguir instala a maioria dos pacotes que serão usados neste livro:

```
install.packages(c("descr", "memisc", "gdata", "igraph", "maptools"))
```

2.9 RCMDR

Na fase inicial de uso do R, o leitor poderá considerar útil o uso de um sistema de menus para a realização de alguns procedimentos estatísticos comuns, no estilo do SPSS. Um pacote do R que fornece um sistema de menus com essas características é o `Rcmdr`. Para instalá-lo, digite no console do R:

```
install.packages("Rcmdr")
```

Uma vez instalado, para usar o `Rcmdr`, digite:

```
library(Rcmdr)
```

CAPÍTULO 3

VETORES, MATRIZES, LISTAS

3.1 VETORES

Criando

Vimos na seção 2.4 que mesmo quando atribuímos um único valor numérico ou textual a um objeto, o R, na verdade, cria um vetor de números ou de textos com comprimento 1. Uma das formas de criar um vetor é pelo uso da função `vector()`. A função recebe como argumentos *mode* (modo) e *length* (comprimento). O primeiro argumento é uma variável do tipo `character` informando o tipo de vetor a ser criado, o qual pode ser, entre outros, `logical`, `numeric` e `character`. O segundo argumento é o comprimento do vetor. Vetores do tipo `character` são inicializados com strings vazias; do tipo `numeric`, com zeros; e `logical`, com `FALSE`s. No código seguinte temos alguns exemplos:

```
vector(mode = "character", length = 5)
# [1] "" "" "" "" ""

vector(mode = "numeric", length = 7)
# [1] 0 0 0 0 0 0 0

vector(mode = "logical", length = 4)
# [1] FALSE FALSE FALSE FALSE
```

Uma função básica para criar vetores com valores pré-determinados é `c()`, abreviatura de *concatenate*:

```
c("Marx", "Weber", "Durkheim")
# [1] "Marx"     "Weber"    "Durkheim"

c(5, 3, 11, 6, 1, 4)
# [1]  5  3 11  6  1  4

c(TRUE, FALSE, TRUE, TRUE, FALSE)
# [1]  TRUE FALSE  TRUE  TRUE FALSE
```

Nos exemplos acima, os objetos foram criados mas não foram guardados; foram impressos diretamente no console. Para guardá-los, como vimos na seção 2.4, devemos utilizar o símbolo de atribuição `<-`:

```
lgc <- c(TRUE, FALSE, TRUE, TRUE, FALSE, FALSE, TRUE, TRUE)
lgc
# [1]  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE
```

Em alguns casos, podemos querer registrar que não temos informação sobre um dos elementos de um vetor. Por exemplo, imagine ter feito para 10 pessoas uma pergunta cujas opções de resposta eram "Sim" e "Não" e que o 3º e o 8º entrevistados não responderam à pergunta. Neste caso, atribuiríamos o valor `NA` a esses elementos do vetor:

```
txt <- c("Sim", "Não", NA, "Sim", "Sim", NA, "Não", NA, "Sim", "Sim")
txt
#  [1] "Sim" "Não" NA    "Sim" "Sim" NA    "Não" NA    "Sim" "Sim"
```

Se o vetor a ser criado for uma sequência de números inteiros, podemos usar o operador `:` (dois pontos) como nos exemplos abaixo:

```
1:4
# [1] 1 2 3 4

3:9
# [1] 3 4 5 6 7 8 9

10:1
#  [1] 10  9  8  7  6  5  4  3  2  1
```

Para outros tipos de sequências uniformes, existe a função `seq()`, a qual pode ser invocada de diversas formas, sendo uma delas com os argumentos *from*, *to* e *by*, (de, para, tamanho dos passos) como nos exemplos abaixo:

```
seq(2, 10, 2)
# [1]  2  4  6  8 10

seq(1, 10, 2)
# [1] 1 3 5 7 9

seq(3, 5, 0.5)
# [1] 3.0 3.5 4.0 4.5 5.0
```

Outro comando útil para criar vetores é `rep()`, que pode ser invocada com dois argumentos: o valor a ser repetido e o número de repetições:

```
rep("texto", 5)
# [1] "texto" "texto" "texto" "texto" "texto"

rep(3, 7)
# [1] 3 3 3 3 3 3 3

rep(c(1, 2, 3), 4)
#  [1] 1 2 3 1 2 3 1 2 3 1 2 3
```

Como o leitor percebeu pelos exemplos anteriores, números são impressos na tela sem aspas, enquanto textos são limitados por aspas. Como todos os elementos de um vetor obrigatoriamente pertencem a uma mesma classe, se misturarmos números e texto no mesmo vetor, os números serão convertidos em texto e impressos entre aspas. No exemplo abaixo, os números 1 e 7 são convertidos nos textos `"1"` e `"7"`.

```
c(1, "Texto", 7, "Outro texto")
# [1] "1"           "Texto"       "7"           "Outro texto"
```

Além dos vetores dos tipos `character`, `numeric` e `logical`, são muito importantes, nas ciências sociais, os vetores que representam variáveis categóricas. Esses, do tipo `factor`, armazenam os dados em

forma de números inteiros, mas possuem rótulos — vetores do tipo `character` — indicadores do significado dos valores armazenados. Existem duas formas diferentes de criar um vetor de classe `factor`. A forma mais simples é converter um vetor do tipo `character` em `factor`:

```
rspt <- as.factor(txt)
rspt
#  [1] Sim  Não  <NA> Sim  Sim  <NA> Não  <NA> Sim  Sim
# Levels: Não Sim
```

Ao imprimir um vetor do tipo factor, observe que o R exibe os rótulos dos valores e não os valores numéricos, e os rótulos são impressos sem aspas. Após a impressão dos valores do `factor`, é impressa a lista de rótulos (no caso, `Levels: Não Sim`). Ao criar um `factor` pela conversão de um `character`, os rótulos das categorias serão armazenados em ordem alfabética. Por isso, embora o primeiro valor tenha sido `Sim`, a primeira categoria (*level*) apresentada é `Não`.

Outra forma de criar um vetor de classe `factor` é por meio da atribuição de rótulos a um vetor numérico, usando a função `factor()`, que recebe como argumentos um vetor numérico com os dados, um vetor numérico — *levels* — listando os algarismos utilizados para representar as categorias, e um vetor do tipo `character` — *labels* — com os rótulos das categorias. Exemplo:

```
codigo <- c(23, 22, 23, 23, 22, 22, 23, 22)
uf <- factor(codigo, levels = c(22, 23), labels = c("Piauí", "Ceará"))
uf
# [1] Ceará Piauí Ceará Ceará Piauí Piauí Ceará Piauí
# Levels: Piauí Ceará
```

É interessante observar que, embora os valores numéricos no vetor `codigo` fossem 22 e 23, os valores numéricos armazenados no vetor `uf` serão 1 e 2. De fato, independentemente dos valores numéricos originais, os valores numéricos armazenados num vetor do tipo `factor` são sempre uma sequência de números inteiros iniciadas com o número 1.

Nomeando elementos

Os elementos de um vetor podem receber nomes e, nesse caso, ao imprimir o objeto, o R exibe uma linha de nomes abaixo dos quais estão os valores:

```
idh05 <- c(0.677, 0.742, 0.723, 0.683, 0.718, 0.718, 0.703, 0.738,
           0.742)
names(idh05) <- c("AL", "BA", "CE", "MA", "PB", "PE", "PI", "RN", "SE")
idh05
#    AL    BA    CE    MA    PB    PE    PI    RN    SE
# 0.677 0.742 0.723 0.683 0.718 0.718 0.703 0.738 0.742
```

A função `names()` tanto pode ser usada para atribuir nomes aos elementos de um objeto, como no exemplo acima, como para obter os nomes já atribuídos:

```
names(idh05)
# [1] "AL" "BA" "CE" "MA" "PB" "PE" "PI" "RN" "SE"
```

Obtendo informações

Existem diversas funções úteis para se obter informações sobre vetores e outros tipos de objetos. Para exemplificar o uso dessas funções, vamos utilizar os vetores criados nas seções anteriores.

Para saber o número de elementos de um vetor usamos a função `length()`:

```
length(idh05)
# [1] 9

length(txt)
# [1] 10
```

A função `mode()` informa, genericamente, como o objeto é armazenado na memória do R, e a função `class()` diz qual classe foi atribuída a um objeto. Em alguns casos, `mode` e `class` coincidem; em outros, não. No exemplo abaixo, em que são concatenadas essas informações sobre vários objetos, podemos perceber que objetos do tipo `factor` são armazenados como números:

```
c(class(txt), class(idh05), class(lgc), class(rspt))
# [1] "character" "numeric"   "logical"   "factor"

c(mode(txt), mode(idh05), mode(lgc), mode(rspt))
# [1] "character" "numeric"   "logical"   "numeric"
```

No caso de objetos de classe `factor`, podemos usar a função `levels()` para saber quais são os rótulos das categorias:

```
levels(rspt)
# [1] "Não" "Sim"

levels(uf)
# [1] "Piauí" "Ceará"
```

Uma das funções mais usadas é `summary()` que, de um modo bastante sucinto, apresenta informações sobre um objeto. Trata-se de uma função genérica que possui um método diferente para diferentes tipos de objetos. Ou seja, o sumário apresentado tem uma formatação que depende da classe do objeto, como pode ser visto nos exemplos abaixo:

```
summary(txt)
#    Length     Class      Mode
#        10 character character

summary(lgc)
#    Mode   FALSE    TRUE    NA's
# logical       3       5       0

summary(idh05)
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#   0.677   0.703   0.718   0.716   0.738   0.742

summary(rspt)
#  Não  Sim NA's
#    2    5    3
```

Para vetores do tipo `character` não há estatísticas a serem calculadas, e a função `summary()` basicamente informa o comprimento do vetor. Para vetores do tipo `logical`, são informados quantos elementos têm valor verdadeiro, quantos têm valor falso e quantos valores

estão faltando (mesmo que não haja nenhum). Para vetores numéricos são impressas as seguintes estatísticas: valor mínimo, primeiro quartil, mediana, média, terceiro quartil e valor máximo. Para vetores do tipo `factor`, são impressas as quantidades de cada categoria. No caso de vetores dos tipos `numeric` e `factor`, o número de valores faltantes (*missing values*) é indicado por `NA's` (mas somente quando existem valores faltantes).

No caso dos vetores numéricos, as diferentes estatísticas exibidas pela função `summary()` também poderiam ser obtidas separadamente por funções mais específicas. No código abaixo, podemos ver, respectivamente, a média, a mediana, o valor mínimo, o valor máximo e os quartis de um vetor numérico:

```
mean(idh05)
# [1] 0.716

median(idh05)
# [1] 0.718

min(idh05)
# [1] 0.677

max(idh05)
# [1] 0.742

quantile(idh05)
#    0%   25%   50%   75%  100%
# 0.677 0.703 0.718 0.738 0.742
```

Outra estatística bastante útil é a soma. Exemplo:

```
# Estados do Nordeste com IDH acima de 0.72 em 2005:
sum(idh05 > 0.72)
# [1] 4
```

Outra função que apresenta informações sobre um objeto de forma bastante sucinta é `str()`. Esta função apresenta a estrutura do vetor em apenas uma ou duas linhas. Para maior concisão, algumas informações são abreviadas (como a classe do objeto).

```
str(txt)
#  chr [1:10] "Sim" "Não" NA "Sim" "Sim" NA "Não" ...

str(lgc)
#  logi [1:8] TRUE FALSE TRUE TRUE FALSE FALSE ...

str(idh05)
# Named num [1:9] 0.677 0.742 0.723 0.683 0.718 0.718 0.703 0.738 0.742
#  - attr(*, "names")= chr [1:9] "AL" "BA" "CE" "MA" ...

str(rspt)
#  Factor w/ 2 levels "Não","Sim": 2 1 NA 2 2 NA 1 NA 2 2
```

Por fim, uma função frequentemente útil é `dput()`. A função produz como saída o código necessário para gerar o objeto que recebeu como argumento. Exemplos:

```
dput(txt)
# c("Sim", "Não", NA, "Sim", "Sim", NA, "Não", NA, "Sim", "Sim"
# )

dput(uf)
# structure(c(2L, 1L, 2L, 2L, 1L, 1L, 2L, 1L), .Label = c("Piauí",
# "Ceará"), class = "factor")
```

Observe que a função `dput()` usou números seguidos da letra "L" para representar os dados armazenados no `factor` `uf`. A letra "L" após um número indica que ele é um número inteiro e não um número real. Vetores do tipo `factor` armazenam os dados na forma de números inteiros porque eles ocupam menos espaço na memória do que números reais.

Como vimos na seção 2.7, alguns comandos são usados para conhecer melhor os dados e não precisam ser incluídos no script que está sendo produzido com o objetivo de realizar uma análise dos dados. Esses comandos podem ser digitados diretamente no console para manter o script mais conciso e de fácil leitura. A maior parte das funções vistas nesta seção estão justamente entre as que mais comumente devem ser digitadas diretamente no console, e não registradas nos scripts.

Convertendo

O R possui diversas funções para converter vetores de um tipo em outro. Na página 23, vimos como usar a função `as.factor()` para converter um vetor do tipo `character` em `factor`. Convencionalmente, as funções que fazem conversão de um tipo para outro têm o prefixo `as` seguido de um ponto e do nome da classe de destino.

Qualquer tipo de vetor pode ser convertido num vetor do tipo `character`:

```
as.character(c(TRUE, FALSE, TRUE, FALSE))
# [1] "TRUE"  "FALSE" "TRUE"  "FALSE"

as.character(c(1, 3.4, -5.6, 9))
# [1] "1"    "3.4"  "-5.6" "9"
```

Podem ser convertidos em `numeric` vetores lógicos e representações textuais de números. A tentativa de converter um texto com as palavras `Sim` e `Não` resulta num vetor de `NA`s:

```
as.numeric(c(TRUE, FALSE, TRUE, FALSE))
# [1] 1 0 1 0

as.numeric(c("1.3", "1.4", "1.7"))
# [1] 1.3 1.4 1.7

as.numeric(txt)


# Warning: NAs introduzidos por coerção


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

Merece atenção especial a conversão de vetores de classe `factor` em `character` ou `numeric`, pois, ao serem convertidos em `numeric`, são aproveitados os valores numéricos armazenados na memória, mas, ao serem convertidos em `character`, os valores numéricos são substituídos por seus rótulos:

```
as.numeric(rspt)
#  [1]  2  1 NA  2  2 NA  1 NA  2  2
```

```
as.character(rspt)
#  [1] "Sim" "Não" NA    "Sim" "Sim" NA    "Não" NA    "Sim" "Sim"
```

Vetores numéricos podem ser convertidos em vetores lógicos, sendo que o valor zero se torna `FALSE` e qualquer outro valor se torna `VERDADEIRO`. A tentativa de converter `character` em `logical` funciona apenas para os textos `"FALSE"` e `"TRUE"`. Para qualquer outro texto, o resultado é a produção de `NA`s:

```
as.logical(c(0, 1, 1, 0, 1, 1))
# [1] FALSE  TRUE  TRUE FALSE  TRUE  TRUE

as.logical(c(-1.2, -3.3, 0.5, 0.8, 1.3, 2.4))
# [1] TRUE TRUE TRUE TRUE TRUE TRUE

as.logical(c("0", "FALSE", "TRUE", "3.4"))
# [1]    NA FALSE  TRUE    NA
```

Índices: obtendo e modificando valores

Podemos acessar elementos de um vetor por meio de índices entre colchetes, como nos exemplos abaixo:

```
x <- c(4, 8, 2, 6, 7, 1, 8, 1, 2)
y <- c(6, 7, 3, 4, 1, 5, 2, 8, 9)
z <- c("a", "b", "c", "d", "e", "f", "g", "h", "i")
x[3]
# [1] 2

y[2]
# [1] 7
```

Para selecionar mais de um elemento, deve-se colocar entre colchetes um vetor com os índices dos elementos de interesse. O código seguinte seleciona o 1º, o 3º e o 5º elementos do vetor `x`:

```
x[c(1, 3, 5)]
# [1] 4 2 7
```

Observe que é permitido usar funções dentro dos colchetes. No caso, usamos a já conhecida função `c()`, mas qualquer código que produza um vetor de números inteiros ou de valores lógicos pode ser utilizado para produzir o vetor de índices, como nos exemplos seguintes:

```
z[1:4]
# [1] "a" "b" "c" "d"

i <- (y < 5)
y[i]
# [1] 3 4 1 2

x[i]
# [1] 2 6 7 8

x[y < 5]
# [1] 2 6 7 8
```

Como vimos na seção 3.1, uma característica importante dos vetores no R é que seus elementos podem receber nomes por meio da função `names()`. No código seguinte, usamos os textos criados no vetor `z` como nomes de `x`:

```
names(x) <- z
x
# a b c d e f g h i
# 4 8 2 6 7 1 8 1 2
```

Uma vez que os elementos do vetor possuem nomes, podemos usar os nomes no lugar de números inteiros para acessar os valores do vetor. Exemplo:

```
x["c"]
# c
# 2

x[c("b", "d", "g")]
# b d g
# 8 6 8
```

É possível usar índices com valores negativos, o que informa quais elementos não serão selecionados:

```
x[-1]
# b c d e f g h i
# 8 2 6 7 1 8 1 2

x[-c(1, 2, 3)]
# d e f g h i
# 6 7 1 8 1 2
```

Os mesmos índices e nomes usados para extrair valores de um vetor também podem ser usados para modificar esses valores:

```
y[3] <- 11
y
# [1]  6  7 11  4  1  5  2  8  9

x[c("b", "d", "g")] <- 0
x
# a b c d e f g h i
# 4 0 2 0 7 1 0 1 2
```

Operações

Podemos fazer operações aritméticas e lógicas com vetores. Os vetores são emparelhados e o resultado é um novo vetor em que a operação é aplicada a cada par de elementos. Exemplos:

```
x <- c(5, 2, 4, 3, 2)
y <- c(1, 3, 2, 5, 2)
x + y
# [1] 6 5 6 8 4

x - y
# [1]  4 -1  2 -2  0

x > y
# [1]  TRUE FALSE  TRUE FALSE FALSE

x == y
# [1] FALSE FALSE FALSE FALSE  TRUE

x != y
# [1]  TRUE  TRUE  TRUE  TRUE FALSE
```

Se um dos vetores for maior do que o outro, o menor será "reciclado", ou seja, os elementos serão reutilizados quantas vezes isso for necessário para completar a operação:

```
x <- c(1, 2, 3)
y <- c(1, 1, 1, 4, 4, 4)
x + y
# [1] 2 3 4 5 6 7
```

No exemplo acima, o comprimento de y é múltiplo do comprimento de x, e o R não emite nenhum aviso de possível problema porque é muito comum precisarmos fazer operações com vetores de comprimentos diferentes. Por exemplo, ao somar o valor 1 ao vetor y, estamos, na verdade, somando 1 a cada elemento de y, ou seja, estamos reciclando o vetor 1:

```
y + 1
# [1] 2 2 2 5 5 5
```

Entretanto, se o vetor maior não for múltiplo do vetor menor, haverá um aviso, porque, provavelmente, se trata de um erro de programação. Exemplo:

```
x <- c(1, 5, 7)
y <- c(1, 2, 3, 4, 5, 6, 7)
x + y


# Warning: comprimento do objeto maior não é múltiplo do comprimento do
# objeto menor


# [1]  2  7 10  5 10 13  8
```

3.2 MATRIZES

As funções `cbind()` (*juntar colunas*) e `rbind()` (*juntar linhas*) juntam vetores formando matrizes, ou seja, uma forma retangular de representação dos dados em que eles estão distribuídos em linhas e colunas. Cada vetor fornecido como argumento para a função `cbind()`

se torna uma coluna da matriz resultante; com a função `rbind()`, cada vetor se torna uma linha:

```
x <- c(7, 9, 8, 10, 1)
y <- c(9, 8, 10, 9, 3)
z <- c(10, 9, 9, 9, 2)
cbind(x, y, z)
#       x  y  z
# [1,]  7  9 10
# [2,]  9  8  9
# [3,]  8 10  9
# [4,] 10  9  9
# [5,]  1  3  2

rbind(x, y, z)
#   [,1] [,2] [,3] [,4] [,5]
# x    7    9    8   10    1
# y    9    8   10    9    3
# z   10    9    9    9    2
```

Vamos agora salvar a matriz resultante da função `cbind()` no objeto `m`, e usar a função `colnames()` para atribuir nomes às colunas, e a função `rownames()` para atribuir nomes às linhas:

```
m <- cbind(x, y, z)
colnames(m) <- c("Matemática", "Português", "História")
rownames(m) <- c("Helena", "José", "Maria", "Francisco", "Macunaíma")
m
#           Matemática Português História
# Helena             7         9       10
# José               9         8        9
# Maria              8        10        9
# Francisco         10         9        9
# Macunaíma          1         3        2
```

Para acessar ou extrair dados de uma `matrix`, também podemos utilizar índices, como nos vetores. A diferença é que agora são necessários dois vetores de índices, um para as linhas e outro para as colunas, separados por uma vírgula. Assim como fizemos com os vetores, também podemos usar nomes para localizar elementos de uma matriz:

```
m[5, 3]
# [1] 2

m[1:3, 2]
# Helena   José  Maria
#      9      8     10

m[c(1, 4), 1]
#    Helena Francisco
#         7        10

m["Maria", c("Português", "Matemática")]
#  Português Matemática
#         10          8
```

Observe que na primeira linha do código abaixo nenhuma linha da matriz é selecionada e, consequentemente, são exibidos todos os valores da coluna selecionada (`"História"`). Na segunda linha do código, ocorre o contrário, nenhuma coluna é selecionada:

```
m[, "História"]
#    Helena      José     Maria Francisco Macunaíma
#        10         9         9         9         2

m["Macunaíma", ]
# Matemática  Português   História
#          1          3          2
```

Analogamente ao que fizemos com os vetores, podemos usar os índices para mudar valores de elementos específicos de uma matriz:

```
m["Macunaíma", "Português"] <- 4
```

3.3 LISTAS

Podemos agrupar vários objetos de classes diferentes e, no caso de vetores, de comprimentos diferentes, num único objeto do tipo `list`. Várias funções do R retornam objetos do tipo `list`, sendo bastante útil saber como esses objetos podem ser criados e como seus elementos podem ser acessados. No exemplo abaixo, criamos uma `list` contendo três vetores:

```
numeros <- c(1, 2, 3)
lista <- list(numeros, letras = c("a", "b", "c", "d"), c(TRUE, FALSE))
lista
# [[1]]
# [1] 1 2 3
#
# $letras
# [1] "a" "b" "c" "d"
#
# [[3]]
# [1]  TRUE FALSE
```

No código acima, os elementos foram adicionados à `list` de três formas diferentes e, como podemos ver pela impressão da lista, somente o segundo elemento tem um nome, "letras". Os demais elementos da lista estão sem nome, mas podem ser acessados pela sua posição na lista. Para acessar um elemento de uma `list`, devemos digitar o nome da `list` seguido do símbolo `$` e do nome do elemento, ou, entre colchetes (simples ou duplos), o seu nome ou o número da sua posição, como nos exemplos abaixo:

```
lista$letras
# [1] "a" "b" "c" "d"

lista["letras"]
# $letras
# [1] "a" "b" "c" "d"

lista[["letras"]]
# [1] "a" "b" "c" "d"

lista[3]
# [[1]]
# [1]  TRUE FALSE

lista[[3]]
# [1]  TRUE FALSE
```

Quando estamos trabalhando com listas, é retornado um valor diferente, conforme sejam usados o símbolo `$` ou colchetes duplicados ou colchetes simples. Com o uso do `$` ou dos colchetes duplicados, é

retornado o elemento componente da lista. Com o uso de colchetes simples, é retornada uma lista com os elementos selecionados. Confira:

```
c(class(lista["letras"]), class(lista[["letras"]]),
  class(lista$letras))
# [1] "list"      "character" "character"
```

Com o uso de colchetes simples, podemos selecionar, simultaneamente, vários elementos de uma `list`:

```
lista[c(1, 3)]
# [[1]]
# [1] 1 2 3
#
# [[2]]
# [1]  TRUE FALSE
```

Atenção: nunca deixe espaços em branco nos nomes de objetos ou de elementos de `lists` e `data.frames` porque isso complicará o acesso a esses objetos. Exemplo (resultados omitidos):

```
lst <- list(c("aa", "bb", "cc"), c(1, 2, 3, 4), c(1.1, 1.2, 1.3))
names(lst) <- c("Texto", "Números.inteiros", "Números reais")
lst$Texto
lst$Números.inteiros
lst$Números reais
```

CAPÍTULO 4

CARREGAR BANCO DE DADOS EXISTENTE

4.1 INTRODUÇÃO

Bancos de dados para análises quantitativas nas ciências sociais consistem, tipicamente, em dados em formato tabular onde cada coluna representa uma característica variável de muitos casos. Os casos podem ser indivíduos, organizações, países etc, e todas as informações sobre cada caso ficam numa mesma linha. A Figura 4.1 mostra o conteúdo de um pequeno banco de dados fictício registrado num arquivo de texto plano, com apenas cinco casos (pessoas entrevistadas) e três variáveis (sexo, idade e estado.civil):

FIGURA 4.1: Exemplo de banco de dados

```
        sexo  idade  estado.civil
1  Masculino     40        casado
2   Feminino     37        casado
3   Feminino     17      solteiro
4  Masculino     13      solteiro
5   Feminino     10      solteiro
```

No R, os objetos capazes de guardar informações tabulares presentes em bancos de dados, como os da Figura 4.1, são os `data.frames`. Um `data.frame` é constituído pela concatenação de várias colunas numa única `list`. Assim como uma `matrix`, todas as colunas de um

`data.frame` têm que ter o mesmo número de elementos, mas, ao contrário de uma `matrix`, um `data.frame` pode conter colunas de tipos diferentes, com variáveis numéricas, categóricas, booleanas etc.

Uma forma de criar e visualizar no próprio R o banco de dados acima seria:[1]

```
sexo <- c("Masculino", "Feminino", "Feminino", "Masculino", "Feminino")
idade <- c(40, 37, 17, 13, 10)
estado.civil <- c("casado", "casado", "solteiro", "solteiro",
                  "solteiro")
b <- data.frame(sexo, idade, estado.civil)
```

```
b
#        sexo idade estado.civil
# 1 Masculino    40       casado
# 2  Feminino    37       casado
# 3  Feminino    17     solteiro
# 4 Masculino    13     solteiro
# 5  Feminino    10     solteiro
```

Para fazer referência a uma coluna de um `data.frame`, usamos os mesmos procedimentos vistos para acessar e modificar valores de objetos do tipo `list`. Podemos, por exemplo, digitar o nome do banco de dados seguido pelo símbolo `$` e pelo nome da coluna. Para ver um sumário dos dados de uma das colunas:

```
summary(b$estado.civil)
#   casado solteiro
#        2        3
```

Todas as operações válidas para vetores também podem ser aplicadas às colunas de um `data.frame`. Exemplo:

```
b$estado.civil[3] <- "casado"
summary(b$estado.civil)
#   casado solteiro
#        3        2
```

[1] O R não possui funções adequadas para a entrada de uma grande quantidade de dados. Para esse propósito, é melhor utilizar algum outro software, como o LibreOffice Calc com a opção de *Janela / Congelar*, ou criar um formulário para entrada de dados no LibreOffice Base.

Como todos os objetos do R, `data.frames` devem ser salvos em arquivos com extensão `RData`, usando `save()`, e carregados com `load()` e, portanto, não precisam ser especialmente discutidos neste capítulo. Arquivos no formato `dta`, o formato utilizado pelo `STATA`, são muito bem operados pelo `R`. Eles podem ser lidos com a função `read.dta()` e também não serão discutidos aqui. Os procedimentos expostos neste capítulo se aplicam a bancos de dados salvos por outros programas.

Se o leitor tiver necessidade de utilizar um banco de dados preparado por outra pessoa e tiver oportunidade de escolher o tipo de arquivo a ser utilizado, a melhor opção seria um arquivo `RData` com todas as variáveis categóricas já rotuladas. A segunda melhor opção seria um banco de dados no formato `dta`, se todas as variáveis categóricas já estiverem rotuladas. A terceira melhor opção seria um banco de dados no formato `sav`, o formato utilizado pelo `SPSS` porque ele geralmente já está com todas as variáveis categóricas rotuladas e, se tiver sido bem preparado, com poucos problemas a serem corrigidos. Outra opção seria um arquivo `csv` salvo com os rótulos das variáveis categóricas e não com números que ainda teriam que ser transformados em `factors`. Por fim, um arquivo `csv`, cujas variáveis categóricas ainda precisam ser rotuladas (mas, se for fornecido um script do R para carregar o banco de dados e codificar as variáveis, essa última opção se tornará a melhor de todas).

Nos procedimentos das próximas seções, serão trabalhados os arquivos iniciados com o prefixo `bd_` contidos no arquivo `RparaCS-dados.tar.gz` (ou `zip`, se seu sistema operacional for Windows).[2] Eles contêm dados de candidatos ao Senado nas eleições de 2006 salvos em diferentes formatos.[3] Alguns dos arquivos possuem dados que aos serem carregados no R já estarão prontos para serem utilizados em análises estatísticas. Outros, após carregados, ainda precisarão ser manipulados. No conjunto, os diferentes bancos exemplificam as situações mais comuns com as quais os cientistas sociais se deparam

[2] O arquivo está disponível em `http://www.lepem.ufc.br/jaa/RparaCS.php`. Após fazer o download do arquivo, localize-o com o gerenciador de arquivos, clique sobre ele com o botão *direito* do mouse e escolha a opção "Extrair" ou similar.

[3] Os bancos de dados foram preparados a partir de dados oficiais do TSE (`http://www.tse.gov.br/internet/eleicoes/resultado_2006.htm`), complementados com informações do banco de dados *Eleições 2006: Candidatos ao Poder Legislativo no Brasil* (BOLOGNESI; GOUVÊA; MIRÍADE, 2007).

ao ter que utilizar um banco de dados preparado por outra pessoa ou instituição.[4]

O leitor deve ficar atento para a distinção entre processador e editor de texto. O editor permite modificar apenas o texto, propriamente; o processador aplica formatação especial, como negrito, itálico, alinhamento de parágrafos etc... Para editar código do R devemos usar um editor e não um processador de texto.

Antes de iniciar o processo de carregamento, manipulação e análise de um banco de dados, use o editor de textos de sua preferência para criar um script novo onde registrará os comandos necessários para atingir o seu objetivo.

Um primeiro comando a ser executado e registrado é a configuração da pasta onde foram salvos os arquivos como pasta de trabalho: `setwd()` (ver página 12).

4.2 ARQUIVOS SAV

Arquivos `sav`, o formato nativo do `SPSS`, são arquivos binários, o que significa que não podem ser visualizados em editores de texto. As especificações do formato `sav` não são divulgadas pela SPSS Inc., mas os desenvolvedores de software livre, por meio de engenharia reversa, conseguiram elaborar algoritmos capazes de fazer a leitura da maioria dos arquivos salvos nesse formato, o que torna possível seu carregamento no R.

A função para carregar arquivos `sav` é `read.spss()`, do pacote `foreign`. O comportamento padrão de `read.spss()` é criar uma lista de objetos. Para que a lista seja automaticamente convertida num `data.frame`, devemos passar para a função o argumento *to.data.frame* com o valor *TRUE*:[5]

[4] Consulte a seção *R Data Import/Export* do manual do R para saber como lidar com outras situações.

[5] Se tiver dificuldades para encontrar um arquivo, tente usar a função `file.choose()`. Em alguns sistemas, essa função abre uma caixa de diálogo para localização de arquivos.

```
library(foreign)
b <- read.spss("bd_exemplo.sav", to.data.frame = TRUE)


# re-encoding from CP1252
```

O `data.frame` criado incluirá o atributo `variable.labels` com os rótulos das variáveis utilizados no banco `sav` original. O valor desse atributo pode ser acessado com a função `attr()` (para maiores informações sobre a função `attr()`, veja a seção 5.10):

```
attr(b, "variable.labels")
#          uf    candidat     partido       idade        sexo
#        "uf" "candidato"   "partido"     "idade"      "sexo"
#    resultad
# "resultado"
```

O `data.frame` ou `list` criado poderá ter outros atributos, como `Missing`, se o banco de dados no formato `sav` contiver valores *missing* definidos pelo usuário. O atributo informará quais valores haviam sido codificados como *missing* no arquivo `sav`.

Existe um erro frequente na preparação dos bancos de dados no SPSS, que consiste em usar o mesmo rótulo para duas categorias diferentes de uma variável categórica ou codificar uma variável categórica a partir de uma variável que possui valores fracionários, e não apenas números inteiros. Ao carregar um banco de dados com um desses problemas no R, serão produzidos avisos como os seguintes:

```
Mensagens de aviso perdidas:
1: In `levels<-`(`*tmp*`, value = if (nl == nL)
   as.character(labels) else paste0(labels,  :
   não se permite mais níveis duplicados em fatores
```

O banco foi carregado. O R está apenas avisando que é preciso ficar atento ao problema. Por exemplo, no caso de uma variável indicando o município em que foram coletados os dados, poderá haver repetições porque o Brasil possui alguns municípios com o mesmo nome (como Triunfo, em Pernambuco e no Rio Grande do Sul). Um

procedimento para descobrir quais colunas possuem valores ou rótulos de valores duplicados é examinando o atributo `label.table` do `data.frame` criado pela função `read.spss()`, como exemplificado na seção 10.5 (página 127). O atributo `label.table` somente é adicionado ao `data.frame` quando o valor do argumento *to.data.frame* é *FALSE*. A solução para problemas desse tipo pode ser importar o banco de dados sem aproveitar os rótulos das variáveis categóricas,

```
b <- read.spss("bd_exemplo.sav", use.value.labels = FALSE,
               to.data.frame = FALSE)
```

e, em seguida, explorar o atributo `label.table` do `data.frame` criado pela função para descobrir a causa do problema e, manualmente, codificar as variáveis problemáticas de forma correta.

4.3 ARQUIVOS CSV

Arquivos do tipo `csv`, *comma separated values*,[6] são arquivos de texto plano, podendo ser visualizados em editores de texto simples porque não contêm qualquer formatação especial, como negrito, itálico, cores, espaçamento entre linhas etc...[7] Apesar do nome, devido ao fato da vírgula ser um símbolo comum em textos, inclusive nos valores de variáveis categóricas, e, principalmente, por ser a vírgula o separador de decimais em alguns idiomas, como o português, na prática, é comum encontrar arquivos `csv` com os campos separados por ponto e vírgula, espaços e caracteres de tabulação. Para eliminar qualquer dúvida sobre o tipo de valor de um campo, se numérico ou textual, costuma-se escrever os valores que devem ser tratados como texto entre aspas, simples ou duplas.

Para se habituar a utilizar este tipo de arquivo no R, abra os arquivos `bd_exemplo*.csv` um por um, de acordo com os procedimentos apresentados a seguir.

Para carregar um arquivo no formato `csv`, utilizamos a função `read.table()`, mas, para usar a função corretamente, precisamos sa-

[6] Valores separados por vírgulas.

[7] É por isso que uma planilha de cálculo, ao ser salva como `csv`, perde toda informação sobre formatação das células e sobre fórmulas utilizadas.

ber algumas informações, sendo as mais frequentemente necessárias: (1) qual caractere é utilizado para separar os valores; (2) se os valores textuais estão limitados por aspas simples ou aspas duplas ou se não estão limitados por nenhum caractere especial; (3) se os nomes das variáveis estão na primeira linha do arquivo. Para descobrirmos essas informações, utilizaremos a função `file.head()` do pacote `descr` para inspecionar as primeiras linhas de arquivos de texto, como é o caso dos bancos de dados no formato `csv`:

```
library(descr)
file.head("bd_exemplo1.csv")
# "uf" "candidato" "partido" "logvotos" "sexo" "resultado"
# "RO" "ACIR MARCOS GURGACZ" "PDT" 12,257 "Masculino" "Não eleito"
# "ES" "AFONSO CEZAR CORADINE" "PSOL" 9,937 "Masculino" "Não eleito"
# "DF" "AGNELO SANTOS QUEIROZ FILHO" "PC do B" 13,207 "Masculino" "Não
# "SP" "ALDA MARCO ANTONIO" "PMDB" 13,742 "Feminino" "Não eleito"
# "GO" "ALDO SILVA ARANTES" "PC do B" 11,659 "Masculino" "Não eleito"
```

Como podemos ver pelo resultado da função `file.head()`, a primeira linha do arquivo contém os nomes das variáveis (ou seja, o arquivo possui um cabeçalho, ou *header*). Os diferentes valores estão *sep*arados por um espaço em branco, uma vírgula está sendo usada como separador de decimais e os valores textuais estão limitados por aspas (*quotation marks*) duplas. Logo, o comando para abrir este arquivo será:

```
b1 <- read.table("bd_exemplo1.csv", header = TRUE, sep = " ",
                 dec = ",", quote = '"')
```

Após carregar cada banco dos exemplos desta seção, use a função `summary()` para conferir se a operação foi realizada com sucesso. Por exemplo:

```
summary(b1)
#        uf                              candidato      partido
#  SP     : 15   ACIR MARCOS GURGACZ        :  1   PSOL   : 17
#  RJ     : 11   AFONSO CEZAR CORADINE      :  1   PFL    : 15
#  MG     : 10   AGNELO SANTOS QUEIROZ FILHO:  1   PDT    : 14
#  RS     : 10   ALDA MARCO ANTONIO         :  1   PSDB   : 13
#  DF     :  9   ALDO SILVA ARANTES         :  1   PSTU   : 13
```

```
#  MA     :  9   ALFREDO HELIO SIRKIS       :  1   PCB    : 12
#  (Other):138   (Other)                    :196   (Other):118
#     logvotos            sexo          resultado
#  Min.   : 5.79   Feminino : 30   Eleito    : 27
#  1st Qu.: 8.44   Masculino:172   Não eleito:175
#  Median :10.21
#  Mean   :10.52
#  3rd Qu.:12.79
#  Max.   :16.01
#
```

Para carregar os demais arquivos `csv`, basta repetir o mesmo procedimento: inspecionar as primeiras linhas do arquivo com `file.head()` e, então, fornecer os argumentos adequados para `read.table()`. No exemplo seguinte, o separador de campos é uma vírgula; como delimitadoras de texto foram usadas aspas duplas, e o separador de decimais é um ponto, mas esse já é o valor padrão do argumento *dec* e, portanto, não é preciso acrescentar esse argumento à função:

```
file.head("bd_exemplo2.csv")
# "uf","candidato","partido","logvotos","sexo","resultado"
# "RO","ACIR MARCOS GURGACZ","PDT",12.257,"Masculino","Não eleito"
# "ES","AFONSO CEZAR CORADINE","PSOL",9.937,"Masculino","Não eleito"
# "DF","AGNELO SANTOS QUEIROZ FILHO","PC do B",13.207,"Masculino","Não
# "SP","ALDA MARCO ANTONIO","PMDB",13.742,"Feminino","Não eleito"
# "GO","ALDO SILVA ARANTES","PC do B",11.659,"Masculino","Não eleito"

b2 <- read.table("bd_exemplo2.csv", header = TRUE, sep = ",",
                 quote = '"')
```

No próximo exemplo, o separador de campos é um ponto e vírgula; como delimitadoras de texto foram usadas aspas duplas, e o separador de decimais é a vírgula:

```
file.head("bd_exemplo3.csv")
# "uf";"candidato";"partido";"logvotos";"sexo";"resultado"
# "RO";"ACIR MARCOS GURGACZ";"PDT";12,257;"Masculino";"Não eleito"
# "ES";"AFONSO CEZAR CORADINE";"PSOL";9,937;"Masculino";"Não eleito"
# "DF";"AGNELO SANTOS QUEIROZ FILHO";"PC do B";13,207;"Masculino";"Não
# "SP";"ALDA MARCO ANTONIO";"PMDB";13,742;"Feminino";"Não eleito"
# "GO";"ALDO SILVA ARANTES";"PC do B";11,659;"Masculino";"Não eleito"
b3 <- read.table("bd_exemplo3.csv", header = TRUE, sep = ";",
                 dec = ",", quote = '"')
```

No último exemplo, o separador de valores é o *caractere de tabulação*, o qual, por ser um caractere não imprimível, no R, é representado por `\t`, e não há símbolo delimitador de textos:

```
file.head("bd_exemplo4.csv")
# uf\tcandidato\tpartido\tlogvotos\tsexo\tresultado
# RO\tACIR MARCOS GURGACZ\tPDT\t12.257\tMasculino\tNão eleito
# ES\tAFONSO CEZAR CORADINE\tPSOL\t9.937\tMasculino\tNão eleito
# DF\tAGNELO SANTOS QUEIROZ FILHO\tPC do B\t13.207\tMasculino\tNão ele
# SP\tALDA MARCO ANTONIO\tPMDB\t13.742\tFeminino\tNão eleito
# GO\tALDO SILVA ARANTES\tPC do B\t11.659\tMasculino\tNão eleito

b4 <- read.table("bd_exemplo4.csv", header = TRUE, sep = "\t",
                 quote = '')
```

4.4 ARQUIVOS XLS, ODS E MDB

Se os seus dados estiverem numa planilha do Microsoft Excel (`xls`) ou no Open Document Format (`ods`), abra o arquivo no LibreOffice Calc ou no Microsoft Excel e salve a planilha como arquivo to tipo `csv`. Os dados no formato `csv` poderão ser carregados no R de acordo com os procedimentos explicados na seção 4.3. Existem funções para extrair dados de planilhas de cálculo a partir do R sem a necessidade de conversão manual para `csv`. Uma dessas função é `read.xls()` do pacote `gdata`:[8]

```
library(gdata)
bx <- read.xls("bd_exemplo.xls")
```

A função `read.xls()` poderá falhar no Linux se o arquivo `xls` contiver acentos nos nomes das colunas e se a codificação de caracteres não for UTF-8. Nesse caso, pode-se tentar acrescentar o argumento *encoding* com o valor `"latin1"`:

```
bx <- read.xls("bd_exemplo.xls", encoding = "latin1")
```

[8] No Windows, poderá ser necessário instalar *ActivePerl* ou outro interpretador da linguagem `perl`.

Para carregar um banco de dados contido em arquivo do tipo `mdb`, do *Microsoft Access*, é preciso utilizar a função `mdb.get()` do pacote `Hmisc`. A função `mdb.get()` depende de um conjunto de programas reunidos em `mdb-tools`.[9] Na ausência das `mdb-tools`, será preciso usar o *Microsoft Access* ou o *LibreOffice Base* para abrir o arquivo `mdb` e manualmente exportar as tabelas para o formato `csv`.

4.5 ARQUIVO COM COLUNAS DE LARGURA FIXA

Arquivos com colunas de largura fixa (*fixed width files*, em inglês) eram mais comuns há algumas décadas. Hoje, os arquivos desse tipo mais importantes para cientistas sociais brasileiros são, provavelmente, os bancos de dados das PNADs[10], realizadas anualmente pelo `IBGE`. Arquivos com colunas de largura fixa são arquivos de texto plano, como arquivos `csv`, mas sem delimitadores de campo. Para carregá-los, é preciso saber com antecedência quantos caracteres cada variável ocupa no banco de dados. Usando a função `file.head()` com o arquivo `bd_largurafixa.txt` como argumento, temos o seguinte resultado:

```
file.head("bd_largurafixa.txt")
# ROACIR MARCOS GURGACZ                    PDT    4411
# ESAFONSO CEZAR CORADINE                  PSOL   5111
# DFAGNELO SANTOS QUEIROZ FILHO            PC do B4811
# SPALDA MARCO ANTONIO                     PMDB   6221
# GOALDO SILVA ARANTES                     PC do B6811
# RJALFREDO HELIO SIRKIS                   PV     5611
```

Como podemos observar, os dados são os mesmos dos arquivos do tipo `csv` carregados na seção 4.3, mas sem os nomes das colunas na primeira linha. As duas primeiras letras de cada linha indicam a unidade da federação, e a seguir, sem nenhum espaço, o nome do candidato. Temos, então, o nome do partido e uma sequência de números cujo significado não é possível adivinhar. Somente é possível carregar um arquivo desse tipo com um livro de códigos ou dicionário indicando a posição inicial e final de cada variável e os rótulos das variáveis e dos

[9] Em muitas versões do Linux, as `mdb-tools` podem ser instaladas a partir do gerenciador de software do sistema operacional. Em alguns sistemas, deve ser instalado o pacote `mdbtools`.

[10] Abreviatura de *Pesquisa Nacional por Amostragem de Domicílio*

valores das variáveis categóricas. Para o arquivo `bd_largurafixa.txt`, o livro de códigos é reproduzido a seguir:

Posição inicial	Tamanho	Nome da variável	Rótulo da variável	Categorias	
				Valor	Rótulo
1	2	V001	Estado		
3	39	V002	Candidato		
42	7	V003	Partido		
49	2	V004	Idade		
51	1	V005	Sexo	1	Masculino
				2	Feminino
52	1	V006	Resultado	1	Não Eleito
				2	Eleito

O comando para abrir um arquivo deste tipo é `read.fwf()`, tendo *widths* (larguras) como parâmetro obrigatório. No comando abaixo, os valores do vetor *widths* foram copiados da coluna *Tamanho* do livro de códigos e também informamos quais as classes das colunas que serão lidas:

```
bf <- read.fwf("bd_largurafixa.txt", widths = c(2, 39, 7, 2, 1, 1),
               colClasses = c("factor", "character", "factor",
                              rep("numeric", 3)))
```

O comando acima poderá falhar se o arquivo contiver uma codificação de caracteres incompatível com o sistema operacional. O caso mais frequente será a necessidade de carregar no Linux um banco de dados criado com a codificação de caracteres utilizada no Windows. Neste caso, poderemos usar as funções `readLines()`, `toUTF8()` e writeLines para, respectivamente, ler as linhas presentes no arquivo, recodificar os caracteres para UTF-8 e escrever uma nova versão do arquivo (para maiores detalhes, ver a seção 4.6):

```
tmp <- readLines("bd_largurafixa.txt")
tmp <- toUTF8(tmp)
writeLines(tmp, "bd_lfx.utf8.txt")
bf <- read.fwf("bd_lfx.utf8.txt", widths = c(2, 39, 7, 2, 1, 1),
               colClasses = c("factor", "character", "factor",
                              rep("numeric", 3)))
```

O arquivo `bd_largurafixa.txt` não inclui os nomes das variáveis e, o R, automaticamente, atribuirá nomes no formato V_1, V_2, ..., V_n. Para renomear as variáveis de um banco de dados, usamos o comando `names()`:

```
names(bf) <- c("uf", "candidato", "partido", "idade", "sexo",
               "resultado")
```

Se a base de dados tiver muitas colunas, a função `dput()` será útil para produzir um código com os nomes atuais. O código, gerado no console do R, poderá ser copiado para o script que está sendo escrito e, então, editado:

```
dput(names(bf))
# c("uf", "candidato", "partido", "idade", "sexo", "resultado")
```

As variáveis categóricas `sexo` e `resultado` foram carregadas como números e será preciso codificá-las. Veremos na seção 4.6 como resolver este tipo de problema.

4.6 SOLUÇÃO DE PROBLEMAS

Os procedimentos vistos nas seções anteriores são suficientes para carregar na área de trabalho do R a maioria dos bancos de dados com os quais podemos nos deparar durante uma pesquisa, mas vários problemas ainda podem precisar ser resolvidos para se ter os dados prontos para análise. Nas próximas subseções, veremos como solucionar os problemas mais comuns.

Variáveis com letras maiúsculas nos nomes

É perfeitamente válido utilizar letras maiúsculas nos nomes dos objetos do R, mas quase todas as funções têm nomes iniciados com letras minúsculas e o R faz distinção entre maiúsculas e minúsculas, tornando desconfortável e confuso ter constantemente que alternar a caixa durante a digitação. A função `tolower()` converte todas as letras de um vetor de caracteres em minúsculas, e podemos usar o seguinte comando para converter os nomes das variáveis de um banco de dados:

```
names(b) <- tolower(names(b))
```

Espaços excedentes

Alguns bancos de dados trazem variáveis do tipo texto ou os valores de variáveis categóricas com espaços em branco adicionais para completar uma largura pré-definida pelo programa em que o banco de dados foi gerado. Esses espaços podem ser eliminados com a função `trim()`, do pacote `gdata`. É possível fornecer um `data.frame` como argumento para a função, o que implicará na remoção dos espaços desnecessários de todas as variáveis que os contenham. Segue um exemplo simples do uso da função `trim()` (o argumento *recode.factor* `= FALSE` evita que as categorias das variáveis categóricas sejam reordenadas):

```
library(gdata)
trim("   pouco texto, muito espaço            ", recode.factor = FALSE)
# [1] "pouco texto, muito espaço"
```

Compare, por exemplo, o sumário de `bf$partido` antes e depois da função `trim()`. Os nomes dos partidos e a quantidade de candidatos de cada partido estão alinhados à direita, mas, antes do uso da função `trim()`, somente PC do B e PT do B estão corretamente alinhados. Esses são os partidos com siglas mais extensas e todos os outros ficaram com espaços em branco excedentes, tornando-os aparentemente desalinhados em relação às respectivas quantidades:

```
summary(bf$partido)
# PAN     PCB     PC do B PCO     PDT     PFL     PHS     PL
#       2      12       6       9      14      15       2       2
# PMDB    PMN     PP      PPS     PRB     PRONA   PRP     PRTB
#      12       2       2       4       1       5       5       6
# PSB     PSC     PSDB    PSDC    PSL     PSOL    PSTU    PT
#       7       3      13      12       7      17      13      10
# PTB     PTC     PT do B PTN     PV
#       4       4       3       2       8

bf$partido <- trim(bf$partido)
summary(bf$partido)
```

```
#     PAN     PCB PC do B     PCO     PDT     PFL     PHS      PL
#       2      12       6       9      14      15       2       2
#    PMDB     PMN      PP     PPS     PRB   PRONA     PRP    PRTB
#      12       2       2       4       1       5       5       6
#     PSB     PSC    PSDB    PSDC     PSL    PSOL    PSTU      PT
#       7       3      13      12       7      17      13      10
#     PTB     PTC PT do B     PTN      PV
#       4       4       3       2       8
```

Necessidade de codificar variáveis

Como vimos na seção 4.5, o arquivo `bd_largurafixa.txt` não inclui os nomes das variáveis na primeira linha e algumas variáveis categóricas receberam números no lugar dos rótulos. É comum encontrar arquivos com essas características porque alguns programas de estatística não convertem automaticamente variáveis com valores textuais em variáveis categóricas. Nesses programas, é preciso carregar números que, depois, serão convertidos em categorias. Outro motivo para usar números no lugar de texto representando categorias é por economia de espaço em disco, o que é especialmente importante em bancos de dados grandes. Para carregar o `bd_largurafixa.txt`, siga os procedimentos da seção 4.5, inclusive renomeando os nomes das colunas.

Para codificar variáveis numéricas como categóricas, é preciso conhecer a correspondência entre os números e as categorias que representam.[11] Como vimos na seção 3.1, o comando para criar um vetor de variáveis categóricas (chamados de *factors* na linguagem R), é `factor()`, sendo *levels* (níveis) o vetor dos números utilizados para representar as categorias, e *labels* o vetor dos rótulos das categorias representadas. Assim, para codificar as duas variáveis categóricas do banco de dados carregado na seção 4.5 que se apresentam em formato numérico, devemos utilizar os seguintes comandos:

```
bf$sexo <- factor(bf$sexo, levels = c(1, 2),
                  labels = c("Masculino", "Feminino"))
bf$resultado <- factor(bf$resultado, levels = 1:2,
```

[11] O próprio fornecedor dos dados deve disponibilizar uma tabela com a correspondência entre os valores.

```
                        labels = c("Não eleito", "Eleito"))
```

Use o comando `summary()` para conferir se o banco está corretamente codificado.

Codificação de caracteres errada

No Linux, utiliza-se UTF-8 para codificação de caracteres, enquanto no Windows é utilizada a codificação WINDOWS-1252. Com UTF-8, é possível incluir caracteres de qualquer idioma num arquivo de texto plano, mas um texto produzido em texto plano (sem nenhuma formatação especial) em ambiente Linux não será corretamente exibido em ambiente Windows. Por exemplo, a palavra "ação", codificada em UTF-8, seria exibida no Windows como "aÃ§Ã£o", e, codificada em WINDOWS-1252, seria exibida no Linux como "a\xe7\xe3o". Além disso, quando o ambiente for UTF-8, algumas funções do R falharão se a codificação utilizada for outra, sendo, portanto, necessário converter a codificação dos objetos antes de usar essas funções. Isso pode ser feito com as funções `toUTF8()` e `fromUTF8()` (para e de UTF-8, respectivamente), do pacote `descr`. Essas funções convertem vetores ou bancos de dados inteiros de uma codificação para outra. O código abaixo exemplifica como a função `fromUTF8()` pode ser usada para converter um script do R escrito em UTF-8 para o WINDOWS-1252:

```
library(descr)
linhas <- readLines("RparaCS.R")
linhas <- fromUTF8(linhas)
writeLines(linhas, "RparaCS-win.R")
```

As funções `fromUTF8()` e `toUTF8()` podem receber um `data.frame` como argumento. Nesse caso, os nomes de todas as variáveis e os rótulos de todas as variáveis categóricas serão convertidos de uma codificação para outra. Se um banco de dados for originário do Windows, como a maioria dos bancos no formato do `SPSS`, experimente convertê-lo para UTF-8 logo após carregá-lo no Linux, pois até mesmo a função `summary()` pode falhar se a codificação estiver errada.

A função `read.table()` possui o argumento *encoding*, sendo possível definir qual a codificação de caracteres utilizada no arquivo con-

tendo os dados a serem carregados. No Linux, para carregar um arquivo preparado para Windows, deve-se usar *encoding* = `"latin1"`.

Conversão entre numeric, factor e character

Nem sempre as variáveis de um banco de dados estão classificadas como deveriam. No banco de dados que estamos utilizando neste capítulo, a variável *candidato* foi automaticamente classificada como `factor` pela função `read.table()`, o que não faz sentido porque não temos intenção de usar os nomes das pessoas para categorizá-las. Mesmo que houvesse algum nome repetido, seria apenas uma coincidência, sem nenhum valor estatístico para uma pesquisa na área de política. A variável *candidato* deve ser tratada como texto, e deve, portanto, ser convertida em `character`. Como vimos na seção 3.1, os nomes das funções para conversão de objetos de um tipo em outro têm o formato `as.classeDeDestino`. Assim, o comando que precisamos é:

```
b4$candidato <- as.character(b4$candidato)
```

A classe correta para variáveis categóricas que podem ser consideradas ordenáveis é `ordered`. Nem sempre variáveis categóricas aparentemente ordenáveis devem ser classificadas como `ordered`. A variável idade, por exemplo, se convertida em categórica, em geral, não deverá ser convertida em `ordered` porque muitas das características dos indivíduos atingem intensidade máxima — ou mínima — durante a fase adulta da vida da pessoa, e não durante a infância ou velhice. Uma variável muito comum, a *escolaridade*, por outro lado, fica mais corretamente classificada como `ordered` do que simplesmente como `factor`. Para converter uma variável em `ordered`, utilizamos a função `ordered()` ou `as.ordered()`.

Conversão de datas

Alguns bancos de dados possuem datas como variáveis. Essas variáveis podem ser lidas pelo R como se fossem texto para, em seguida, serem convertidas em objetos da classe `Date`. A função para converter de `character` para data é `as.Date()`. No exemplo abaixo, são criados dois pequenos vetores, cada um com apenas duas datas expressas na forma de texto. Observando as datas com atenção, percebemos que em

cada vetor foi seguida uma convenção diferente para representar as datas, sendo diferentes as sequências em que se encontram ano, mês e dia. Além disso, no vetor `ini`, o ano está representado por apenas dois algarismos, enquanto, no vetor `fim`, foram usados 4 algarismos. O vetores são, em seguida, convertidos para `Date`, o que permite a realização de operações algébricas:

```
ini <- c("03/05/96", "17/08/97")
fim <- c("1997-27-01", "1999-14-03")
ini <- as.Date(ini, format = "%d/%m/%y")
fim <- as.Date(fim, format = "%Y-%d-%m")
fim - ini
# Time differences in days
# [1] 269 574
```

O argumento *format* consiste na repetição dos caracteres que não representam a data propriamente e na indicação das unidades de tempo na mesma sequência em que se encontram no objeto de tipo `character`. Cada unidade de tempo é indicada por uma letra precedida pelo símbolo `%`: *d* para dia, *m* para mês, *y* para ano com dois algarismos e *Y* para ano com 4 algarismos. Consulte a ajuda da função `as.Date()` para saber quais letras representam cada uma das unidades de tempo.

Se as datas forem originárias de um banco do tipo `sav`, do `SPSS`, a variável poderá consistir num vetor de números inteiros que informam quantos segundos se passaram desde o dia 14 de outubro de 1582. Uma variável deste tipo poderá ser convertida para o formato utilizado pelo R com o seguinte comando:

```
x <- as.Date(as.POSIXct(x, origin = "1582-10-14"))
```

Carregar bancos de grandes dimensões

Ao tentar carregar um arquivo de texto de grandes dimensões, a memória do computador poderá não ser suficiente. Quando se tratar de um arquivo com colunas de largura fixa, como os das PNADs, pode-se tentar usar a função `read.fwf()` com o valor do argumento *buffersize* = *500*, ou menor para reduzir o número de linhas lidas de uma única vez. Outra opção é converter o arquivo para `csv` usando a

função `fwf2csv()` do pacote `descr` e carregar o banco de dados com a função `fread()` do pacote `data.table`. A vantagem da função `fread()` sobre a função `read.table()` é o uso menor de memória, mas se o banco de dados for maior do que a memória disponível, a solução é usar as funções do pacote `sqldf` para trabalhar com apenas parte das colunas do banco de dados.

Se o arquivo for do tipo `sav` (SPSS) e a memória do computador não for suficiente para o uso da função `read.spss()`, uma possibilidade seria usar o SPSS para converter o arquivo para o formato `dta` (STATA). Se o SPSS não estiver disponível, pode-se usar o script `sav2dat.sh`.[12] Este script usa o `pspp`[13] para criar um arquivo de texto plano com colunas separadas por caracteres de tabulação.

Uma particularidade das PNADs antigas é que as linhas de pessoas e domicílios encontram-se mescladas num mesmo arquivo. Uma solução será dividir o arquivo em dois antes de tentar carregar os dados com `read.fwf()`. Na PNAD de 1976, por exemplo, o valor presente na coluna 11 do arquivo de dados indica se a linha contém informação sobre um domicílio ou sobre uma pessoa (1 = domicílio, 2 = pessoa). Uma alternativa é utilizar a função `readLines()` para carregar o arquivo na memória do R, `grep()`, para identificar as linhas contendo o valor 2 na posição 11, e `writeLines()`, para salvar novos arquivos contendo apenas as linhas selecionadas (para detalhes sobre o uso da função `grep()`, ver a seção 10.1):

```
pnad76 <- readLines("PNAD76BR.DAT")
ind <- grep("^.........2", pnad76)
pes76 <- pnad76[ind]
dom76 <- pnad76[-ind]
writeLines(pes76, "pessoas76.dat")
writeLines(dom76, "domicilios76.dat")
```

[12] Disponível em http://www.lepem.ufc.br/jaa/sav2dat.html.
[13] Ver http://www.gnu.org/software/pspp/.

CAPÍTULO 5

MANIPULANDO BANCOS DE DADOS

Usualmente, a maior parte do trabalho de análise está na manipulação do banco de dados e não propriamente na elaboração de modelos analíticos. Mesmo que o banco de dados esteja com todas as variáveis categóricas codificadas, ainda podem ser necessários vários ajustes. Neste capítulo, veremos várias das tarefas comumente enfrentadas por um cientista social na fase de preparação de um banco de dados para análise, mas, antes de começar a manipular os dados, é preciso conhecê-los bem. Por isso, algumas seções deste capítulos tratam da descrição dos diferentes tipos de dados.

Não esqueça de registrar, num script, os comandos necessários para a adequada manipulação do banco de dados.

5.1 VISUALIZAÇÃO DOS DADOS

No R, não vemos os dados permanentemente, como numa planilha de cálculo e em alguns softwares de estatística. Não seria mesmo fácil visualizar o conteúdo dos objetos presentes na área de trabalho do R porque eles não são apenas dados tabulares; podem ser texto, vetores de diversos tamanho, tabelas, funções, listas de objetos etc. Além disso, pode não ser desprezível o custo computacional de atualizar a exibição, na tela, dos valores do banco de dados que estão sendo manipulados. Assim, para conhecer o conteúdo dos objetos, é

preciso, por meio de comandos, dizer ao R o que queremos. Para visualizar alguns desses comandos em ação, carregue o banco de dados `bd_exemplo1.csv` utilizado no capítulo anterior no objeto de nome "b". Digite:

```
b <- read.table("bd_exemplo1.csv", header = TRUE, sep = "",
                quote = '"')
```

Antes de iniciar a visualização dos dados, convém converter a variável `candidato` de `factor` para `character`, conforme explicado na seção 4.6:

```
b$candidato <- as.character(b$candidato)
```

Com o banco já carregado, digite e observe o resultado dos seguintes comandos:

```
head(b, n = 3)
#   uf                   candidato partido logvotos      sexo
# 1 RO         ACIR MARCOS GURGACZ     PDT   12,257 Masculino
# 2 ES       AFONSO CEZAR CORADINE    PSOL    9,937 Masculino
# 3 DF AGNELO SANTOS QUEIROZ FILHO PC do B   13,207 Masculino
#    resultado
# 1 Não eleito
# 2 Não eleito
# 3 Não eleito

tail(b, n = 3)
#     uf                    candidato partido logvotos      sexo
# 200 PB      WALTER AMORIM DE ARAÚJO    PRTB    9,513 Masculino
# 201 TO WEDER MARCIO DA SILVA SANTOS    PSDC    6,353 Masculino
# 202 PI ZILTON VICENTE DUARTE JUNIOR    PSOL     7,81 Masculino
#      resultado
# 200 Não eleito
# 201 Não eleito
# 202 Não eleito
```

O comando `head()` imprime no Console os seis primeiros valores do objeto que lhe é passado como argumento. Se esse objeto for um `data.frame`, ele imprime as seis primeiras linhas de todas as colunas. O número de valores ou linhas a ser impresso pode ser controlado pelo

parâmetro *n*. O comando `tail()` imprime os seis últimos valores ou linhas. Veremos, a seguir, várias outras formas de selecionar e exibir linhas e colunas específicas. Basicamente, tudo o que foi dito sobre extração de elementos e modificação de valores de matrizes e listas, nas seções 3.2 e 3.3, também é válido para `data.frames`.

Para escolher uma linha específica, por exemplo, a 26ª, podemos digitar:

```
b[26,]
#    uf                  candidato partido logvotos     sexo
# 26 AP CELISA PENNA MELO CAPELARI    PSOL    8,191 Feminino
#     resultado
# 26 Não eleito
```

De acordo com o sistema de índices do R, podemos fazer referência a um elemento de um `data.frame`, de uma `matrix` ou de uma `table`, indicando entre colchetes, respectivamente, o número da linha e o da coluna do elemento. No exemplo acima, não indicamos a coluna, e, por isso, todas foram impressas. É possível também selecionar várias linhas ou colunas simultaneamente e usar os nomes das linhas ou das colunas ao invés de seus índices, e imprimir fora da sequência em que se encontram no banco de dados, como nos exemplos abaixo (resultados dos comandos omitidos):

```
b[1:10,]
b[10:1, 1:3]
b[c(1, 2, 3, 4), c(1, 2, 3)]
b[c(4, 2, 1, 3), c(2, 3, 1)]
b[20:30, "candidato"]
b[1:10, c("partido", "resultado")]
```

É possível utilizar índices negativos para *não* imprimir linhas ou colunas (resultados omitidos):

```
b[1:10,-1]
b[c(1, 3, 5), -c(5, 3, 1)]
```

Também podemos utilizar condições lógicas para selecionar os índices das linhas a serem impressas. Nesse caso, a condição aplicada

a uma ou mais colunas deve retornar os índices que correspondem às linhas desejadas, e, por isso, deve ocupar o primeiro dos dois campos que estão entre os colchetes e que são separados por uma vírgula (resultados omitidos):

```
b[b$uf == "CE", c(2, 3, 6)]
b[b$idade > 80, c(1, 2, 4)]
b[b$uf == "AP" & b$sexo == "Feminino", 1:4]
b[b$uf == "RJ" | b$uf == "SP", 1:3]
```

A Tabela 2.3 (p. 15) explica o significa de cada símbolo.

E, é claro, para visualizar o banco de dados inteiro, basta digitar `b` (ou `b[,]` ou, ainda, `b[]`), mas não faça isso se o banco for muito grande, com dezenas de colunas e milhares de linhas. Para saber o número de linhas e de colunas de um `data.frame`, use o comando `dim()`, abreviatura de *dimensions*. Também é possível obter o número de colunas com comando `length()`, fornecendo o `data.frame` como parâmetro. Para saber o número de linhas, poderíamos fornecer uma das variáveis como parâmetro, uma vez que todas as variáveis têm o mesmo comprimento. Podemos, ainda, usar as funções `nrow()` e `ncol()` para saber as dimensões de um `data.frame`:

```
dim(b)
# [1] 202   6

c(length(b$resultado), length(b))
# [1] 202   6

c(nrow(b), ncol(b))
# [1] 202   6
```

Também pode ser importante saber qual a classe das diferentes variáveis de um banco de dados porque, às vezes, uma análise falha devido ao fato de a variável estar classificada de modo errado. Estar habituado às diferentes classes de objetos facilitará o entendimento das mensagens de erro e a consequente correção dos problemas. Para isso, podemos usar o comando `class()`.

5.2 EXTRAIR SUBCONJUNTO DOS DADOS

A partir de agora, utilizaremos uma versão mais completa do banco com dados dos candidatos ao Senado em 2006. Para tanto, carregue a área de trabalho `senado2006.RData`:

```
load("senado2006.RData")
```

Quando não se precisa de todas as variáveis do banco de dados, é recomendável a criação de um novo banco contendo apenas um subconjunto dos dados. É possível extrair um subconjunto com a seleção de índices de linhas e colunas, como visto na seção 5.1. No exemplo abaixo, será criado um novo banco de dados, contendo apenas as linhas do banco original cujo valor da variável `uf` é `"CE"`, sendo selecionadas apenas as variáveis `candidato`, `partido` e `votos`:

```
ce <- sen[sen$uf == "CE", c("candidato", "partido", "votos")]
ce
#                                  candidato partido   votos
# 16        ANTONIO FERNANDES DA SILVA FILHO    PSDC    3640
# 74  INÁCIO FRANCISCO DE ASSIS NUNES ARRUDA PC do B 1912663
# 139             MARIA NAIR FERNANDES SILVA     PDT   39327
# 149                     MORONI BING TORGAN     PFL 1680362
# 172             RAIMUNDO PEREIRA DE CASTRO    PSTU   18545
# 193            TARCÍSIO LEITÃO DE CARVALHO     PCB    6084
```

5.3 ORDENAR UM BANCO DE DADOS

Em algumas ocasiões, pode ser necessário ordenar um banco de dados. A forma de fazer isso no R é usar a função `order()` para criar um vetor de índices ordenados de acordo com algum critério e, em seguida, atribuir a um objeto o banco original reordenado pelo vetor. Por exemplo, para reordenar o banco `ce` criado na seção anterior pelo número decrescente de votos recebidos, podemos usar o seguinte código:

```
idx <- order(ce$votos, decreasing = TRUE)
ce <- ce[idx,]
ce
```

```
#                                 candidato partido   votos
# 74  INÁCIO FRANCISCO DE ASSIS NUNES ARRUDA PC do B 1912663
# 149                      MORONI BING TORGAN     PFL 1680362
# 139             MARIA NAIR FERNANDES SILVA     PDT   39327
# 172             RAIMUNDO PEREIRA DE CASTRO    PSTU   18545
# 193            TARCÍSIO LEITÃO DE CARVALHO     PCB    6084
# 16         ANTONIO FERNANDES DA SILVA FILHO    PSDC    3640
```

5.4 VISUALIZAÇÃO GRÁFICA DE VARIÁVEIS

Como já vimos em outras seções, para obter um sumário de um objeto temos a função `summary()`, que tanto pode ser aplicada a uma das colunas de um banco de dados quanto ao `data.frame` completo. O sumário é apenas numérico, mas a visualização de uma variável num gráfico é muitas vezes mais informativa. A função genericamente usada para produzir gráficos no R é `plot()`, a qual determina o tipo de gráfico a ser produzido conforme a classe do objeto que lhe é passado como argumento. Por exemplo, com uma variável categórica, ou seja, um objeto da classe `factor`, é produzido um gráfico de barras (figura não incluída):

```
plot(sen$resultado)
```

A função `freq()`, do pacote `descr`, produz um objeto que, impresso, exibe uma tabela de frequência de um objeto do tipo `factor`. O argumento *user.missing* indica quais categorias devem ser consideradas dados faltantes, o que afeta o cálculo do percentual válido. No código abaixo, produzimos uma tabela de frequência da renda familiar dos entrevistados da *Pesquisa Social Brasileira*, realizada em 2002:

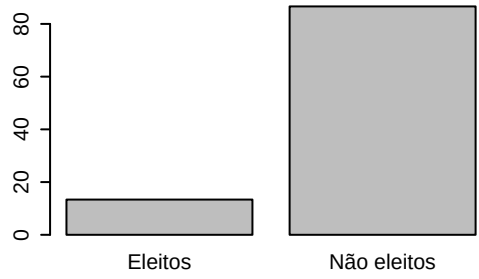
```
load("pesb2002.RData")
library(descr)
freq(pesb$q546, user.missing = c("NR", "NS"))
# Renda familiar - faixas
#                          Frequência Percentual % Válido
# Até R$ 200,00                   379     16.032   17.661
# De R$ 1001,00 a 2000,00         322     13.621   15.005
# De R$ 2001,00 a 4000,00         142      6.007    6.617
# De R$ 201,00 a 600,00           730     30.880   34.017
```

```
# De R$ 601,00 a 1000,00          398     16.836   18.546
# Mais de R$ 4001,00               81      3.426    3.774
# NR                               61      2.580
# NS                              157      6.641
# Sem renda                        94      3.976    4.380
# Total                          2364    100.000  100.000
```

O objeto produzido pela função `freq()` também possui um método para a função `plot()`, gerando um gráfico de barras quando plotado. Entre os argumentos da função `freq()`, está a escolha do tipo de apresentação das quantidades das categorias por meio de números absolutos ou relativos no eixo y do gráfico de barras. No exemplo da Figura 5.1, optou-se pelo uso de valores percentuais.

```
f <- freq(sen$resultado, plot = FALSE)
plot(f, y.axis = "percent")
```

FIGURA 5.1: Eleitos e não eleitos para o Senado em 2006



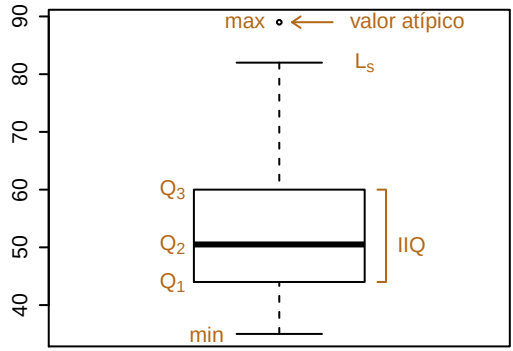
Para variáveis numéricas, a função `summary()` nos fornece o valor mínimo, o 1º quartil, a mediana, a média, o 3º quartil e o valor máximo. Uma ferramenta útil para visualização das características de uma variável numérica é o diagrama em caixa e bigodes (*boxplot*), o qual permite visualizar quase todas as informações exibidas pela função `summary()`. Ele revela, ainda, a existência de valores extremos ou atípicos (*outliers*).

```
summary(sen$idade)
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#    35.0    44.0    50.5    52.2    60.0    89.0
```

Na Figura 5.2, a base da caixa do diagrama representa o primeiro quartil, Q_1, a linha central representa a mediana, Q_2, e o topo da caixa, o terceiro quartil, Q_3. A distância entre Q_3 e Q_1 é o intervalo interquartil, IIQ. O limite do bigode inferior do diagrama, L_i, invisível no gráfico, é calculado como $Q_1 - (1,5 \cdot IIQ)$, e o limite superior, $L_s = Q_3 + (1,5 \cdot IIQ)$. Os valores que estiverem abaixo do limite inferior ou acima do limite superior são considerados atípicos. No caso da idade dos candidatos ao Senado, em 2006, não há nenhum valor abaixo de L_i, o que faz sentido, pois a Constituição do Brasil não permite a candidatura ao Senado de pessoas muito jovens, com menos de 35 anos. O valor mínimo, min, está acima de L_i e o único valor atípico, acima de L_s, é a idade de 89 anos de uma candidata do Rio Grande do Sul.

```
boxplot(sen$idade)
```

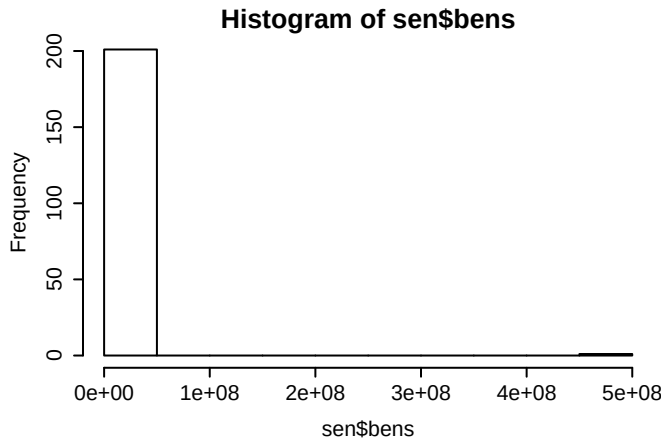
FIGURA 5.2: Diagrama em caixa explicado



Outro gráfico muito importante para a visualização de variáveis numéricas é o histograma. Nele, o vetor de números é subdividido em classes de valores e as barras representam a frequência dos valores em cada classe. Na Figura 5.3 temos um gráfico produzido por `hist()`:

```
hist(sen$bens)
```

FIGURA 5.3: Exemplo de histograma



O histograma da Figura 5.3 tem uma série de problemas que precisam ser corrigidos. O mais sério deles é que quase todos os candidatos estão na classe inferior de posse de bens. Verificando o resultado de

```
summary(sen$bens)
#     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
# 0.00e+00 2.46e+04 1.67e+05 3.57e+06 6.98e+05 4.93e+08
```

percebemos que o valor mínimo de `sen$bens` é 0, e o máximo é de 492 milhões. A função `hist()` dividiu esse intervalo em dez partes iguais; a primeira faixa, de 0 a 50 milhões, inclui todos, menos um candidato:

```
sen[sen$bens > 50000000, c("candidato", "uf", "partido")]
#                candidato uf partido
# 183 RONALDO CEZAR COELHO RJ    PSDB
```

Isso explica porque o histograma ficou tão deformado, com apenas uma grande barra à esquerda, e uma diminuta, quase invisível barra à direita: a barra da esquerda representa 221 candidatos e a barra da direita apenas um. Em sociedades com alto índice de desigualdade social, a riqueza não só é mal distribuída como a distribuição não é linear, mas geométrica. Por isso, comumente utilizamos o logaritmo da renda em análises estatísticas. Assim, antes de produzir um novo gráfico, vamos criar uma nova variável. Para evitar valores infinitos

(logaritmo de zero), vamos adicionar 1 à variável `bens` antes de calcular o logaritmo:

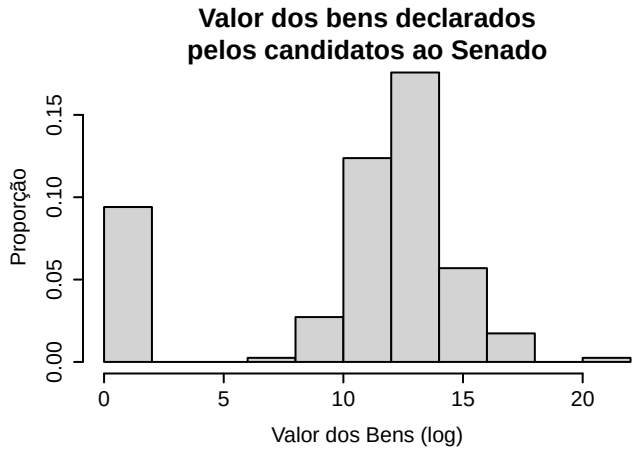
```
sen$log.bens <- log(sen$bens + 1)
```

No código acima, acrescentamos uma variável ao banco de dados atribuindo um vetor a um nome de variável inexistente no banco.

O gráfico da Figura 5.3 também tem problemas estéticos. Algumas das características definidas automaticamente pela função `hist()` ficariam mais apropriadas se definidas manualmente, como o título principal, os rótulos dos eixos e a cor das barras. Para melhorar o gráfico, podemos passar para `hist()` os argumentos *main*, *xlab*, *ylab* e *col*, cujos significados ficarão claros pela comparação dos comandos seguintes com a figura resultante (5.4):

```
hist(sen$log.bens, ylab = "Proporção", xlab = "Valor dos Bens (log)",
     main = "Valor dos bens declarados\npelos candidatos ao Senado",
     col = "lightgray", probability = TRUE)
```

FIGURA 5.4: Exemplo de histograma melhorado



Além de melhorarmos a aparência do gráfico, utilizamos o argumento *probability* para substituir a apresentação das frequências em valores absolutos por valores relativos. Os candidatos que declaram não possuir bens ficaram isolados na barra mais à esquerda, e Ronaldo

Cezar Coelho continua solitário na barra mais à direita do histograma, mas o restante do gráfico possui uma forma de sino, típica de uma distribuição normal. Chamamos de log-normal a uma distribuição cujo logaritmo se assemelha a uma distribuição normal.

5.5 RECODIFICAR VARIÁVEIS

Quando são aplicados questionários, muitas questões ficam sem receber resposta de boa parte dos entrevistados. Em alguns casos, a pergunta não se aplica ao indivíduo, em outros, o entrevistado, por algum motivo, preferiu não responder à pergunta e, finalmente, o entrevistado pode não saber responder à pergunta. Antes de realizar a análise dos dados pode ser necessário recodificar as variáveis para que as análises posteriores sejam bem-sucedidas. Digamos que uma determinada variável categórica `x` tenha entre seus `levels` os valores `"NS"` e `"NR"`, correspondendo, respectivamente, a "Não sabe" e "Não respondeu":

```
x <- as.factor(c("Sim", "Não", "NR", "Não", "NS", "Não", "Sim", "Sim"))
summary(x)
# Não  NR  NS Sim
#   3   1   1   3
```

Se quisermos tratar `"NS"` e `"NR"` como valores omissos (*missing values*), poderíamos fazê-lo da seguinte forma:

NA

```
x[x == "NS" | x == "NR"] <- NA
summary(x)
#  Não   NR   NS  Sim NA's
#    3    0    0    3    2

x <- droplevels(x)
summary(x)
#  Não  Sim NA's
#    3    3    2
```

Os valores entre colchetes na primeira linha do código acima, como o leitor já está habituado, fazem uma seleção dos índices de `x`.

Mas há várias novidades nessa linha. Observe que o símbolo `NA`, embora constituído por duas letras, não está entre aspas. O valor `NA` que está sendo atribuído a alguns dos valores de `x` tem um significado especial para o R: *not available*, ou seja, valor ausente. Vimos, na seção 2.6 (p. 14), os significados dos símbolos lógicos utilizados no código acima. Na primeira parte do código entre colchetes, ele significa `VERDADEIRO` se `x` for igual a `"NS"`, e FALSO se não for. O símbolo `|` significa a condição lógica `OU`. Assim, o código dentro dos colchetes pode ser lido como: `VERDADEIRO` se `x` for igual a `"NS"` ou `x` igual a `"NR"`; `FALSO` se `x` não for igual a nenhum dos dois valores. Ou seja, o valor `NA` será atribuído a todos os elementos do vetor `x` que corresponderem às categorias "Não sabe", "Não respondeu".

A função `droplevels()` tem o objetivo de eliminar, da variável categórica, os `levels` vazios, no caso, `"NS"` e `"NR"`.

No código acima, modificamos a própria variável, mas em outras ocasiões precisamos criar uma nova variável, como no exemplo abaixo, em que a função `recode()`, do pacote `memisc`, é utilizada para recodificar a variável *Partido político* em *Posicionamento em relação ao Governo Federal*.

```
library(memisc)
sen$pos <- recode(sen$partido,
  "Situação" <- c("PT", "PMDB", "PSB", "PV", "PTB", "PC do B", "PP",
                  "PRTB", "PL", "PMN", "PT do B", "PAN", "PSC", "PTC",
                  "PHS", "PTN", "PRB", "PRONA"),
  "Intermediário" <- c("PCB", "PDT", "PRP", "PSDC", "PSL"),
  "Oposição" <- c("PSDB", "PFL", "PPS", "PSOL", "PSTU", "PCO"))
summary(sen$pos)
#      Situação Intermediário      Oposição
#            81            50            71
```

Foram classificados na *Situação* aqueles partidos que não lançaram candidatura própria à Presidência da República em 2006 e que em 2008 estavam na base aliada do Governo Federal; como *Intermediários*, aqueles que lançaram candidatura própria, mas em 2008 estavam na base aliada do Governo; finalmente, como pertencentes à *Oposição*, aqueles que lançaram candidatura própria e faziam oposição ao Governo em 2008.

5.6 CRIAR VARIÁVEL CATEGÓRICA A PARTIR DE VARIÁVEL NUMÉRICA

Em alguns casos, uma variável, embora numérica, não tem efeito sempre positivo ou negativo sobre outra variável. A idade, por exemplo, é uma característica quantificável dos indivíduos que pode exercer influência sobre outras características de modo complexo. Para pessoas relativamente jovens, quanto maior a idade, em média, maior a escolaridade. Entretanto, pessoas um pouco mais velhas passaram sua juventude num período em que o acesso à escola era menos comum e, por isso, a partir de uma certa idade, quanto maior a idade, em média, menor a escolaridade. Em casos como esses, pode ser melhor converter a variável numérica numa variável categórica antes de continuar a análise dos dados. Isso pode ser feito com o comando `cut()`, fornecendo-lhe como argumentos a variável numérica, um vetor com os valores de corte mínimo, intermediários e máximo, e, opcionalmente, um vetor com os rótulos, como no exemplo:

```
sen$faixa.etaria <- cut(sen$idade, c(0, 44, 65, 90),
                        labels = c("Jovem", "Experiente", "Idoso"))
summary(sen$faixa.etaria)
#      Jovem Experiente      Idoso
#         56        120         26
```

Não há problema se os pontos de corte mínimo e máximo estiverem para além dos valores mínimo e máximo da variável numérica. Por outro lado, se o ponto de corte mínimo for superior ao valor mínimo da variável numérica ou se o ponto de corte máximo for inferior ao valor máximo, serão introduzidos `NA`s no novo vetor. Observe que o número de rótulos das categorias é igual ao número de pontos de corte −1.

5.7 ELIMINAR VARIÁVEIS EXISTENTES E ACRESCENTAR NOVAS

Quando atribuímos um novo valor a uma variável, como já fizemos várias vezes nas seções anteriores, na verdade, estamos destruindo a variável antiga e criando uma nova. Mas se a intenção for realmente eliminar uma variável de um banco de dados, basta lhe atribuir o valor `NULL`. A título de exemplo, digite `names(sen)` antes e depois do seguinte comando e compare os resultados:

```
sen$p.valido <- NULL
```

5.8 REUNIR DOIS BANCOS DE DADOS

Para reunir dois bancos de dados utilizamos a função `merge()`. Por exemplo, se tivermos um `data.frame` `b1` contendo o valor do IDH dos Estados do Sul do Brasil para o ano de 2005 e um `data.frame` b2 contendo a população estimada desses mesmos Estados, em milhões, para 2007

```
b1 <- data.frame(uf = c("RS", "SC", "PR"), idh = c(0.83, 0.84, 0.82))
b2 <- data.frame(UF = c("PR", "SC", "RS"), pop = c(10.3, 5.9, 10.6))
```

poderemos reuni-los num único banco de dados com o comando `merge(b1, b2)`. O pré-requisito para o correto funcionamento de `merge()` é que a variável chave para fusão esteja presente com o mesmo nome nos dois bancos. No exemplo em questão, para a comparação das chaves realizada por `merge()` ser possível, teremos que renomear uma das variáveis chave para que os nomes fiquem exatamente iguais e converter as colunas correspondentes à Unidade da Federação de `factor` para `character`:

```
names(b2) <- c("uf", "pop")
b1$uf <- as.character(b1$uf)
b2$uf <- as.character(b2$uf)
merge(b1, b2)
#   uf  idh  pop
# 1 PR 0.82 10.3
# 2 RS 0.83 10.6
# 3 SC 0.84  5.9
```

Note que se as variáveis `uf` e `UF` fossem exatamente iguais, ou seja, se as unidades da federação estivessem na mesma ordem, seria desnecessário usar `merge()`, sendo suficiente utilizar `cbind()`. Em outro cenário, se os dois bancos de dados contivessem exatamente as mesmas variáveis, e na mesma ordem, poderíamos estar interessados em acrescentar novas linhas (casos) ao banco, e não novas colunas. Nesse caso, o comando apropriado seria `rbind()`.

5.9 REFORMATAR BANCO DE DADOS

A função `reshape()` reformata um banco de dados de modo que medições diferentes de uma variável que estejam em linhas diferentes tornem-se diferentes colunas de uma mesma linha, ou o contrário, que medições diferentes de uma variável que estão em colunas diferentes fiquem na mesma coluna, mas em linhas diferentes. No exemplo abaixo, criamos um banco com dados sobre a população de alguns países em três anos diferentes e, em seguida, usamos a função `reshape()` para reformatar o banco de dados. O argumento *timevar* indica qual variável contém informação sobre o momento da observação e o argumento *idvar* indica qual variável deve ser usada como chave para reformatar o banco de dados. O argumento *direction* indica se queremos reformatar o banco de dados de modo a torná-lo mais largo ou mais comprido:

```
pais <- c(rep("Brasil", 3), rep("Bulgária", 3), rep("Índia", 3))
ano <- rep(c(1960, 1980, 2000), 3)
populacao <- c(71695, 122958, 175553, 7867, 8844, 7818, 445857,
               687029, 1002708)
b3 <- data.frame(pais, ano, populacao)
b3
#       pais  ano populacao
# 1   Brasil 1960     71695
# 2   Brasil 1980    122958
# 3   Brasil 2000    175553
# 4 Bulgária 1960      7867
# 5 Bulgária 1980      8844
# 6 Bulgária 2000      7818
# 7    Índia 1960    445857
# 8    Índia 1980    687029
# 9    Índia 2000   1002708

reshape(b3, timevar = "ano", idvar = "pais", direction = "wide")
#       pais populacao.1960 populacao.1980 populacao.2000
# 1   Brasil          71695         122958         175553
# 4 Bulgária           7867           8844           7818
# 7    Índia         445857         687029        1002708
```

No exemplo seguinte, fizemos o contrário, reformatamos um banco de dados de modo a ter as informações arranjadas no formato mais comprido. O argumento *varying* indica quais colunas no formato largo correspondem a uma única variável no formato comprido. Após a

reformatação do banco, os nomes das linhas ficaram longos e com informação redudante, e, por isso, usamos a função `rownames()` para modificá-los:

```
pais <- c("Brasil", "Bulgária", "Índia")
pop.1960 <- c(71695, 7867, 445857)
pop.1980 <- c(122958, 8844, 687029)
pop.2000 <- c(175553, 7818, 1002708)
b5 <- data.frame(pais, pop.1960, pop.1980, pop.2000)
b5
#       pais pop.1960 pop.1980 pop.2000
# 1   Brasil    71695   122958   175553
# 2 Bulgária     7867     8844     7818
# 3    Índia   445857   687029  1002708

b6 <- reshape(b5, idvar = "pais", timevar = "ano", direction = "long",
              varying = c("pop.1960", "pop.1980", "pop.2000"))
b6
#                   pais  ano     pop
# Brasil.1960     Brasil 1960   71695
# Bulgária.1960 Bulgária 1960    7867
# Índia.1960       Índia 1960  445857
# Brasil.1980     Brasil 1980  122958
# Bulgária.1980 Bulgária 1980    8844
# Índia.1980       Índia 1980  687029
# Brasil.2000     Brasil 2000  175553
# Bulgária.2000 Bulgária 2000    7818
# Índia.2000       Índia 2000 1002708

rownames(b6) <- 1:9
```

5.10 ATRIBUTOS DE OBJETOS

Ao contrário do SPSS, em que todas as variáveis de um banco de dados possuem um rótulo descritivo, os objetos no R não possuem esses rótulos, mas podemos acrescentá-los manualmente. A função `descr()` verifica se o objeto possui o atributo *label* e o imprime antes de chamar a função `summary()`. Isso é bastante útil quando temos um banco de dados com dezenas de variáveis e não lembramos o que exatamente cada uma delas representa.[1] Para acrescentar um atributo

[1] Entre as funções presentes no pacote `Hmisc`, temos `label()`, que acrescenta o atributo `label` a um objeto, e `spss.get()`, que atribui automaticamente rótulos às

a um objeto, usamos o comando `attr()`. O mesmo comando permite obter o valor de um atributo e o comando `attributes()` lista todos os atributos de um objeto. Seguem alguns exemplos que utilizam os comandos mencionados:

```
class(sen$sexo)
# [1] "factor"

attr(sen$sexo, "label")
# NULL

attr(sen$sexo, "label") <- "Sexo do candidato"
descr(sen$sexo)
# sen$sexo - Sexo do candidato
#  Feminino Masculino
#        30       172

attributes(sen$sexo)
# $levels
# [1] "Feminino"  "Masculino"
#
# $class
# [1] "factor"
#
# $label
# [1] "Sexo do candidato"
```

Como fica claro pelo resultado do comando `attributes()`, os objetos no R armazenam muitas informações como atributos. Um objeto do tipo `factor`, por exemplo, é apenas um vetor do tipo `integer` com alguns atributos adicionais: `levels` e `class`.

A possibilidade de atribuição de características arbitrárias a qualquer objeto, combinada com a possibilidade de criar novas classes e atribuí-las aos objetos, é uma das características da linguagem R que lhe proporcionam grande flexibilidade e poder. Entretanto, os atributos de um objeto são perdidos quando ele é transformado de um tipo em outro, o que frequentemente precisamos fazer durante a fase de manipulação do banco de dados. Por isso, quando se pretende acres-

variáveis de um banco de dados importado do `SPSS`. A função `label()`, entretanto, muda a classe do objeto, o que pode causar problemas de compatibilidade com funções de outros pacotes. Por isso, neste livro, não usaremos o pacote `Hmisc`.

centar rótulos ou outros atributos às variáveis de um banco de dados, é recomendável que isso seja feito como última etapa da manipulação.

Concluída a manipulação do banco de dados, salve o script que escreveu durante essa fase do processo de análise de dados. Ele será necessário se for percebida alguma falha na manipulação dos dados. Nesse caso, bastará fazer a correção no script e executá-lo novamente. Salve também o banco de dados, pois, geralmente, é mais rápido carregar o banco de dados já manipulado e salvo no formato `RData` do que executar novamente todo o script de manipulação dos dados:

```
save(sen, file = "senado2006.novo.RData")
```

CAPÍTULO 6

ANÁLISE DESCRITIVA

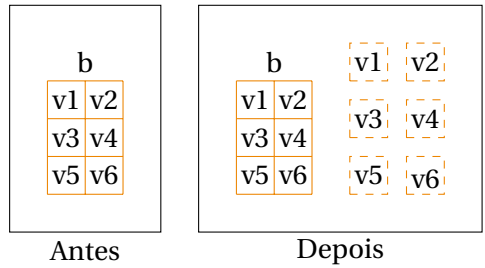
Neste capítulo, veremos as análises que podem ser feitas sem o uso de técnicas avançadas de estatística. Elas são muito úteis para a exploração inicial do banco de dados e podemos ter que nos limitar a elas quando vamos apresentar um relatório de pesquisa para um público leigo em estatística.

6.1 VARIÁVEIS DE UM BANCO DE DADOS NA ÁREA DE TRABALHO

Até aqui, sempre que precisamos fazer referência a uma variável de um banco de dados, usamos a notação `b$v`, onde `b` representa um `data.frame` e `v` uma de suas colunas ou variáveis. A função `attach()` permite fazer referência a uma variável de um banco de dados diretamente, sem o prefixo `b$`. Ao digitarmos `attach(b)`, o R faz cópias das variáveis de `b` que, embora não sejam exibidas pela função `ls()`, podem ser acessadas pelas demais funções. A Figura 6.1 (próxima página) é uma representação gráfica do processo.

O inconveniente de usar `attach()` durante o processo de manipulação dos dados decorre do fato de alterações nos objetos anexados à área de trabalho não se refletirem nas variáveis do `data.frame()` e vice-versa. Seria preciso usar, constantemente, a função `detach()` para desanexar os objetos anexados pela última chamada a `attach()`. Como isso é causa frequente de confusão entre iniciantes, evitaremos o uso de `attach()` neste livro.

FIGURA 6.1: Área de trabalho antes e depois de `attach(b)`



Nas primeiras seções deste capítulo, utilizaremos um subconjunto do banco de dados da *Pesquisa Social Brasileira* de 2002. O primeiro passo é, pois, carregar o banco de dados:

```
load("pesb2002.RData")
```

A Tabela 6.1 reproduz o nome de algumas variáveis e seus respectivos rótulos, conforme o livro de códigos da PESB 2002.

Tabela 6.1: Algumas variáveis da PESB 2002 e seus respectivos rótulos

Var.	Rótulo
q2	Região do País
q66	Pedir a um amigo que trabalha no serviço público para ajudar a tirar um documento mais rápido do que no tempo normal é:
q68	Um funcionário público receber um presente de Natal de uma empresa que ele ajudou a ganhar um contrato do governo é:
q78	Alguém consegue um empréstimo do governo, mas que demora muito a sair. Como ele tem um parente no governo, conseguir liberar o empréstimo mais rápido é:
q105	Atitude que a empregada doméstica deveria ter se a patroa diz que ela pode assistir televisão na sala junto com ela
q511	Religião
q531	Sexo
q546	Renda familiar - faixas
q660	Peso final para desproporcionalidade amostral e diferenças sóciodemográficas

Para facilitar o nosso trabalho, vamos renomear as variáveis do nosso subconjunto da PESB 2002 com nomes mais significativos do

que os originais. Para evitar erros de digitação na produção do novo vetor com nomes, usaremos o comando `dput(names(pesb))` para produzir um código que podemos copiar e editar, modificando os nomes conforme nosso interesse:

```
dput(names(pesb))
# c("q2", "q531", "q511", "q546", "q66", "q68", "q78", "q105",
# "q660")

names(pesb) <- c("regiao", "sexo", "religiao", "rendafam", "docrapido",
                 "natal", "emprestimo", "atitude", "peso")
```

Usaremos várias funções do pacote `descr`. Por isso, vamos carregá-lo na memória do R:

```
library(descr)
```

6.2 CONSTRUÇÃO DE ÍNDICES

Índices são construídos pela computação de novas variáveis a partir de variáveis existentes. Eles são úteis para condensar informações que de outra forma seriam difíceis de analisar. Na PESB, por exemplo, existem 19 perguntas sobre situações que o entrevistado deve classificar como favor, jeitinho ou corrupção. As três que Almeida (2007) aponta como as que obtiveram respostas mais variadas foram: (1) Pedir a um amigo que trabalha no serviço público para ajudar a tirar um documento mais rápido do que o normal. (2) Um funcionário público recebe um presente de Natal de uma empresa que ele ajudou a ganhar um contrato do governo. (3) Alguém consegue um empréstimo do governo, mas que demora muito a sair. Como ela tem um parente no governo consegue liberar o empréstimo mais rápido.

No nosso banco de dados, estas variáveis estão nomeadas `docrapido`, `natal` e `emprestimo`. As três, é claro, têm as mesmas categorias:

```
levels(pesb$docrapido)
# [1] "Favor"                          "Mais favor do que jeitinho"
# [3] "Mais jeitinho do que favor"     "Jeitinho"
# [5] "Mais jeitinho do que corrupção" "Mais corrupção do que jeitinho"
# [7] "Corrupção"                      "NR"
# [9] "NS"
```

```
levels(pesb$emprestimo)
# [1] "Favor"                          "Mais favor do que jeitinho"
# [3] "Mais jeitinho do que favor"     "Jeitinho"
# [5] "Mais jeitinho do que corrupção" "Mais corrupção do que jeitinho"
# [7] "Corrupção"                      "NR"
# [9] "NS"

levels(pesb$natal)
# [1] "Favor"                          "Mais favor do que jeitinho"
# [3] "Mais jeitinho do que favor"     "Jeitinho"
# [5] "Mais jeitinho do que corrupção" "Mais corrupção do que jeitinho"
# [7] "Corrupção"                      "NR"
# [9] "NS"
```

Mesmo utilizando apenas essas três variáveis, a quantidade de números a serem visualizados em tabelas cruzadas tornaria difícil a interpretação dos dados. Em situações como essas, é útil construir um índice que sumarize as informações de um conjunto de variáveis. Assim, vamos utilizar da PESB as variáveis `docrapido`, `natal` e `emprestimo` para construir um *índice de percepção da corrupção*, `ipc` (não presente no livro de Almeida). Entretanto, antes da construção do índice, vamos eliminar, das variáveis, as respostas *NS* e *NR*:

```
pesb$emprestimo[pesb$emprestimo == "NS" |
                pesb$emprestimo == "NR"] <- NA
pesb$docrapido[pesb$docrapido == "NS" | pesb$docrapido == "NR"] <- NA
pesb$natal[pesb$natal == "NS" | pesb$natal == "NR"] <- NA
```

O índice será simplesmente a soma dos `levels` das variáveis (que variam de 1 a 7). Para que o índice varie de 0 a 18 e não de 3 a 21, subtrairemos 3 da soma:

```
pesb$ipc <- as.numeric(pesb$docrapido) + as.numeric(pesb$natal) +
            as.numeric(pesb$emprestimo) - 3
attr(pesb$ipc, "label") <- "Índice de percepção de corrupção"
summary(pesb$ipc)
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
#     0.0     6.0    11.0    10.1    14.0    18.0     206
```

Nas seção seguinte teremos oportunidade de utilizar o índice. O atributo *label* adicionado ao objeto `ipc` será utilizado pela função

`compmeans()` para criar o título da tabela impressa na tela (ver Figura 6.2).

6.3 UMA VARIÁVEL NUMÉRICA E OUTRA CATEGÓRICA

Quando queremos saber a relação entre uma variável numérica e outra categórica, a análise descritiva mais adequada é a comparação do valor médio da variável numérica segundo as diversas categorias da categórica, mas, antes de fazer essa comparação, vamos usar a função `levels()` para abreviar os nomes das regiões e, assim, melhorar a aparência da tabela e do gráfico que serão produzidos:

```
levels(pesb$regiao)
# [1] "Centro-Oeste" "Nordeste"     "Norte"        "Sudeste"
# [5] "Sul"

levels(pesb$regiao) <- c("CO", "NE", "N", "SE", "S")
```

A comparação de média pode ser feita com a função `compmeans()` do pacote `descr`. A função produz um objeto que, ao ser impresso, produz uma tabela com as médias e, ao ser plotado, produz um conjunto de diagramas de caixa. A seguir, utilizamos a função `compmeans()`, tendo por argumentos o índice de percepção de corrupção que criamos na seção 6.2 e a Região do País onde morava o entrevistado. Também forneceremos como argumento o peso do indivíduo na amostra (variável `Q660` no banco original do CIS, `peso` em nosso subconjunto dos dados), o que torna mais acurada a descrição das características da população:[1]

```
compm <- compmeans(pesb$ipc, pesb$regiao, pesb$peso)
```

```
compm
# Valor médio de "Índice de percepção de corrupção" segundo
# "Estrato-Região do País"
#        Média    N Desv. Pd.
# CO    10.960  149     4.402
```

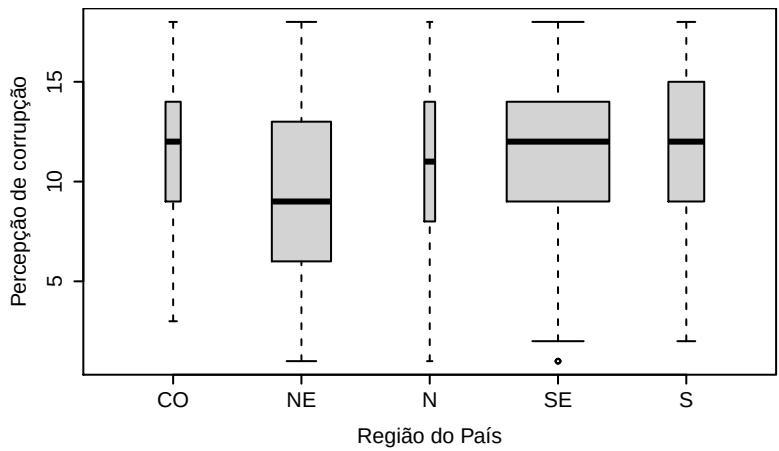
[1] Alguns programas de estatística permitem definir uma variável como o peso a ser utilizado nas análises. No R isso não é possível, sendo necessário fornecer o peso como argumento para cada uma das funções que incluem essa opção.

```
# NE     8.705  580     4.895
# N     10.717  106     4.372
# SE    11.047 1005     4.159
# S     10.993  351     4.317
# Total 10.397 2191     4.528
```

Na tabela de comparação de médias acima, temos o valor médio do `ipc` segundo a Região, o número de indivíduos entrevistados em cada Região (`N`), e o desvio padrão do `ipc` dos indivíduos segundo a Região. Ao plotarmos o objeto resultante da função `compmeans()`, obtemos o conjunto de diagramas de caixa mostrados na Figura 6.2. Nesta figura, a largura das caixas é proporcional ao número de entrevistados na pesquisa.

```
plot(compm, ylab = "Percepção de corrupção", xlab = "Região do País")
```

FIGURA 6.2: Diagramas em caixa: percepção da corrupção segundo a Região

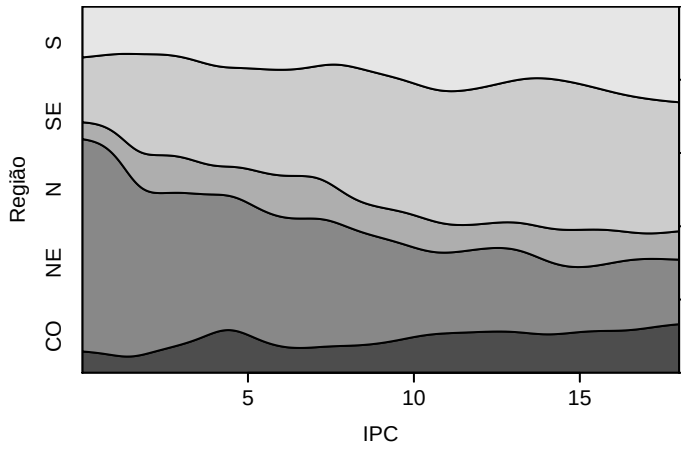


As pessoas das regiões com `ipc` mais alto, em média, interpretam em maior proporção as situações apresentadas como casos de corrupção (algo negativo) do que de jeitinho (algo aceitável) ou favor (algo positivo), e suponho que elas se sentirão mais inibidas em praticar atos semelhantes do que aquelas que consideram que as situações representam casos de jeitinho ou favor. Na Figura 6.3, temos um gráfico de densidade condicional também ilustrando a relação entre percepção

da corrupção e Região de moradia dos entrevistados. Os nordestinos constituem mais da metade dos que têm baixa percepção de corrupção e são menos de 20% dos que têm a mais alta percepção de corrupção.

```
b <- pesb[is.na(pesb$ipc) == FALSE, c("ipc", "regiao")]
cdplot(b$ipc, b$regiao, xlab = "IPC", ylab = "Região",
       border = "black")
```

FIGURA 6.3: Gráfico de densidade condicional: proporção de entrevistados de cada Região segundo a percepção de corrupção



6.4 DUAS VARIÁVEIS CATEGÓRICAS

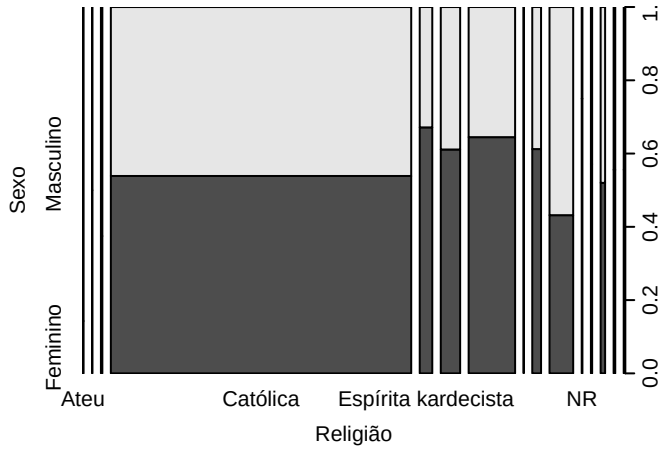
Para saber se existe alguma correlação entre duas variáveis categóricas, podemos fazer uma tabela cruzada e produzir um gráfico com as duas variáveis. As funções que fazem isso são `table()` e `plot()` (ver Figura 6.4). Nesta seção, exemplificaremos como fazer análise de duas variáveis categóricas, verificando se diferentes religiões atraem igualmente homens e mulheres.

```
table(pesb$religiao, pesb$sexo)
#
#                                          Feminino Masculino
#   Ateu                                          1         6
#   Budista                                       1         1
#   Candomblé                                     3         5
```

```
#   Católica                                902       772
#   Espírita kardecista                      47        23
#   Evangélica não-pentecostal               66        42
#   Evangélica pentecostal                  167        92
#   Judaica                                   1         0
#   Mormon, Adventista, Testemunha de Jeová  30        19
#   Não tem religião                         57        75
#   NR                                        3         1
#   NS                                        4         3
#   Santo Daime, Esotérica, Outras           13        12
#   Seisho-Nô-Iê, Messiânica                  5         4
#   Umbanda                                   5         4
```

```
plot(pesb$religiao, pesb$sexo, xlab = "Religião", ylab = "Sexo")
```

FIGURA 6.4: Gráfico mosaico: religião e sexo do indivíduo



A tabela acima e a Figura 6.4 precisam de algumas melhorias. Um primeiro problema a ser resolvido é a eliminação das categorias com pequeno número de casos. Vamos criar uma nova variável, `rlg`, reunindo algumas categorias e eliminando outras:

```
library(memisc)
pesb$rlg <- recode(pesb$religiao,
  "Evangélica" <- c("Mormon, Adventista, Testemunha de Jeová",
                    "Evangélica não-pentecostal",
                    "Evangélica pentecostal"),
```

```
  "Católica" <- "Católica",
  "Nenhuma" <- c("Ateu", "Não tem religião"))
```

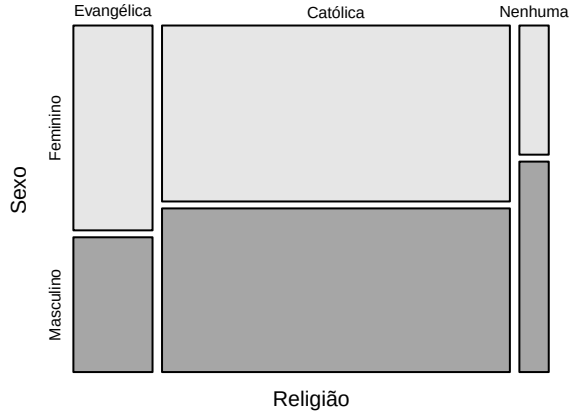
A forma mais fácil de fazer isso é utilizando `crosstab()`, do pacote `descr` que, além de aceitar um vetor de pesos como argumento, produz um objeto que, ao ser impresso, imprime na tela uma tabela e, ao ser plotado, produz um `mosaicplot`. Temos, abaixo, a tabela produzida pela função `crosstab()` e, em seguida, o gráfico (Figura 6.5):[2]

```
ct <- crosstab(pesb$rlg, pesb$sexo, pesb$peso)
ct
#    Conteúdo das células
# |-------------------------|
# |                Contagem |
# |-------------------------|
#
# ========================================
#               pesb$sexo
# pesb$rlg      Feminino   Masculino   Total
# ----------------------------------------
# Evangélica         231         152     383
# ----------------------------------------
# Católica           872         811    1683
# ----------------------------------------
# Nenhuma             54          88     142
# ----------------------------------------
# Total             1157        1051    2208
# ========================================
```

[2] A função `crosstab()` aceita vários outros argumentos úteis, mas eles tornam os resultados mais complexos, exigindo maior conhecimento estatístico do leitor, e, por isso, somente serão utilizados no Capítulo 7.

```
plot(ct, xlab = "Religião", ylab = "Sexo")
```

FIGURA 6.5: Gráfico mosaico: religião e sexo do indivíduo (II)



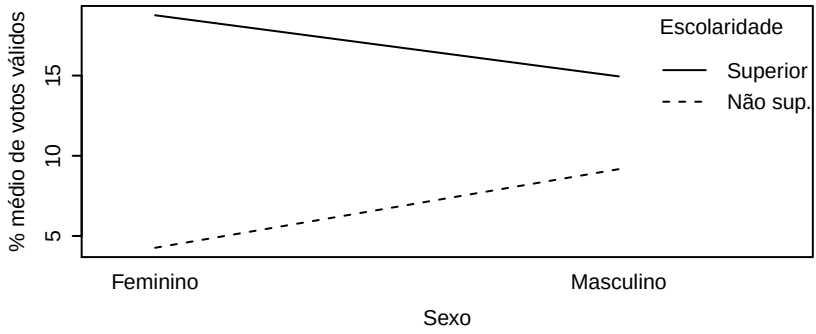
Nos próximos exemplos, usaremos os dados sobre candidatos ao Senado nas eleições de 2006. No código abaixo, carregamos o banco de dados e recodificamos a variável escolaridade de modo a reduzir seu número de categorias:

```
load("senado2006.RData")
sen$escola.superior <- recode(sen$escola,
                              "Superior" <- "Ensino Superior completo",
                              otherwise = "Não sup.")
```

Em seguida, na Figura 6.6, produzimos um gráfico de interação que permite perceber a relação entre duas variáveis categóricas e uma numérica. Entre as pessoas com escolaridade alta, a média do percentual de votos válidos é maior para as mulheres, mas entre as pessoas com escolaridade baixa ocorre o contrário.

```
interaction.plot(sen$sexo, sen$escola.superior, sen$p.valido,
                 xlab = "Sexo", ylab = "% médio de votos válidos",
                 trace.label = "Escolaridade", lty = 1:2)
```

FIGURA 6.6: Gráfico de interação: sexo, escolaridade e votos válidos



6.5 DUAS VARIÁVEIS NUMÉRICAS

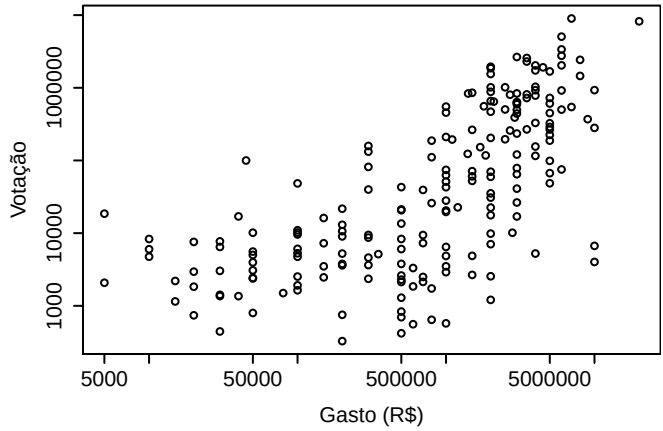
A abordagem mais adequada para verificar a existência de correlação entre duas variáveis numéricas é a análise de regressão, como veremos no capítulo 7. Se o público leitor não tiver conhecimento de estatística suficiente para interpretar uma tabela com os resultados de uma análise de regressão, o que podemos fazer é produzir um gráfico de dispersão das duas variáveis. Para exemplificar, verificaremos a relação entre gasto de campanha e votação recebida. Utilizaremos um subconjunto dos dados, excluindo os candidatos que não receberam nenhum voto. Assim como a variável renda no capítulo anterior (ver Figura 5.4), as duas variáveis têm distribuição mais próxima de uma log-normal do que de uma normal. Por isso, em algumas análises, utilizaremos o logaritmo das variáveis, e na produção de gráficos poderemos utilizar o argumento *log* para informar se a escala do eixo x e/ou do eixo y deverá(ão) ser logarítmica(s).

Para produzir um gráfico de dispersão entre duas variáveis numéricas, basta utilizar a função `plot()`, fornecendo as duas variáveis como argumento. A função reconhecerá que as duas variáveis são numéricas e chamará o método adequado, como na Figura 6.7. Antes de chamarmos a função `compmeans()`, modificamos o parâmetro gráfico *scipen* para evitar o uso de notação científica nos rótulos dos eixos do gráfico.[3]

[3] A manipulação de parâmetros gráficos será abordada no Capítulo 8.

```
options(scipen = 1000)
plot(sen$vgasto, sen$votos, log = "xy", xlab = "Gasto (R$)",
     ylab = "Votação")
```

FIGURA 6.7: Diagrama de dispersão: votação segundo os gastos de campanha



É possível perceber uma maior densidade dos pontos nas interseções entre baixo gasto e baixa votação e entre alto gasto e alta votação, mas seria preciso fazer uma análise de regressão para saber o valor exato dessa correlação e sua significância estatística. O que ainda podemos fazer nos limites da proposta deste capítulo é converter as variáveis em categóricas e utilizar `compmeans()` e `crosstab()` para verificar a existência de relação entre elas. A conversão de uma variável numérica em categórica pode ser feita com a função `cut()`, como vimos na seção 5.6:

```
sen$gastos <- cut(sen$vgasto, c(0, 760000, 2000000, 100000000),
                  labels = c("Baixo", "Médio", "Alto"),
                  ordered_result = TRUE)
sen$votac <- cut(sen$votos, c(0, 14000, 230000, 10000000),
                 labels = c("Baixa", "Média", "Alta"),
                 ordered_result = TRUE)
```

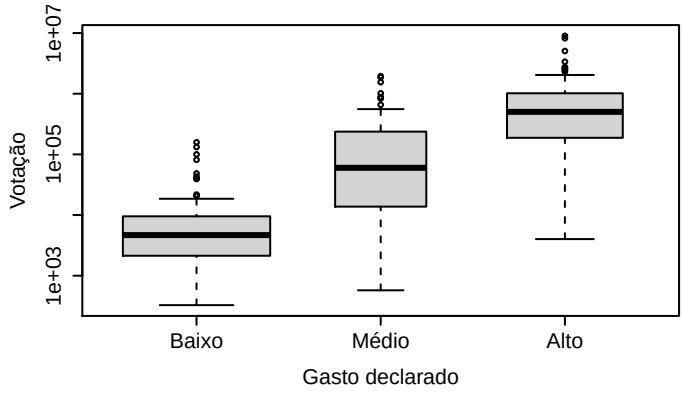
Criadas as novas variáveis categóricas, podemos agora usar `compmeans()`. A tabela produzida pode ser vista a seguir, assim como o

gráfico resultante pode ser visto na Figura 6.8. O argumento *log* = "y" torna logarítmica a apresentação dos valores no eixo *y*.

```
vg <- compmeans(sen$votos, sen$gastos)
vg
# Valor médio de "sen$votos" segundo "sen$gastos"
#         Média   N Desv. Pd.
# Baixo   12709  84     26287
# Médio  269561  52    462881
# Alto  1049986  66   1652811
# Total  417742 202   1069372
```

```
plot(vg, ylab = "Votação", xlab = "Gasto declarado", log = "y")
```

FIGURA 6.8: Diagramas em caixa: votação segundo os gastos de campanha



Também podemos apresentar uma tabela cruzada com as duas novas variáveis categóricas:

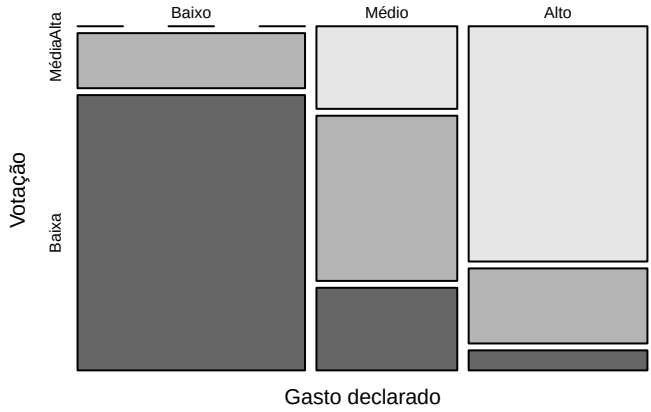
```
ct2 <- crosstab(sen$gastos, sen$votac)
ct2
#    Conteúdo das células
# |-------------------------|
# |                Contagem |
# |-------------------------|
#
# ========================================
#               sen$votac
# sen$gastos    Baixa   Média   Alta   Total
```

```
# ----------------------------------------
# Baixo            70      14       0      84
# ----------------------------------------
# Médio            13      26      13      52
# ----------------------------------------
# Alto              4      15      47      66
# ----------------------------------------
# Total            87      55      60     202
# ========================================
```

Para evitar que o gráfico produzido por `crosstab()` apresente a categoria "Baixa votação" na parte superior do gráfico e "Alta votação" na parte inferior, acrescentaremos o argumento *inv.y = TRUE* (*inverter a ordem das categorias no eixo y*). O resultado é a Figura 6.9.

```
plot(ct2, xlab = "Gasto declarado", ylab = "Votação", inv.y = TRUE)
```

FIGURA 6.9: Gráfico mosaico: votação segundo os gastos de campanha



As figuras 6.8 e 6.9 contêm, respectivamente, diagramas em caixa e um gráfico mosaico, e apresentam mais claramente a existência de correlação entre gastos de campanha e votação do que o gráfico de dispersão da Figura 6.7, sendo boas alternativas de apresentação visual dos dados quando se prefere não utilizar análise de regressão para demonstrar a relação entre duas variáveis numéricas.

Neste capítulo, não vamos mais precisar dos dados sobre senadores e, portanto, vamos remover o objeto `sen` da área de trabalho:

```
rm(sen)
```

6.6 SÉRIES TEMPORAIS

Usamos a função `ts()` para converter um vetor numérico em objeto da classe `ts` (série temporal). A função recebe como argumentos obrigatórios o vetor de números, e os parâmetros de tempo inicial (*start*) e final (*end*). No exemplo a seguir, criamos duas séries temporais, relativas ao número de ocorrências de furtos consumados e de latrocínios no Estado de São Paulo no período de 1997 a 2004.[4] O número de furtos é quase mil vezes maior do que o número de latrocínios e, para melhor visualização das duas variáveis num único gráfico, dividimos o número de furtos por 1000 no momento de criar a série temporal.

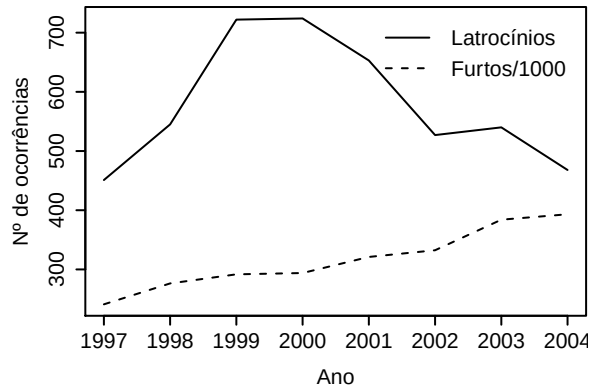
```
fur <- c(241026, 276326, 291701, 293900, 321086, 332379,
         384004, 393153)
lat <- c(451, 545, 722, 724, 653, 527, 540, 468)
ts.fur <- ts(fur / 1000, start = 1997, end = 2004)
ts.lat <- ts(lat, start = 1997, end = 2004)
```

Gráficos de séries temporais são criados pela função `ts.plot()`, cujos primeiros argumentos devem ser objetos da classe `ts`, como no exemplo da Figura 6.10.

```
ts.plot(ts.lat, ts.fur, lty = 1:2, ylab = "Nº de ocorrências",
        xlab = "Ano")
legend("topright", legend = c("Latrocínios", "Furtos/1000"),
       lty = 1:2, bty = "n")
```

[4] Dados disponibilizados pela Fundação Sistema Estadual de Análise de Dados, SEADE, e pela Secretaria de Segurança Pública do Estado de São Paulo, SSP/SP: http://www.seade.gov.br/projetos/acervossp/imp.php.

FIGURA 6.10: Séries temporais: latrocínios e furtos em São Paulo



Para maiores detalhes sobre como trabalhar com séries temporais, consulte Torgo (2006, p. 31 e ss.).

CAPÍTULO 7

QUI-QUADRADO E REGRESSÃO

7.1 QUI-QUADRADO

Vimos no capítulo 6 que a função `crosstab()` pode ser utilizada no lugar da função `table()` com algumas vantagens. Faremos isto neste capítulo ao cruzar as respostas dadas à pergunta *Atitude que a empregada doméstica deveria ter se a patroa diz que ela pode assistir televisão na sala junto com ela*, cujas opções de resposta foram *Assistir TV na sala, mas pegar uma cadeira na cozinha*, *Assistir TV no seu quarto*, *Sentar no sofá junto da patroa e assistir TV com ela*. Mas, antes, para reduzir o espaçamento horizontal ocupado pela tabela, usamos a função `levels()`, para abreviar os rótulos da variável, e, para obter resultados mais significativos, reduzimos o número de categorias da variável *Renda familiar*, recodificando-a com a função `recode()` do pacote `memisc`:

```
load("pesb2002.RData")
peso <- pesb$q660
atitude <- pesb$q105
atitude[atitude == "NR" | atitude == "NS"] <- NA
atitude <- droplevels(atitude)
dput(levels(atitude))
# c("Assistir TV na sala, mas pegar uma cadeira na cozinha",
# "Assistir TV no seu quarto",
# "Sentar no sofá junto da patroa e assistir TV com ela")
```

```
levels(atitude) <- c("Pegar cadeira", "Assistir no quarto",
                     "Sentar no sofá")
library(gdata)
atitude <- reorder(atitude, new.order = c(2, 1, 3))
library(memisc)
rendafam <- recode(factor(pesb$q546),
   "Baixa" <- c("Sem renda", "Até R$ 200,00", "De R$ 201,00 a 600,00"),
   "Média" <- c("De R$ 601,00 a 1000,00", "De R$ 1001,00 a 2000,00"),
   "Alta"  <- c("De R$ 2001,00 a 4000,00", "Mais de R$ 4001,00"))
```

Vamos agora passar para a função `crosstab()` os argumentos *expected* = *TRUE*, o que fará a função mostrar, no interior das células da tabela, os valores esperados no caso de ausência de correlação entre as variáveis, e *chisq* = *TRUE* para a realização de um teste de qui-quadrado.

```
library(descr)
crosstab(atitude, rendafam, peso, expected = TRUE,
         chisq = TRUE, plot = FALSE)
#    Conteúdo das células
# |-------------------------|
# |                Contagem |
# |       Valores esperados |
# |-------------------------|
#
# ================================================
#                       rendafam
# atitude               Baixa   Média    Alta   Total
# ------------------------------------------------
# Assistir no quarto      401     211      50     662
#                       329.1   240.4    92.5
# ------------------------------------------------
# Pegar cadeira            99      78      19     196
#                         97.4    71.2    27.4
# ------------------------------------------------
# Sentar no sofá          553     480     227    1260
#                       626.4   457.5   176.1
# ------------------------------------------------
# Total                  1053     769     296    2118
# ================================================
#
# Estatísticas para todos os fatores da tabela
#
# Pearson's Chi-squared test
```

```
# ------------------------------------------------------------
# Qui² = 66.51      g.l. = 4      p = 1.237e-13
#
#         Frequência esperada mínima: 27.39
```

A estatística p do teste de qui-quadrado indica a probabilidade da correlação encontrada ser um acaso de amostragem. No exemplo acima, o χ^2 foi de 66, o que para um grau de liberdade 4 implica numa probabilidade de $1{,}2 \times 10^{-13}$ de não haver nenhuma correlação entre as duas variáveis ou de a correlação ter o sentido contrário.

7.2 REGRESSÃO LINEAR

Para realizar uma análise de regressão linear no R, usamos a função `lm()`, que recebe como argumento obrigatório a equação de regressão no formato $y \sim x_1 + x_2 + x_3 + ... + x_n$, onde y é a variável dependente (necessariamente numérica) e cada variável x_i é independente ou explicativa (numérica ou categórica). No exemplo abaixo, é realizada uma análise de regressão tendo como variável dependente o logaritmo do número de votos recebidos pelos candidatos ao Senado em 2006, e como variável independente o logaritmo do valor gasto na campanha:

```
load("senado2006.RData")
sen$log.vgasto <- log(sen$vgasto)
sen$log.votos <- log(sen$votos)
modelo1 <- lm(sen$log.votos ~ sen$log.vgasto)
summary(modelo1)
#
# Call:
# lm(formula = sen$log.votos ~ sen$log.vgasto)
#
# Residuals:
#    Min     1Q Median     3Q    Max
# -4.696 -1.183  0.098  1.449  4.044
#
# Coefficients:
#                Estimate Std. Error t value Pr(>|t|)
# (Intercept)     -2.2955     0.9829   -2.34    0.021 *
# sen$log.vgasto   0.9486     0.0721   13.16   <2e-16 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
#
# Residual standard error: 1.85 on 200 degrees of freedom
# Multiple R-squared:  0.464,Adjusted R-squared:  0.461
# F-statistic:  173 on 1 and 200 DF,  p-value: <2e-16
```

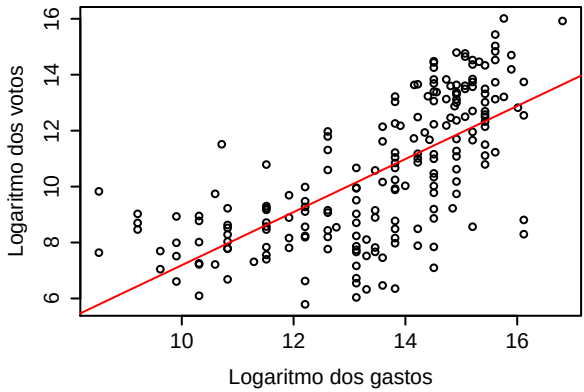
As variáveis são as mesmas utilizadas na seção 6.5, mas agora, com o sumário do modelo de regressão, temos algumas informações importantes para a interpretação dos resultados. A primeira coluna da tabela de coeficientes contém os nomes das variáveis. A segunda coluna contém a estimativa da contribuição de cada variável independente para o valor da variável dependente. O intercepto −2,30 indica que, teoricamente, se o logaritmo do gasto fosse zero (portanto, se o gasto fosse de R$ 1,00), o logaritmo do número de votos seria −2,30, o que obviamente é impossível, mas isso é apenas uma extrapolação dos dados (o menor valor do logaritmo dos gastos é 8,5). A estimativa da contribuição do logaritmo do gasto para o logaritmo dos votos é de 0,95, ou seja, para cada unidade a mais no logaritmo dos gastos, o logaritmo dos votos aumenta em 0,95. As colunas seguintes mostram, para cada variável independente, o erro padrão, a estatística t e a probabilidade de a contribuição da variável independente para o valor da variável dependente ser zero ou ter valor contrário ao indicado pelo coeficiente estimado. No exemplo acima, a probabilidade de o valor dos gastos não estar relacionado com o número de votos é de praticamente zero ($< 2 \times 10^{-16}$). A estatística R^2 indica qual proporção da variação da variável dependente pode ser explicada pelo modelo. No caso, 46% da variação do logaritmo dos votos pode ser explicada pelo logaritmo dos gastos.

Quando a análise de regressão é realizada com apenas duas variáveis, é possível criar um gráfico de dispersão para visualizar a correlação entre as variáveis e usar a função `abline()` para adicionar a linha de regressão ao gráfico, como no exemplo da Figura 7.1.[1]

```
plot(sen$log.votos ~ sen$log.vgasto, ylab = "Logaritmo dos votos",
     xlab = "Logaritmo dos gastos")
abline(modelo1, col = "red")
```

[1] Ver, na p. 105, uma explicação sobre o uso da função `abline()`.

FIGURA 7.1: Correlação entre duas variáveis numéricas



A grande vantagem da análise de regressão é a possibilidade de fazer análise com muitas variáveis independentes simultaneamente. Antes, porém, de fazer uma nova análise de regressão, vamos recodificar `escola` numa nova variável — `instr` (*nível de instrução*) — para reduzir seu número de categorias e renomear as categorias de `est.civil` para tornar os rótulos mais curtos:

```
library(memisc)
sen$instr <- recode(sen$escola,
    "Baixo" <- c("Lê e Escreve", "Ensino Fundamental incompleto",
                 "Ensino Fundamental completo",
                 "Ensino Médio incompleto", "Ensino Médio completo"),
    "Alto"  <- c("Ensino Superior incompleto",
                 "Ensino Superior completo"))
levels(sen$est.civil) <- c("Casado", "Divorciado", "Separado",
                           "Solteiro", "Viúvo")
```

Com o argumento *data = sen*, a função irá procurar no `data.frame` `sen` os objetos que não encontrar diretamente na área de trabalho. Com isso, torna-se dispensável anexar o banco de dados antes da análise com `attach()` ou digitar `sen$` repetidamente:

```
modelo2 <- lm(log.votos ~ log.vgasto + instr + idade + est.civil
              + sexo, data = sen)
summary(modelo2)
#
# Call:
```

```
# lm(formula = log.votos ~ log.vgasto + instr + idade + est.civil +
#     sexo, data = sen)
#
# Residuals:
#    Min     1Q Median     3Q    Max
#  -4.47  -1.21   0.10   1.35   3.97
#
# Coefficients:
#                     Estimate Std. Error t value Pr(>|t|)
# (Intercept)          -2.2850     1.1166   -2.05  0.04208 *
# log.vgasto            0.8284     0.0746   11.10  < 2e-16 ***
# instrAlto             1.0212     0.2953    3.46  0.00067 ***
# idade                 0.0351     0.0122    2.87  0.00455 **
# est.civilDivorciado  -0.1741     0.3731   -0.47  0.64132
# est.civilSeparado    -0.6436     0.4675   -1.38  0.17020
# est.civilSolteiro    -0.6775     0.3983   -1.70  0.09054 .
# est.civilViúvo       -1.2384     0.5962   -2.08  0.03911 *
# sexoMasculino        -0.9068     0.3625   -2.50  0.01320 *
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 1.74 on 193 degrees of freedom
# Multiple R-squared:  0.543,Adjusted R-squared:  0.524
# F-statistic: 28.7 on 8 and 193 DF,  p-value: <2e-16
```

No `modelo2`, algumas variáveis têm uma alta probabilidade de estar apenas por acaso correlacionadas com a variável dependente, como algumas das categorias da variável `estado civil`, que possuem valor $p > 0{,}05$. É recomendável que um modelo de regressão não inclua variáveis com correlações estatisticamente pouco significativas como essas. Ao construir um modelo de regressão, somente devem ser adicionadas variáveis cuja correlação com a variável dependente é elevada e, principalmente, teoricamente explicável. Uma forma de resolver a baixa significância estatística de algumas categorias da variável `est.civil` é agrupar categorias.

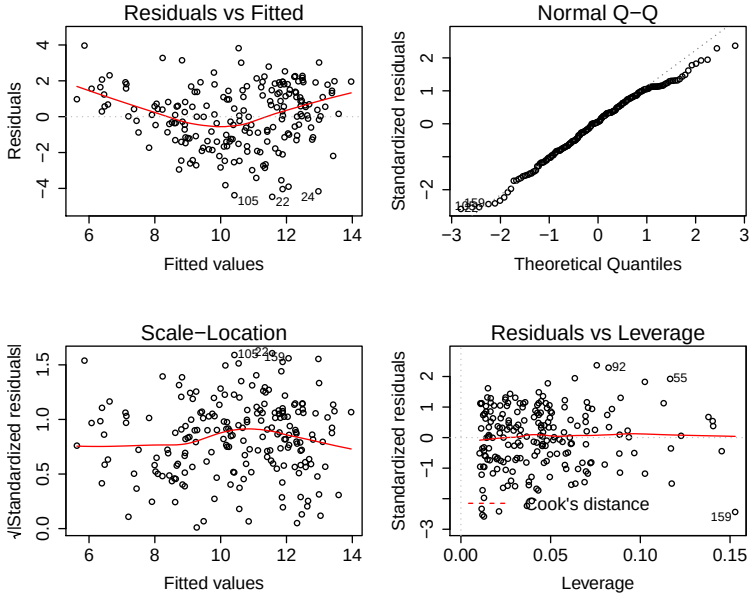
Para avaliar a adequação de um modelo de regressão linear, podem ser úteis as funções `predict()` (para obter os valores preditos pela equação de regressão) e `residuals()` (para obter os resíduos). O comando a seguir, por exemplo, permite comparar alguns dos valores da variável dependente com os valores preditos e os resíduos:

```
head(cbind(sen$log.votos, predict(modelo2), residuals(modelo2)))
#     [,1]   [,2]    [,3]
# 1 12.257  9.795  2.4612
# 2  9.937  9.810  0.1265
# 3 13.207 12.569  0.6386
# 4 13.742 13.584  0.1578
# 5 11.659 12.806 -1.1476
# 6 13.117 11.973  1.1435
```

Com a função `plot()` podemos produzir vários gráficos de diagnóstico do modelo. Na Figura 7.2, utilizamos o parâmetro gráfico *mfrow* para exibir quatro gráficos numa única figura.[2]

```
plot(modelo2)
```

FIGURA 7.2: Gráficos de diagnóstico de um modelo de regressão



[2]Ver Seção 8.4 para detalhes sobre o uso da função `par()` e digite `?plot.lm` para maiores informações sobre os gráficos.

```
par(mfrow = c(1, 1))
```

7.3 PROCEDIMENTO *step wise*

Uma forma automática de eliminar, do modelo, as variáveis pouco significativas é por meio da função `step()`, que repetidamente testa a inclusão e a retirada de variáveis do modelo, mantendo somente aquelas com maior contribuição para a sua significância estatística. O argumento *trace = 0* inibe a impressão na tela dos passos de remoção e inclusão de variáveis. Compare o sumário do último modelo de regressão da seção anterior com o do modelo obtido com o uso da função `step()`:

```
summary(step(modelo2, trace = 0))
#
# Call:
# lm(formula = log.votos ~ log.vgasto + instr + idade + sexo,
#    data = sen)
#
# Residuals:
#    Min     1Q Median     3Q    Max
# -4.628 -1.040  0.172  1.360  4.329
#
# Coefficients:
#               Estimate Std. Error t value Pr(>|t|)
# (Intercept)    -3.0875     1.0152   -3.04  0.00268 **
# log.vgasto      0.8600     0.0724   11.88  < 2e-16 ***
# instrAlto       1.0305     0.2963    3.48  0.00062 ***
# idade           0.0333     0.0118    2.81  0.00544 **
# sexoMasculino  -0.6355     0.3490   -1.82  0.07011 .
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 1.76 on 197 degrees of freedom
# Multiple R-squared:  0.526,Adjusted R-squared:  0.516
# F-statistic: 54.5 on 4 and 197 DF,  p-value: <2e-16
```

Neste caso, se houvesse interesse em tentar manter a variável *estado civil* no modelo, uma alternativa ao uso de stepwise seria a redução do número de categorias da variável `est.civil`. Podemos ver, pelo sumário do `modelo2`, antes do procedimento *step wise*, que, em relação

à categoria *Casado*, todas as demais categorias têm impacto negativo sobre a votação recebida pelo candidato. Vamos, portanto, criar uma nova variável, indicando simplesmente se o candidato é casado ou não e produzir um novo modelo de regressão:

```
sen$casado <- recode(sen$est.civil,
                "Não" <- c("Divorciado", "Solteiro", "Separado",
                           "Viúvo"),
                "Sim" <- "Casado")
modelo3 <- lm(log.votos ~ log.vgasto + instr + idade + casado + sexo,
              data = sen)
summary(modelo3)
#
# Call:
# lm(formula = log.votos ~ log.vgasto + instr + idade + casado +
#     sexo, data = sen)
#
# Residuals:
#    Min     1Q Median     3Q    Max
# -4.505 -1.216  0.105  1.329  4.019
#
# Coefficients:
#               Estimate Std. Error t value Pr(>|t|)
# (Intercept)    -2.9812     1.0072   -2.96  0.00346 **
# log.vgasto      0.8352     0.0727   11.49  < 2e-16 ***
# instrAlto       1.0522     0.2938    3.58  0.00043 ***
# idade           0.0330     0.0117    2.82  0.00536 **
# casadoSim       0.5538     0.2574    2.15  0.03265 *
# sexoMasculino  -0.7603     0.3506   -2.17  0.03134 *
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 1.74 on 196 degrees of freedom
# Multiple R-squared:  0.536,Adjusted R-squared:  0.525
# F-statistic: 45.4 on 5 and 196 DF,  p-value: <2e-16
```

7.4 REGRESSÃO LOGÍSTICA

Para realizar uma regressão logística com variável categórica que possua apenas duas categorias como dependente, usa-se a função `glm()` com o argumento *family = binomial(link = "logit")*, como no exemplo abaixo:

```
sen$eleito <- (sen$resultado == "Eleitos")
modelo4 <- glm(eleito ~ log.vgasto + idade, data = sen,
               family = binomial(link = "logit"))
summary(modelo4)
#
# Call:
# glm(formula = eleito ~ log.vgasto + idade,
#     family = binomial(link = "logit"),
#     data = sen)
#
# Deviance Residuals:
#     Min       1Q   Median       3Q      Max
# -1.5540  -0.5588  -0.2786  -0.0833   2.5112
#
# Coefficients:
#             Estimate Std. Error z value Pr(>|z|)
# (Intercept) -18.4673     4.0218   -4.59  4.4e-06 ***
# log.vgasto    0.9835     0.2598    3.79  0.00015 ***
# idade         0.0442     0.0207    2.14  0.03250 *
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# (Dispersion parameter for binomial family taken to be 1)
#
#     Null deviance: 158.89  on 201  degrees of freedom
# Residual deviance: 123.32  on 199  degrees of freedom
# AIC: 129.3
#
# Number of Fisher Scoring iterations: 6
```

Não é possível calcular a estatística R^2 para modelos de regressão logísticos, mas a função `LogRegR2()` do pacote `descr` calcula algumas das estatísticas comumente utilizadas em substituição ao R^2:

```
library(descr)
LogRegR2(modelo4)
# Qui²                 35.57
# Gl                   2
# Sig.                 1.887e-08
# Índice de Cox e Snell 0.1615
# Índice de Nagelkerke 0.2965
# R² de McFadden       0.2239
```

CAPÍTULO 8

GRÁFICOS

Neste capítulo, veremos como modificar o comportamento padrão do R ao criar gráficos e como adicionar novos elementos a gráficos já criados. Os gráficos são criados por funções específicas, muitas das quais já foram vistas nos capítulos anteriores.

Após a chamada à função principal, como `plot()`, `boxplot()` ou `hist()`, o dispositivo do gráfico continua aberto para receber novos elementos, até que seja feita uma nova chamada a uma dessas funções ou que o dispositivo seja explicitamente fechado com a função `dev.off()`. Os três procedimentos básicos para produzir gráficos com características especiais são: (a) modificar parâmetros globais seguidos pelo R na criação de gráficos *antes* da criação do gráfico, (b) especificar valores de argumentos da função que irá gerar o gráfico e (c) acrescentar novos elementos a gráficos *já criados*. Na prática, o procedimento para produzir gráficos consiste em:

1. Chamar a função que irá produzir o gráfico sem especificar muitos argumentos.
2. Se o gráfico não ficar com qualidade satisfatória para ser incluído num relatório de pesquisa, executar novamente a função, agora especificando os valores de um maior número de argumentos.
3. Se isso ainda não for suficiente para atingir o objetivo pretendido, modificar os parâmetros globais da produção dos gráficos e executar a função novamente.

4. Adicionar elementos ao gráfico (texto, legenda, pontos, linhas, desenhos etc.).

5. Repetir os passos 2, 3 e 4 até obter o resultado desejado.

Formas específicas de implementar esses procedimentos serão apresentadas nas próximas seções.

A função `plot()` é genérica, ou seja, possui vários métodos para objetos de classes diferentes. Isso significa que, ao chamarmos a função `plot()`, ela verifica qual a classe do objeto que lhe foi passado como argumento e chama o método apropriado. Consequentemente, para descobrir os argumentos que podem ser adicionados à função `plot()` é preciso ler não apenas a ajuda da própria função, mas também a ajuda do método `plot` para o objeto que está sendo plotado. Por exemplo, a função `freq()`, do pacote `descr`, cria um objeto de classe `freqtable`, e o pacote possui um método `plot` para objetos dessa classe. Assim, para obter informações sobre esse método, deve-se digitar `help(plot.freqtable)`. Por outro lado, a função `compmeans()`, embora também crie um gráfico, não possui um método `plot` próprio porque não acrescenta nenhum argumento especial à função `boxplot()`, que é chamada internamente. Assim, para conhecer os argumentos específicos do gráfico criado por `compmeans()`, é preciso consultar a documentação da função `boxplot()`. Para encontrar todas as opções de configuração relacionadas à produção de gráficos no `R` é preciso consultar a documentação da função que estiver sendo utilizada e de várias outras, entre elas:

```
?plot
?plot.default
?par
```

8.1 TÍTULO, SUBTÍTULO E RÓTULOS

Os argumentos mais comuns para a inclusão de títulos e rótulos nos gráficos já foram utilizados nos capítulos anteriores. Algumas funções e alguns métodos da função `plot()` aceitam os argumentos diretamente. Em outros casos, precisamos usar a função `title()` para

adicionar os títulos e rótulos após produzir o conteúdo principal do gráfico. Os principais argumentos para controlar títulos e rótulos são: *main* (título principal), *sub* (subtítulo), *xlab* (rótulos do eixo x) e *ylab* (rótulos do eixo y).

Os argumentos acima, com ou sem a função `title()`, foram utilizados nos capítulos anteriores e não há necessidade de apresentar novos exemplos (veja no Índice Remissivo as páginas em que ocorrem as palavras-chaves `title`, `plot`, `boxplot`, `freq`, `crosstab`, `mosaicplot` etc.).

8.2 CORES

Existem várias formas de fazer referência a cores no R. Até agora utilizamos nomes, como "red", "green", "yellow", e uma lista completa dos nomes de cores conhecidos pelo R pode ser obtida com a função `colors()`. Podemos também criar cores personalizadas usando a função `rgb()`, que recebe como argumentos as quantidades de vermelho (*red*), verde (*green*) e azul (*blue*) e, opcionalmente, o grau de opacidade (*alpha*). Os valores devem ser números reais entre 0 e 1. Também é possível especificar vetores e, nesse caso, acrescentar um argumento com os nomes das cores resultantes. Exemplos:

```
goiaba <- rgb(0.94, 0.41, 0.40)
goiaba.semitrans <- rgb(0.94, 0.41, 0.40, alpha = 0.5)
vitamina <- rgb(red = c(0.87, 0.70), green = c(0.83, 0.77),
                blue = c(0.71, 0.30), names = c("leite", "abacate"))
```

Outra forma de especificar cores é utilizando o código `HTML` correspondente, ou seja, o símbolo # seguido dos valores vermelho, verde, azul e (opcionalmente) opacidade, em notação hexadecimal, de 00 a FF. Podemos utilizar outros programas, como `gcolor2`, `gimp` ou `kcolorchooser`, para criar ou capturar cores e obter seu código `HTML`. No exemplo abaixo, criamos mais uma cor e reunimos todas as cores criadas até agora num único vetor:

```
uva <- "#AD4DA3"
salada <- c(vitamina, uva, goiaba, goiaba.semitrans)
```

Uma terceira forma de fazer referência a uma cor é pelo seu índice na palheta de cores do R, cujos valores podem ser obtidos chamando-se a função `palette()` sem nenhum argumento. Por padrão, a palheta possui oito cores, mas podemos modificá-la chamando a função `palette()` com um novo vetor de cores. Para voltar à palheta padrão, passamos o argumento `"default"` para a função:

```
palette()
# [1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta"
# [7] "yellow"  "gray"

palette(salada)
palette()
# [1] "#DED4B5"   "#B3C44D"   "#AD4DA3"   "#F06966"   "#F0696680"

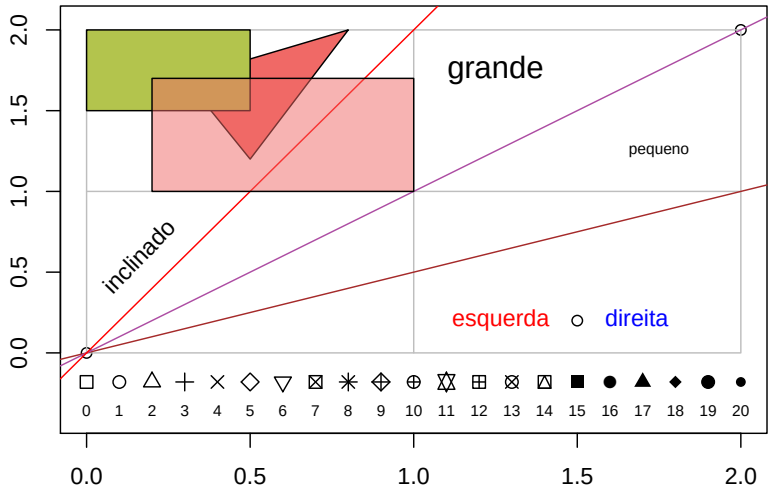
palette("default")
palette()
# [1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta"
# [7] "yellow"  "gray"
```

Nas figuras 11.1 e 11.2 (p. 136 e 138) modificamos as cores da palheta para colorir o gráfico com os `levels` de variáveis categóricas.

8.3 ADICIONAR PONTOS, LINHAS, POLÍGONOS, TEXTOS

Na maioria das vezes, o comportamento padrão das funções gráficas do R produz resultado satisfatório, mas, em algumas circunstâncias, podemos precisar adicionar mais elementos aos gráficos, como textos, pontos ou desenhos. Nesta seção, a inclusão desses elementos em gráficos será exemplificada pela produção do gráfico da Figura 8.1, cuja montagem será explicada passo a passo.

FIGURA 8.1: Ferramentas de desenho



O primeiro passo será criar um gráfico com bastante espaço vazio, o que possibilitará a adição de elementos sem sobreposição. Vamos produzir um gráfico de dispersão com apenas dois pontos, situados nas coordenadas (0, 0) e (2, 2). Aumentaremos o limite inferior do eixo y, que normalmente seria apenas um pouco além dos limites de y. Com isso, teremos um espaço vazio que utilizaremos para acrescentar pontos e texto:

```
plot(c(0, 2), c(0, 2), ylim = c(-0.4, 2.05), xlab = "", ylab = "")
```

O R possui 21 caracteres diferentes — numerados de 0 a 20 — que podem ser usados para plotar pontos em gráficos. A escolha do ponto é feita pela escolha de um valor numérico para o argumento *pch*. Com o código abaixo, acrescentamos os 21 pontos acompanhados de 21 rótulos, correspondentes aos números dos pontos. Usamos as funções `seq()` e `rep()` para produzir, respectivamente, as coordenadas `x` e `y` dos pontos e dos textos. Os pontos, na verdade, são caracteres, e foram escalonados para ficarem 20% maiores do que o normal (*cex = 1.2*). Os rótulos, argumento *labels* da função `text()`, por sua vez, foram reduzidos para 70% do tamanho normal.

```
points(seq(0, 2, 0.1), rep(-0.18, 21), cex = 1.2, pch = 0:20)
text(seq(0, 2, 0.1), rep(-0.36, 21), cex = 0.7, labels = 0:20)
```

Normalmente, pontos e textos são posicionados no centro das coordenadas fornecidas, mas, com o argumento *pos*, podemos definir que o texto ficará abaixo, à esquerda, acima ou à direita das coordenadas (cujos valores correspondentes são, respectivamente, 1, 2, 3 e 4). Isso é útil se precisamos alinhar uma sequência de rótulos. No código abaixo, os textos "esquerda" e "direita" possuem as mesmas coordenadas, mas alinhamentos contrários. O argumento *offset* define quantos caracteres o texto ficará distante das suas coordenadas, no caso de *pos* ter sido utilizado.

```
points(1.5, 0.2)
text(1.5, 0.2, labels = c("esquerda", "direita"), pos = c(2, 4),
     offset = 1, col = c("red", "blue"))
```

Os textos normalmente são plotados na posição horizontal, mas podemos definir que eles fiquem inclinados com o argumento *srt*, cujo valor deve ser fornecido em graus:

```
text(0.16, 0.6, labels = "inclinado", srt = 46)
```

Como já exemplificado, o argumento *cex* modifica o tamanho da letra. Isso fica mais claro com os textos "pequeno" e "grande" adicionados ao gráfico:

```
text(c(1.75, 1.25), c(1.25, 1.75), labels = c("pequeno", "grande"),
     cex = c(0.7, 1.3))
```

Para adicionar linhas, usamos a função `lines()`, que recebe como argumentos obrigatórios vetores com as coordenadas `x` e `y`. Neste exemplo, usamos as linhas para produzir uma grade cinza:

```
lines(c(0, 0, 2, 2, 0), c(0, 2, 2, 0, 0), col = "gray")
lines(c(1, 1), c(0, 2), col = "gray")
lines(c(0, 2), c(1, 1), col = "gray")
```

É possível, ainda, com a função `abline()`, adicionar linhas que correspondem a uma função de 1º grau. Os dois argumentos obrigatórios são a coordenada y, quando `x = 0`, e a inclinação da linha (o acréscimo em y para cada aumento de uma unidade em `x`):[1]

```
abline(0, 0.5, col = "brown")
abline(0, 1, col = uva)
abline(0, 2, col = "red")
```

Podemos, por fim, desenhar polígonos. No exemplo abaixo, são desenhados dois retângulos e um triângulo. Note que o último retângulo desenhado se sobrepõe parcialmente sobre os demais polígonos, mas tem cor semitransparente:

```
polygon(c(0.3, 0.8, 0.5), c(1.7, 2.0, 1.2), col = goiaba)
rect(0, 1.5, 0.5, 2.0, col = vitamina["abacate"])
rect(0.2, 1.0, 1.0, 1.7, col = goiaba.semitrans)
```

8.4 PARÂMETROS GLOBAIS

Os gráficos do R obedecem a certos parâmetros globais que podem ser modificados por meio da função `par()`. A alteração do valor de qualquer um desses parâmetros afetará todos os gráficos produzidos até que o parâmetro seja modificado por nova chamada a `par()`, ou que o dispositivo gráfico seja fechado com `dev.off()`. A ajuda da função lista e explica todos os parâmetros, mas, a seguir, apresento os mais comumente utilizados:

- *bg*: cor de fundo do gráfico.
- *cex*: valor para redimensionamento de caracteres (padrão = 1.0).
- *cex.axis*: valor para redimensionamento do texto dos eixos.
- *cex.lab*: valor para redimensionamento dos rótulos dos eixos.

[1] Veja a ajuda da função para formas alternativas de especificar a linha a ser plotada. Um uso comum é a adição da linha de regressão a um gráfico de dispersão entre duas variáveis numéricas.

- *cex.main*: valor para redimensionamento do título principal.
- *cex.sub*: valor para redimensionamento do subtítulo.
- *col*: vetor das cores a serem utilizadas nos gráficos.
- *col.axis*: cor das anotações nos eixos.
- *col.lab*: cor dos rótulos dos eixos.
- *col.main*: cor do título principal.
- *col.sub*: cor do subtítulo.
- *fg*: cor de frente do gráfico.
- *las*: orientação dos rótulos dos eixos `x` e `y` (0 = rótulos sempre paralelos aos eixos, 1 = sempre horizontais, 2 = sempre perpendiculares aos eixos, 3 = sempre verticais).
- *mai*: vetor de números especificando as margens do gráfico em polegadas (baixo, esquerda, topo, direita).
- *mar*: vetor de números especificando as margens do gráfico em linhas (baixo, esquerda, topo, direita).
- *mfcol, mfrow*: vetor na forma `c(nlinhas, ncolunas)` especificando quantos gráficos serão desenhados por dispositivo.
- *new*: define se o próximo gráfico deve considerar que está sendo produzido num dispositivo novo e, portanto, não precisa limpar o dispositivo antes de ser desenhado (padrão = *FALSE*); se o valor for *TRUE*, o gráfico anterior não será apagado.
- *pch*: tipo de ponto a ser desenhado, especificado por um número inteiro ou por um caractere.
- *srt*: rotação de textos em graus.

Para consultar o valor de um parâmetro, chamamos a função `par()` usando o nome do parâmetro como argumento:

```
par("mar")
# [1] 5.1 4.1 4.1 2.1
```

Para modificar um parâmetro, digitamos um comando na forma: `par(nome.do.parâmetro = valor)`, mas, antes de modificar parâmetros gráficos, convém salvar os valores atuais para facilitar a sua restauração após a produção do gráfico que precisa de parâmetros especiais. Para isso, chamamos `par()` com o argumento *no.readonly = TRUE*:

```
antigopar <- par(no.readonly = TRUE)
par(mar = c(5.1, 4.1, 4.1, 2.1))
```

Algumas funções repassam para `par()` parâmetros gráficos passados para elas como argumentos, mas outras não. Assim, em alguns casos, não é necessário modificar os parâmetros globalmente apenas para alterar o resultado de um gráfico, mas em outros isso é necessário. Além disso, nem todos os parâmetros podem ser modificados dessa forma; alguns somente podem ser alterados pela função `par()`. Para sabermos se uma função é ou não capaz de alterar temporariamente os parâmetros globais da produção de gráficos, temos duas alternativas: ler a documentação de `par()` e da função que produz o gráfico ou usar o método da tentativa e erro (este é, geralmente, o procedimento mais rápido).

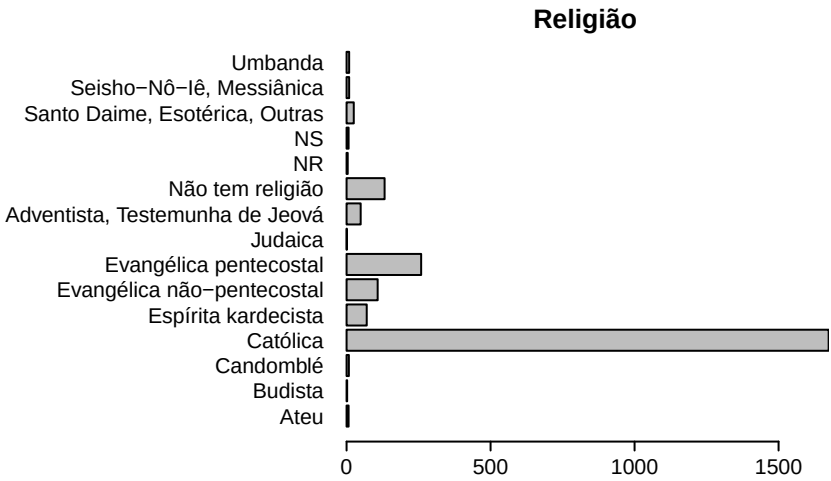
A seguir, um exemplo de gráfico produzido após manipulação dos parâmetros. A necessidade de manipular os parâmetros é decorrente do fato de algumas das categorias da variável *religião*, da PESB, terem rótulos demasiadamente longos. Antes de fazer os gráficos, vamos carregar o banco de dados e criar uma cópia da variável que nos interessa:

```
load("pesb2002.RData")
relig <- pesb$q511
```

Agora, vamos fazer o gráfico de duas formas diferentes. Na primeira, mais simples, vamos apenas fazer um gráfico de barras horizontais. Os ajustes necessários nos parâmetros serão a largura da margem esquerda e o tamanho da fonte utilizada nas anotações do eixo `y`. O resultado está na Figura 8.2.

```
par(mar = c(2.1, 18.8, 2.1, 0.1), cex = 0.7)
plot(relig, main = "Religião", las = 1, horiz = TRUE)
```

FIGURA 8.2: Gráfico de barras de variável com rótulos longos

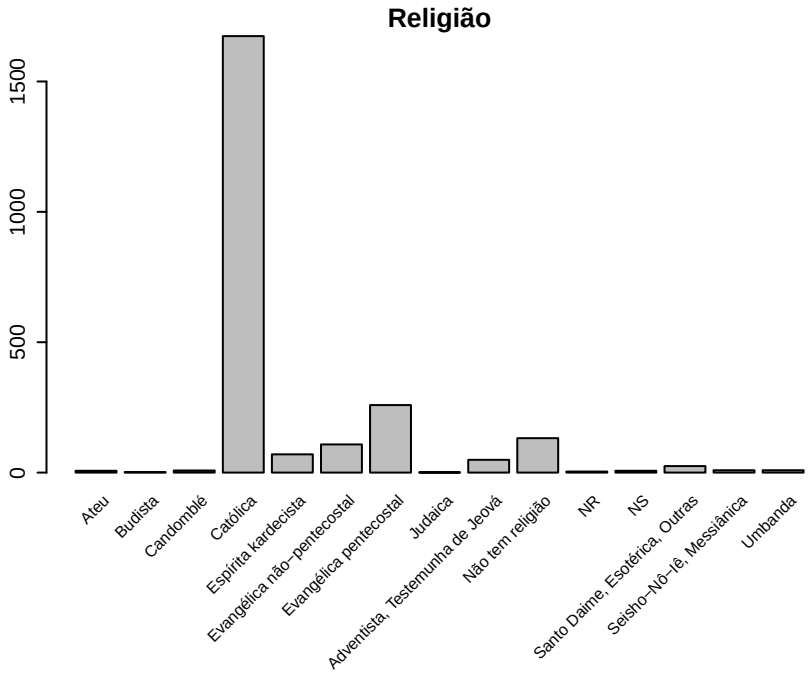


O objetivo, na segunda versão do gráfico, é manter as barras verticais. A solução mais simples seria aumentar a margem inferior e, analogamente ao que fizemos no gráfico anterior, posicionar, perpendicularmente ao eixo, as anotações com os rótulos das categorias da variável. Mas, para facilitar a leitura dos rótulos, resolvemos posicioná-los diagonalmente, como na Figura 8.3. Para obter o resultado esperado, além de redimensionar as margens, produzimos o gráfico sem os nomes das barras (*names = FALSE*). Depois, usamos a função `text()` para acrescentar as anotações do eixo `x` com a fonte reduzida (*cex = 0.7*), e com vários outros argumentos para obter o efeito desejado. Um dos problemas que precisamos resolver foi o posicionamento de cada rótulo. O valor retornado pela função `plot()`, nesse caso, é o vetor com o posicionamento no eixo `x` da extremidade esquerda das barras. Assim, utilizamos o resultado da função para posicionar os rótulos da barra (somando 0,5 para compensar a largura das barras). Todos os rótulos têm a mesma coordenada no eixo `y`, -100 (lembre-se que o R recicla um vetor quantas vezes for necessário; nesse caso, o valor -100 será repetido 15 vezes). Normalmente, a função `text()` é utilizada apenas para acrescentar texto na região interna dos gráficos, e qualquer texto

que exceda essa região é truncado. Para que isso não ocorresse, usamos o argumento *xpd = TRUE*. Também usamos os argumentos *pos*, para alinhar todos os rótulos à direita, e *srt*, para inclinar os rótulos.

```
par(mar = c(10.8, 2.1, 2.1, 0.1), cex = 0.7)
rotulos <- levels(relig)
bx <- plot(relig, main = "Religião", names = FALSE) + 0.5
text(bx, -100, labels = rotulos, pos = 2, cex = 0.7,
     srt = 45, xpd = TRUE)
```

FIGURA 8.3: Gráfico de barras de variável com rótulos longos (II)



Concluída a produção do gráfico, pode ser conveniente reconfigurar os parâmetros para os valores iniciais:

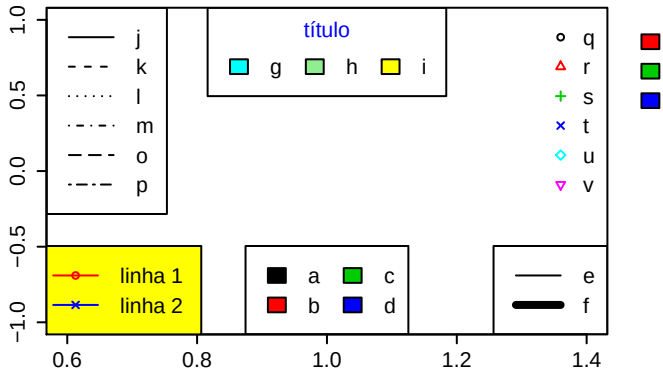
```
par(antigopar)
```

8.5 LEGENDAS

Para acrescentar uma legenda a um gráfico, utilizamos a função `legend()`. A Figura 8.4 ilustra como colocar legendas num gráfico, controlando os parâmetros: posicionamento da legenda (por meio de palavra-chave ou pelas coordenadas *x* e *y*), cor das linhas ou dos símbolos, (*col*), tipo de linha (*lty*), tipo de símbolo (*pch*), cor de preenchimento dos quadrados (*fill*), largura das linhas (*lwd*), presença de título (*title*), cor do título (*title.col*), presença de caixa em torno da legenda (*bty*), cor de fundo da legenda (*bg*) e divisão da legenda em colunas (*ncol*).

```
par(mar = c(2.1, 2.1, 1.1, 4.1))
plot(0, xlab = "", ylab = "", col = "white")
legend("bottom", legend = c("a", "b", "c", "d"), fill = 1:4, ncol = 2)
legend("bottomleft", legend = c("linha 1", "linha 2"), col = c(2, 4),
       lty = 1, pch = c(1, 4), bg = "yellow")
legend("bottomright", legend = c("e", "f"), lwd = c(1, 4), lty = 1)
legend("top",  legend = c("g", "h", "i"),
       title = "título", title.col = "blue", horiz = TRUE,
       fill = c("cyan", "lightgreen", "yellow"))
legend("topleft", legend = c("j", "k", "l", "m", "o", "p"), lty = 1:6)
legend("topright", legend = c("q", "r", "s", "t", "u", "v"), col = 1:6,
       pch = 1:6, bty= "n")
legend(1.45, 1.05, legend = c("x", "y", "z"), fill = 2:5, bty = "n",
       xpd = TRUE)
```

FIGURA 8.4: Exemplos de legendas



CAPÍTULO 9

PRODUÇÃO DE RELATÓRIOS

9.1 RESULTADO EM TEXTO PLANO

A forma mais simples de relatório é obtida pela execução do script utilizado para realizar as análises dos dados de forma a produzir um documento com a cópia do código intercalada com os resultados da execução. No Linux, um documento desse tipo pode ser obtido executando-se o seguinte comando num terminal:

```
R CMD BATCH script.R
```

onde `script.R` é o script contendo os comandos necessários para a análise. O comando criará o arquivo `script.Rout`, contendo uma cópia dos comandos executados precedidos por > e seguidos pelos resultados tal como teriam sido impressos na tela numa seção interativa do R. Os gráficos que seriam exibidos numa seção interativa são salvos no arquivo `Rplots.pdf`.

Antes de iniciarmos os procedimentos de produção de relatórios, vamos carregar o banco de dados sobre os candidatos ao Senado em 2006 que será utilizado durante o capítulo:

```
load("senado2006.RData")
```

Em qualquer sistema operacional, é possível salvar os resultados dos comandos num arquivo usando a função `sink()`. Os resultados não serão acompanhados dos comandos, mas pode-se acrescentar comentários usando a função `cat()`, como no exemplo a seguir:

```
sink("~/analise2006.txt")
cat("Tabela cruzada: Sexo dos candidatos X Resultado da eleição\n")
table(sen$sexo, sen$resultado)
sink()
```

Após o comando `sink("~/analise2006.txt")`, os resultados, ao invés de impressos na tela, serão redirecionados para o arquivo `analise2006.txt`, que será criado na pasta `~/`.[1] O redirecionamento dos resultados é finalizado com a execução da função `sink()`, sem nenhum argumento. No exemplo acima, o conteúdo do arquivo seria:

```
Tabela cruzada: Sexo dos candidatos X Resultado da eleição
           resultado
sexo        Eleito Não eleito
  Feminino       4         26
  Masculino     23        149
```

Observe que o título que demos à tabela não inclui os caracteres "`\n`". Isso ocorre porque o símbolo `\` em strings torna especial o significado de alguns caracteres. No caso, `n` passa a significar *nova linha*. Sem a inclusão de "`\n`", a primeira linha do resultado do comando `table(sexo, resultado)` teria sido impressa ao lado do título da tabela.

Um relatório em texto plano, criado por `R CMD BATCH` ou pela função `sink()`, é satisfatório apenas como um relatório preliminar para o próprio pesquisador. Para apresentar os resultados da pesquisa numa publicação, impressa ou eletrônica, é preciso utilizar outros recursos, explorados nas seções seguintes.

[1] O significado de `~/` foi explicado na seção 2.4, tabelas 2.1 e 2.2.

9.2 MAPA DE BITS *versus* GRÁFICO VETORIAL

Para apresentar os resultados numa publicação, basicamente é necessário inserir figuras e tabelas no texto. Nesta seção, veremos como inserir figuras em documentos de processadores de texto.

Existem dois tipos básicos de figuras: mapas de bits e gráficos vetoriais escalonáveis. Um mapa de bits é um espaço retangular dividido em células; cada célula pode ter uma cor e grau de transparência específicos. Normalmente, as células têm tamanhos suficientemente pequenos para serem invisíveis isoladamente a olho nu, mas, quando sofrem ampliação, se tornam visíveis e a imagem fica com um aspecto grosseiro, como ilustrado pela Figura 9.1.

FIGURA 9.1: Exemplo de Bitmap em tamanho natural e ampliado

Figuras na forma de mapas de bits ocupam muito espaço em disco e, por isso, costumam ser comprimidas por meio de diferentes algoritmos. Alguns desses algoritmos causam perda de informação; outros não.

Geralmente, a melhor alternativa é o uso de gráficos vetoriais, que contêm informações sobre as coordenadas nas quais devem ser traçados linhas, pontos, bem como a espessura das linhas e dos pontos, e informações sobre em quais coordenadas escrever textos e com qual fonte. Cabe ao programa que visualiza o gráfico desenhá-lo. Por ser redesenhada a cada ampliação ou redução, a figura se mantém sempre com qualidade em qualquer nível de zoom e com qualquer resolução de impressão. A única desvantagem de um gráfico vetorial é quando a figura inclui milhares de pontos, linhas ou textos superpostos. Embora os pontos, linhas e textos das camadas inferiores não estejam visíveis no resultado final, a informação sobre eles estará presente no arquivo,

podendo torná-lo, em alguns casos, absurdamente grande e lento de desenhar. Nesses casos, é preferível usar um gráfico no formato bitmap.

Segue uma lista dos formatos mais comumente utilizados dentre os que o R pode produzir:

- `svg`: Formato vetorial. O LibreOffice consegue incluir imagens nesse formato e esta geralmente será a melhor opção. O arquivo `svg` se tornará muito grande se o gráfico contiver muitos milhares de pontos e, neste caso, poderá ser preferível salvá-lo no formato `png`.
- `eps`: Formato vetorial. Pode ser a melhor opção para inserção em documento do Word.
- `png`: Formato bitmap com compressão, mas sem perda de qualidade com a compressão. É o formato mais adequado quando, por algum motivo, não é possível usar um formato vetorial.
- `jpg`: Formato bitmap com compressão e com perda de qualidade devido à compressão. É o formato ideal para imagens complexas, como fotos. Geralmente é inadequado para gráficos estatísticos.
- `pdf`: Formato vetorial. Pode ser inserido em documentos produzidos com LaTeX.

9.3 INSERÇÃO DE GRÁFICOS EM RELATÓRIOS

Como já comentado, para inserir gráficos com qualidade satisfatória no LibreOffice Writer, existem muitas opções. Uma delas é salvar o gráfico no formato `svg`. O primeiro argumento é o nome do arquivo que conterá a figura. A largura (*width*) e a altura (*height*) do gráfico são expressas em polegadas. O gráfico somente é efetivamente gravado em disco após o comando `dev.off()` (resultados omitidos):

```
library(descr)
svg("civilXres.svg", height = 4, width = 6, onefile = FALSE)
par(xpd = TRUE)
crosstab(sen$resultado, sen$est.civil, las = 1, ylab = "Estado civil",
         xlab = "Resultado")
dev.off()
```

Outra opção é salvar o gráfico no formato `png`, escolhendo uma resolução alta (no nosso exemplo, *res = 600*), o que garantirá boa qualidade de impressão. Normalmente, a altura e a largura de figuras `png` são expressas em pixels, mas podemos expressá-las em centímetros se usarmos o argumento *units = cm* (neste caso, o uso de *res* é obrigatório). No exemplo abaixo, o R calculará o tamanho em pixels da figura que, impressa nas dimensões 10x15 cm, terá 600 pontos por polegada (resultados omitidos):

```
png("civilXres.png", height = 10, width = 15, units = "cm", res = 600)
par(xpd = TRUE)
crosstab(sen$resultado, sen$est.civil, las = 1,
         ylab = "Estado civil", xlab = "Resultado")
dev.off()
```

Para inserir a figura num documento de texto do LibreOffice Writer, deve-se clicar no menu *Inserir / Figura / De um arquivo...* A informação sobre a resolução da figura (pontos por polegada) é armazenada no arquivo `png` e, no momento da inserção, o Writer reconhecerá o tamanho correto, não sendo necessário fazer manualmente qualquer redimensionamento.

Se a figura se destinar a uma página de internet, não deveremos usar os argumentos *units* e *res*, e deveremos informar o tamanho desejado da figura diretamente em pixels (resultados omitidos):

```
png("civilXres2.png", height = 500, width = 700)
crosstab(sen$resultado, sen$est.civil, las = 1,
         ylab = "Estado civil", xlab = "Resultado")
dev.off()
```

9.4 R E MARKDOWN

Uma forma prática de produzir relatórios de pesquisa com qualidade satisfatória é pelo uso de arquivos do tipo `Rmd`, que combinam as linguagens *R* e *Markdown*. Por serem arquivos de texto plano, podem ser editados pelos mesmos editores utilizados para elaborar código do R, incluindo a possibilidade de enviar código para o console, permitindo testá-lo. Isso torna a produção do relatório dinâmica, simultânea

à realização da análise dos dados. O código do R é usado para gerar figuras, tabelas e outros elementos do relatório. O código em linguagem *Markdown* é convertido em *HTML* e código *HTML* existente no corpo do texto será preservado.

O código do R deve ser precedido por uma linha no formato

```
```{r apelidodocodigo, opções}
```

e seguido por uma linha contendo apenas os símbolos ```, como no exemplo abaixo:

```
```{r exemplo, echo=TRUE}
x <- 1:10
x
```
```

No documento processado, o trecho acima desapareceria, e em seu lugar teríamos:

```
x <- 1:10
x
#  [1]  1  2  3  4  5  6  7  8  9 10
```

Na lista abaixo estão as principais opções para processamento do código, com os valores aceitos entre parênteses, sendo o primeiro deles o valor padrão:[2]

- `echo` (*TRUE*, *FALSE*): Define se os comandos devem ser impressos ou não.
- `results` (*'markup'*, *'asis'* , *'hide'*): Define como devem ser apresentados os resultados produzidos, equivalentes ao que seria impresso no console do R. Com a opção *'hide'*, nenhum resultado é incluído no documento final, e com a opção *'markup'* a função `knit()` inclui o resultado diretamente no documento `tex` resultante, mas com o cuidado de adicionar um ambiente

[2] A lista completa de opções está disponível em http://yihui.name/knitr.
```

que torna o resultado semelhante ao que vemos no console do R. Com a opção *'asis'*, o resultado bruto é inserido no documento sem nenhum cuidado especial, ou seja, o próprio resultado já deve estar na linguagem *Markdown* ou *HTML*.

- `fig.width` e `fig.height`: Referem-se, respectivamente, à largura e à altura em polegadas da figura que será produzida pelo trecho de código.
- `dpi` (*72*): Indica o número de pontos por polegada das figuras. O valor padrão é adequado para exibição em tela, mas para impressão é recomendável uma resolução maior.

O código abaixo inclui as opções *echo = FALSE* e *results = 'hide'* e, consequentemente, não incluirá nada no documento final. Entretanto os valores de `n` e `x` serão computados e poderão ser usados nos trechos seguintes do documento:

````
```{r exemplo, echo=FALSE, results='hide'}
n <- 1:100
x <- mean(n)
```
````

Em qualquer lugar do texto, podemos acrescentar o código `` `r x` ``, onde `x` é algum código válido do R que resulte na impressão de algum texto. No nosso exemplo, o resultado seria: 50.5. O arquivo `exemploRMarkdown.Rmd` contém exemplos de inserção de figura, de tabela e de uso de `` `r x` `` no corpo do texto.

A função `knit2html()` converte um arquivo de extensão `Rmd` em `html`. O arquivo `html` pode ser aberto num processador de texto como o *Libre Office* ou o *Microsoft Word*. No Linux, é possível converter o documento `html` em `odt` se o *Libre Office* estiver instalado. Caso contrário, para ter maior liberdade de edição do conteúdo, pode ser necessário selecionar todo o conteúdo e colar em novo documento. Se a figura ficar com tamanho incorreto, pode-se dar um duplo clique sobre ela e clicar no botão *Tamanho original* da caixa de diálogo das propriedades da imagem. Dica: no *Libre Office*, clique em *Arquivo / Recarregar* para

atualizar a visualização do documento sempre que fizer alterações no arquivo `Rmd` e gerar um novo `odt`.

```
library(knitr)
knit2html("exemploRMarkdown.Rmd", options = "")
# O comando seguinte funciona no Linux com Libre Office instalado,
# e não funciona se o Libre Office estiver aberto:
system("soffice --invisible --convert-to odt exemploRMarkdown.html")
```

Leia atentamente o arquivo `exemploRMarkdown.Rmd`, compare com o resultado produzido e experimente fazer alterações para se acostumar com os comandos.

9.5 R E LaTeX

A forma de produção de relatórios mais utilizada por usuários avançados do R é por meio de scripts combinando código LaTeX com código R em arquivos que, usualmente, recebem extensão `Rnw`. Usamos o R para converter o arquivo `Rnw` num documento contendo apenas código LaTeX. O documento LaTeX pode, então, ser convertido em `pdf`, finalizando, assim, a produção do relatório.

Além da facilidade de integração como R, o uso de LaTeX tem a vantagem de automatizar o processo de referências cruzadas e referências bibliográficas. A numeração de capítulos, seções, tabelas e figuras é automática, e a produção da bibliografia final, de acordo com as normas da ABNT, também é automatizada: quando alguma referência é feita a uma obra, a referência bibliográfica completa é automaticamente adicionada à bibliografia final e, se for apagado o último trecho do texto contendo referência a uma obra, a referência na bibliografia final é eliminada. Entretanto, foge do escopo deste livro ensinar o uso de LaTeX. Nos próximos parágrafos, serão apresentados apenas os aspectos mais importantes da integração entre LaTeX e R.

Em documentos `Rnw`, o código do R deve ser precedido por uma linha no formato

```
<<apelidodocodigo,opções>>=
```

e seguido por uma linha contendo o símbolo `@`, como no exemplo abaixo:

```
<<exemplo>>=
x <- 1:10
x
@
```

Existem pelos menos duas formas de processar documentos desse tipo para a produção de um relatório no formato `pdf`: usando a função `Sweave()`, que faz parte da distribuição básica do R, e usando a função `knit()`, do pacote `knitr`. A função `knit()` foi desenvolvida mais recentemente e tem inúmeras vantagens em relação à `Sweave()`. Por isso, o arquivo `exemploRnoweb.Rnw` foi preparado para ser processado pela função `knit()`. Em documentos `Rnw`, a inserção de código do R no corpo do texto é feita com o código `\Sexpr{x}`, onde `x` é algum código válido do R que resulta na impressão de algum texto. Assim como o arquivo `exemploRMarkdown.Rmd`, o arquivo `exemploRnoweb.Rnw` contém exemplos de inserção de figura, de tabela e de uso de `\Sexpr{}`.

Para processar o documento, o procedimento é:

```
library(knitr)
knit2pdf("exemploRnoweb.Rnw")
```

Capítulo 10

Tópicos em programação

10.1 Manipulação de texto

A função `paste()` é usada para concatenar strings. Os elementos fornecidos como argumentos serão convertidos em `character` se necessário:

```
load("senado2006.RData")
paste("O banco de dados tem", nrow(sen), "linhas.")
# [1] "O banco de dados tem 202 linhas."
```

Por padrão, a função `paste()` concatena diferentes strings usando um espaço em branco como separador. Se não quisermos nenhum espaço entre as strings, tempos duas opções: definir o valor do argumento *sep* como `""` ou usar a função `paste0()`:

```
paste0("Mais homens do que mulheres se candidataram ao Senado (",
       sum(sen$sexo == "Masculino"), "x",
       sum(sen$sexo == "Feminino"), ").")
# [1] "Mais homens do que mulheres se candidataram ao Senado (172x30)."
```

Se os argumentos fornecidos a `paste()` forem vetores, o resultado será um vetor de strings, mas podemos usar o argumento *collapse* para concatenar essas strings:

```
numeros <- c(13, 45, 43)
candidatos <- c("Dilma", "Serra", "Marina")
paste(candidatos, numeros)
# [1] "Dilma 13"  "Serra 45"  "Marina 43"

paste(candidatos, numeros, sep = ", ", collapse = "; ")
# [1] "Dilma, 13; Serra, 45; Marina, 43"
```

Duas tarefas frequentemente importantes na manipulação de dados são a localização, e edição de texto. Para localizar um texto ou um número específicos, podemos usar a função `grep()`, a qual retorna o(s) índice(s) de um vetor em que um determinado texto pode ser encontrado. A função recebe como argumentos o padrão procurado e um vetor com textos ou números, como no exemplo a seguir:

```
palavras <- c("É", "difícil", "achar", "uma", "agulha", "num",
              "palheiro")
grep("agulha", palavras)
# [1] 5
```

Para fazer substituições em textos, podemos usar a função `sub()` se o objetivo for substituir apenas a primeira ocorrência do padrão procurado, e `gsub()` se quisermos substituir todas as ocorrências. Ambas as funções recebem como argumentos o padrão procurado, o texto substituto e o vetor a ser modificado:

```
x <- c("abc abc abc", "abc abc abc")
sub("abc", "zz", x)
# [1] "zz abc abc" "zz abc abc"

gsub("abc", "zz", x)
# [1] "zz zz zz" "zz zz zz"
```

Na busca e substituição de strings, podemos usar *expressões regulares*, ou seja, strings em que algumas sequências de caracteres têm significado especial, como `"\n"` que já usamos várias vezes para produzir quebra de linha. A seguir, veremos mais alguns exemplos.

Os símbolos `^` e `$` têm significado especial quando em expressões regulares, significando, respectivamente, início e final da *string*:

```
x <- c("abcx", "xabc", "abc")
grep("^abc", x)
# [1] 1 3

sub("abc$", "zz", x)
# [1] "abcx" "xzz"  "zz"
```

No código acima, o comando usando a função `grep()` não identificou o segundo elemento de `x` como correspondendo à expressão regular porque as letras `abc` não eram as três primeiras letras da string, e o comando usando a função `sub()` não fez a substituição no primeiro elemento de `x` porque as últimas letras não eram `abc`.

O símbolo `.` significa *qualquer caractere*, e, do conjunto de strings de quatro letras abaixo, somente serão substituídas aquelas em que a primeira letra for `a` e última for `d`:

```
x <- c("abcd", "aaaa", "aefd", "bbbb")
sub("a..d", "zzzz", x)
# [1] "zzzz" "aaaa" "zzzz" "bbbb"
```

O símbolo $*$ indica que o caractere precedente deve ser repetido zero ou mais vezes:

```
sub("a*", "", x)
# [1] "bcd"  ""     "efd"  "bbbb"

sub("b.*", "", x)
# [1] "a"    "aaaa" "aefd" ""
```

Os símbolos `[` e `]` delimitam uma sequência de caracteres a serem identificados, podendo ser usado um hífen para dispensar a digitação de valores intermediários:

```
x <- "abcdefghijklmnopqrstuvwxyz"
gsub("[bf]", ".", x)
# [1] "a.cde.ghijklmnopqrstuvwxyz"

gsub("[b-f]", ".", x)
# [1] "a.....ghijklmnopqrstuvwxyz"

gsub("[b-fp-ry]", ".", x)
# [1] "a.....ghijklmno...stuvwx.z"
```

No primeiro comando do código acima, são identificadas apenas as letras `b` e `f`, enquanto no segundo são identificadas as letras `bcdef`, sendo substituídas na expressão regular por um hífen as letras `cde` (ou seja, as letras que, por ordem alfabética, estão entre `b` e `f`). Observe que ponto não tem significado especial porque ele não está no primeiro argumento passado para a função `sub()` e, portanto, não faz parte de uma expressão regular.

Os símbolos `(` e `)` delimitam um grupo de caracteres que poderá ser utilizado numa substituição, sendo, então, referenciado por `\\n`, sendo `n` o número correspondente ao grupo:

```
x <- c("xcdab", "xghef", "xklij")
sub("x(..)(..)", "*\\2\\1", x)
# [1] "*abcd" "*efgh" "*ijkl"
```

No exemplo acima, especificamos que os trechos contendo um `x`, seguido de quatro caracteres, sendo os dois primeiros parte do que definimos como primeiro grupo e os outros dois parte do que definimos como segundo grupo, deveriam ser substituídos por um $*$ seguido do segundo grupo e do primeiro grupo.

Vimos aqui apenas alguns exemplos ilustrativos das situações mais comuns de uso de expressões regulares. Para um tratamento mais completo do tema, consulte a documentação do R sobre o assunto:

```
?regex
```

10.2 FUNÇÕES

Como já foi dito antes, um princípio que deve sempre ser seguido quando se escreve um script ou código de algum programa é o de se evitar repetições de código. Sempre que houver a necessidade de se repetir o mesmo código em diferentes partes de um script, é recomendável a criação de uma função que execute a tarefa desejada. Assim, se for encontrado algum erro no código, será suficiente fazer a correção num único lugar. O código do exemplo abaixo cria a função `hipotenusa()`, que recebe como argumentos dois objetos, `a` e `b` (que deverão

corresponder ao comprimento dos catetos de um triângulo retângulo), e retorna um terceiro valor, `h`, a hipotenusa calculada:

```
hipotenusa <- function(a, b){
  h <- sqrt(a ^ 2 + b ^ 2)
  h
}
hipotenusa(4, 3)
# [1] 5

hipotenusa(c(4, 5, 6), c(3, 4, 5))
# [1] 5.000 6.403 7.810
```

Para criar uma função, usamos o símbolo de atribuição `<-`, como na criação de qualquer objeto. Em seguida a expressão `function(){}`. Entre os parênteses devem ser indicados os argumentos que a função receberá, e, entre as chaves, deverá estar o código a ser executado pela função. Se a última linha da função contiver um objeto, como no exemplo acima, ele será retornado para a área de trabalho do R.

10.3 BLOCOS ENTRE CHAVES

Na função que nos serviu de exemplo na seção anterior, o código da função está entre chaves. As chaves delimitam as linhas de código que o R deverá interpretar como parte da função, e serão desnecessárias se o código a ser executado contiver apenas uma linha. Por exemplo, o código acima poderia ser reescrito como:

```
hipotenusa <- function(a, b) sqrt(a ^ 2 + b ^ 2)
hipotenusa(4, 3)
# [1] 5
```

A delimitação de trechos de código entre chaves é utilizada em diversas circunstâncias, como na execução condicional de parte do código e na execução de loops, como veremos nas seções seguintes. Embora o uso de chaves seja obrigatório apenas quando o trecho do código a ser executado contiver mais de uma linha, às vezes é útil acrescentar chaves para tornar a leitura do código mais clara.

10.4 EXECUÇÃO CONDICIONAL DE CÓDIGO

Para executar parte do código somente se determinada condição for verdadeira, utilizamos o comando `if()`, incluindo a condição a ser testada entre os parênteses. Se houve algum código alternativo a ser executado no caso da condição ser falsa, podemos informar isso para o R com o comando `else`. Para se familiarizar com o comando `if()`, execute o código abaixo com diferentes valores para os objetos `a` e `b`:

```
a <- 1
b <- 2
if(a > b) {
  cat("'a' é maior do que 'b'\n")
} else {
  if(b > a)
    cat("'b' é maior do que 'a'\n")
  else
    cat("'a' e 'b' têm o mesmo valor\n")
}
# 'b' é maior do que 'a'
```

Uma fonte comum de confusão para iniciantes é que os valores `NULL` e `NA` não podem ser usados diretamente em testes lógicos, sendo necessário usar as funções `is.null()` e `is.na()`. Exemplos:

```
nada <- NULL
nada == NULL
# logical(0)

is.null(nada)
# [1] TRUE

ausente <- c(1, 0, 1, NA, 0, 0, 1)
ausente == NA
# [1] NA NA NA NA NA NA NA

is.na(ausente)
# [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
```

10.5 FAMÍLIA DE FUNÇÕES *apply*

As funções da família *apply* podem ser usadas para evitar o uso dos loops que serão vistos na seção 10.7. No R, a execução de loops é extremamente lenta e, geralmente, a realização da mesma tarefa com as funções da família *apply* é dezenas de vezes mais rápida.

A função `apply()` aplica uma função a todas as linhas ou a todas as colunas de uma `matrix` ou `data.frame`. O primeiro argumento a ser passado para a função `apply()` deve ser uma `matrix` ou um `data.frame`; o segundo argumento deve ser o número 1 ou o número 2, indicando se a função do terceiro argumento deverá ser aplicada, respectivamente, às linhas ou às colunas da `matrix` ou `data.frame`; o terceiro argumento deve ser a função a ser aplicada. O valor retornado por `apply()` é um vetor com os resultados da aplicação da função fornecida como terceiro argumento. No exemplo seguinte, criamos um `data.frame` com dois vetores de números e usamos a função `apply()` para calcular a soma dos valores das linhas e, em seguida, a soma dos valores das colunas:

```
b <- data.frame(x = c(1, 2, 3, 4, 2, 1, 3, 2, 1),
                y = c(4, 5, 3, 6, 3, 5, 4, 4, 3))
apply(b, 1, sum)
# [1]  5  7  6 10  5  6  7  6  4

apply(b, 2, sum)
#  x  y
# 19 37
```

As funções `lapply()` e `sapply()` aplicam uma função a uma lista de objetos. A diferença entre elas é que a primeira retorna uma nova lista, e a segunda, sempre que possível, um vetor ou matriz. No código abaixo, criamos uma lista de dois vetores e, em seguida, calculamos o valor médio dos valores dos dois vetores:

```
lista <- list(a = c(3, 5, 1), b = c(8, 8, 2, 7))
lapply(lista, mean)
# $a
# [1] 3
#
# $b
# [1] 6.25
```

```
sapply(lista, mean)
#    a    b
# 3.00 6.25
```

A função a ser aplicada por qualquer uma das funções da família *apply* não precisa existir previamente, podendo até mesmo ser criada dentro da própria chamada a uma das funções *apply*. No exemplo seguinte, chamamos a função `lapply()` com uma função simples que adiciona o valor 1 ao objeto recebido como argumento:

```
lapply(lista, function(x){x + 1})
# $a
# [1] 4 6 2
#
# $b
# [1] 9 9 3 8
```

No código abaixo (resultados omitidos), usamos a função `sapply()` para descobrir quais colunas de um data.frame produzido pela função `read.spss()` do pacote `foreign` contêm categorias (*levels*) duplicadas. Esse é um problema comum quando usamos um banco de dados do SPSS preparado por terceiros, como, por exemplo, o banco de dados do *Estudo Eleitoral Brasileiro de 2002* disponível no *Consórcio de Informações Sociais*.[1] No caso do ESEB 2002, uma das colunas com rótulos repetidos é a *Q5*. Esta variável contém os nomes dos municípios em que a pesquisa foi realizada, entre os quais, dois têm o mesmo nome: Santa Rita.

```
eseb <- read.spss("BD_CIS_0012_vFinal.sav")
lbtb <- attr(eseb, "label.table")
dup <- sapply(lbtb, function(x) sum(duplicated(x)))
dup[dup > 0]
dup <- sapply(lbtb, function(x) sum(duplicated(names(x))))
dup[dup > 0]
q5lev <- levels(eseb$Q5)
q5lev[duplicated(q5lev)]
```

[1] http://www.cis.org.br.

A função `tapply()` é uma das mais importantes para cientistas sociais. Ela agrega os valores de um vetor numérico segundo os valores de alguma variável categórica e, então, aplica a função a cada agregado do vetor numérico. A função recebe, como argumentos obrigatórios, a variável numérica, a variável categórica e a função a ser aplicada. No exemplo a seguir, calculamos a estatura média de algumas pessoas, segundo o sexo:

```
sexo <- c("M", "F", "F", "M", "F", "M")
estatura <- c(1.70, 1.68, 1.73, 1.83, 1.60, 1.76)
tapply(estatura, sexo, mean)
#     F     M
# 1.670 1.763
```

Os exemplos apresentados até aqui contêm apenas as formas mais simples de usar as funções da família *apply*. Para todas elas, é possível fornecer argumentos adicionais a serem passados à função a ser aplicada. Por exemplo, no código a seguir, a primeira tentativa de calcular a média falha para as mulheres porque existe um `NA` entre os valores de estatura; na segunda tentativa, passamos o argumento *na.rm = TRUE* para a função `mean()` e o cálculo é realizado corretamente:

```
sexo <- c("M", "F", "F", "M", "F", "M")
estatura <- c(1.70, NA, 1.73, 1.83, 1.60, 1.76)
tapply(estatura, sexo, mean)
#     F     M
#    NA 1.763

tapply(estatura, sexo, mean, na.rm = TRUE)
#     F     M
# 1.665 1.763
```

No próximo exemplo com a função `tapply()`, ao invés de passar como argumento uma variável categórica, passamos um `data.frame` com duas variáveis categóricas que deverão ser utilizadas para agregar os valores numéricos antes de aplicar a função `mean()`. O resultado é uma matriz de valores médios de votação segundo o sexo e a escolaridade:

```
load("senado2006.RData")
tapply(sen$votos, sen[, c("escola", "sexo")], mean)
#                                sexo
# escola                          Feminino Masculino
#   Lê e Escreve                        NA    1416.0
#   Ensino Fundamental incompleto   834785   26924.5
#   Ensino Fundamental completo         NA   44556.9
#   Ensino Médio incompleto           7050     600.5
#   Ensino Médio completo            55509  159130.4
#   Ensino Superior incompleto       27828  465331.1
#   Ensino Superior completo        446720  529390.2
```

Algumas células da tabela estão preenchidas com `NA` porque não havia nenhum candidato ao senado com a correspondente combinação de características.

A função `aggregate()`, embora não tenha "apply" em seu nome, pertence à mesma família das demais funções vistas nesta seção. Ela é semelhante à `tapply()`, aplicando uma função a subconjuntos de um banco de dados e retornando uma tabela com os resultados. A função `aggregate()` recebe como argumentos obrigatórios uma `list` de variáveis numéricas, uma `list` de variáveis categóricas e a função a ser aplicada (lembre-se que um `data.frame` é uma `list`). Os subconjuntos são criados segundo as categorias das variáveis categóricas fornecidas como argumentos para a função. Exemplo:

```
aggregate(sen[, c("vgasto", "votos", "p.valido")],
  sen[, c("escola", "sexo")], mean)
#                          escola      sexo  vgasto    votos p.valido
# 1  Ensino Fundamental incompleto  Feminino 1410000 834785.0  28.1870
# 2        Ensino Médio incompleto  Feminino 2000000   7050.0   0.0380
# 3          Ensino Médio completo  Feminino  941250  55508.8   1.0917
# 4     Ensino Superior incompleto  Feminino  125000  27827.5   0.7585
# 5       Ensino Superior completo  Feminino 2254091 446720.2  18.7605
# 6                   Lê e Escreve Masculino   30000   1416.0   0.0250
# 7  Ensino Fundamental incompleto Masculino 1002500  26924.5   0.9375
# 8    Ensino Fundamental completo Masculino 1083571  44556.9   2.1624
# 9        Ensino Médio incompleto Masculino  700000    600.5   0.2020
# 10         Ensino Médio completo Masculino 1884828 159130.4   9.4845
# 11    Ensino Superior incompleto Masculino 1668214 465331.1  15.1171
# 12      Ensino Superior completo Masculino 2318761 529390.2  14.9489
```

Para perceber melhor a diferença entre `aggregate()` e `tapply()`, compare os resultados do código abaixo (resultados omitidos):

```
tapply(sen$votos, sen[, c("escola", "sexo")], mean)
aggregate(list(votos = sen$votos), sen[, c("escola", "sexo")], mean)
```

10.6 STRSPLIT(), UNLIST() E DO.CALL()

Ocasionalmente, pode ser necessário partir uma string em pedaços, o que pode ser feito com a função `strsplit()`, que recebe como argumentos a string a ser partida e uma string que será utilizada para identificar o separador. O resultado é uma lista de strings. Imagine, por exemplo, que obtivemos um texto contendo uma lista de nomes de pessoas e suas respectivas idades e que queremos construir uma matriz com duas colunas, uma para os nomes e outra para as idades. O primeiro passo seria partir a string de modo a separar os nomes das idades:

```
x <- c("Maria = 25", "José = 27", "Pedro = 3")
xs <- strsplit(x, " = ")
xs
# [[1]]
# [1] "Maria" "25"
#
# [[2]]
# [1] "José" "27"
#
# [[3]]
# [1] "Pedro" "3"
```

O próximo passo seria usar a função `do.call()`:

```
do.call("rbind", xs)
#      [,1]    [,2]
# [1,] "Maria" "25"
# [2,] "José"  "27"
# [3,] "Pedro" "3"
```

A função `do.call()` recebe como argumentos o nome de uma função e uma lista contendo os argumentos a serem passados à função.

Cada elemento da lista é passado para a função como se fosse um argumento, O código acima, por exemplo, é equivalente a:

```
rbind(xs[[1]], xs[[2]], xs[[3]])
#      [,1]    [,2]
# [1,] "Maria" "25"
# [2,] "José"  "27"
# [3,] "Pedro" "3"
```

Se o objetivo fosse simplesmente criar um vetor concatenando todos os valores (nomes e idades), poderíamos usar a função `unlist()` ou, novamente, a função `do.call()`:

```
unlist(xs)
# [1] "Maria" "25"    "José"  "27"    "Pedro" "3"

do.call("c", xs)
# [1] "Maria" "25"    "José"  "27"    "Pedro" "3"
```

A função `unlist()` é genérica e possui vários métodos. O método padrão tenta concatenar os elementos de uma lista convertendo-a num vetor.

A propósito, a partir da lista `xs`, poderíamos reconstituir o objeto `x` com o seguinte comando:

```
sapply(xs, function(x) paste(x[1], "=", x[2]))
# [1] "Maria = 25" "José = 27"  "Pedro = 3"
```

10.7 LOOPS *for* E *while*

Os loops `for` e `while` permitem executar repetidamente uma sequência de comandos. Como sempre, o uso de chaves é obrigatório apenas se o código a ser executado contiver mais de uma linha. O loop `while` tem sintaxe semelhante à da condição `if`, ou seja, escrevemos entre os parênteses a condição a ser testada, e o código será executado enquanto o teste resultar verdadeiro. O código seguinte exemplifica o uso do loop `while`:

```
i <- 1
fim <- 4
while(i < fim){
  i <- i + 1
  if(i < fim)
    cat("Circulando...\n")
  else
    cat("Esta é a última volta.\n")
}
# Circulando...
# Circulando...
# Esta é a última volta.
```

O loop `for` tem uma sintaxe um pouco mais complexa porque o código entre parênteses não apenas testa a condição: ele cria o objeto a ser testado. No exemplo a seguir, o objeto `i` é criado e seu valor é incrementado de acordo com os valores do vetor `1:10`. Ou seja, o loop é executado 10 vezes e cada novo valor de `i` é somado a `j`:

```
j <- 0
for(i in 1:10)
  j <- j + i
j
# [1] 55
```

O vetor que especifica os valores a serem assumidos pelo objeto criado entre parênteses no loop `for` não precisa, necessariamente, ser numérico. No exemplo seguinte, ele é o vetor do tipo `character` criado na seção 10.1:

```
for(i in palavras)
  if(i == "agulha") cat("Achei!\n")
# Achei!
```

Por ser mais compacto, o loop `for` deve ser usado quando sabemos com antecedência o número de vezes que o código deverá ser executado. Se essa informação não for conhecida, será necessário o uso do loop `while`. É ainda possível usar os comandos `next` e `break` no interior de loops para interromper a sua execução e, respectivamente, reiniciar o loop ou sair dele.

10.8 A FUNÇÃO SOURCE()

A função `source()` executa os comandos existentes num script do R. Um princípio básico de programação é evitar repetição de código. Ao colocarmos um trecho de código que deverá ser executado repetidamente num arquivo separado, podemos facilmente inclui-lo em outros scripts do R sem ter que copiar e colar o texto, com a vantagem adicional de que se percebermos a necessidade de melhorar ou modificar o código, faremos isso em apenas um arquivo. Se fizéssemos cópias do código em cada novo script que produzíssemos, seria preciso repetir as modificações em todos eles.

Capítulo 11

Mapas

Neste capítulo, veremos como usar o R para criar mapas usando arquivos no formato `Arcview`, facilmente encontráveis na internet.[1] Para carregar um mapa no R é preciso obter os arquivos com extensão `shp`, `shx` e `dbf`, devendo-se passar o diretório em que se encontram os arquivos e a raiz comum dos nomes dos arquivos como argumento para a função `readOGR()`, do pacote `rgdal`. A função cria um objeto contendo os polígonos cujos perímetros correspondem às unidades geográficas do mapa. As coordenadas `x` e `y` dos polígonos têm os mesmos valores, respectivamente, da longitude e da latitude dos pontos do mapa. O objeto criado pela função `readOGR()` também inclui um `data.frame` com informações adicionais, e o mapa pode ser desenhado com a função `plot()`. No exemplo abaixo, criamos o objeto `br` contendo o mapa do Brasil:

```
dir("mapa_BR")
# [1] "BRASIL.dbf" "BRASIL.shp" "BRASIL.shx"

library(rgdal)
br <- readOGR("mapa_BR", "BRASIL")
# OGR data source with driver: ESRI Shapefile
# Source: "mapa_BR", layer: "BRASIL"
# with 27 features and 3 fields
# Feature type: wkbPolygon with 2 dimensions
```

[1] Ver, por exemplo, `ftp://geoftp.ibge.gov.br/malhas_digitais`. O mapa do Brasil utilizado neste capítulo foi obtido do sítio `http://www.gismaps.com.br/divpol/divpol.htm` em 05 de setembro de 2009.

```
summary(br)
# Object of class SpatialPolygonsDataFrame
# Coordinates:
#      min     max
# x -73.84 -34.858
# y -33.77   5.383
# Is projected: NA
# proj4string : [NA]
# Data attributes:
#        UF          ESTADO   REGIAO
#  AC     : 1   Acre    : 1   CO:4
#  AL     : 1   Alagoas : 1   NE:9
#  AM     : 1   Amap<a0>: 1   NO:7
#  AP     : 1   Amazonas: 1   SE:4
#  BA     : 1   Bahia   : 1   SU:3
#  CE     : 1   Cear<a0>: 1
#  (Other):21   (Other) :21
```

O objeto `br`, criado com os comandos acima, é uma lista de polígonos, e cada linha do `data.frame` contém informações sobre um dos polígonos da lista. Como podemos observar pelo resultado de `summary()`, o `data.frame` deste mapa contém três variáveis, `UF`, `ESTADO` e `REGIAO`. Não utilizaremos a variável `ESTADO` do `data.frame`, mas o leitor interessado pode corrigir a codificação de caracteres dessa variável com a função `toUTF8()`, do pacote `descr`:

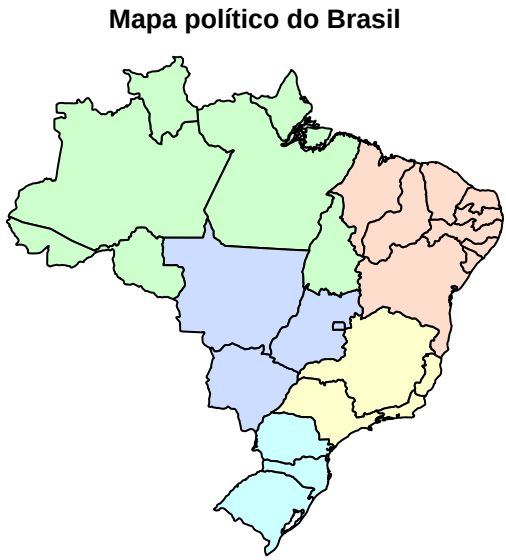
```
library(descr)
br$ESTADO <- toUTF8(br$ESTADO, "IBM850")
```

O procedimento para produção de mapas consiste na colorização dos polígonos de acordo com algum critério. No exemplo da Figura 11.1, utilizamos a variável `REGIAO` do próprio mapa para colori-lo:[2]

```
palette(c("#ccddff", "#ffddcc", "#ccffcc", "#ffffcc", "#ccffff"))
plot(br, col = br$REGIAO)
title("Mapa político do Brasil")
```

[2] Ver seção 8.2 para maiores informações sobre o uso de cores.

FIGURA 11.1: Exemplo de mapa



Objetos desse tipo não aceitam o parâmetro *main* quando produzindo um gráfico. Por isso, tivemos que usar `title()` para adicionar o título principal.

O próximo mapa será colorido de acordo com o IDH estadual calculado pelo PNUD para o ano de 2005 (CEPAL; PNUD; OIT, 2008) (coluna `ano05`). O primeiro passo para produzir o gráfico será carregar um banco de dados com o IDH estadual:

```
idh <- read.table("IDH_Brasil.csv", sep = "\t", header = TRUE)
idh <- idh[, c("UF", "ano05")]
names(idh) <- c("UF", "idh05")
```

Em seguida, teremos que fazer a junção dos dados do IDH com o `data.frame` do objeto `br` contendo informações não cartográficas sobre os polígonos do mapa. Para isso, usaremos a função `merge()`, mas, antes, vamos converter de `factor` para `character` as variáveis `UF` do mapa e do banco de dados com os IDHs. Isso permitirá à função `merge()` comparar as duas variáveis que possuem os mesmos rótulos, mas em sequências diferentes:

```
idh$UF <- as.character(idh$UF)
br$UF <- as.character(br$UF)
br <- merge(br, idh)
```

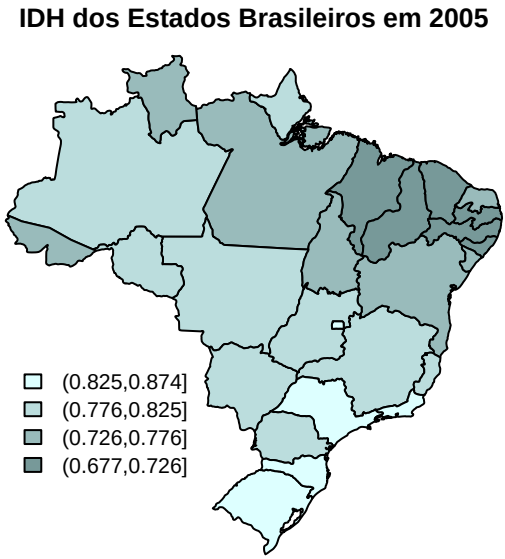
Finalmente, vamos criar, com a função `cut()`, uma variável categórica com faixas de valores de IDH, `idhc`, que será usada para colorir o mapa. Deixamos para a função `cut()` a tarefa de determinar os pontos de corte ao dividir o IDH em quatro intervalos:

```
br$idhc05 <- cut(br$idh05, 4)
levels(br$idhc05)
# [1] "(0.677,0.726]" "(0.726,0.776]" "(0.776,0.825]" "(0.825,0.874]"
```

Assim como no mapa anterior, na Figura 11.2 colorimos as Unidades de Federação usando os `levels` de uma variável categórica como números das cores da palheta. Para o posicionamento da legenda, usamos coordenadas geográficas de um ponto localizado numa das áreas em branco do mapa. Os valores apresentados na legenda são o resultado do comando `levels(idhc)`, digitado acima.

```
palette(c("#779999", "#99bbbb", "#bbdddd", "#ddffff"))
plot(br, col = br$idhc05)
title("IDH dos Estados Brasileiros em 2005")
legend(-74, -18, bty = "n", fill = 4:1,
       legend = levels(br$idhc05)[4:1])
```

FIGURA 11.2: Exemplo de mapa (II)



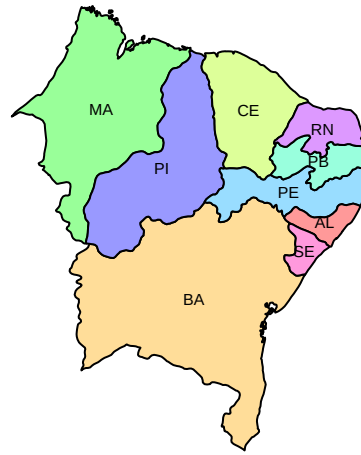
Poderíamos acrescentar linhas, pontos ou texto ao mapa usando as coordenadas de acidentes geográficos, estradas ou cidades como valores para os eixos `x` e `y`. Por exemplo, as coordenadas geográficas de Fortaleza e Manaus são, respectivamente, 03° 43' 01" S, 38° 32' 34" O e 03° 08' 07" S, 60° 01' 34" O. Para adicioná-las ao mapa (resultado omitido):

```
points(c(-38.33, -60.03), c(-3.82, -3.14), pch = 23, cex = 0.6,
       col = "red", bg = "yellow")
text(c(-38.33, -60.03), c(-3.82, -3.14), pos = c(4, 2), cex = 0.6,
     labels = c("Fortaleza", "Manaus"))
```

Como mostra a Figura 11.3, um subconjunto dos polígonos de um mapa pode ser selecionado pelo uso de sintaxe análoga à empregada com um `data.frame`. No exemplo, acrescentamos as abreviaturas das unidades da federação ao mapa. Para tanto, utilizamos a função `coordinates()` para extrair do objeto `ne` uma matrix com as coordenadas dos centroides dos polígonos. Os centroides correspondem às coordenadas geográficas do centro da unidade da federação.

```
ne <- br[br$REGIAO == "NE", ]
plot(ne, col = rainbow(9, 0.4))
text(coordinates(ne), as.character(ne$UF), cex = 0.7)
```

FIGURA 11.3: Mapa do Nordeste do Brasil



Capítulo 12

Análise de redes sociais

Existem vários pacotes do R voltados para análise de redes sociais. Neste capítulo, veremos o uso básico do `igraph`, que possui funções de uso bastante intuitivo para a criação de redes pequenas, e funções capazes de lidar de modo eficiente com redes grandes. Uma forma de criar um objeto de classe `igraph`, representando uma rede social, é pelo uso da função `graph.formula()`, como exemplificado no código abaixo:

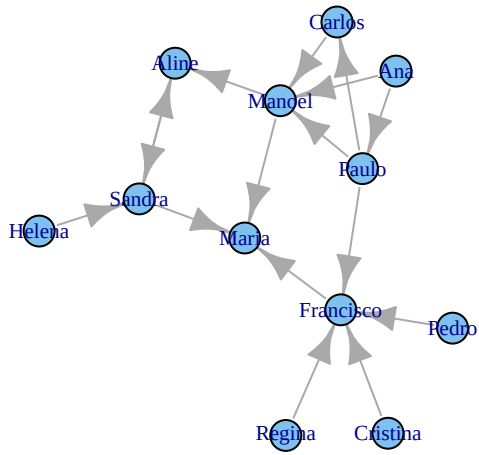
```
library(igraph)
g <- graph.formula(
    Regina --+ Francisco,
    Maria +-- Sandra,
    Pedro --+ Francisco,
    Paulo --+ Francisco +-- Cristina,
    Maria +-- Manoel +-- Carlos,
    Ana --+ Paulo --+ Carlos,
    Manoel --+ Aline +-+ Sandra +-- Helena,
    Paulo --+ Manoel +-- Ana,
    Francisco --+ Maria
)
```

A função `graph.formula()` recebe como argumentos dois ou mais nomes de vértices ligados pelos sinais - e + simbolizando arestas, em que o sinal + representa a ponta da seta. É possível também fazer

a ligação usando apenas o símbolo - e, nesse caso, a rede será não-direcional. Não é necessário o uso de aspas nos nomes dos vértices se eles não contiverem espaços em branco. Para visualizar a rede, usamos `plot()`, como ilustrado na Figura 12.1.

```
plot(g)
```

FIGURA 12.1: Sociograma



A função `tkplot()` permite escolher manualmente a posição dos vértices. Para tanto, deve-se criar um objeto com o resultado da função e utilizá-lo como argumento para a função `tkplot.getcoords()`, que somente deve ser chamada quando os vértices tiverem sido manualmente posicionados. Depois de guardadas as coordenadas num objeto, a janela do gráfico interativo pode ser fechada. No exemplo abaixo, a referência ao gráfico foi salva no objeto `tkp` e as coordenadas no objeto `g.coord` (resultados omitidos):

```
tkp <- tkplot(g)
g.coord <- tkplot.getcoords(tkp)
g <- set.graph.attribute(g, "layout", value = g.coord)
```

Raramente será conveniente posicionar os vértices manualmente. O mais prático é estabelecer o layout do sociograma de modo automático. Os objetos da classe `igraph` podem receber, por meio da função

`set.graph.attribute()`, vários atributos que serão, então, utilizados em todos os sociogramas produzidos. A função recebe como argumentos o objeto da classe `igraph`, o nome do atributo e o valor do atributo. No exemplo anterior, estipulamos que o *layout* do gráfico seria determinado pelas coordenadas dos vértices definidas manualmente com a função `tkplot()`.

No código abaixo, primeiramente, chamamos a função `set.seed()` com um valor qualquer para evitar que os gráficos produzidos por nosso script tenham layouts diferentes cada vez que o script for executado. Isso ocorreria porque o algoritmo de produção de layout faz uso de números pseudoaleatórios. Ao usar a função `set.seed()`, garantimos que os números pseudoaleatórios serão produzidos sempre na mesma sequência.

```
set.seed(333)
g <- set.graph.attribute(g, "layout",
                         value = layout.fruchterman.reingold(g))
```

Os comandos executados abaixo acrescentam outros atributos úteis para a produção dos sociogramas. Por padrão, a função `plot()` utiliza os índices dos vértices como rótulos, mas ao criar a rede com a função `graph.formula()`, os nomes foram armazenados no atributo *name*. Assim, utilizamos a função `get.vertex.attribute()` para obter a lista de nomes, e a função `set.vertex.attribute()` para determinar que os rótulos dos vértices deverão ser os nomes. Outros atributos que acrescentamos foram o tamanho dos vértices, a distância entre os vértices e os seus rótulos e o tamanho das pontas das arestas:

```
nomes <- get.vertex.attribute(g, "name")
g <- set.vertex.attribute(g, "label", value = nomes)
g <- set.vertex.attribute(g, "size", value = 6)
g <- set.vertex.attribute(g, "label.dist", value = 0.7)
g <- set.edge.attribute(g, "arrow.size", value = 0.5)
```

Outra alteração que faremos nos sociogramas seguintes será colorir os vértices de acordo com alguma característica dos indivíduos. Na Figura 12.2, os vértices estão coloridos de acordo com o grau de proximidade e de intermediação dos indivíduos na rede social. O grau

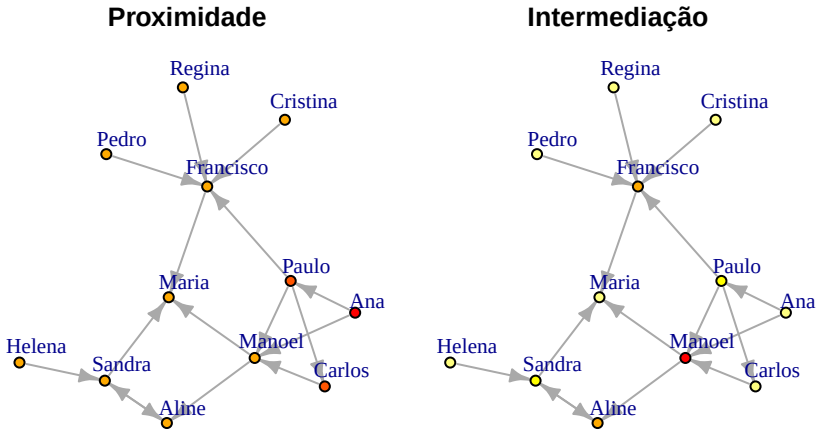
de proximidade de um vértice é proporcional à distância média dele para todos os outros vértices, e o grau de intermediação indica o número de vezes em que um vértice representa o caminho mais curto entre dois outros vértices. O grau de proximidade dos vértices foi calculada com a função `closeness()`, e o grau de intermediação com `betweenness()`. Para colorir os vértices, usamos as cores produzidas pela função `heat.colors()`, mas na ordem inversa: quanto maior a métrica de centralidade, mais próxima do vermelho a cor utilizada:

```
cores <- heat.colors(5)
proxi <- closeness(g)
proxi.max <- max(proxi)
cores.p <- 5 - round(4 *(proxi / proxi.max))
cores.p <- cores[cores.p]
inter <- betweenness(g)
inter.max <- max(inter)
cores.i <- 5 - round(4 *(inter / inter.max))
cores.i <- cores[cores.i]
```

Na Figura 12.2, para colocar os dois sociogramas lado a lado no mesmo gráfico, usamos o parâmetro gráfico *mfrow*, e para reduzir o tamanho da fonte, o parâmetro *cex*. O parâmetro *mar* foi ajustado para reduzir o espaço do gráfico gasto com margens em branco.

```
par(mfrow = c(1, 2), mar = c(0, 0, 1, 2) + 0.1, cex = 0.7, xpd = TRUE)
plot(g, vertex.color = cores.p, main = "Proximidade")
par(mar = c(0, 1, 1, 1) + 0.1)
plot(g, vertex.color = cores.i, main = "Intermediação")
```

FIGURA 12.2: Sociogramas dos graus de centralidade e intermediação



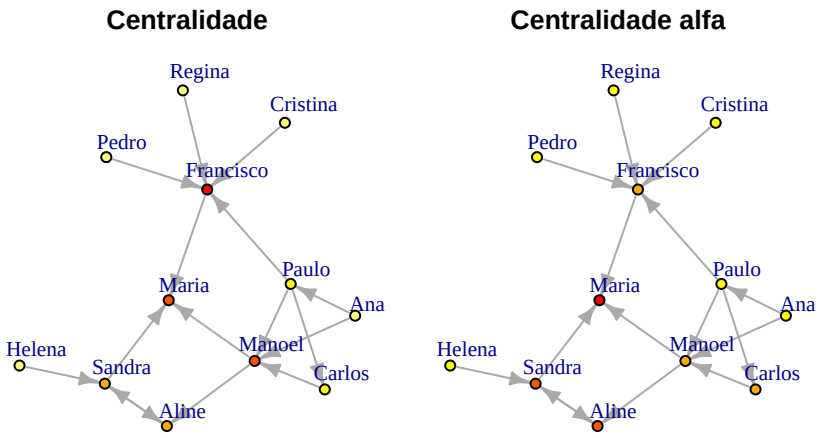
No código a seguir, são calculadas a *centralidade* e a *centralidade alfa* dos indivíduos. No cálculo da *centralidade*, passamos o argumento *mode = "in"* para a função `degree()` para que somente fossem contadas as arestas que apontam para o vértice. O cálculo da *centralidade alfa* de Bonacich, realizado pela função `alpha.centrality()`, pode falhar para algumas redes e valores de *alpha*. Por isso, usamos a função `try()`, que testa o código antes de executá-lo, evitando interrupção na execução do script causada por erro de computação, e a função `exists()` que retorna `TRUE` se o objeto cujo nome lhe foi passado como argumento existir. Os procedimentos seguintes que utilizam o objeto `alfa`, correspondendo à *centralidade alfa*, somente serão executados se `alfa` tiver sido criado. A centralidade alfa é calculada considerando não apenas o número de arestas que apontam para um vértice, mas também a centralidade dos vértices onde se originam as arestas. O argumento *alpha* indica a "importância relativa de fatores endógenos versus fatores exógenos na determinação da centralidade" (CSÁRDI; NEPUSZ, 2006).

```
central <- degree(g, mode = "in")
central.max <- max(central)
cores.c <- 5 - round(4 * (central / central.max))
cores.c <- cores[cores.c]
try(alfa <- alpha.centrality(g, alpha = 0.5))
```

```
if(exists("alfa")){
    alfa.min <- min(alfa)
    if(alfa.min < 0)
        alfa.min <- alfa.min * (-1)
    alfa2  <- alfa + alfa.min
    alfa2.max <- max(alfa2)
    cores.a <- 5 - round(4 * (alfa2 / alfa2.max))
    cores.a <- cores[cores.a]
}
```

```
par(mfrow = c(1, 2), mar = c(0, 0, 1, 2) + 0.1, cex = 0.7, xpd = TRUE)
plot(g, vertex.color = cores.c, main = "Centralidade")
par(mar = c(0, 1, 1, 1) + 0.1)
if(exists("alfa"))
    plot(g, vertex.color = cores.a, main = "Centralidade alfa")
```

FIGURA 12.3: Sociogramas dos graus de centralidade



Outro procedimento frequentemente útil é a identificação dos maiores cliques de uma rede, ou seja, dos maiores grupos de vértices mutuamente relacionados. Isso pode ser feito com a função `largest.cliques()`, que recebe como argumento uma rede não direcionada. Como a rede que criamos é direcionada, será preciso antes convertê-la, usando a função `as.undirected()`. A função `largest-`

`.cliques()` retorna uma lista dos índices dos vértices pertencentes aos maiores cliques.

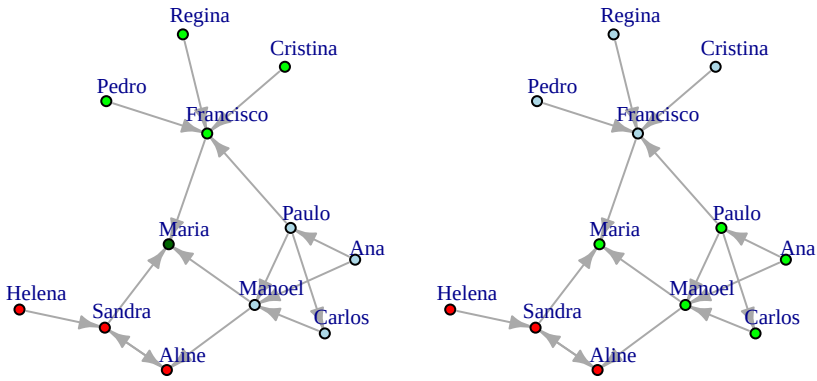
```
g.undir <- as.undirected(g)
g.mc <- largest.cliques(g.undir)
g.mc
# [[1]]
# [1] 10  6  8
#
# [[2]]
# [1] 9 6 8

nomes[g.mc[[1]]]
# [1] "Ana"    "Paulo"  "Manoel"
```

Além de localizar os maiores cliques, podemos usar as funções `walktrap.community()`, `spinglass.community()` e outras para identificar automaticamente a existência de possíveis subgrupos na rede. A primeira revela a existência de comunidades dando passos aleatórios a partir de cada um dos vértices. Por isso, quanto maior o valor do argumento *steps*, maior o número de membros considerados pertencentes a cada comunidade. O valor "correto" dependerá de considerações teóricas e metodológicas a serem feitas pelo pesquisador. Nos exemplos a seguir, usamos dois valores diferentes para comparar os resultados:

```
wtc <- membership(walktrap.community(g))
wtc12 <- membership(walktrap.community(g, steps = 12))
```

```
palette(c("red", "green", "lightblue", "darkgreen"))
par(mfrow = c(1, 2), mar = c(0, 0, 1, 2) + 0.1, cex = 0.7, xpd = TRUE)
plot(g, vertex.color = wtc)
par(mar = c(0, 1, 1, 1) + 0.1)
plot(g, vertex.color = wtc12)
```

FIGURA 12.4: Identificação de grupos I

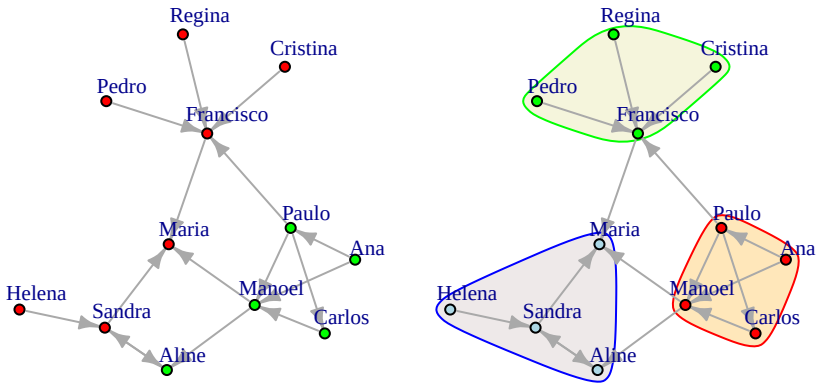


A função `spinglass.community()` permite especificar o número máximo de grupos a serem detectados, *spins*, e permite ponderar os vértices (argumento *weights*). No exemplo a seguir, usamos os argumentos *mark.groups* e *mark.col* para realçar a comunidade detectada:

```
sgc <- membership(spinglass.community(g))
sgc2 <- membership(spinglass.community(g, spins = 2))
sgc.g <- list(grep("1", sgc), grep("2", sgc), grep("3", sgc))
```

```
palette(c("red", "green", "lightblue", "snow2", "beige", "wheat1"))
par(mfrow = c(1, 2), mar = c(0, 0, 1, 2) + 0.1, cex = 0.7, xpd = TRUE)
plot(g, vertex.color = sgc2)
par(mar = c(0, 1, 1, 1) + 0.1)
plot(g, vertex.color = sgc, mark.groups = sgc.g,  mark.col = 6:4)
```

FIGURA 12.5: Identificação de grupos II



Como o gráfico da Figura 12.5 foi o último do capítulo, convém chamar novamente a função `par()` para reconfigurar os parâmetros gráficos para os valores originais:

```
par(mfrow = c(1, 1), cex = 1.0, mar = c(5.1, 4.1, 4.1, 2.1))
```

Depois de ter calculado várias métricas dos indivíduos na rede social que nos serviu de exemplo e de ter identificado as comunidades às quais pertencem os vértices, podemos criar um `data.frame` contendo todas essas variáveis, como no código abaixo:

```
gdf <- data.frame(nome = names(inter), inter, proxi, central, alfa,
                  wtc, sgc)
print(gdf, row.names = FALSE, digits = 3)
#       nome inter   proxi central alfa wtc sgc
#     Regina   0.0 0.00901       0 1.00   2   2
#  Francisco   3.5 0.00826       4 3.25   2   2
#      Maria   0.0 0.00758       3 6.04   4   3
#     Sandra   3.0 0.00909       2 3.71   1   3
#      Pedro   0.0 0.00901       0 1.00   2   2
#      Paulo   2.0 0.01429       1 1.50   3   1
#   Cristina   0.0 0.00901       0 1.00   2   2
#     Manoel   8.5 0.01000       3 3.12   3   1
#     Carlos   0.0 0.01087       1 1.75   3   1
```

```
#        Ana   0.0 0.01639       0 1.00   3   1
#      Aline   4.0 0.00901       2 4.42   1   3
#     Helena   0.0 0.00990       0 1.00   1   3
```

Se tivéssemos outro `data.frame` com características dos indivíduos constituintes da rede, poderíamos combinar os dois bancos de dados com a função `merge()` e realizar novas análises estatísticas, descritivas (como as vistas no Capítulo 6) ou inferenciais (como as do Capítulo 7).

Além de calcular métricas dos vértices, também pode ser necessário calcular métricas globais da rede, principalmente quando se pretende comparar as propriedades de várias redes. Entre essas propriedades, estão a razão entre o número de ligações entre os vértices da rede e o número teoricamente possível de ligações (*densidade*), a proporção de pares recíprocos em relação a todos os pares de vértices entre os quais há alguma aresta (*reciprocidade*), a probabilidade de dois vértices quaisquer que são conectados ao mesmo vértice estarem eles próprios conectados (*transitividade* ou *coeficiente de agrupamento*) e o número de agrupamentos. Esta última propriedade pode ser calculada considerando ou não a reciprocidade das relações:

```
d <- data.frame("Métrica" = c("Densidade", "Reciprocidade",
                              "Transitividade",
                              "Coef. de agrupamento (fraco)",
                              "Coef. de agrupamento (forte)"),
                "Valor" = c(graph.density(g), reciprocity(g),
                            transitivity(g),  no.clusters(g),
                            no.clusters(g, mode = "strong")))
print(d, row.names = FALSE, digits = 3)
#                       Métrica  Valor
#                     Densidade  0.121
#                  Reciprocidade  0.125
#                 Transitividade  0.171
#  Coef. de agrupamento (fraco)  1.000
#  Coef. de agrupamento (forte) 11.000
```

O pacote `igraph` também contém funções para criar redes a partir de matrizes de adjacência, vetores de índices ou vetores de nomes, que poderão ser muito úteis quando importando redes elaboradas em

outros programas ou criando redes a partir de dados já existentes. Por exemplo, a rede que utilizamos neste capítulo poderia ter sido criada da seguinte forma:

```
A <- c("Regina", "Pedro", "Sandra", "Paulo", "Francisco", "Manoel",
       "Carlos", "Regina", "Ana", "Paulo", "Manoel", "Aline",
       "Sandra", "Helena", "Manoel", "Ana")
B <- c("Francisco", "Maria", "Maria", "Francisco", "Cristina",
       "Carlos", "Regina", "Cristina", "Paulo", "Carlos", "Aline",
       "Sandra", "Helena", "Maria", "Ana", "Francisco")
g2 <- graph.edgelist(cbind(A, B))
```

GLOSSÁRIO

arrow: seta.
blue: azul.
bottom: chão.
choose: escolher.
cut: cortar.
data: dados.
device: dispositivo.
directory: diretório, pasta.
end: fim.
foreign: estrangeiro.
frame: quadro.
green: verde.
header: cabeçalho.
head: cabeça.
heat: calor.
height: altura.
hide: escoder.
label: rótulo.
left: esquerda.
length: comprimento.
level: nível.
library: biblioteca.
load: carregar.
lower: letra minúscula.
mode: modo.
print: imprimir.
read: ler.
record: registro, campo.
red: vermelho.
right: direita.
script: texto, código, sequência de comandos.
search: pesquisar.
set: configurar.
shape: forma, moldar, dar forma.
size: tamanho.
sink: afundar.
start: início.
step: dar um passo.
table: tabela.
tail: cauda.
top: topo.
true: verdadeiro.
wide: largo.
width: largura.
working: de trabalho.

Índice Remissivo

BIBLIOGRAFIA

ALMEIDA, Alberto Carlos. *A cabeça do brasileiro*. Rio de Janeiro: Record, 2007.

AQUINO, Jakson Alves de et al. *descr: Descriptive statistics*. [S.l.], 2014. R package version 1.0.3. Disponível em: <https://github.com-/jalvesaq/descr>.

BARNIER, Julien. *R pour les sociologues*. [S.l.], dez. 2008. Version provisoire.

BIVAND, Roger; KEITT, Tim; ROWLINGSON, Barry. *rgdal: bindings for the geospatial data abstraction library*. [S.l.], 2014. R package version 0.8-16. Disponível em: <http://CRAN.R-project.org/package=rgdal>.

BOLOGNESI, Bruno; GOUVÊA, Júlio; MIRÍADE, Angel. Eleições 2006: candidatos ao poder legislativo no Brasil (Banco de dados). In: *Consórcio de Informações Sociais*. Curitiba: Núcleo de Pesquisa em Sociologia Política Brasileira, 2007. Disponível em: <http://www.cis.org.br>. Acesso em: 12/082009.

CEPAL; PNUD; OIT. Índice de Desenvolvimento Humano (IDH), Brasil, regiões e estados, 1991-2005. In: CEPAL; PNUD; OIT (Ed.). *Emprego, desenvolvimento humano e trabalho decente: a experiência brasileira recente*. [s.n.], 2008. Disponível em: <http://www.cepal.org/brasil/noticias/noticias/3/34013-/EmpregoDesenvHumanoTrabDecente.pdf>. Acesso em: 10/04/2009.

CSÁRDI, Gábor; NEPUSZ, Tamás. The igraph software package for complex network research. *InterJournal*, Complex Systems, Manuscript Number 1695, 2006. Disponível em: <http://igraph.sf.net>.

DOWLE, M et al. *data.table: Extension of data.frame*. [S.l.], 2014. R package version 1.9.2. Disponível em: <http://CRAN.R-project.org-/package=data.table>.

ELFF, Martin. *memisc: tools for management of survey data, graphics, programming, statistics, and simulation*. [S.l.], 2013. R package version 0.96-9. Disponível em: <http://CRAN.R-project.org-/package=memisc>.

LANDEIRO, Victor Lemes. *Introdução ao uso do programa R*. Manaus, 2011.

R Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2014. Disponível em: <http://www.R-project.org/>.

TORGO, Luís. *Introdução à programação em R*. Porto, out. 2006.

WARNES, Gregory R. et al. *gdata: Various R programming tools for data manipulation*. [S.l.], 2014. R package version 2.13.3. Disponível em: <http://CRAN.R-project.org/package=gdata>.

XIE, Yihui. *knitr: a general-purpose package for dynamic report generation in R*. [S.l.], 2013. R package version 1.5. Disponível em: <http://yihui.name/knitr>.