Oswaldo Santos Baquero

Manipulação e visualização de dados no R

Versão preliminar

São Paulo
Faculdade de Medicina Veterinária e Zootecnia
Universidade de São Paulo
2016

Table of Contents

Folha de rostro 1.1
Autor 1.2
Introdução 1.3
Instalação e interface 1.4
Estilo de programação 1.5
Tipos e estruturas de objetos 1.6
Importação e exportação de arquivos 1.7
Geração de dados 1.8
Gramática dos gráficos 1.9
Estruturas de controle 1.10
Estruturação de bancos de dados 1.11
Operações em banco de dados 1.12
Protocolo de exploração de bancos de dados 1.13
Referências 1.14

Oswaldo Santos Baquero

Manipulação e visualização de dados no R

Versão preliminar

São Paulo
Faculdade de Medicina Veterinária e Zootecnia
2016

Oswaldo Santos Baquero
Laboratório de Epidemiologia e Estatística
Departamento de Medicina Veterinária Preventiva e Saúde Animal
Faculdade de Medicina Veterinária e Zootecnia
Universidade de São Paulo

Manipulação e visualização de dados no R

Oswaldo Santos Baquero

Contato: baquero@usp.br
Última revisão: 2 de abril de 2017
Em caso de não estar lendo a versão online, está versão pode estar desatualizada.
Clique aqui para acessar a página do livro.

A preparação de dados facilmente compreende 80% do processo de análise de dados (Dasu e Johnson, 2003) e a exploração de dados é um mecanismo que facilita a detecção de erros e a verificação de pressupostos para aplicar métodos de análise. Por outro lado, o domínio de técnicas de manipulação e visualização de dados traz ganhos na qualidade, produtividade e reproduzibilidade das análises de dados.

O R é uma linguagem de programação especializada na comunicação de instruções para a manipulação, cálculo e visualização de dados. O objetivo deste livro é apresentar os fundamentos de programação no R, sob o ponto de vista da manipulação e visualização de dados. Embora a teoria estatística não faça parte do conteúdo, alguns capítulos valem-se de métodos estatísticos para poder abordar aspectos de preparação e exploração de dados.

Após a apropriação dos conhecimentos apresentados neste livro, os leitores terão embasamento tanto para cuidar do que costuma ser a maior parte da análise de dados como para aprender com mais facilidade a teoria por trás dos métodos de análise (a programação dos conceitos teóricos é um recurso que aprimora o aprendizado).

Requisitos para reproduzir os códigos

- R versão 3.3.1 ou superior
- RStudio Desktop 0.99.902 ou superior
- Material suplementar do artigo *Zuur, Alain F., Elena N. Ieno, and Chris S. Elphick. "A protocol for data exploration to avoid common statistical problems." Methods in Ecology and Evolution 1.1 (2010): 3-14.*
- No capítulo de importação e exportação de arquivos são necessários os "domicilios.csv" e "dom.csv". Esses arquivos estão disponíveis repositório do livro:

https://github.com/leb-fmvz-usp/manipulacao-e-visualizacao-de-dados-no-r. Para optê-los há que baixar o repositório (botão verde "Clone or download"), descompactar a pasta, e entrar na pasta "rmds".

Instalação e interface

Uma linguagem de programação é um método padronizado para comunicar instruções para um computador e o R é uma linguagem de programação especializada na comunicação de instruções para a manipulação, cálculo e visualização de dados. Formalmente o R é definido como uma linguagem e ambiente de programação estatística e produção de gráficos. Essa definição inclui o termo *ambiente* para indicar que o R é um sistema flexível, coerente e planejado; não apenas um conjunto de ferramentas estatísticas. Por outro lado, o R é um software gratuito, de código aberto e extensível. Isso quer dizer que não há que pagar para usá-lo (gratuito), a implementação das suas funções está disponível (código aberto), e os usuários podem acrescentar novas funções (extensível). A expressão código aberto indica que as instruções (código fonte) executadas por cada função são disponibilizadas sob uma licença de código aberto na qual o direito autoral fornece o direito de estudar, modificar e distribuir essas instruções de graça para qualquer um e para qualquer finalidade.

Instalação

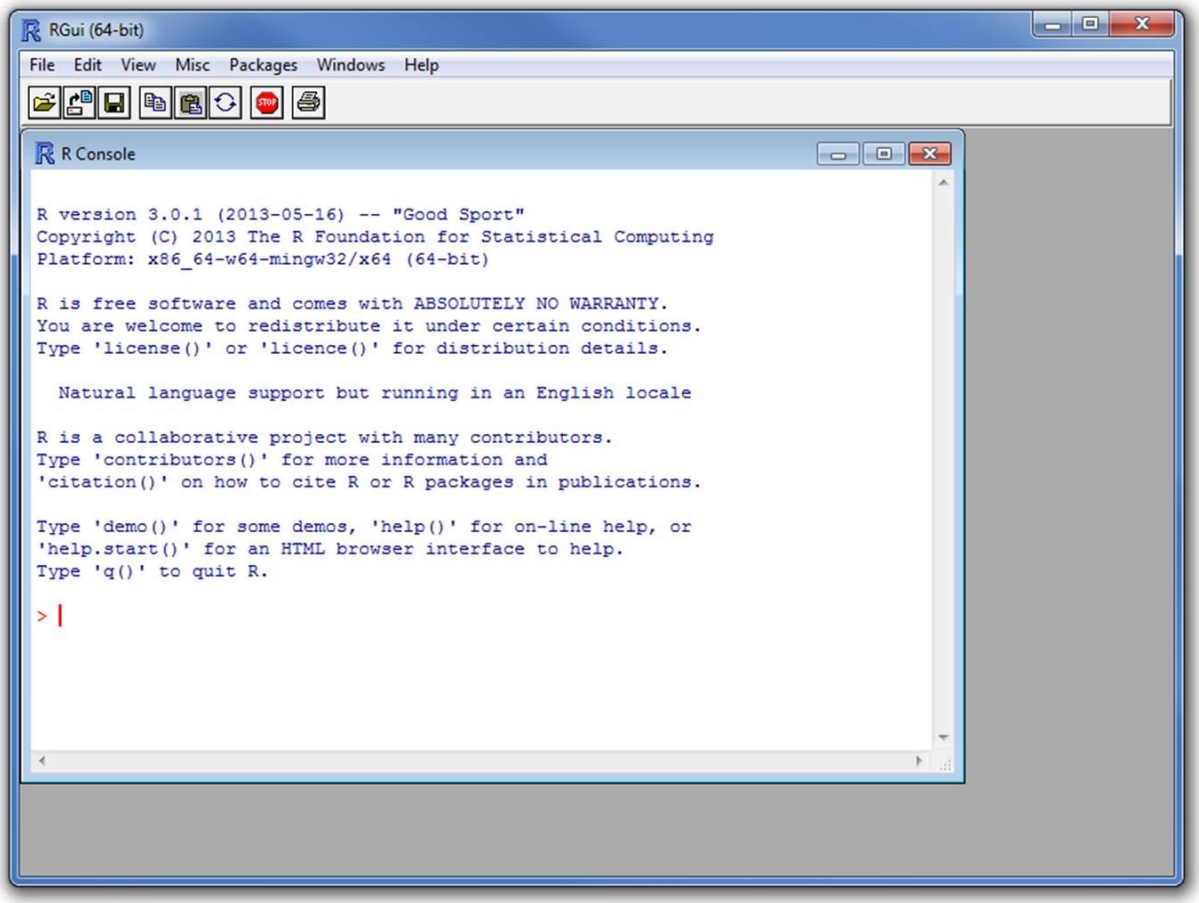
Na página inicial do R, o primeiro parágrafo contém o link CRAN mirror. Ao seguir o link aparacerão outros links agrupados por países. Siga o link geograficamente mais próximo de você. Na página resultante aparecerão links para descarregar o R. Descarregue o R para o seu sistema operacional. Clique duas vezes no arquivo descarregado e siga as instruções. Embora seja possível escolher um língua de instalação diferente do inglês, eu recomendo a instalação em inglês. Por quê? Como veremos mais para frente, a execução de comandos pode produzir mensagens de erro e uma forma de encontrar a solução para um erro é copiar e colar e mensagem no Google. Ao fazermos a búsqueda em inglês, quase sempre teremos mais sucesso.

RStudio

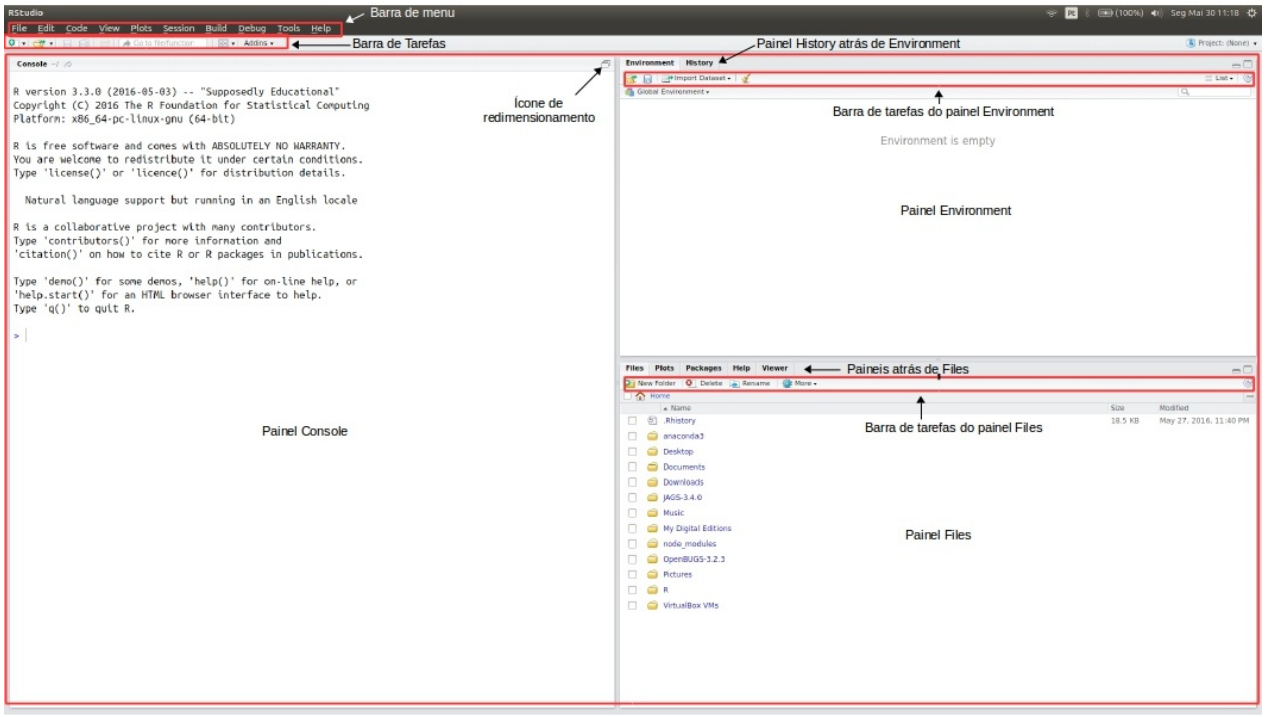
O RStudio é uma companhia que oferece produtos gratuitos e pagos baseados no R. Os produtos gratuitos dão acesso a toda funcionalidade do R, e um desses, o RSudio Desktop, é uma interface que oferece recursos para facilitar o uso do R. No contexto da programação, produtos como o RStudio Desktop são conhecidos como IDE, pelas siglas em inglês de *ambiente de desenvolvimento integrado*. Para descarregar o RStudio vá na seção **Installers for supported platforms**, escolha a opção para o seu sistema operacional, clique duas vezes no arquivo descarregado e siga as instruções.

Interface

Ao abrir o R no lugar do RStudio a interface será semelhante à seguinte:

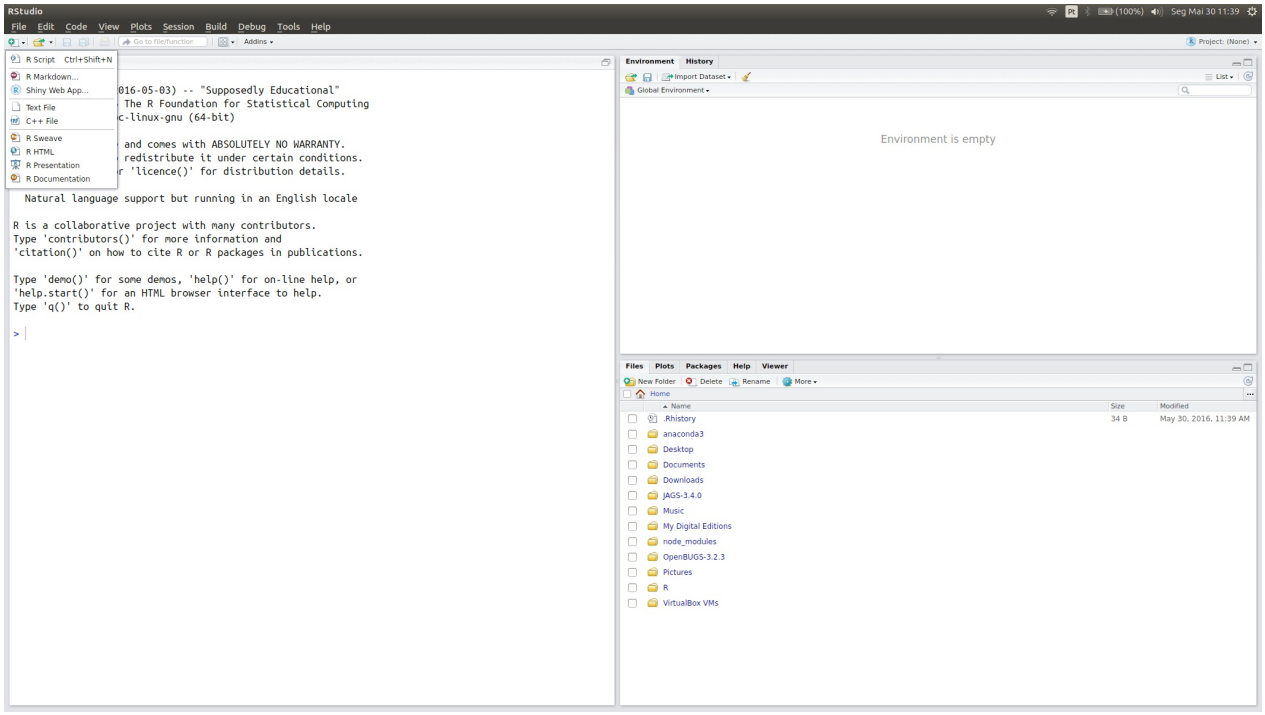


Porém, a interface que usaremos não é essa e sim a do RStudio (ao abrir o programa, não confundir o ícone do RStudio com o do R).



A interface do RStudio, está composta por uma barra de menu, uma barra de tarefas, e painéis. A barra de menu contém janelas (*File, Edit, Code*, etc.) que ao ser clicadas oferecem múltiplas funcionalidades. A barra de tarefas está composta por ícones que executam funções frequentes (alguns abrem uma janela de opções) e quando o mouse é colocado sobre um desses ícones, aparece uma mensagem de texto que descreve a funcionalidade. Os painéis são os espaços que ocupam a maior parte da interface, e cada um, excetuando o painel *Console*, tem sua própria barra de tarefas. Cada painel também tem ícones no canto superior direito, que servem para colapsar ou redimensionar o painel.

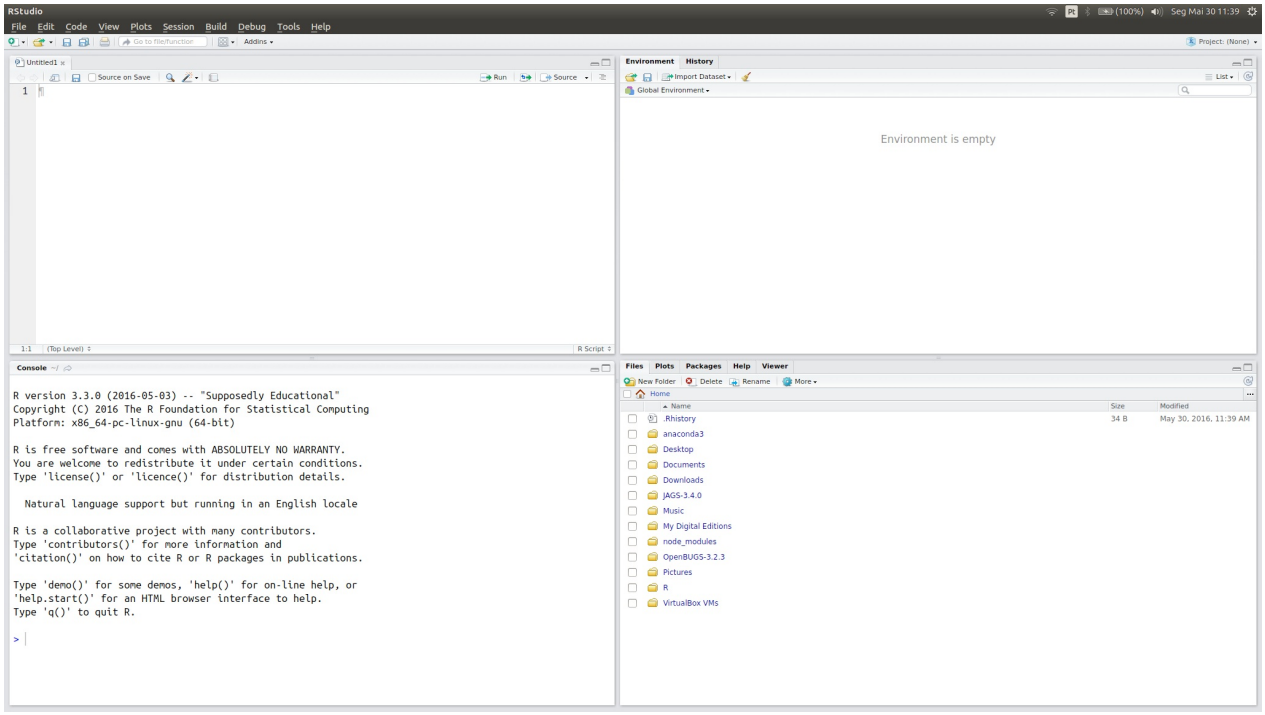
Como veremos em breve, o R permite escrever e executar códigos (instruções) diretamente na consola. Entretanto, os códigos podem ser escritos em arquivos com extensão ".R" e enviados à consola para serem executados. A vantagem disso é que os códigos podem ser guardados e executados quando necessário, sem necessidade de reescrevê-los. Esses arquivos se conhecem como *scripts*. Para abrir um novo script, basta clicar no primeiro ícone da barra de tarefas e clicar na primeira opção, *R Script*.



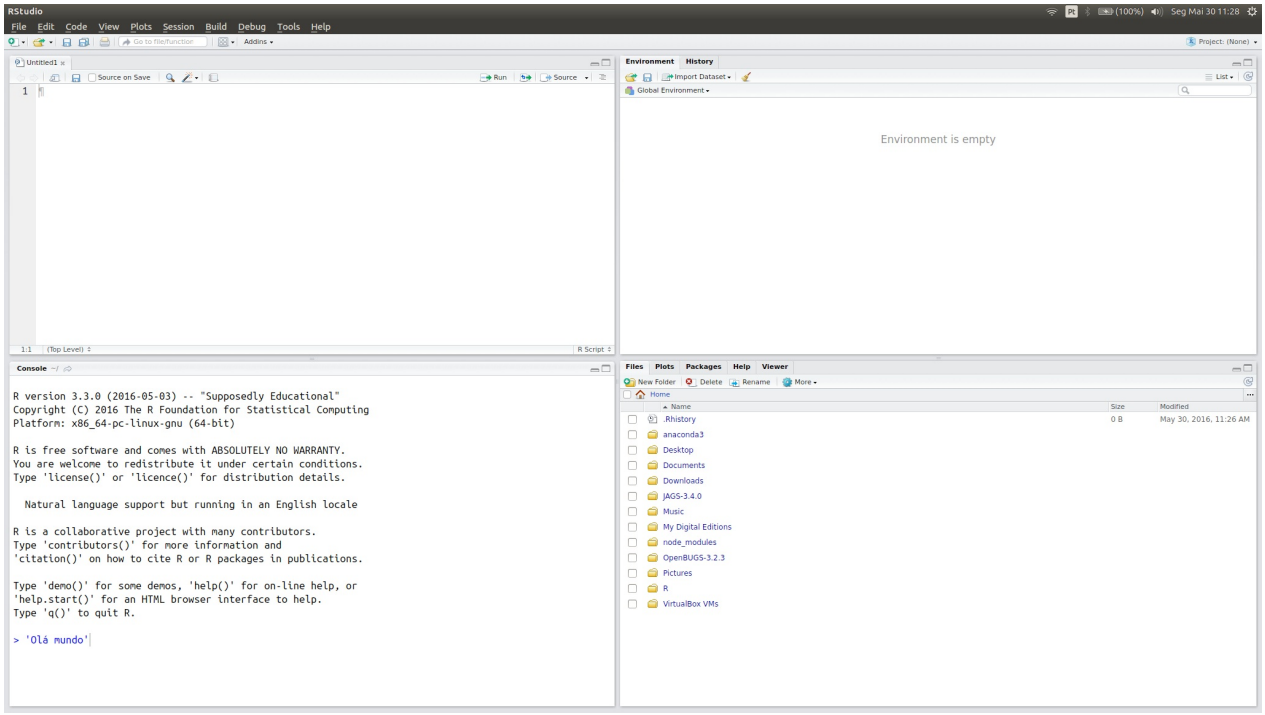
No lado direito dessa opção aparece o atalho de teclado que abre um novo script. Assim, outra forma de abrir um novo script em Windows ou Linux é apertando simultaneamente as teclas *Ctrl*+*Shift*+*N* (o símbolo "+" não deve ser incluído, apenas idnica que as três teclas devem ser apertadas simultâneamente). Ainda, a janela *File* da barra de menu contém a opção *New File* que é equivalente ao primeiro ícone da barra de tarefas.

O anterior exemplifica dois fenômenos comuns. Primeiro, a maioria das opções das barras de menu e de tarefas mostram um atalho de teclado para executar a funcionalidade em questão. Inicialmente, é trabalhoso lembrar os atalhos, mas com o tempo torna-se mais rápido e prático trabalhar com os mesmos (lembrar os atalhos é um bom investimento!). Segundo, geralmente há mais de uma forma para obter o mesmo resultado.

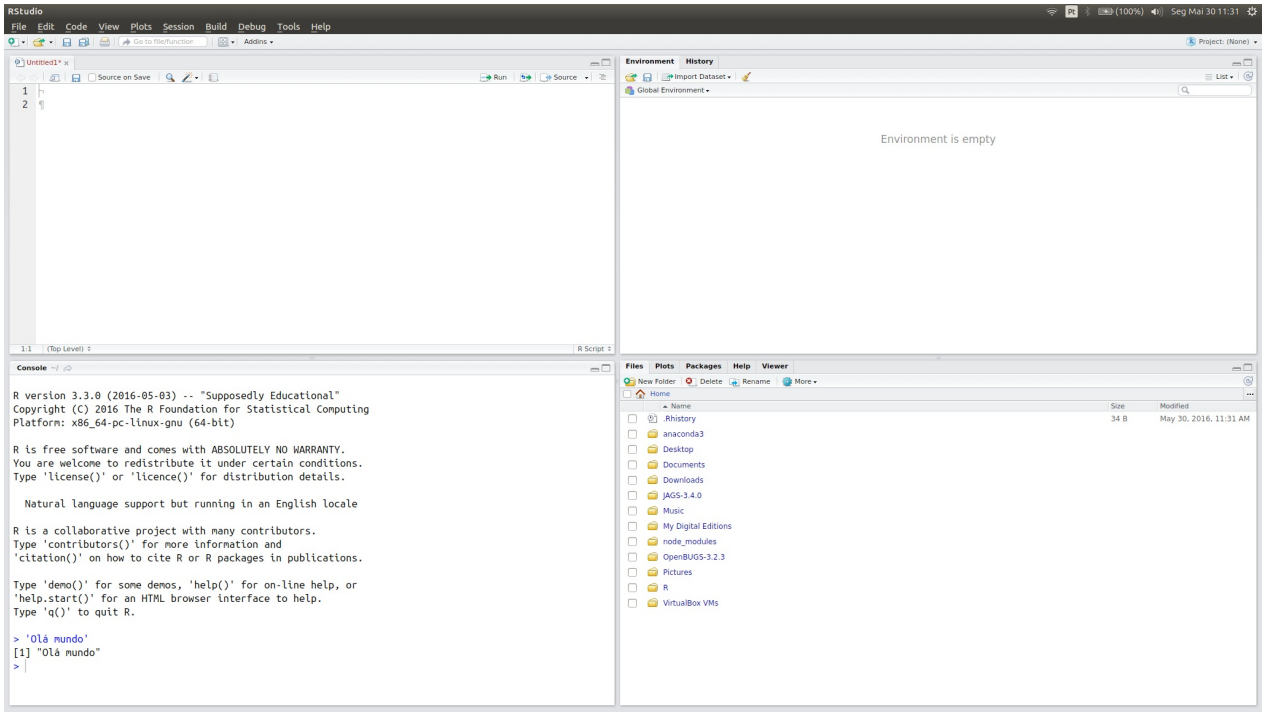
Qualquer uma das possibilidades mencionadas abrirá um painel para mostrar o novo script.



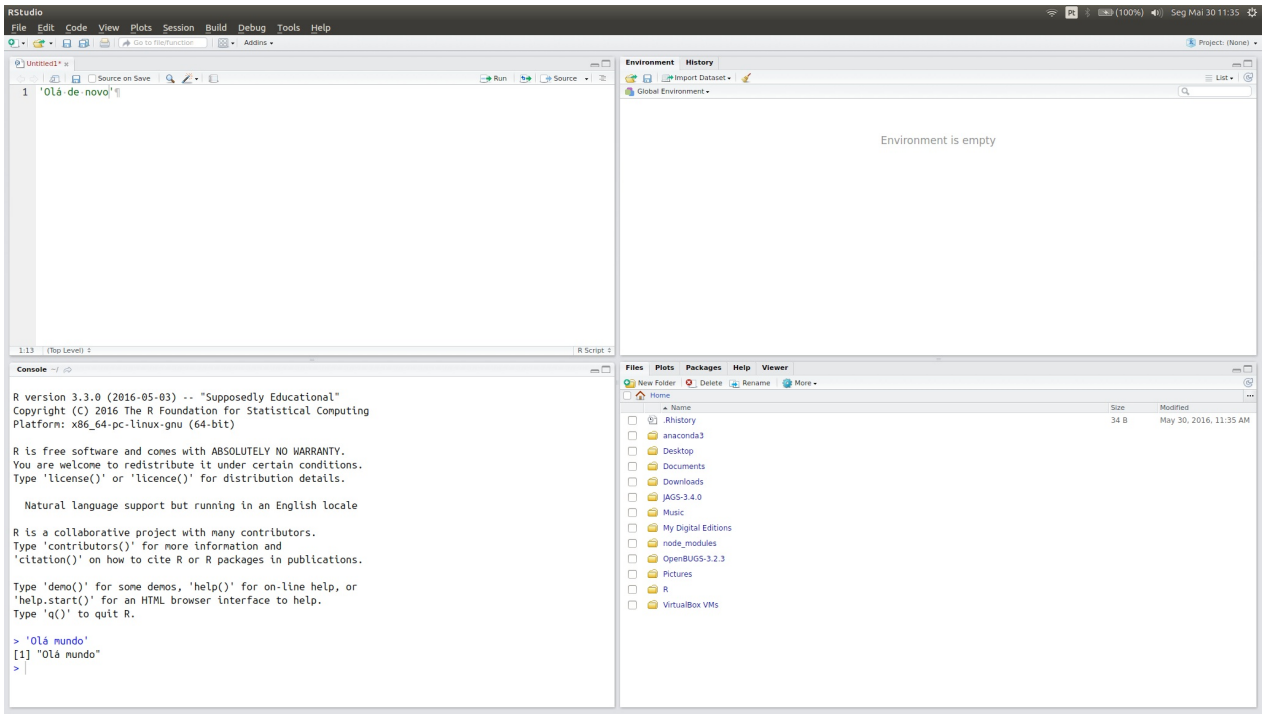
Agora estamos prontos para continuar explorando a interface do RStudio e ao mesmo tempo começarmos a usar o R! Ao escrever `'Olá mundo'` na consola



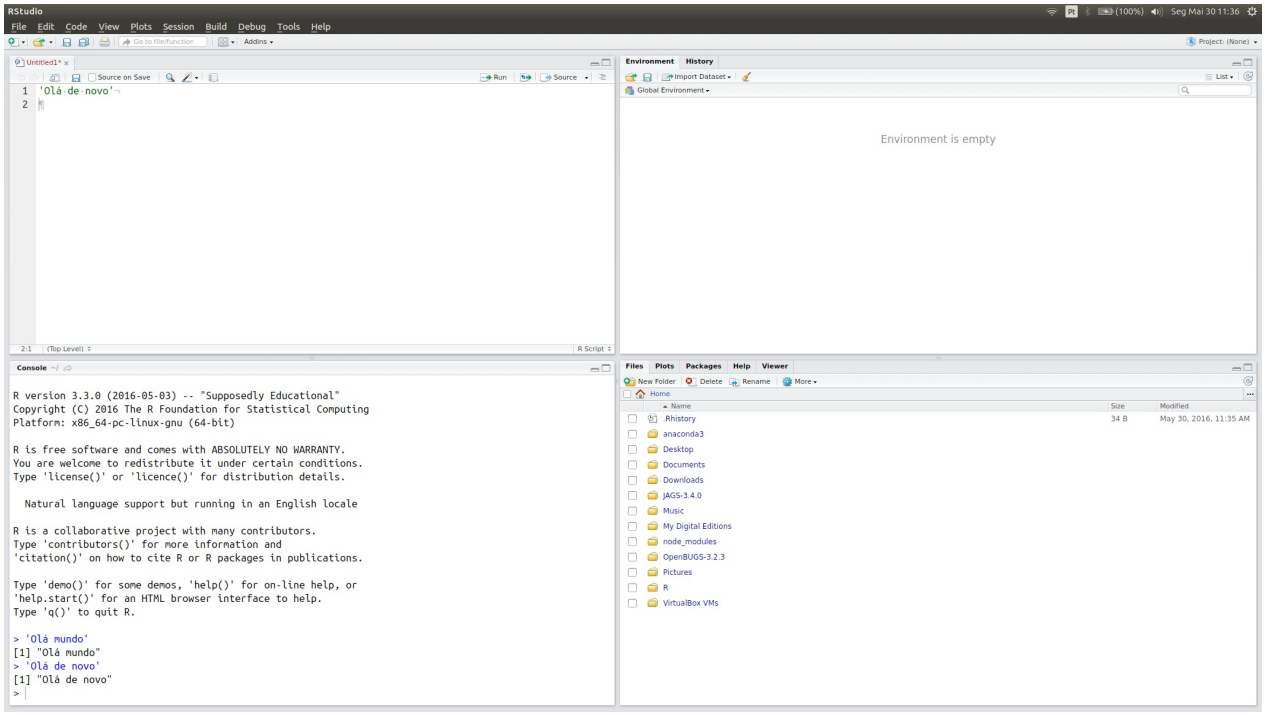
e depois apertar *Enter*, o comando `'Olá mundo'` é executado.



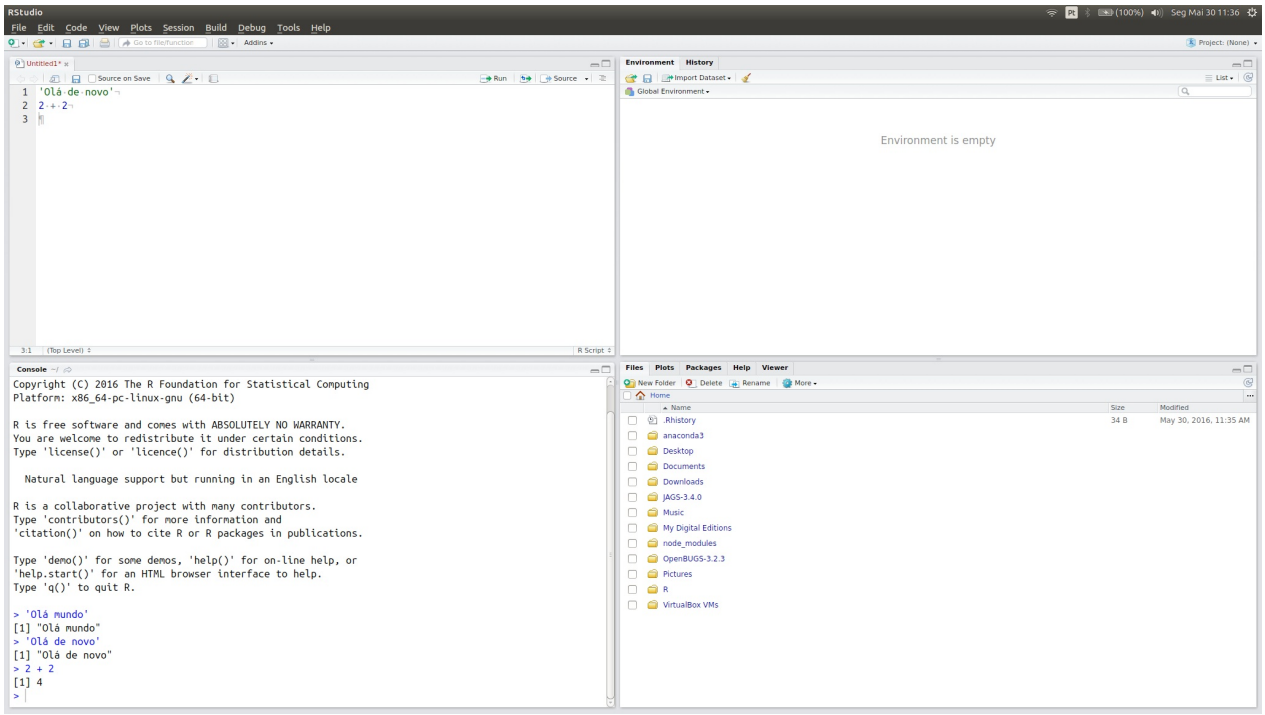
Se o comando é escrito no script



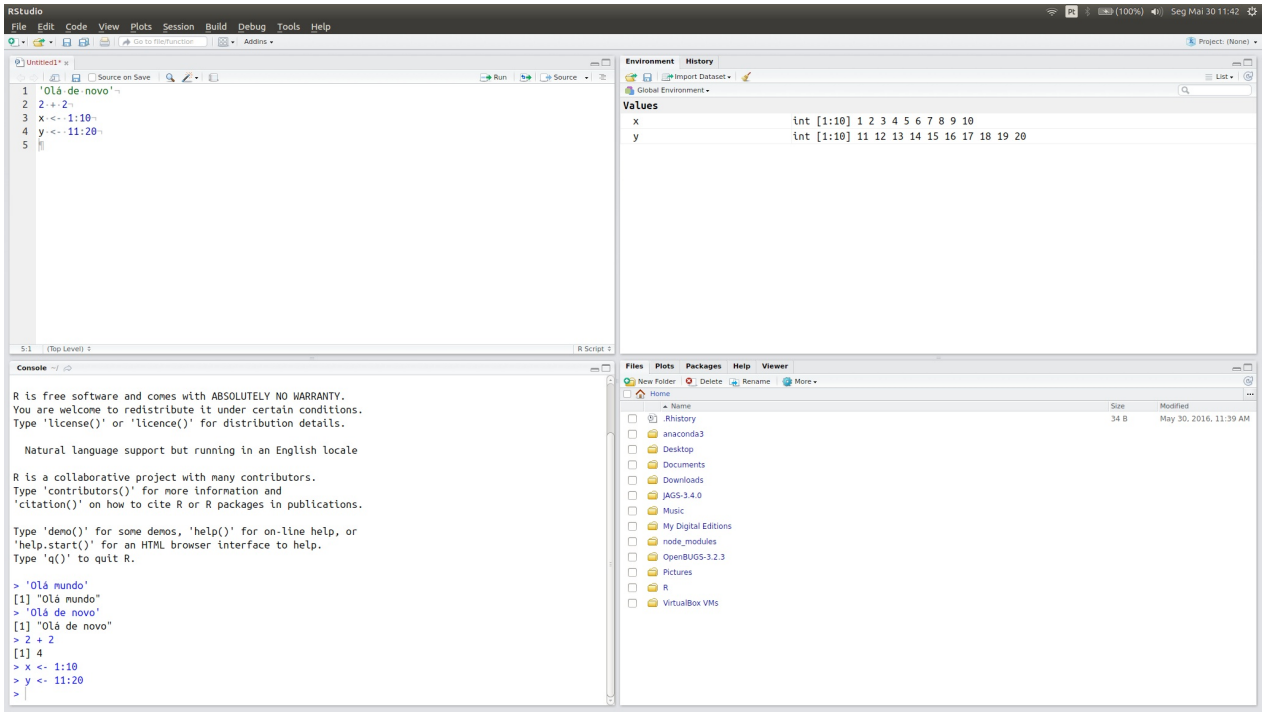
a execução é feita posicionando o cursor na linha que contém o comando e apertando *Ctrl+Enter* (no Mac, Ctrl é usualmente substituído por Command).



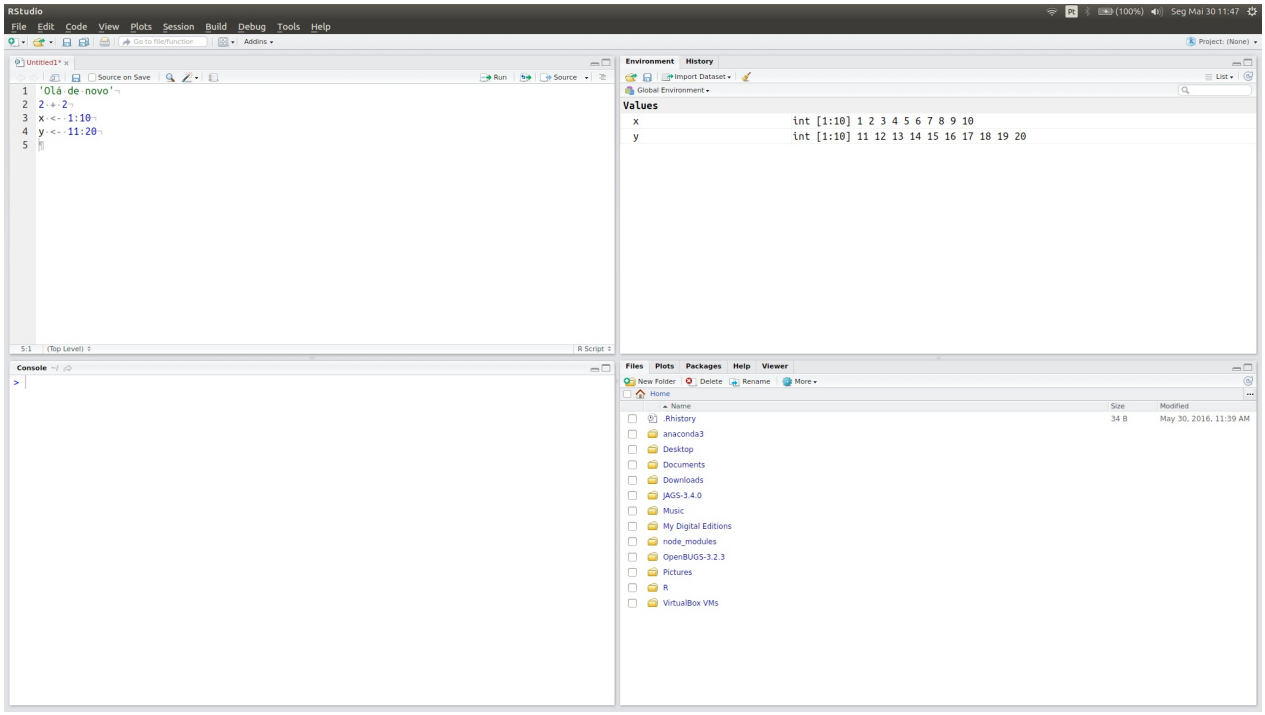
De forma semelhante, podemos executar uma operação matemática.



Um tipo de operação importante é a desingação (assign). No exemplo a seguir, o comando `1:10` cria uma sequência de 1 a 10 e a armazena em um objeto nomeado `x`. O operador de designação `<-` armazena o que está à direita em um objeto que recebe o nome que está à esquerda (os nomes dos objetos podem conter, mas não começar com números). Assim, o comando `y <- 21:20` armazena a sequência 11 a 20 no objeto `y`.



Os dois comandos anteriores criaram dois objetos e todos os objetos são listados no painel *Environment*. Para imprimir o conteúdo de um objeto basta executar o nome. Vejamos esse comportamento com os objetos `x` e `y`, mas antes disso, limpemos a consola com o atalho *Ctrl+L*.




```
> x <- 1:10
> y <- 11:20
```

```
> x
```

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

```
> y
```

```
 [1] 11 12 13 14 15 16 17 18 19 20
```

A criação de um objeto e a impressão do seu conteúdo podem ser feitas simultaneamente se o comando é colocado entre paréntesis.

```
> (z <- 5:1)
```

```
[1] 5 4 3 2 1
```

Uma forma de imprimir vários objetos simultaneamente é colocando os nomes na mesma linha, mas separados por ";".

```
> x; y; z
```

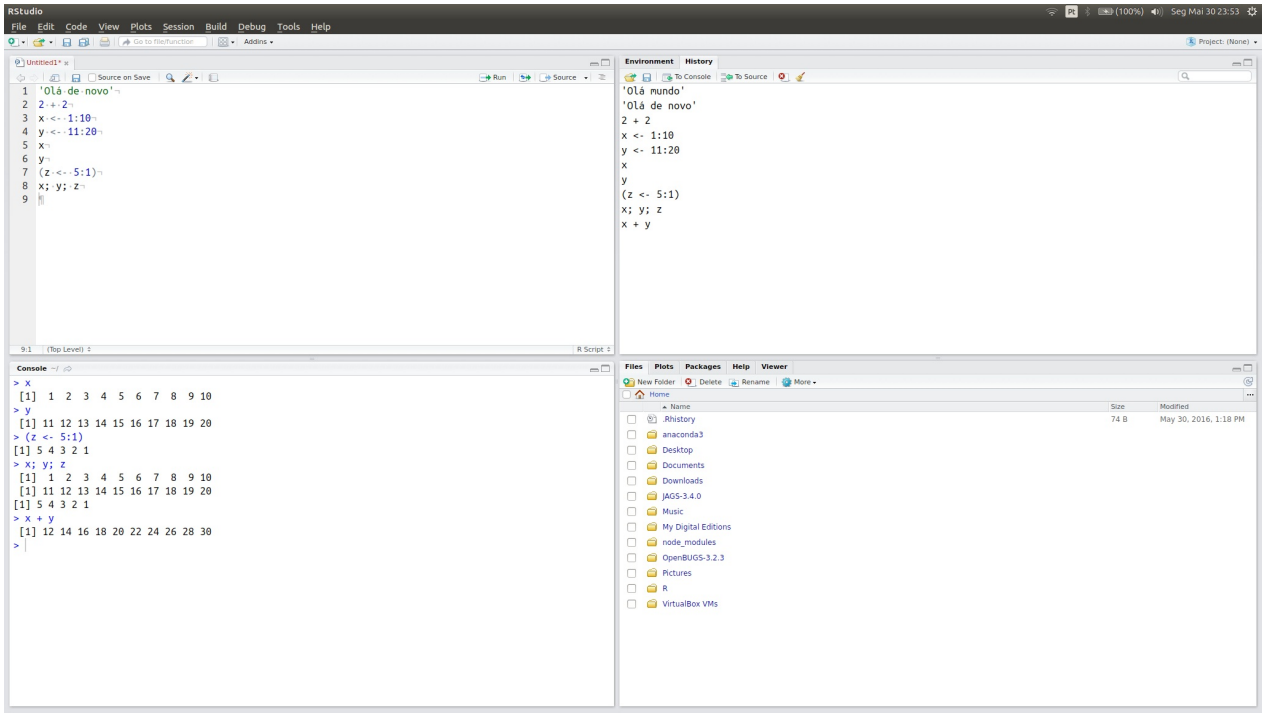
```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
 [1] 11 12 13 14 15 16 17 18 19 20
```

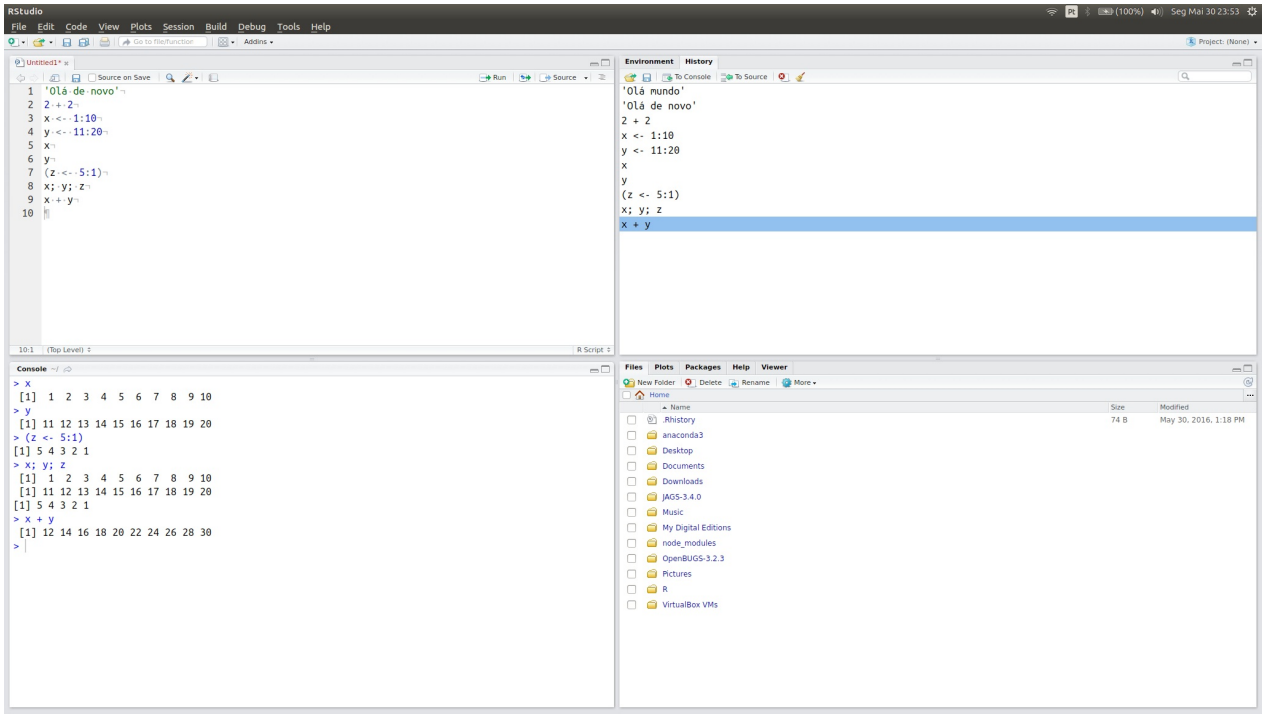
```
[1] 5 4 3 2 1
```

Antes de explorarmos o painel *History*, vamos aprender umas poucas dicas. *Ctrl+1* posiciona o cursor no script e *Ctrl+2* posiciona o cursor na consola (vale a pena lembrar que existem muitos outros atalhos úteis e as janelas das barras de menu e de tarefas os mostram). Com o cursor na consola e apertando *Ctrl* mais a tecla com a flecha apontando para acima, abre-se uma janela com o histórico de comandos executados; ao digitar um ou vários caracteres, por exemplo `nom` e depois apertar *Ctrl* mais a flecha para acima, a janela só mostrará o histórico de comandos que começam com `nom` (se tivéssemos executado comandos para criar os objetos `nome_das_ruas` e `nome_dos_bairros` e nenhum outro comando tivesse começado por `nom`, so apareceriam esses dois comandos). O anterior ajuda a encontrar e reexecutar comandos prévios.

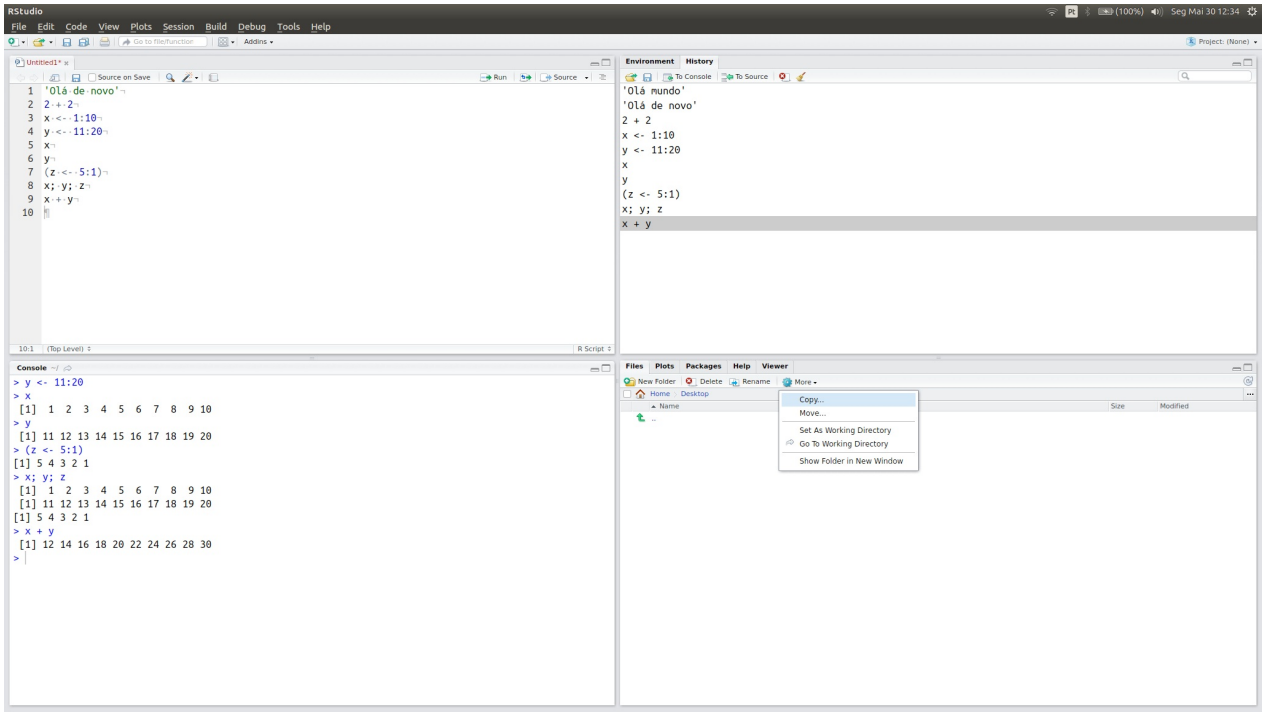
Se executarmos o comando `x + y` e depois cliquamos na aba do painel *History*, veremos esse e todos os comandos executados até o momento.



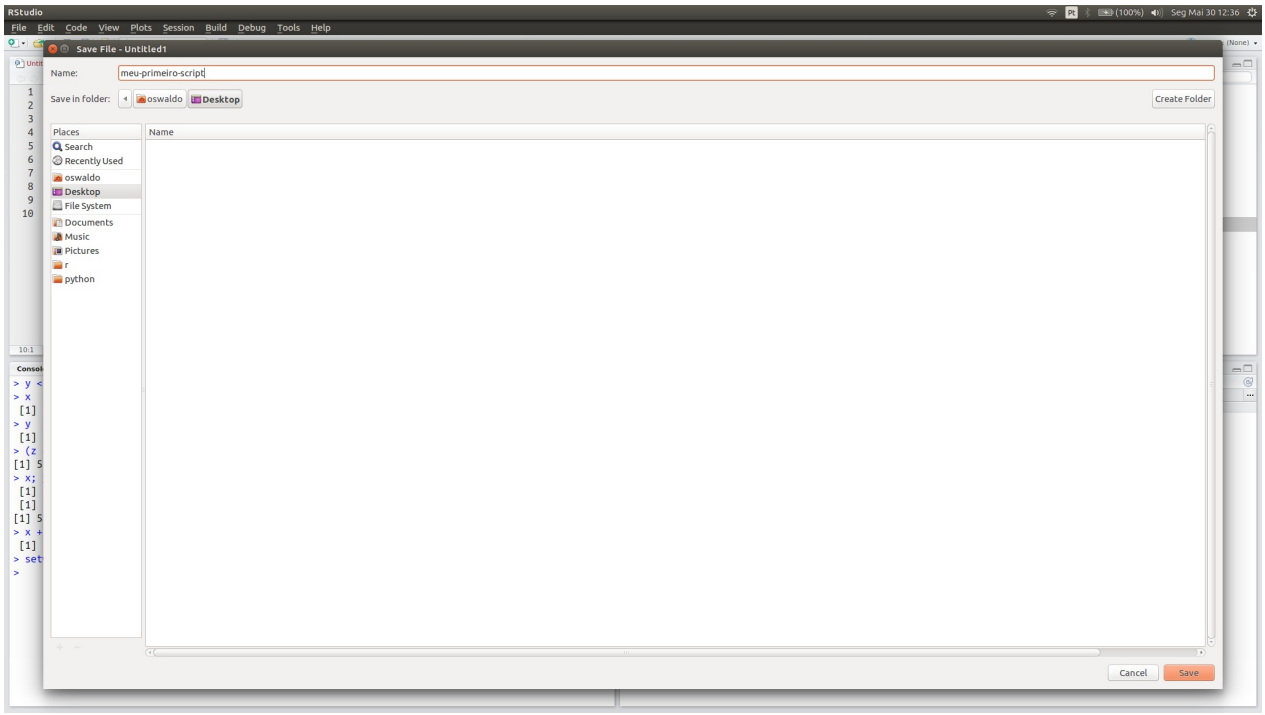
A seleção de uma ou mais linhas do histórico junto com o uso do ícone *To Console*, envia as linhas respectivas à consola. O ícone *To Source* envia ao script.



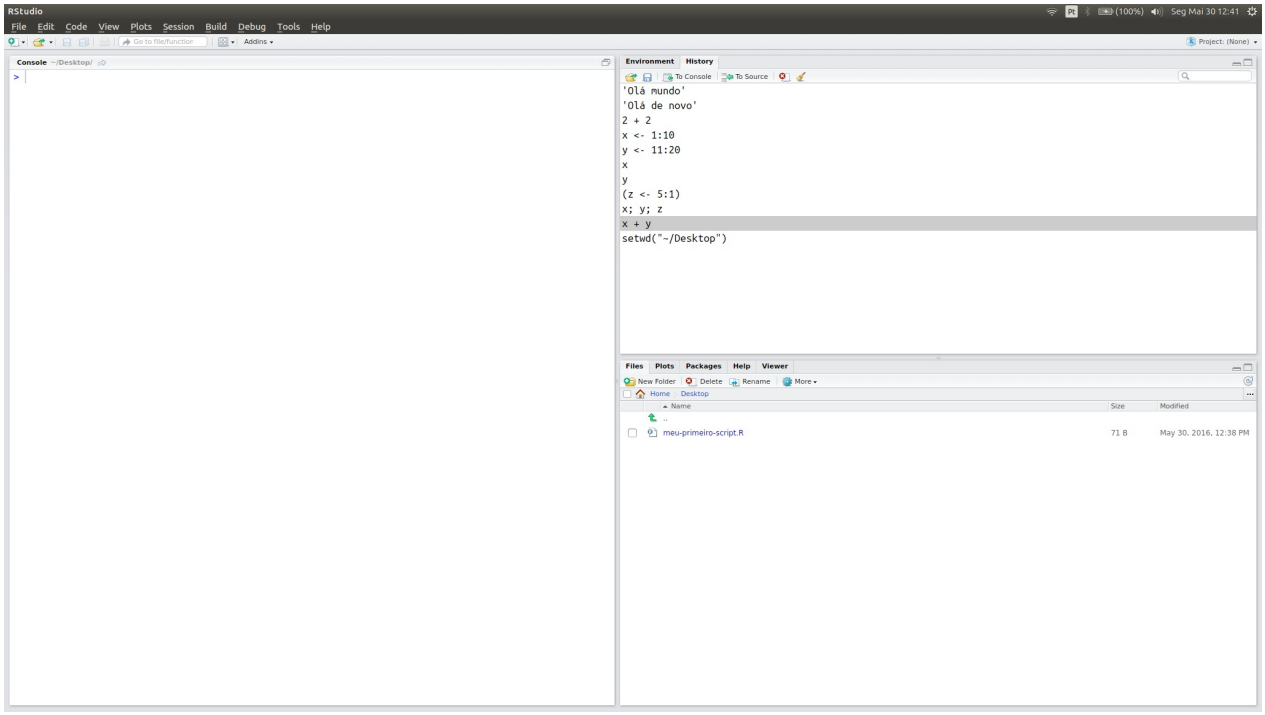
O painel *Files* mostra os diretórios do computador. No caso de meu computador, vou entrar no diretório *Desktop* que está vazio, clicar no ícone *More* e depois na opção *Set as Working Directory*.



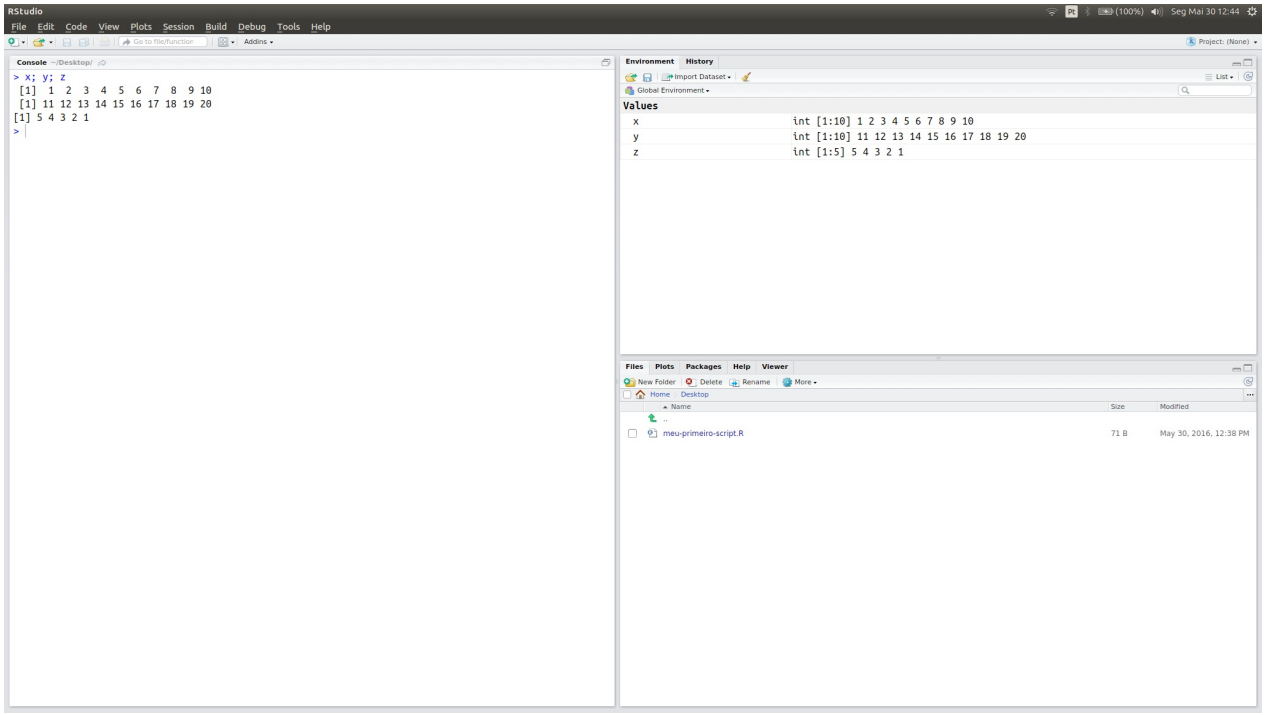
Ao fixar o diretório de trabalho (*Working Directory*) no *Desktop*, a importação de arquivos presentes nesse diretório ou a exportação para o mesmo será mais fácil. Por exemplo, para salvar o meu script no diretório *Desktop*, só preciso usar o atalho *Ctrl+S* (ou o ícone de salvar na barra de tarefas) e digitar algum nome para o arquivo, digamos, *meu-primeiro-script*.



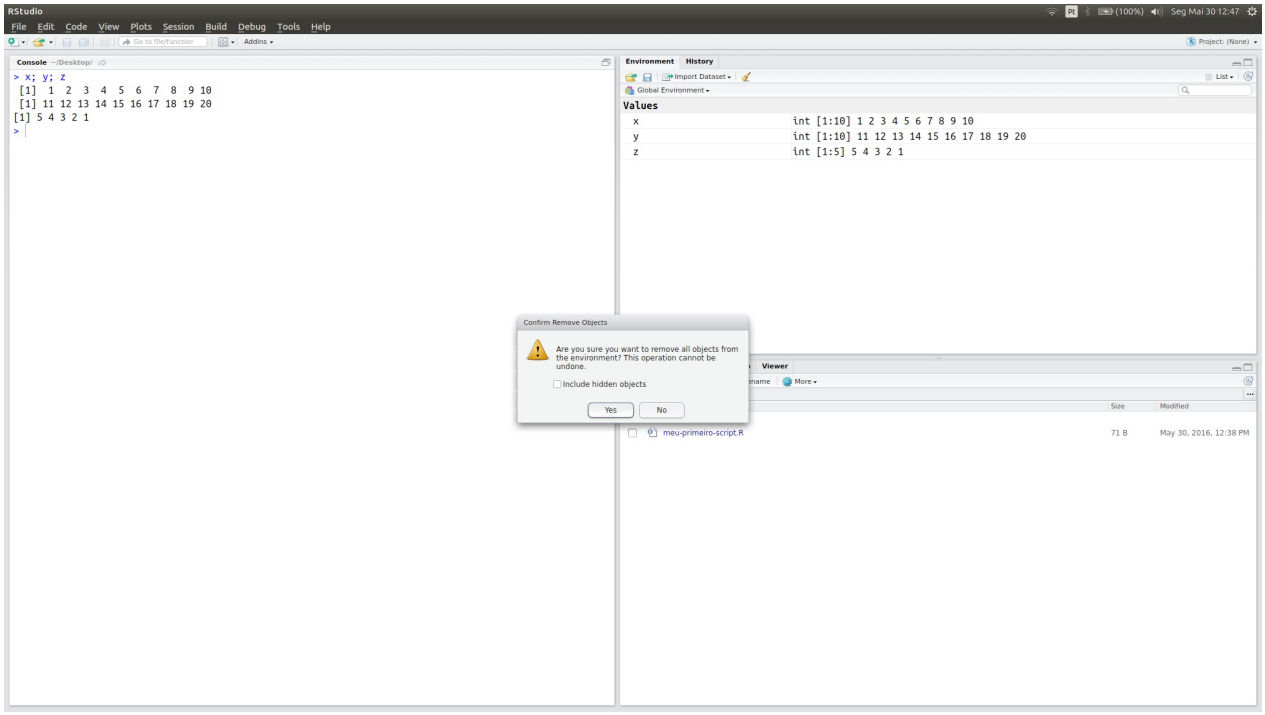
Após isso, o arquivo aparecerá no *Desktop* e seu fechamento não apagará os objetos criados.



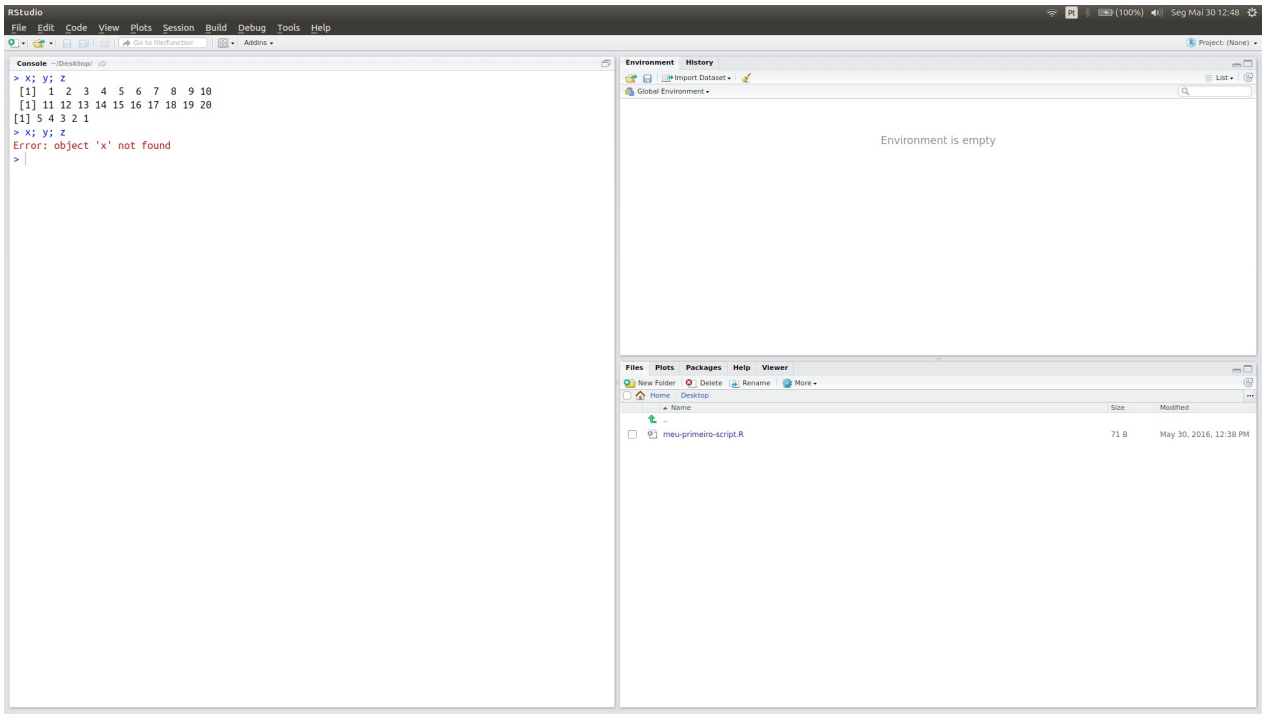
De fato, podemos ir à consola, navegar no histórico e escolher e executar o comando `x; y; z` .



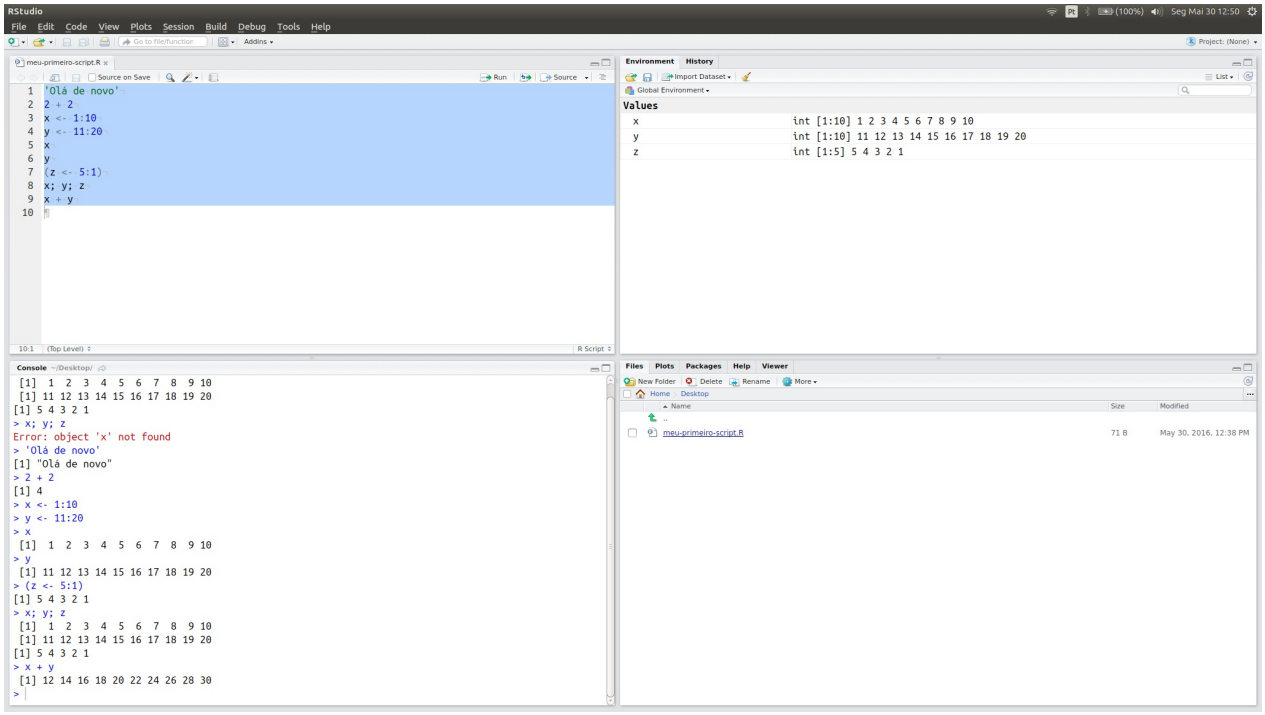
Para apagar os objetos é preciso clicar no ícone da vassoura na barra de tarefas do painel *Environment*



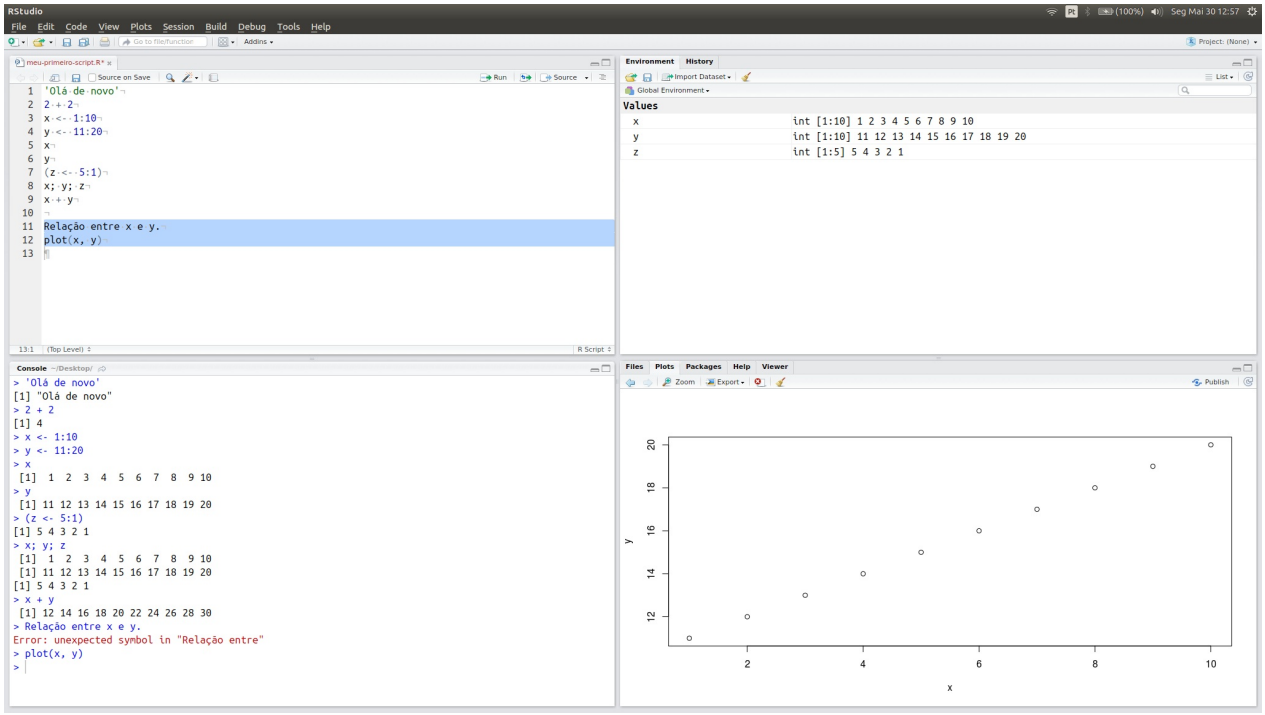
e como é de se esperar, as tentivas de usar os objetos apagados gerarão erros.



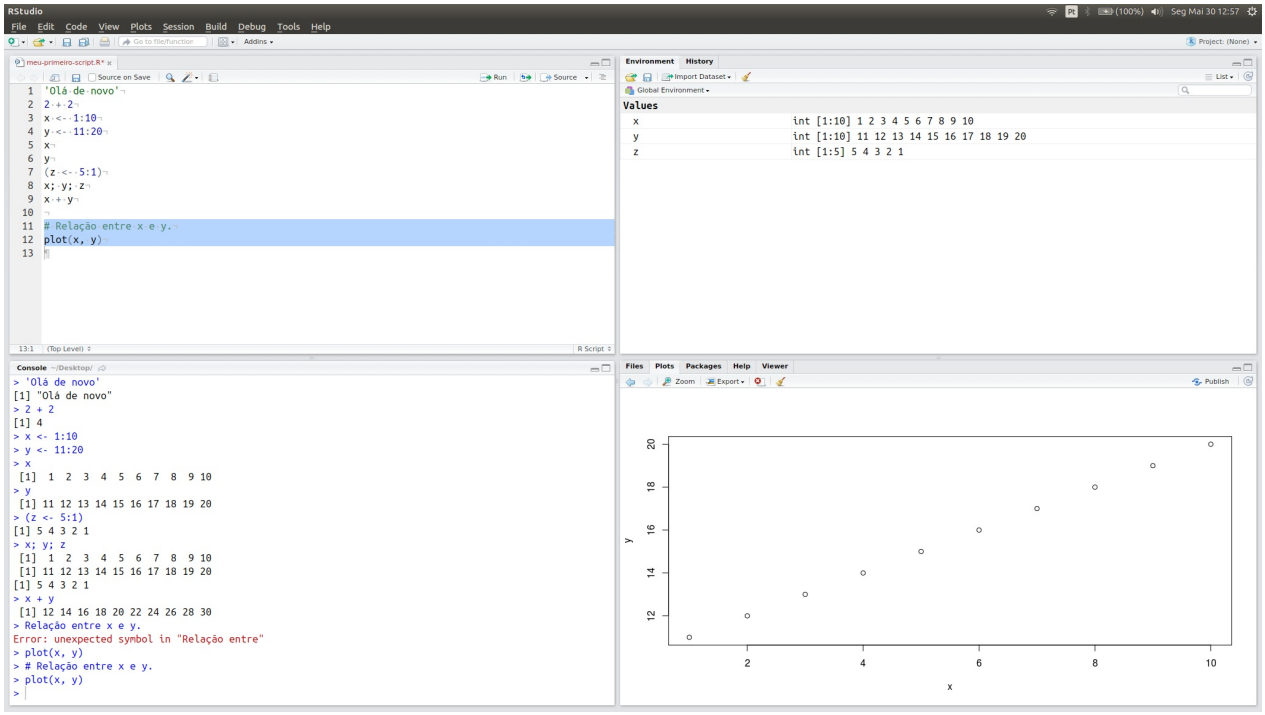
Entretanto, o anterior não é problema, pois ao abrir o script (clicando nele no painel *Files*), selecionar todo, e executar o conteúdo, todos os objetos serão recriados.



Ao usar `x` e `y` como argumentos da função `plot`, o gráfico resultante aparecerá no painel *Plots*.



Além do gráfico, a figura acima mostra que a execução da linha 11 do script gerou um erro. Porém, se essa linha começa com um ou mais "#", vira um *comentário* cujo conteúdo não é interpretado, apenas imprimido.



Os comentários são úteis para documentar um determinado código, pois embora no momento de criarmos o script possa ser desnecessário, ao revê-lo depois de um tempo a razão de ter usado o código pode não ser clara (obviamente um código simples como o anterior dispensa de comentários).

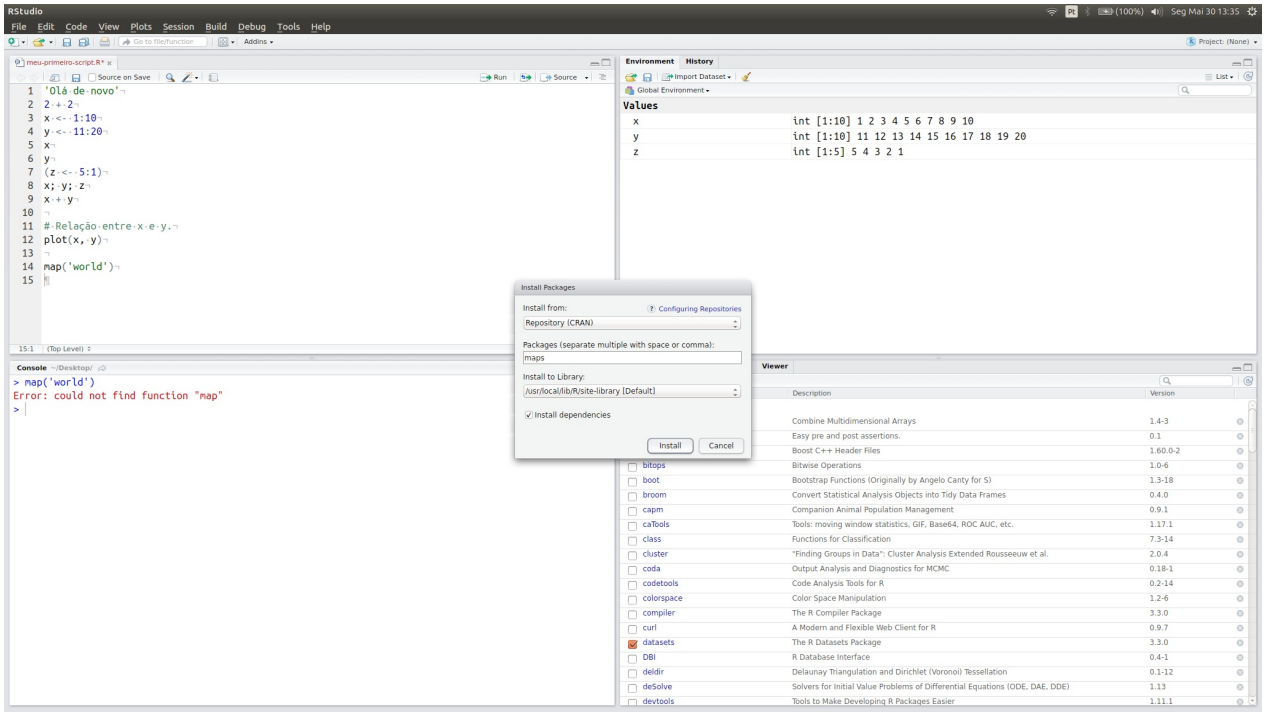
O R organiza suas funções em *pacotes e para poder usá-las, os pacotes devem estar* carregados*. Alguns pacotes vêm com a instalação do R e são carregados automaticamente quando o R é iniciado; portanto, suas funções podem ser usadas diretamente como no caso do pacote* graphics* que contém a função `plot` usada anteriormente. Outros pacotes vêm com a instalação do R, mas não são carregados automaticamente e portanto, devem ser carregados antes de usar suas funções. Adicionalmente, existem pacotes que não vêm com a instalação do R, mas podem ser instalados e carregados. Nesse caso, a instalação só precisa ser feita uma vez, mas o carregamento deve ser feito cada vez que se inicia o R.

Usando o pacote *maps* como exemplo, podemos ver que a tentativa de usar `map` - uma das suas funções - sem carregar o pacote, gerará um erro se o pacote não é previamente carregado.

```
> map('world')
```

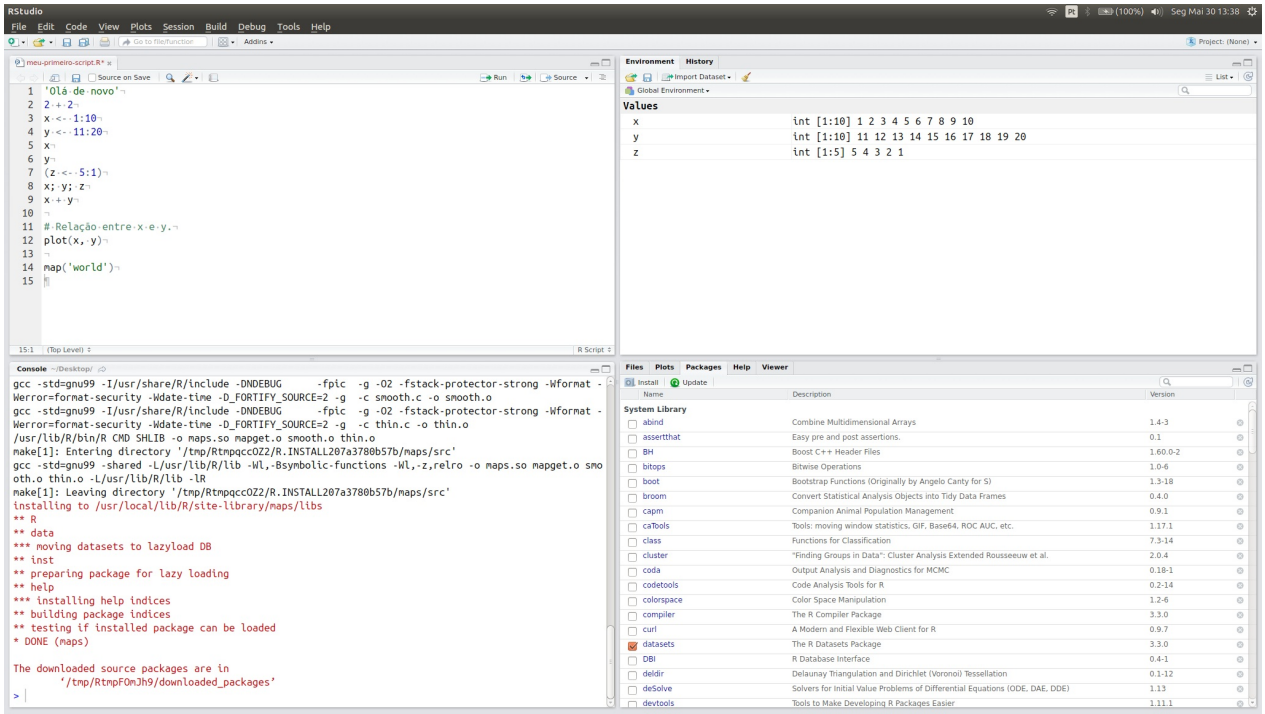
```
Error in eval(expr, envir, enclos): could not find function "map"
```

Para instalar o pacote podemos executar o comando `install.packages('maps')` ou ir no painel *Packages*, clicar no ícone *install*, digitar *maps* na janela aberta, ativar a opção *Install dependencies*, e clicar *Install*.



O painel *Packages* também apresenta uma lista dos pacotes instalados e indica com o box da esquerda quais estão carregados (na imagem acima o *datasets* está carregado).

Se instalação for bem sucedida, veremos que na consola as últimas linhas da mensagem de instalação são *The downloaded source packages are in ...*.



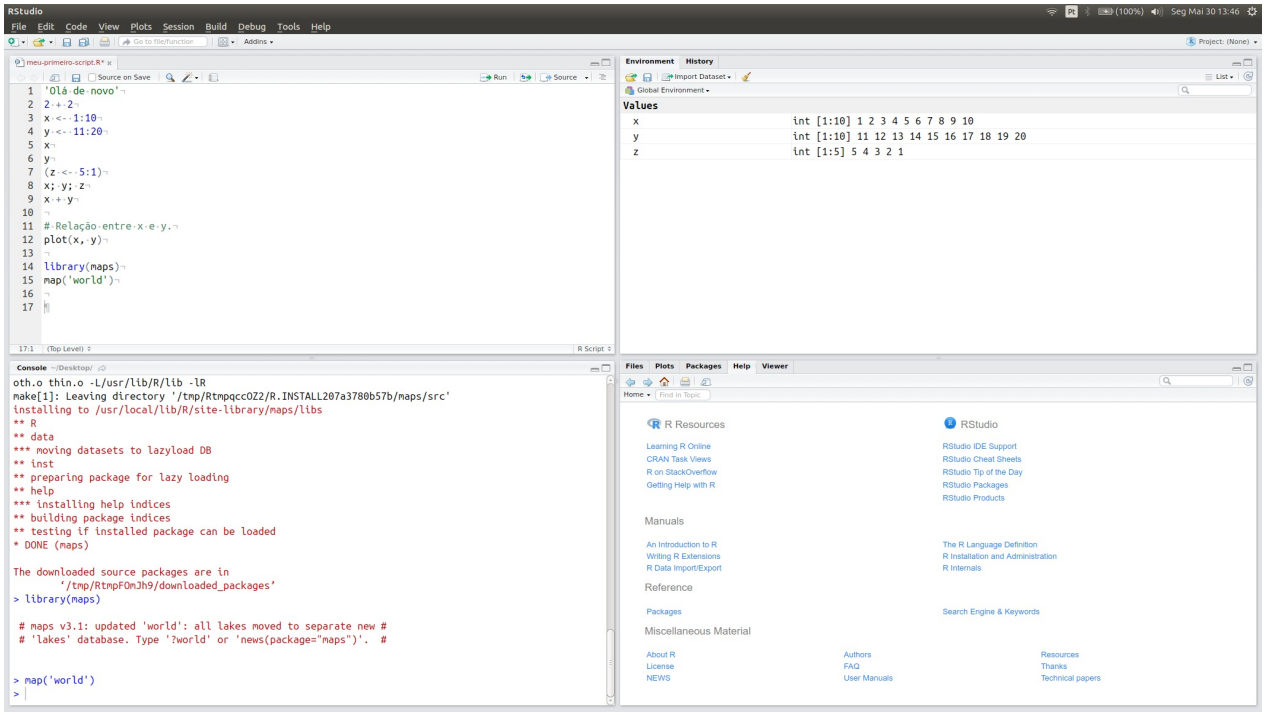
Após a instalação, o pacote deve ser carregado com a função `library`, antes de executar a função `map`.

```
> library(maps)
```

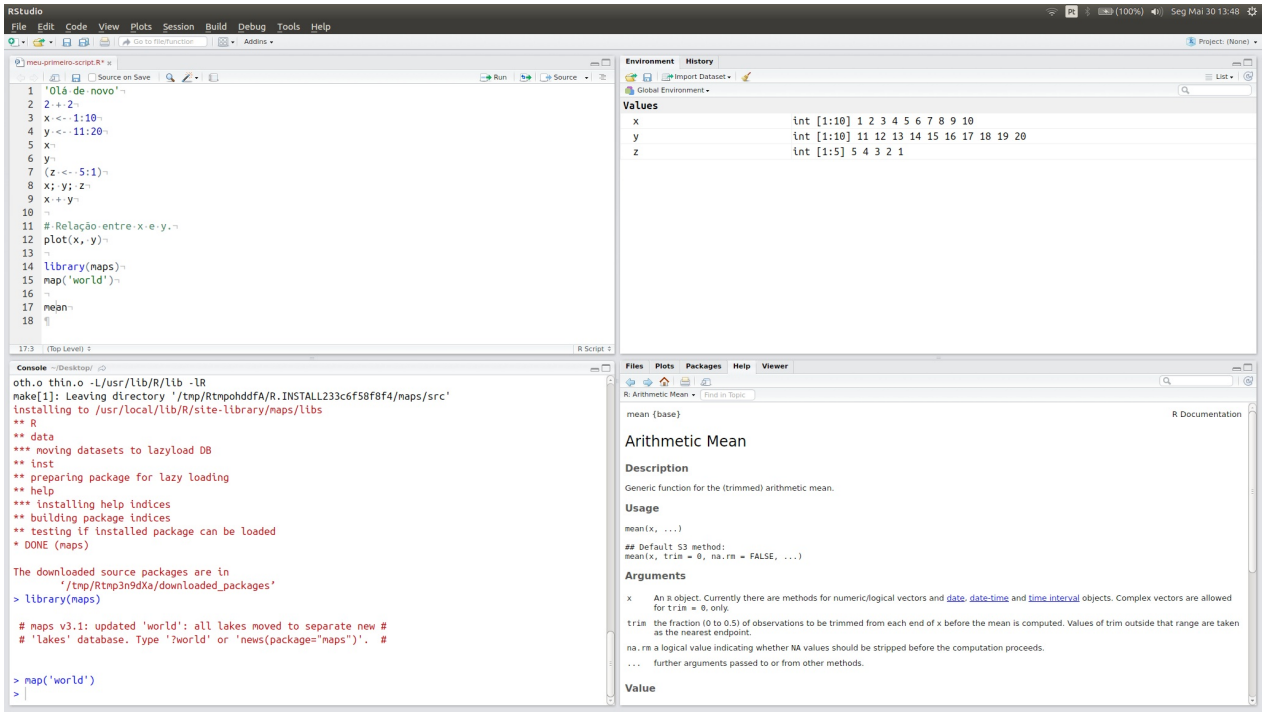
```
 # maps v3.1: updated 'world': all lakes moved to separate new #
 # 'lakes' database. Type '?world' or 'news(package="maps")'.  #
```

```
> map('world')
```

Deixando de lado os pacotes, podemos ver que o painel *Help* oferece uma série de materiais na sua página de início (ícone com o símbolo de uma casa).



Adicionalmente, a maioria das funções possui uma página de ajuda que pode ser visualizada nesse painel. Tomemos como exemplo a função `mean`. Ao digitar, `help(mean)` ou `?mean`, aparecerá a página de aujuda. Também podemos digitar `mean` e com o cursor no começo, no final ou sobre `mean`, apertar *F1*.



As páginas de ajuda geralmente possuem uma descrição, uma seção *Usage* que mostra de forma condensada as diferentes formas de usar a função, uma seção *Arguments* que descreve os argumentos da função, uma seção *Value* que descreve os possíveis resultados gerados pela função, e uma seção *Examples* que apresenta códigos de exemplo (algumas páginas de ajuda tem outras seções). Inicialmente as páginas de ajuda são difíceis de

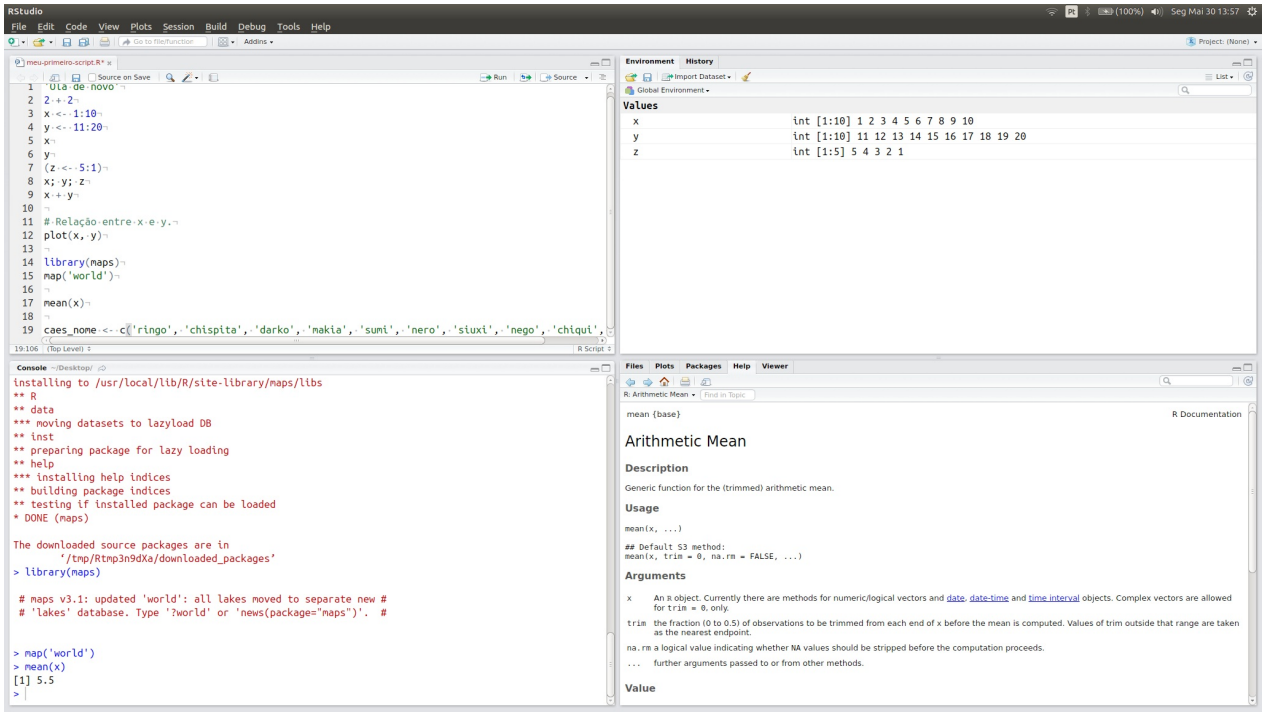
entender porque estão escritas em linguagem técnica. No entanto, com o tempo é possível adquirir familiaridade com esse formato. Além dos recursos mencionados está o Google! Se digitarmos *r mean* no buscador, aparecerão várias páginas incluindo a própria página de ajuda da função. É bom olhar várias páginas, pois umas são mais amigáveis do que outras e combinando as informações de diferentes fontes, fica mais fácil entender o modo de uso.

Nota: o uso de termos de busca adequados é uma das ferramentas mais importantes para o uso proficiente do R e a escolha dos termos é uma habilidade que se aprimora com a prática.

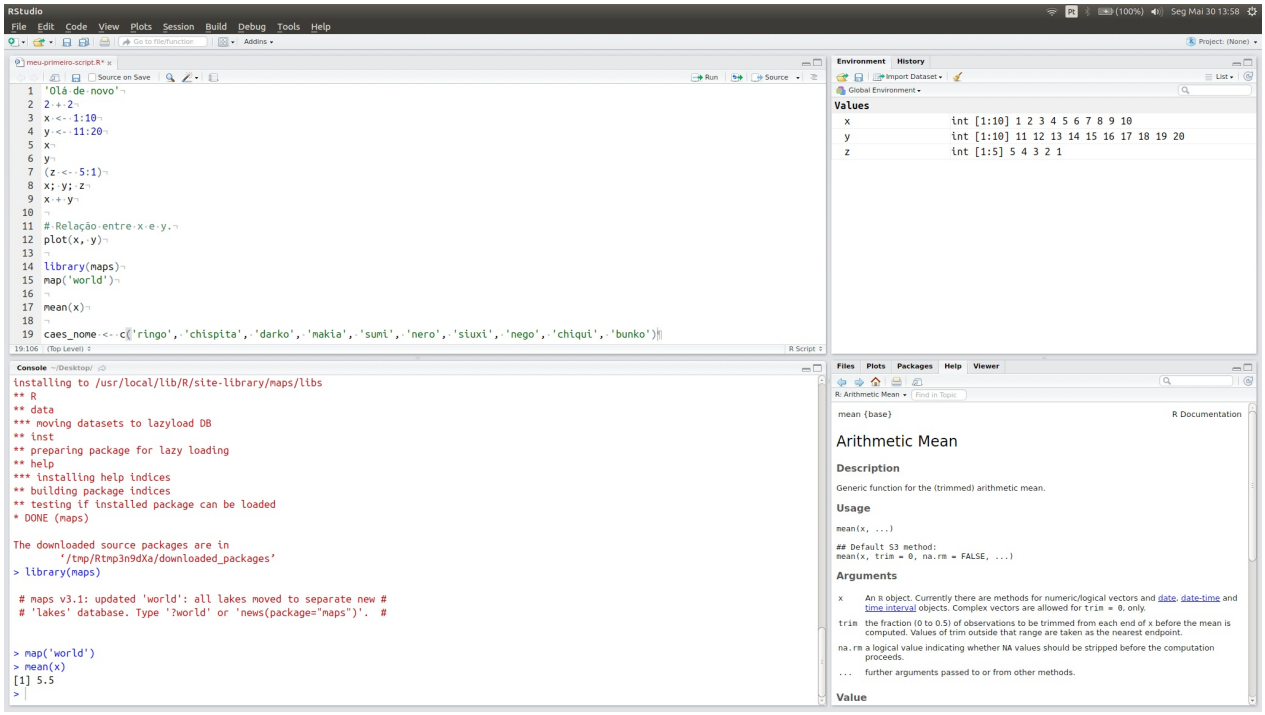
Quando não sabemos exatamente o nome de uma função podemos executar parte do nome precedido por dois sinais de interrogação (`??mea`) para ver todas as páginas de ajuda que contém essa parte. Alternativamente, podemos digitar parte do nome no script ou na consola e apertar a tecla *TAB* para ver diferentes formas de autocompletar.

O painel *Viewer* permite visualizar aplicações web locais, mas está além do propósito da presente introdução ao RStudio.

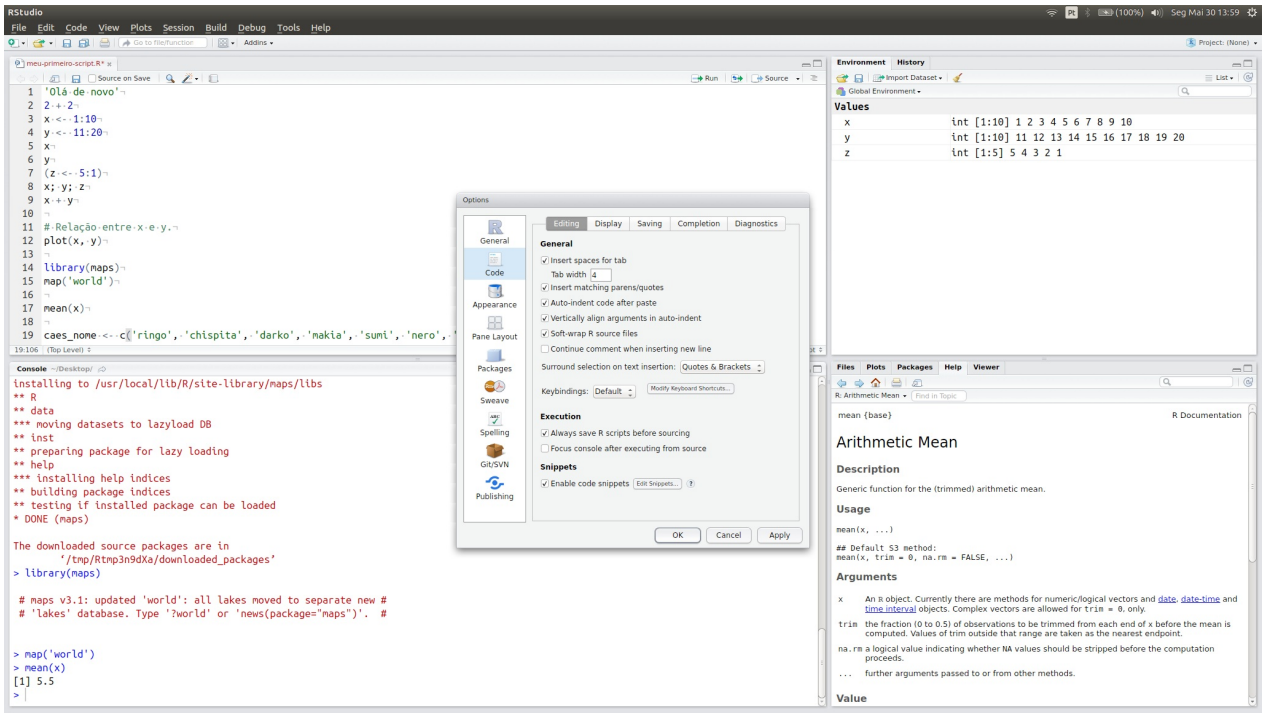
Por fim vejamos uma poucas opções de configuração do script. Como mostra a linha 19 na figura abaixo, o comando se estende além do limite direito do painel do script.



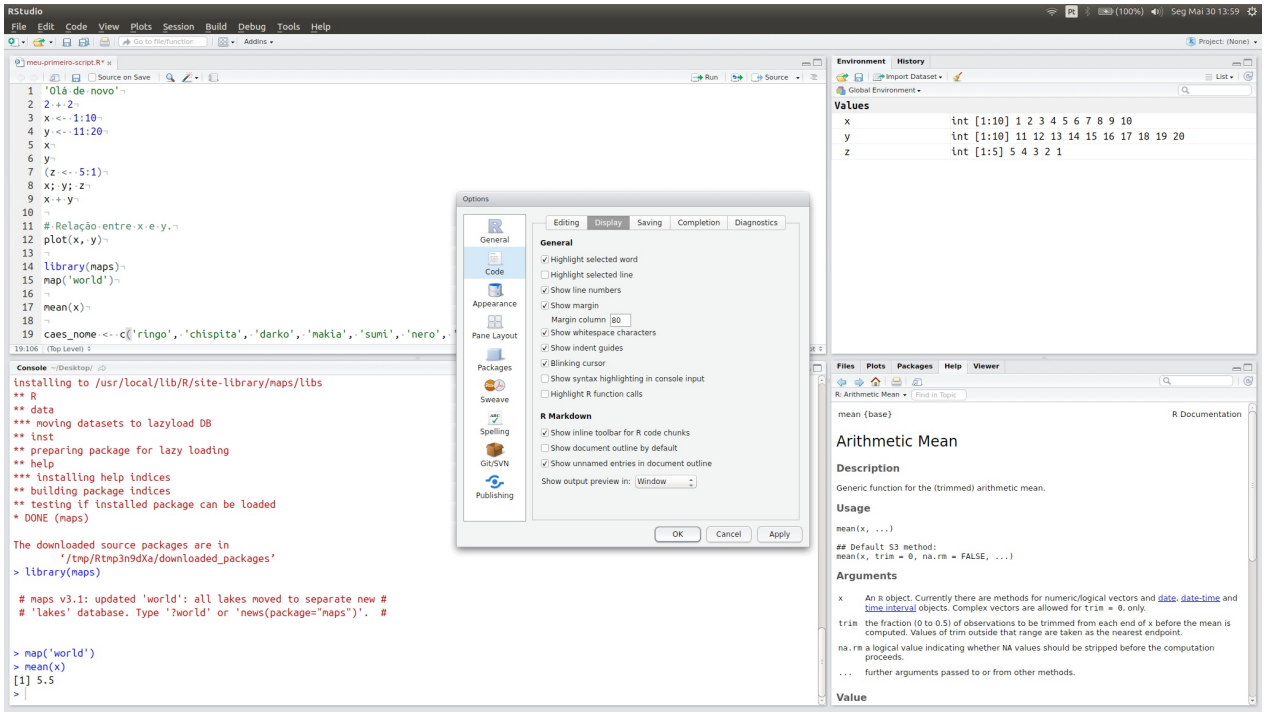
Embora seja possível posicionar o mouse na divisão entre dois painéis e deslocar essa divisão,

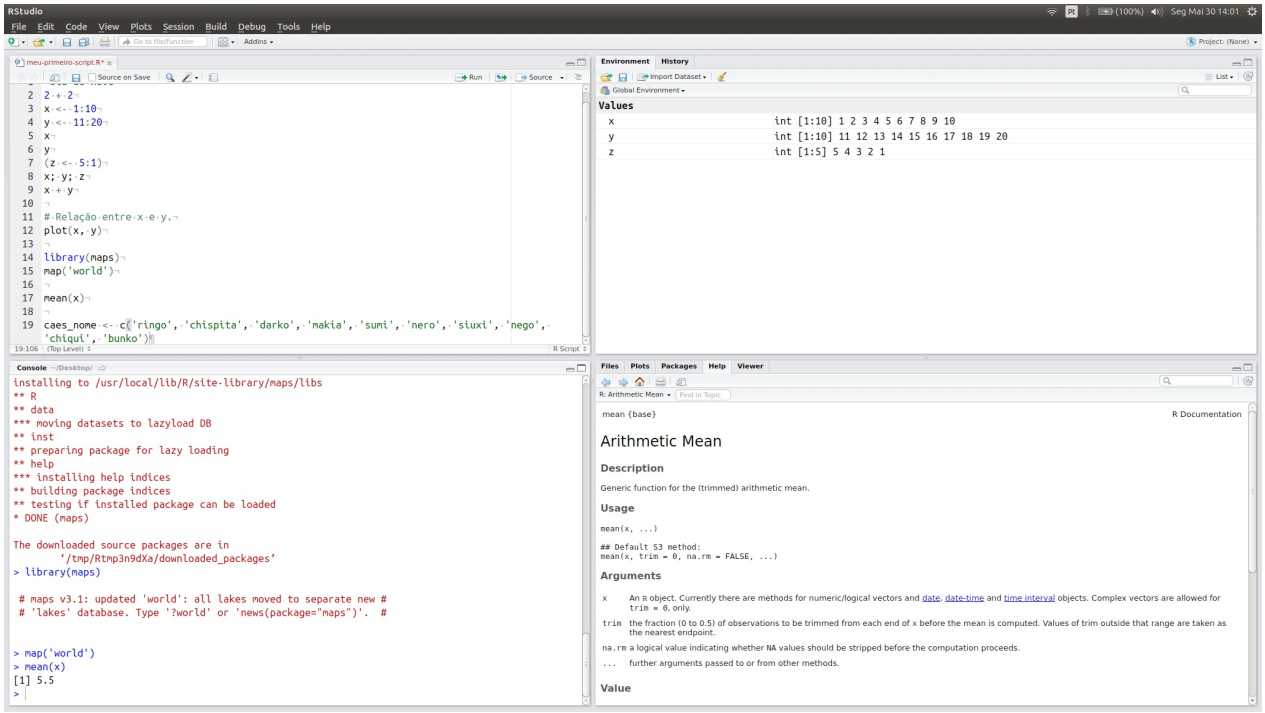


podemos modificar a configuração do script para que a linha seja quebrada. Para isso, precisamos clicar na janela *Tools* da barra de menu e depois em *Global Options*, *Code*, e ativar a opção *Soft-wrap R source files*.

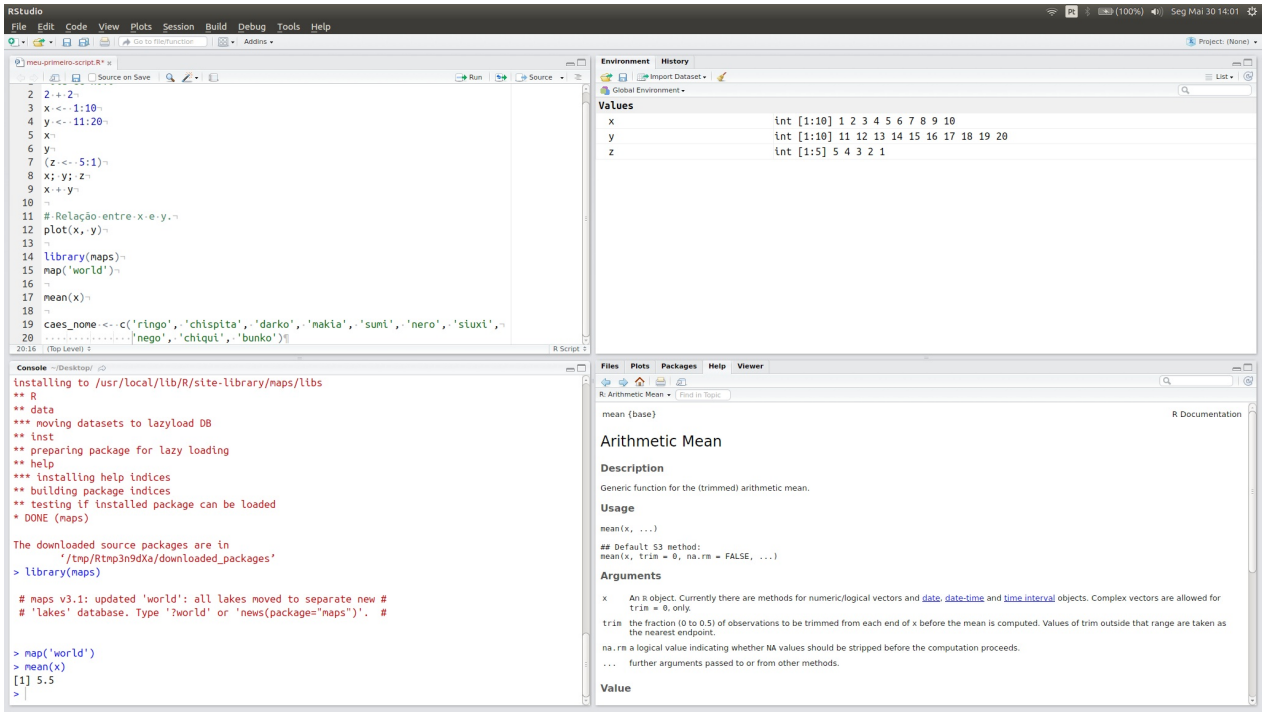


Além disso, podemos visualizar uma margem que define uma largura de 80 caracteres, clicando em *Display* e ativando a opção *Show margin*.





Em programação, essa largura é um padrão. Para evitar que os comandos passem a margem como na figura acima, simplesmente devemos clicar *Enter* antes de sobrepassá-la. O lugar certo para dividir um comando é depois de uma vírgula ou de um operador matemático.


RStudio
File Edit Code View Plots Session Build Debug Tools Help
meu-primeiro-script.R*
2 + 2
x <- 1:10
y <- 11:20
x
y
(z <- 5:1)
x; y; z
x + y
#·Relação entre x e y.
plot(x, y)
library(maps)
map('world')
mean(x)
caes_nome <- c('ringo','chispita','darko','makia','sumi','nero','siuxi',
'nego','chiqui','bunko')
Environment History
Values
x int [1:10] 1 2 3 4 5 6 7 8 9 10
y int [1:10] 11 12 13 14 15 16 17 18 19 20
z int [1:5] 5 4 3 2 1
Console ~/Desktop/
installing to /usr/local/lib/R/site-library/maps/libs
** R
** data
*** moving datasets to lazyload DB
** inst
** preparing package for lazy loading
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded
* DONE (maps)
The downloaded source packages are in
'/tmp/Rtmp3n9dXa/downloaded_packages'
> library(maps)
maps v3.1: updated 'world': all lakes moved to separate new
'lakes' database. Type '?world' or 'news(package="maps")'.
> map('world')
> mean(x)
[1] 5.5
Files Plots Packages Help Viewer
mean {base}
R Documentation
Arithmetic Mean
Description
Generic function for the (trimmed) arithmetic mean.
Usage
mean(x, ...)
Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
Arguments
Value

Estilo de programação no R

Os profissionais que lidam com dados costumam investir uma parte importante dos seus esforços produzindo e usando códigos. Entretanto, a utilidade desses códigos depende da eficiência dos esforços, que por sua vez depende da aplicação de boas práticas de programação. Ainda mais importante do que a eficiência é a validade das informações geradas pelos códigos, pois conclusões obtidas de uma análise podem ser erradas, mesmo quando o código não gera mensagens de erro ou avisos. Portanto, é fundamental incorporar boas práticas de programação na rotina de trabalho, para tirar o maior proveito dos esforços realizados e diminuir a chance de produzir resultados errados.

As boas práticas de programação incluem, entre outras coisas:

- Construção de códigos de fácil interpretação e reprodução
- Documentação
- Automatização de tarefas repetitivas
- Controle de versionamento
- Testes de validade
- Otimização
- Revisão por pares

A consideração dos itens anteriores está além dos propósitos deste livro, mas são discutidos por outros (Wilson e col., 2014; Wickham, 2014). No entanto, o estilo de programação, que é transversal aos itens listados, será o foco deste capítulo.

No R, as convenções quanto ao estilo são menos uniformes em comparação com outras linguagens, e isso se evidencia na falta de um único estilo e na existência de mais de uma recomendação em relação a uma mesma questão. Os estilos propostos por Google e por Wickham, 2014 são referências populares, específicas para o R. Esses estilos são a base do que veremos neste capítulo. Por outro lado, os artigos de Wilson e col., 2014 e Wikipedia#Need_for_comments) são referências genéricas.

Designação

Embora o símbolo `=` funcione como operador de designação, `<-` é o padrão.

```
> # Recomendado
> x <- 10
>
> # Não recomendado
> x = 10
```

Espaçamento

Operadores

Colocar um espaço antes e depois de operadores (`=` , `+` , `-` , `<-` , etc.).

```
> # Recomendado
> x <- 10
> 3 + x
>
>
> # Não recomendado
> x<-10
> 3+x
```

No caso do operador `:` que serve para criar sequências, não se devem colocar espaços antes ou depois.

```
> # Recomendado
> 1:5
>
> # Não recomendado
> 1 : 5
```

Vírgulas

Colocar um espaço depois, mas não antes das vírgulas. Por exemplo, a função `seq` cria uma sequência e seus dois primeiros argumentos (`from` e `to`) definem o começo e o fim da sequência.

```
> # Recomendado
> seq(from = 1, to = 10)
>
> # Não recomendado
> seq(from = 1,to = 10)
> seq(from = 1 , to = 10)
```

Código entre parêntesis ou colchetes

Não colocar espaço ao redor do código contido em parêntesis ou colchetes.

```
> x <- 1:5
>
> # Recomendado
> mean(x)
>
> # Não recomendado
> mean( x )
```

Como veremos no próximo capítulo (*Objetos*), os colchetes servem para selecionar (indexar) um subconjunto de elementos de um objeto. Por exemplo, o seguinte código indexa os elementos 2 a 4 do objeto `x` que é uma sequência de 1 a 5.

```
> # Recomendado
> x[2:4]
>
> # Nao recomendado
> x[ 2:4 ]
```

Em objetos com mais de uma dimensão como é o caso das matrizes, a seleção de elementos é feita usando colchetes e vírgulas (ver capítulo *Objetos*). No seguinte código, x é uma matriz com duas dimensões: linhas e colunas. Especificamente, uma matriz com uma sequência de 1 a 10, com cinco colunas (e consequentemente com duas linhas).

```
> (x <- matrix(1:10, ncol = 5))
```

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

Para selecionarmos elementos da matriz, devemos especificar entre colchetes as posições das linhas e colunas a serem selecionadas. No que diz do estilo de programação, o código dentro dos colchetes não deve ter espaços ao redor, mas deve ter um espaço depois da vírgula. Assim, o seguinte código seleciona o número 8 (linha 2 e coluna 4).

```
> # Recomendado
> x[2, 4]
```

```
[1] 8
```

```
> # Não recomendado
> x[2,4]
```

```
[1] 8
```

```
> x[2 , 4]
```

```
[1] 8
```

Parêntesis esquerdo

O parêntesis esquerdo deve ser precedido por um espaço, exceto quando segue o nome de uma função. No R existem palavras reservadas que servem para controlar o fluxo de execução dos códigos (capítulo *Estruturas de controle*) e nesses casos, o espaço precede o parêntesis esquerdo. No código a seguir a palavra reservada `if` testa a condição entre parêntesis, e se for verdadeira, executa o código contido dentro das chaves.

```
> # Recomendado
> if (2 > 1) {
+   'Sim'
+ }
>
> # Não recomendado
> if(2 > 1) {
+   'Sim'
+ }
```

```
> # Recomendado
> mean(x)
>
> # Não recomendado
> mean (x)
```

Chaves

No exemplo anterior de uma estrutura de controle, o código cuja execução depende da condição, foi colocado entre chaves. A primeira chave nunca deve ficar só em uma linha e deve ser seguida por uma linha nova. A última chave deve ficar só em uma linha e deve ser seguida por uma linha nova.

```
> # Recomendado
> if (2 > 1) {
+   'sim'
+ }
>
> # Não recomendado
> if (2 > 1) {'sim'}
>
> if (2 > 1)
+ {
+   'sim'
+ }
```

Uma exceção em relação à última chave é quando as palavras reservas `if` e `else` são usadas conjuntamente. No comando a seguir o código contido nas chaves do `else` é executado se a condição testa pelo `if` é falsa. A exceção consiste na colocação do `else` na mesma linha da última chave do `if`, para evitar mensagens de erro.

```
> # Recomendado
> if (2 > 1) {
+   'sim'
+ } else {
+   'nao'
+ }
```

Largura das linhas

A largura das linhas deve ser de 80 caracteres ou menos. O RStudio permite visualizar uma margem para evitar linhas com mais de 80 caracteres. Para ativar a visualização devemos entrar em *Tools* (barra de menú), posteriormente em *Global options* > *Code* > *Display*, e ativar a opção *Show margin* com *Margin column* igual a 80. Se um dado código tiver mais do que 80 carateres, o lugar certo para dividí-lo é depos de uma vírgula ou operador matemático.

```
> # Recomendado
> paises <- c('Panamá', 'Cuba', 'Costa Rica', 'Argentina', 'Brasil',
+             'Equador', 'Colomia', 'Venezuela')
>
> # Não recomendado
> paises <- c('Panamá', 'Cuba', 'Costa Rica', 'Argentina', 'Brasil'
+             , 'Equador', 'Colomia', 'Venezuela')
```

Indentação

A indentação do código dentro de chaves deve conter 2 espaços; dentro de parêntesis ou colchetes deve alinhar o código de todas as linhas.

```
> # Recomendado
> if (2 > 1) {
+   'sim'
+ }
> seq(from = 1,
+     to = 10)
>
> # Não recomendado
> if (2 > 1) {
+ 'sim'
+ }
> seq(from = 1,
+   to = 10)
```

Nome de objetos

O nome dos objetos deve conter só minúsculas, ser informativo, conciso, e no possível não coincidente com nomes de funções preexistentes ou palavras reservadas. Se formado por mais de uma palavra, o estilo do Google recomenda o `.` como separador e estilo do Wickham recomenda o `_`.

```
> ## Recomendado (não misturar os dois estilos)
> casos.dia1 <- 53 # Google
> casos_dia1 <- 53 # Wickham
>
>
> # Não recomendado
> CasosNoPrimeiroDia <- 53
> a <- 53
```

Nome de arquivos

O nome dos arquivos deve ser informativo, conciso, e no possível não coincidente com nomes de funções preexistentes ou palavras reservadas. Quando formado por mais de uma palavra, o Google recomenda o `_` como separador e o Wickham o `-`.

Recomendado: casos-raiva-brasil-2016.R ou casos_raiva_brasil_2016.R
Não recomendado: meuScript.R

Definição de funções

O nome das funções deve ser informativo, conciso, e no possível não coincidente com nomes de funções preexistentes ou palavras reservadas. Para nomes com mais de uma palavra, o Google recomenda usar a primeira letra de cada palavra em maiúscula, enquanto Wickham recomenda usar `_` como separador. O seguinte exemplo é apenas para ilustrar o estilo de nomeação, pois a construção de funções será abordada no no próximo capítulo.

```
> ## Recomendado (não misturar os dois estilos)
>
> # Google
> MediaAritmetica <- function(x, y) {
+   (x + y) / 2
+ }
> MediaAritmetica(3, 7)
>
> # Wickham
> media_aritmetica <- function(x, y) {
+   (x + y) / 2
+ }
> media_aritmetica(3, 7)
>
> # Não recomendado
> foo <- function(x, y) {
+   (x + y) / 2
+ }
> foo(3, 7)
```

Comentários

Os comentários são fundamentais para facilitar a interpretação e reproduzibilidade. Devem ser concisos e focados no esclarecimento ou contextualização, não na mecânica dos códigos. Por exemplo, se em um estudo consideram-se dados anuais no período 1990-2010 e precisa-se criar um objeto com a sequência de anos nesse período para usá-la posteriormente como variável explicativa em um modelo estátistico, o seguinte código cria a sequência.

```
> anos <- seq(from = 1990, to = 2010)
```

Um possível comentário para esclarecer o conteúdo do objeto `anos` é:

```
> # Variável explicativa.
> anos <- seq(from = 1990, to = 2010)
```

O anterior contrasta com um comentário focado na mecânica

```
> # Sequência do 2000 ao 2010. "from" define o começo e "to" o final.
> anos <- seq(from = 2000, to = 2010)
```

A documentação de funções deve esclarecer os propósitos, os argumentos e o comportamento das mesmas. A função `MediaAritmetica` que criamos anteriormente é simples e o nome é informativo, mas a título do exemplo, sua documentação poderia ser da seguinte maneira.

```
> MediaAritmetica <- function(x, y) {
+   # Calcula a média aritmética entre dois números.
+   # Argumentos:
+   #  x: número inteiro, real ou imaginário.
+   #  y: número inteiro, real ou imaginário.
+   # Rsultado:
+   #  Média aritmética de x e y.
+   (x + y) / 2
+ }
```

Extensão do código

Recomenda-se que os arquivos tenham entre 50 e 200 linhas.

Objetos

No R existem vários tipos de objetos que servem para armazenar e manipular dados. O tipo *integer* armazena números inteiros, o *double* números reais, o *character* caracteres (variáveis de texto), e o *logical* variáveis booleanas (variáveis que assumem os valores "falso" ou "verdadeiro"). A função `typeof` mostra o tipo de um dado objeto, e a página de ajuda dessa função apresenta os tipos de objetos existentes.

Em muitas situações é irrelevante que um número inteiro seja representado como double ou como integer, porém, essa última representação é mais eficiente desde o ponto de vista computacional e portanto é mais conveniente em alguns casos em que os números inteiros não são usados em operações que resultam em números reais, e fazem parte de códigos que demandam bastantes recursos computacionais.

No terceiro exemplo a seguir, o objeto é tipo *double*, apesar de não ter uma parte decimal explícita. Para definirmos um número inteiro como sendo do tipo *integer*, precisamos acrescentar a letra "L".

```
> typeof(1.1)
```

```
[1] "double"
```

```
> typeof(1.0)
```

```
[1] "double"
```

```
> typeof(1)
```

```
[1] "double"
```

```
> typeof(1L)
```

```
[1] "integer"
```

```
> typeof('1')
```

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

```
> typeof(TRUE)
```

```
[1] "logical"
```

```
> typeof(FALSE)
```

```
[1] "logical"
```

Os objetos além de serem de um determinado tipo, possuem uma estrutura especifica determinada pelo número de dimensões, e pela homegeneidade do conteúdo em relação ao tipo dos seus elementos.

Estruturas de objetos segundo o número de dimensões e a homogeneidade dos seus elementos (adaptado de Wickham, 2014).

Dimensões	Homogênea	Heterogênea
1	Vetor atômico e Fator	Lista
2	Matriz	Data frame
n	Array	

Com as estruturas anteriores é possível aplicar as seguintes operações:

Operação	Descrição
Criação	Do objeto em si
Inspecção	Exploração de atributos e da estrutura em si
Indexação	Seleção de um subconjunto de elementos
Substituição	Redefinição de um subconjunto de elementos
Reposicionamento	Alteração da ordem dos elementos
Eliminação	De um subconjunto de elementos
Combinação	De objetos com a mesma estrutura
Coerção	De uma tipo/estrutura para outro/a

Em vetores, matrizes, arrays e data frames, também é possível aplicar operações matemátcias e lógicas.

No presente capítulo não veremos a estrutura array, porque os exemplos de operações em matrizes são facilemente extrapoláveis. Por outro lado, veremos conceitos básicos de funções (objetos especiais que não se encaixam nas tabelas anteriores).

Vetor atômico

Os vetores atômicos armazenam um único tipo de dados, por exemplo, inteiro ou real, mas não ambos. Qualquer um dos exemplos anteriores é uma instância de um vetor atômico, o qual podemos verificr com a função `is.atomic`.

```
> is.atomic(1.1)
```

```
[1] TRUE
```

```
> is.atomic(FALSE)
```

```
[1] TRUE
```

Daqui para frente chamaremos os vetores atômicos simplesmente de vetores, mas na apresentação de outra das estruturas, a lista, veremos que os conceitos vetor e vetor atômico não são estritamente iguais.

Criação

Uma das principais formas para criar vetores com comprimento maior a 1, é usando a função `c`, que concatena objetos do mesmo tipo. Para vetores com comprimento igual a 1, também podemos usar essa função, mas como vimos não é necessário.

```
> (casos <- c(42, 25))
```

```
[1] 42 25
```

```
> c(1)
```

```
[1] 1
```

Além da função `c`, podemos usar outras funções que retornam vetores, como no caso da função `seq` que cria sequências.

```
> seq(from = 1, to = 10)
```

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

```
> 1:10 # Equivalente ao comando anterior.
```

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

```
> seq(from = 1, to = 10, by = 2)
```

```
[1] 1 3 5 7 9
```

Atributos e estrutura

Os vetores têm um atributo chamado `name` quando os seus objetos possuem nome; caso contrário, o atributo é igual a `NULL`.

```
> attributes(casos)
```

```
NULL
```

```
> casos2015 <- c(notificados = 42, autoctones = 25, importados = 5)
> attributes(casos2015)
```

```
$names
[1] "notificados" "autoctones"  "importados"
```

O exemplo anterior também mostra uma característica da designação de nomes aos objetos. O nome `casos2015` é designado ao resultado da função `c` usando o símbolo "<-", mas os nomes `notificados`, `autoctones` e `importados` são designados dentro da função `c` usando o símbolo "=". Dentro das funções, o comportamento de "=" e "<-" é diferente, pois o uso do primeiro criará os objetos só dentro do ambiente da função, enquanto o uso do segundo criará esses objetos tanto no ambiente da função como no ambiente global (aparecerão no painel *Environment* do RStudio).

A estrutura detalhada dos objetos pode ser inspecionada com a função `str`.

```
> str(casos)
```

```
 num [1:2] 42 25
```

O código acima mostra que `casos` é um vetor numérico (`num`) com dois elementos: 42 e 45. "Numérico" não é um tipo, é um *modo* que abrange os tipos inteiro e real.

```
> mode(1L)
```

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

```
> mode(1.1)
```

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

`casos2015` é um vetor numérico nomeado com três elementos. Os nomes estão contidos no atributo `names` que é um vetor tipo caracter com três elementos.

A comprimento dos vetores indica o número de objetos contidos.

```
> length(1.1)
```

```
[1] 1
```

```
> length(casos)
```

```
[1] 2
```

Indexação

Para indexar um elemento específico de um vetor, podemos usar sua posição ou seu nome entre aspas.

```
> casos2015[2]
```

```
autoctones
        25
```

```
> casos2015['autoctones']
```

```
autoctones
        25
```

Para selecionar dois ou mais elementos, devemos indicar as posições ou nomes dentro de um vetor ou mediante uma sequência de posições.

```
> casos2015[c('autoctones', 'importados')]
```

```
autoctones importados
        25          5
```

```
> casos2015[c(2, 3)]
```

```
autoctones importados
        25          5
```

```
> casos2015[2:3]
```

```
autoctones importados
        25          5
```

A indexação usando um vetor lógico é outra possibilidade.

```
> idades <- c(3, 7, 12, 4, 1, 5, 6)
> idades[c(TRUE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE)]
```

```
[1] 3 7
```

```
> idades > 5
```

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

```
> idades[idades > 5]
```

```
[1]  7 12  6
```

```
> idades[idades > 2 & idades < 7]
```

```
[1] 3 4 5 6
```

```
> idades[idades < 4 | idades == 12]
```

```
[1]  3 12  1
```

Um elemento pode ser indexado mais de uma vez e a não especificação de nenhum elemento equivale a indexação de todos.

```
> idades[c(1, 1)]
```

```
[1] 3 3
```

```
> idades[]
```

```
[1]  3  7 12  4  1  5  6
```

Substituição

Tudo o que pode ser indexado também pode ser substituído.

```
> casos2015
```

```
notificados  autoctones  importados
         42          25           5
```

```
> (casos2015['notificados'] <- 50)
```

```
[1] 50
```

Para substituir o nome de um elemento precisamos indexar o nome ou nomes a serem substituidos no resultado da função `names`.

```
> names(casos2015)
```

```
[1] "notificados" "autoctones"  "importados"
```

```
> names(casos2015)[1] <- 'suspeitos'
> names(casos2015)
```

```
[1] "suspeitos"  "autoctones" "importados"
```

Reposicionamento

Para reposicionar os elementos, basta indexar na sequência desejada

```
> casos2015[c(1, 3, 2)]
```

```
 suspeitos importados autoctones
        50          5         25
```

e se queremos as novas posições em `casos2015`, devemos reassignar o nome.

```
> (casos2015 <- casos2015[c(1, 3, 2)])
```

```
 suspeitos importados autoctones
        50          5         25
```

Eliminação

Ao indexar uma posição precedida pelo sinal "-", o elemento correspondente será removido.

```
> casos2015[-1]
```

```
importados autoctones
         5         25
```

Para eliminar dois ou mais elementos, o sinal "-" deve preceder o vetor com os elementos a serem eliminados.

```
> casos2015[-c(2, 3)]
```

```
suspeitos
       50
```

Combinação

A combinação de dois ou mais vetores equivale a concatenação dos mesmos.

```
> c(c(1, 3), c(10, 20))
```

```
[1]  1  3 10 20
```

```
> c(c(1, 3), c(10, 20), 5)
```

```
[1]  1  3 10 20  5
```

Como a função `append` há maior controle, pois podemos especificar, em que posição do primeiro vetor deve ser colocado o segundo.

```
> append(c(1, 3), c(10, 20), after = 1)
```

```
[1]  1 10 20  3
```

Coerção

Dado que os vetores só podem ter um único tipo de objetos, a combinação de tipos diferentes resulta na coerção para um único tipo. Assim, no segundo comando do exemplo a seguir, o 1 passa a ser de tipo caracter.

```
> typeof(c(1, 3))
```

```
[1] "double"
```

```
> typeof(c(1, 'a'))
```

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

Também há funções que coercionam de um tipo para outro quando é possível.

```
> as.character(c(1, 3))
```

```
[1] "1" "3"
```

```
> typeof(c(1L, 3L))
```

```
[1] "integer"
```

```
> typeof(as.double(c(1L, 3L)))
```

```
[1] "double"
```

Operações

A seguinte tabela mostra operadores matemáticos comuns.

Operador	Descrição	Exemplo
+	Soma	10 + 5
-	Subtração	10 - 5
*	Multiplicação	10 * 5
/	Divisão	10 / 5
^	Exponenciação	10 ^ 5
exp	Exponenciação (base e)	exp(10)
log	Logaritmo	log(10)
		log(10, base = exp(1))
		log(8, base = 2)
sqrt	Raíz quadrada	sqrt(64)
%/%	Resultado inteiro da divisão	10 %/% 3
%%	Módulo (resto)	10 %% 3
abs	Valor absoluto	abs(-10)

Nas operações matemáticas a precedência dos operadores é: ^ > *, / > +, - > operadores lógicos. O contido entre parêntesis altera a precdência.

```
> 2 ^ 3 * 4
```

```
[1] 32
```

```
> 2 ^ (3 * 4)
```

```
[1] 4096
```

```
> 2 * 3 + 4
```

```
[1] 10
```

```
> 2 * (3 + 4)
```

```
[1] 14
```

```
> 2 + 3 == 5
```

```
[1] TRUE
```

Com dois ou mais vetores do mesmo comprimento (dois ou mais elementos), as operações matemáticas são feitas entre elementos de posições correspondentes.

```
> c(1, 3) + c(2, 4)
```

```
[1] 3 7
```

Se os vetores tem tamanhos diferentes, os elementos do mais curto são recilados.

```
> c(10, 12, 14, 16) - c(2, 3)
```

```
[1]  8  9 12 13
```

Quando o comprimento do mais curto não é múltiplo dos outros vetores, continua havendo reciclagem, mas acompanhada de uma mensagem de aviso.

```
> c(10, 12, 14) - c(2, 3)
```

```
Warning in c(10, 12, 14) - c(2, 3): longer object length is not a
multiple of shorter object length
```

```
[1]  8  9 12
```

As operações mediadas por funções são aplicadas a cada um dos elementos de um vetor.

```
> sqrt(c(64, 100))
```

```
[1]  8 10
```

Existem funções que servem para realizar operações entre os elementos de um mesmo vetor.

```
> sum(c(2, 3, 5)) # Soma
```

```
[1] 10
```

```
> cumsum(c(2, 3, 5)) # Soma acumulada
```

```
[1]  2  5 10
```

```
> prod(c(2, 3, 5)) # Produto
```

```
[1] 30
```

```
> cumprod(c(2, 3, 5)) # Produto acumulado
```

```
[1]  2  6 30
```

Mesmo quando não há uma função específica para uma dada operação entre os elementos de um vetor, podemos aplicar a operação mediante a função `Reduce`. No código a seguir, o operador de devisão "/", é aplicado aos dois primeiros elementos (100 / 2) e posteriormente, ao resultado (50) com o seguinte elemento (50 / 2).

```
> Reduce('/', c(100, 2, 2))
```

```
[1] 25
```

Como o operador / não é uma função convencional com argumentos entre parêntesis, deve ser colocado entre aspas. No caso de outras funções, as aspas são desnecessárias.

```
> Reduce(sum, c(2, 3, 5)) # Equivalente a cumsum(c(2, 3, 5))
```

```
[1] 10
```

Os operadores lógicos servem para construir expresões lógicas cujo resultado é falso ou verdadeiro.

```
> 3 < 5 # 3 menor do que 5?
```

```
[1] TRUE
```

```
> 3 <= 5 # 3 menor o igual a 5?
```

```
[1] TRUE
```

```
> 3 >= 5 # 3 maior ou igual a 5?
```

```
[1] FALSE
```

```
> 3 > 5 # 3 maior do que 5?
```

```
[1] FALSE
```

```
> 3 == 5 # 3 igual a 5?
```

```
[1] FALSE
```

```
> 3 != 5 # 3 diferente de 5?
```

```
[1] TRUE
```

```
> 3 > 4 & 5 > 4 # 3 e 5 são maiores do que 4?
```

```
[1] FALSE
```

```
> 3 > 4 | 5 > 4 # 3 ou 5 são maiores do que 4?
```

```
[1] TRUE
```

```
> a <- 1:3
> any(a > 2) # Algum dos valores em "a" é maior do que 12?
```

```
[1] TRUE
```

```
> all(a > 2) # Todos os valores em "a" são maiores do que 12?
```

```
[1] FALSE
```

```
> identical(c(1, 3), c(4, 5)) # Os dois vetores são iguais
```

```
[1] FALSE
```

```
> c(1, 3, 5, 7) %in% c(4, 5, 6, 7) # Quais elementos do 1o vetor estão no 2o
```

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

As seguintes funções executam operações de conjuntos.

```
> which(c(1, 2, 3, 4) > 2) # Quais elementos são maiores do que 2
```

```
[1] 3 4
```

```
> union(x = 1:3, y = 3:5) # Quais elementeos estão em x ou em y
```

```
[1] 1 2 3 4 5
```

```
> intersect(x = 1:3, y = 3:5) # Quais elementeos estão em x e em y
```

```
[1] 3
```

```
> setdiff(x = 1:3, y = 3:5) # Quais elementeos de x não estão em y
```

```
[1] 1 2
```

Matrizes

As matrizes também armazenam um único tipo de objetos, mas a diferença dos vetores, possuem duas dimensões: linhas e colunas.

Criação

A função `matrix` cria matrizes e nos dá a possibilidade de definir o número de linhas e de colunas.

```
> matrix(1:6, ncol = 3)
```

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

```
> matrix(1:6, nrow = 3)
```

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

Por padrão, as posições são preenchidas por coluna, mas isso é controlado pelo argumento `byrow`.

```
> matrix((1:6), ncol = 3, byrow = TRUE)
```

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

Cada linha ou coluna de uma matriz é um vetor e portanto outra forma de criar matrizes é combinando vetores do mesmo comprimento em linhas (com `rowbind`) ou em colunas (com `cbind`).

```
> rbind(1:3, c(10, 20, 30))
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]   10   20   30
```

```
> cbind(1:3, c(10, 20, 30))
```

```
     [,1] [,2]
[1,]    1   10
[2,]    2   20
[3,]    3   30
```

```
> is.matrix(cbind(1:3, c(10, 20, 30)))
```

```
[1] TRUE
```

Atributos e estrutura

As matrizes cujas linhas e colunas não têm nome, possuem um único atributo `dim` que indica o comprimento de cada dimensão. O primeiro valor indica o número de linhas e segundo o número de colunas.

```
> (vacinados <- matrix(c(8, 5, 0, 13, 4, 1, 10, 6, 2), ncol = 3))
```

```
     [,1] [,2] [,3]
[1,]    8   13   10
[2,]    5    4    6
[3,]    0    1    2
```

```
> attributes(vacinados)
```

```
$dim
[1] 3 3
```

As funções `rownames` e `colnames` nomeam as linhas e as colunas respectivamente.

```
> rownames(vacinados) <- c('a', 'b', 'c')
> vacinados
```

```
  [,1] [,2] [,3]
a    8   13   10
b    5    4    6
c    0    1    2
```

```
> colnames(vacinados) <- c('sim', 'nao', 'desconhecido')
> vacinados
```

```
  sim nao desconhecido
a   8  13           10
b   5   4            6
c   0   1            2
```

A função `dimnames` mostra os nomes dentro de cada uma das dimensões e a função `names` nomea as próprias dimensões.

```
> dimnames(vacinados)
```

```
[[1]]
[1] "a" "b" "c"

[[2]]
[1] "sim"          "nao"          "desconhecido"
```

```
> names(dimnames(vacinados)) <- c('grupo', 'vacinado')
> vacinados
```

```
     vacinado
grupo sim nao desconhecido
    a   8  13           10
    b   5   4            6
    c   0   1            2
```

A função `matrix` também tem o argumento `dimnames` que cria os nomes no momento em que a matriz é criada. Nesse argumento, os nomes devem ser definidos dentro da função `list`, que veremos posteriormente.

```
> matrix(c(8, 5, 0, 13, 4, 1, 10, 6, 2), ncol = 3,
+        dimnames = list(c('a', 'b', 'c'), NULL))
```

```
  [,1] [,2] [,3]
a    8   13   10
b    5    4    6
c    0    1    2
```

```
> matrix(c(8, 5, 0, 13, 4, 1, 10, 6, 2), ncol = 3,
+        dimnames = list(NULL, c('sim', 'nao', 'desconhecido')))
```

```
     sim nao desconhecido
[1,]   8  13           10
[2,]   5   4            6
[3,]   0   1            2
```

```
> matrix(c(8, 5, 0, 13, 4, 1, 10, 6, 2), ncol = 3,
+        dimnames = list(c('a', 'b', 'c'), c('sim', 'nao', 'desconhecido')))
```

```
  sim nao desconhecido
a   8  13           10
b   5   4            6
c   0   1            2
```

```
> matrix(c(8, 5, 0, 13, 4, 1, 10, 6, 2), ncol = 3,
+        dimnames = list(grupo = c('a', 'b', 'c'),
+                        vacinado = c('sim', 'nao', 'desconhecido')))
```

```
     vacinado
grupo sim nao desconhecido
    a   8  13           10
    b   5   4            6
    c   0   1            2
```

Quando pelo menos uma das dimensões tem nomes, as matrizes ganham o argumento `dimnames`.

```
> attributes(vacinados)
```

```
$dim
[1] 3 3

$dimnames
$dimnames$grupo
[1] "a" "b" "c"

$dimnames$vacinado
[1] "sim"          "nao"          "desconhecido"
```

`vacinados` é uma matriz numérica com três linhas, três colunas e um atributo `dimnames` que é uma lista com dois vetores nomeados de tipo caracter.

```
> str(vacinados)
```

```
 num [1:3, 1:3] 8 5 0 13 4 1 10 6 2
 - attr(*, "dimnames")=List of 2
  ..$ grupo   : chr [1:3] "a" "b" "c"
  ..$ vacinado: chr [1:3] "sim" "nao" "desconhecido"
```

A função `length` mostra o total de elementos,

```
> length(vacinados)
```

```
[1] 9
```

enquanto `dim`, `nrow` e `ncol` mostram o comprimentos de cada uma das dimensões.

```
> dim(vacinados)
```

```
[1] 3 3
```

```
> nrow(vacinados)
```

```
[1] 3
```

```
> ncol(vacinados)
```

```
[1] 3
```

Indexação

A indexação de linhas ou colunas é feita pela posição, pelo nome ou logicamente como no caso dos vetores. Para selecionar linhas e manter todas as colunas, devemos especificar os índices seguidos de uma vŕigula.

```
> vacinados[1:2, ]
```

```
     vacinado
grupo sim nao desconhecido
    a   8  13           10
    b   5   4            6
```

Para selecionar colunas e manter todas as linhas, devemos preceder os índices com uma vírgula.

```
> vacinados[, c('nao', 'desconhecido')]
```

```
     vacinado
grupo nao desconhecido
    a  13           10
    b   4            6
    c   1            2
```

A seleção de uma única linha ou coluna resulta em um vetor.

```
> vacinados[, 'nao']
```

```
 a  b  c
13  4  1
```

A seleção simultânea de linhas e colunas é feita especificando os índices das linhas e das colunas antes e depois da vírgula, respectivamente.

```
> vacinados[1:2, c(1, 3)]
```

```
     vacinado
grupo sim desconhecido
    a   8           10
    b   5            6
```

Na indexação lógica avaliamos uma condição em uma dimensão, para selecionar elementos da outra dimensão. No nosso exemplo, isso permite entre outras coisas, selecionar os grupos onde há mais de um vacinado.

```
> vacinados[ , 'sim']
```

```
a b c
8 5 0
```

```
> vacinados[ , 'sim'] > 1
```

```
    a     b     c
 TRUE  TRUE FALSE
```

```
> vacinados[vacinados[ , 'sim'] > 1, ]
```

```
     vacinado
grupo sim nao desconhecido
    a   8  13           10
    b   5   4            6
```

```
> vacinados[c(TRUE, TRUE), ]
```

```
     vacinado
grupo sim nao desconhecido
    a   8  13           10
    b   5   4            6
    c   0   1            2
```

Substituição

Como no caso dos vetores, a substituição é a designação do indexado aos novos valores.

```
> vacinados[3, 3] <- 0
> vacinados[2, 1:2] <- c(6, 9)
```

A substituição de nomes e colunas é feita com `rownames` e `colnames` respectivamente.

```
> colnames(vacinados)[3] <- 'desc'
```

Reposicionamento

Mesmo princípio aplicado aos vetores, em cada uma das dimensões.

```
> vacinados[c('c', 'b', 'a'), 3:1]
```

```
     vacinado
grupo desc nao sim
    c    0   1   0
    b    6   9   6
    a   10  13   8
```

Eliminação

Seguindo o razocínio aplicado aos vetores:

```
> vacinados[-1, ]
```

```
     vacinado
grupo sim nao desc
    b   6   9    6
    c   0   1    0
```

```
> vacinados[-c(1, 2) , -3]
```

```
sim nao
  0   1
```

Combinação

Matrizes com o mesmo número de linhas são combinadas com `cbind`.

```
> a <- matrix(1:6, ncol = 3)
> b <- matrix(11:16, ncol = 3)
> cbind(a, b)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5   11   13   15
[2,]    2    4    6   12   14   16
```

A combinação de matrizes com o mesmo número de colunas é feita com `rbind`.

```
> rbind(a, b)
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
[3,]   11   13   15
[4,]   12   14   16
```

Coerção

A combinação de tipos diferentes coerciona para um único tipo.

```
> matrix(c(1:3, 'a'), ncol = 2)
```

```
     [,1] [,2]
[1,] "1"  "3"
[2,] "2"  "a"
```

A coerção de matriz para vetor e de vetor para matriz também é possível.

```
> as.vector(vacinados)
```

```
[1]  8  6  0 13  9  1 10  6  0
```

```
> as.matrix(casos)
```

```
     [,1]
[1,]   42
[2,]   25
```

```
> as.matrix(casos2015)
```

```
           [,1]
suspeitos    50
importados    5
autoctones   25
```

Operações

Uma operação entre uma matriz e um número, equivale à operação entre cada um dos elementos da matriz e o número.

```
> vacinados + 100
```

```
     vacinado
grupo sim nao desc
    a 108 113  110
    b 106 109  106
    c 100 101  100
```

A divisão de cada elemento pela soma de todos os elementos produz uma matriz de proporções. Podemos gerar o mesmo resultado com a função `prop.table`.

```
> vacinados / sum(vacinados)
```

```
     vacinado
grupo       sim        nao      desc
    a 0.1509434 0.24528302 0.1886792
    b 0.1132075 0.16981132 0.1132075
    c 0.0000000 0.01886792 0.0000000
```

```
> prop.table(vacinados)
```

```
     vacinado
grupo       sim        nao      desc
    a 0.1509434 0.24528302 0.1886792
    b 0.1132075 0.16981132 0.1132075
    c 0.0000000 0.01886792 0.0000000
```

Na operação entre uma matriz y um vetor, os elementos são reciclados se o número de elementos do vetor é menor.

```
> vacinados + c(10, 100, 1000)
```

```
     vacinado
grupo  sim  nao desc
    a   18   23   20
    b  106  109  106
    c 1000 1001 1000
```

Uma operação entre duas matrizes com o mesmo número de linhas e colunas equivale à operação entre elementos com posições correspondentes.

```
> vacinados2 <- vacinados + 100
> vacinados2 - vacinados
```

```
     vacinado
grupo sim nao desc
    a 100 100  100
    b 100 100  100
    c 100 100  100
```

O operador `*` multiplica elementos com posições correspondentes. A multiplicação como entendida em álgebra linear, é mediada pelo operador `%*%`.

```
> matrix(1:4, ncol = 2) * matrix(5:8, ncol = 2)
```

```
     [,1] [,2]
[1,]    5   21
[2,]   12   32
```

```
> matrix(1:4, ncol = 2) %*% matrix(5:8, ncol = 2)
```

```
     [,1] [,2]
[1,]   23   31
[2,]   34   46
```

Algumas funções como `sum`, `cumsum`, `prod` e `cumprod` entre outras, coercionam as matrizes para vetor antes de aplicar a operação.

```
> sum(vacinados)
```

```
[1] 53
```

```
> cumsum(vacinados)
```

```
[1]  8 14 14 27 36 37 47 53 53
```

A função `t` transpõe uma matriz (as linhas viram colunas e as colunas viram linhas).

```
> t(vacinados)
```

```
        grupo
vacinado  a b c
    sim   8 6 0
    nao  13 9 1
    desc 10 6 0
```

Outras funções operam dentro das colunas ou dentro das linhas.

```
> colSums(vacinados)
```

```
 sim  nao desc
  14   23   16
```

```
> rowSums(vacinados)
```

```
 a  b  c
31 21  1
```

```
> addmargins(vacinados)
```

```
     vacinado
grupo sim nao desc Sum
  a     8  13   10  31
  b     6   9    6  21
  c     0   1    0   1
  Sum  14  23   16  53
```

As funções `all.equal`, `identical`, `%in%`, `which`, `union`, `intersect`, e `setdiff` são aplicáveis em matrizes.

Data frame

Como a matriz, o data frame é uma estrutura com duas dimensões: linhas e colunas. Porém, as colunas podem ser de tipos diferentes. Isto faz com que o data frame seja a estrutura utilizada para armazenar bancos de dados com diferentes tipos de informação.

Criação

Com a função `data.frame` podemos combinar vetores de diferentes tipos em um data frame.

```
> (banco <- data.frame(cidade = c('a', 'b', 'c', 'd'),
+                      populacao = c(1500, 3300, 2000, 4500),
+                      casos = c(133, 37, 76, 503),
+                      vigilancia = c('nao', 'sim', 'sim', 'nao')))
```

```
  cidade populacao casos vigilancia
1      a      1500   133        nao
2      b      3300    37        sim
3      c      2000    76        sim
4      d      4500   503        nao
```

Atributos e estrutura

O objeto `banco` tem um atributo `names` com os nomes das colunas e um atributo `row.names` com os nomes das linhas que por padrão é uma sequência de 1 até o número de linhas do data frame. O atributo `class` informa a \textit{classe} do objeto.

```
> attributes(banco)
```

```
$names
[1] "cidade"     "populacao"  "casos"      "vigilancia"

$row.names
[1] 1 2 3 4

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

`banco` é um data frame com quatro observações (linhas) e quatro variáveis (colunas). Todas as colunas são de tipo numérico, mas a primeira e a última possuem uma estrutura adicional conhecida como fator (\textit{factor}) que veremos mais para frente.

```
> str(banco)
```

```
'data.frame':    4 obs. of  4 variables:
 $ cidade    : Factor w/ 4 levels "a","b","c","d": 1 2 3 4
 $ populacao : num  1500 3300 2000 4500
 $ casos     : num  133 37 76 503
 $ vigilancia: Factor w/ 2 levels "nao","sim": 1 2 2 1
```

Indexação

A forma de indexar matrizes também serve para indexar data frames.

```
> banco[1:2, c('populacao', 'casos')]
```

```
  populacao casos
1      1500   133
2      3300    37
```

A seleção apenas de colunas não precisa ser precedida de vírgula como no caso das matrizes.

```
> banco[c('populacao', 'casos')]
```

```
  populacao casos
1      1500   133
2      3300    37
3      2000    76
4      4500   503
```

Para a indexação de uma única coluna exitem dois métodos adicionais, que substituem "[" por "[[" e "$" respectivamente. Os dois últimos operadores de indexação retornam o conteúdo da coluna sem nome.

```
> banco['populacao']
```

```
  populacao
1      1500
2      3300
3      2000
4      4500
```

```
> banco[['populacao']]
```

```
[1] 1500 3300 2000 4500
```

```
> banco$populacao
```

```
[1] 1500 3300 2000 4500
```

Substituição

```
> banco[3, 'casos'] <- 83
```

A substituição dos nomes das linhas e das colunas é feita com `row.names` (no caso das matrizes a função é `rownames`) e `names` respectivamente.

```
> names(banco)[2] <- 'pop'
```

Reposicionamento

Mesmo princípio aplicado às matrizes.

Eliminação

Mesmo princípio aplicado às matrizes.

Combinação

Criemos dois bancos adicionais para ver os métodos de combinação.

```
> (banco2 <- data.frame(cidade = rep(c('a', 'b', 'c', 'd'), each = 2),
+                       mortes = c(17, 15, 4, 4, 7, 5, 23, 26),
+                       ano = rep(c(2015, 2016), 4)))
```

```
  cidade mortes  ano
1      a     17 2015
2      a     15 2016
3      b      4 2015
4      b      4 2016
5      c      7 2015
6      c      5 2016
7      d     23 2015
8      d     26 2016
```

```
> banco3 <- data.frame(c('e', 'f'),
+                      c(25e3, 27.2e3),
+                      c(19, 21), c('sim', 'sim'))
> names(banco3) <- names(banco)
```

No `banco2`, a função `rep` repetiu 2 vezes seguidas cada uma das letras, e 4 vezes alternadas cada um dos anos. No `banco3` criamos as colunas sem nome e depois designamos os nomes com base nos nomes em `banco`.

Para combinar `banco` e `banco2` podemos usar a função `merge` indicando a posição ou o nome da coluna em comum no argumento `by`.

```
> merge(banco, banco2, by = 1)
```

```
  cidade  pop casos vigilancia mortes  ano
1      a 1500   133        nao     17 2015
2      a 1500   133        nao     15 2016
3      b 3300    37        sim      4 2015
4      b 3300    37        sim      4 2016
5      c 2000    83        sim      7 2015
6      c 2000    83        sim      5 2016
7      d 4500   503        nao     23 2015
8      d 4500   503        nao     26 2016
```

```
> merge(banco, banco2, by = 'cidade')
```

```
  cidade  pop casos vigilancia mortes  ano
1      a 1500   133        nao     17 2015
2      a 1500   133        nao     15 2016
3      b 3300    37        sim      4 2015
4      b 3300    37        sim      4 2016
5      c 2000    83        sim      7 2015
6      c 2000    83        sim      5 2016
7      d 4500   503        nao     23 2015
8      d 4500   503        nao     26 2016
```

Se a posição da coluna em comum não é a mesma, há duas opções:

```
> (banco4 <- banco[c(2, 1, 3:4)])
```

```
   pop cidade casos vigilancia
1 1500      a   133        nao
2 3300      b    37        sim
3 2000      c    83        sim
4 4500      d   503        nao
```

```
> merge(banco4, banco2, by.x = 2, by.y = 1)
```

```
  cidade  pop casos vigilancia mortes  ano
1      a 1500   133        nao     17 2015
2      a 1500   133        nao     15 2016
3      b 3300    37        sim      4 2015
4      b 3300    37        sim      4 2016
5      c 2000    83        sim      7 2015
6      c 2000    83        sim      5 2016
7      d 4500   503        nao     23 2015
8      d 4500   503        nao     26 2016
```

```
> merge(banco4, banco2, by = 'cidade')
```

```
  cidade  pop casos vigilancia mortes  ano
1      a 1500   133        nao     17 2015
2      a 1500   133        nao     15 2016
3      b 3300    37        sim      4 2015
4      b 3300    37        sim      4 2016
5      c 2000    83        sim      7 2015
6      c 2000    83        sim      5 2016
7      d 4500   503        nao     23 2015
8      d 4500   503        nao     26 2016
```

Se tanto a posição como o nome são diferentes, devemos especificar qual é a coluna em comum em cada data frame.

```
> names(banco4)[2] <- 'cid'
> merge(banco4, banco2, by.x = 2, by.y = 1)
```

```
  cid  pop casos vigilancia mortes  ano
1   a 1500   133        nao     17 2015
2   a 1500   133        nao     15 2016
3   b 3300    37        sim      4 2015
4   b 3300    37        sim      4 2016
5   c 2000    83        sim      7 2015
6   c 2000    83        sim      5 2016
7   d 4500   503        nao     23 2015
8   d 4500   503        nao     26 2016
```

```
> merge(banco4, banco2, by.x = 'cid', by.y = 'cidade')
```

```
  cid  pop casos vigilancia mortes  ano
1   a 1500   133        nao     17 2015
2   a 1500   133        nao     15 2016
3   b 3300    37        sim      4 2015
4   b 3300    37        sim      4 2016
5   c 2000    83        sim      7 2015
6   c 2000    83        sim      5 2016
7   d 4500   503        nao     23 2015
8   d 4500   503        nao     26 2016
```

Como `banco` e `bacon3` têm os mesmos nomes de colunas, as linhas de ambos bancos podem ser combinadas.

```
> rbind(banco, banco3)
```

```
  cidade   pop casos vigilancia
1      a  1500   133        nao
2      b  3300    37        sim
3      c  2000    83        sim
4      d  4500   503        nao
5      e 25000    19        sim
6      f 27200    21        sim
```

Coerção

A combinação de diferentes tipos de objetos dentro de uma coluna coerciona para um mesmo tipo de objeto.

```
> banco[2, 2] <- 'z'
> banco
```

```
  cidade  pop casos vigilancia
1      a 1500   133        nao
2      b    z    37        sim
3      c 2000    83        sim
4      d 4500   503        nao
```

A coerção de vetor ou matriz para data frame é possível.

```
> as.data.frame(casos)
```

```
  casos
1    42
2    25
```

```
> as.data.frame(casos2015)
```

```
           casos2015
suspeitos         50
importados         5
autoctones        25
```

```
> as.data.frame(vacinados)
```

```
  sim nao desc
a   8  13   10
b   6   9    6
c   0   1    0
```

A coerção de um data frame para uma matriz, muda tanto a estrutura como o tipo do conteúdo.

```
> as.matrix(banco)
```

```
     cidade pop    casos vigilancia
[1,] "a"    "1500" "133" "nao"
[2,] "b"    "z"    " 37" "sim"
[3,] "c"    "2000" " 83" "sim"
[4,] "d"    "4500" "503" "nao"
```

Operações

Para cada coluna, mesmo princípio das operações em vetores. As funções `all.equal` e `identical` permitem comparar data frames.

Lista

A lista é a estrutura mais flexível quanto à heterogeneidade do seu conteúdo, pois seus próprios elementos são listas que podem conter objetos com qualquer uma das estruturas vistas, inclusive outras listas.

Criação

```
> (lista <- list(1:3, vacinados, banco))
```

```
[[1]]
[1] 1 2 3

[[2]]
     vacinado
grupo sim nao desc
    a   8  13   10
    b   6   9    6
    c   0   1    0

[[3]]
  cidade  pop casos vigilancia
1      a 1500   133        nao
2      b    z    37        sim
3      c 2000    83        sim
4      d 4500   503        nao
```

Atributos e estrutura

`lista` é um objeto com atributo `NULL` e três elementos.

```
> attributes(lista)
```

```
NULL
```

```
> str(lista)
```

```
List of 3
 $ : int [1:3] 1 2 3
 $ : num [1:3, 1:3] 8 6 0 13 9 1 10 6 0
  ..- attr(*, "dimnames")=List of 2
  .. ..$ grupo   : chr [1:3] "a" "b" "c"
  .. ..$ vacinado: chr [1:3] "sim" "nao" "desc"
 $ :'data.frame':    4 obs. of  4 variables:
  ..$ cidade    : Factor w/ 4 levels "a","b","c","d": 1 2 3 4
  ..$ pop       : chr [1:4] "1500" "z" "2000" "4500"
  ..$ casos     : num [1:4] 133 37 83 503
  ..$ vigilancia: Factor w/ 2 levels "nao","sim": 1 2 2 1
```

No exemplo acima, cada um dos três cifrões à esquerda está associado, respectivamente, a um dos elementos de `lista`. Aproveitando a semelhança entre gavetas e os elementos de uma lista, designemos os elementos a três nomes.

```
> names(lista) <- c('gaveta1', 'gaveta2', 'gaveta3')
> lista
```

```
$gaveta1
[1] 1 2 3

$gaveta2
     vacinado
grupo sim nao desc
    a   8  13   10
    b   6   9    6
    c   0   1    0

$gaveta3
  cidade  pop casos vigilancia
1      a 1500   133        nao
2      b    z    37        sim
3      c 2000    83        sim
4      d 4500   503        nao
```

```
> str(lista)
```

```
List of 3
 $ gaveta1: int [1:3] 1 2 3
 $ gaveta2: num [1:3, 1:3] 8 6 0 13 9 1 10 6 0
  ..- attr(*, "dimnames")=List of 2
  .. ..$ grupo   : chr [1:3] "a" "b" "c"
  .. ..$ vacinado: chr [1:3] "sim" "nao" "desc"
 $ gaveta3:'data.frame':    4 obs. of  4 variables:
  ..$ cidade    : Factor w/ 4 levels "a","b","c","d": 1 2 3 4
  ..$ pop       : chr [1:4] "1500" "z" "2000" "4500"
  ..$ casos     : num [1:4] 133 37 83 503
  ..$ vigilancia: Factor w/ 2 levels "nao","sim": 1 2 2 1
```

```
> attributes(lista)
```

```
$names
[1] "gaveta1" "gaveta2" "gaveta3"
```

O comprimento de uma lista é dado pelo número de elementos contidos.

```
> length(lista)
```

```
[1] 3
```

Indexação

Criemos outra lista mais complexa para ver alguns exemplos.

```
> (gaveta <- list('g1',
+                 gaveta2 = 'g2',
+                 gaveta3 = list('g3a', 'g3b', gaveta3c = 'g3c')))
```

```
[[1]]
[1] "g1"

$gaveta2
[1] "g2"

$gaveta3
$gaveta3[[1]]
[1] "g3a"

$gaveta3[[2]]
[1] "g3b"

$gaveta3$gaveta3c
[1] "g3c"
```

A indexação com corchetes simples retorna cada um dos elementos selecionados dentro de uma lista. A indexação com corchetes duplos ou com o cifrão retorna o objeto sem contê-lo em uma lista, mas sõ permite indexar 1 elemento.

```
> gaveta[1]
```

```
[[1]]
[1] "g1"
```

```
> gaveta[[1]]
```

```
[1] "g1"
```

```
> gaveta[2]
```

```
$gaveta2
[1] "g2"
```

```
> gaveta[[2]]
```

```
[1] "g2"
```

```
> gaveta[3]
```

```
$gaveta3
$gaveta3[[1]]
[1] "g3a"

$gaveta3[[2]]
[1] "g3b"

$gaveta3$gaveta3c
[1] "g3c"
```

```
> gaveta[[3]]
```

```
[[1]]
[1] "g3a"

[[2]]
[1] "g3b"

$gaveta3c
[1] "g3c"
```

```
> gaveta[['gaveta']]
```

```
NULL
```

```
> gaveta$gaveta3
```

```
[[1]]
[1] "g3a"

[[2]]
[1] "g3b"

$gaveta3c
[1] "g3c"
```

```
> gaveta[1:2]
```

```
[[1]]
[1] "g1"

$gaveta2
[1] "g2"
```

```
> gaveta[[1:2]]
```

```
Error in gaveta[[1:2]]: subscript out of bounds
```

A indexação não está restrita ao primeiro nível hierárquico da estrutura

```
> gaveta$gaveta3[[2]]
```

```
[1] "g3b"
```

```
> gaveta$gaveta3$gaveta3c
```

```
[1] "g3c"
```

e os objetos contidos também podem ser indexados

```
> lista$gaveta3
```

```
  cidade  pop casos vigilancia
1      a 1500   133        nao
2      b    z    37        sim
3      c 2000    83        sim
4      d 4500   503        nao
```

```
> lista$gaveta3[1:2, 2:4]
```

```
   pop casos vigilancia
1 1500   133        nao
2    z    37        sim
```

Tendo visto a forma de indexar listas, podemos entender melhor o resultado da função `str` aplicada ao objeto `vacinados`.

```
> str(vacinados)
```

```
 num [1:3, 1:3] 8 6 0 13 9 1 10 6 0
 - attr(*, "dimnames")=List of 2
  ..$ grupo   : chr [1:3] "a" "b" "c"
  ..$ vacinado: chr [1:3] "sim" "nao" "desc"
```

A Expressão `attr(*, "dimnames")` retorna uma lista se substituimos "*" pelo nome do objeto.

```
> attr(vacinados, 'dimnames')
```

```
$grupo
[1] "a" "b" "c"

$vacinado
[1] "sim"  "nao"  "desc"
```

Essa lista pode ser indexada.

```
> attr(vacinados, 'dimnames')$grupo
```

```
[1] "a" "b" "c"
```

Substituição

Os elementos indexados podem ser substituidos por qualquer valor.

```
> gaveta[1:2] <- c(100, 200)
> gaveta
```

```
[[1]]
[1] 100

$gaveta2
[1] 200

$gaveta3
$gaveta3[[1]]
[1] "g3a"

$gaveta3[[2]]
[1] "g3b"

$gaveta3$gaveta3c
[1] "g3c"
```

Reposicionamento

```
> gaveta[c(3, 1, 2)]
```

```
$gaveta3
$gaveta3[[1]]
[1] "g3a"

$gaveta3[[2]]
[1] "g3b"

$gaveta3$gaveta3c
[1] "g3c"


[[2]]
[1] 100

$gaveta2
[1] 200
```

Eliminação

```
> gaveta[-3]
```

```
[[1]]
[1] 100

$gaveta2
[1] 200
```

Combinação

A combinação pode ser feita com as funções `list` e `append` ou arcescentado elementos.

```
> lista2 <- list('a', 'b')
> lista3 <- list('c', 'd')
> list(lista2, lista3)
```

```
[[1]]
[[1]][[1]]
[1] "a"

[[1]][[2]]
[1] "b"


[[2]]
[[2]][[1]]
[1] "c"

[[2]][[2]]
[1] "d"
```

```
> append(lista2, lista3)
```

```
[[1]]
[1] "a"

[[2]]
[1] "b"

[[3]]
[1] "c"

[[4]]
[1] "d"
```

```
> lista2[[3]] <- 'z'
> lista2[[4]] <- lista3
> lista2
```

```
[[1]]
[1] "a"

[[2]]
[1] "b"

[[3]]
[1] "z"

[[4]]
[[4]][[1]]
[1] "c"

[[4]][[2]]
[1] "d"
```

Coerção

Vetores, matrizes e data frames podem ser coercionados para listas.

```
> as.list(casos2015)
```

```
$suspeitos
[1] 50

$importados
[1] 5

$autoctones
[1] 25
```

```
> as.list(matrix(1:4, ncol = 2))
```

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

[[2]]
[1] 2

[[3]]
[1] 3

[[4]]
[1] 4
```

Os data frames são listas com uma estrutura específica.

```
> is.data.frame(banco)
```

```
[1] TRUE
```

```
> is.list(banco)
```

```
[1] TRUE
```

Comtudo, podemos coercionar um data frame para a estrutura geral das listas.

```
> as.list(banco)
```

```
$cidade
[1] a b c d
Levels: a b c d

$pop
[1] "1500" "z"    "2000" "4500"

$casos
[1] 133  37  83 503

$vigilancia
[1] nao sim sim nao
Levels: nao sim
```

Se a estrutura o permite, uma lista pode ser coercionada para data frame.

```
> as.data.frame(list(a = 1:2, b = 3:4))
```

```
  a b
1 1 3
2 2 4
```

Na apresentação da estrutura vetor atômico mencionei que os conceitos vetor e vetor atômico não eram iguais. Existem dois tipos de vetores: atômicos e recursivos. Esses últimos são um sinônimo de lista. A função `is.vector` testa se o objeto em questão é um vetor sem diferenciar o tipo.

```
> is.vector(c(1, 3))
```

```
[1] TRUE
```

```
> is.vector(list(1, 3))
```

```
[1] TRUE
```

Para testar diferenciadamente devemos usar `is.atomic` e `is.list` respectivamente.

```
> is.atomic(c(1, 3))
```

```
[1] TRUE
```

```
> is.list(list(1, 3))
```

```
[1] TRUE
```

```
> is.atomic(c(1, 3))
```

```
[1] TRUE
```

```
> is.list(list(1, 3))
```

```
[1] TRUE
```

Não existe a função `as.atomic` e a função `as.vector` não coerciona as listas para vetores atômicos.

Operações

As funções `all.equal` e `identical` permitem comparar listas

Fator

Embora as categorias de uma variável qualitativa possam ser representadas por números ou strings, a estrutura fator é usualmente mais paropriada.

Tomando como exemplo a variável "tratamento", podemos representar a categorias "a", "b" e "c" como 0, 1 e 2 em um vetor numérico, ou como "a", "b" e "c" em um vetor tipo carctere.

```
> (trat_num <- c(1, 0, 0, 1))
```

```
[1] 1 0 0 1
```

```
> (trat_char <- c('b', 'a', 'a', 'b'))
```

```
[1] "b" "a" "a" "b"
```

A categoria "c" não aprecerá em nehuma operção com os vetores anteriores pelo fato de não estar contida nos mesmos. Por exemplo, a função `table` mostra a frequência com que aprace cada valor distinto, e o valor 2 ou "c" não parece.

```
> table(trat_num)
```

```
trat_num
0 1
2 2
```

```
> table(trat_char)
```

```
trat_char
a b
2 2
```

Criação

Com a estrutura fator podemos manter explicitamente todas as categorias usando o argumento `levels`

```
> (trat_fat <- factor(c('b', 'a', 'a', 'b'), levels = c('a', 'b', 'c')))
```

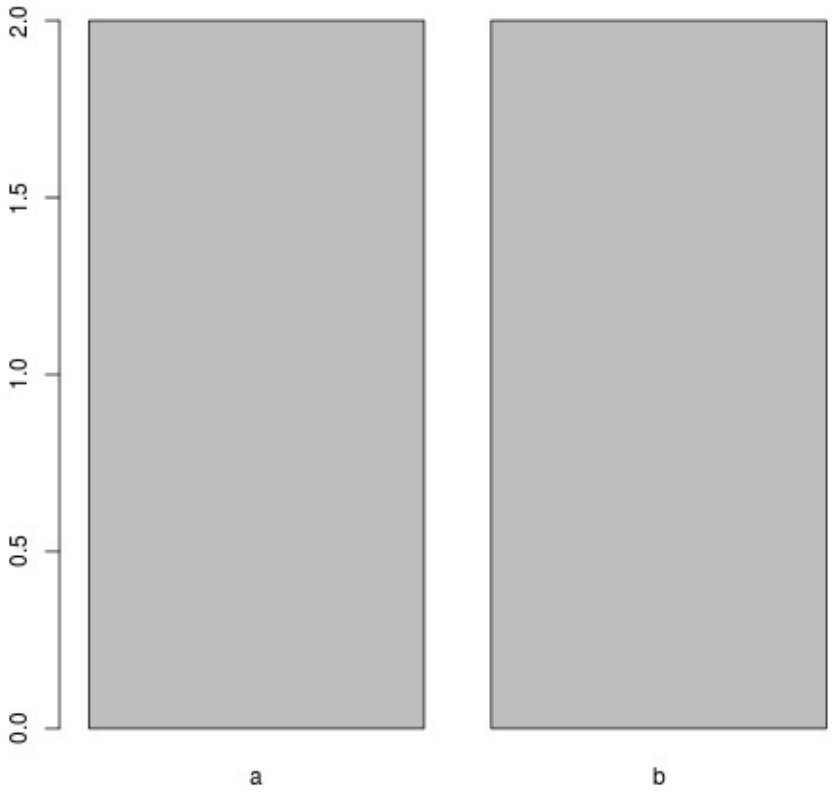
```
[1] b a a b
Levels: a b c
```

e assim, a categoria "c" aparecerá em operações posteriores.

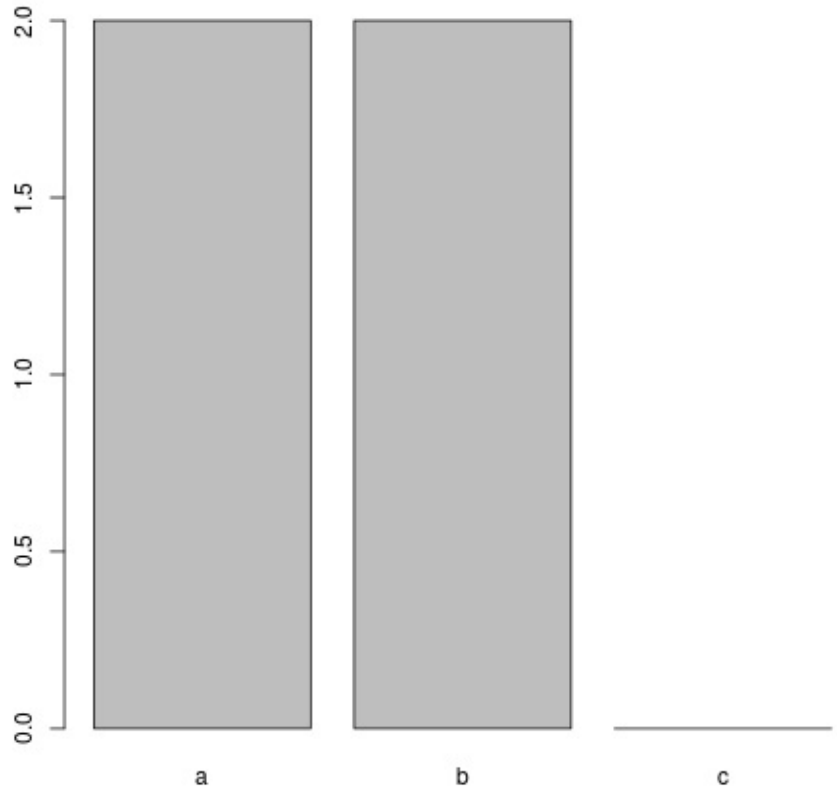
```
> table(trat_fat)
```

```
trat_fat
a b c
2 2 0
```

```
> barplot(table(trat_char))
```




```
> barplot(table(trat_fat))
```



Se a intensidade do tratamento "c" é menor do que a do "b" e esta menor do que a do "a", podemos explicitar essa ordem com o argumento `order`.

```
> (trat_fat2 <- factor(c('b', 'a', 'a', 'b'),
+                      levels = c('c', 'b', 'a'),
+                      ordered = TRUE))
```

```
[1] b a a b
Levels: c < b < a
```

Atributos e estrutura

O atributo `level` mostra as diferentes categorias. O argumento `class` especifica a classe do objeto e se o mesmo é ordenado.

```
> attributes(trat_fat)
```

```
$levels
[1] "a" "b" "c"

$class
[1] "factor"
```

```
> attributes(trat_fat2)
```

```
$levels
[1] "c" "b" "a"

$class
[1] "ordered" "factor"
```

A existência de ordem entre as categorias também é verificável com a função `str`.

```
> str(trat_fat)
```

```
 Factor w/ 3 levels "a","b","c": 2 1 1 2
```

```
> str(trat_fat2)
```

```
 Ord.factor w/ 3 levels "c"<"b"<"a": 2 3 3 2
```

Os fatores são de tipo inteiro, pois internamente cada categoria é representada por um número para facilitar determinadas operações.

```
> typeof(trat_fat)
```

```
[1] "integer"
```

Indexação

A indexação funciona como no caso dos vetores.

Substituição

A susbstituição entre as categorias existentes funciona como no caso dos vetores.

```
> trat_fat[1] <- 'c'
> trat_fat
```

```
[1] c a a b
Levels: a b c
```

Porém, a substituição por uma categoria nova introduz o valor `NA` (NA: valor não disponível, missing ou valor faltante).

```
> trat_fat[1] <- 'd'
```

```
Warning in `[<-.factor`(`*tmp*`, 1, value = "d"): invalid factor
level, NA generated
```

```
> trat_fat
```

```
[1] <NA> a    a    b
Levels: a b c
```

Para acrescentar uma categoria nova é preciso acrescentar primeiro o nível correspondente.

```
> levels(trat_fat)
```

```
[1] "a" "b" "c"
```

```
> c(levels(trat_fat), 'd')
```

```
[1] "a" "b" "c" "d"
```

```
> levels(trat_fat) <- c(levels(trat_fat), 'd')
> trat_fat[1] <- 'd'
> trat_fat
```

```
[1] d a a b
Levels: a b c d
```

Reposicionamento

O reposicionamento dos elementos não afeta a posição dos níveis

```
> levels(trat_fat)
```

```
[1] "a" "b" "c" "d"
```

```
> (trat_fat <- trat_fat[4:1])
```

```
[1] b a a d
Levels: a b c d
```

```
> levels(trat_fat)
```

```
[1] "a" "b" "c" "d"
```

e o reposicionamento dos níveis não afeta nem a posição dos elementos, nem a ordem dos níveis.

```
> trat_fat
```

```
[1] b a a d
Levels: a b c d
```

```
> trat_fat <- factor(trat_fat, levels = levels(trat_fat)[4:1])
> trat_fat
```

```
[1] b a a d
Levels: d c b a
```

Para mudar a ordem dos níveis precisamos a função `ordered` .

```
> trat_fat2
```

```
[1] b a a b
Levels: c < b < a
```

```
> (trat_fat2 <- ordered(trat_fat2, levels = c('a', 'b', 'c')))
```

```
[1] b a a b
Levels: a < b < c
```

Eliminação

A eliminação de um elemento não elimina o respectivo nível.

```
> trat_fat
```

```
[1] b a a d
Levels: d c b a
```

```
> (trat_fat <- trat_fat[-2])
```

```
[1] b a d
Levels: d c b a
```

A função `droplevels` elimina níveis inutilizados.

```
> (trat_fat <- droplevels(trat_fat))
```

```
[1] b a d
Levels: d b a
```

Para eliminar tanto o elemento como o nível correspondente é necessário coercionar para caractere, eliminar o elementos e coercionar para fator.

```
> trat_char <- as.character(trat_fat)
> (trat_char <- trat_char[trat_char != 'a'])
```

```
[1] "b" "d"
```

```
> (trat_fat <- as.factor(trat_char))
```

```
[1] b d
Levels: b d
```

Combinação

Para combinar também é preciso coercionar para caractere.

```
> f1 <- factor(c('a', 'b'))
> f2 <- factor(c('c', 'd'))
> char <- c(as.character(f1), as.character(f2))
> as.factor(char)
```

```
[1] a b c d
Levels: a b c d
```

Coerção

Além da coerção do tipo,

```
> as.integer(trat_fat)
```

```
[1] 1 2
```

```
> as.character(trat_fat)
```

```
[1] "b" "d"
```

é possível a coerção da estrutura.

```
> as.matrix(trat_fat)
```

```
     [,1]
[1,] "b"
[2,] "d"
```

```
> as.data.frame(trat_fat)
```

```
  trat_fat
1        b
2        d
```

```
> as.list(trat_fat)
```

```
[[1]]
[1] b
Levels: b d

[[2]]
[1] d
Levels: b d
```

Operações

As funções `all.equal` , `identical` e %in% permitem comparar fatores.

Funções

As funções são objetos que podemos usar e criar para automatizar tarefas repetitivas e simplificar o código. As funções costumam ter argumentos que determinam o resultado, mas nem todas as funções têm ou precisam argumentos. Por exemplo, a função `date` gera a data do momento de execução e não tem argumentos.

```
> date()
```

```
[1] "Fri Oct 28 17:53:04 2016"
```

Nas funções que têm argumentos, os mesmos têm nomes, posições específicas e opcionalmente, valores predefinidos. Tomando a função `seq` como exemplo, os argumentos `from` e `to` que definem o começo e fim da sequência, tem "1" como valor

padrão. O valor padrão de `by` é equivalente a "1". Os outros argumentos (excetuando `...` que será explorado em breve) não são usados por padrão, pois seu valor padrão é `NULL` (nulo). Se usarmos a função sem argumentos, serão usados os valores padrão.

```
> seq()
```

```
[1] 1
```

Como a sequência vai de 1 até 1, e a amplitude do intervalo é 1. o resultado anterior é o esperado.

Para mudar o comportamento, podemos modificar os valores padrão instanciando os argumentos de interesse.

```
> seq(from = 0, to = 10, by = 2)
```

```
[1]  0  2  4  6  8 10
```

Se os nomes dos argumentos são omitidos, o primeiro valor será a instância do primeiro argumento, o segundo valor a instância do segundo argumento e assim por diante. A omissão dos nomes só deve ser feita quando se tem certeza das posições ocupadas por cada argumento (o confundimento de posições pode gerar erros desapercibidos).

```
> seq(0, 1, 0.2)
```

```
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

Se os nomes dos argumentos são explicitados, a ordem não interessa. Pro exemplo, podemos omitir o terceiro argumento e usar o quarto para definir uma sequência com três valores equidistantes.

```
> seq(from = 0, to = 10, length.out = 3)
```

```
[1]  0  5 10
```

É possível explicitar só alguns nomes, e instanciar os restantes com base na posição.

```
> seq(0, 10, length.out = 3)
```

```
[1]  0  5 10
```

A função `function` cria funções que executam o código contido entre chaves (as chaves não são estrictamente necessárias, mas são um estilo padrão). Assim, o seguinte código cria uma função que recebe dois argumentos (`vetor1` e `vetor2`), calcula a média aritmética de cada vetor assim como a média aritmética global, e retorna as três médias.

```
> Medias <- function(vetor1 = NULL, vetor2 = NULL) {
+   # Média de cada um dos vetores e média global.
+   # Argumentos:
+   #  vetor1: vetor numérico.
+   #  vetor2: vetor numérico.
+   # Resultado:
+   #  Vetor numérico com as três médias.
+   media1 = mean(vetor1)
+   media2 = mean(vetor2)
+   media_global = mean(c(media1, media2))
+   c('Media 1' = media1, 'Media 2' =  media2, ' Media global' = media_global)
+ }
> Medias(c(1, 3), c(5, 7))
```

```
      Media 1       Media 2  Media global
            2             6             4
```

Na função `Medias` usamos internamente a função `mean` para calcular as médias. Entre os argumentos de `mean` está `trim` , que representa uma fração removida de cada extremo dos dados originais antes de clacular a média. Por exemplo, se definimos `trim = .01` para calcular a média de uma sequência logarítmica entre 1 e 100, 1% do dados de cada extremo são removidos antes de calcular a média (o logaritmo de 1 e de 100 são removidos da sequência).

```
> mean(log(1:100))
```

```
[1] 3.637394
```

```
> mean(log(1:100), trim = .01)
```

```
[1] 3.664635
```

```
> mean(log(2:99))
```

```
[1] 3.664635
```

Para poder usar o argumento `trim` na função `Medias`, podemos definir explicitamente outro argumento

```
> Medias <- function(vetor1 = NULL, vetor2 = NULL, frac = NULL) {
+   # Média de cada um dos vetores e média global.
+   # Argumentos:
+   #  vetor1: vetor numérico.
+   #  vetor2: vetor numérico.
+   #  frac: fração removida de cada extremo.
+   # Resultado:
+   #  Vetor numérico com as três médias.
+   media1 = mean(vetor1, trim = frac)
+   media2 = mean(vetor2, trim = frac)
+   media_global = mean(c(media1, media2))
+   c('Media 1' = media1, 'Media 2' =  media2, ' Media global' = media_global)
+ }
> Medias(log(1:100), log(101:200), frac = .2)
```

```
      Media 1       Media 2  Media global
     3.855625      5.007262      4.431444
```

ou usar `...`.

```
> Medias <- function(vetor1 = NULL, vetor2 = NULL, ...) {
+   # Média de cada um dos vetores e média global.
+   # Argumentos:
+   #  vetor1: vetor numérico.
+   #  vetor2: vetor numérico.
+   #  ...: argumentos passados à função mean.
+   # Resultado:
+   #  Vetor numérico com as três médias.
+   media1 = mean(vetor1, ...)
+   media2 = mean(vetor2, ...)
+   media_global = mean(c(media1, media2))
+   c('Media 1' = media1, 'Media 2' =  media2, ' Media global' = media_global)
+ }
> Medias(log(1:100), log(101:200), trim = .2)
```

```
      Media 1       Media 2  Media global
     3.855625      5.007262      4.431444
```

A vantagem da última opção é que se as funções usadas internamente têm múltiplos argumentos, todos poderão ser usados sem necessidade de definí-los explicitamente. O argumento `...` também permite criar funções que recebem um número indefinido de argumentos. No capítulo *Estilo de programação*, criamos uma função para calcular a média aritmética entre dois números.

```
> MediaAritmetica <- function(x, y) {
+   # Calcula a média aritmética entre dois números.
+   # Argumentos:
+   #  x: número inteiro, real ou imaginário.
+   #  y: número inteiro, real ou imaginário.
+   # Rsultado:
+   #  Média aritmética de x e y.
+   (x + y) / 2
+ }
> MediaAritmetica(1, 3)
```

```
[1] 2
```

```
> MediaAritmetica(1, 3, 5, 7)
```

```
Error in MediaAritmetica(1, 3, 5, 7): unused arguments (5, 7)
```

Si na função anterior substituimos os argumentos, `x` e `y` por `...`, poedremos calcular a média de um número indefinido de argumentos.

```
> MediaAritmetica <- function(...) {
+   # Calcula a média aritmética.
+   # Argumentos:
+   #  ...: números.
+   # Rsultado:
+   #  Média aritmética de x e y.
+   valores <- unlist(list(...))
+   sum(valores) / length(valores)
+ }
> MediaAritmetica(1, 3, 5, 7)
```

```
[1] 4
```

No código anterior usamos `...` como argumento da função lista (`list(...)`) para poder usar os argumentos instanciados. Adicionalmente, a lista resultante foi convetida em um vetor atômico (`unlist(list(...))`) para poder usar as funções `sum` e `length` no cálculo da média.

Os objetos vistos são os tijolos de construção no R. Nos próximos capítulos veremos como importar dados de planilhas e armazená-los em data frames.

Importação e exportação de arquivos

Uma forma comum de estruturar os dados é em tabelas com linhas e colunas, sendo que cada linha corresponde a uma observação, e cada coluna a uma variável medida nas observações. Por exemplo, na tabela a seguir a unidade de observação é o domicílio, há cinco observações, e duas variáveis medidas em cada observação.

domicilio	moradores	caes_ou_gatos
1	3	sim
2	2	sim
3	4	nao
4	2	sim
5	1	nao

Editores de planilhas como o Excel o ou Calc são uma alternativa simples e útil para criar tabelas como a anterior, quando a quantidade de dados não compromete a capacidade de processamento e armazenamento. Embora seja possível importar diretamente no R os dados armazenados em uma planilha de Excel, é mais comum salvar os dados em um arquivo CSV para depois importá-los.

Nos arquivos CSV, as tabelas são armazenadas em formato de texto, sendo que cada linha continua sendo uma observação, e as variáveis medidas em cada observação estão separadas por "," ou ";". Assim, o conteúdo da tabela anterior em um arquivo CSV separado por vírgulas seria visualizado em um editor de texto da seguinte maneira:

```
domicilio,moradores,caes_ou_gatos
1,3,sim
2,2,sim
3,4,nao
4,2,sim
5,1,nao
```

Ao criarmos uma tabela como a primeira em um editor de planilhas, podemos salvar o conteúdo em um arquivo CSV, usando a opção *salvar como*, nomeando o arquivo como *domicilios* (por exemplo), e escolhendo a extensão CSV (em Excel escolher *CSV (separado por vírgulas) (*\.csv) *no campo* Tipo*; em Calc escolher* Texto CSV (.csv) *no campo* Todos os formatos*).

Importação

Se no painel *Files* do RStudio escolhermos o diretório em que foi salvo o arquivo, este último aparecerá listado. Se esse diretório não for o diretório de trabalho, podemos defini-lo como tal usando a janela *More* e a opção *Set as Working Directory*. Não é necessário que o diretório com os arquivos seja o diretório de trabalho, mas isso facilita o uso de funções como a `read.csv` que importa o conteúdo de arquivos CSV.

Ao usarmos o nome do arquivo (domicilios.csv) como argumento da função `read.csv` e designar o nome `domicilios` ao resultado da função, o conteúdo do arquivo fica armazenado em um data frame com o nome `domicilios`.

```
> (domicilios <- read.csv('domicilios.csv'))
```

```
  domicilio moradores caes_ou_gatos
1         1         3           sim
2         2         2           sim
3         3         4           nao
4         4         2           sim
5         5         1           nao
```

Se o comando anterior gera o seguinte resultado, é porque o delimitador do arquivo CSV é ";" e não ",". Isso acontece em alguns computadores cuja configuração em português usa ";" como delimitador padrão.

```
  domicilio.moradores.caes_ou_gatos
1                          1;3;sim
2                          2;2;sim
3                          3;4;nao
4                          4;2;sim
5                          5;1;nao
```

Nesses casos, podemos usar a função `read.csv2` cujo separador padrão é ";", ou usar a função `read.csv` e definir o argumento `sep` como ";".

```
> read.csv2('domicilios.csv')
> read.csv('domicilios.csv', sep = ';')
```

Tanto, `read.csv` como `read.csv2` são versões específicas para arquivos CSV, simplificadas a partir da função `read.table`. Com esta última função precisamos definir outros argumentos como para importar o arquivo (como o arquivo tem cabeçalho, `header = TRUE`).

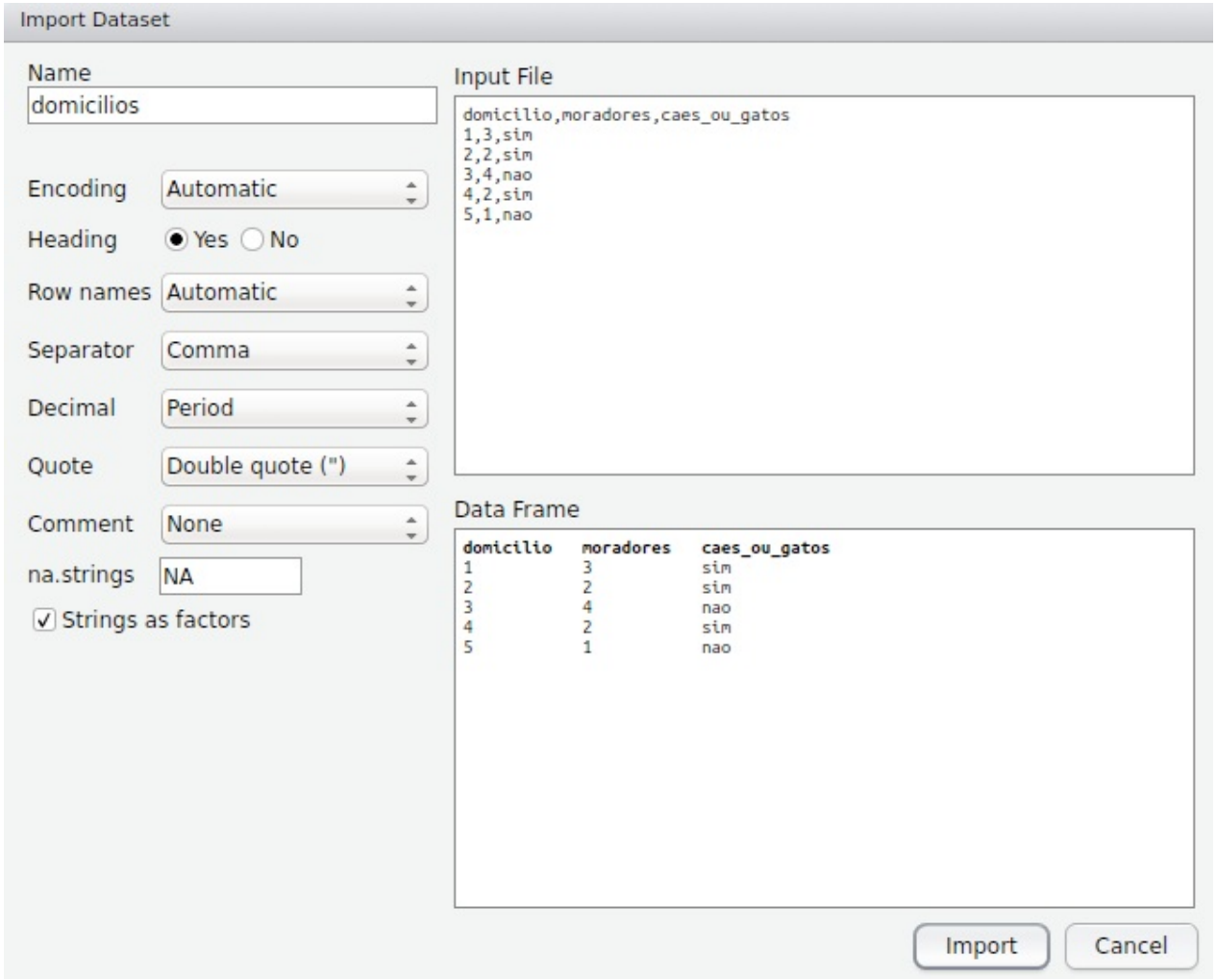
```
> read.table('domicilios.csv', header = TRUE, sep = ',')
```

```
  domicilio moradores caes_ou_gatos
1         1         3           sim
2         2         2           sim
3         3         4           nao
4         4         2           sim
5         5         1           nao
```

A vantagem de `read.table` é que além de importar arquivos CSV, importa outras extensões como TXT.

Interface do RStudio

O RStudio tem uma interface para importar arquivos. No painel *Envorinment*, janela *Import Daset*, a opção *From Local File* abre a seguinte interface que permite definir várias opções, visualizar o arquivo de entrada e visualizar a forma em o arquivo será importado.



Caracteres especiais

Se criamos um arquivo *dom* semelhante ao arquivo *domicilios*, mas com acentos e espaços

domicílio	moradores	cães ou gatos
1	3	sim
2	2	sim
3	4	não
4	2	sim
5	1	não

as tentativas de importação anteriores podem gerar o seguinte erro se o sistema de codificação definido como padrão no RStudio não reconhece caracteres especiais:

```
> read.csv('dom.csv')
```

```
Error in make.names(col.names, unique = TRUE): invalid multibyte string 1
```

Se eliminarmos o cabeçalho com o argumento `header`, o arquivo é importado, mas o cabeçalho é considerado como mais uma linha e os caracteres especiais não são adequadamente representados.

```
> read.csv('dom.csv', header = FALSE)
```

```
            V1        V2               V3
1 domic\xedlio moradores c\xe3es ou gatos
2            1         3              sim
3            2         2              sim
4            3         4           n\xe3o
5            4         2              sim
6            5         1           n\xe3o
```

Embora seja possível especificar a codificação *latin1* para reconhecer caracteres do português, é preferível evitar caracteres especiais nos bancos de dados.

```
> read.csv('dom.csv', encoding = 'latin1')
```

```
  domicílio moradores cães.ou.gatos
1         1         3           sim
2         2         2           sim
3         3         4           não
4         4         2           sim
5         5         1           não
```

Exportação

Um data frame pode ser exportado para um arquivo CSV com a função `write.csv`, especificando o nome do data frame e o nome do arquivo a ser criado. Por padrão, a função `write.csv` acrescenta uma coluna que numera as linhas, mas isso pode ser prevenido com o argumento `row.names`.

```
> write.csv(domicilios, 'domicilios2.csv', row.names = FALSE)
```

Assim como na importação, `write.csv` e `write.csv2` são versões simplificadas de `write.table`.

Outras extensões

Arquivos com extensão de outros programas podem ser importados com funções dos pacotes *Hmisc* e *foreign*, entre outros. Por exemplo, se o arquivo domicílios tivesse a extensão do software *STATA* ou *SPSS*, a importação seria da seguinte maneira.

```
> library(foreign)
> domicilios <- read.dta('domicilios') # STATA
> domicilios <- read.spss('domicilios') # SPSS
```

Geração de dados

A geração de dados é uma ferramenta que amplia as possibilidades de análise de dados. Gerando dados podemos criar cenários de interesse, realizar simulações, e entender melhor o comportamento de funções ou estratégias de programação. Por exemplo, a maioria dos exemplos deste livro são baseados em dados gerados para entender conceitos de programação no R.

Sequências

Como vimos em capítulos anteriores, a forma mais simples de criar sequências é com `:`.

```
> 1:5
```

```
[1] 1 2 3 4 5
```

```
> 7:2
```

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

Porém, o uso de `:` limita-se à criação de sequências que variam de um em um. Com a função `seq` podemos definir a magnitude da variação,

```
> seq(0, 100, by = 10) # Abreviação: seq(0, 100, 10)
```

```
 [1]   0  10  20  30  40  50  60  70  80  90 100
```

o comprimento da sequência,

```
> seq(-200, -100, l = 5) # l: abreviação de length.out
```

```
[1] -200 -175 -150 -125 -100
```

ou passar um vetor de comprimento *x* para gerar uma sequência de 1 até *x*.

```
> vec <- c('a', 'b', 'c')
> length(vec)
```

```
[1] 3
```

```
> seq(vec) # Mais eficiente: seq_along(vec)
```

```
[1] 1 2 3
```

Repetições

Em capítulos anteriores também vimos que a função `rep` permite-nos criar repeteições de um elemento,

```
> rep(1, 5)
```

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

de mais de um elemento alternadamente,

```
> rep(1:2, 5)
```

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

ou de mais de um elemento sequencialmente.

```
> rep(1:2, e = 5) # e: abreviação de each
```

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

Ordenção

A função `sort` ordena por padrão os números de forma crescente, e as letras alfabeticamente dando precedência às minúsculas. Com o argumento `decreascing` sendo verdadeiro, a forma de ordenação é oposta ao padrão.

```
> populacao <- c(778, 511, 903, 150)
> sort(populacao)
```

```
[1] 150 511 778 903
```

```
> sort(populacao, d = T) # d: abreviação de decreascing
```

```
[1] 903 778 511 150
```

```
> regiao <- c('b', 'a', 'd', 'c')
> sort(regiao)
```

```
[1] "a" "b" "c" "d"
```

```
> sort(regiao, d = T)
```

```
[1] "d" "c" "b" "a"
```

A função `order` também auxilia no ordenamento de dados, mas no lugar de retornar os valores ordenados, retorna os índices originas ordenados pelo valor dos elementos. Portanto, no vetor `populacao` o primeiro índice será o 4 (menor valor = 150), o segundo índice será o 2 (segundo menor valor = 511) e assim por diante.

```
> order(populacao)
```

```
[1] 4 2 1 3
```

`order` também tem o argumento `deacreasing`, mas não pode ser abreviado como no caso de `sort`.

```
> order(populacao, decreasing = T)
```

```
[1] 3 1 2 4
```

A ordenação de índices permite ordenar um ou mais vetores com base em outro vetor. Dessa maneira, podemos ordenar as regiões do exemplo anterior, com base no tamanho populacional.

```
> regiao[order(populacao)]
```

```
[1] "c" "a" "b" "d"
```

Tratando-se de data frames, `order` permite-nos usar uma ou mais colunas para ordenar as restantes.

```
> regioes <- data.frame(regiao, populacao, casos = c('sim', 'nao', 'sim', 'sim'))
> regioes[order(regioes$populacao), ]
```

```
  regiao populacao casos
4      c       150   sim
2      a       511   nao
1      b       778   sim
3      d       903   sim
```

```
> regioes[order(regioes$casos, regioes$populacao), ]
```

```
  regiao populacao casos
2      a       511   nao
4      c       150   sim
1      b       778   sim
3      d       903   sim
```

A função `rank` mostra a posição que cada valor tem, na sequência ordenada de valores. Portanto, no vetor `populacao` a posição do primeiro elemento é a terceira (terceiro menor valor = 778), a segunda 2 (segundo menor valor = 511) e assim por diante.

```
> rank(populacao)
```

```
[1] 3 2 4 1
```

Texto

As funções `letters` e `LETTERS` são formas alternativas de gerar letras.

```
> letters
```

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

```
> LETTERS
```

```
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O"
[16] "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```
> sort(c(letters[1:7], LETTERS[1:7]), d = T)
```

```
 [1] "G" "g" "F" "f" "E" "e" "D" "d" "C" "c" "B" "b" "A" "a"
```

Por outro lado, podemos concatenar caracteres com a função `paste`.

```
> paste('regiao', 'a')
```

```
[1] "regiao a"
```

```
> paste('regiao', 1, sep = '-')
```

```
[1] "regiao-1"
```

```
> paste('regiao', letters, sep = ':')
```

```
 [1] "regiao:a" "regiao:b" "regiao:c" "regiao:d" "regiao:e"
 [6] "regiao:f" "regiao:g" "regiao:h" "regiao:i" "regiao:j"
[11] "regiao:k" "regiao:l" "regiao:m" "regiao:n" "regiao:o"
[16] "regiao:p" "regiao:q" "regiao:r" "regiao:s" "regiao:t"
[21] "regiao:u" "regiao:v" "regiao:w" "regiao:x" "regiao:y"
[26] "regiao:z"
```

O argumento `sep` define o caractere de separação. Para concatenar caracteres sem espaços nem outros caracteres de separação, podemos usar `sep = ''` ou a função `paste0` .

```
> paste('regiao', 1, sep = '')
```

```
[1] "regiao1"
```

```
> paste0('regiao', 1)
```

```
[1] "regiao1"
```

O argumento `collapse` colapsa o resultado em um único caractere.

```
> paste(letters[1:5], 1:5, sep = '_', collapse = '--')
```

```
[1] "a_1--b_2--c_3--d_4--e_5"
```

Seleção aleatória

Nesta seção usaremos distribuições probabilísticas comuns para gerar dados aleatórios (amostras probabilísticas). Tecnicamente, os dados gerados pelos software estatísticos são pseudoaleatórios, mas isso, assim como a teoria estatística, está além do escopo deste livro.

Dado um conjunto de dados A, uma amostra é um subconjunto de A. Nas amostras probabilísticas, cada elemento do conjunto A tem uma probabilidade conhecida e diferente de zero, de ser selecionado na amostra. A probabilidade atribuída a cada elemento depende da distribuição probabilística de base.

Distribuição uniforme

Com a distribuição uniforme, cada elemento tem a mesma probabilidade de ser selecionado. Para obtermos uma amostra de uma distribuição uniforme podemos usar a função `runif` especificando o número de elementos (tamanho da amostra) `n` que queremos amostrar e os valores mínimo (`min`) e máximo (`max`) do conjunto de elementos a serem amostrados.

```
> runif(n = 10, min = -50, max = 50)
```

```
 [1] -32.390062   7.448826 -34.533168 -45.838146  -8.382488
 [6] -32.311678  45.082097  41.735794  26.586496 -20.087438
```

Cada vez que for executado o comando anterior, serão selecionados 10 elementos de um conjunto de números contínuos que vão de -50 a 50. Para obtermos sempre o mesmo resultado, devemos executar a função `set.seed` antes da função que produz resultados aleatórios.

```
> set.seed(43)
> runif(10, -50, 50)
```

```
 [1]  -1.4962318  41.0766036 -44.2326110  20.5398411 -18.6504901
 [6]   4.3032635  17.9100947   0.6703597 -31.7028303  37.2258511
```

A função `set.seed` é instanciada com um número inteiro e no exemplo anterior o número 43 foi uma escolha arbitrária. Se mudarmos o número os resultados serão diferentes, mas se usarmos sempre o mesmo número, os resultados serão reproduzíveis.
Para arredondarmos os resultados, podemos usar a função `round`,

```
> set.seed(43)
> round(runif(10, -50, 50), digits = 3)
```

```
 [1]  -1.496  41.077 -44.233  20.540 -18.650   4.303  17.910
 [8]   0.670 -31.703  37.226
```

```
> set.seed(43)
> round(runif(10, -50, 50), digits = 0)
```

```
 [1]  -1  41 -44  21 -19   4  18   1 -32  37
```

```
> set.seed(43)
> round(runif(10, -50, 50))
```

```
 [1]  -1  41 -44  21 -19   4  18   1 -32  37
```

ou as funções `floor` e `ceiling` que arredondam para baixo e para cima respectivamente.

```
> set.seed(43)
> floor(runif(10, -50, 50))
```

```
 [1]  -2  41 -45  20 -19   4  17   0 -32  37
```

```
> set.seed(43)
> ceiling(runif(10, -50, 50))
```

```
 [1]  -1  42 -44  21 -18   5  18   1 -31  38
```

A função `sample` também serve para gerar amostras a partir de uma distribuição uniforme. Entretanto, a sintaxe é diferente e os resulatos são números inteiros.

```
> set.seed(43)
> sample(x = -50:50, size = 10, replace = T)
```

```
 [1]  -2  41 -45  21 -19   4  18   1 -32  38
```

O argumento `x` é o conjunto de elementos a serem amostrados, o argumento `size` é o tamanho da amostra, e o argumento `replace` (abreviado como `r`) indica se a amostra é com ou sem reposição. Em amostras com reposição (`replace = T`) um dado elemento pode ser selecionado mais de uma vez. Em amostras sem reposição um dado elemento pode ser selecionado no máximo uma vez.

Reparem que os resultados da função `sample` são iguais aos resultados da função `runif` quando nesta última os número negativos são arredondados para baixo e os positivos para cima.

Distribuição normal

A distribuição normal é caracterizada por dois parâmetros que são a média e o desvio padrão. Para uma amostra com base em uma distribuição normal com média 100 e desvio padrão 10, o valor mais provável será 100, e a probabilidade será menor quanto mais distante seja um valor da média (graficamente, a variação das probabilidades corresponde à famosa campana de Gauss, cujo pico coincide com a média).

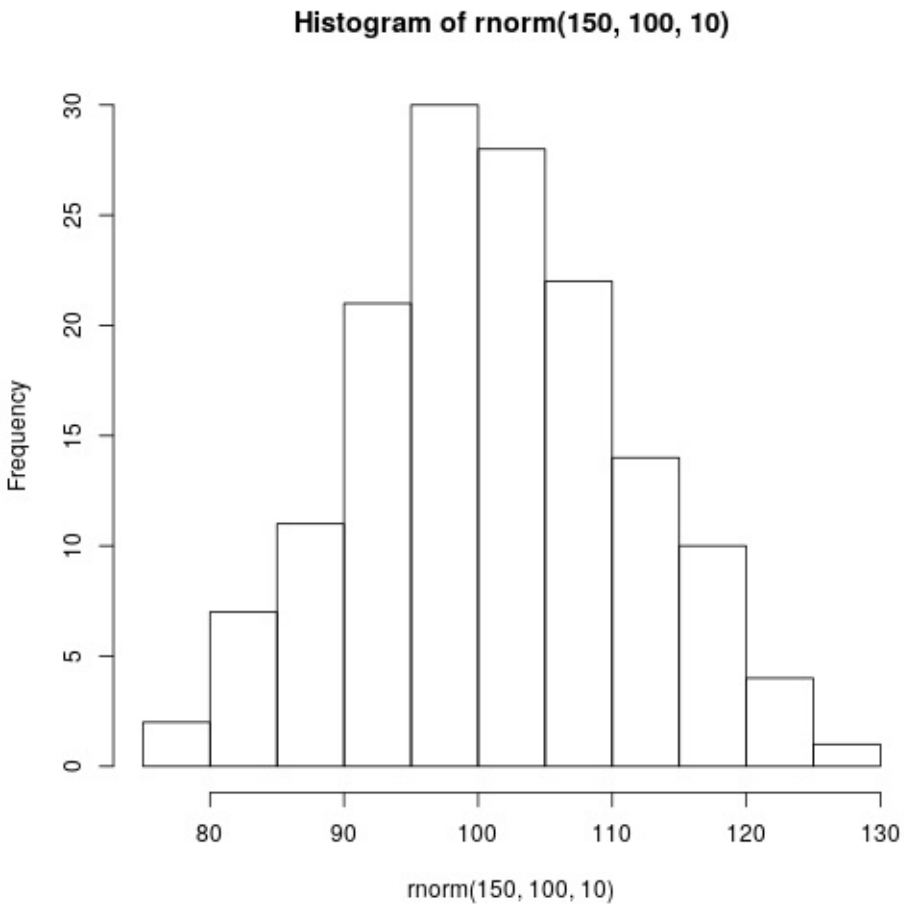
Supondo que o peso dos animais de uma dada população segue uma distribuição normal com média (`mean`) 100 e desvio padrão (`sd`) 10, podemos sortear 20 pesos dessa população com a função `rnorm` .

```
> set.seed(43)
> rnorm(n = 20, mean = 100, sd = 10)
```

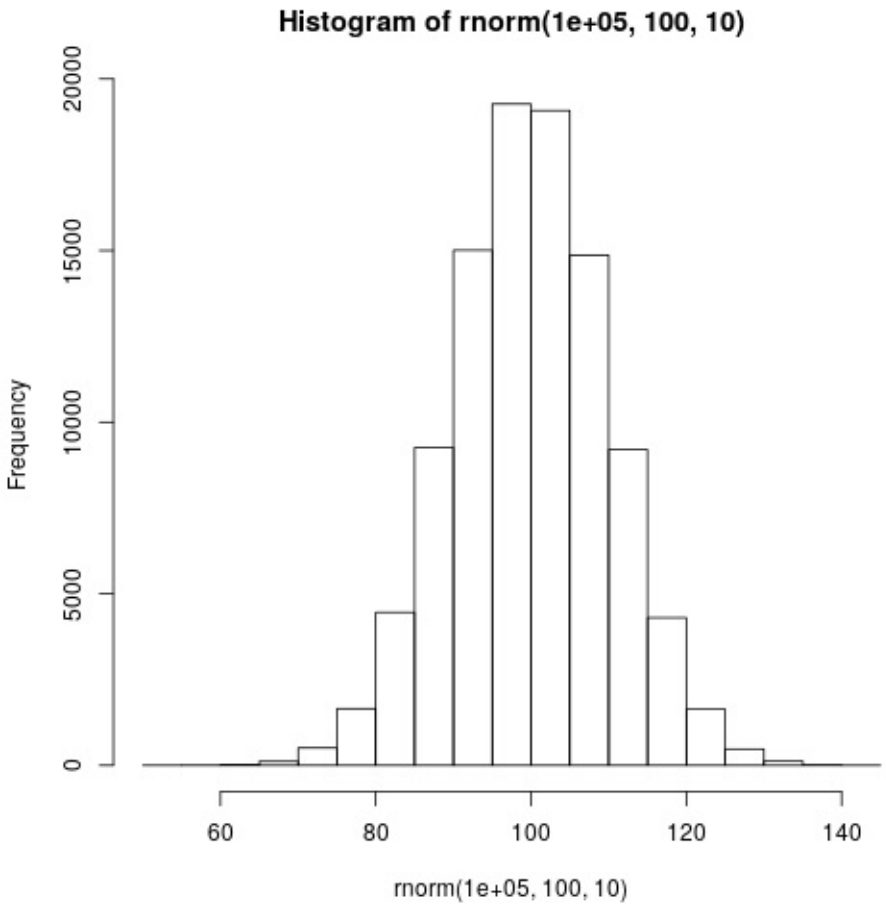
```
 [1]  99.62486  84.25396  95.14032 104.65186  90.95902  97.22567
 [7] 103.86434  99.39596  93.13820  80.93863 118.03760  90.33127
[13]  96.46882 111.06937 105.66324 120.64309 114.69278  83.48500
[19] 102.02636  92.79020
```

Se graficarmos a distribuição das frequências dos valores soretados, veremos que o formato se aproxima ao da campana de Gauss. Quanto mais números soretarmos, o formato da distribuição das frequências será mais próximo ao da campana de Gauss.

```
> set.seed(43)
> hist(rnorm(150, 100, 10))
```



```
> set.seed(43)
> hist(rnorm(100000, 100, 10))
```



Distribuição de Poisson

A distribuição de Poisson é caracterizada por um único parâmetro *lambda*, equivalente à média e ao desvio padrão (a média e o desvio padrão são iguais). Esta distribuição expressa a probabilidade de ocorrência de um dado número de eventos por unidade de espaço ou tempo. Para obtermos uma amostra com base em uma distribuição de Poisson devemos especificar o tamanho da amostra `n` é o parâmetro `lambda` na função `rpois`.

Se a média de cães por domicílio é 0.88 em um dado município, podemos simular o número de cães em 20 domicílios, usando a distribuição de Poisson correspondente.

```
> set.seed(43)
> rpois(n = 20, lambda = .88)
```

```
 [1] 1 2 0 1 0 1 1 1 0 2 0 0 1 0 1 1 0 0 0 0
```

Distribuição binomial

A distribuição binomial é caracterizada pelos parâmetros *size* e *p*, sendo *size* uma sequência de experimentos de Bernoulli independentes. Nesses experimentos os possíveis resultados são sucesso ou fracasso, e os sucessos ocorrem com probabilidade *p*. A independência implica que o resultado de um experimento não está condicionado pelo resultado de outros experimentos. Com a função `rbinom` podemos especificar esses parâmetros, assim como o tamanho da amostra `n` (a literatura estatística costuma usar a letra *n* para referir-se ao número de experimentos de Bernoulli, porém, na função `rbinom`, `n` representa o tamanho da amostra e `size` é o número de experimentos).

Se a soroprevalência de uma dada doença é de 15%, podemos criar 10 grupos hipotéticos cada um com 100 animais testados, e a cada grupo atribuir um número animais positivos de acordo com a distribuição binomial correspondente.

```
> set.seed(43)
> rbinom(n = 10, size = 100, p = .15)
```

```
 [1] 15 20 10 17 13 15 17 15 12 19
```

Outras formas de atribuir as probabilidades

A função `sample` permite-nos amostrar sem reposição (por padrão `rapleace = F`) e atribuir probabilidades de seleção diferentes para cada elemento. Em ambos casos, a distribuição probabilística de base deixa de ser uniforme.

```
> set.seed(43)
> sample(-50:50, 10)
```

```
 [1]  -2  41 -45  19 -20   2  14  -3 -33  30
```

A atribuição de probabilidades é mediada pelo argumento `prob`, que deve ser um vetor de probabilidades com comprimento igual ao número de elementos no conjunto do qual se faz a amostragem. Se a soma das probabilidades em `prob` não é igual a 1, a função normaliza internamente essas probabilidades para que o seja.

```
> set.seed(43)
> sample(1:3, 20, r = T, prob = c(.9, .4, .5))
```

```
 [1] 1 2 1 3 1 3 3 3 1 2 1 1 3 1 1 3 1 1 1 1
```

Combinação

As vezes precisamos gerar o total de possíveis combinações de tamanho `m`, de um conjunto `x`. A função `ncombn` faz exatamente isso.

```
> combn(x = letters[1:3], m = 2)
```

```
     [,1] [,2] [,3]
[1,] "a"  "a"  "b"
[2,] "b"  "c"  "c"
```

```
> combn(letters[1:5], 3)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,] "a"  "a"  "a"  "a"  "a"  "a"  "b"  "b"  "b"  "c"
[2,] "b"  "b"  "b"  "c"  "c"  "d"  "c"  "c"  "d"  "d"
[3,] "c"  "d"  "e"  "d"  "e"  "e"  "d"  "e"  "e"  "e"
```

Dados *n* conjuntos de elementos, a função `expand.grid` gera um data frame com todas as possíveis combinações

```
> expand.grid(letters[1:3], 1:2)
```

```
  Var1 Var2
1    a    1
2    b    1
3    c    1
4    a    2
5    b    2
6    c    2
```

```
> expand.grid(letters[1:3], 1:2, LETTERS[1:2])
```

```
   Var1 Var2 Var3
1     a    1    A
2     b    1    A
3     c    1    A
4     a    2    A
5     b    2    A
6     c    2    A
7     a    1    B
8     b    1    B
9     c    1    B
10    a    2    B
11    b    2    B
12    c    2    B
```

Outra função que serve para criar combinações é `outer`, que com dois conjuntos de elementos, gera todas as possíveis combinações, sendo que os elementos são combinados usando uma determinada função.

```
> outer(letters[1:3], 1:3, FUN = paste)
```

```
     [,1]  [,2]  [,3]
[1,] "a 1" "a 2" "a 3"
[2,] "b 1" "b 2" "b 3"
[3,] "c 1" "c 2" "c 3"
```

```
> outer(letters[1:3], 1:3, paste, sep = '-')
```

```
     [,1]  [,2]  [,3]
[1,] "a-1" "a-2" "a-3"
[2,] "b-1" "b-2" "b-3"
[3,] "c-1" "c-2" "c-3"
```

```
> outer(1:3, 1:3, '*')
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    4    6
[3,]    3    6    9
```

Gráficos

Gramática de gráficos

Existem várias abordagens para criar gráficos no R e o foco aqui é a *gramática de gráficos* implementada no pacote ggplot2. A ideia da gramática dos gráficos é usar geometrias com propriedades estéticas para representar dados em um sistema de coordenadas. Para entendermos melhor a gramática de gráficos, criemos um banco fictício de casos notificados e confirmados durante 200 semanas, com o sexo e o local de ocorrência de cada caso.

```
> set.seed(12)
> casos <- data.frame(
+   semana = 1:200,
+   notificados = round(seq(50, 250, l = 200) + runif(200, -40, 40)),
+   confirmados = round(seq(5, 25, l = 200) + runif(200, -5, 5)),
+   sexo = c(sample(c('M', 'F'), 100, replace = T),
+            sample(c('M', 'F'), 100, replace = T, prob = c(.8, .2))),
+   local = rep(c('A', 'B'), each = 50))
```

```
> str(casos, v = 1)
```

```
'data.frame':    200 obs. of  5 variables:
 $ semana     : int  1 2 ...
 $ notificados: num  16 76 ...
 $ confirmados: num  10 2 ...
 $ sexo       : Factor w/ 2 levels "F","M": 2 2 ...
 $ local      : Factor w/ 2 levels "A","B": 1 1 ...
```

```
> head(casos)
```

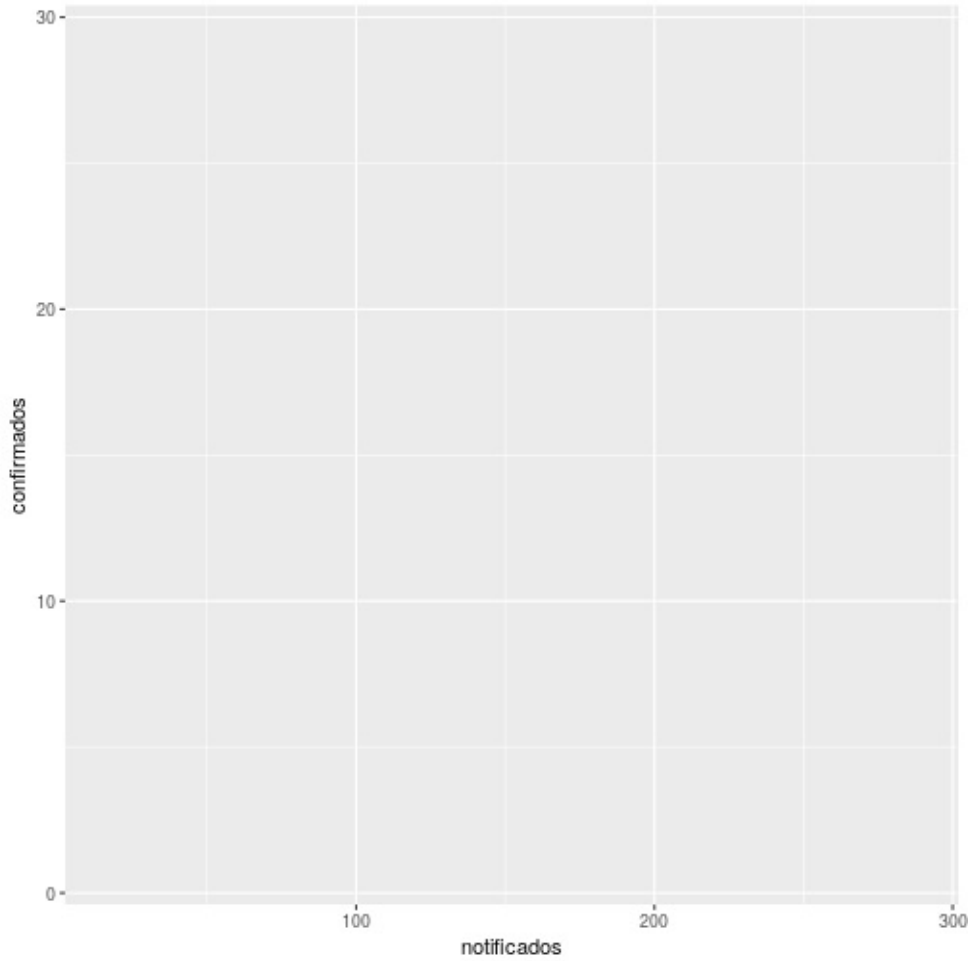
```
  semana notificados confirmados sexo local
1      1          16          10    M     A
2      2          76           2    M     A
3      3          87           5    M     A
4      4          35           9    F     A
5      5          28          10    F     A
6      6          18           7    M     A
```

A partir do banco `casos` podemos graficar a relação entre o número de casos notificados e confirmados usando a função `ggplot`. O primeiro passo é a especificação do banco de dados no argumento `data`, e das variáveis a serem representadas no argumento `mapping`. Este último argumento é definido com a função `aes` que permite especificar as propriedades estéticas que mapearão (representarão) as variáveis.

```
> library(ggplot2)
```

```
Find out what's changed in ggplot2 at
http://github.com/tidyverse/ggplot2/releases.
```

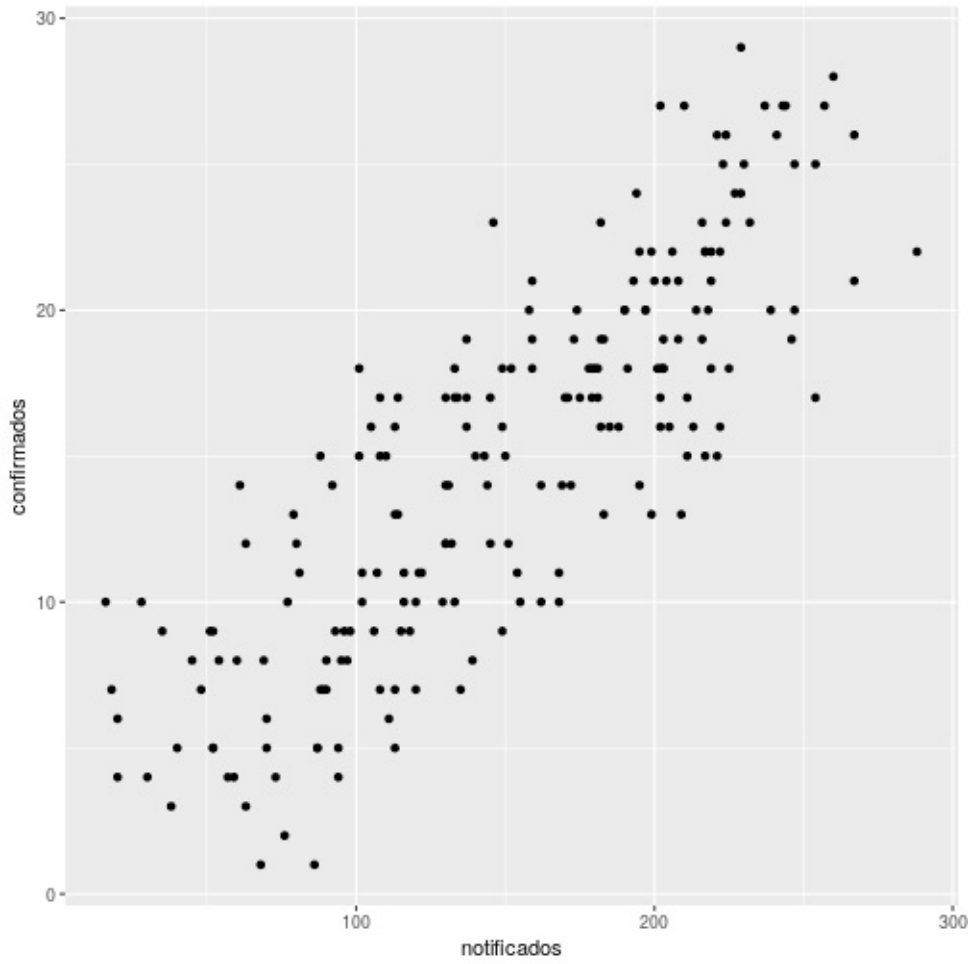
```
> ggplot(data = casos, mapping = aes(x = notificados, y = confirmados))
```



No gráfico acima, as variáveis `notificados` e `confirmados` foram mapeadas nos eixos `x` e `y` que são duas propriedades estéticas. Entretanto, o valor dessas variáveis para cada observação (semana) não foi graficado porque não especificamos nenhuma geometria. Se

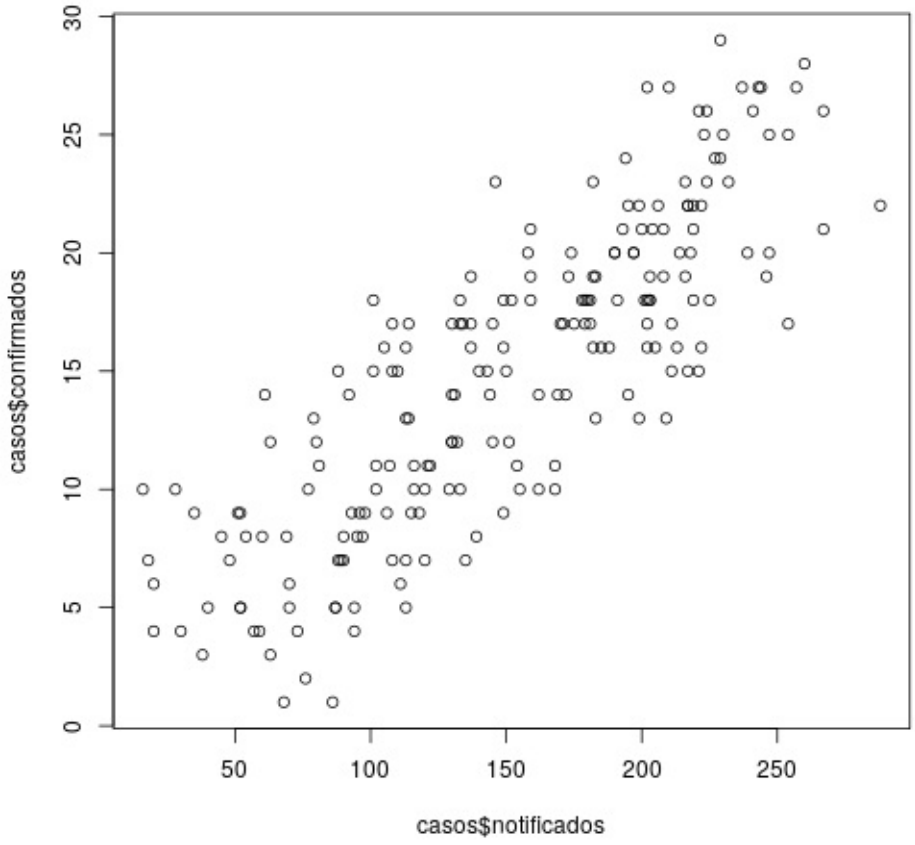
o objetivo é criar um gráfico de dispersão (scatterplot) podemos acrescentar a função `geom_point` (reparem no sinal + que acrescenta a função) para usar o ponto como geometria.

```
> ggplot(data = casos, mapping = aes(notificados, confirmados)) +
+   geom_point()
```



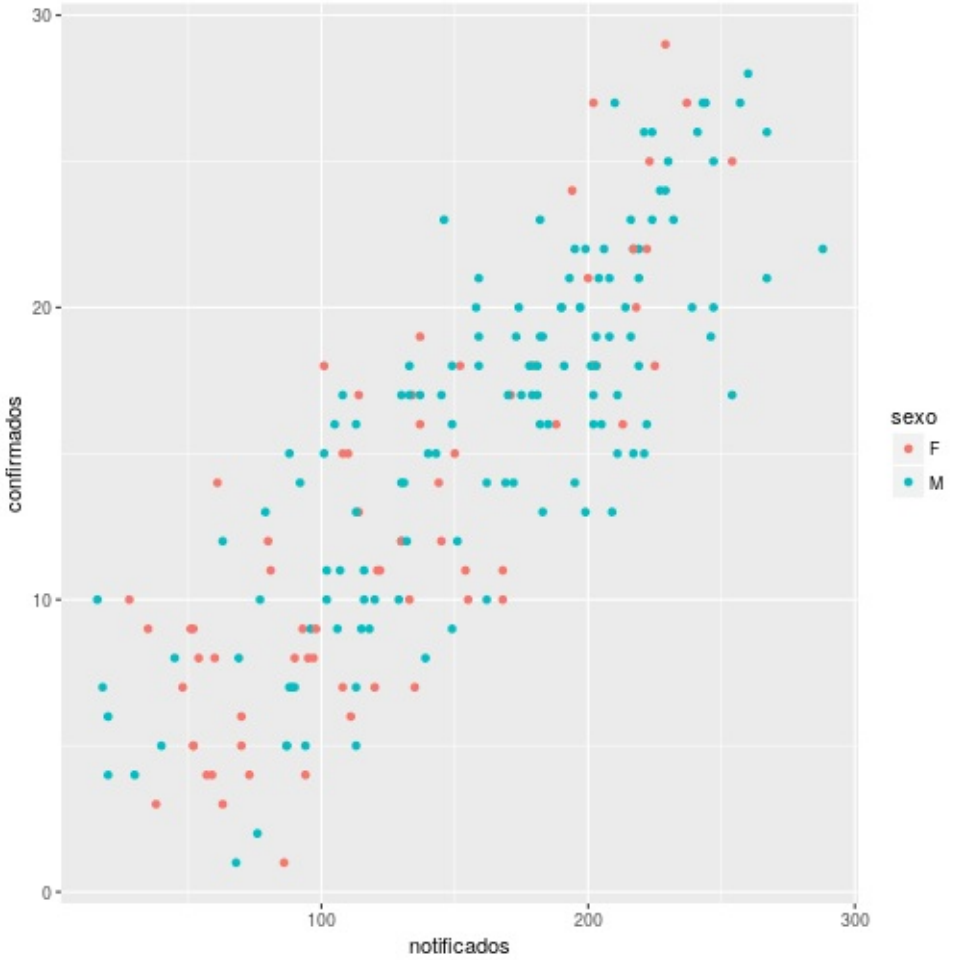
Em capítulos anteriores usamos a função `plot` - que não segue os princípios da gramática de gráficos - e aqui podemos usá-la de novo para criar um gráfico equivalente ao anterior.

```
> plot(casos$notificados, casos$confirmados)
```



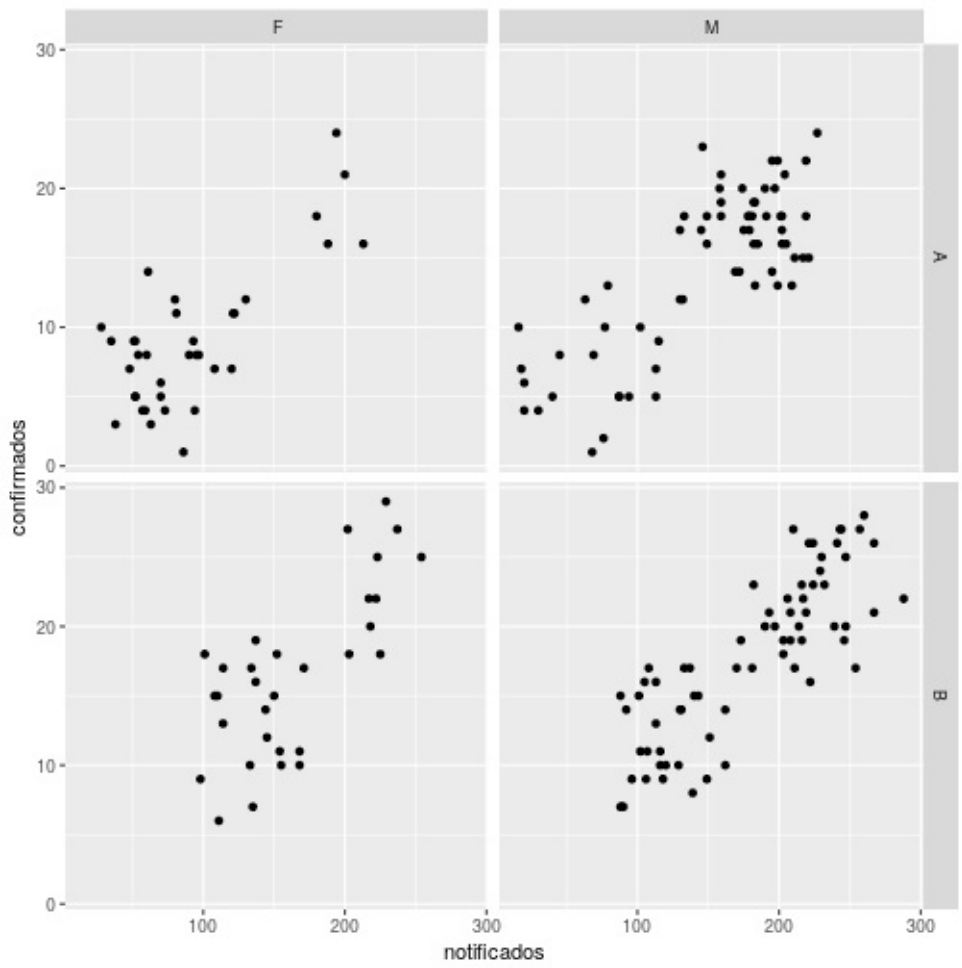
Por que usar a função `ggplot` se a função `plot` é muito mais simples? Porque é comum criar gráficos com mais informações (legendas, relações multivariadas, etc.) e nesses casos a função `ggplot` é mais versátil e fácil de usar. Por exemplo, para representar o sexo com cores diferentes e criar a legeneda correspondente, só precisamos mapear a variável `sexo` na propriedade estética `color`.

```
> ggplot(data = casos, aes(notificados, confirmados, color = sexo)) +
+   geom_point()
```



Embora seja possível criar um gráfico equivalente com a função `plot`, o nível de dificuldade é maior. E conforme aumenta a complexidade dos gráficos, maior a diferença entre as abordagens. Por exemplo, como a gramática de gráficos é simples criar um gráfico da relação entre o o número de casos notificados e confirmados, condicionada no local de ocorrência e o sexo.

```
> ggplot(data = casos, aes(notificados, confirmados)) +
+   geom_point() +
+   facet_grid(local ~ sexo)
```

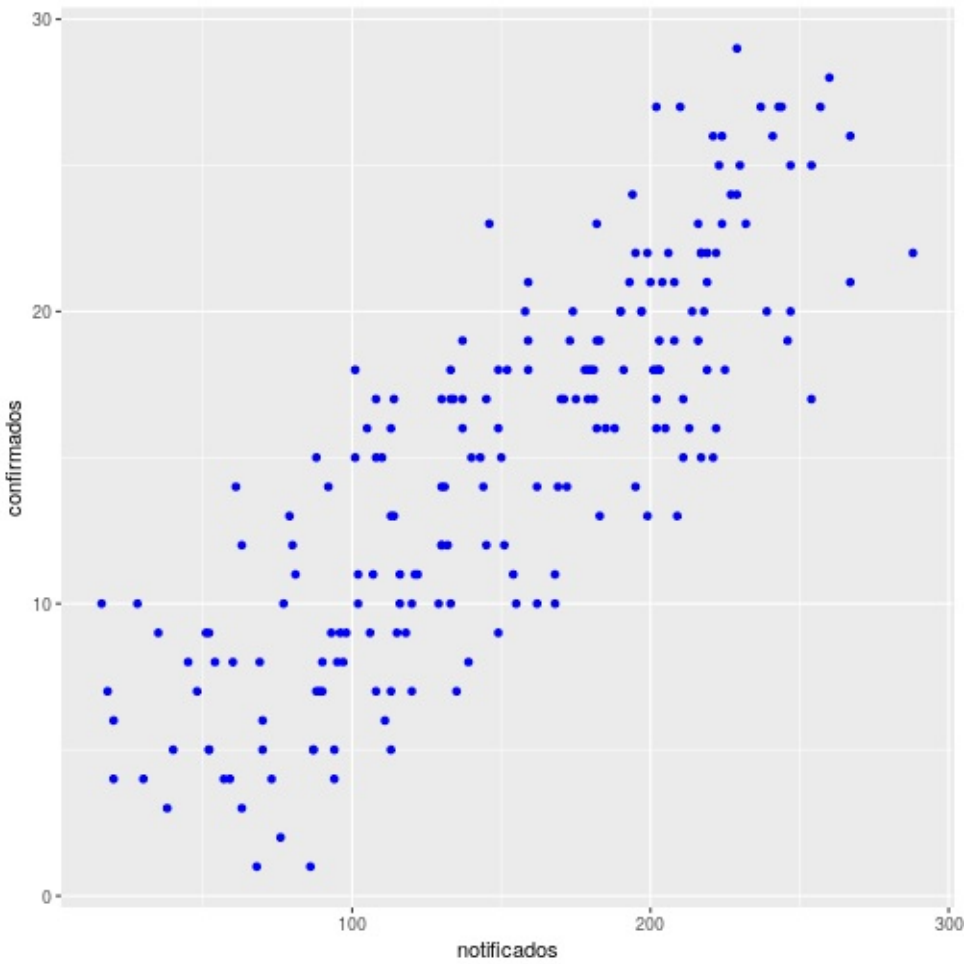


Criar um gráfico equivalente ao anterior com as funções básicas do R seria um bom exercício para quem está entediado e quer reproduzir com muitas linhas de código o que dá para fazer com poucas. Cabe mencionar o lattice que um pacote básico do R e implementa uma outra abordagem também apropriada para gráficos de diversa complexidade. Porém, o foco aqui é o ggplot2.

Mapeamento de variáveis em propriedades estéticas

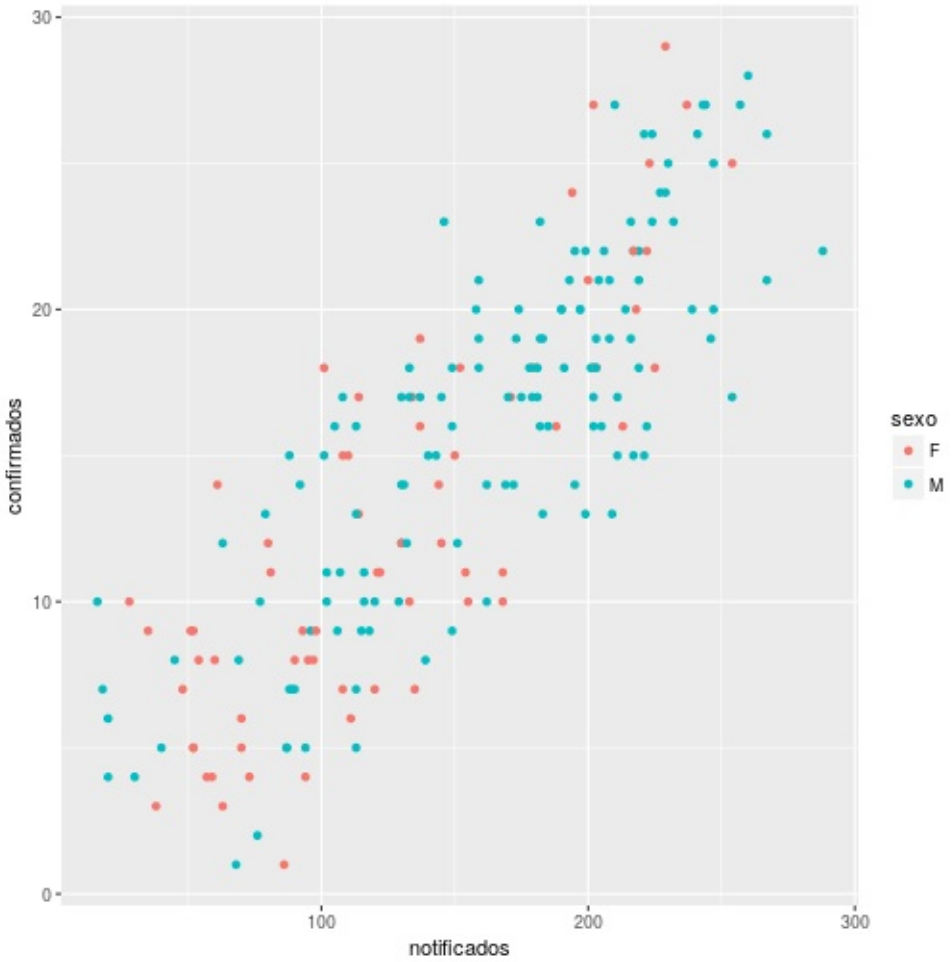
Uma das principais fontes de confusão no uso do ggplot2 refere-se à forma de usar as propriedades estéticas. Uma propriedade estética pode ser definida como igual a um determinado valor ou mapear uma variável. Por exemplo, a propriedade estética `color` pode ser definida como igual a `'blue'`

```
> ggplot(data = casos, mapping = aes(notificados, confirmados)) +
+   geom_point(color = 'blue')
```

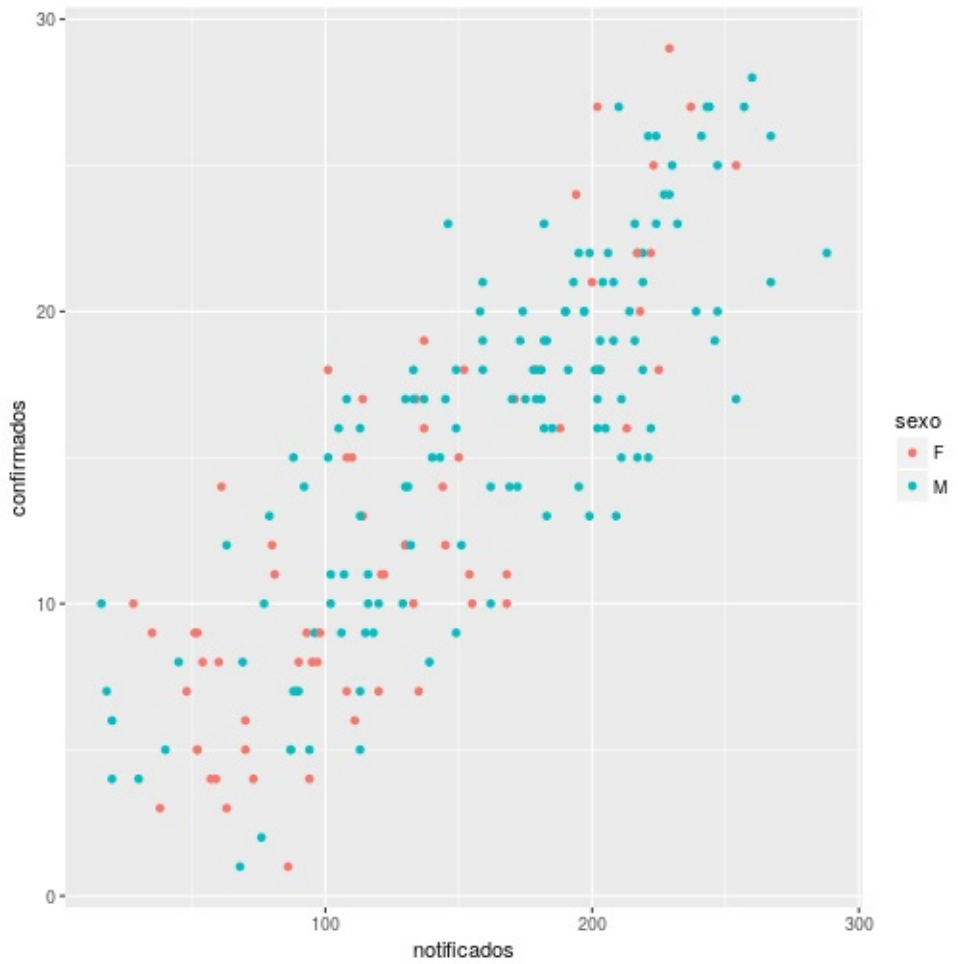


ou mapear a variável `sexo` como feito anteriormente.

```
> ggplot(data = casos, aes(notificados, confirmados, color = sexo)) +
+   geom_point()
```



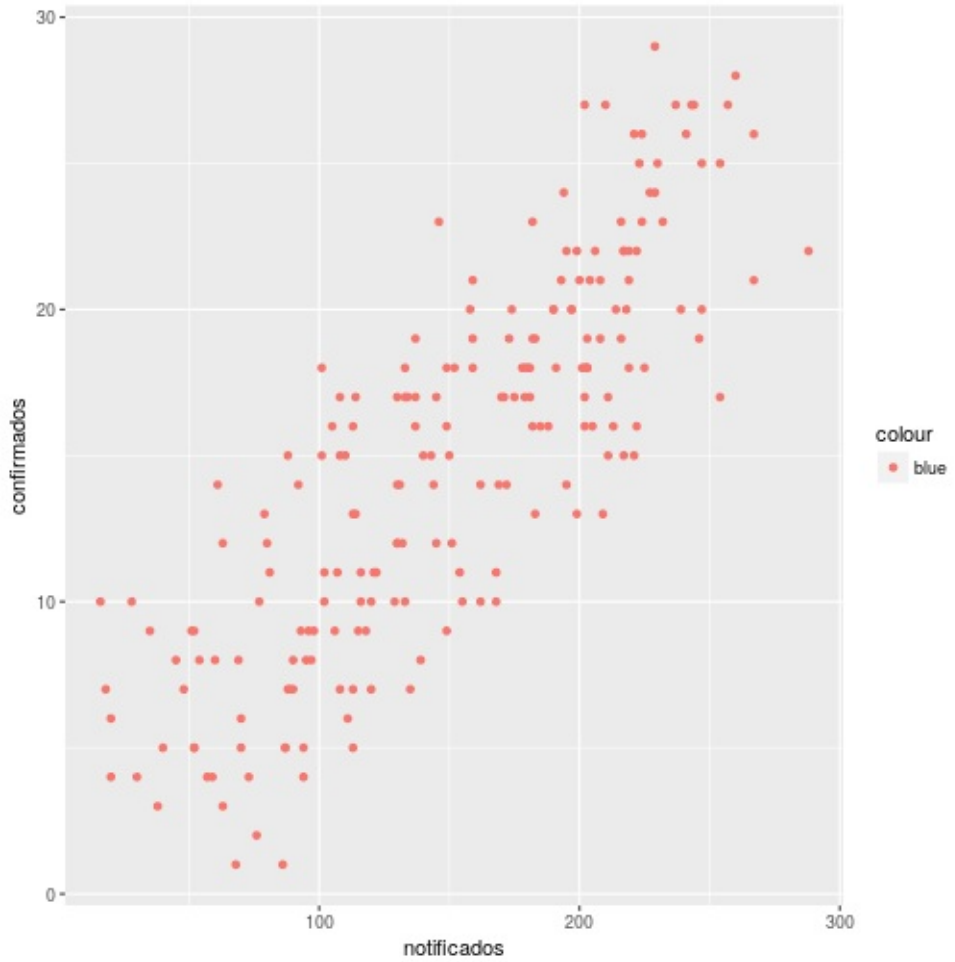

```
> ggplot(data = casos, aes(notificados, confirmados)) +
+   geom_point(aes(color = sexo))
```



A propriedade estética sempre mapeará uma variável quando usada dentro da função `aes`, que pode ir tanto na função `ggplot` como nas funções com prefixo `geom_`.

Ao usar `color = 'blue'` dentro da função `aes`, `blue` passa a ser uma variável que por não existir no banco, gera resultados sem sentido.

```
> ggplot(data = casos, aes(notificados, confirmados, color = 'blue')) +
+   geom_point()
```



Reiterando, a propriedade estética deve ir dentro da função `aes` para mapear variáveis, e nas funções com prefixo `geom_` (e fora da função `aes`) para defini-las como igual a um determinado valor.

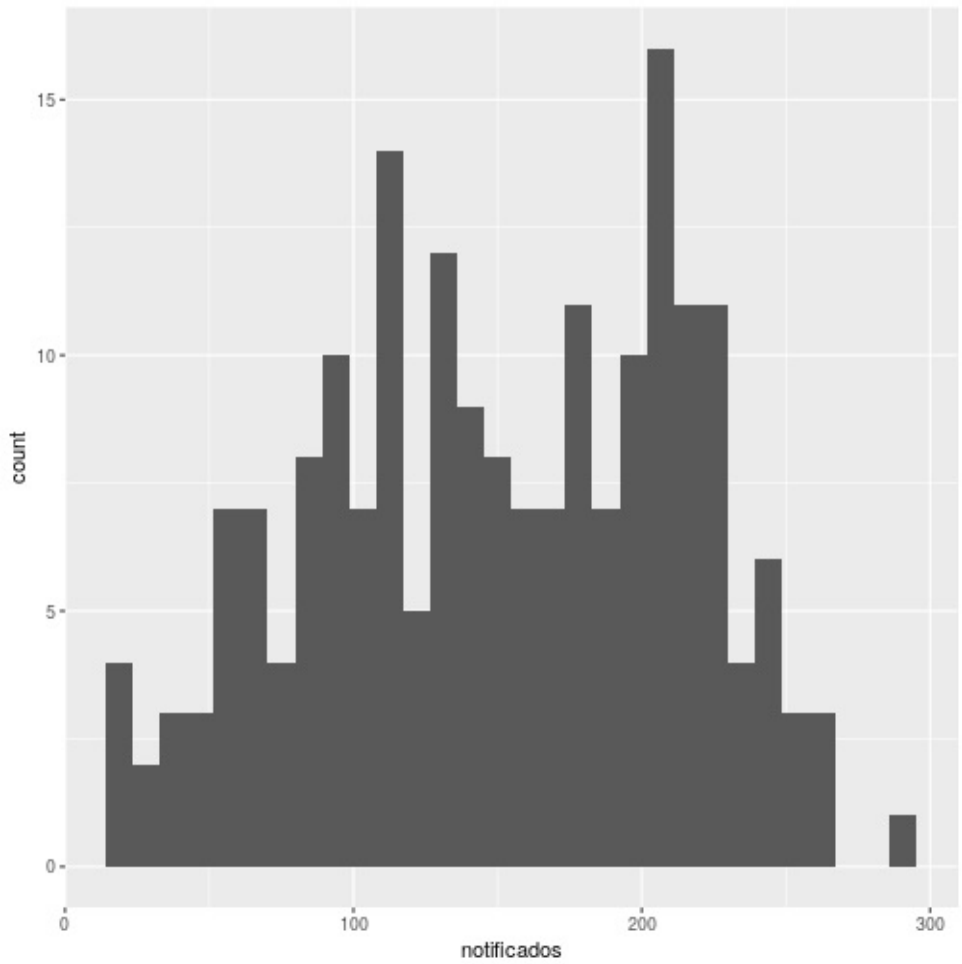
Gráficos de uma variável quantitativa

Histograma

Em um histograma, a variável contínua é dividida em intervalos e a quantidade de observações em cada intervalo é representada pela altura de uma barra.

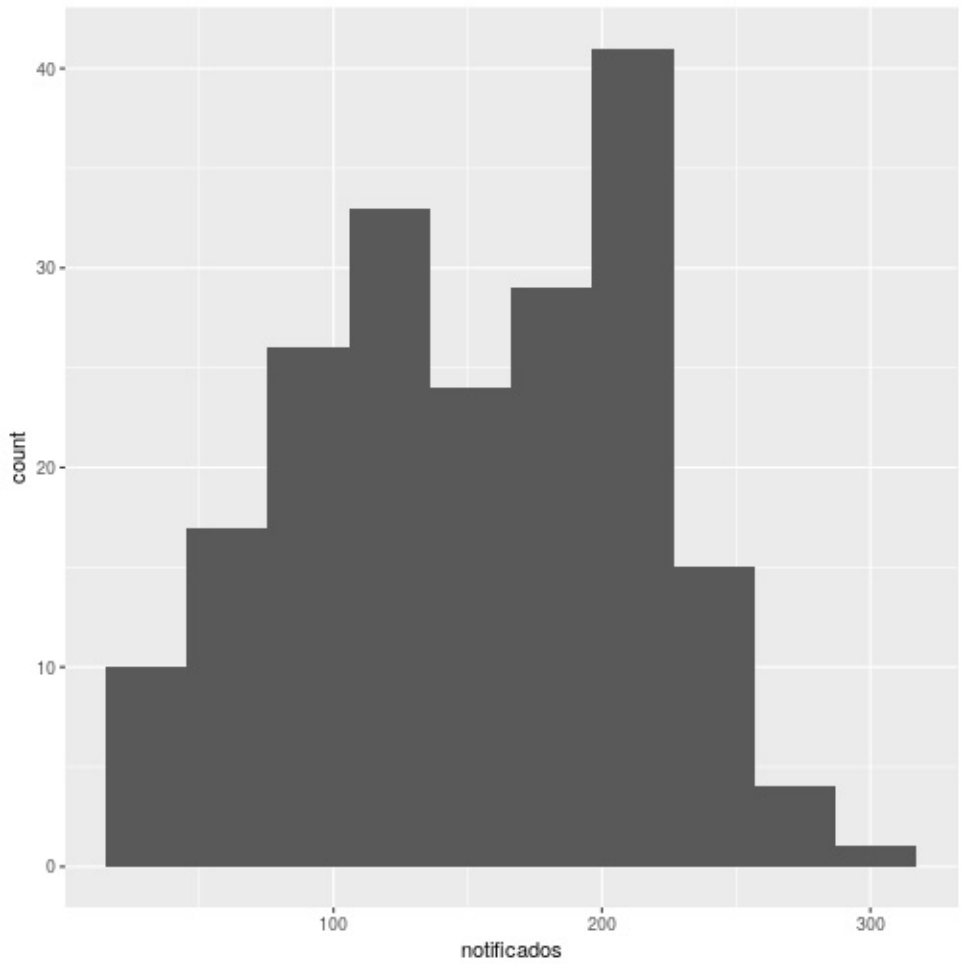
```
> his <- ggplot(casos, aes(notificados))
> his + geom_histogram()
```

```
`stat_bin()` using `bins = 30`. Pick better value with
`binwidth`.
```



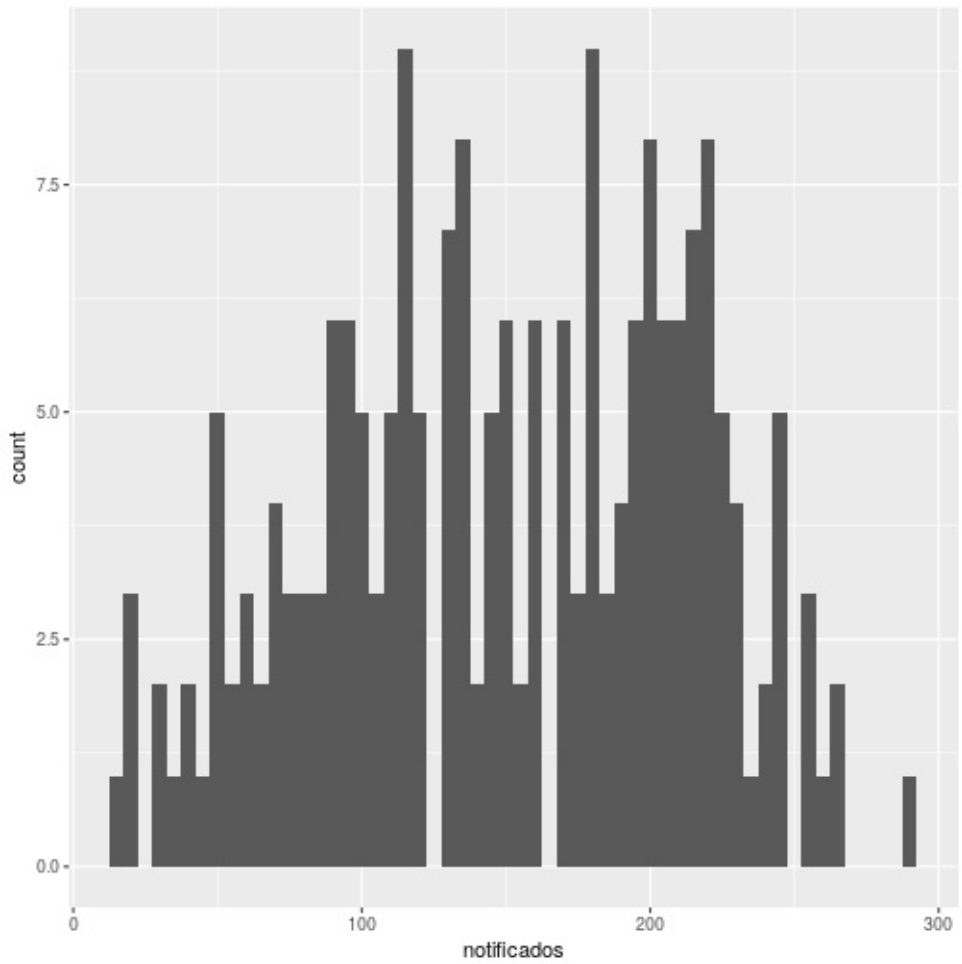
A mensagem gerada pelo comando acima indica que por padrão foi usado `bins = 30` para criar 30 intervalos. Por tanto, podemos controlar o número de intervalos com o argumento `bins`.

```
> his + geom_histogram(bins = 10)
```



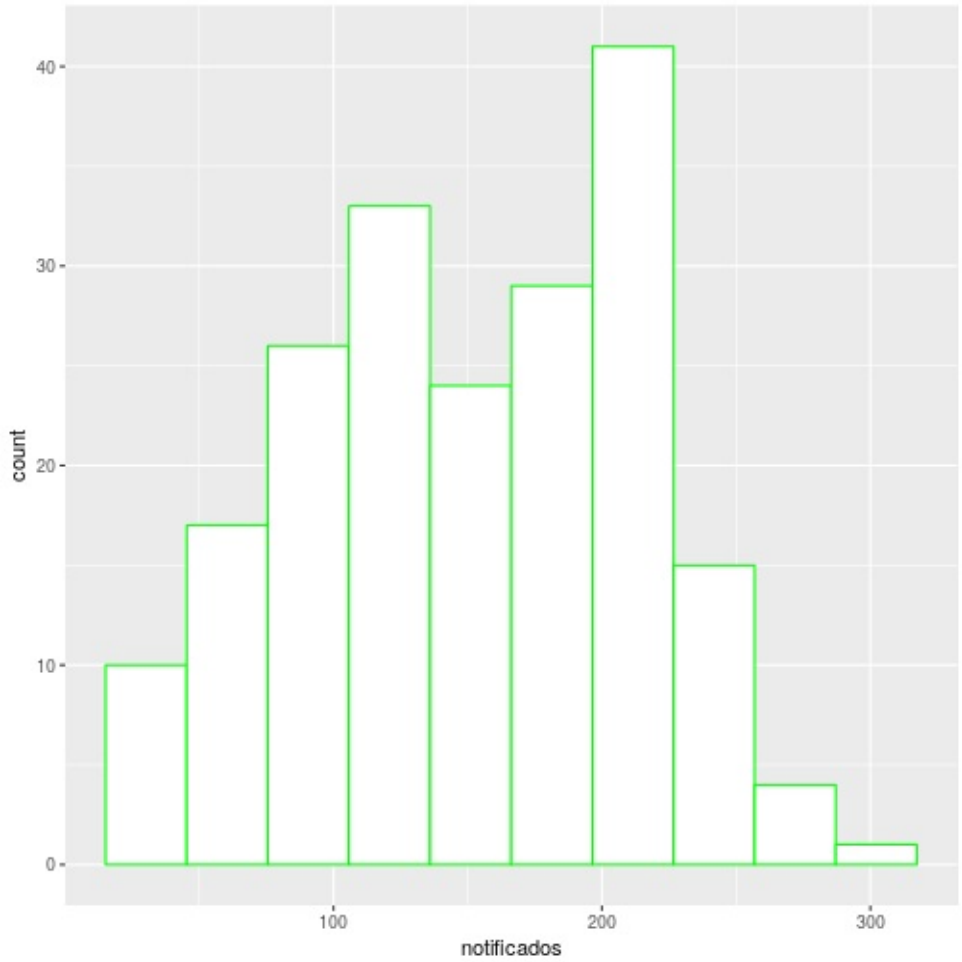
Também podemos controlar a amplitude dos intervalos.

```
> his + geom_histogram(binwidth = 5)
```



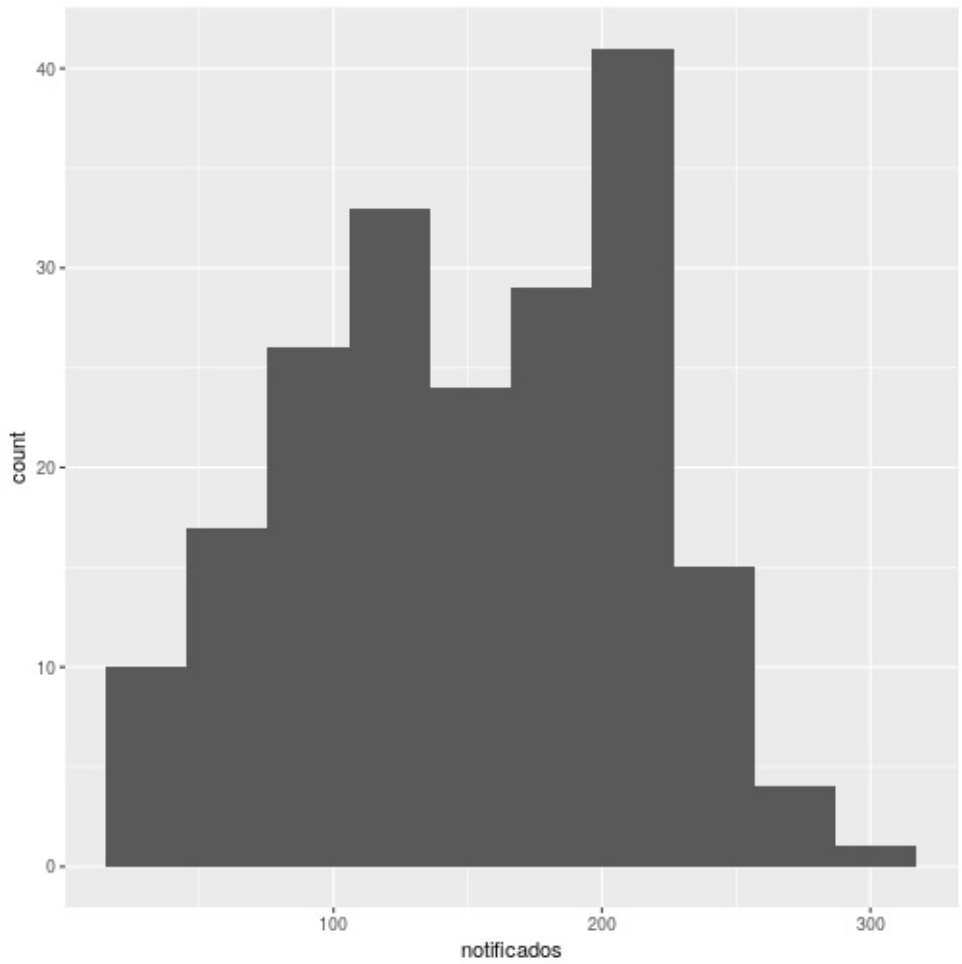
A cor de prenchimento é definida pelo argumento `fill` e a cor de contorno pelo argumento `color`.

```
> his + geom_histogram(bins = 10, fill = 'white', color = 'green')
```



Para representar a contribuição relativa de cada espécie em cada intervalo, o argumento `fill` deve ser usado como propriedade estética para mapear a variável `Species`. Em outras palavras, `fill` deve ser usado como argumento da função `aes`, e não como argumento de `geom_histogram`.

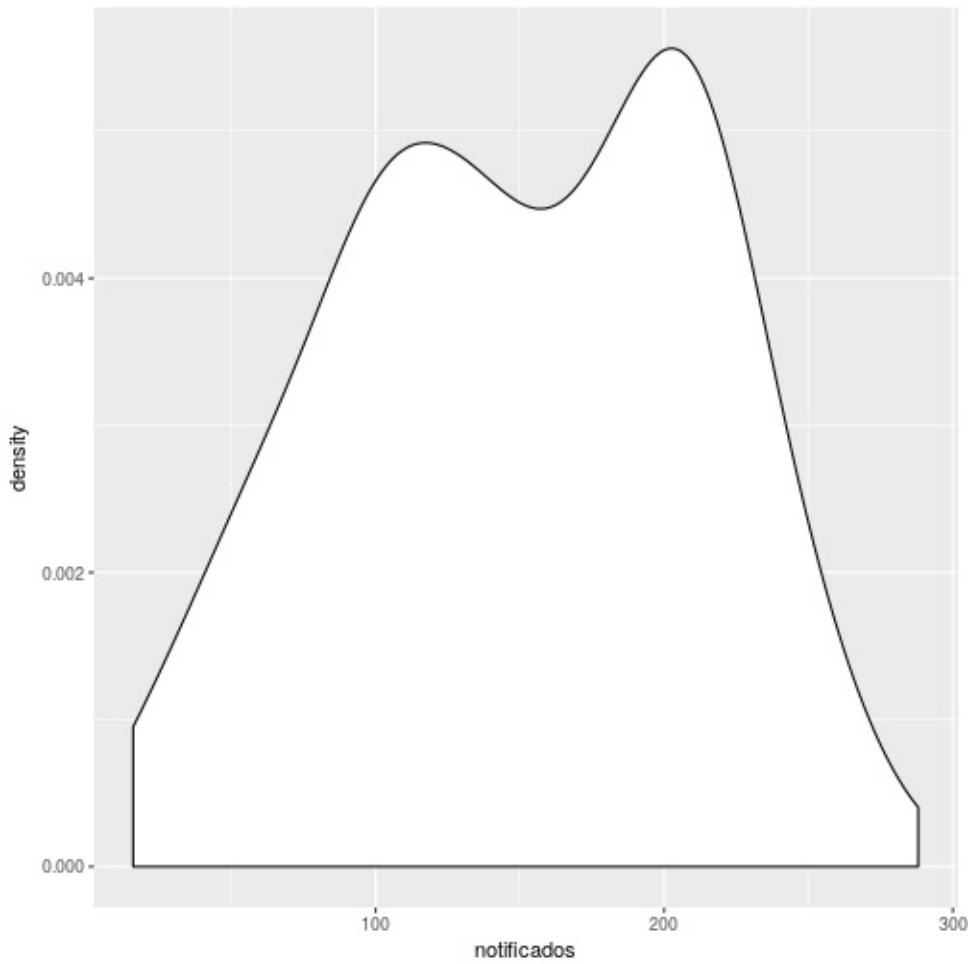
```
> his <- ggplot(casos, aes(notificados))
> his + geom_histogram(bins = 10)
```



Densidade

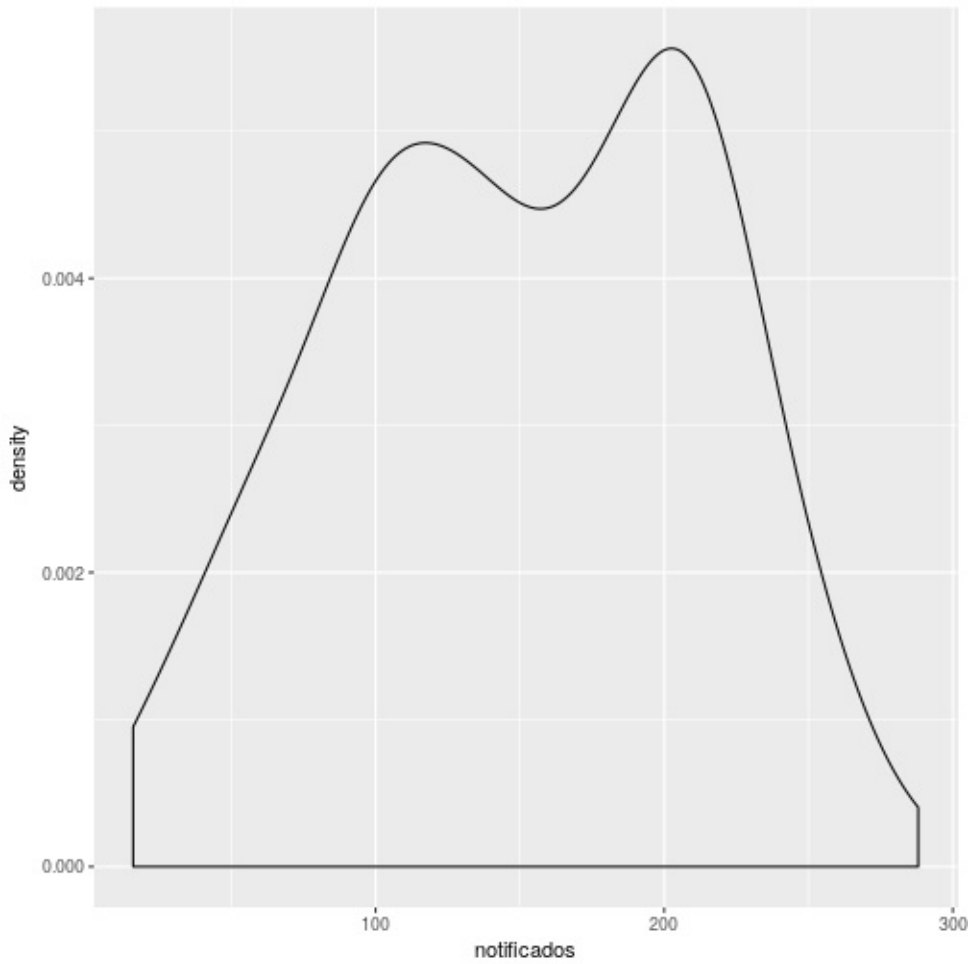
Os gráficos de densidade representam a densidade probabilística de uma variável quantitativa. A área sob a curva é uma medida de probabilidade e assim, ao dividir a variável em intervalos equidistantes, a probabilidade associada um dado intervalo será maior, quanto maior a altura da curva nesse intervalo. No gráfico abaixo, o eixo *y* representa a densidade probabilística (que pode ser maior do que 1), não a probabilidade em si, e podemos pensar que os intervalos são infinitesimalmente pequenos.

```
> den <- ggplot(casos, aes(notificados))
> den + geom_density(fill = 'white')
```



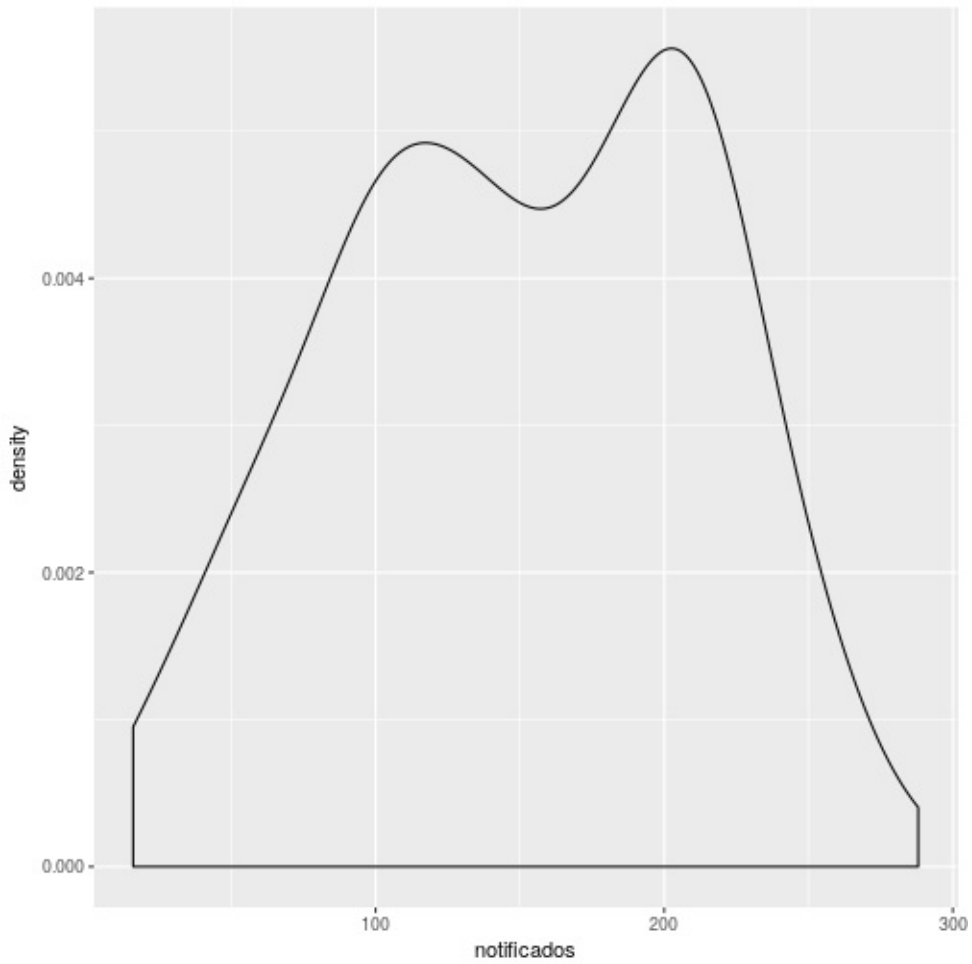
Ao mapearmos a variável `Species` na propriedade `fill`, as denisdades sobrepõem-se.

```
> den <- ggplot(casos, aes(notificados))
> den + geom_density()
```



Para vissualizar melhor cada uma das densidades podemos controlar a transparência que varia entre 0 e 1.

```
> den <- ggplot(casos, aes(notificados))
> den + geom_density(alpha = .5)
```



Outras geometrias comuns

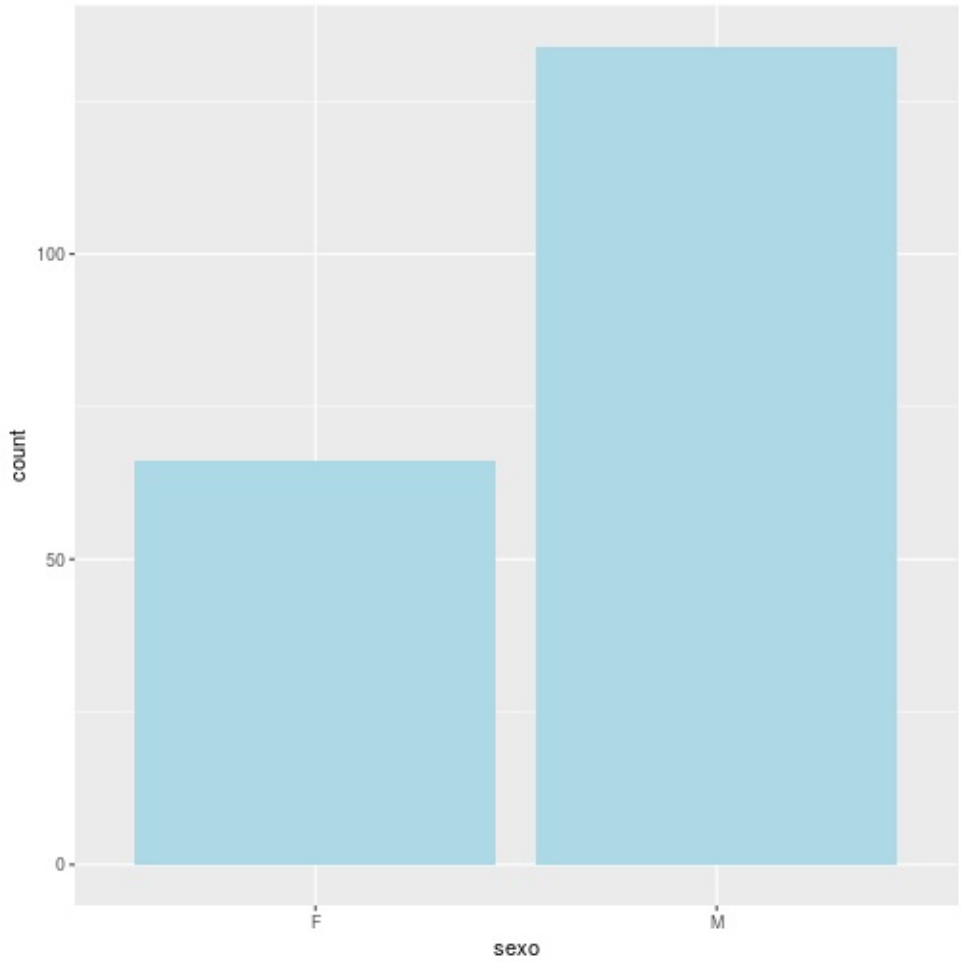
- `geom_area`
- `geom_dotplot`
- `geom_freqploy`

Gráficos de uma variável qualitativa

Barras

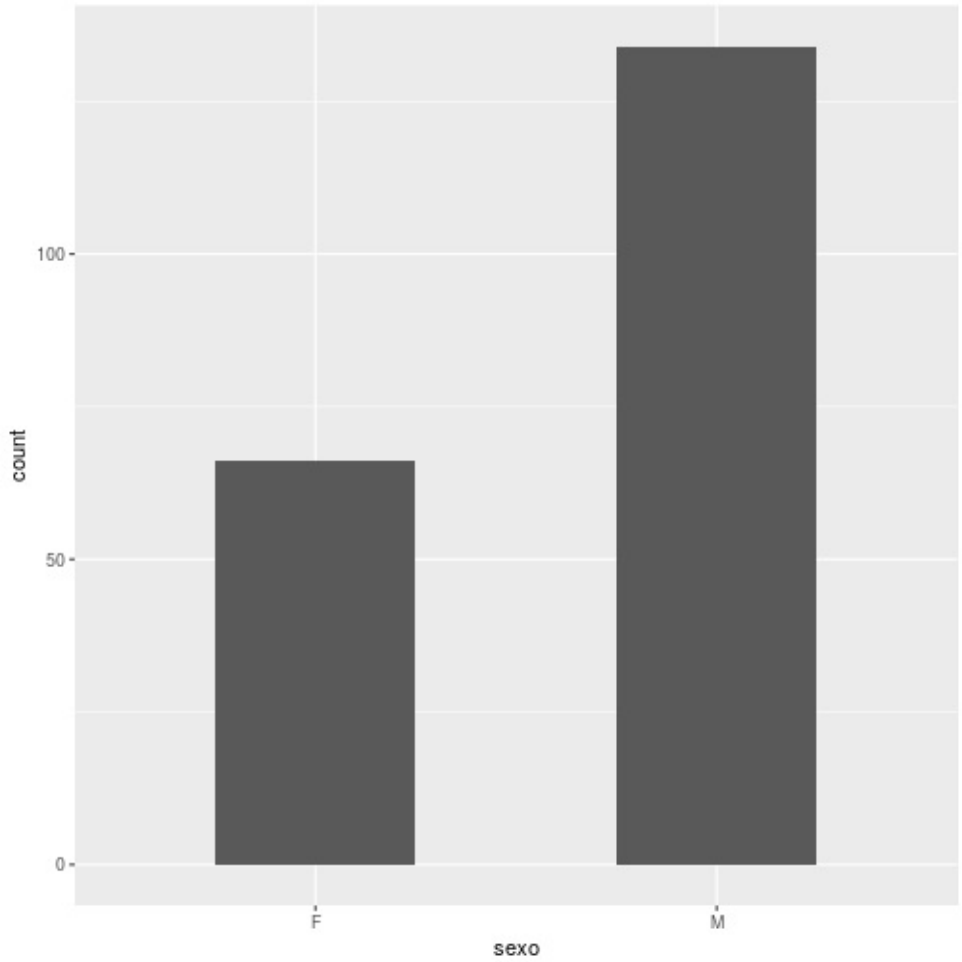
Os gráficos de barras representam cada uma das categorias de uma variável. A altura das barras representa a quantidade de observações.

```
> ggplot(casos, aes(sexo)) +
+   geom_bar(fill = 'lightblue')
```



A largura (ou separação) das barras é controlada com a propriedade `width`.

```
> ggplot(casos, aes(sexo)) +
+   geom_bar(width = .5)
```

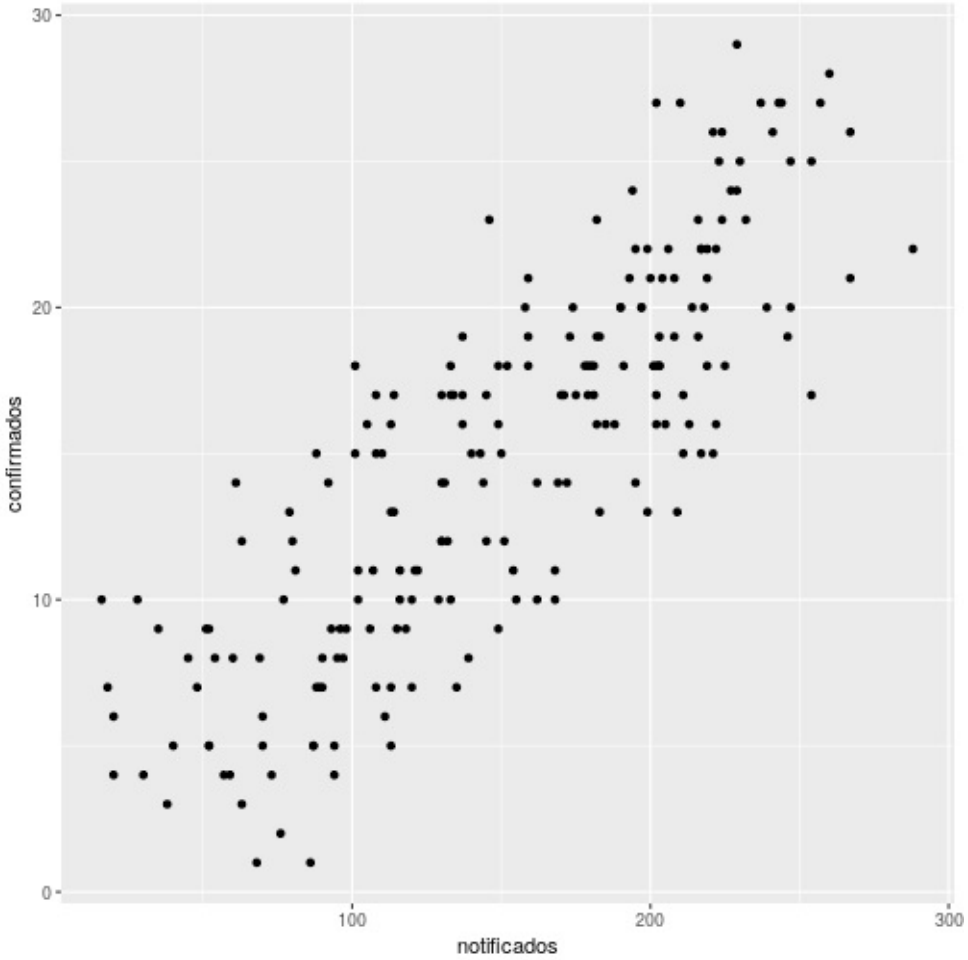


Gráficos de duas variáveis quantitativas

Pontos

Os gráficos de pontos ou de dispersão representam com pontos o valor das duas variáveis para cada observação.

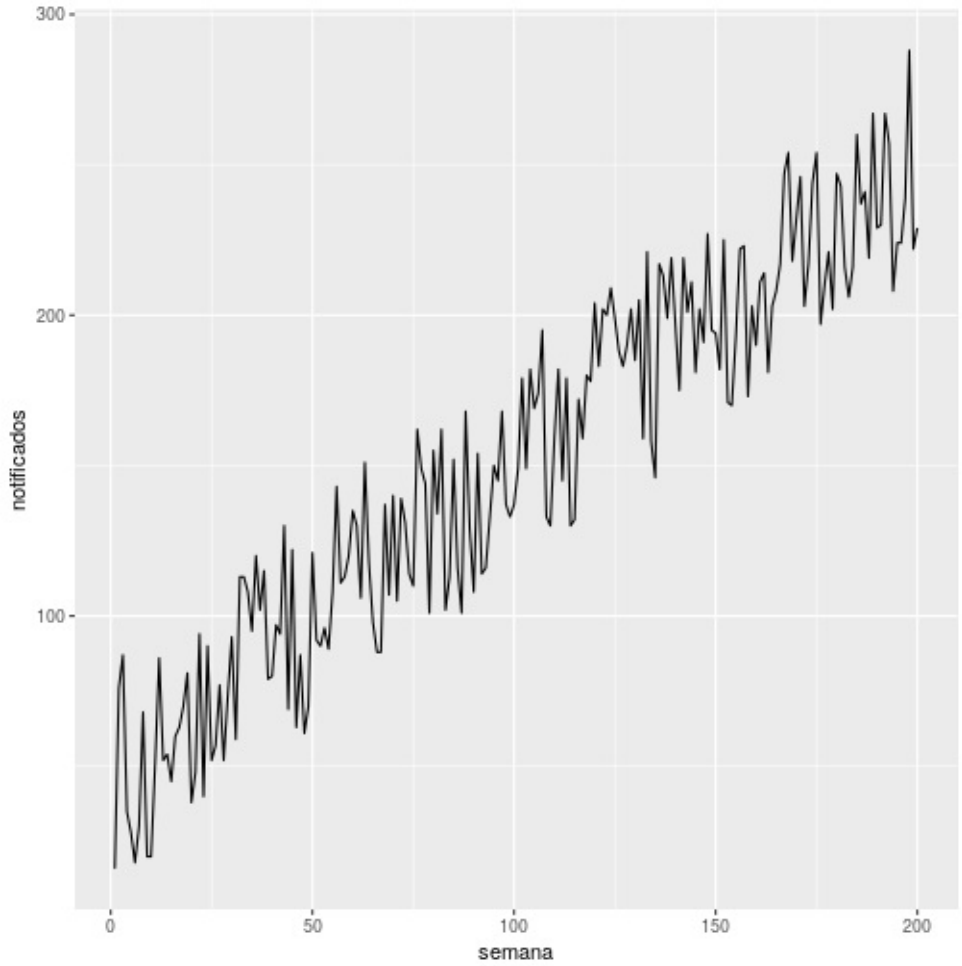
```
> ggplot(casos, aes(notificados, confirmados)) +
+   geom_point()
```



Linhas

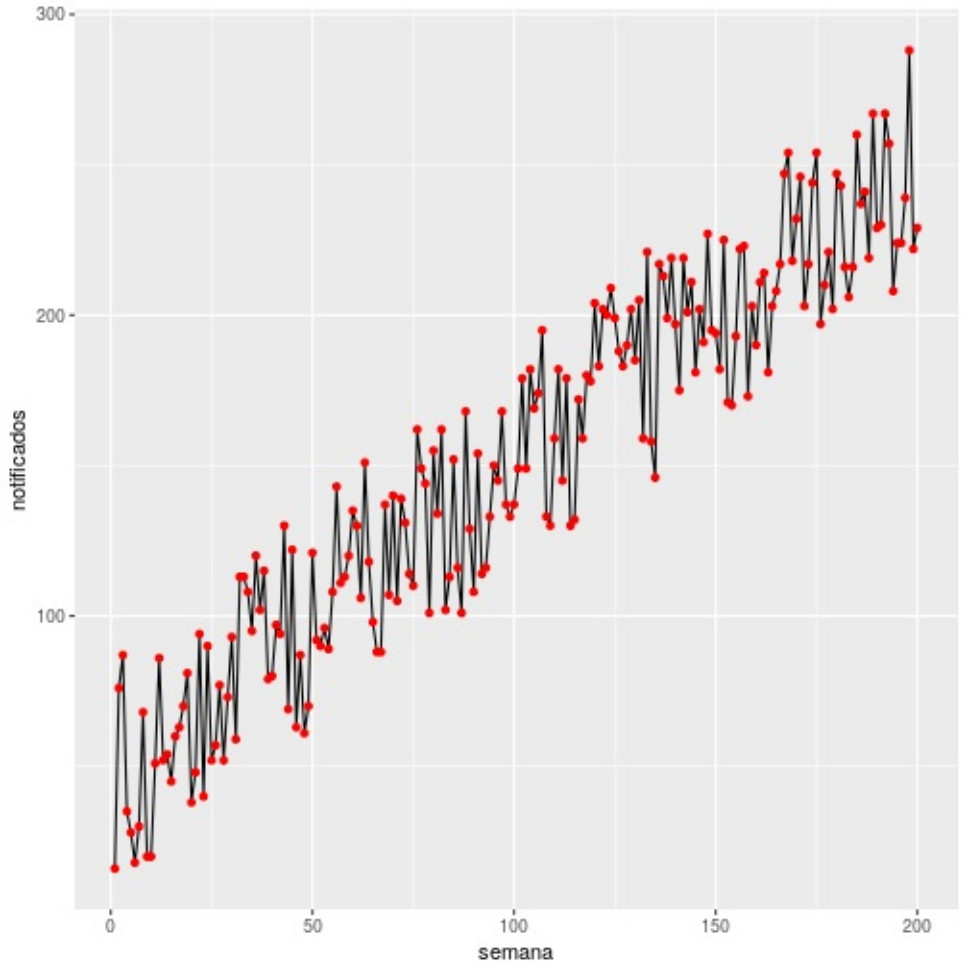
Nos gráficos de linhas, para cada valor na amplitude de uma variável, existe um valor na outra variável.

```
> ggplot(casos, aes(semana, notificados)) +
+   geom_line()
```



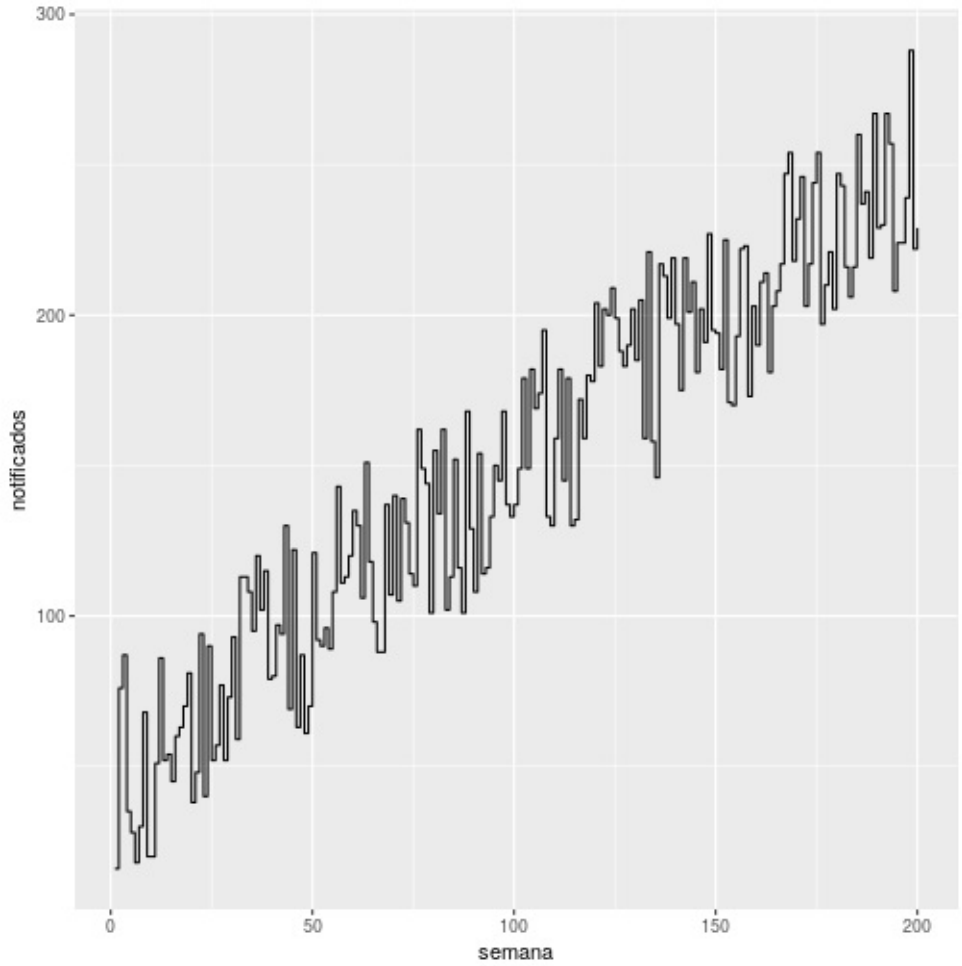
No nosso banco, há 200 semanas e embora o registro de notificações tenha sido semanal, podemos ver que o gráfico anterior associa um número de notificações a qualquer ponto no período de 200 semanas (podemos associar um número de notificações à semana 3.4581). Como as notificações não foram registradas de forma contínua, o número de notificações no transcurso de uma semana, que não foi registrado, foi interpolado linearmente (estimado a partir de uma linha reta entre o número de notificações no começo da semana e o número de notificações no começo da semana seguinte). Isto fica mais claro ao combinarmos um gráfico de pontos e de linhas.

```
> ggplot(casos, aes(semana, notificados)) +
+   geom_line() +
+   geom_point(color = 'red')
```

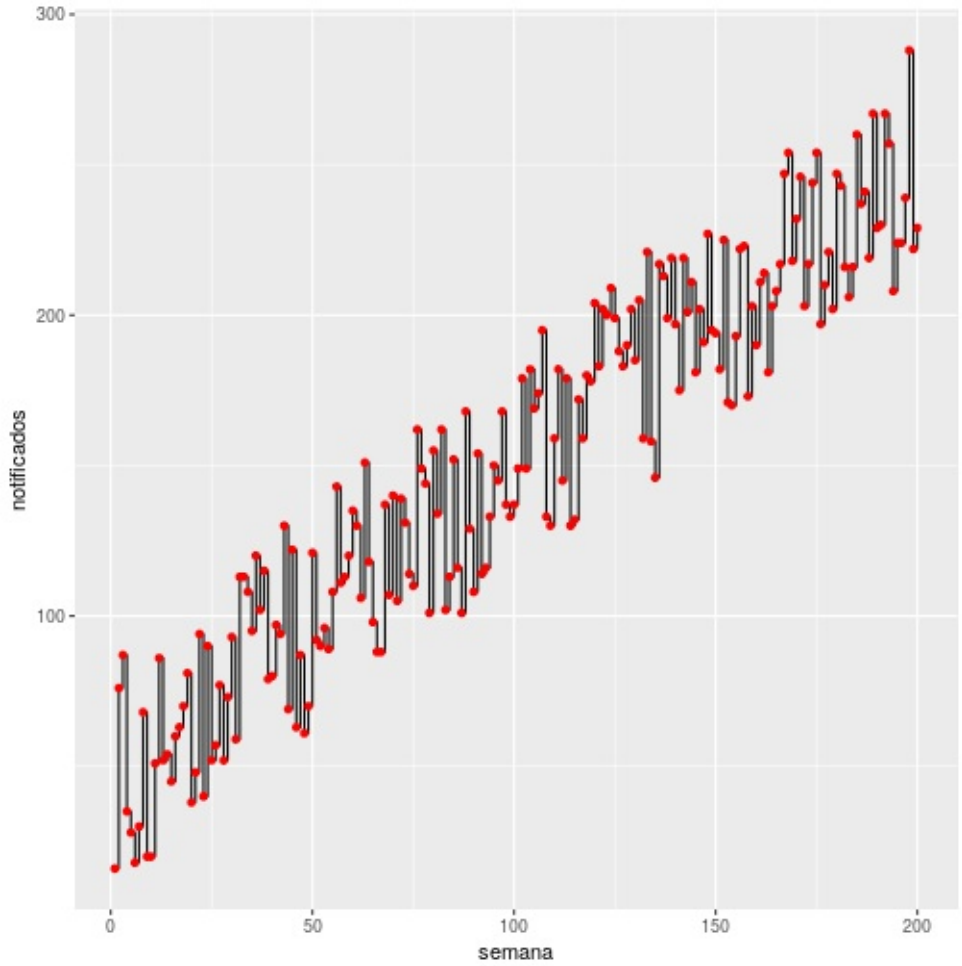


A interpolação linear não é a única forma de estimar o valor de uma variável. Podemos criar gráficos de passos que mantêm constante o valor da variável desde o último ponto de observação até o seguinte

```
> ggplot(casos, aes(semana, notificados)) +
+   geom_step()
```

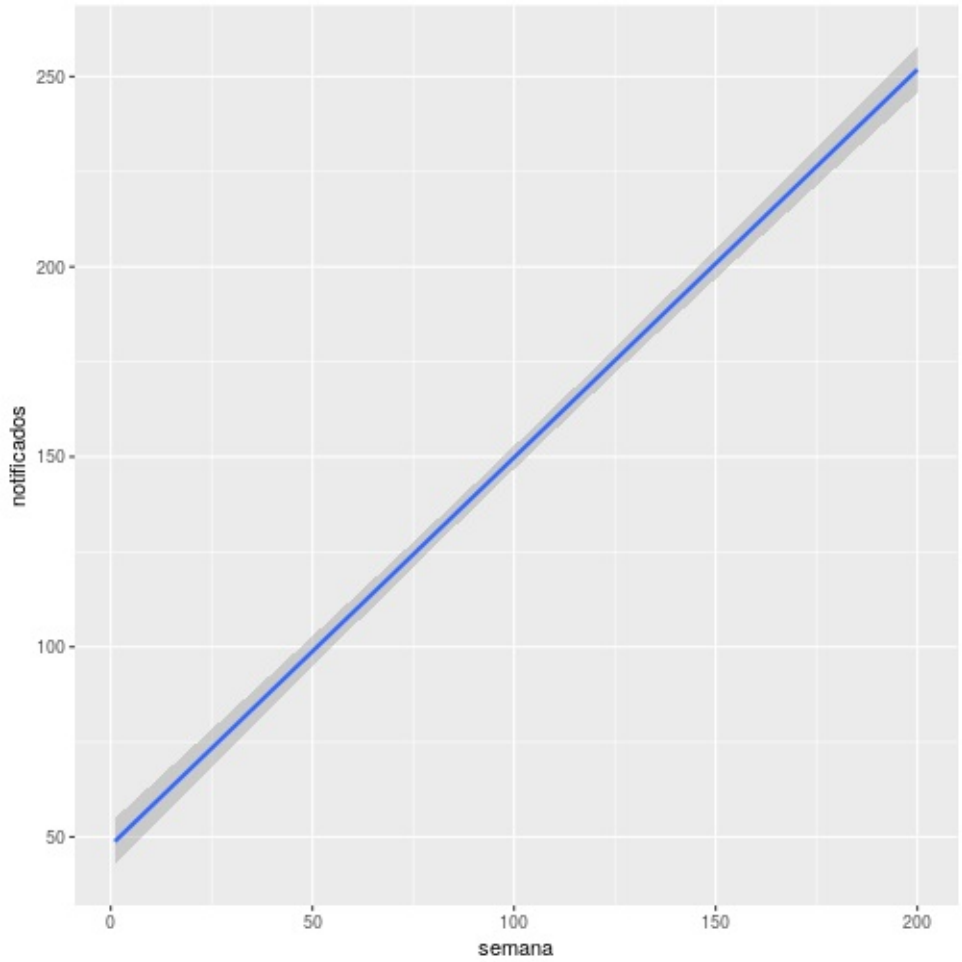



```
> ggplot(casos, aes(semana, notificados)) +
+   geom_step() +
+   geom_point(color = 'red')
```



ou usar modelos estatísticos cuja explicação está além do escopo deste livro, mas a título de exemplo para quem tem familiaridade, são simples de incorporar especificando o método na função `geom_smooth` .

```
> ggplot(casos, aes(semana, notificados)) +
+   geom_smooth(method = 'lm')
```

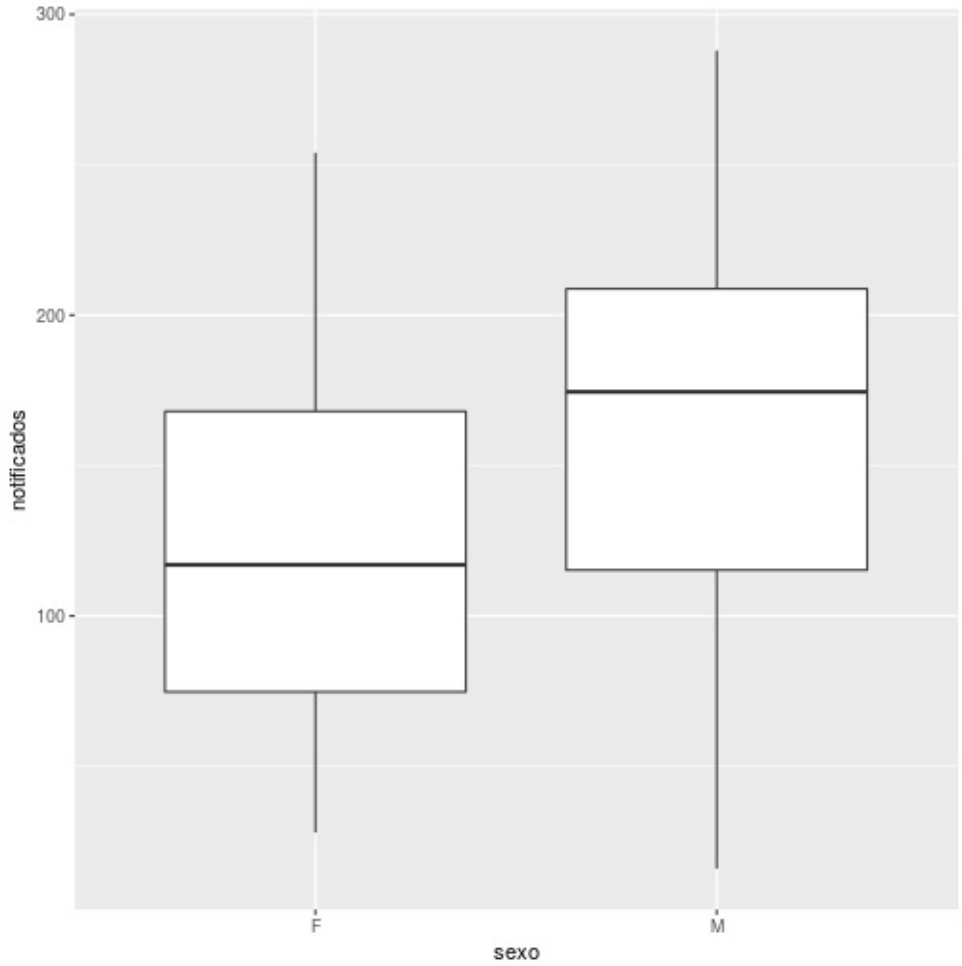


Gráficos de uma variável qualitativa e uma quantitativa

Boxplot

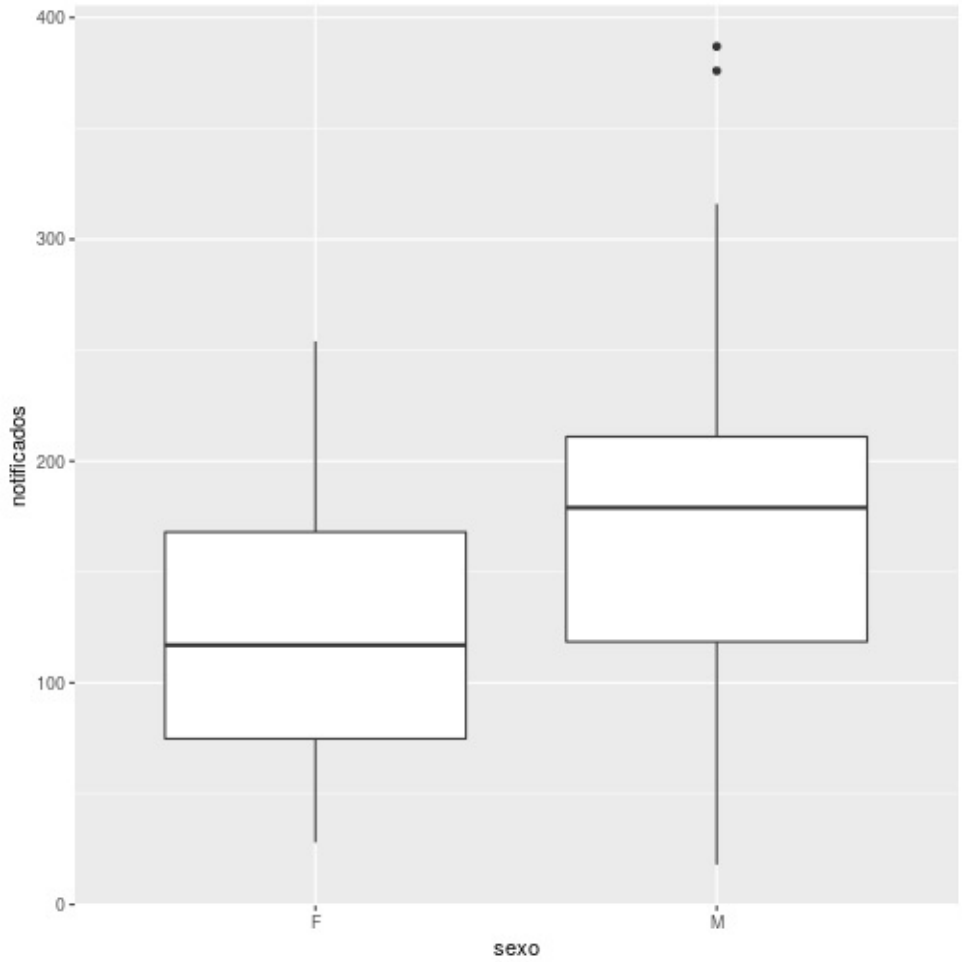
Os boxplot representam a distribuição de uma variável quantitativa em cada uma das categorias da variável qualitativa. Um boxplot está composto por uma caixa que contém o 50% das observações; a base da caixa representa o quantil 25 (abaixo da base está o 25% das observações), uma linha dentro da caixa representa o quantil 50 ou mediana (abaxo da mediana está o 50% das observações) e o topo da caixa representa o quantil 75 (abaixo do topo está o 75% das observações). Portanto, a caixa contém o 50% dos dados (75% - 25%). Da caixa estendem-se "bigodes" que ajudam a identificar valores extremos (pontos além dos bigodes).

```
> ggplot(casos, aes(sexo, notificados)) +
+   geom_boxplot()
```



O gráfico acima sugere a inexistência de valores extremos, pois não há pontos além dos bigodes. Para vermos como seria um boxplot com dados extremos, somemos 300 ao número de casos registrados nas 3 primeiras semanas, criemos os boxplot e depois voltemos aos valores originais.

```
> casos$notificados[1:3] <- casos$notificados[1:3] + 300
> ggplot(casos, aes(sexo, notificados)) +
+   geom_boxplot()
```




```
> casos$notificados[1:3] <- casos$notificados[1:3] - 300
```

A soma de 300 aos 3 primeiros valores criou 2 valores extremos. Por que dois e não 3? Os valores extremos são os que se estendem além dos bigodes e a extensão dos bigodes é determinada por padrão da seguinte maneira:

O bigode superior vai do topo da caixa até o menor valor entre:

- Quartil 75 + amplitude interquartil multiplicada * 1.5.
- Maior valor observado.

O bigode inferior vai da base da caixa até o maior valor entre:

- Quartil 25 - amplitude interquartil multiplicada * 1.5.
- Menor valor observado.

Portanto, a soma de 300 fez com que 2 dos 3 primeiros valores fossem maiores do que *quartil 75 + amplitude interquartil multiplicada* \ 1.5*.

```
> quantile(casos$notificados, c(.75, .25))
```

```
   75%    25%
203.00 101.75
```

```
> 203 + (203 - 101.75) * 1.5
```

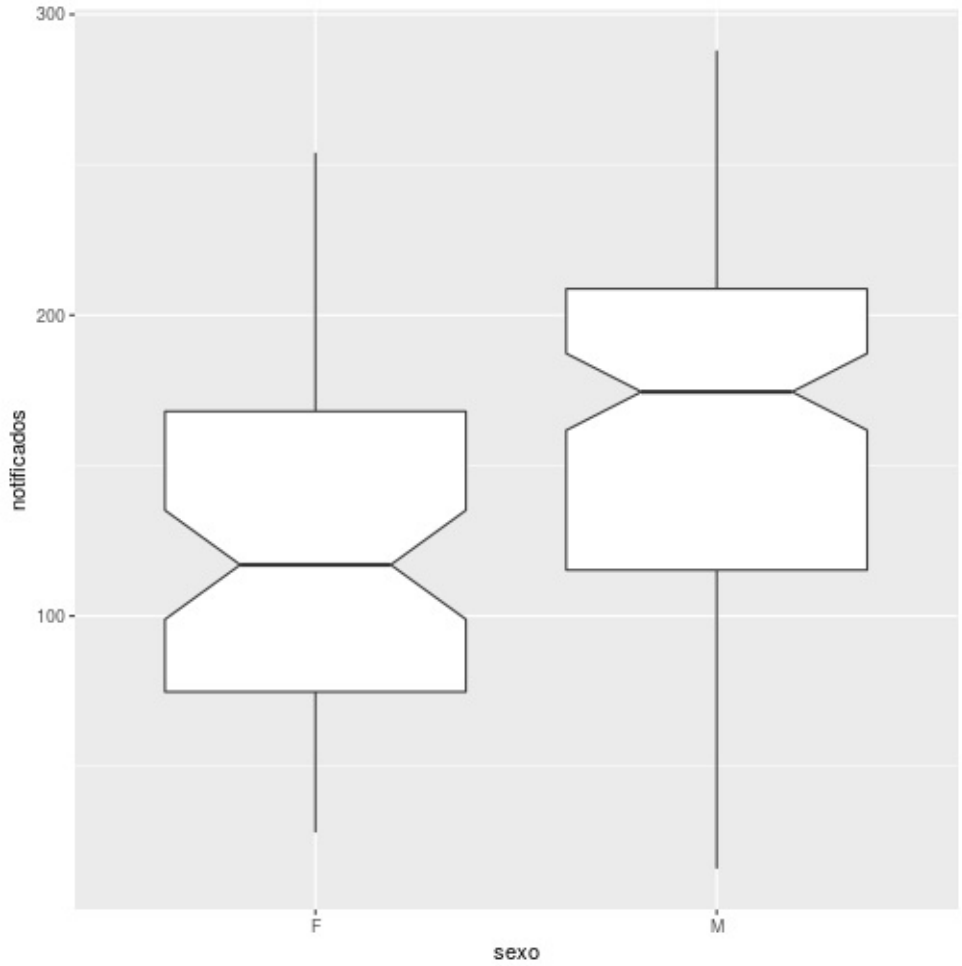
```
[1] 354.875
```

```
> casos$notificados[1:3] + 300
```

```
[1] 316 376 387
```

Para facilitar comparação das medianas de duas categorias diferentes, a propriedade `notch` cria entrâncias centradas nas medianas. A não superposição dessas entrâncias no eixo *y*, como no gráfico a seguir, sugere que as medianas são diferentes.

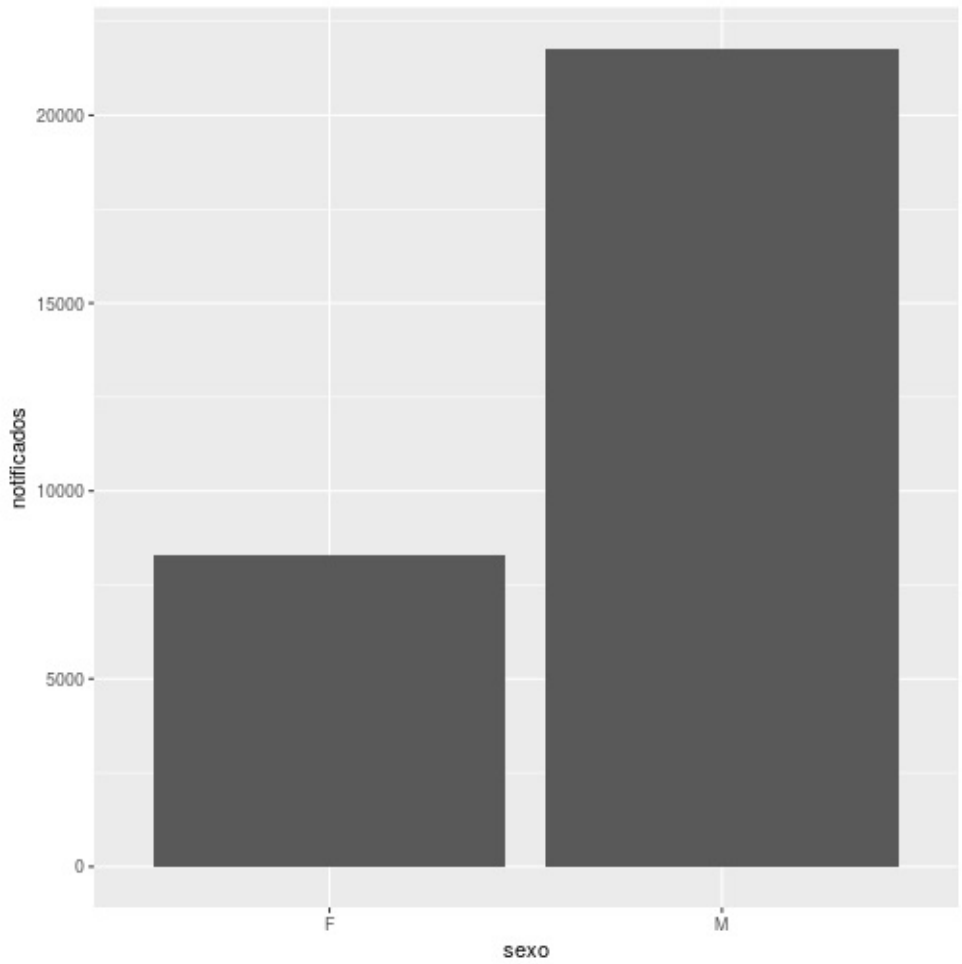
```
> ggplot(casos, aes(sexo, notificados)) +
+   geom_boxplot(notch = T)
```



Barras

Os gráficos de barras também permitem visualizar a relação entre as categorias de uma variável e o valor de uma variável quantitativa, de tal forma que a altura das barras representa o valor da variável quantitativa. A função `geom_bar` nos permite isso mediante a propriedade `stat` que ao ser definida como `identity` define a altura das barras de cada categoria da variável mapeada em `x`, com base na variável mapeada em `y`.

```
> ggplot(casos, aes(sexo, notificados)) +
+   geom_bar(stat = 'identity')
```



Outras gemotrias comuns

- `geom_violin`
- `geom_dotplot`

Gráficos com três variáveis quantitativas

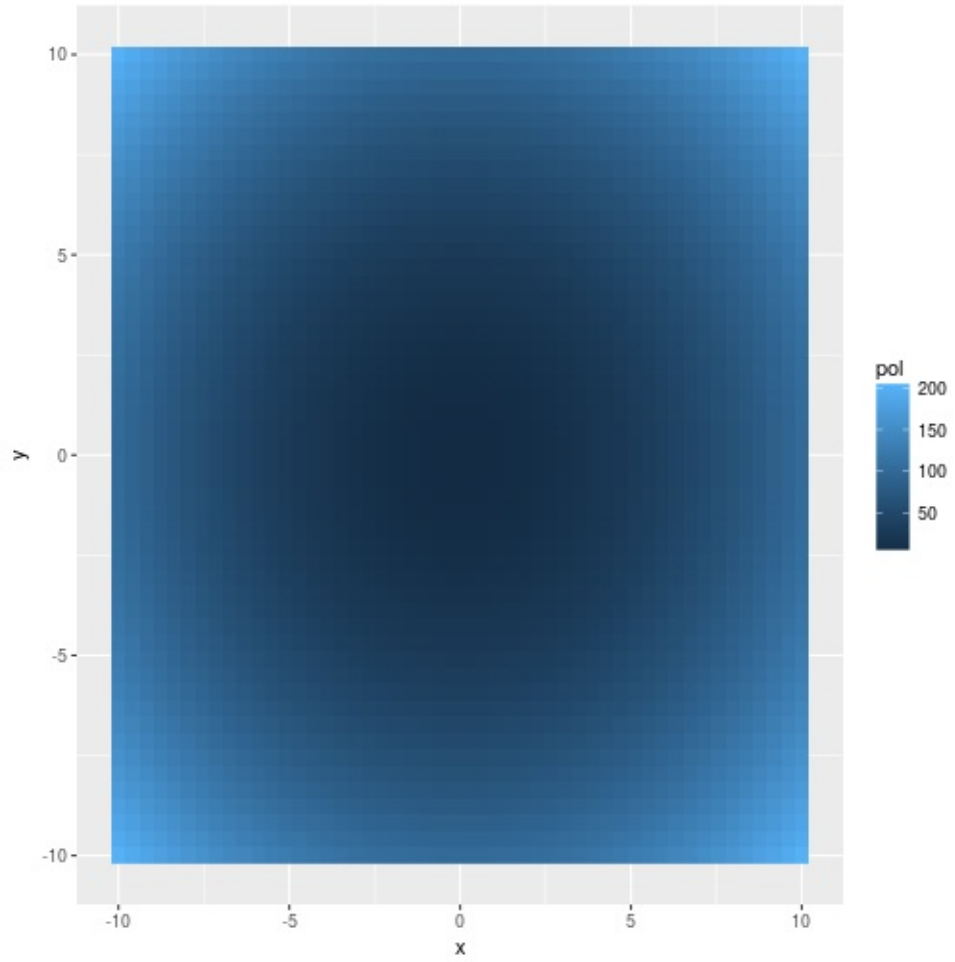
Criemos um banco fictício com coordenadas `x` e `y` variando de -10 a 10, e a cada coordenada atribuamos um valor para representar a concentração de um poluente, de tal forma que quanto mais central a coordenada, menor o valor.

```
> poluente <- expand.grid(x = seq(-10, 10, l = 50), y = seq(-10, 10, l = 50))
> poluente$pol <- 0 + (poluente$x ^ 2 + poluente$y ^ 2)
```

Raster

Nos gráficos de raster ou de retículas, cada coordenada é o centro de uma célula ou pixel e o valor ou atributo de cada coordenada é mapeado em uma escala de cor.

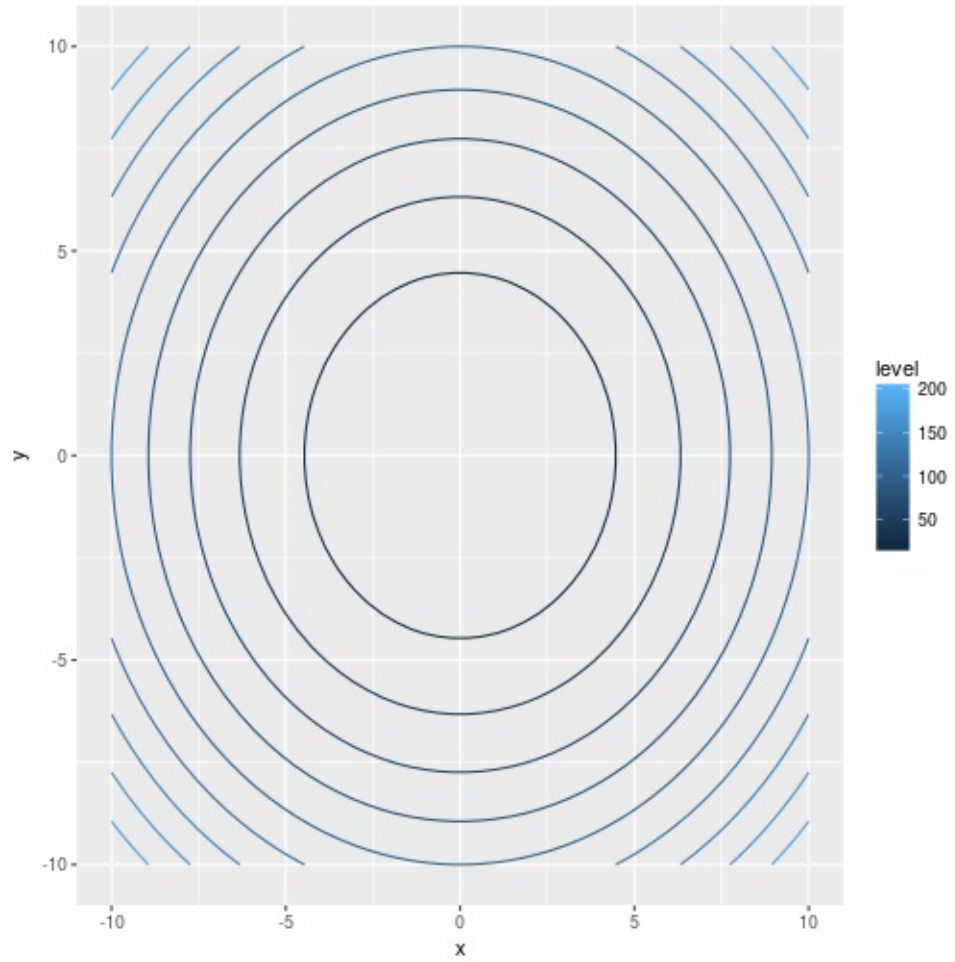
```
> ggplot(poluente, aes(x, y, fill = pol)) +
+   geom_raster()
```



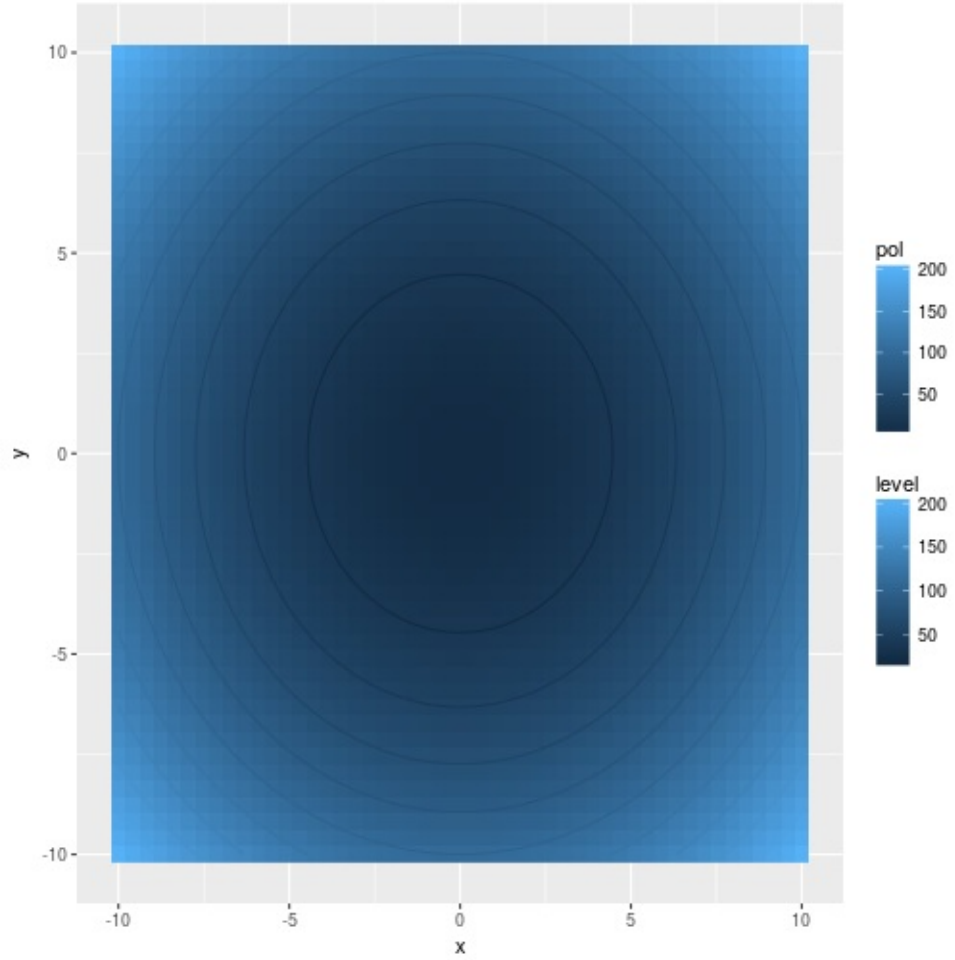
Linhas de contorno

Nos gráficos com linhas de contorno ou isolinhas, cada linha representa um valor da variável mapeada (todas as coordenadas têm o mesmo valor se estão na mesma linha).

```
> ggplot(poluente, aes(x, y)) +
+   geom_contour(aes(z = pol, color = ..level..))
```



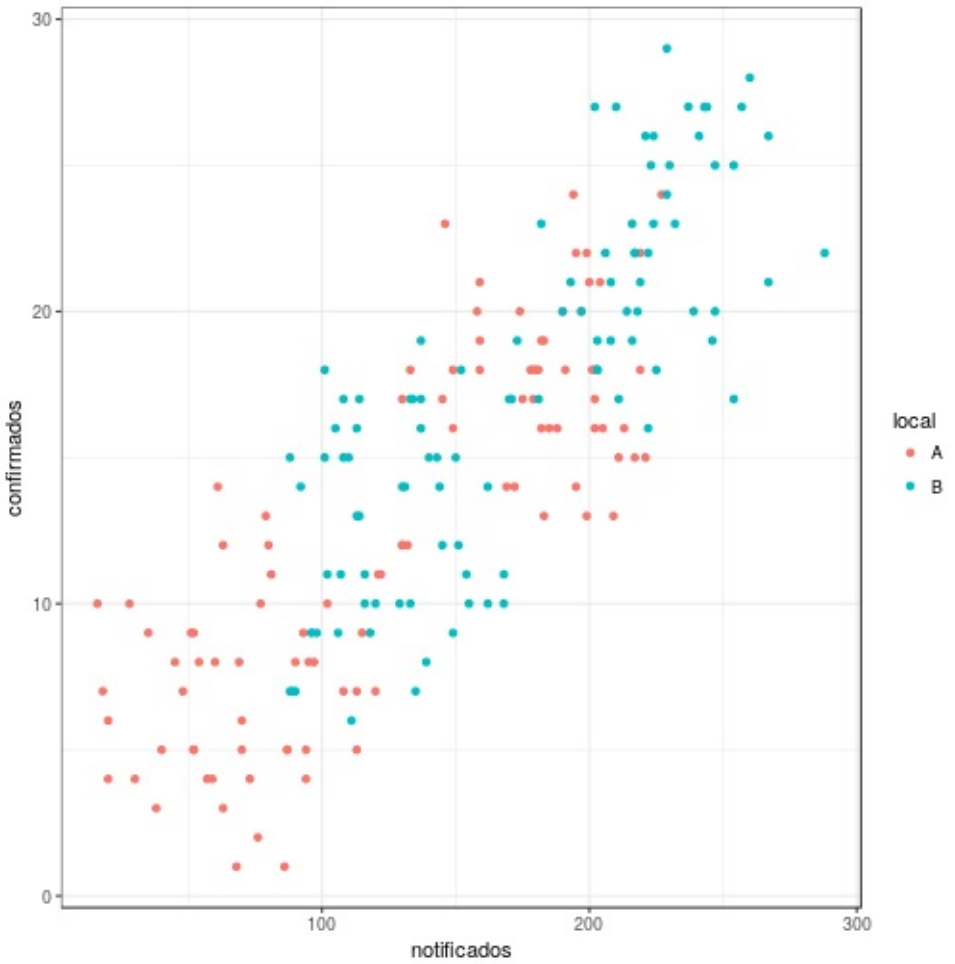

```
> ggplot(poluente, aes(x, y, fill = pol)) +
+   geom_raster() +
+   geom_contour(aes(z = pol, color = ..level..))
```



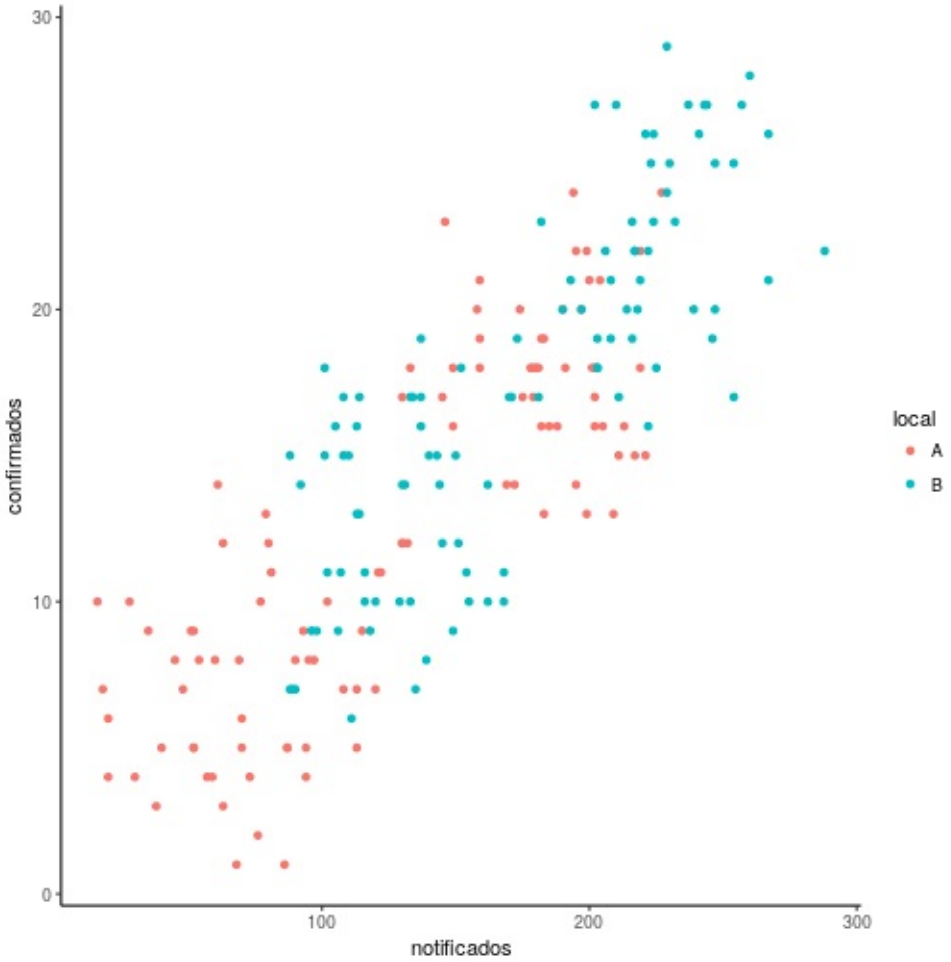
Temas

Além das propriedades estéticas das geometrias, existem outros elementos modificáveis e não dependentes dos dados. O estilo ou *tema* dos gráficos é dado por alguns desses elementos e há temas predefinidos. Ao digitarmos `theme_` e dar um `TAB`, aparecerão os temas disponíveis, entre os quais estão os seguintes:

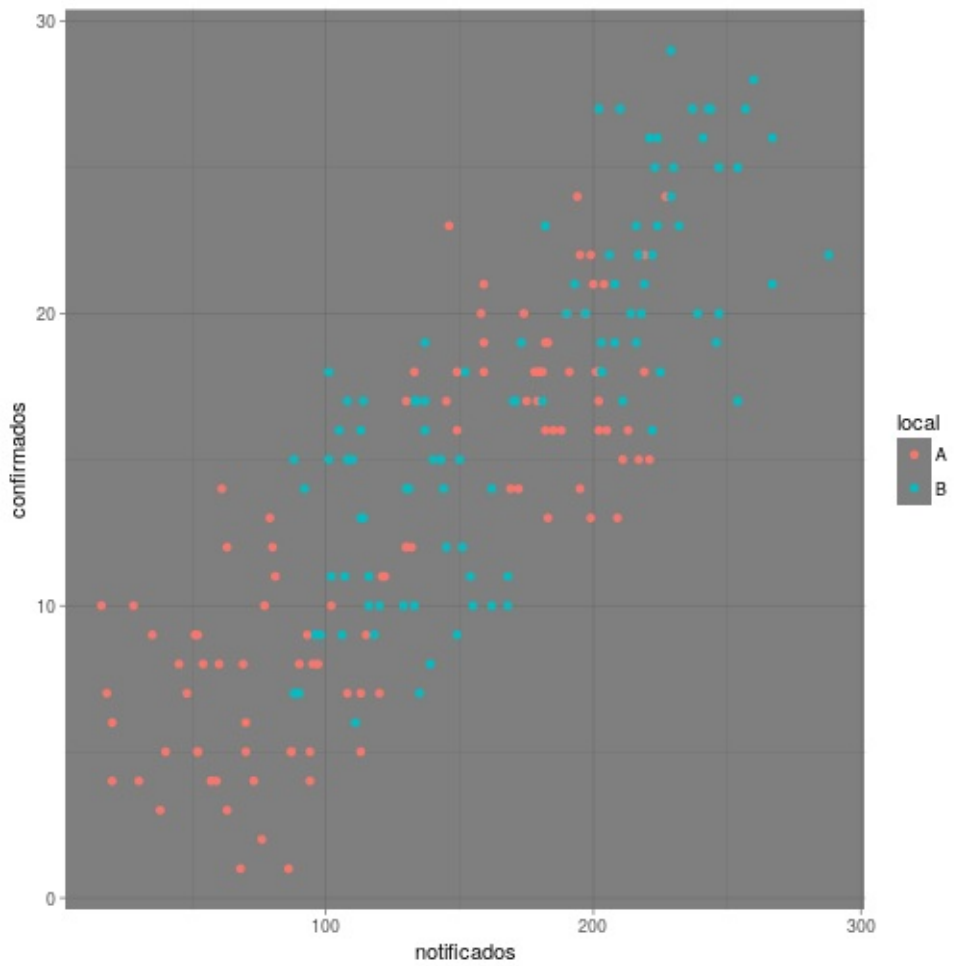
```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   theme_bw()
```




```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   theme_classic()
```




```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   theme_dark()
```



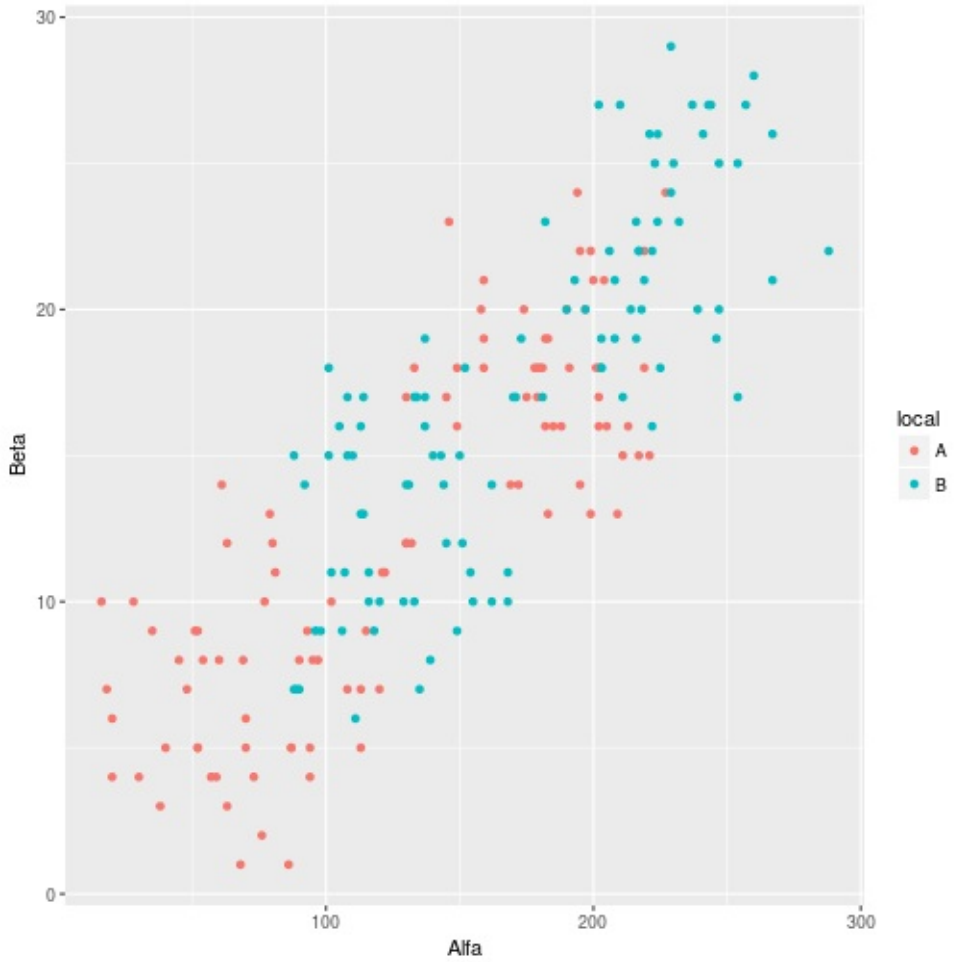
Para maior controle dos elementos que compõem os eixos e as legendas, devemos usar a função `theme` em conjunto com *funções de elementos* que modificam elementos específicos.

Eixos

Títulos

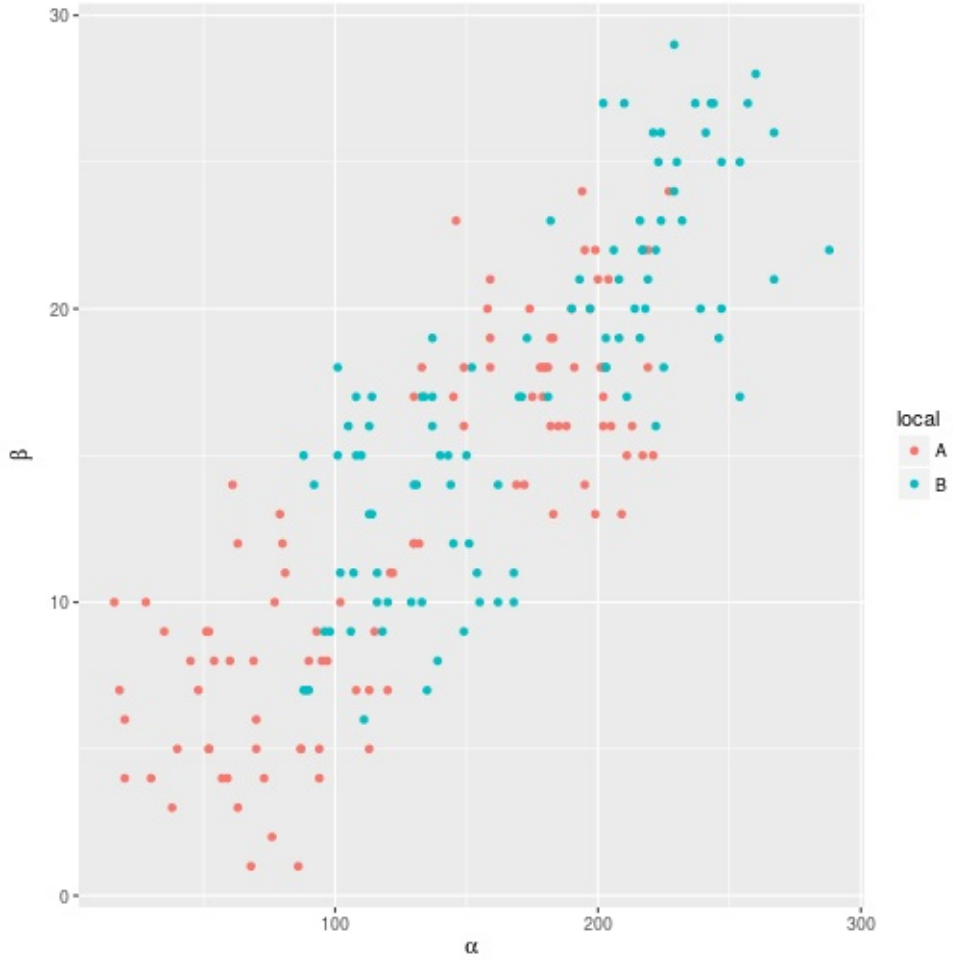
O título do eixos é dado pelo nome das variáveis mapeadas nas propriedades estéticas e é modificado pelas funções `xlab` e `ylab`.

```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   xlab('Alfa') +
+   ylab('Beta')
```



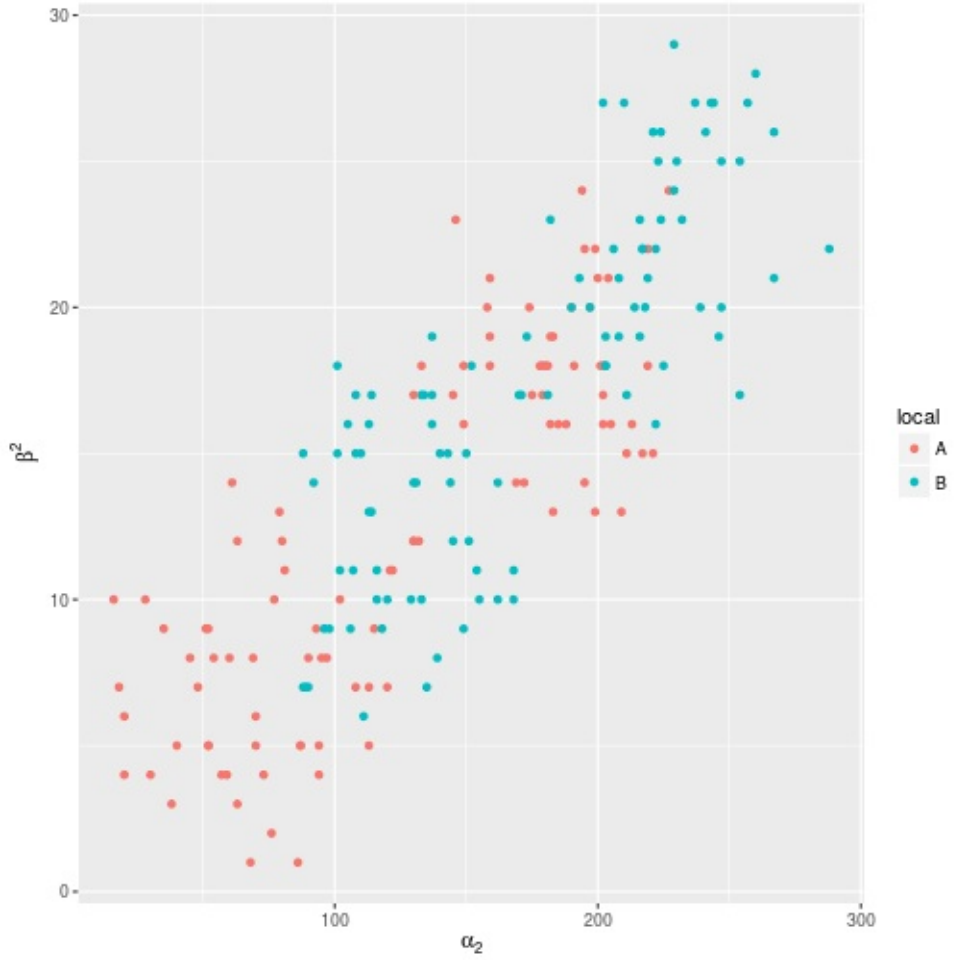
Caracteres especiais são interpretados pela função `expression`,

```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   xlab(expression(alpha)) +
+   ylab(expression(beta))
```



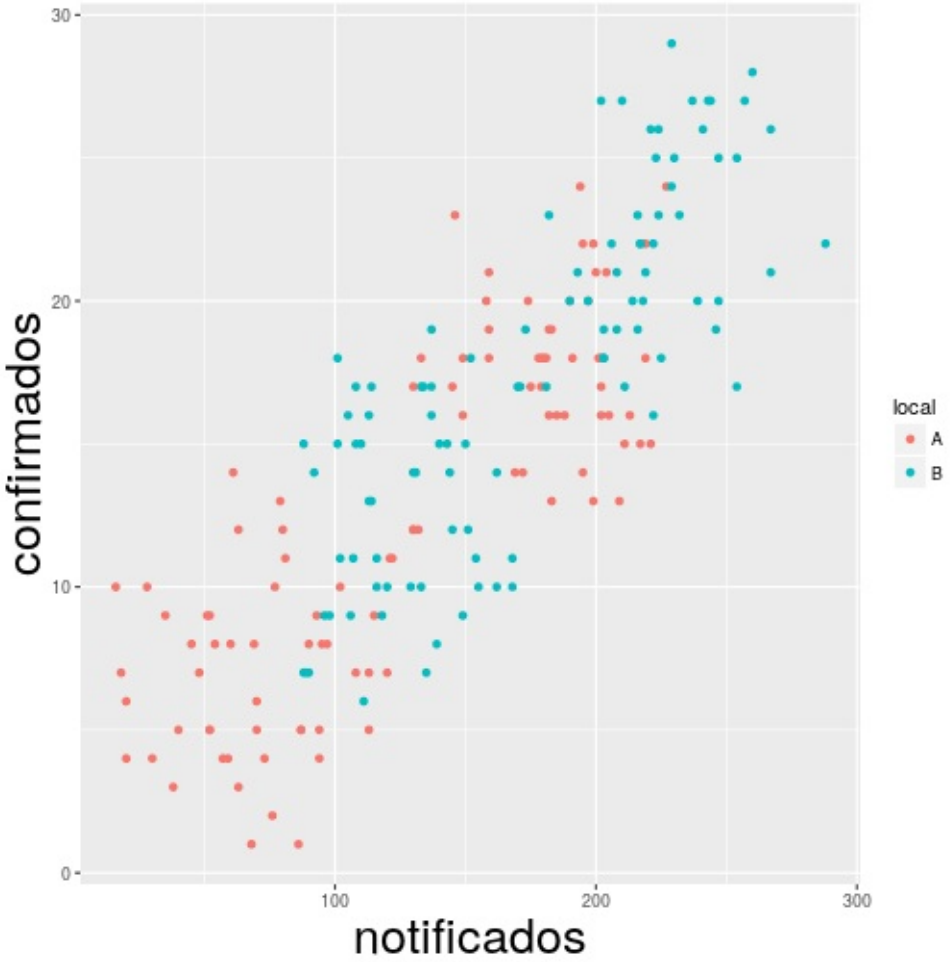
enquanto sub e superescritos são incluídos pela função `bquote`.

```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   xlab(bquote(alpha[2])) +
+   ylab(bquote(beta^2))
```



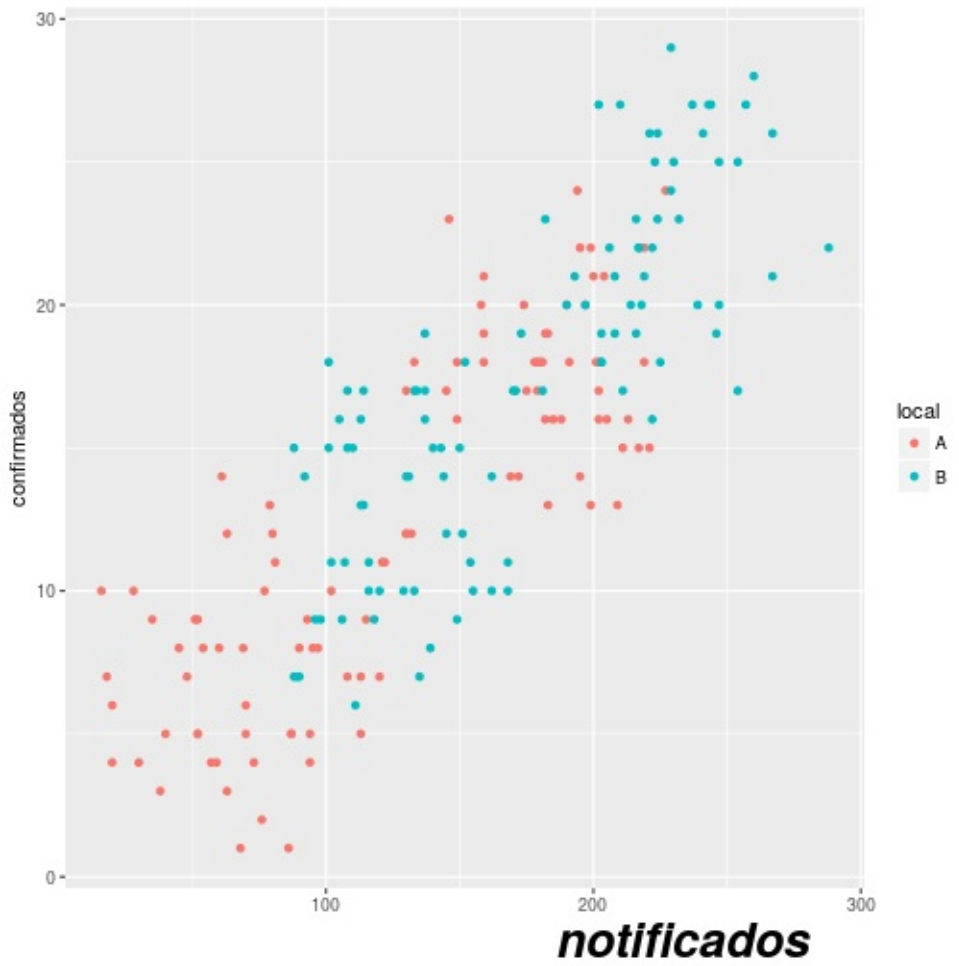
O tamanho e a orientação são características do texto do elemento `axis.title`, modificadas pela função `element.text` para esse elemento. `axis.title` e `element.text` devem ser usados dentro da função `theme`.

```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   theme(axis.title = element_text(size = 25, angle = 0))
```

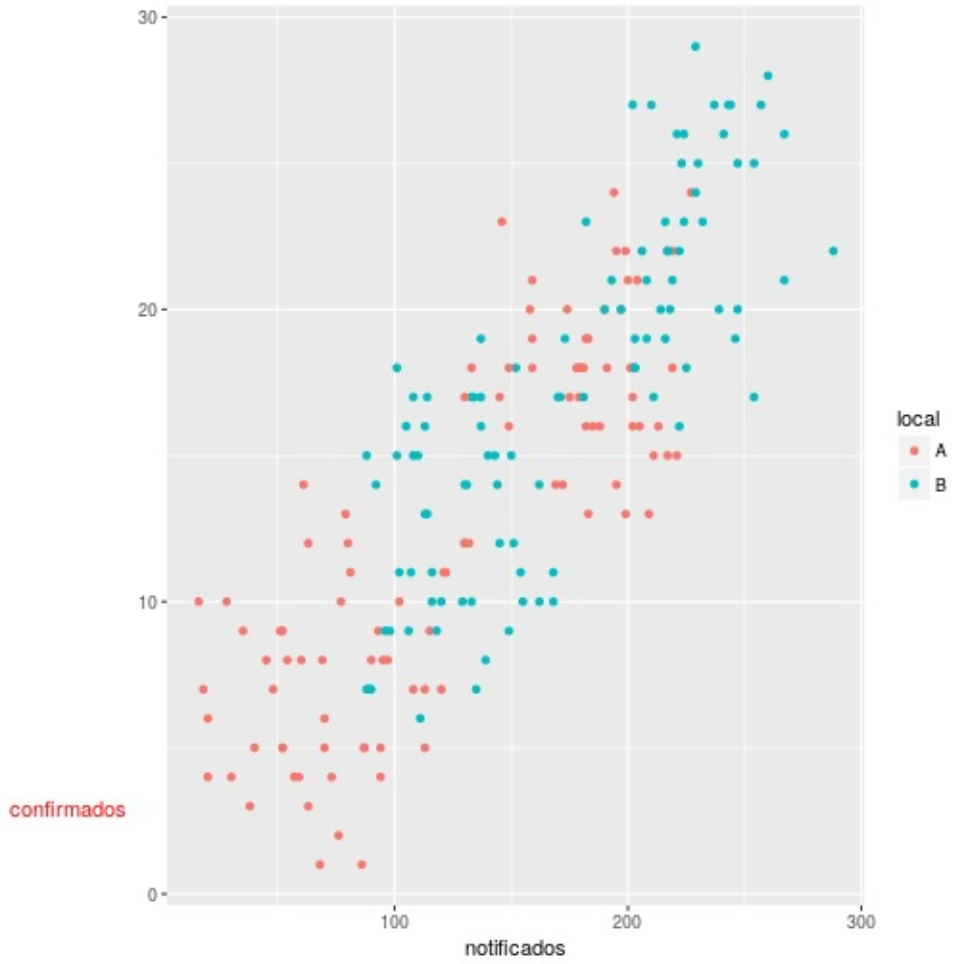


É possível mudar só um dos eixos e outras características como a fonte (`face`), a justificação horizontal (`hjust`) e vertical (`vjust`), e a cor (`color`).

```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   theme(axis.title.x = element_text(size = 25, face = 'bold.italic',
+                                     hjust = .9))
```



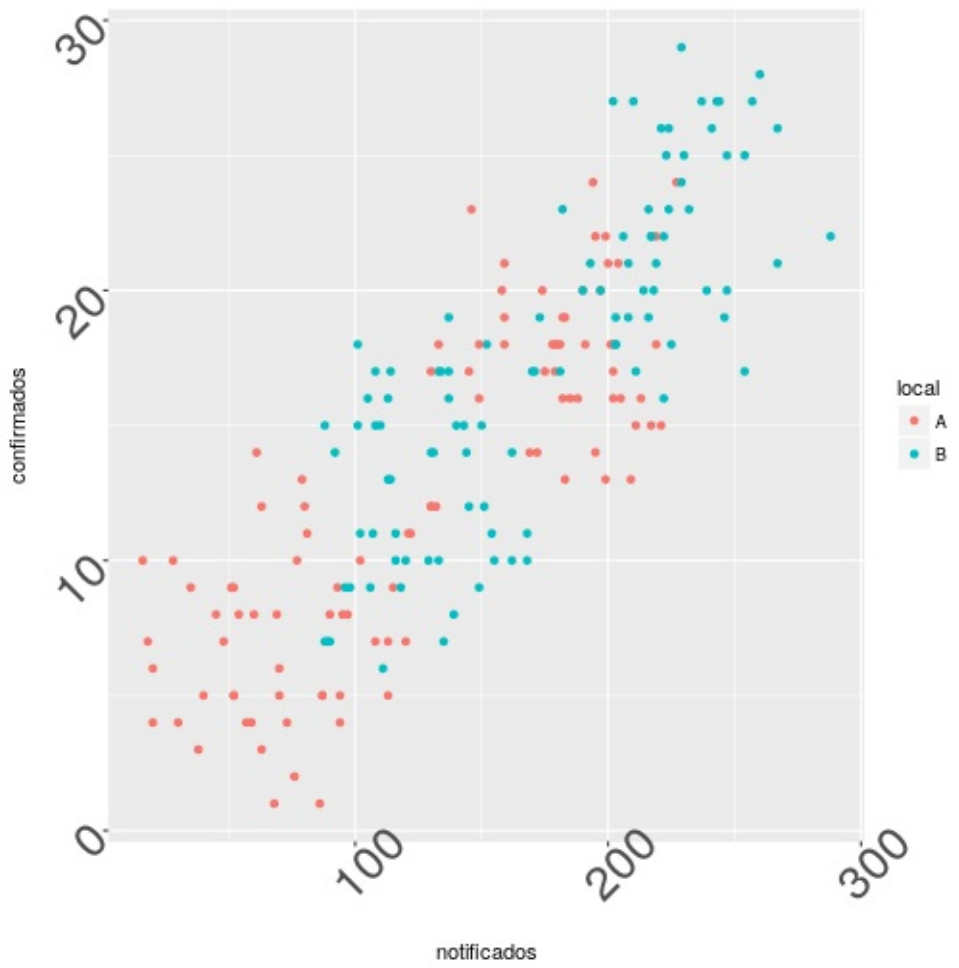

```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   theme(axis.title.y = element_text(angle = 0, vjust = .1, color = 'red'))
```



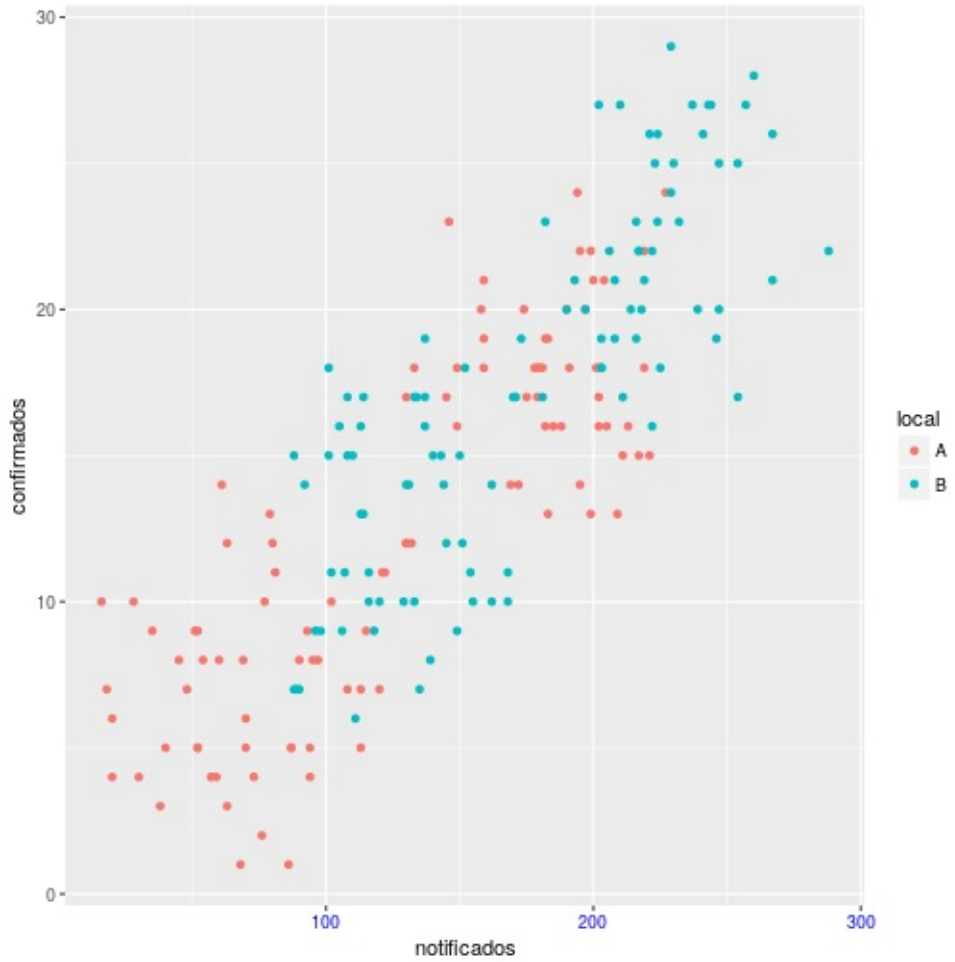
Marcas

As características das marcas são definidas pelo elemento `axis.text` e modificadas pela função `element_text`.

```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   theme(axis.text = element_text(size = 25, angle = 45))
```



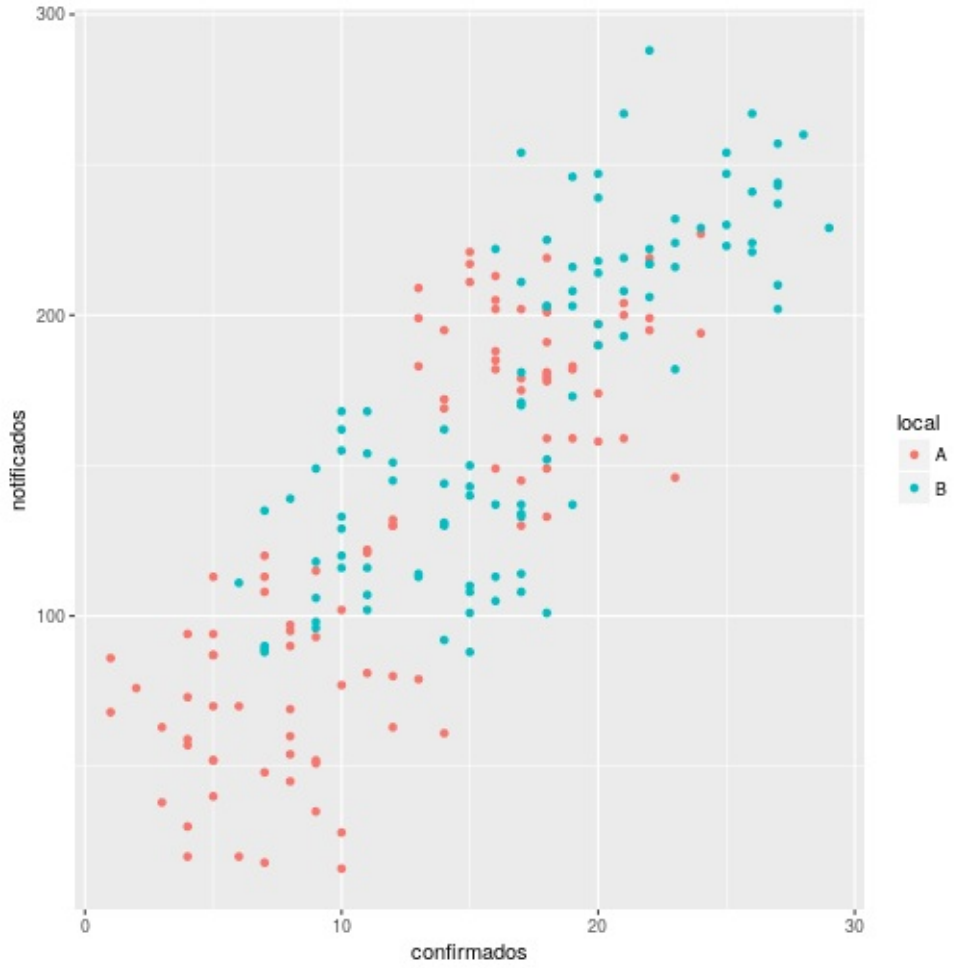

```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   theme(axis.text.x = element_text(color = 'blue'))
```



Transposição

Ao acrescentarmos a função `coord_flip` , os eixos são transpostos.

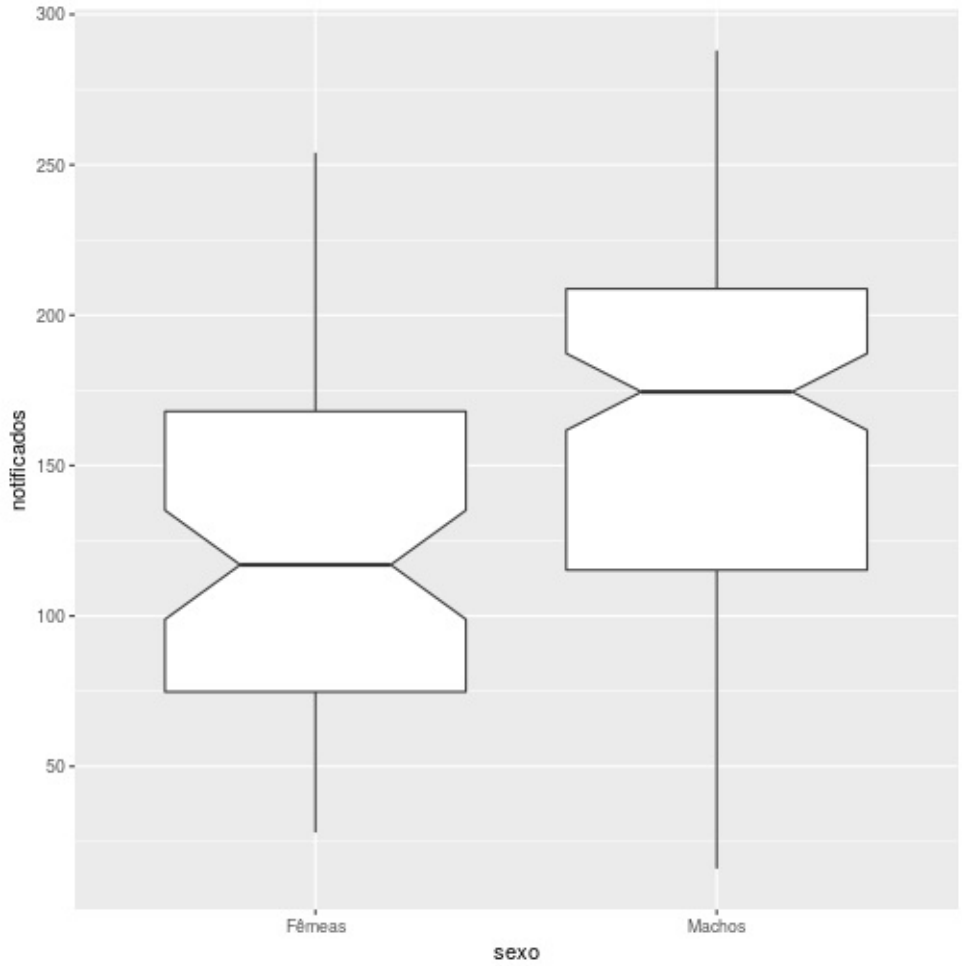
```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   coord_flip()
```



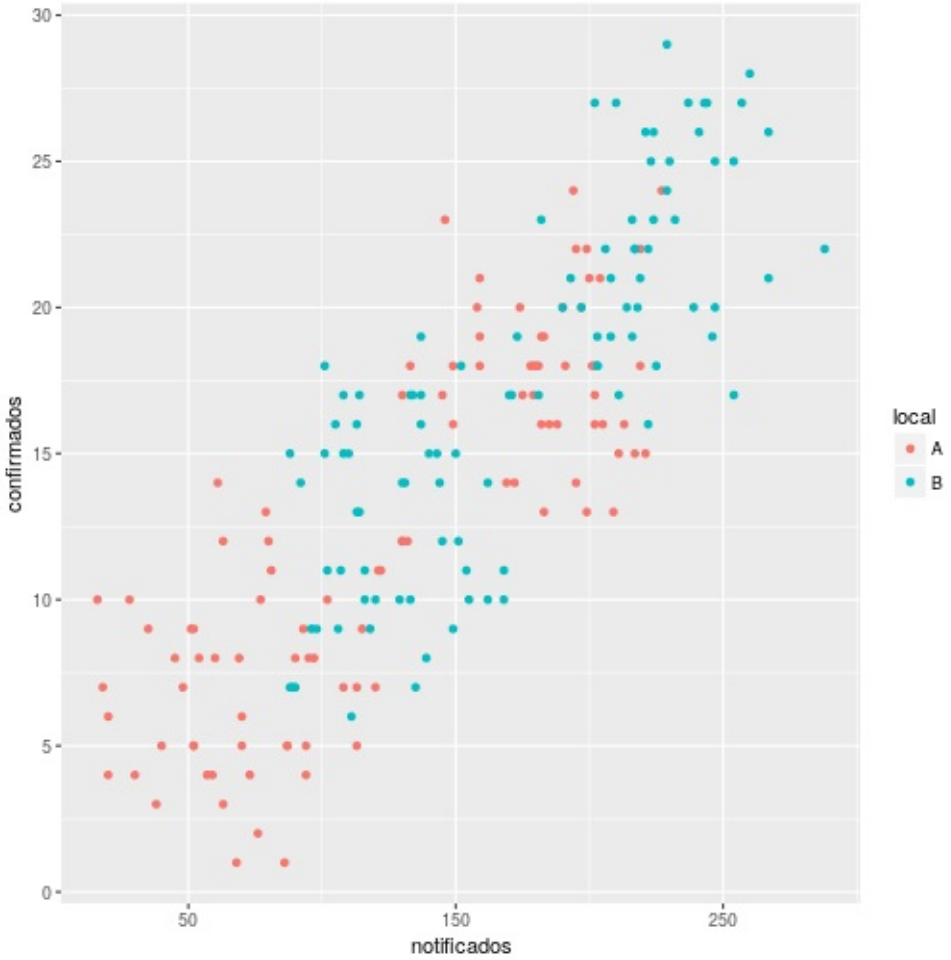
Posição e etiqueta das marcas

Para alterar a posição e a etiqueta devemos usar a função `scale_x_continuous` com variáveis quantitativas e `scale_x_discrete` com variáveis qualitativas. Em qualquer caso, o argumento `breaks` define a posição e `levels` as etiquetas.

```
> ggplot(casos, aes(sexo, notificados)) +
+   geom_boxplot(notch = T) +
+   scale_x_discrete(breaks = c('F', 'M'), labels = c('Fêmeas', 'Machos')) +
+   scale_y_continuous(breaks = seq(0, 300, by = 50))
```




```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   scale_x_continuous(breaks = c(50, 150, 250)) +
+   scale_y_continuous(breaks = seq(0, 30, by = 5))
```



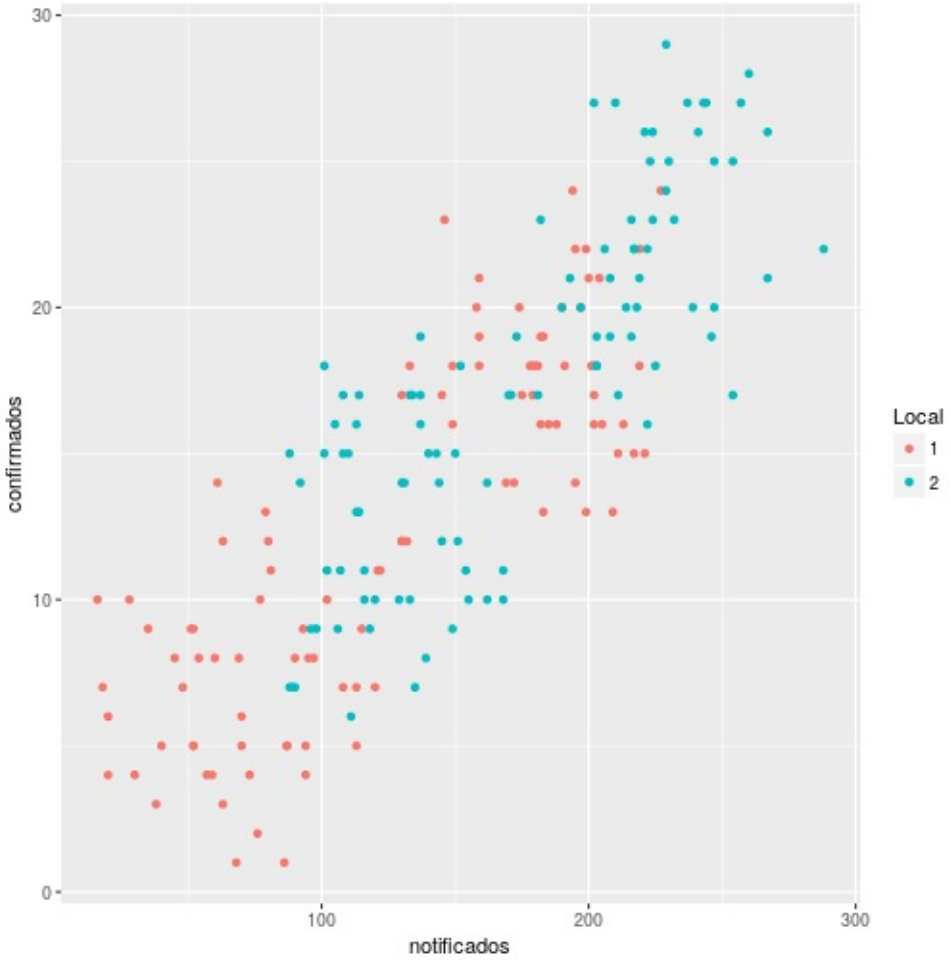
No caso de variáveis qualitativas, também podemos mudar as etiquetas mudando o nome das colunas do banco.

Legendas

Título e marcas

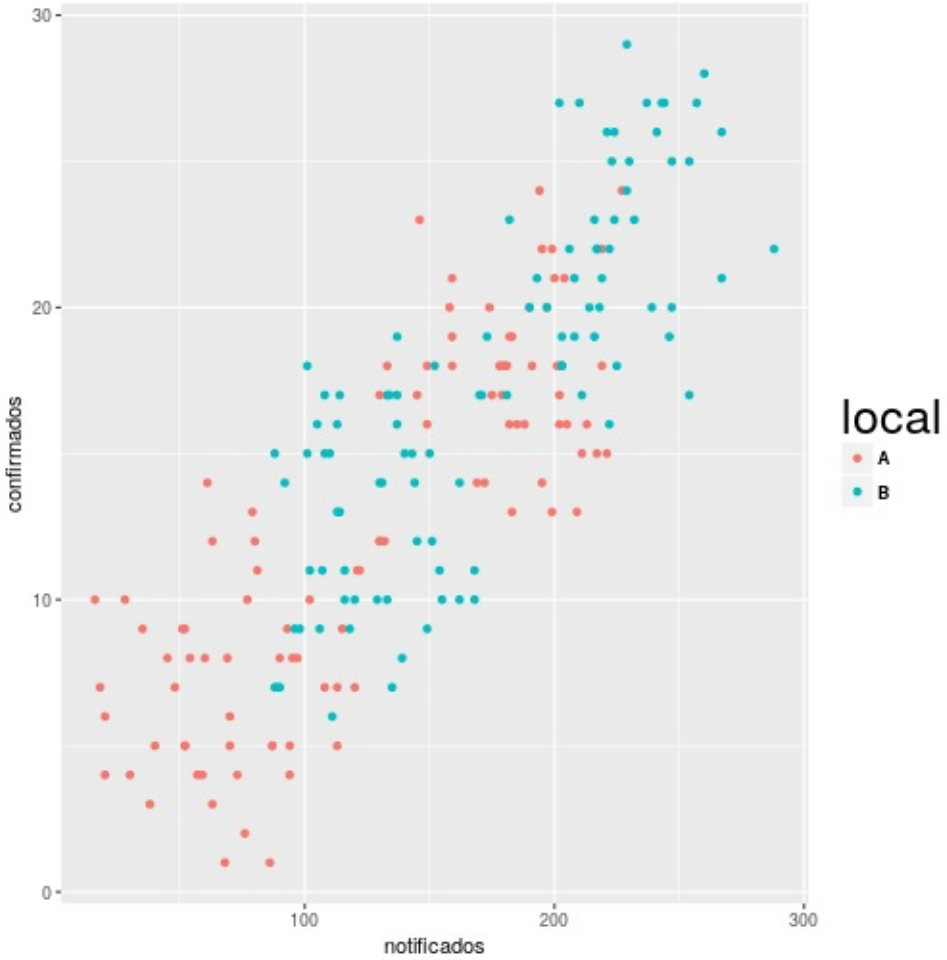
Tanto o título como a posição e a etiqueta das marcas são definidos pelas funções da família `scale_` (por exemplo, `scale_fill_discrete`).

```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   scale_color_discrete(name = 'Local', breaks = c('A', 'B'), labels = 1:2)
```



As características do texto do título e das marcas são modificadas seguindo a lógica usada com os eixos.

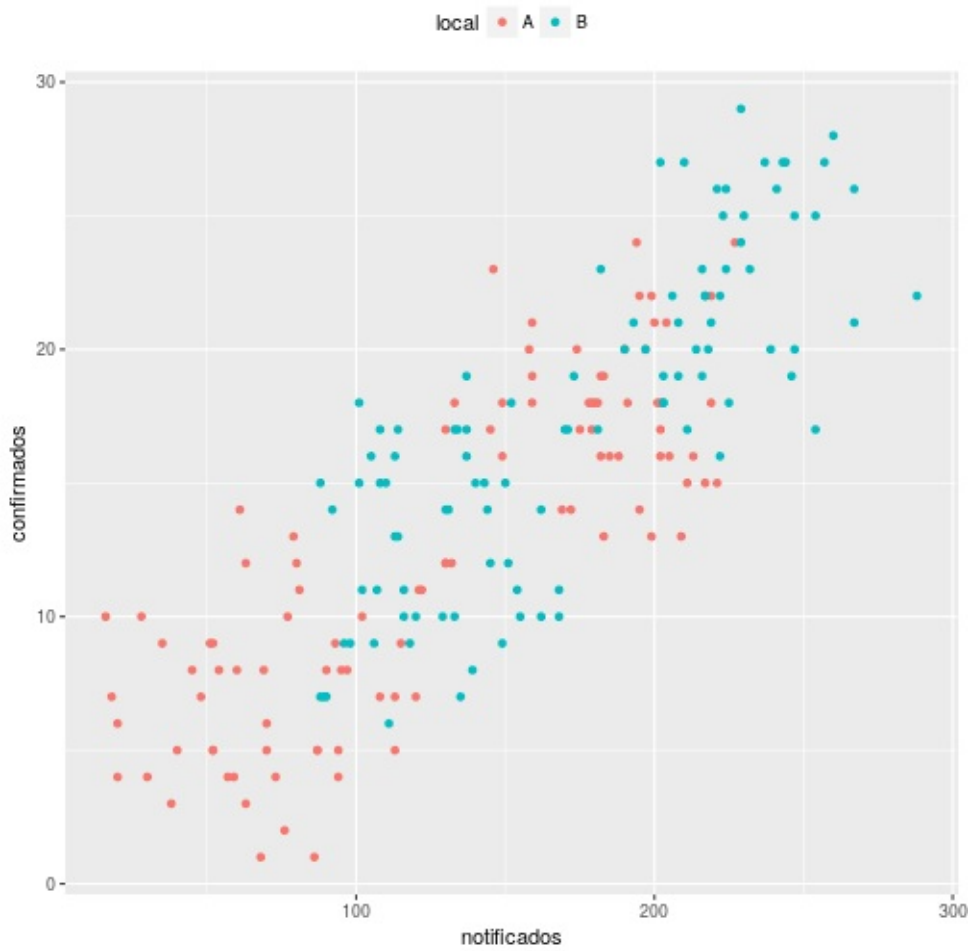
```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   theme(legend.title = element_text(size = 25),
+         legend.text = element_text(face = 'bold'))
```



Posição

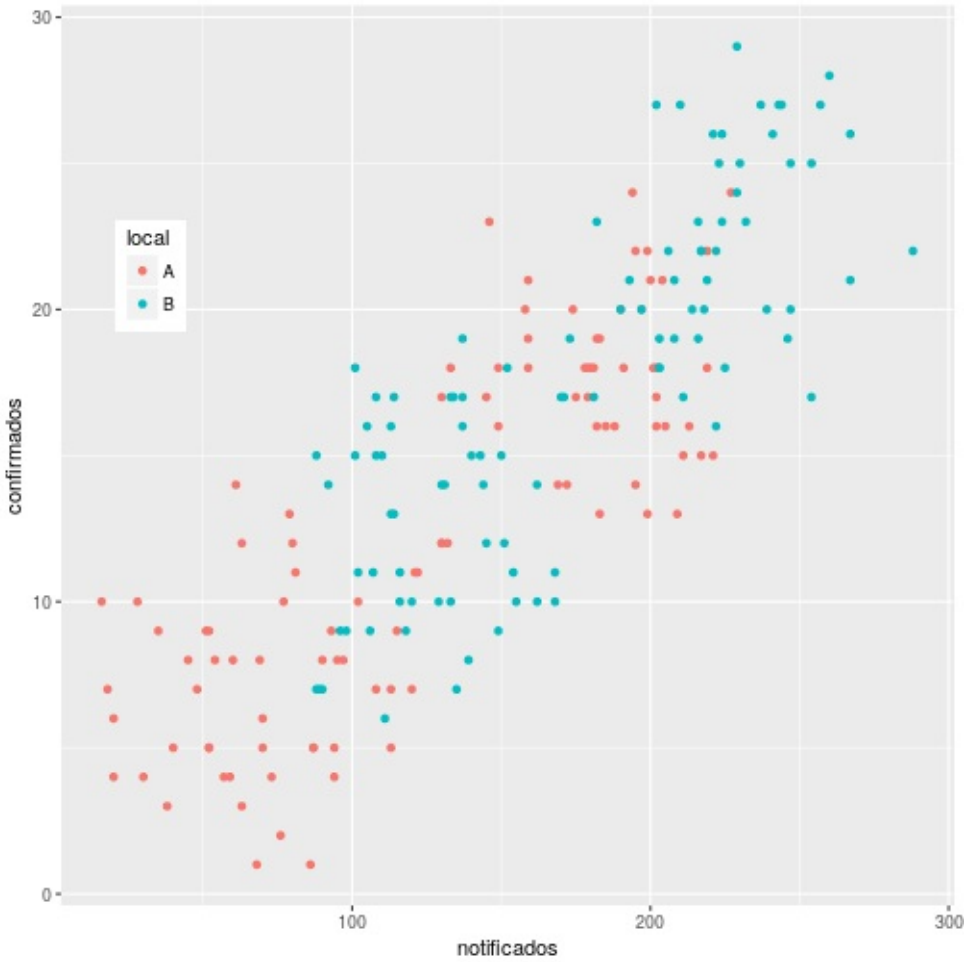
Podemos posicionar a legenda acima (`top`), abaixo (`bottom`) ou aos lados (`left` , `right`).

```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   theme(legend.position = 'top')
```



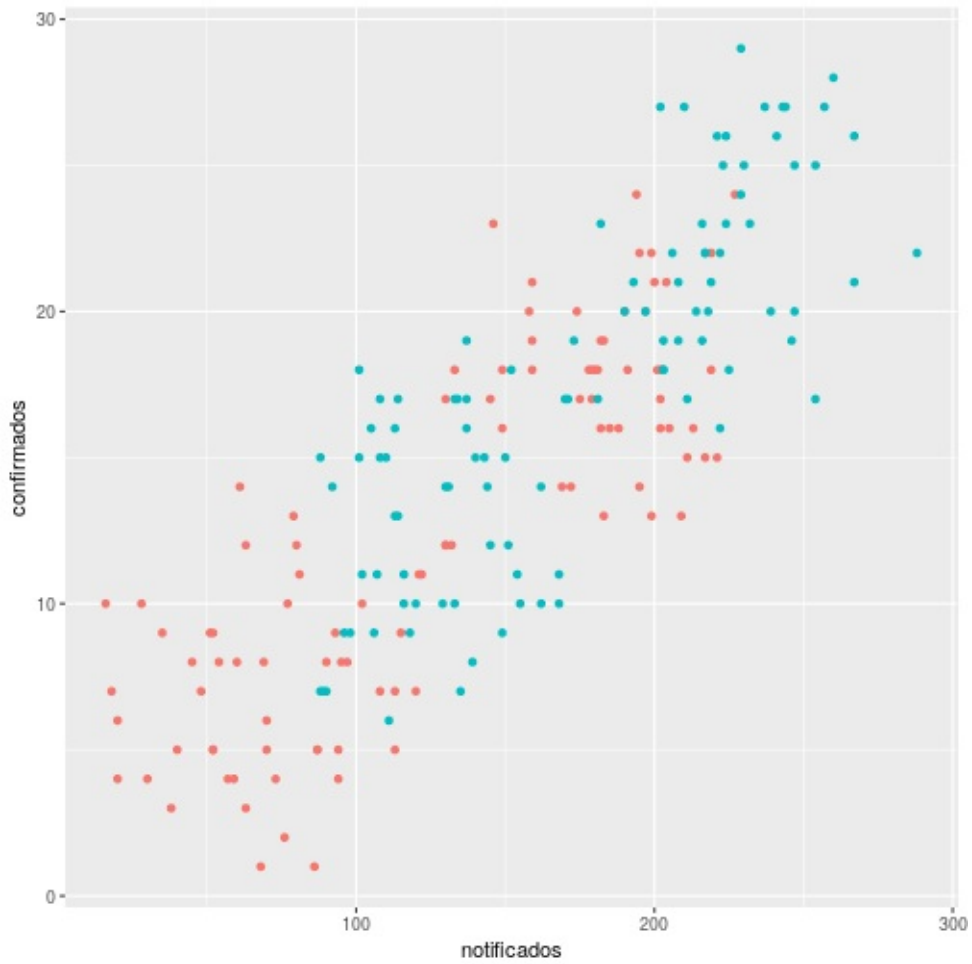
Para posicioná-la dentro dentro dentro da área graficada, devemos especificar um par de coordenadas escaladas (cada eixo vai de 0 até 1).

```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   theme(legend.position = c(.1, .7))
```



Também podemos tirar a legenda.

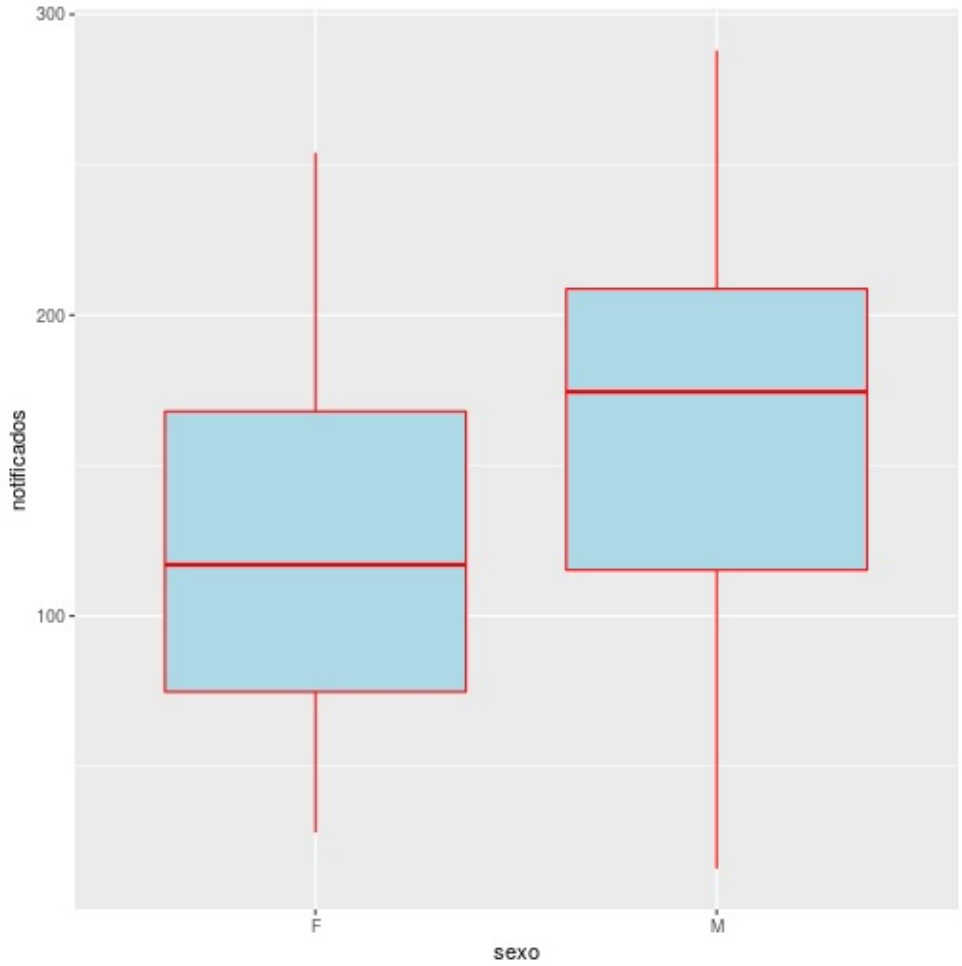
```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point() +
+   theme(legend.position = 'none')
```



Cores

A definição da cor é dada em termos de preenchimento (`fill`) e de contorno (`colour` ou `color`).

```
> ggplot(casos, aes(sexo, notificados)) +
+   geom_boxplot(fill = 'lightblue', color = 'red')
```



São várias as cores disponíveis e além do nome é possível usar os formatos RGB e hexadecimal. Quanto às cores disponíveis, há 657 opções cujos nomes são listados pela função `colors`.

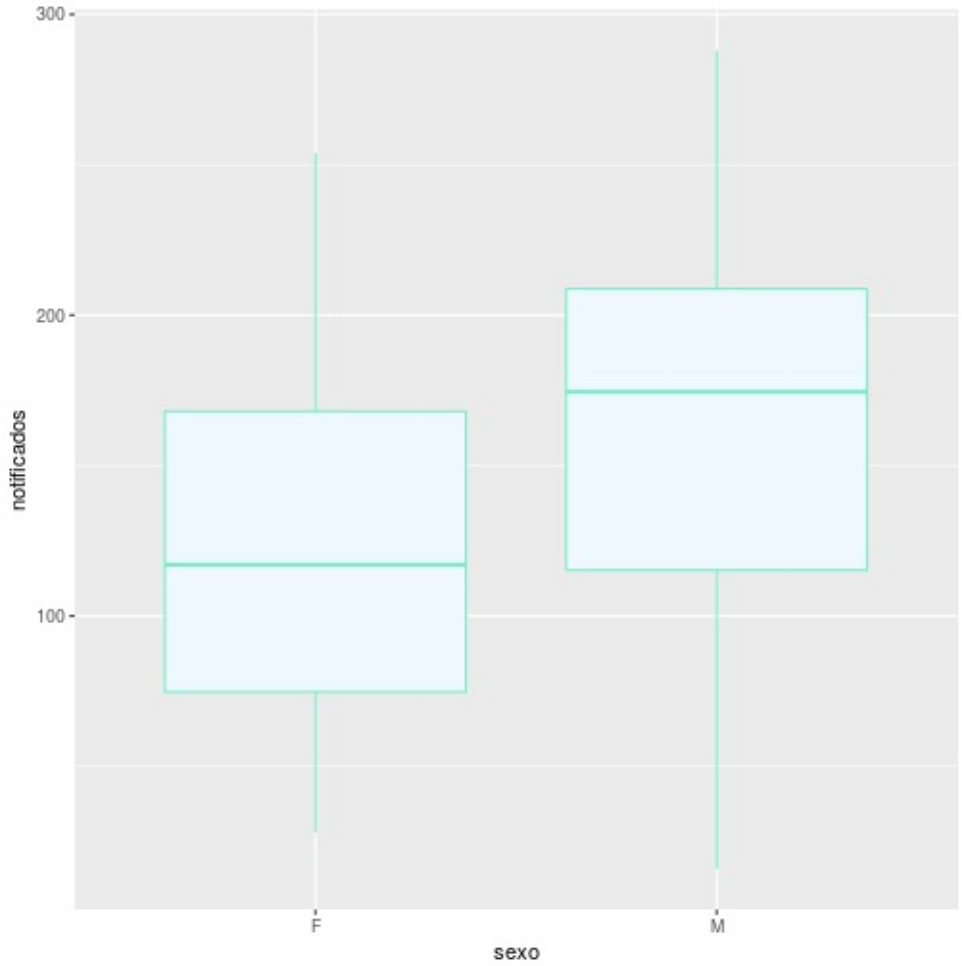
```
> str(colors())
```

```
 chr [1:657] "white" "aliceblue" "antiquewhite" ...
```

```
> colors()[1:10]
```

```
 [1] "white"         "aliceblue"     "antiquewhite"
 [4] "antiquewhite1" "antiquewhite2" "antiquewhite3"
 [7] "antiquewhite4" "aquamarine"    "aquamarine1"
[10] "aquamarine2"
```

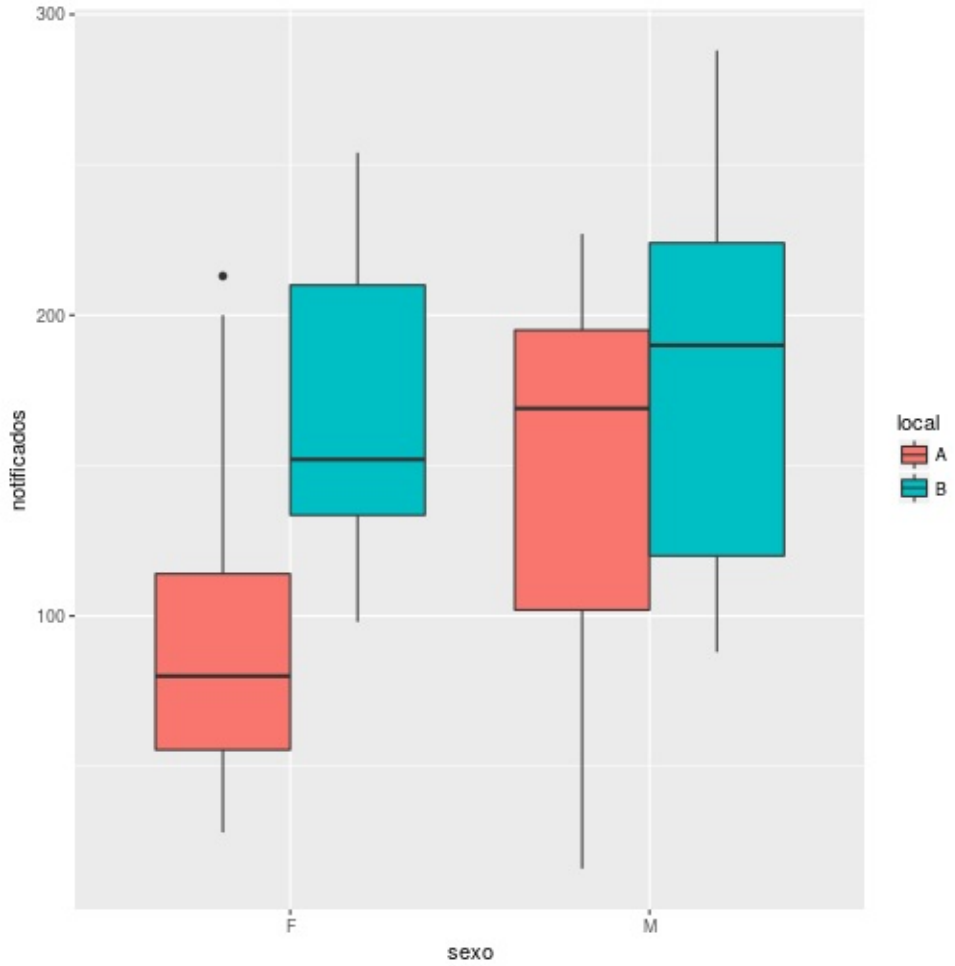
```
> ggplot(casos, aes(sexo, notificados)) +
+   geom_boxplot(fill = colors()[2], color = colors()[10])
```



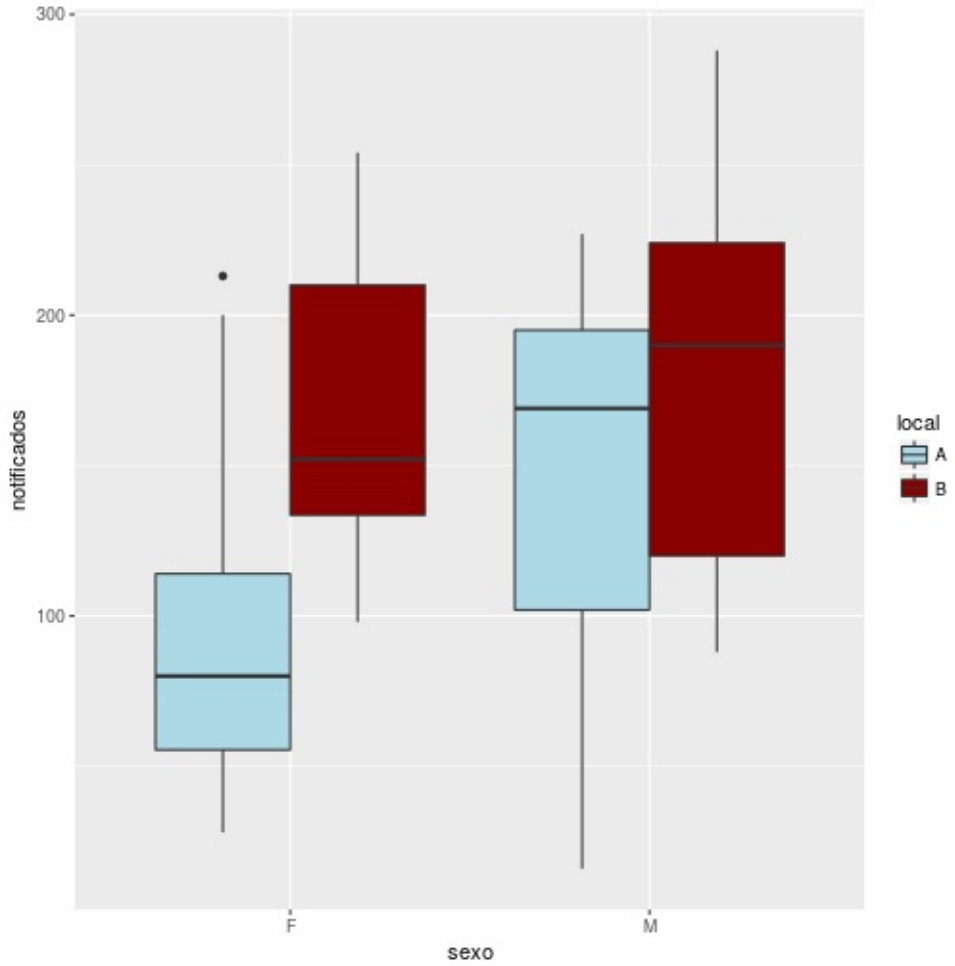
As vezes é difícil escolher a cor usando a lista da função `colors`, mas se digitarmos *r color chart* na seção de imagens do google, aparecerão exemplos de cores com seus respectivos nomes, e formatos RGB e hexadecimal.

Nos exemplos anteriores as cores foram definidas, mas nenhuma variável foi mapeada nas cores (as categorias fêmea e macho ficaram com a mesma cor). Se o objetivo for mapear uma variável em cores mudando o padrão, devemos usar as funções da família `scale_`. Com a função `scale_fill_manual` (`scale_color_manual`) podemos definir a cor para cada categoria.

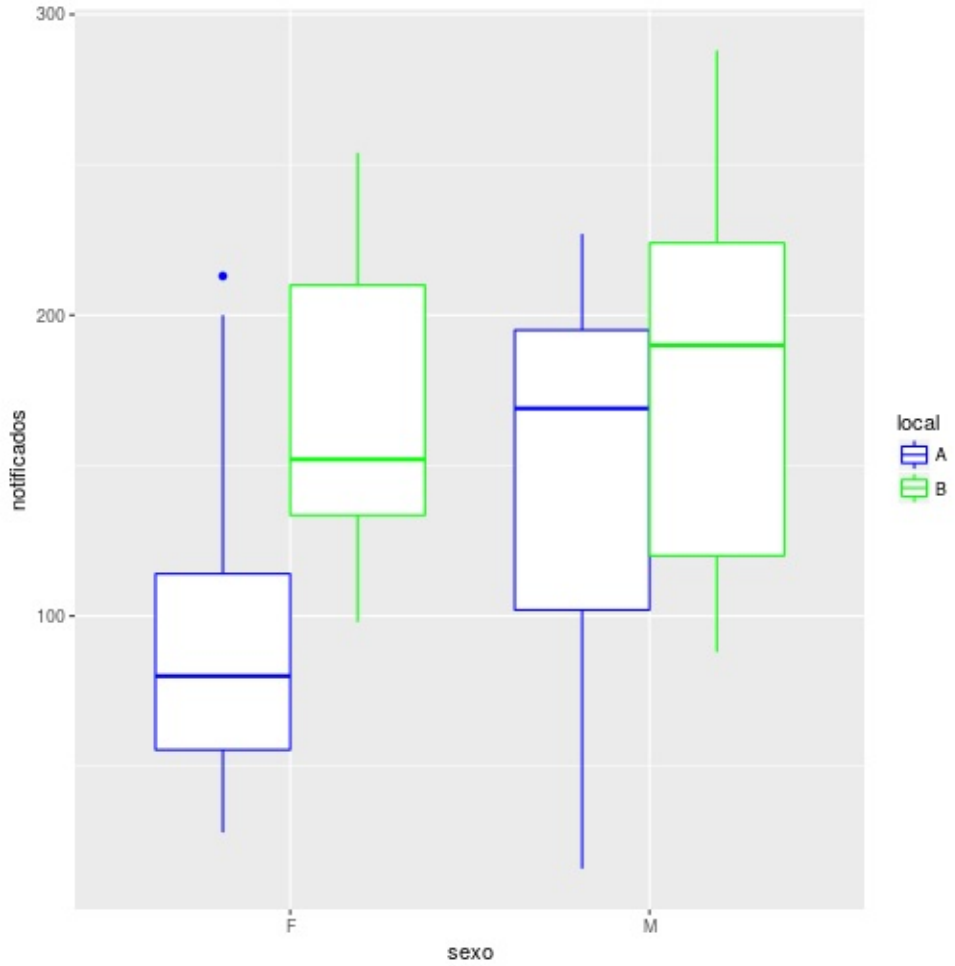
```
> ggplot(casos, aes(sexo, notificados, fill = local)) +
+   geom_boxplot()
```




```
> ggplot(casos, aes(sexo, notificados, fill = local)) +
+   geom_boxplot() +
+   scale_fill_manual(values = c('lightblue', 'darkred'))
```

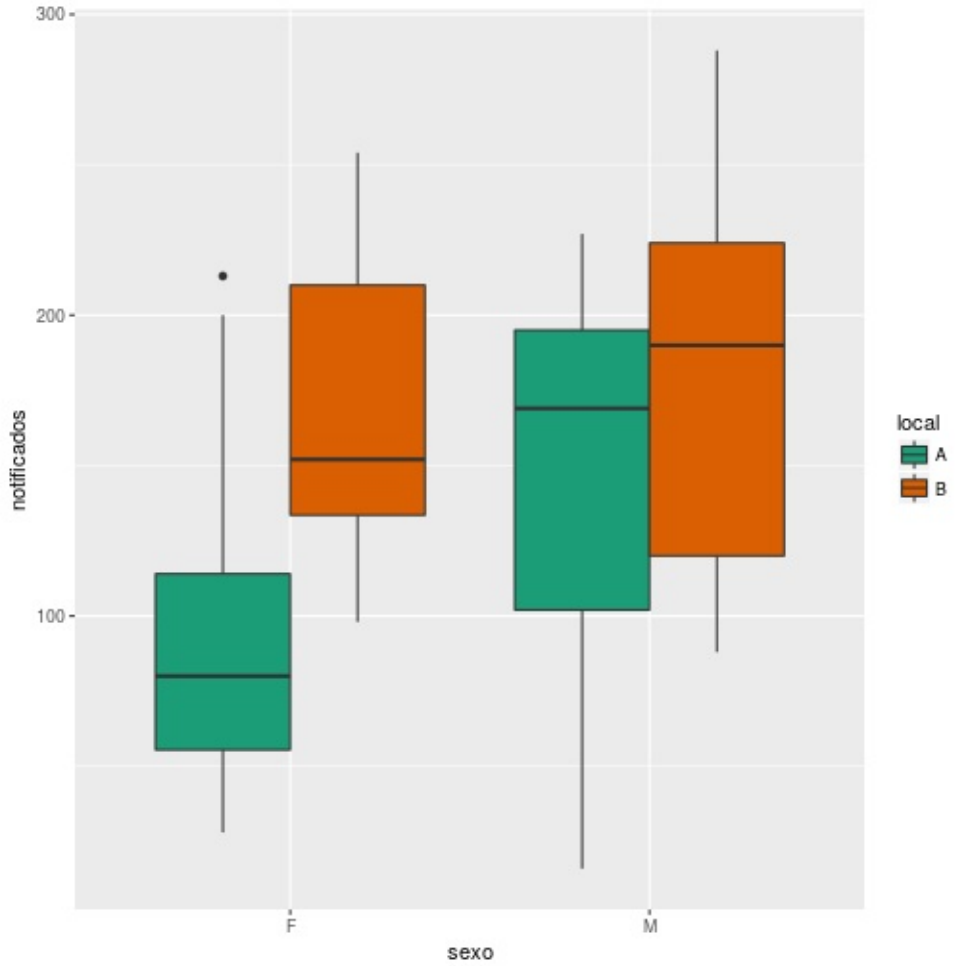



```
> ggplot(casos, aes(sexo, notificados, color = local)) +
+   geom_boxplot() +
+   scale_color_manual(values = c('blue', 'green'))
```



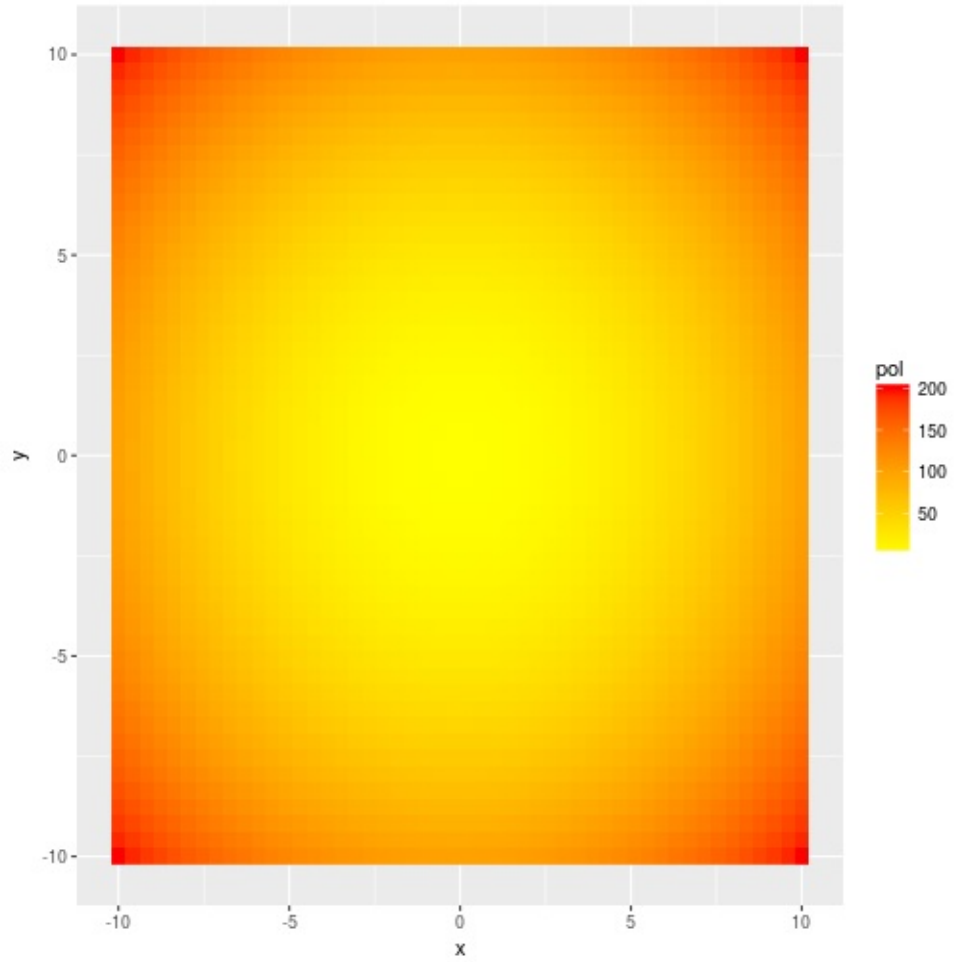
No lugar de definirmos manualmente a cor de cada categoria, podemos escolher paletas de cores predefinidas (ver *r color brewer palette* no google), com a função `scale_fill_brewer` (`scale_color_brewer`).

```
> ggplot(casos, aes(sexo, notificados, fill = local)) +
+   geom_boxplot() +
+   scale_fill_brewer(palette = 'Dark2')
```



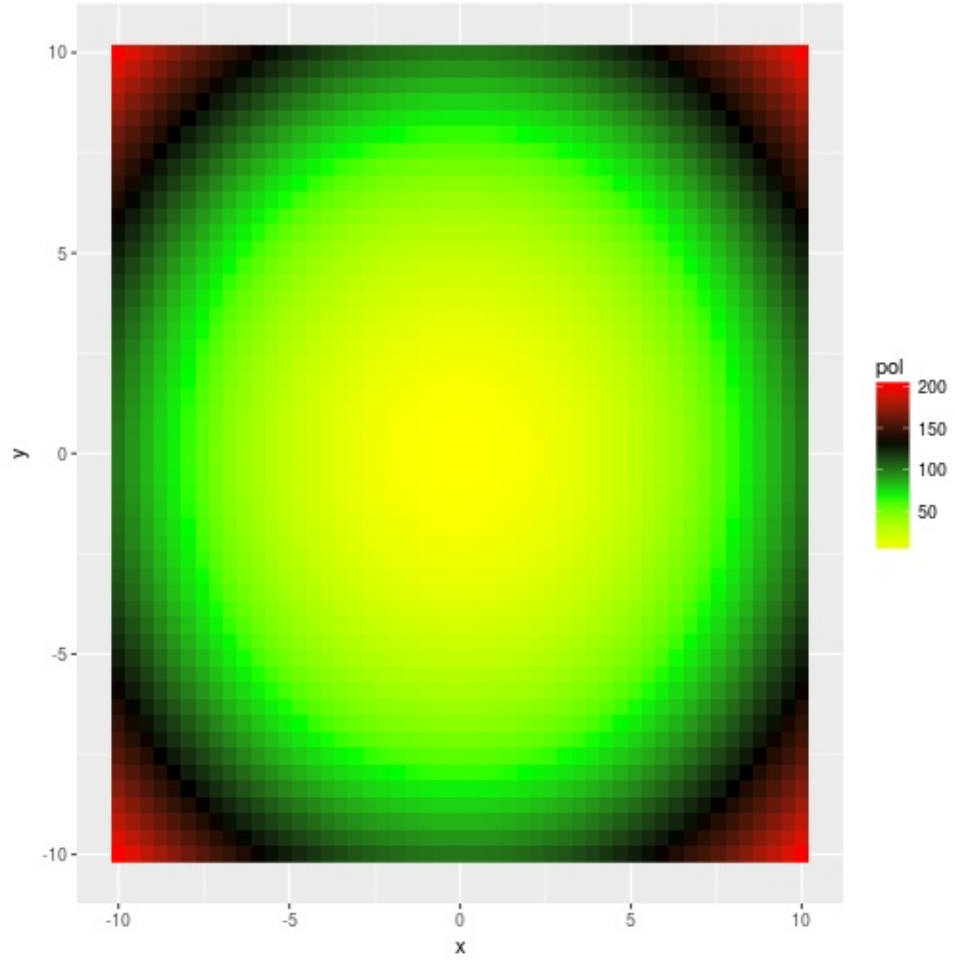
Quando a variável mapeada é quantitativa, devemos usar a função `scale_fill_continuous` (`scale_color_continuous`) e definir os extremos da escala de cor.

```
> ggplot(poluente, aes(x, y, fill = pol)) +
+   geom_raster() +
+   scale_fill_continuous(low = 'yellow', high = 'red')
```



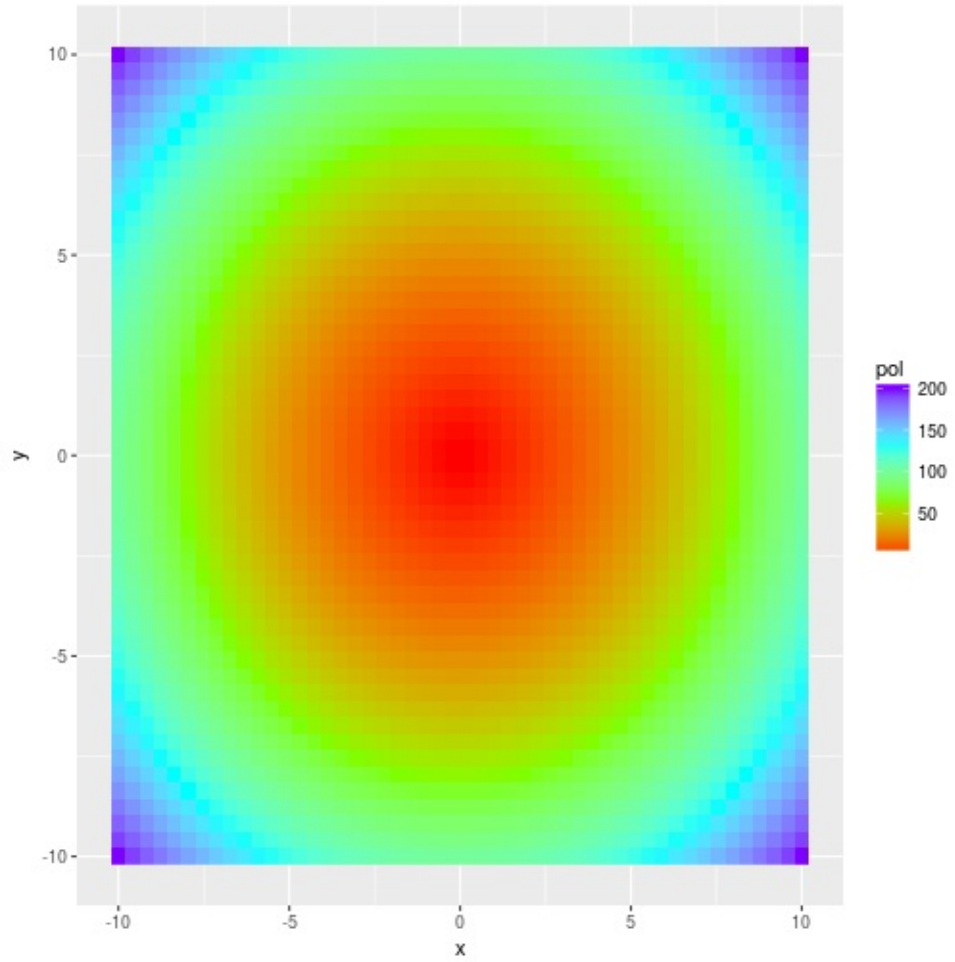
Para incluir mais de duas cores na escala, podemos definir os extremos com mais de uma cor

```
> ggplot(poluente, aes(x, y, fill = pol)) +
+   geom_raster() +
+   scale_fill_continuous(low = c('yellow', 'green'), high = c('black', 'red'))
```

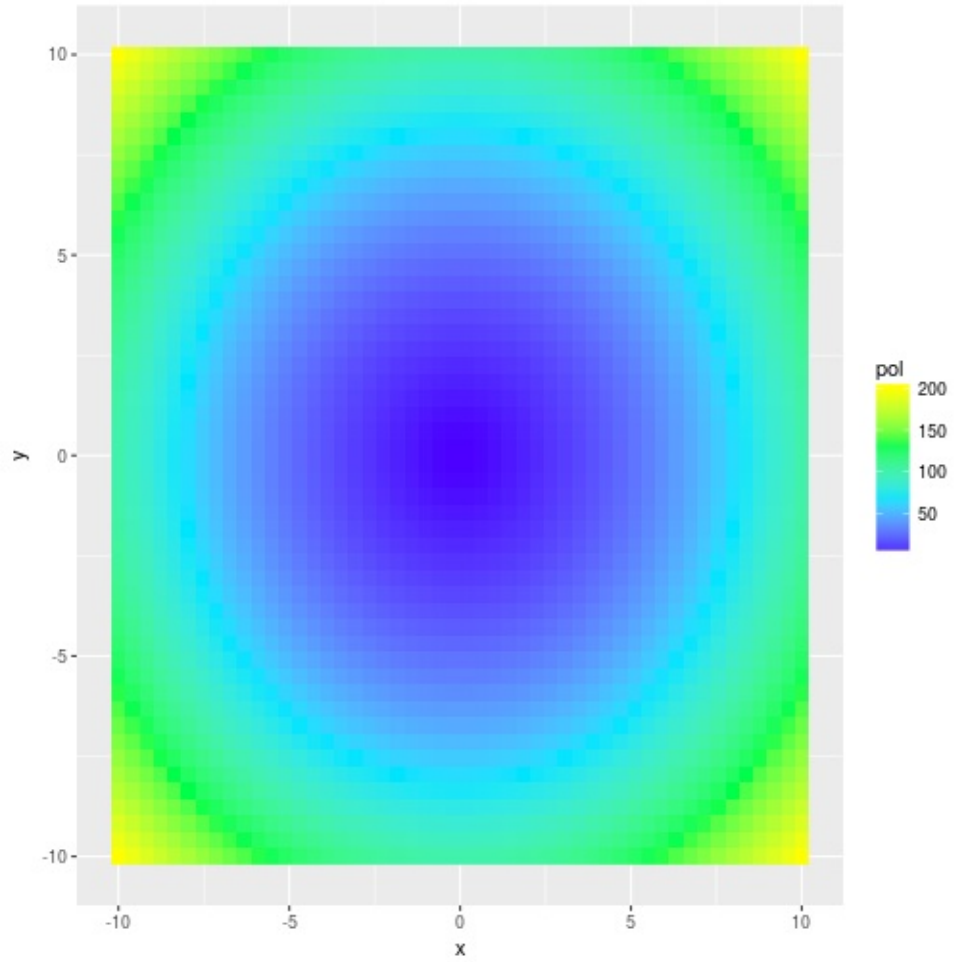


ou usar a função `scale_fill_gradientn` (`scale_color_gradientn`) com escalas de cores predefinidas pelas funções `rainbow` , `topo.colors` ou `terrain.colors` .

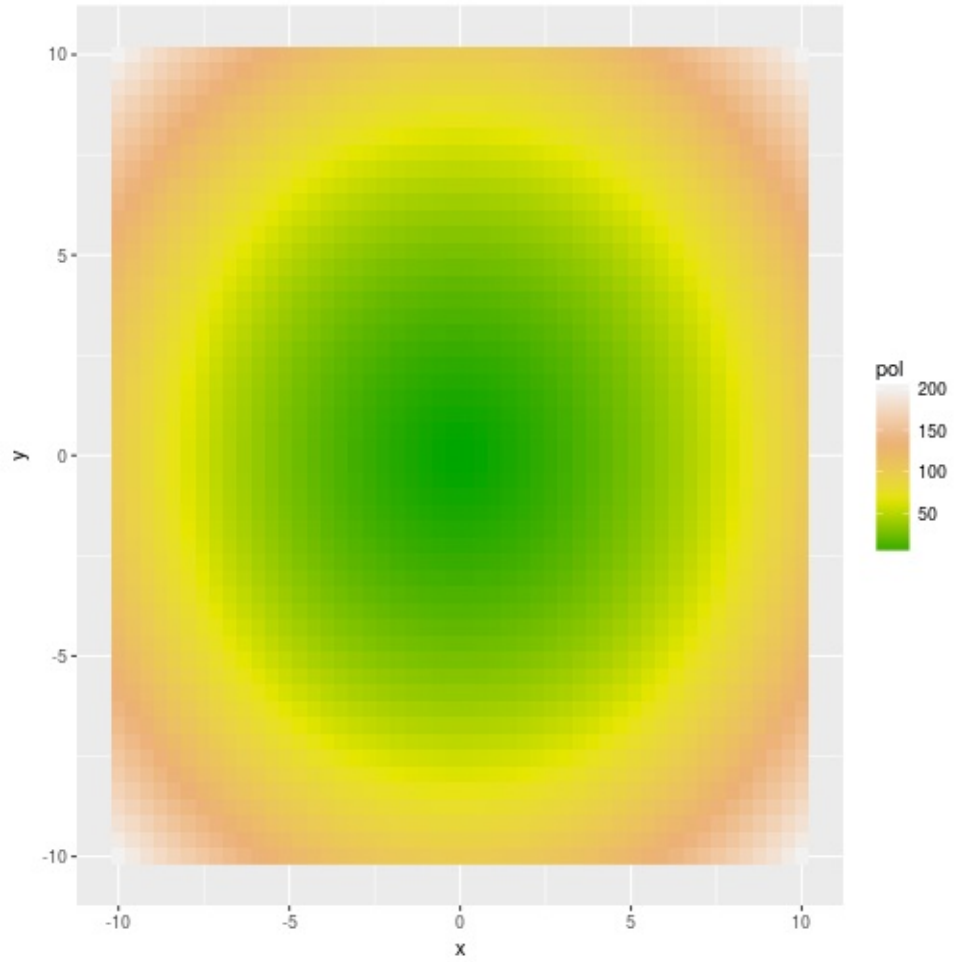
```
> ggplot(poluente, aes(x, y, fill = pol)) +
+   geom_raster() +
+   scale_fill_gradientn(colors = rainbow(4))
```




```
> ggplot(poluente, aes(x, y, fill = pol)) +
+   geom_raster() +
+   scale_fill_gradientn(colors = topo.colors(4))
```



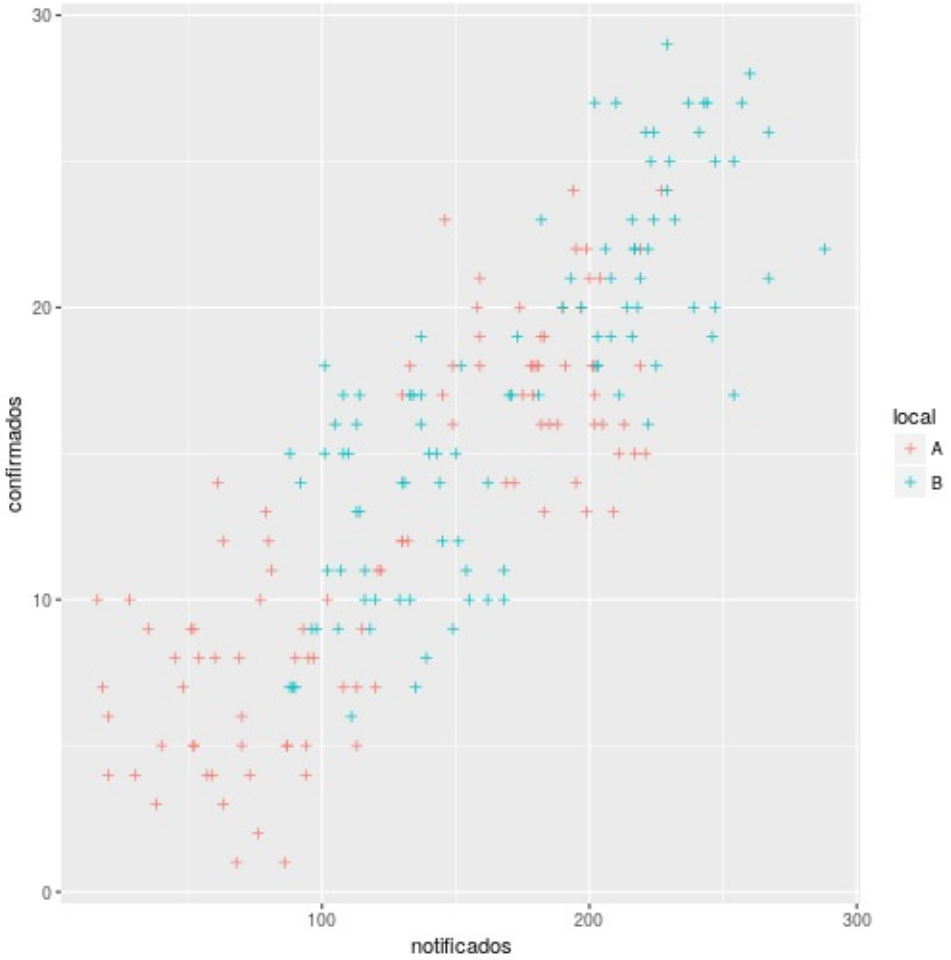

```
> ggplot(poluente, aes(x, y, fill = pol)) +
+   geom_raster() +
+   scale_fill_gradientn(colors = terrain.colors(4))
```



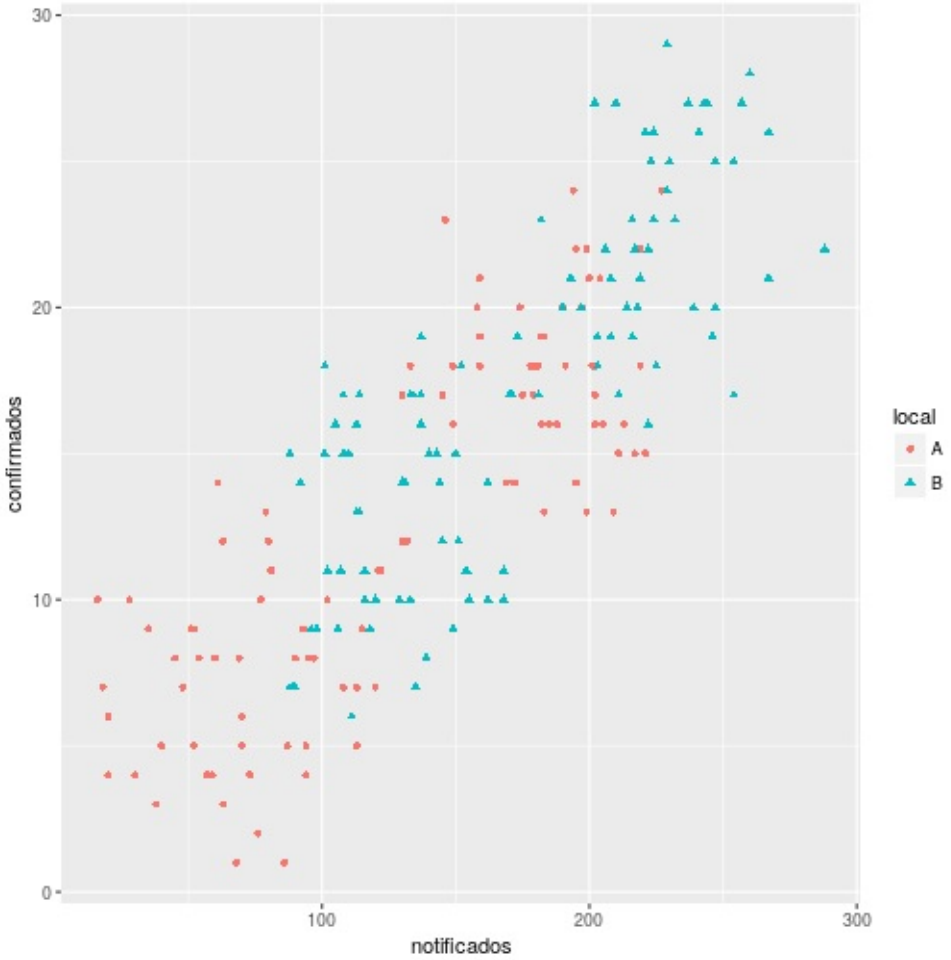
Formas dos pontos e das linhas

Como no caso da cor e outras propriedades estéticas, temos como definir a forma dos pontos e das linhas ou mapear uma variável na forma.

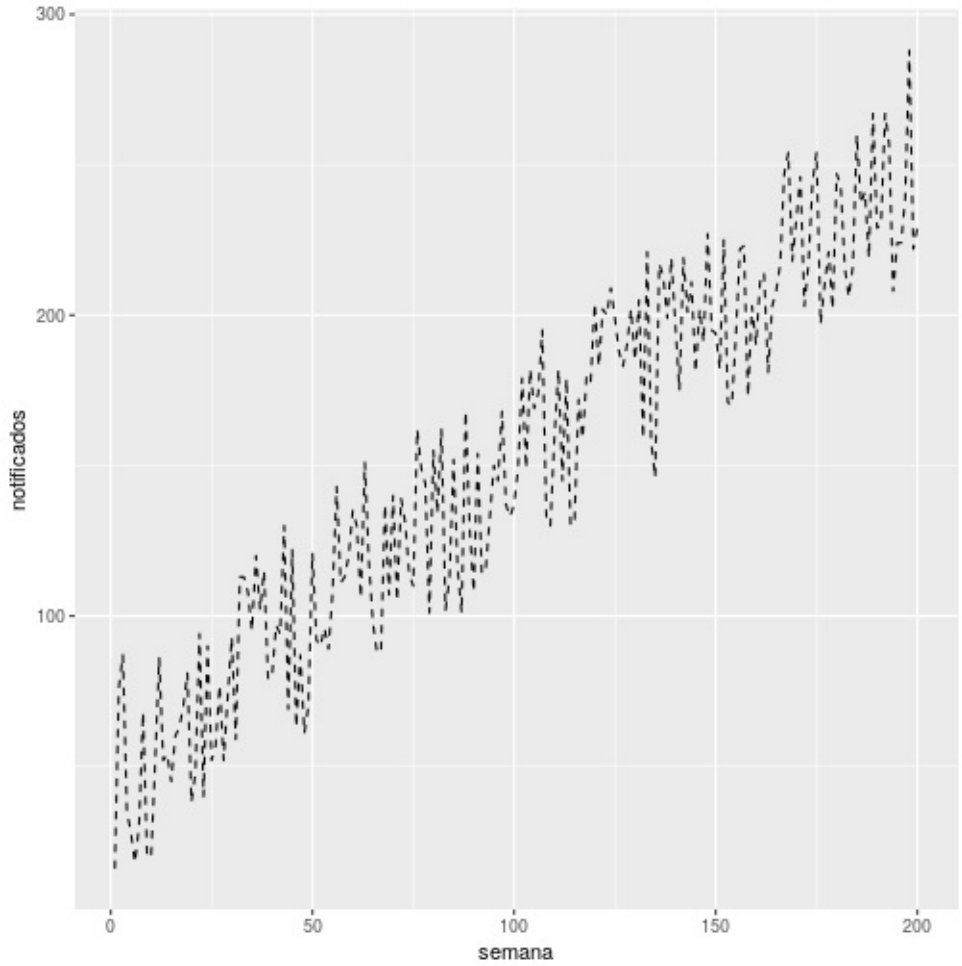
```
> ggplot(casos, aes(notificados, confirmados, color = local)) +
+   geom_point(shape = 3)
```



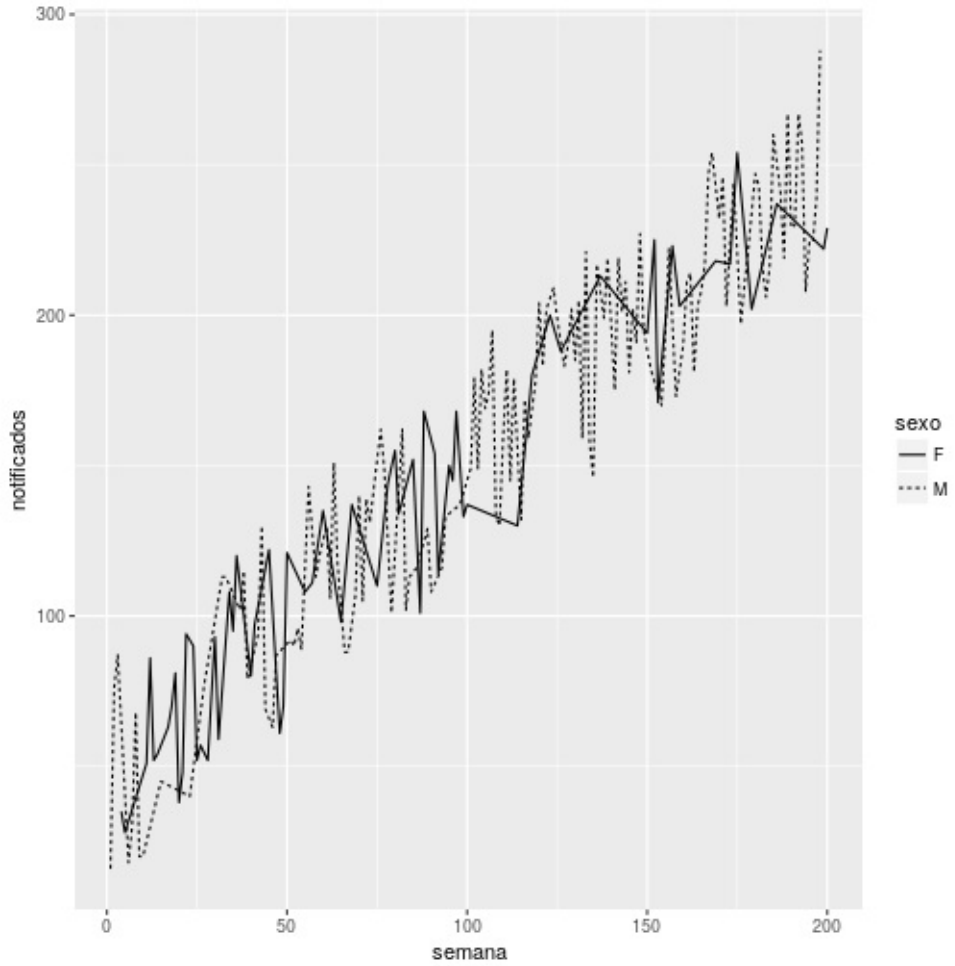

```
> ggplot(casos, aes(notificados, confirmados,
+                   color = local, shape = local)) +
+   geom_point()
```




```
> ggplot(casos, aes(semana, notificados)) +
+   geom_line(linetype = 2)
```

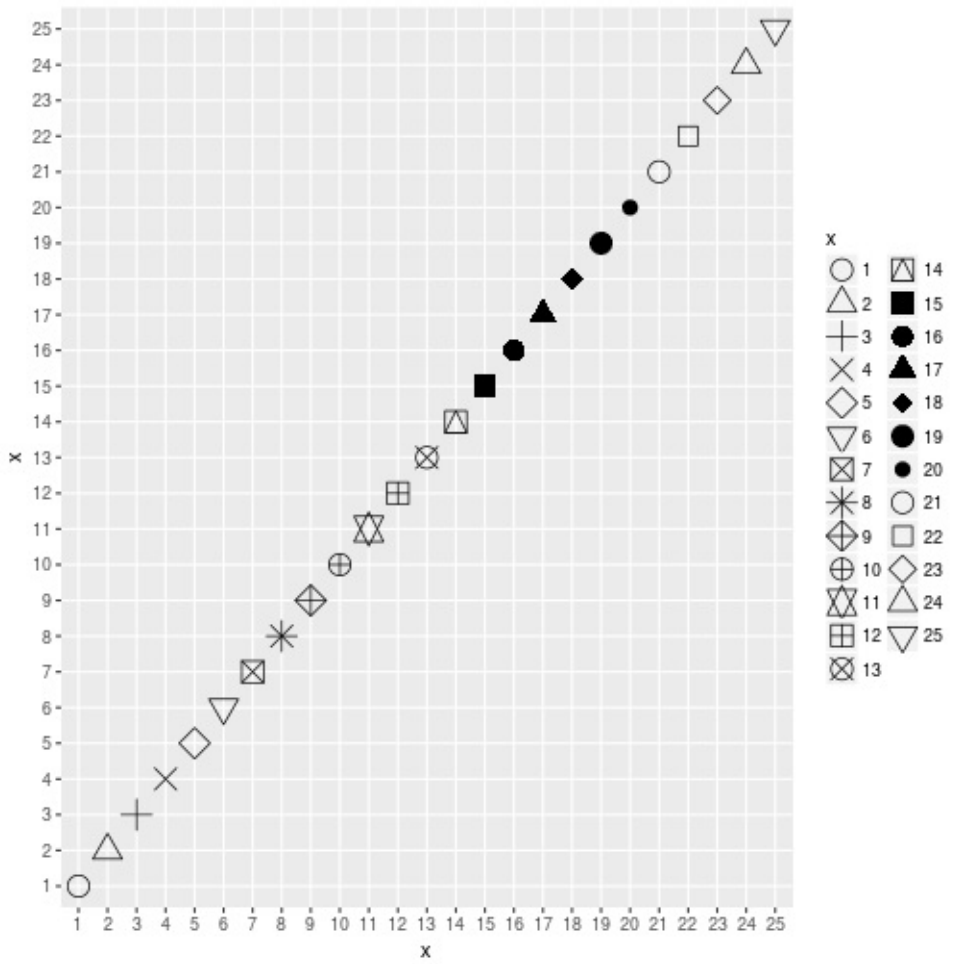



```
> ggplot(casos, aes(semana, notificados, linetype = sexo)) +
+   geom_line()
```

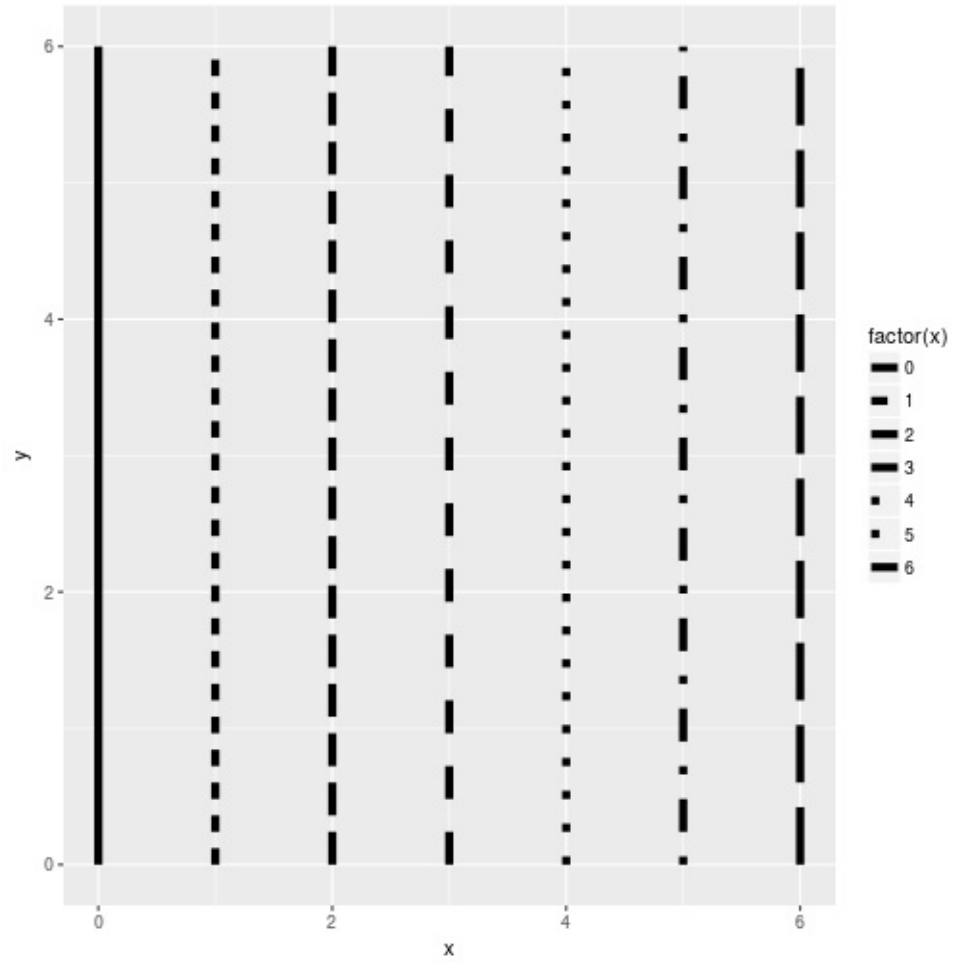


Pontos e linhas disponíveis:

```
> ggplot(data.frame(x = factor(1:25)), aes(x = x, y = x, shape = x)) +
+   geom_point(size = 5) +
+   scale_shape_manual(values = 1:25)
```



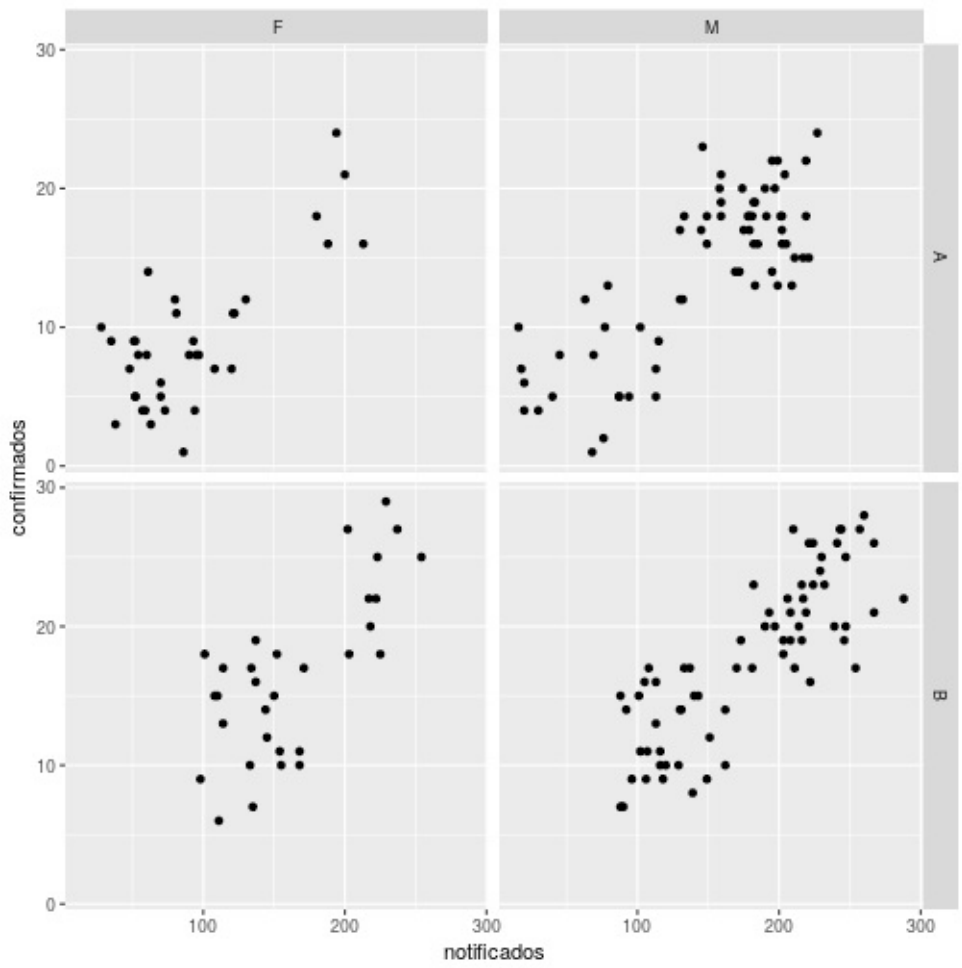

```
> ggplot(data.frame(x = rep(0:6, 10), y = rep(0:6, e = 10)),
+        aes(x, y, linetype = factor(x))) +
+   geom_line(size = 2)
```



Gráficos multivariados

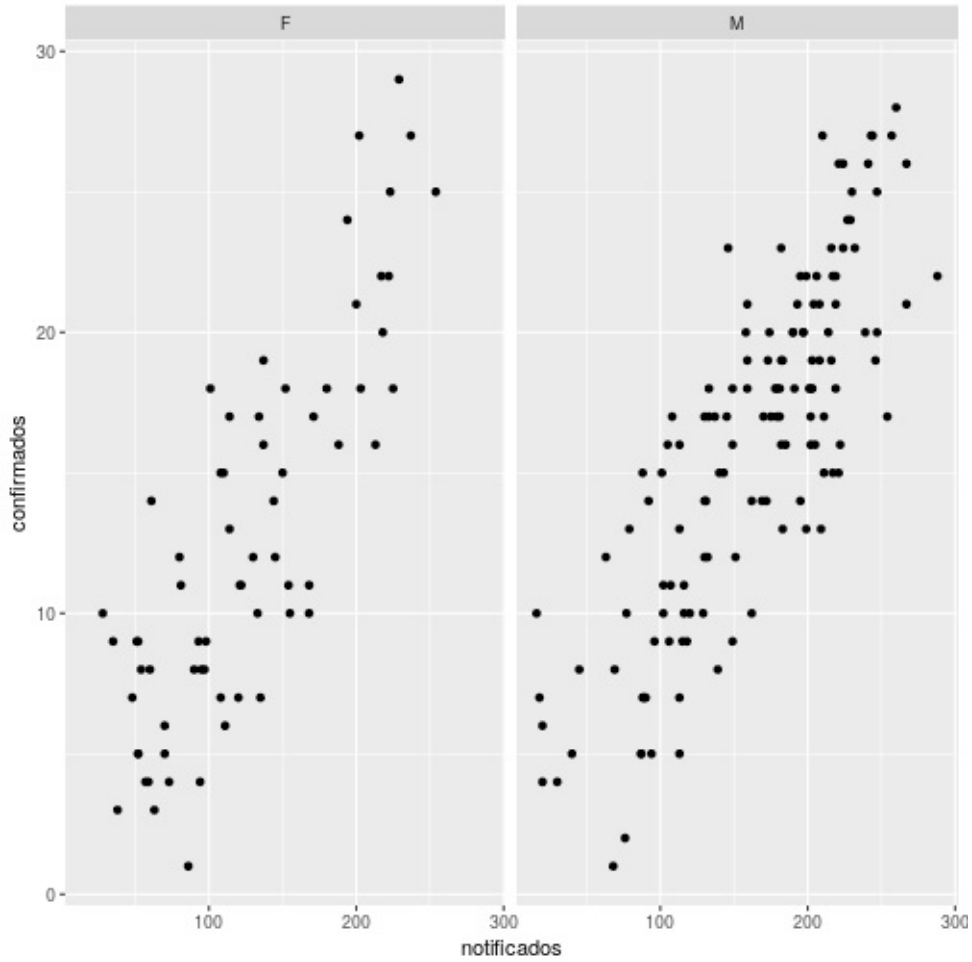
No começo do capítulo criamos um gráfico multivariado.

```
> ggplot(data = casos, aes(notificados, confirmados)) +
+   geom_point() +
+   facet_grid(local ~ sexo)
```

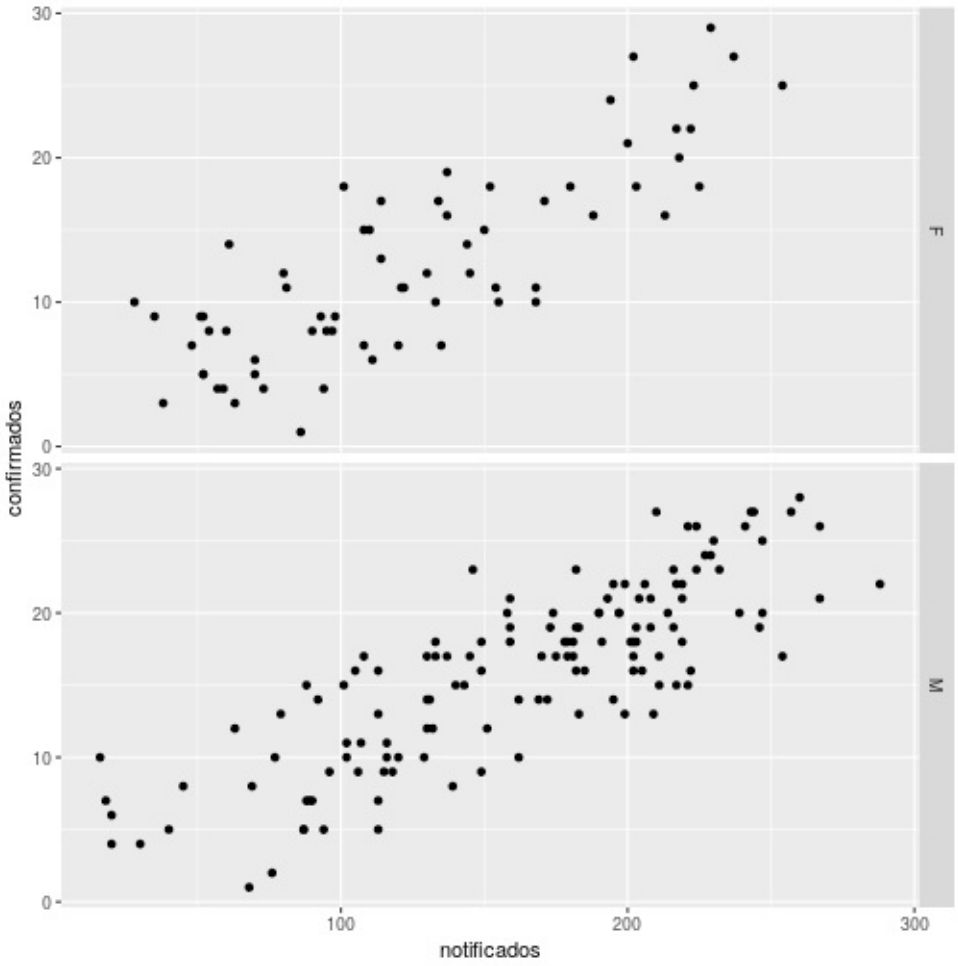


A função `facet_grid` mapeia variáveis qualitativas nas linhas e colunas de uma matriz de gráficos. No exemplo anterior mapeamos o local nas linhas e o sexo nas colunas. Para mapear só nas linhas ou nas colunas, há que colocar um ponto na dimensão que não se quer mapear.

```
> ggplot(data = casos, aes(notificados, confirmados)) +
+   geom_point() +
+   facet_grid(. ~ sexo)
```

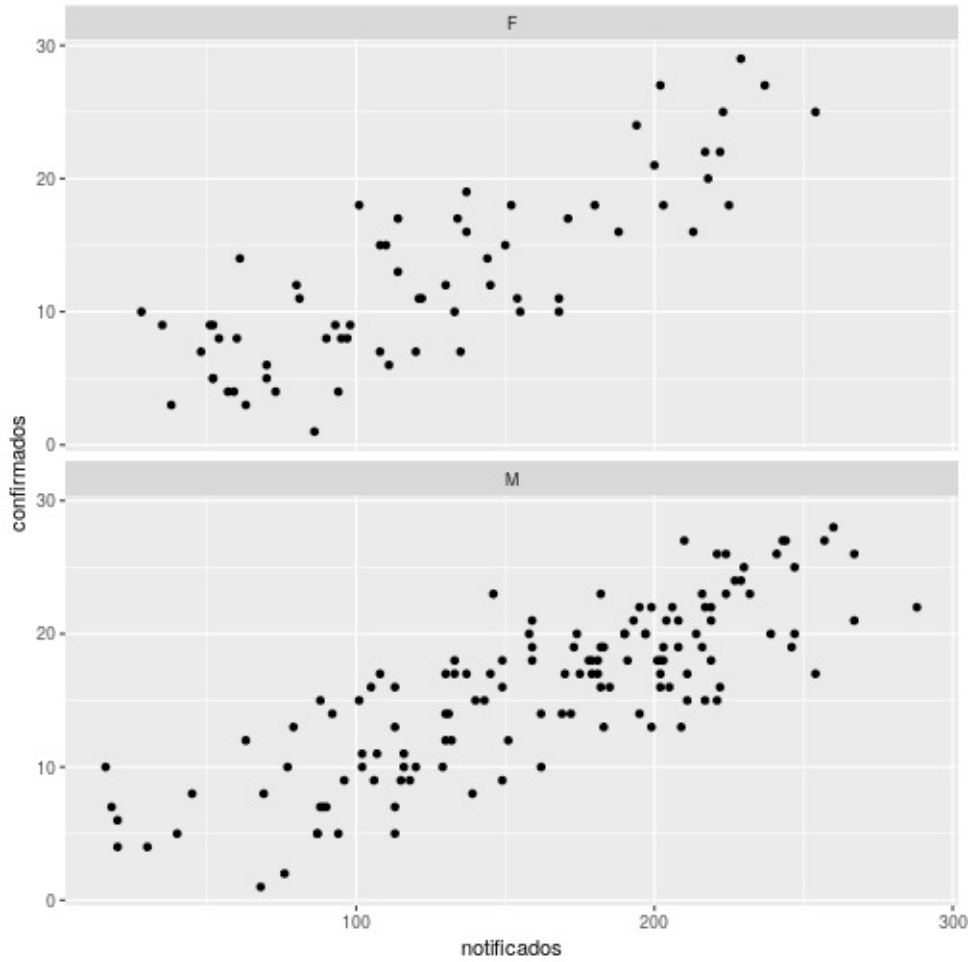



```
> ggplot(data = casos, aes(notificados, confirmados)) +
+   geom_point() +
+   facet_grid(sexo ~ .)
```



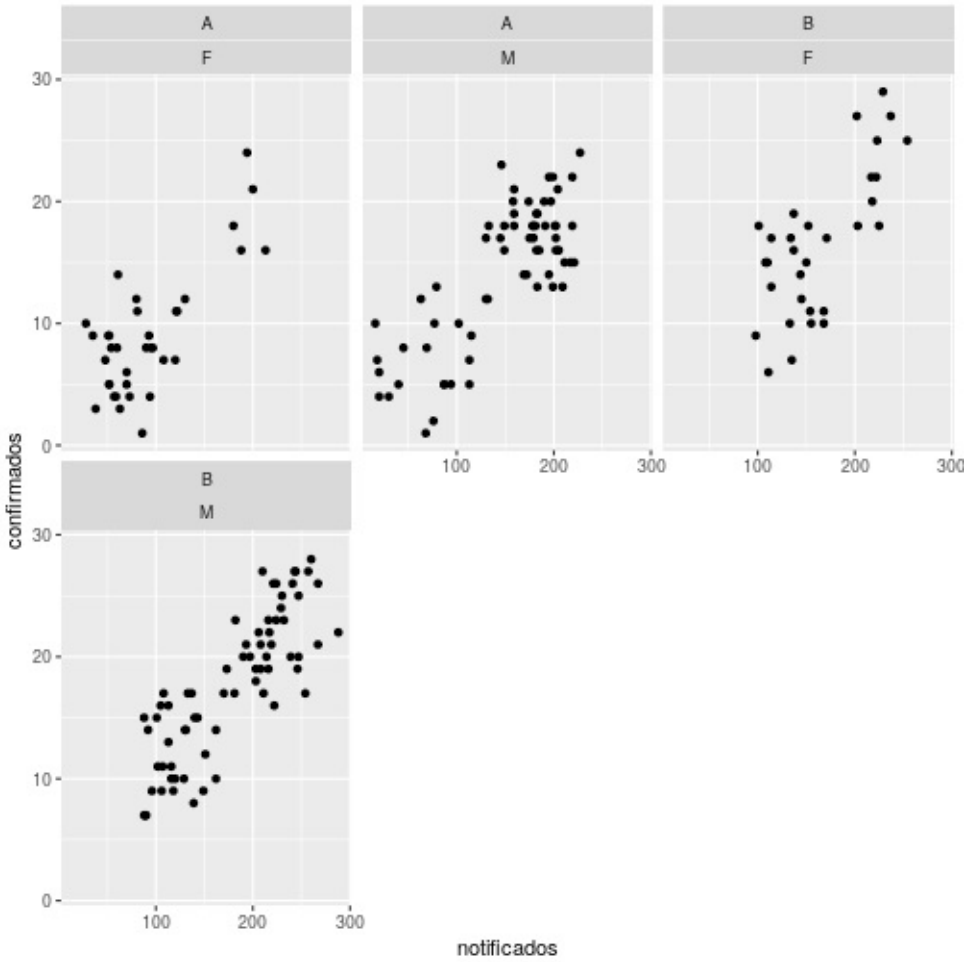
`facet_wrap` também gera gráficos multivariados e é conveniente quando se tem uma única variável condicionante e se quer controlar o número de colunas da matriz de gráficos

```
> ggplot(data = casos, aes(notificados, confirmados)) +
+   geom_point() +
+   facet_wrap(~ sexo, ncol = 1)
```



Quando se usam duas variáveis condicionantes, as combinações das categorias das mesmas são representadas em um único cabecalho (acima, no lugar de acima e à direita).

```
> ggplot(data = casos, aes(notificados, confirmados)) +
+   geom_point() +
+   facet_wrap(local ~ sexo, ncol = 3)
```

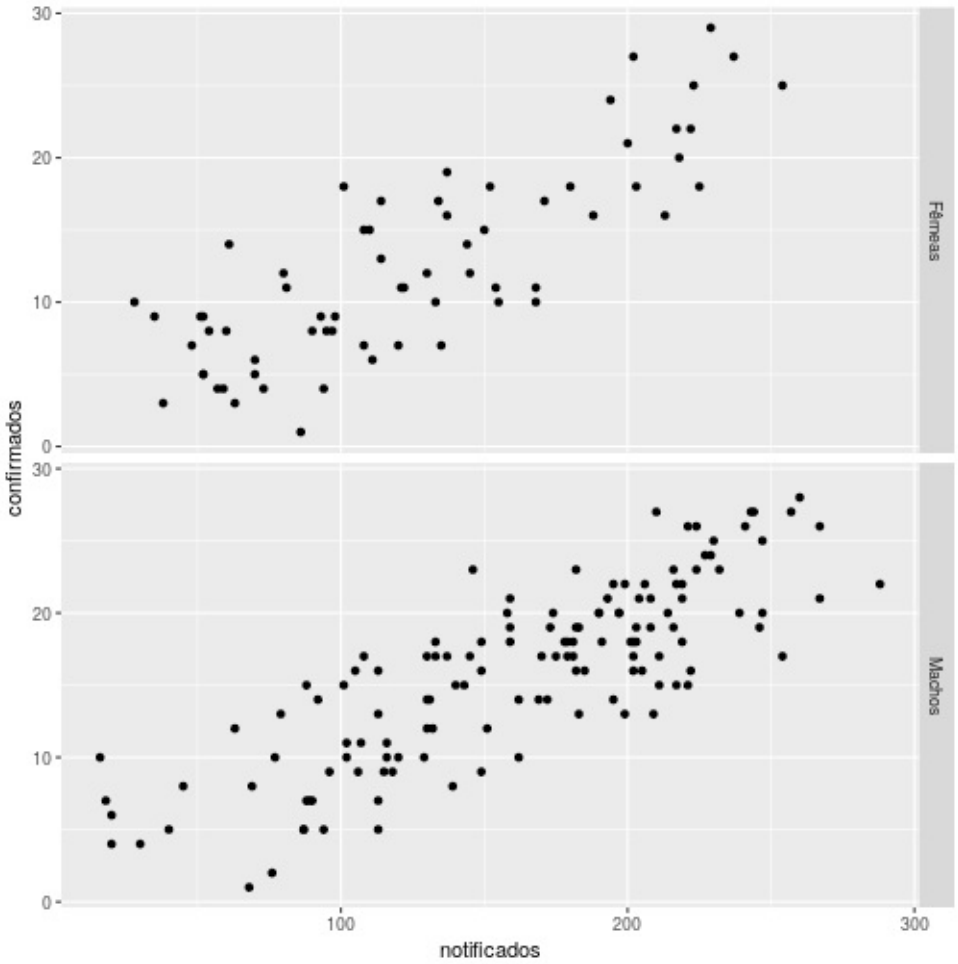


A forma mais simples de mudar as etiquetas de cada painel é definindo as etiquetas do fator usado para condicionar o gráfico.

```
> levels(casos$sexo)
```

```
[1] "F" "M"
```

```
> casos$sexo <- factor(as.character(casos$sexo), levels = c('F', 'M'),
+                      labels = c('Fêmeas', 'Machos'))
> ggplot(data = casos, aes(notificados, confirmados)) +
+   geom_point() +
+   facet_grid(sexo ~ .)
```

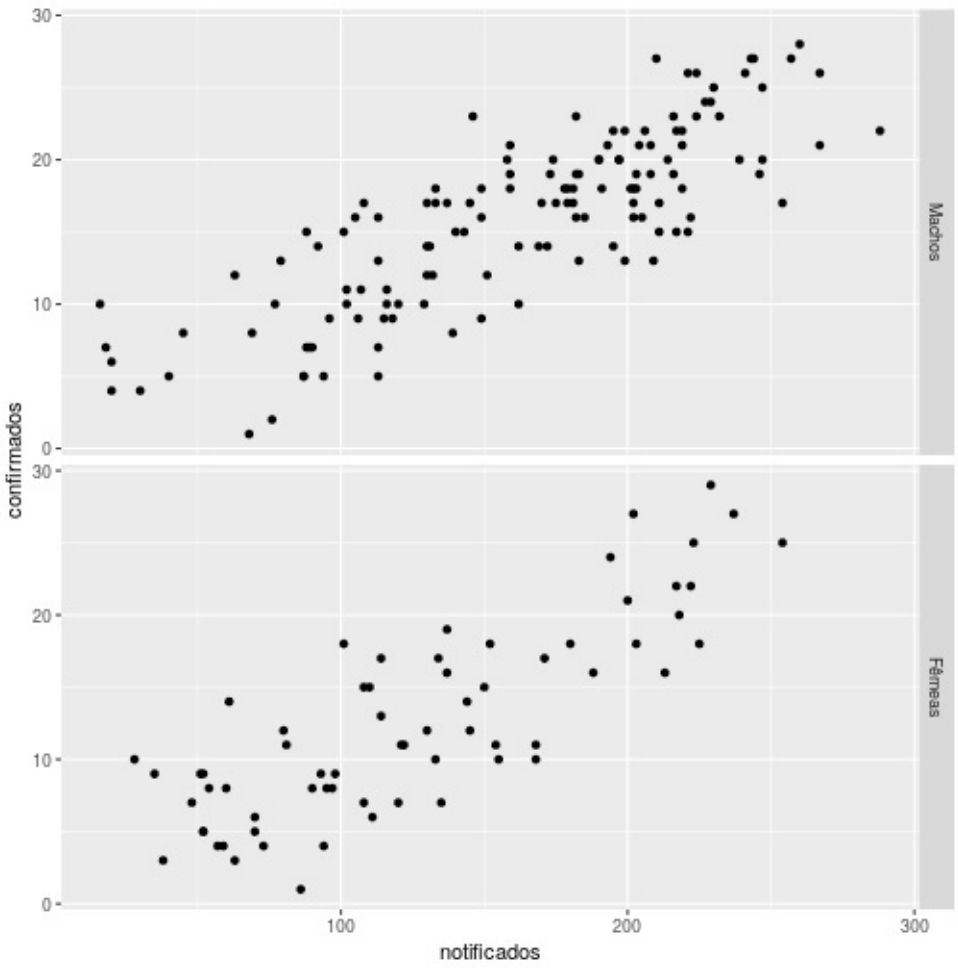


Mudando a ordem dos níveis do fator também mudamos a ordem dos paineis.

```
> levels(casos$sexo)
```

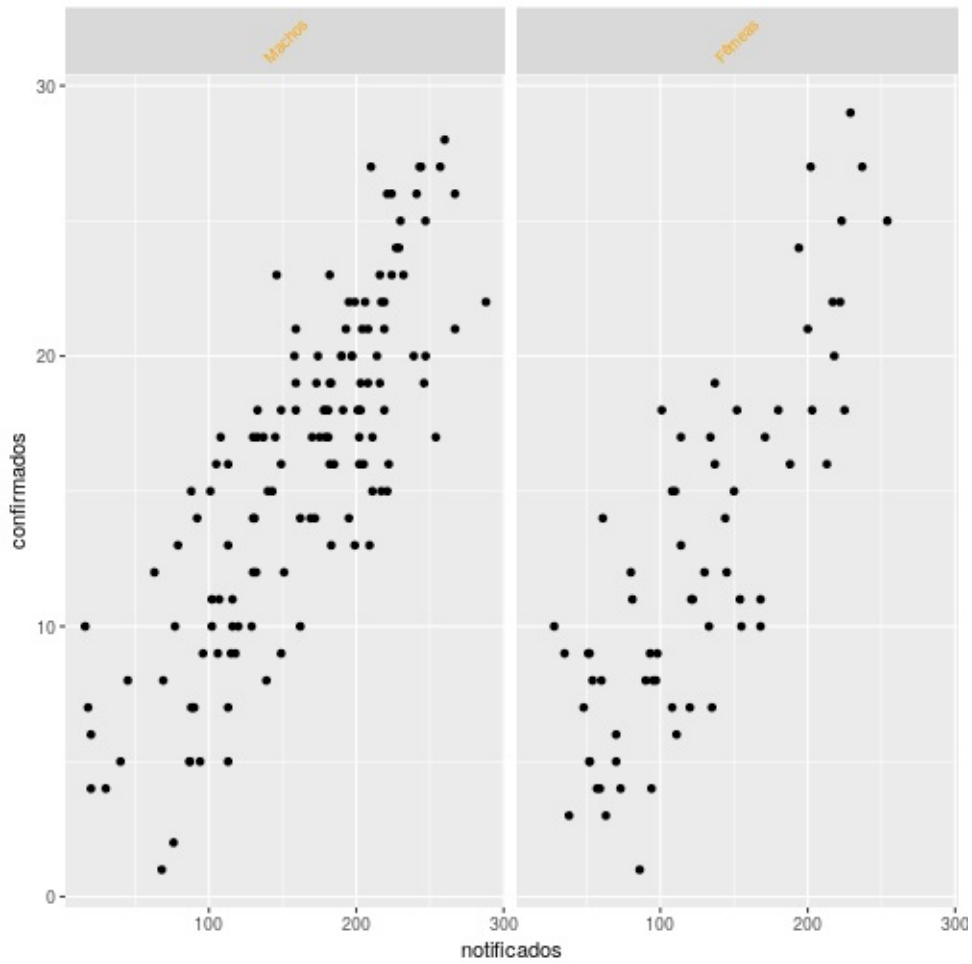
```
[1] "Fêmeas" "Machos"
```

```
> casos$sexo <- factor(casos$sexo, levels = c('Machos', 'Fêmeas'))
> ggplot(data = casos, aes(notificados, confirmados)) +
+   geom_point() +
+   facet_grid(sexo ~ .)
```



Com o elemento `strip.text.x` modificamos a apariência das etiquetas dos paineis.

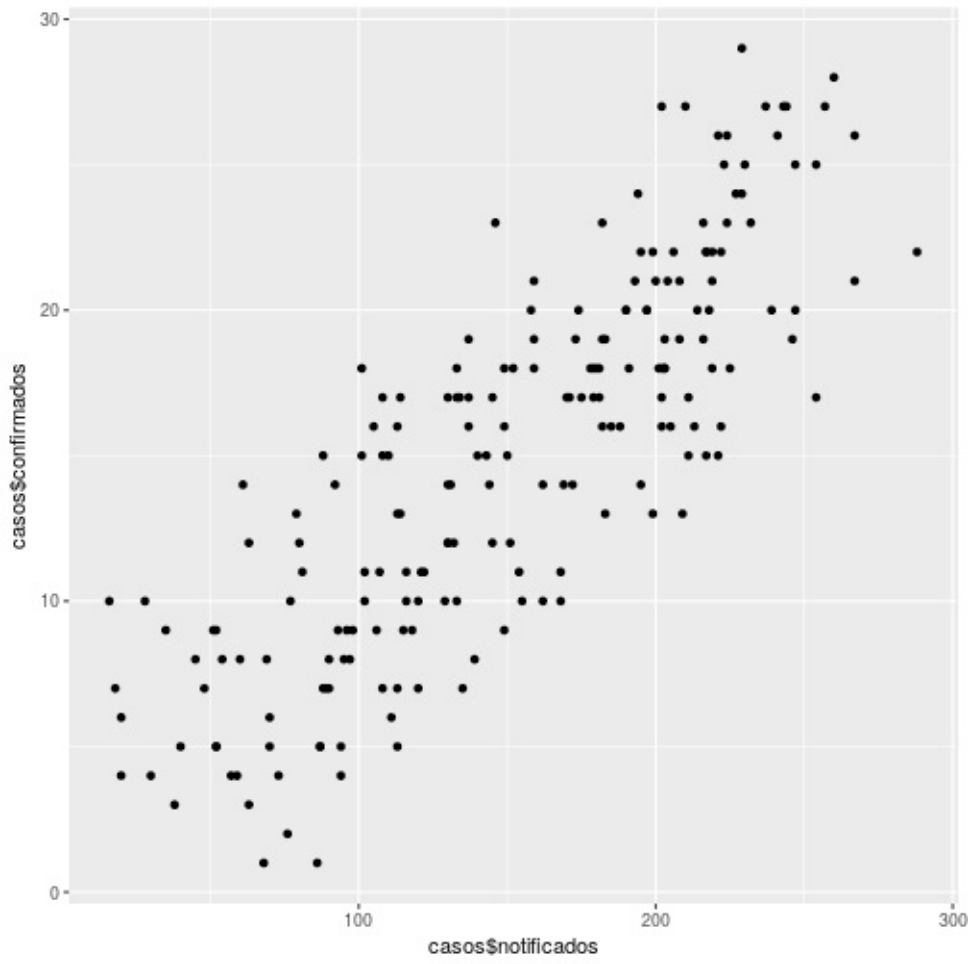
```
> ggplot(data = casos, aes(notificados, confirmados)) +
+   geom_point() +
+   facet_grid(. ~ sexo) +
+   theme(strip.text.x = element_text(size = 8, colour = 'orange', angle = 45))
```



Sintaxe abreviada

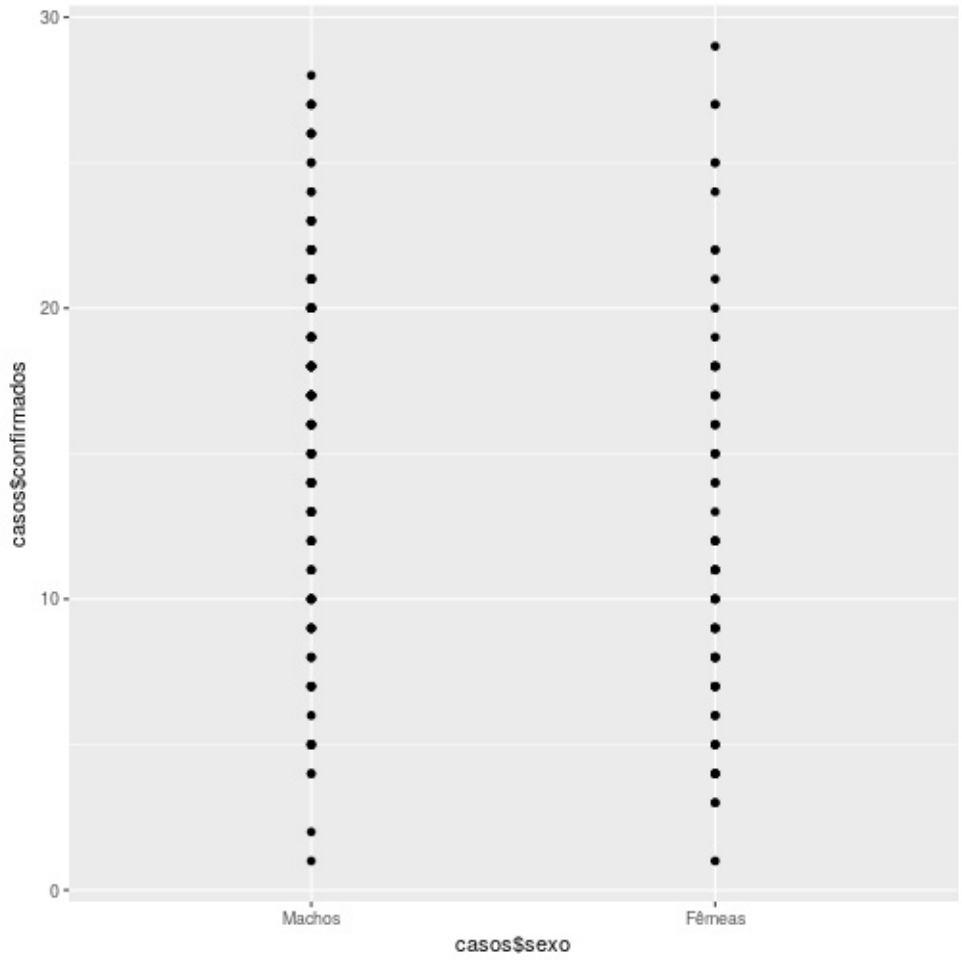
O `ggplot2` tem a função `qplot` que usa uma sintaxe abreviada. Os dois primeiros argumentos são `x` e `y`, e a geometria é escolhida automaticamente.

```
> qplot(casos$notificados, casos$confirmados)
```

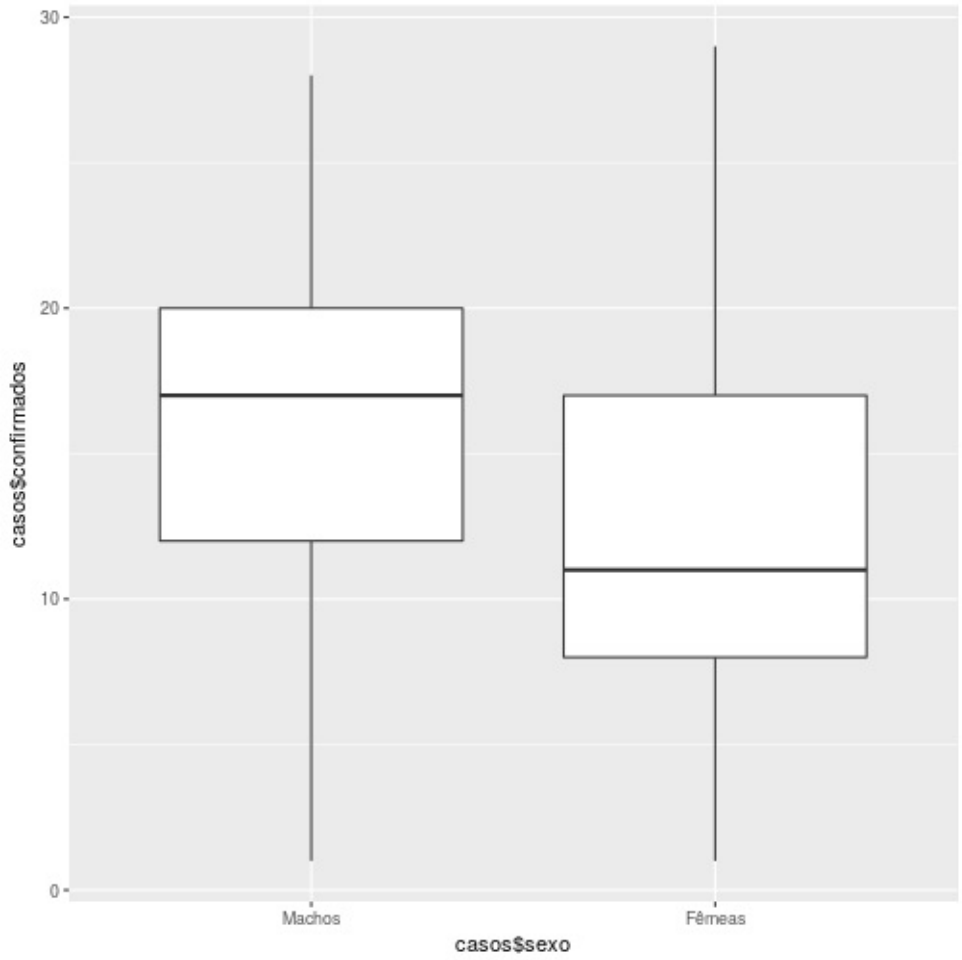


Se a geometria escolhida não é adequada, o argumento `geom` permite definir a gemoetria desejada.

```
> qplot(casos$sexo, casos$confirmados)
```

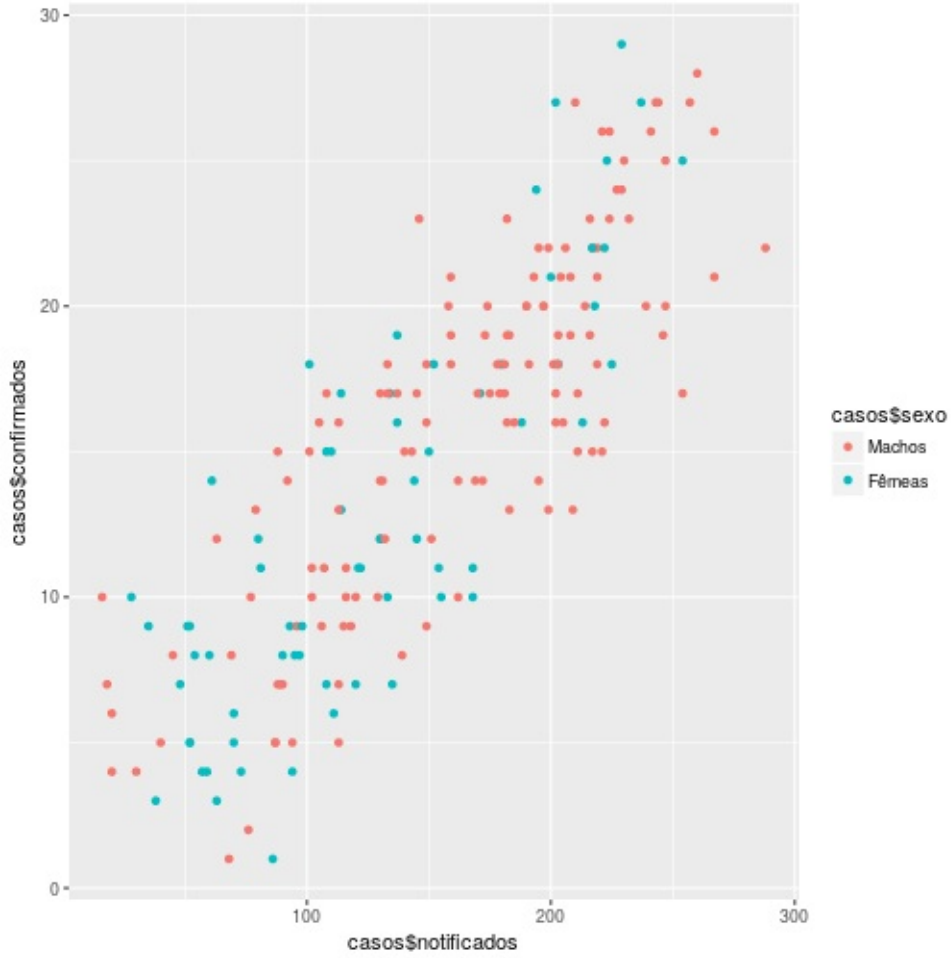



```
> qplot(casos$sexo, casos$confirmados, geom = 'boxplot')
```



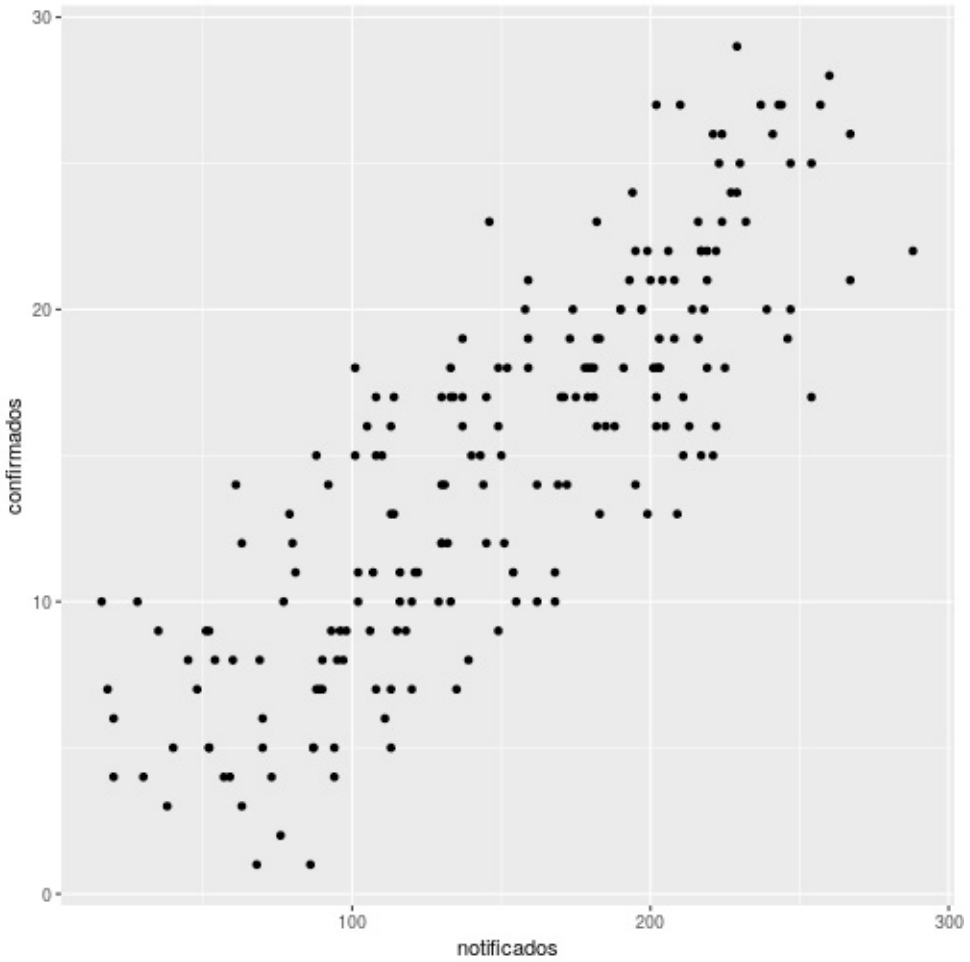
Outras propriedades estéticas também são definíveis dentro de `qplot`,

```
> qplot(casos$notificados, casos$confirmados, color = casos$sexo)
```



e alterantivamente, podemos especificar só o nome das variáveis se especificamos o banco no argumento `data`.

```
> qplot(notificados, confirmados, data = casos)
```


30
20
10
0
confirmados
100
200
300
notificados

Estruturas de controle

As estruturas de controle (estruturas ou expressões condicionais) são códigos cuja execução depende de uma condição. Por exemplo, se `x` e `y` são dois números a serem somados só se `x` é maior do que `y`, podemos usar uma estrutura de controle na qual a condição é `x > y`, e o código executado se a condição é verdadeira é `x + y`.

if

A função `if` serve justamente para criar estruturas de controle como a descrita anteriormente. A condição é definida dentro de parêntesis logo após a função `if` e o código a ser executado caso a condição seja verdadeira é agrupado por chaves.

```
> x <- 4; y <- 3
> if (x > y) {
+   x + y
+ }
```

```
[1] 7
```

```
> x <- 4; y <- 5
> if (x > y) {
+   x + y
+ }
```

Para continuarmos explorando estruturas de controle, criemos um banco com o número de casos por 100 mil habitantes de uma doença, por 3 anos, em 26 regiões.

```
> set.seed(3)
> length(letters)
```

```
[1] 26
```

```
> banco <- data.frame(regiao = letters,
+                     ano1 = rpois(26, 50),
+                     ano2 = rpois(26, 50),
+                     ano3 = rpois(26, 50),
+                     stringsAsFactors = F)
```

Supondo que o risco da doença é alto em uma região se o total de casos é superior a 150, podemos usar a função `if` para criar uma estrutura de controle que imprime o texto `'alto'` se a condição é verdadeira para uma dada região.

```
> if (sum(banco[banco$regiao == 'd', -1]) > 150) {
+   'alto'
+ }
```

```
[1] "alto"
```

else

A função `else` é um complemento da função `if`. Se usarmos a estrutura de controle anterior para classificar o risco da região *m*, nenhum texto é impresso porque a condição é falsa.

```
> if (sum(banco[banco$regiao == 'm', -1]) > 150) {
+   'alto'
+ }
```

Com a função `else`, podemos executar um dado código (por exemplo a impressão de `baixo` se o número de casos é meni=or ou igual a 150) se a condição do `if` é falsa.

```
> if (sum(banco[banco$regiao == 'm', -1]) > 150) {
+   'alto'
+ } else {
+   'baixo'
+ }
```

```
[1] "baixo"
```

Adicionalmente, a função `else` pode ser complementada por outra função `if`.

```
> if (sum(banco[banco$regiao == 'm', -1]) > 145) {
+   'alto'
+ } else if (sum(banco[banco$regiao == 'm', -1]) < 135) {
+   'baixo'
+ } else {
+   'medio'
+ }
```

```
[1] "medio"
```

`ifelse` é outra alternativa para construir estruturas de controle simples. Se a condição é verdadeira, o segundo argumento é executado, caso contrário, é executado o terceiro.

```
> ifelse(sum(banco[3, -1]) > 150, 'alta', 'baixa')
```

```
[1] "baixa"
```

for

Os laços de repetição (loops) são estruturas de controle que como o nome sugere, repetem um código de acordo com uma dada condição. Com a função `for` que permite criar laços de repetição, um índice assume uma sequência de valores e o código da estrutura de controle é executado tantas vezes quanto valores na sequência. Por exemplo, se o índice *i* assume os valores 1, 2, e 3 (`i in 1:3`), o código da estrutura de controle é executado 3 vezes.

```
> for (i in 1:3) {
+   print('repete')
+ }
```

```
[1] "repete"
[1] "repete"
[1] "repete"
```

Reparem que com `for` a condição é dada por `i in 1:3` (não é uma expressão lógica) e é necessário o uso da função `print` para imprimir o conteúdo de um objeto.

```
> for (i in 1:10) {
+   print(i)
+ }
```

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

A ideia dos laços de repetição é automatizar procedimentos repetitivos. Por exemplo, se queremos realizar três simulações do número de cães por domicílio com base em uma distribuição de Poisson, variando o número de domicílios e o parâmetro *lambda*, para posteriormente armazenar os resultados em uma lista, duas possíveis alternativas são: realizar cada uma das simulações separadamente ou usar um laço de repetição. Criemos uma lista vazia com comprimento igual a 3 (`vector('list', length = 3)`) para armazenar os resultados e depois implementemos cada uma das alternativas.

Alternativa 1:

```
> caes_por_domicilio <- vector('list', length = 3)
> set.seed(2)
> caes_por_domicilio[[1]] <- rpois(20, .7)
> set.seed(2)
> caes_por_domicilio[[2]] <- rpois(30, .85)
> set.seed(2)
> caes_por_domicilio[[3]] <- rpois(50, 1)
> caes_por_domicilio
```

```
[[1]]
 [1] 0 1 1 0 2 2 0 1 0 1 1 0 1 0 0 2 3 0 0 0

[[2]]
 [1] 0 1 1 0 2 2 0 2 1 1 1 0 1 0 0 2 3 0 1 0 1 0 2 0 0 1 0 0 3 0

[[3]]
 [1] 0 1 1 0 3 3 0 2 1 1 1 0 2 0 1 2 3 0 1 0 1 1 2 0 0 1 0 0 3 0 0
[32] 0 2 2 1 1 2 0 1 0 4 0 0 0 3 2 3 0 1 2
```

Alternativa 2:

```
> caes_por_domicilio <- vector('list', length = 3)
> domicilios <- c(20, 30, 50)
> lambdas <- c(.7, .85, 1)
> for (i in 1:length(domicilios)) {
+   set.seed(2)
+   caes_por_domicilio[[i]] <- rpois(domicilios[i], lambdas[i])
+ }
> caes_por_domicilio
```

```
[[1]]
 [1] 0 1 1 0 2 2 0 1 0 1 1 0 1 0 0 2 3 0 0 0

[[2]]
 [1] 0 1 1 0 2 2 0 2 1 1 1 0 1 0 0 2 3 0 1 0 1 0 2 0 0 1 0 0 3 0

[[3]]
 [1] 0 1 1 0 3 3 0 2 1 1 1 0 2 0 1 2 3 0 1 0 1 1 2 0 0 1 0 0 3 0 0
[32] 0 2 2 1 1 2 0 1 0 4 0 0 0 3 2 3 0 1 2
```

O laço de repetição parece mais complicado e gastamos a mesma quantidade de linhas na implementação. Porém, se queremos realizar muitas simulações, o laço de repetição é certamente mais conveniente. Por exemplo, o seguinte código realiza 100 simulações com um laço de repetição, gastando o mesmo número de linhas.

```
> caes_por_domicilio <- vector('list', length = 100)
> domicilios <- sample(20:60, 100, r = T)
> lambdas <- runif(100, .7, 1)
> for (i in 1:length(domicilios)) {
+   set.seed(2)
+   caes_por_domicilio[[i]] <- rpois(domicilios[i], lambdas[i])
+ }
> length(caes_por_domicilio)
```

```
[1] 100
```

while

Outra forma de criar estruturas de controle é com a função `while` que repete um dado código enquanto a condição for verdadeira. Se o código não torna falsa a condição, a execução do código continua indefinidamente. Assim, se a condição é que *i* seja menor ou igual a 5 (`i <= 5`), o objeto `i` deve ser instanciado antes de executar a estrutura de controle, e mudar dentro do código desta última até que eventualmente seja igual a 5 para que a execução seja interrompida.

```
> i <- 1
> while(i <= 5) {
+   print(i)
+   i <- i + 1
+ }
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Seguindo a mesma lógica podemos imprimir sequencialmente o nome das regiões enquanto o total de casos no segundo ano seja maior do que 40.

```
> i <- 1
> while (banco[i, 'ano2'] > 40) {
+   print(banco[i, 'regiao'])
+   i <- i + 1
+ }
```

```
[1] "a"
[1] "b"
[1] "c"
[1] "d"
[1] "e"
[1] "f"
[1] "g"
[1] "h"
[1] "i"
[1] "j"
[1] "k"
[1] "l"
[1] "m"
[1] "n"
[1] "o"
[1] "p"
[1] "q"
```

break

`break` é uma função que serve para interromper a execução de uma estrutura de controle. Por exemplo, podemos modificar o código anterior para imprimir sequencialmente o nome das regiões enquanto o total de casos no segundo ano seja maior do que 40 ou igual a 50.

```
> i <- 1
> while (banco[i, 'ano2'] > 40) {
+   print(banco[i, 'regiao'])
+   i <- i + 1
+   if (banco[i, 'ano2'] == 50) {
+     break
+   }
+ }
```

```
[1] "a"
[1] "b"
[1] "c"
```

É claro que o mesmo resultado pode ser obtido criando uma condição com os dois requisitos,

```
> i <- 1
> while (banco[i, 'ano2'] > 40 & banco[i, 'ano2'] != 50) {
+   print(banco[i, 'regiao'])
+   i <- i + 1
+ }
```

```
[1] "a"
[1] "b"
[1] "c"
```

ou usando a função `for`.

```
> for (i in 1:nrow(banco)) {
+   if (banco[i, 'ano2'] == 50 | banco[i, 'ano2'] <= 40) {
+     break
+   }
+   print(banco[i, 'regiao'])
+ }
```

```
[1] "a"
[1] "b"
[1] "c"
```

Estruturas de controle aninhadas

Uma estrutura de controle pode estar aninhada dentro de outra estrutura de controle, sendo possíveis vários níveis de aninhamento. O primeiro e último exemplo da seção `break`, bem como os seguintes, são exemplos de aninhamento.

```
> if (banco[banco$regiao == 'a', 'ano1'] < 50) {
+   print('Ano 1 menor do que 50')
+   if (banco[banco$regiao == 'a', 'ano2'] < 50) {
+     'Ano 2 menor do que 50'
+   }
+ }
```

```
[1] "Ano 1 menor do que 50"
```

```
[1] "Ano 2 menor do que 50"
```

```
> matriz <- matrix(nrow = 5, ncol = 5)
> for (i in 1:nrow(matriz)) {
+   for (j in 1:ncol(matriz)) {
+     matriz[i, j] <- i * j
+   }
+ }
> matriz
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

Laços de repetição e funções vetorizadas

Existem muitas funções que evitam o uso de laços de repetição e geram resultados de forma mais eficiente. Por exemplo, para somar os elementos de posições correspondentes em dois vetores diferentes `x` e `y`, um possível laço de repetição é:

```
> z <- c()
> x <- 1:3
> y <- 11:13
> for (i in 1:3) {
+   z[i] <- x[i] + y[i]
+ }
> z
```

```
[1] 12 14 16
```

Entretanto, é muito mais simples e eficiente usar uma função vetorizada

```
> x + y # o operador + é uma função vetorizada.
```

```
[1] 12 14 16
```

Voltando ao `banco` que criamos no início, se queremos adicionar uma nova coluna `total` com a soma dos casos em cada região, uma estrutura de controle permite-nos acrescentar a soma do número de casos região por região.

```
> banco$total <- 0
> for (i in 1:nrow(banco)) {
+   banco[i, 'total'] <- sum(banco[i, 2:4])
+ }
```

Porém, com a função vetorizada `rowSums` obtemos o mesmo resultado.

```
> banco$total2 <- rowSums(banco[ , 2:4])
> all.equal(banco$total, banco$total2)
```

```
[1] TRUE
```

Para adicionar uma coluna `risco` que identifique como risco alto as regiões com mais de 150 casos e como risco baixo as regiões com 150 ou menos casos, também podemos usar um laço de repetição ou uma estratégia vetorizada com a função `which` .

```
> banco$risco <- rep('baixo', nrow(banco))
> for (i in 1:nrow(banco)) {
+   if (sum(banco[i, 2:4]) > 150) {
+     banco[i, 'risco'] <- 'alto'
+   }
+ }
```

```
> which(c('a', 'b', 'c') == 'b') # Lembrando a função which
```

```
[1] 2
```

```
> banco$risco2 <- rep('baixo', nrow(banco))
> banco[which(rowSums(banco[ , 2:4]) > 150), 'risco2'] <- 'alto'
> all.equal(banco$risco, banco$risco2)
```

```
[1] TRUE
```

Até o exemplo que implementamos com dois `for` aninhados tem uma versão vetorizada.

```
> matriz <- matrix(nrow = 5, ncol = 5)
> for (i in 1:nrow(matriz)) {
+   for (j in 1:ncol(matriz)) {
+     matriz[i, j] <- i * j
+   }
+ }
> matriz
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

```
> outer(1:5, 1:5, '*')
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

Embora seja mais fácil e eficiente usar funções vetorizadas, há procedimentos para os que não existem versões vetorizadas e portanto, os laços de repetição são necessários.

Estruturação de bancos de dados

Um conjunto de dados pode ser estruturado de várias formas, e com frequência uma dada estrutura é mais conveniente para determinados propósitos. Nas análises estatísticas, a estrutura mais comum organiza os dados de tal forma que cada observação é uma linha e cada variável é uma coluna. Essa estrutura é conhecida como *tidy* em inglês e aqui usaremos a tradução *arrumada*. Entretanto, a coleta dos dados as vezes é mais simples usando outras estruturas e os bancos de fontes secundárias nem sempre são disponibilizados na estrutura arrumada.

O pacote `tidyr` possui funções específicas para mudar a estrutura dos bancos de dados, e neste capítulo, seguindo a própria documentação do pacote, veremos como obter uma estrutura arrumada a partir de bancos nos quais:

- Os nomes de colunas são valores de uma variável, não variáveis em si.
- Mais de uma variável está contida em uma coluna só.

Por outro lado, as funções do pacote `broom` arrumam a estrutura dos resultados de funções, retornados em estruturas complexas (por exemplo listas aninhadas) de difícil manipulação. Veremos a funcionalidade do `broom` no contexto dos modelos de regressão, mas antes disso exploraremos o *encadeamento* de operações com `%>%`.

Os nomes de colunas são valores de uma variável, não variáveis em si

Um conjunto de dados com 20 observações e 3 variáveis (setor censitário, tipo de região: rural ou urbano, e total de vacinados) pode estar estruturado da seguinte maneira:

```
> set.seed(100)
> (caes_vacinados <- data.frame(setor = letters[1:10],
+                               rural = rpois(10, 40),
+                               urbano = rpois(10, 600)))
```

```
   setor rural urbano
1      a    36    595
2      b    29    629
3      c    45    585
4      d    40    590
5      e    42    629
6      f    36    608
7      g    42    572
8      h    37    618
9      i    40    606
10     j    40    618
```

Se os nomes das colunas `rural` e `urbano` são valores da variável *tipo de região*, a estrutura arrumada deve ter 20 linhas e 3 colunas.

```
   setor   tipo vacinados
1      a  rural        36
2      b  rural        29
3      c  rural        45
4      d  rural        40
5      e  rural        42
6      f  rural        36
7      g  rural        42
8      h  rural        37
9      i  rural        40
10     j  rural        40
11     a urbano       595
12     b urbano       629
13     c urbano       585
14     d urbano       590
15     e urbano       629
16     f urbano       608
17     g urbano       572
18     h urbano       618
19     i urbano       606
20     j urbano       618
```

Com a função `gather` do `tidyr` podemos transformar a primeira estrutura na segunda especificando o banco, o nome de duas colunas *chave* e *valor*, e as colunas que não são variáveis.

```
> library(tidyr)
> gather(data = caes_vacinados, key = tipo, value = vacinados, rural, urbano)
```

```
   setor   tipo vacinados
1      a  rural        36
2      b  rural        29
3      c  rural        45
4      d  rural        40
5      e  rural        42
6      f  rural        36
7      g  rural        42
8      h  rural        37
9      i  rural        40
10     j  rural        40
11     a urbano       595
12     b urbano       629
13     c urbano       585
14     d urbano       590
15     e urbano       629
16     f urbano       608
17     g urbano       572
18     h urbano       618
19     i urbano       606
20     j urbano       618
```

Pelo código anterior a coluna *chave* (`key`) recebeu o nome `tipo` e armazenou os valores `rural` e `urbano` . A coluna *valor* (`value`) recebeu o nome `vacinados` e armazenou os valores correspondentes aos valores das variáveis `setor` e `tipo` .

No lugar de especificar as colunas que não são variáveis, outra opção consiste em especificar as que são variáveis, precedidas por um sinal negativo.

```
> caes_vacinados <- gather(caes_vacinados, tipo, vacinados, -setor)
```

Reparem que os nomes das colunas não estão contidos em aspas. No capítulo *Objetos* vimos que para selecionar colunas de um data frame tínhamos que usar um vetor com as posições das colunas ou com os nomes entre aspas. O `tidyr` é mais flexível na hora de selecionar colunas. Vejamos alguns exemplos com o seguinte conjunto de dados que tem 15 observações e 5 variáveis (`setor` , `tipo` , `ano` , `vacinados`), e uma estrutura não arrumada.

```
> (caes_vacinados2 <- data.frame(setor = rep(letters[1:5], 2),
+                                tipo = rep(c('rural', 'urbano'), e = 5),
+                                ano1 = rpois(5, 600),
+                                ano2 = rpois(5, 500),
+                                ano3 = rpois(5, 550)))
```

```
   setor   tipo ano1 ano2 ano3
1      a  rural  580  539  557
2      b  rural  579  496  567
3      c  rural  605  496  518
4      d  rural  571  495  582
5      e  rural  580  541  508
6      a urbano  580  539  557
7      b urbano  579  496  567
8      c urbano  605  496  518
9      d urbano  571  495  582
10     e urbano  580  541  508
```

```
> gather(caes_vacinados2, ano, vacinados, ano1:ano3) # Colunas entre ano1 e ano3.
```

```
> gather(caes_vacinados2, ano, vacinados, ano1, ano2, ano3)
```

```
> gather(caes_vacinados2, ano, vacinados, -setor, -tipo)
```

```
> gather(caes_vacinados2, ano, vacinados, -c(setor, tipo))
```

```
> gather(caes_vacinados2, ano, vacinados, -c(setor:tipo))
```

Mais de uma variável está contida em uma coluna só

Mais de uma variável pode estar contida em uma coluna só quando cada valor dessa coluna é a combinação dos valores de mais de uma variável, ou quando os valores dessa coluna são variáveis.

No banco a seguir há 10 observações de 10 fêmeas identificados pela variável `id`. A coluna `idade_filhotes` contém o número de anos das fêmeas no momento do seu último parto e o número de filhotes nesse parto (9/3: 9 anos e 3 filhotes). Assim, o banco tem 2 colunas e 3 variáveis, sendo que cada valor da coluna `idade_filhotes` contém o valor de duas variáveis: anos das fêmeas no momento do seu último parto e o número de filhotes nesse parto.

```
> set.seed(7)
> (idades <- data.frame(id = 1:10,
+                       idade_filhotes = paste(sample(c(1:9), 10, r = T),
+                                              sample(1:12, 10, r = T),
+                                              sep = '/')))
```

```
   id idade_filhotes
1   1            9/3
2   2            4/3
3   3           2/10
4   4            1/2
5   5            3/6
6   6            8/2
7   7            4/7
8   8            9/1
9   9           2/12
10 10            5/4
```

Para arrumar a estrutura do banco podemos usar a função `separate`.

```
> (idades <-separate(idades, idade_filhotes,
+                    into = c('idade', 'filhotes'),
+                    sep = '/'))
```

```
   id idade filhotes
1   1     9        3
2   2     4        3
3   3     2       10
4   4     1        2
5   5     3        6
6   6     8        2
7   7     4        7
8   8     9        1
9   9     2       12
10 10     5        4
```

Pelo código anterior, a coluna separada foi `idade_filhotes`, o critério de separação foi `/`, o conteúdo à esquerda de `/` passou a ser a coluna `idade` e o conteúdo à direita a coluna `filhotes`. Como a coluna inicial era tipo caractere, as resultantes também. No capítulo *Operações em bancos de dados* veremos alternativas eficientes para mudar o conteúdo dos data frames (por exemplo o tipo das colunas).

```
> str(idades)
```

```
'data.frame':    10 obs. of  3 variables:
 $ id      : int  1 2 3 4 5 6 7 8 9 10
 $ idade   : chr  "9" "4" "2" "1" ...
 $ filhotes: chr  "3" "3" "10" "2" ...
```

Consideremos um outro banco com casos notificados e confirmados em 6 municípios diferentes.

```
> (casos <- data.frame(municipio = letters[1:6],
+                      status = rep(c('notificados', 'confirmados'), e = 6),
+                      total = c(617, 534, 811, 233, 184, 79)))
```

```
   municipio      status total
1          a notificados   617
2          b notificados   534
3          c notificados   811
4          d notificados   233
5          e notificados   184
6          f notificados    79
7          a confirmados   617
8          b confirmados   534
9          c confirmados   811
10         d confirmados   233
11         e confirmados   184
12         f confirmados    79
```

No banco `casos` os valores da coluna `status` são variáveis. Para arrumarmos a estrutura de `casos` precisamos que cada valor diferente passe a ser uma coluna. Com a função `spread` só precisamos especificar a coluna que tem mais de uma variável, e a coluna cujos valores serão espalhados nas colunas a serem criadas.

```
> spread(casos, status, total)
```

```
  municipio confirmados notificados
1         a         617         617
2         b         534         534
3         c         811         811
4         d         233         233
5         e         184         184
6         f          79          79
```

Encadeamento

`spread` é a função inversa de `gather`.

```
> gather(spread(casos, status, total), status, total, -municipio)
```

```
   municipio      status total
1          a confirmados   617
2          b confirmados   534
3          c confirmados   811
4          d confirmados   233
5          e confirmados   184
6          f confirmados    79
7          a notificados   617
8          b notificados   534
9          c notificados   811
10         d notificados   233
11         e notificados   184
12         f notificados    79
```

No código anterior aplicamos a função `spread` ao banco `casos` (`spread(casos, status, total)`) e o resultado foi o primeiro argumento da função `gather` .

No lugar de escrevermos um código que é executado de dentro para fora, podemos usar o operador `%>%` que aplica o resultado do que está à esquerda como primeiro argumento do que está à direita.

```
> exp(sqrt(log(10)))
```

```
[1] 4.560477
```

```
> 10 %>% log() %>% sqrt() %>% exp()
```

```
[1] 4.560477
```

```
> casos %>%
+   spread(status, total) %>%
+   gather(status, total, -municipio)
```

```
   municipio      status total
1          a confirmados   617
2          b confirmados   534
3          c confirmados   811
4          d confirmados   233
5          e confirmados   184
6          f confirmados    79
7          a notificados   617
8          b notificados   534
9          c notificados   811
10         d notificados   233
11         e notificados   184
12         f notificados    79
```

Data frame do resultado de uma função

Muitas funções retornam resultados em estruturas complexas (por exemplo listas aninhadas) de difícil manipulação. Um exemplo é a função `lm` que ajusta uma regressão linear. A teoria da regressão linear está além do escopo deste livro, mas para os exemplos a seguir, podemos pensar que dado um banco com uma variável desfecho *y* e uma variável explicativa *x*, uma regressão linear simples determina a associação linear entre *x* e *y*, e quantifica entre outras coisas a proporção da variação em *y* explicada por *x* (R^2 ajustado), e a significância estatística da associação (valor *p* para o coeficiente estimado de *x*).

Assim, dado o seguinte banco

```
> set.seed(4)
> x <- rnorm(500, 50, 10)
> banco <- data.frame(grupo = rep(1:10, e = 50),
+                     y = 1 + 0.4 * x + rnorm(500, 0, 10),
+                     x = x)
> head(banco)
```

```
  grupo         y        x
1     1  5.420216 52.16755
2     1 10.630271 44.57507
3     1  7.782182 58.91145
4     1 27.625572 55.95981
5     1 24.054606 66.35618
6     1 23.233784 56.89275
```

podemos ajustar uma regressão linear e ver o sumário do resultado.

```
> modelo <- lm(y ~ x, data = banco)
> summary(modelo)
```

```
Call:
lm(formula = y ~ x, data = banco)

Residuals:
     Min       1Q   Median       3Q      Max
-25.7194  -5.9298   0.0961   6.8075  26.7512

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.77163    2.27262   -0.34    0.734
x            0.42765    0.04488    9.53   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9.713 on 498 degrees of freedom
Multiple R-squared:  0.1542,    Adjusted R-squared:  0.1525
F-statistic: 90.81 on 1 and 498 DF,  p-value: < 2.2e-16
```

Na função `lm` o primeiro argumento é uma fórmula (`variável resposta ~ varfiável explicativa`) que usa variáveis contidas no banco especificado no argumento `data` .

A estrutura de `summary(modelo)` é uma lista aninhada

```
> str(summary(modelo))
```

```
List of 11
 $ call         : language lm(formula = y ~ x, data = banco)
 $ terms        :Classes 'terms', 'formula'  language y ~ x
  .. ..- attr(*, "variables")= language list(y, x)
  .. ..- attr(*, "factors")= int [1:2, 1] 0 1
  .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. ..$ : chr [1:2] "y" "x"
  .. .. .. ..$ : chr "x"
  .. ..- attr(*, "term.labels")= chr "x"
  .. ..- attr(*, "order")= int 1
  .. ..- attr(*, "intercept")= int 1
  .. ..- attr(*, "response")= int 1
  .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
  .. ..- attr(*, "predvars")= language list(y, x)
  .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
  .. .. ..- attr(*, "names")= chr [1:2] "y" "x"
 $ residuals    : Named num [1:500] -16.12 -7.66 -16.64 4.47 -3.55 ...
  ..- attr(*, "names")= chr [1:500] "1" "2" "3" "4" ...
 $ coefficients : num [1:2, 1:4] -0.7716 0.4277 2.2726 0.0449 -0.3395 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:2] "(Intercept)" "x"
  .. ..$ : chr [1:4] "Estimate" "Std. Error" "t value" "Pr(>|t|)"
 $ aliased      : Named logi [1:2] FALSE FALSE
  ..- attr(*, "names")= chr [1:2] "(Intercept)" "x"
 $ sigma        : num 9.71
 $ df           : int [1:3] 2 498 2
 $ r.squared    : num 0.154
 $ adj.r.squared: num 0.153
 $ fstatistic   : Named num [1:3] 90.8 1 498
  ..- attr(*, "names")= chr [1:3] "value" "numdf" "dendf"
 $ cov.unscaled : num [1:2, 1:2] 5.47e-02 -1.06e-03 -1.06e-03 2.13e-05
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:2] "(Intercept)" "x"
  .. ..$ : chr [1:2] "(Intercept)" "x"
 - attr(*, "class")= chr "summary.lm"
```

e nós já sabemos como indexar os elementos de essas estruturas. Temos como extrair o R^2 ajustado e o valor p para *x* com o seguinte código:

```
> summary(modelo)$coefficients[2, 4]
```

```
[1] 6.918045e-20
```

```
> summary(modelo)$adj.r.squared
```

```
[1] 0.1525333
```

Porém é muito mais fácil trabalhar com os resultados contidos em data frames arrumados, especialmente quando esses resultados são os argumentos para funções a serem aplicada subsequentemente. O pacote `broom` tem funções que a partir de resultados em uma estrutura complexa, produzem data frames arrumados do conteúdo principal (`tidy`), das medidas de sumarização (`glance`), e da combinação dos dados com o resultado da função aplicada aos mesmos (`augment`).

Assim, com a função `glance` é mais simples obter os dois valores anteriores.

```
> library(broom)
> glance(summary(modelo))
```

```
  r.squared adj.r.squared    sigma statistic      p.value df
1 0.1542317     0.1525333 9.712656  90.81373 6.918045e-20  2
```

```
> glance(summary(modelo))[ , c(2, 5)]
```

```
  adj.r.squared      p.value
1     0.1525333 6.918045e-20
```

A utilidade da `tidy` e `augment` aplicadas em `modelo` pode ser difícil de enxergar para que não tem familiaridade com as regressões, mas a título de exemplo seguem os códigos.

```
> tidy(summary(modelo))
```

```
         term   estimate  std.error  statistic      p.value
1 (Intercept) -0.7716281 2.27261907 -0.3395325 7.343517e-01
2           x  0.4276520 0.04487607  9.5296237 6.918045e-20
```

```
> head(augment(modelo))
```

```
          y        x  .fitted   .se.fit      .resid        .hat
1  5.420216 52.16755 21.53793 0.4481616 -16.1177145 0.002129086
2 10.630271 44.57507 18.29099 0.4916720  -7.6607229 0.002562565
3  7.782182 58.91145 24.42197 0.5993604 -16.6397901 0.003808027
4 27.625572 55.95981 23.15970 0.5170784   4.4658748 0.002834241
5 24.054606 66.35618 27.60573 0.8641769  -3.5511216 0.007916428
6 23.233784 56.89275 23.55867 0.5409372  -0.3248908 0.003101828
    .sigma      .cooksd  .std.resid
1 9.695447 2.944051e-03 -1.66122423
2 9.716332 8.011928e-04 -0.78974867
3 9.693620 5.631234e-03 -1.71647825
4 9.720353 3.013069e-04  0.46045251
5 9.721107 5.375982e-04 -0.36707380
6 9.722412 1.746164e-06 -0.03350225
```

Operações em bancos de dados

Em capítulos anteriores aplicamos funções em data frames que podem ser entendidas como operações em bancos de dados. Neste capítulo daremos continuidade às operações de bancos de dados sob uma abordagem mais eficiente, consistente e sofisticada implementada no pacote `dplyr`, que além de oferecer funções para operar com um ou dois data frames, suporta operações encadeadas e uma abordagem conhecida como *partição, aplicação, e combinação*.

Várias das operações que veremos são realizáveis com alternativas que prescinde do `dplyr`. Entretanto, com este último a execução usualmente é mais rápida (relevante com grandes bancos de dados) e consistente.

Operações em um data frame

Seleção de linhas

A função `filter` seleciona linhas do banco que recebe como primeiro argumento, de acordo com uma ou mais condições especificadas nos outros argumentos.

```
> set.seed(3)
> (domicilios <- data.frame(municipio = rep(c('a1', 'b', 'c'), e = 10),
+                           id = 1:30,
+                           caes = rpois(30, .95),
+                           gatos = rpois(30, .6),
+                           stringsAsFactors = F))
```

```
   municipio id caes gatos
1         a1  1    0     0
2         a1  2    2     0
3         a1  3    0     0
4         a1  4    0     0
5         a1  5    1     0
6         a1  6    1     0
7         a1  7    0     2
8         a1  8    0     0
9         a1  9    1     1
10        a1 10    1     0
11         b 11    1     0
12         b 12    1     1
13         b 13    1     0
14         b 14    1     1
15         b 15    2     0
16         b 16    2     0
17         b 17    0     0
18         b 18    1     0
19         b 19    2     0
20         b 20    0     1
21         c 21    0     0
22         c 22    0     0
23         c 23    0     1
24         c 24    0     3
25         c 25    0     1
26         c 26    2     2
27         c 27    1     0
28         c 28    2     0
29         c 29    1     0
30         c 30    2     0
```

Reparem que com as funções do `dplyr` os nomes das colunas não estão contidos em aspas.

```
> library(dplyr)
> filter(domicilios, caes > 1)
```

```
  municipio id caes gatos
1        a1  2    2     0
2         b 15    2     0
3         b 16    2     0
4         b 19    2     0
5         c 26    2     2
6         c 28    2     0
7         c 30    2     0
```

```
> filter(domicilios, caes == 1, gatos == 1)
```

```
  municipio id caes gatos
1        a1  9    1     1
2         b 12    1     1
3         b 14    1     1
```

Ordenação das linhas

A função `arrange` ordena o banco com base em uma ou mais colunas. Ao aplicarmos a função `desc` nas colunas, a ordenação e descendente.

```
> arrange(domicilios, caes)
```

```
   municipio id caes gatos
1         a1  1    0     0
2         a1  3    0     0
3         a1  4    0     0
4         a1  7    0     2
5         a1  8    0     0
6          b 17    0     0
7          b 20    0     1
8          c 21    0     0
9          c 22    0     0
10         c 23    0     1
11         c 24    0     3
12         c 25    0     1
13        a1  5    1     0
14        a1  6    1     0
15        a1  9    1     1
16        a1 10    1     0
17         b 11    1     0
18         b 12    1     1
19         b 13    1     0
20         b 14    1     1
21         b 18    1     0
22         c 27    1     0
23         c 29    1     0
24        a1  2    2     0
25         b 15    2     0
26         b 16    2     0
27         b 19    2     0
28         c 26    2     2
29         c 28    2     0
30         c 30    2     0
```

```
> arrange(domicilios, desc(caes))
```

```
   municipio id caes gatos
1         a1  2    2     0
2          b 15    2     0
3          b 16    2     0
4          b 19    2     0
5          c 26    2     2
6          c 28    2     0
7          c 30    2     0
8         a1  5    1     0
9         a1  6    1     0
10        a1  9    1     1
11        a1 10    1     0
12         b 11    1     0
13         b 12    1     1
14         b 13    1     0
15         b 14    1     1
16         b 18    1     0
17         c 27    1     0
18         c 29    1     0
19        a1  1    0     0
20        a1  3    0     0
21        a1  4    0     0
22        a1  7    0     2
23        a1  8    0     0
24         b 17    0     0
25         b 20    0     1
26         c 21    0     0
27         c 22    0     0
28         c 23    0     1
29         c 24    0     3
30         c 25    0     1
```

```
> arrange(domicilios, caes, gatos)
```

```
   municipio id caes gatos
1         a1  1    0     0
2         a1  3    0     0
3         a1  4    0     0
4         a1  8    0     0
5          b 17    0     0
6          c 21    0     0
7          c 22    0     0
8          b 20    0     1
9          c 23    0     1
10         c 25    0     1
11        a1  7    0     2
12         c 24    0     3
13        a1  5    1     0
14        a1  6    1     0
15        a1 10    1     0
16         b 11    1     0
17         b 13    1     0
18         b 18    1     0
19         c 27    1     0
20         c 29    1     0
21        a1  9    1     1
22         b 12    1     1
23         b 14    1     1
24        a1  2    2     0
25         b 15    2     0
26         b 16    2     0
27         b 19    2     0
28         c 28    2     0
29         c 30    2     0
30         c 26    2     2
```

```
> arrange(domicilios, desc(caes), gatos)
```

```
   municipio id caes gatos
1         a1  2    2     0
2          b 15    2     0
3          b 16    2     0
4          b 19    2     0
5          c 28    2     0
6          c 30    2     0
7          c 26    2     2
8         a1  5    1     0
9         a1  6    1     0
10        a1 10    1     0
11         b 11    1     0
12         b 13    1     0
13         b 18    1     0
14         c 27    1     0
15         c 29    1     0
16        a1  9    1     1
17         b 12    1     1
18         b 14    1     1
19        a1  1    0     0
20        a1  3    0     0
21        a1  4    0     0
22        a1  8    0     0
23         b 17    0     0
24         c 21    0     0
25         c 22    0     0
26         b 20    0     1
27         c 23    0     1
28         c 25    0     1
29        a1  7    0     2
30         c 24    0     3
```

Seleção de colunas

No capítulo *Objetos* vimos que para selecionar colunas de um data frame tínhamos que usar um vetor com as posições das colunas ou com os nomes entre aspas. O `dplyr` é mais flexível na hora de selecionar colunas.

```
> head(select(domicilios, 1:3))
```

```
  municipio id caes
1        a1  1    0
2        a1  2    2
3        a1  3    0
4        a1  4    0
5        a1  5    1
6        a1  6    1
```

Nomes sem aspas.

```
> head(select(domicilios, c(municipio, id, caes)))
```

```
  municipio id caes
1        a1  1    0
2        a1  2    2
3        a1  3    0
4        a1  4    0
5        a1  5    1
6        a1  6    1
```

Podemos colocar as colunas como argumentos diferentes

```
> head(select(domicilios, municipio, id, gatos))
```

```
  municipio id gatos
1        a1  1     0
2        a1  2     0
3        a1  3     0
4        a1  4     0
5        a1  5     0
6        a1  6     0
```

```
> head(select(domicilios, 1, 2, 3))
```

```
  municipio id caes
1        a1  1    0
2        a1  2    2
3        a1  3    0
4        a1  4    0
5        a1  5    1
6        a1  6    1
```

e criar sequências de colunas não apenas com as posições (`1:3`), mas também com os nomes (`municipio:caes`).

```
> head(select(domicilios, municipio:caes))
```

```
  municipio id caes
1        a1  1    0
2        a1  2    2
3        a1  3    0
4        a1  4    0
5        a1  5    1
6        a1  6    1
```

O sinal `-` omite colunas.

```
> head(select(domicilios, -municipio))
```

```
  id caes gatos
1  1    0     0
2  2    2     0
3  3    0     0
4  4    0     0
5  5    1     0
6  6    1     0
```

```
> head(select(domicilios, -municipio, -gatos))
```

```
  id caes
1  1    0
2  2    2
3  3    0
4  4    0
5  5    1
6  6    1
```

```
> head(select(domicilios, -c(municipio:caes)))
```

```
  gatos
1     0
2     0
3     0
4     0
5     0
6     0
```

Para selecionar com base em um vetor de caracteres, devemos usar a função `one_of`.

```
> head(select(domicilios, one_of(c('municipio', 'id', 'caes'))))
```

```
  municipio id caes
1        a1  1    0
2        a1  2    2
3        a1  3    0
4        a1  4    0
5        a1  5    1
6        a1  6    1
```

Há um conjunto de funções que servem para selecionar colunas cujo nome começa, termina ou contém um dado caractere.

```
> head(select(domicilios, starts_with('ca')))
```

```
  caes
1    0
2    2
3    0
4    0
5    1
6    1
```

```
> head(select(domicilios, ends_with('s')))
```

```
  caes gatos
1    0     0
2    2     0
3    0     0
4    0     0
5    1     0
6    1     0
```

```
> head(select(domicilios, contains('a')))
```

```
  caes gatos
1    0     0
2    2     0
3    0     0
4    0     0
5    1     0
6    1     0
```

Quando os nomes das colunas têm um prefixo seguido por um número, podemos especificar o prefixo e uma sequência de números.

```
> (banco_numerico <- as.data.frame(matrix(runif(50, 10, 20), ncol = 5)))
```

```
         V1       V2       V3       V4       V5
1  18.16106 17.59755 17.84215 10.11744 17.66338
2  10.57613 17.60820 16.54355 12.67403 16.82132
3  18.02829 19.03261 13.78104 14.35572 12.09131
4  11.04388 19.66283 10.08567 18.29467 17.11944
5  17.66606 15.15257 19.55330 18.70944 16.05298
6  13.04811 15.49481 18.38616 12.51069 13.40559
7  17.69287 11.63720 12.13425 13.24358 10.41170
8  15.40656 11.64597 14.94714 13.06238 14.01753
9  13.62371 17.86333 16.36244 11.84282 10.79060
10 10.92557 17.51113 19.21091 16.79977 13.12553
```

```
> head(select(banco_numerico, num_range(prefix = 'V', 3:5)))
```

```
        V3       V4       V5
1 17.84215 10.11744 17.66338
2 16.54355 12.67403 16.82132
3 13.78104 14.35572 12.09131
4 10.08567 18.29467 17.11944
5 19.55330 18.70944 16.05298
6 18.38616 12.51069 13.40559
```

Com `select_if` selecionamos só as colunas que satisfazem uma dada condição.

```
> head(select_if(domicilios, is.character))
```

```
  municipio
1        a1
2        a1
3        a1
4        a1
5        a1
6        a1
```

Renomeação

A função `rename` renomeia usando a sintaxe `novo_nome = nome_atual`.

```
> head(rename(domicilios, city = municipio, dogs = caes, cats = gatos))
```

```
  city id dogs cats
1   a1  1    0    0
2   a1  2    2    0
3   a1  3    0    0
4   a1  4    0    0
5   a1  5    1    0
6   a1  6    1    0
```

Remoção de linhas duplicadas

Modifiquemos o banco `domicilios` para que fique com três repetições da primeira linha e duas da terceira.

```
> head(domicilios <- rbind(domicilios[c(1, 1, 3), ], domicilios))
```

```
    municipio id caes gatos
1          a1  1    0     0
1.1        a1  1    0     0
3          a1  3    0     0
110        a1  1    0     0
2          a1  2    2     0
31         a1  3    0     0
```

A função `distinct` remove as linhas duplicadas.

```
> distinct(domicilios)
```

```
   municipio id caes gatos
1         a1  1    0     0
2         a1  3    0     0
3         a1  2    2     0
4         a1  4    0     0
5         a1  5    1     0
6         a1  6    1     0
7         a1  7    0     2
8         a1  8    0     0
9         a1  9    1     1
10        a1 10    1     0
11         b 11    1     0
12         b 12    1     1
13         b 13    1     0
14         b 14    1     1
15         b 15    2     0
16         b 16    2     0
17         b 17    0     0
18         b 18    1     0
19         b 19    2     0
20         b 20    0     1
21         c 21    0     0
22         c 22    0     0
23         c 23    0     1
24         c 24    0     3
25         c 25    0     1
26         c 26    2     2
27         c 27    1     0
28         c 28    2     0
29         c 29    1     0
30         c 30    2     0
```

Se especificamos uma ou mais colunas, as duplicatas são consideradas só nessas colunas.

```
> distinct(domicilios, municipio)
```

```
  municipio
1        a1
2         b
3         c
```

```
> distinct(domicilios, municipio, caes)
```

```
  municipio caes
1        a1    0
2        a1    2
3        a1    1
4         b    1
5         b    2
6         b    0
7         c    0
8         c    2
9         c    1
```

Deixemos o banco como estava.

```
> domicilios <- distinct(domicilios)
```

Transformação e adição de colunas

Com `mutate` podemos transformar e adicionar colunas definido ou redefinindo as colunas de interese.

```
> mutate(domicilios,
+        municipio = rep(1:3, e = 10),
+        id = id + 100,
+        animais = caes + gatos)
```

```
   municipio  id caes gatos animais
1          1 101    0     0       0
2          1 103    0     0       0
3          1 102    2     0       2
4          1 104    0     0       0
5          1 105    1     0       1
6          1 106    1     0       1
7          1 107    0     2       2
8          1 108    0     0       0
9          1 109    1     1       2
10         1 110    1     0       1
11         2 111    1     0       1
12         2 112    1     1       2
13         2 113    1     0       1
14         2 114    1     1       2
15         2 115    2     0       2
16         2 116    2     0       2
17         2 117    0     0       0
18         2 118    1     0       1
19         2 119    2     0       2
20         2 120    0     1       1
21         3 121    0     0       0
22         3 122    0     0       0
23         3 123    0     1       1
24         3 124    0     3       3
25         3 125    0     1       1
26         3 126    2     2       4
27         3 127    1     0       1
28         3 128    2     0       2
29         3 129    1     0       1
30         3 130    2     0       2
```

`transmutate` segue a mesma lógica de `mutate`, porém, só as colunas transformadas ou adicionadas são retornadas.

```
> transmute(domicilios,
+           municipio = rep(1:3, e = 10),
+           id = id + 100,
+           animais = caes + gatos)
```

```
   municipio  id animais
1          1 101       0
2          1 103       0
3          1 102       2
4          1 104       0
5          1 105       1
6          1 106       1
7          1 107       2
8          1 108       0
9          1 109       2
10         1 110       1
11         2 111       1
12         2 112       2
13         2 113       1
14         2 114       2
15         2 115       2
16         2 116       2
17         2 117       0
18         2 118       1
19         2 119       2
20         2 120       1
21         3 121       0
22         3 122       0
23         3 123       1
24         3 124       3
25         3 125       1
26         3 126       4
27         3 127       1
28         3 128       2
29         3 129       1
30         3 130       2
```

`mutate_all` aplica uma função a todas as colunas,

```
> mutate_all(banco_numerico, log)
```

```
         V1       V2       V3       V4       V5
1  2.899280 2.867760 2.881563 2.314261 2.871493
2  2.358600 2.868365 2.805996 2.539555 2.822647
3  2.891942 2.946154 2.623294 2.664149 2.492487
4  2.401876 2.978730 2.311116 2.906610 2.840215
5  2.871645 2.718170 2.973144 2.929028 2.775895
6  2.568643 2.740505 2.911598 2.526583 2.595672
7  2.873162 2.454207 2.496032 2.583513 2.342930
8  2.734793 2.454960 2.704520 2.569736 2.640308
9  2.611811 2.882750 2.794989 2.471722 2.378675
10 2.391106 2.862837 2.955479 2.821365 2.574559
```

enquanto `mutate_at` aplica a função só às funções especificadas. Porém, `mutate_at` só aceita a especificação de sequência de nomes com a função `vars` (`vars(V1:V3)` no lugar de `V1:V3`).

```
> mutate_at(banco_numerico, 1:3, mean)
```

```
         V1       V2       V3       V4       V5
1  14.61722 16.32062 15.88466 10.11744 17.66338
2  14.61722 16.32062 15.88466 12.67403 16.82132
3  14.61722 16.32062 15.88466 14.35572 12.09131
4  14.61722 16.32062 15.88466 18.29467 17.11944
5  14.61722 16.32062 15.88466 18.70944 16.05298
6  14.61722 16.32062 15.88466 12.51069 13.40559
7  14.61722 16.32062 15.88466 13.24358 10.41170
8  14.61722 16.32062 15.88466 13.06238 14.01753
9  14.61722 16.32062 15.88466 11.84282 10.79060
10 14.61722 16.32062 15.88466 16.79977 13.12553
```

```
> mutate_at(banco_numerico, c('V1', 'V2', 'V3'), mean)
```

```
         V1       V2       V3       V4       V5
1  14.61722 16.32062 15.88466 10.11744 17.66338
2  14.61722 16.32062 15.88466 12.67403 16.82132
3  14.61722 16.32062 15.88466 14.35572 12.09131
4  14.61722 16.32062 15.88466 18.29467 17.11944
5  14.61722 16.32062 15.88466 18.70944 16.05298
6  14.61722 16.32062 15.88466 12.51069 13.40559
7  14.61722 16.32062 15.88466 13.24358 10.41170
8  14.61722 16.32062 15.88466 13.06238 14.01753
9  14.61722 16.32062 15.88466 11.84282 10.79060
10 14.61722 16.32062 15.88466 16.79977 13.12553
```

```
> mutate_at(banco_numerico, vars(V1:V3), mean)
```

```
          V1       V2       V3       V4       V5
1  14.61722 16.32062 15.88466 10.11744 17.66338
2  14.61722 16.32062 15.88466 12.67403 16.82132
3  14.61722 16.32062 15.88466 14.35572 12.09131
4  14.61722 16.32062 15.88466 18.29467 17.11944
5  14.61722 16.32062 15.88466 18.70944 16.05298
6  14.61722 16.32062 15.88466 12.51069 13.40559
7  14.61722 16.32062 15.88466 13.24358 10.41170
8  14.61722 16.32062 15.88466 13.06238 14.01753
9  14.61722 16.32062 15.88466 11.84282 10.79060
10 14.61722 16.32062 15.88466 16.79977 13.12553
```

Com `mutate_if` transformamos só as colunas que satisfazem uma dada condição. A condição deve ser especificada no segundo argumento e a função a ser aplicada no terceiro.

```
> mutate_if(domicilios, is.character, toupper)
```

```
   municipio id caes gatos
1         A1  1    0     0
2         A1  3    0     0
3         A1  2    2     0
4         A1  4    0     0
5         A1  5    1     0
6         A1  6    1     0
7         A1  7    0     2
8         A1  8    0     0
9         A1  9    1     1
10        A1 10    1     0
11         B 11    1     0
12         B 12    1     1
13         B 13    1     0
14         B 14    1     1
15         B 15    2     0
16         B 16    2     0
17         B 17    0     0
18         B 18    1     0
19         B 19    2     0
20         B 20    0     1
21         C 21    0     0
22         C 22    0     0
23         C 23    0     1
24         C 24    0     3
25         C 25    0     1
26         C 26    2     2
27         C 27    1     0
28         C 28    2     0
29         C 29    1     0
30         C 30    2     0
```

Sumarização

Podemos sumarizar uma ou mais colunas com `summarise` e suas variações. A sumarização é a aplicação de funções que a partir de uma coluna retornam um único valor.

```
> summarise(domicilios, total_caes = sum(caes), media_gatos = mean(gatos))
```

```
  total_caes media_gatos
1         25   0.4333333
```

```
> summarise_all(banco_numerico, sum)
```

```
        V1       V2       V3       V4       V5
1 146.1722 163.2062 158.8466 141.6105 141.4994
```

```
> summarise_at(banco_numerico, vars(V1:V3), mean)
```

```
        V1       V2       V3
1 14.61722 16.32062 15.88466
```

```
> summarise_if(domicilios, is.numeric, max)
```

```
  id caes gatos
1 30    2     3
```

Amostragem de colunas

`sample_n` e `sample_frac` amostram um número ou fração de linhas respectivamente.

```
> set.seed(7)
> sample_n(domicilios, 5)
```

```
   municipio id caes gatos
30         c 30    2     0
12         b 12    1     1
4         a1  4    0     0
2         a1  3    0     0
7         a1  7    0     2
```

```
> set.seed(7)
> sample_n(domicilios, 5, replace = T)
```

```
   municipio id caes gatos
30         c 30    2     0
12         b 12    1     1
4         a1  4    0     0
3         a1  2    2     0
8         a1  8    0     0
```

```
> set.seed(7)
> sample_frac(domicilios, .1)
```

```
   municipio id caes gatos
30         c 30    2     0
12         b 12    1     1
4         a1  4    0     0
```

Operações com dois data frames

Criemos mais um banco para explorar as funções que operam em dois data frames.

```
> set.seed(5)
> (municipios <- data.frame(municipio = letters[1:10],
+                           idh = runif(10, .7, .8),
+                           stringsAsFactors = F))
```

```
   municipio       idh
1          a 0.7200214
2          b 0.7685219
3          c 0.7916876
4          d 0.7284399
5          e 0.7104650
6          f 0.7701057
7          g 0.7527960
8          h 0.7807935
9          i 0.7956500
10         j 0.7110453
```

Junção interna

Com base em uma ou mais colunas (no caso `municipio`), `inner_join` retorna todas as linhas de `domicílios` que também estão em `municipios`, e todas as colunas de ambos bancos.

```
> inner_join(domicilios, municipios, by = 'municipio')
```

```
   municipio id caes gatos       idh
1          b 11    1     0 0.7685219
2          b 12    1     1 0.7685219
3          b 13    1     0 0.7685219
4          b 14    1     1 0.7685219
5          b 15    2     0 0.7685219
6          b 16    2     0 0.7685219
7          b 17    0     0 0.7685219
8          b 18    1     0 0.7685219
9          b 19    2     0 0.7685219
10         b 20    0     1 0.7685219
11         c 21    0     0 0.7916876
12         c 22    0     0 0.7916876
13         c 23    0     1 0.7916876
14         c 24    0     3 0.7916876
15         c 25    0     1 0.7916876
16         c 26    2     2 0.7916876
17         c 27    1     0 0.7916876
18         c 28    2     0 0.7916876
19         c 29    1     0 0.7916876
20         c 30    2     0 0.7916876
```

Junção esquerda

Com base em uma ou mais colunas (no caso `municipio`), `left_join` retorna todas as linhas de `domicilios`, e todas as colunas de `domicilios` e `municipios`. As linhas de `domicilios` que não estão em `municipios` assumem o valor `NA` nas colunas de `municipios`.

```
> left_join(domicilios, municipios, by = 'municipio')
```

```
   municipio id caes gatos       idh
1         a1  1    0     0        NA
2         a1  3    0     0        NA
3         a1  2    2     0        NA
4         a1  4    0     0        NA
5         a1  5    1     0        NA
6         a1  6    1     0        NA
7         a1  7    0     2        NA
8         a1  8    0     0        NA
9         a1  9    1     1        NA
10        a1 10    1     0        NA
11         b 11    1     0 0.7685219
12         b 12    1     1 0.7685219
13         b 13    1     0 0.7685219
14         b 14    1     1 0.7685219
15         b 15    2     0 0.7685219
16         b 16    2     0 0.7685219
17         b 17    0     0 0.7685219
18         b 18    1     0 0.7685219
19         b 19    2     0 0.7685219
20         b 20    0     1 0.7685219
21         c 21    0     0 0.7916876
22         c 22    0     0 0.7916876
23         c 23    0     1 0.7916876
24         c 24    0     3 0.7916876
25         c 25    0     1 0.7916876
26         c 26    2     2 0.7916876
27         c 27    1     0 0.7916876
28         c 28    2     0 0.7916876
29         c 29    1     0 0.7916876
30         c 30    2     0 0.7916876
```

Junção direita

Com base em uma ou mais colunas (no caso `municipio`), `right_join` retorna todas as linhas de `municipios`, e todas as colunas de `domicilios` e `municipios`. As linhas de `municipios` que não estão em `domicilios` assumem o valor `NA` nas colunas de `domicilios`.

```
> right_join(domicilios, municipios, by = 'municipio')
```

```
   municipio id caes gatos       idh
1          a NA   NA    NA 0.7200214
2          b 11    1     0 0.7685219
3          b 12    1     1 0.7685219
4          b 13    1     0 0.7685219
5          b 14    1     1 0.7685219
6          b 15    2     0 0.7685219
7          b 16    2     0 0.7685219
8          b 17    0     0 0.7685219
9          b 18    1     0 0.7685219
10         b 19    2     0 0.7685219
11         b 20    0     1 0.7685219
12         c 21    0     0 0.7916876
13         c 22    0     0 0.7916876
14         c 23    0     1 0.7916876
15         c 24    0     3 0.7916876
16         c 25    0     1 0.7916876
17         c 26    2     2 0.7916876
18         c 27    1     0 0.7916876
19         c 28    2     0 0.7916876
20         c 29    1     0 0.7916876
21         c 30    2     0 0.7916876
22         d NA   NA    NA 0.7284399
23         e NA   NA    NA 0.7104650
24         f NA   NA    NA 0.7701057
25         g NA   NA    NA 0.7527960
26         h NA   NA    NA 0.7807935
27         i NA   NA    NA 0.7956500
28         j NA   NA    NA 0.7110453
```

Junção completa

Com base em uma ou mais colunas (no caso `municipio`), `full_join` retorna todas as linhas e as colunas de `domicilios` e `municipios`. As linhas de `domicilios` que não estão em `municipios` assumem o valor `NA` nas colunas de `municipios` (e vice versa).

```
> full_join(domicilios, municipios, by = 'municipio')
```

```
   municipio id caes gatos       idh
1         a1  1    0     0        NA
2         a1  3    0     0        NA
3         a1  2    2     0        NA
4         a1  4    0     0        NA
5         a1  5    1     0        NA
6         a1  6    1     0        NA
7         a1  7    0     2        NA
8         a1  8    0     0        NA
9         a1  9    1     1        NA
10        a1 10    1     0        NA
11         b 11    1     0 0.7685219
12         b 12    1     1 0.7685219
13         b 13    1     0 0.7685219
14         b 14    1     1 0.7685219
15         b 15    2     0 0.7685219
16         b 16    2     0 0.7685219
17         b 17    0     0 0.7685219
18         b 18    1     0 0.7685219
19         b 19    2     0 0.7685219
20         b 20    0     1 0.7685219
21         c 21    0     0 0.7916876
22         c 22    0     0 0.7916876
23         c 23    0     1 0.7916876
24         c 24    0     3 0.7916876
25         c 25    0     1 0.7916876
26         c 26    2     2 0.7916876
27         c 27    1     0 0.7916876
28         c 28    2     0 0.7916876
29         c 29    1     0 0.7916876
30         c 30    2     0 0.7916876
31         a NA   NA    NA 0.7200214
32         d NA   NA    NA 0.7284399
33         e NA   NA    NA 0.7104650
34         f NA   NA    NA 0.7701057
35         g NA   NA    NA 0.7527960
36         h NA   NA    NA 0.7807935
37         i NA   NA    NA 0.7956500
38         j NA   NA    NA 0.7110453
```

Anti junção

Com base em uma ou mais colunas (no caso `municipio`), `anti_join` retorna todas as linhas de `domicilios` que não estão em `municipios`, e todas as colunas de `domicilios`.

```
> anti_join(domicilios, municipios, by = 'municipio')
```

```
   municipio id caes gatos
1         a1  1    0     0
2         a1  3    0     0
3         a1  2    2     0
4         a1  4    0     0
5         a1  5    1     0
6         a1  6    1     0
7         a1  7    0     2
8         a1  8    0     0
9         a1  9    1     1
10        a1 10    1     0
```

Partição, aplicação e combinação

Na abordagem partição, aplicação e combinação consiste na partição de um banco de dados de acordo com os valores de uma ou mais colunas, a aplicação de operações a cada uma das partições de forma independente, e na combinação dos resultados das operações em um novo banco.

Por exemplo, podemos particionar o banco `domicilios` com base na coluna `municipio` que por ter três valores únicos (`a1` , `b` , e `c`), resultará em três partições (`a1` , `b` , e `c`). Em cada partição podemos calcular a média de cães e o total de gatos, e posteriormente criar um banco da média de cães e do total de gatos por município. A partição é mediada pela função `group_by` , enquanto os cálculos da média e do total são operações de sumarização (`summarise` em cada um dos grupos).

```
> domicilios %>%
+   group_by(municipio) %>%
+   summarise(media_caes = mean(caes), total_gatos = sum(gatos))
```

```
# A tibble: 3 × 3
  municipio media_caes total_gatos
      <chr>      <dbl>       <int>
1        a1        0.6           3
2         b        1.1           3
3         c        0.8           7
```

Quando aplicamos a função `group_by` , a maioria das funções subsequentes são aplicadas dentro de cada grupo. Portanto, se particionamos, depois amostramos e finalmente sumarizamos, o banco resultante terá tantas linhas como grupos, como no caso anterior.

```
> domicilios %>%
+   group_by(municipio) %>%
+   sample_n(5) %>%
+   summarise(media_caes = mean(caes), media_gatos = mean(gatos))
```

```
# A tibble: 3 × 3
  municipio media_caes media_gatos
      <chr>      <dbl>       <dbl>
1        a1        0.8         0.0
2         b        1.4         0.2
3         c        0.8         0.0
```

Em ocasiões é necessário aplicar funções separadamente nas partições, para depois aplicar funções no total de observações. Nesses casos devemos desagrupar o banco com `ungroup` antes de aplicar as funções no total das observações.

```
> domicilios %>%
+   group_by(municipio) %>%
+   sample_n(5) %>%
+   ungroup() %>%
+   summarise(media_caes = mean(caes), media_gatos = mean(gatos))
```

```
# A tibble: 1 × 2
  media_caes media_gatos
       <dbl>       <dbl>
1  0.8666667   0.2666667
```

A função `group_by` atribui novas *classes* ao objeto resultante e uma das consequências disso é que o padrão de apresentação do data frame resultante é diferente: o tipo de cada coluna aparece abaixo do nome (`<chr>` : caractere, `<int>` : inteiro, etc.) e só as 10 primeiras linhas são mostradas. Para obtermos a apresentação padrão dos data frames, devemos coercionar para essa estrutura.

```
> (doms <- domicilios %>%
+   group_by(municipio) %>%
+   mutate(prop_caes = caes / sum(caes)))
```

```
Source: local data frame [30 x 5]
Groups: municipio [3]

   municipio    id  caes gatos prop_caes
       <chr> <int> <int> <int>     <dbl>
1         a1     1     0     0 0.0000000
2         a1     3     0     0 0.0000000
3         a1     2     2     0 0.3333333
4         a1     4     0     0 0.0000000
5         a1     5     1     0 0.1666667
6         a1     6     1     0 0.1666667
7         a1     7     0     2 0.0000000
8         a1     8     0     0 0.0000000
9         a1     9     1     1 0.1666667
10        a1    10     1     0 0.1666667
# ... with 20 more rows
```

```
> as.data.frame(doms)
```

```
   municipio id caes gatos  prop_caes
1         a1  1    0     0 0.00000000
2         a1  3    0     0 0.00000000
3         a1  2    2     0 0.33333333
4         a1  4    0     0 0.00000000
5         a1  5    1     0 0.16666667
6         a1  6    1     0 0.16666667
7         a1  7    0     2 0.00000000
8         a1  8    0     0 0.00000000
9         a1  9    1     1 0.16666667
10        a1 10    1     0 0.16666667
11         b 11    1     0 0.09090909
12         b 12    1     1 0.09090909
13         b 13    1     0 0.09090909
14         b 14    1     1 0.09090909
15         b 15    2     0 0.18181818
16         b 16    2     0 0.18181818
17         b 17    0     0 0.00000000
18         b 18    1     0 0.09090909
19         b 19    2     0 0.18181818
20         b 20    0     1 0.00000000
21         c 21    0     0 0.00000000
22         c 22    0     0 0.00000000
23         c 23    0     1 0.00000000
24         c 24    0     3 0.00000000
25         c 25    0     1 0.00000000
26         c 26    2     2 0.25000000
27         c 27    1     0 0.12500000
28         c 28    2     0 0.25000000
29         c 29    1     0 0.12500000
30         c 30    2     0 0.25000000
```

Outras operações

O banco `reg` tem 500 observações com as variáveis `y` e `x`. Cada observação pertence a um de dez grupos e isso é indicado pela variável `grupo`.

```
> set.seed(4)
> x <- rnorm(500, 50, 10)
> reg <- data.frame(grupo = rep(1:10, e = 50),
+                   y = 1 + 0.4 * x + rnorm(500, 0, 10),
+                   x = x)
```

Se queremos ajustar uma regressão linear dentro de cada grupo, as funções do `dplyr` vistas até o momento não são suficientes. Entretanto, a função `do` permite aplicar operações arbitrárias e retorna um data frame. Por exemplo, podemos aplicar a função `head` dentro de cada grupo.

```
> reg %>%
+   group_by(grupo) %>%
+   do(head(.))
```

```
Source: local data frame [60 x 3]
Groups: grupo [10]

   grupo         y        x
   <int>     <dbl>    <dbl>
1      1  5.420216 52.16755
2      1 10.630271 44.57507
3      1  7.782182 58.91145
4      1 27.625572 55.95981
5      1 24.054606 66.35618
6      1 23.233784 56.89275
7      2  9.477180 43.36257
8      2 24.875976 43.76274
9      2 35.672370 49.20368
10     2  7.730713 54.35625
# ... with 50 more rows
```

Reparem que no lugar de `do(head)` a sintaxe é `do(head(.))`. `.` é um *placeholder* que representa cada um dos grupos. Dentro de `do` podemos aplicar várias funções se as mesmas estão contidas em chaves. Por exemplo, podemos ajustar uma regressão linear dentro de cada grupo como `mod = lm(y ~ x, data = .)` e extrair as medidas de sumarização do modelo com `glance(mod)`.

```
> library(broom)
> reg %>%
+   group_by(grupo) %>%
+   do({mod = lm(y ~ x, data = .) %>%
+       glance(mod)}) %>%
+   select(adj.r.squared, p.value)
```

```
Adding missing grouping variables: `grupo`
```

```
Source: local data frame [10 x 3]
Groups: grupo [10]

   grupo adj.r.squared      p.value
   <int>         <dbl>        <dbl>
1      1    0.30580219 1.863413e-05
2      2    0.13891126 4.463162e-03
3      3    0.27526672 5.469821e-05
4      4    0.19286524 8.358439e-04
5      5    0.13704449 4.724054e-03
6      6    0.14038104 4.267737e-03
7      7    0.05742603 5.158791e-02
8      8    0.04843200 6.770084e-02
9      9    0.06243191 4.437932e-02
10    10    0.11335589 9.666115e-03
```

Protocolo de exploração de dados

Há erros de manipulação e análise que comprometem as conclusões baseadas em dados. Pensando nisso, Zuur e col. propuseram um protocolo de exploração de dados para evitar erros comuns e facilitar a verificação dos pressupostos de métodos estatísticos comuns. O manuscrito A protocol for data exploration to avoid common statistical problems apresenta 8 passos a serem considerados na exploração de dados, assim como bancos de dados e códigos no R para exemplificar a proposta.

Os códigos do presente capítulo são uma implementação alternativa dos 8 passos que ao serem aplicados nos bancos do artigo, produzem resultados equivalentes. Considero mais simples os códigos do presente capítulo, mas cada leitor terá sua própria opinião ao compará-los com os códigos originais.

Os aspectos conceituais devem ser consultados no manuscrito, pois o foco aqui é o código. Como já conhecemos a maioria das ferramentas deste capítulo, me limitarei a comentar só as novidades que sejam consideravelmente diferentes.

Sumário básico

```
> library(MASS); library(caret); library(GGally); library(car)
> library(ggplot2); library(dplyr); library(tidyr)
> sparrows <- read.table('SparrowsElphick.txt', header = T)
> head(sparrows)
```

```
  wingcrd flatwing tarsus head culmen nalospi   wt bandstat
1    59.0     60.0   22.3 31.2   12.3    13.0  9.5        1
2    54.0     55.0   20.3 28.3   10.8     7.8 12.2        1
3    53.0     54.0   21.6 30.2   12.5     8.5 13.8        1
4    55.0     56.0   19.7 30.4   12.1     8.3 13.8        1
5    55.0     56.0   20.3 28.7   11.2     8.0 14.1        1
6    53.5     54.5   20.8 30.6   12.8     8.6 14.8        1
  initials Year Month Day Location SpeciesCode Sex Age
1        2 2002     9  19        4           1   0   2
2        2 2002    10   4        4           3   0   2
3        2 2002    10   4        4           3   0   2
4        8 2002     7  30        9           1   0   2
5        3 2002    10   4        4           3   0   2
6        7 2004     8   2        1           1   0   2
```

```
> str(sparrows, vec.len = 1)
```

```
'data.frame':    1295 obs. of  16 variables:
 $ wingcrd    : num  59 54 ...
 $ flatwing   : num  60 55 ...
 $ tarsus     : num  22.3 20.3 ...
 $ head       : num  31.2 28.3 ...
 $ culmen     : num  12.3 10.8 ...
 $ nalospi    : num  13 7.8 ...
 $ wt         : num  9.5 12.2 ...
 $ bandstat   : int  1 1 ...
 $ initials   : int  2 2 ...
 $ Year       : int  2002 2002 ...
 $ Month      : int  9 10 ...
 $ Day        : int  19 4 ...
 $ Location   : int  4 4 ...
 $ SpeciesCode: int  1 3 ...
 $ Sex        : int  0 0 ...
 $ Age        : int  2 2 ...
```

Com `vec.len = 1`, `str` mostra menos valores de cada uma das variáveis.

Junto com `str` e `head`, `summary` é uma das funções básicas para a exploração de dados.

```
> summary(sparrows)
```

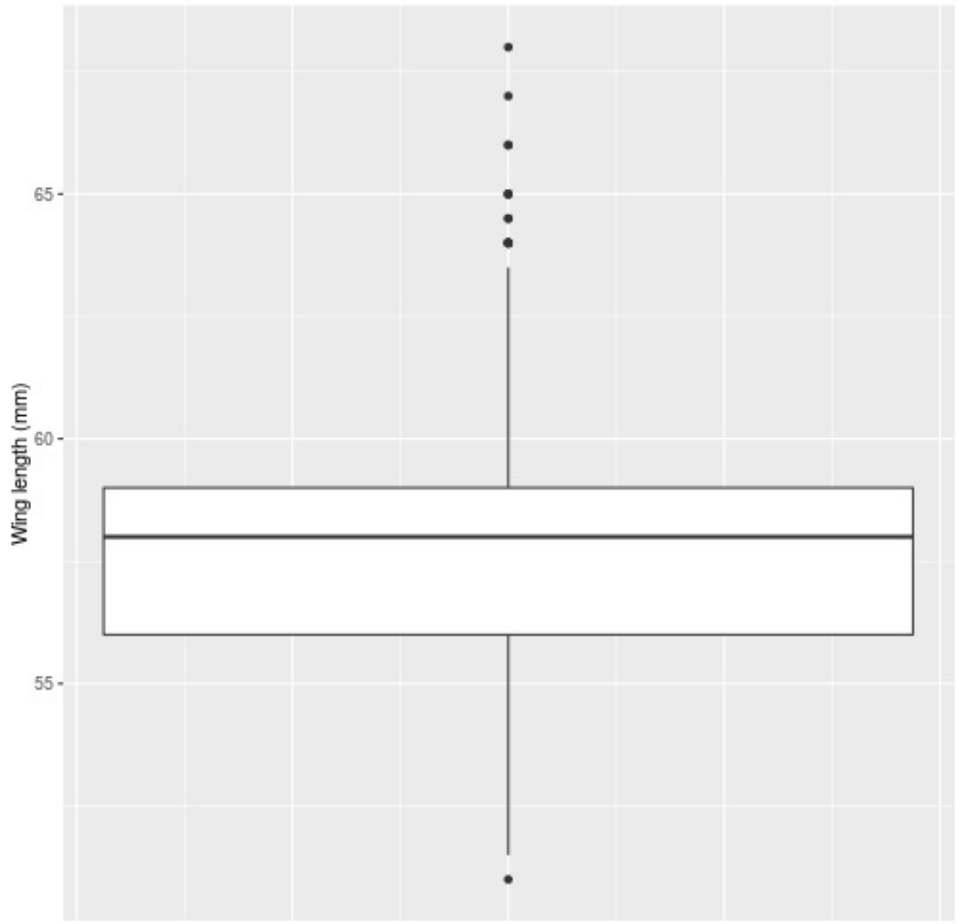
```
    wingcrd         flatwing         tarsus           head
 Min.   :51.00   Min.   :52.00   Min.   :18.90   Min.   :21.10
 1st Qu.:56.00   1st Qu.:57.00   1st Qu.:20.80   1st Qu.:31.30
 Median :58.00   Median :59.00   Median :21.30   Median :31.80
 Mean   :57.69   Mean   :58.77   Mean   :21.42   Mean   :31.85
 3rd Qu.:59.00   3rd Qu.:60.00   3rd Qu.:21.80   3rd Qu.:32.30
 Max.   :68.00   Max.   :68.50   Max.   :31.80   Max.   :35.70
     culmen        nalospi             wt           bandstat
 Min.   : 8.9   Min.   : 6.300   Min.   : 9.50   Min.   :1.000
 1st Qu.:12.5   1st Qu.: 9.100   1st Qu.:18.50   1st Qu.:1.000
 Median :13.0   Median : 9.400   Median :19.90   Median :1.000
 Mean   :13.0   Mean   : 9.527   Mean   :19.76   Mean   :1.036
 3rd Qu.:13.4   3rd Qu.: 9.800   3rd Qu.:20.85   3rd Qu.:1.000
 Max.   :16.6   Max.   :18.800   Max.   :27.50   Max.   :2.000
    initials          Year          Month             Day
 Min.   :0.000   Min.   :2002   Min.   : 5.000   Min.   : 1.00
 1st Qu.:2.000   1st Qu.:2002   1st Qu.: 6.000   1st Qu.: 7.00
 Median :3.000   Median :2003   Median : 7.000   Median :13.00
 Mean   :3.893   Mean   :2003   Mean   : 6.827   Mean   :14.65
 3rd Qu.:6.000   3rd Qu.:2003   3rd Qu.: 7.000   3rd Qu.:23.00
 Max.   :9.000   Max.   :2004   Max.   :10.000   Max.   :31.00
    Location       SpeciesCode         Sex             Age
 Min.   : 1.000   Min.   :1.000   Min.   :0.000   Min.   :1.000
 1st Qu.: 7.000   1st Qu.:1.000   1st Qu.:4.000   1st Qu.:1.000
 Median : 8.000   Median :1.000   Median :4.000   Median :1.000
 Mean   : 7.494   Mean   :1.136   Mean   :3.419   Mean   :1.164
 3rd Qu.: 9.000   3rd Qu.:1.000   3rd Qu.:4.000   3rd Qu.:1.000
 Max.   :11.000   Max.   :3.000   Max.   :5.000   Max.   :2.000
```

Detecção de outliers

Box plot

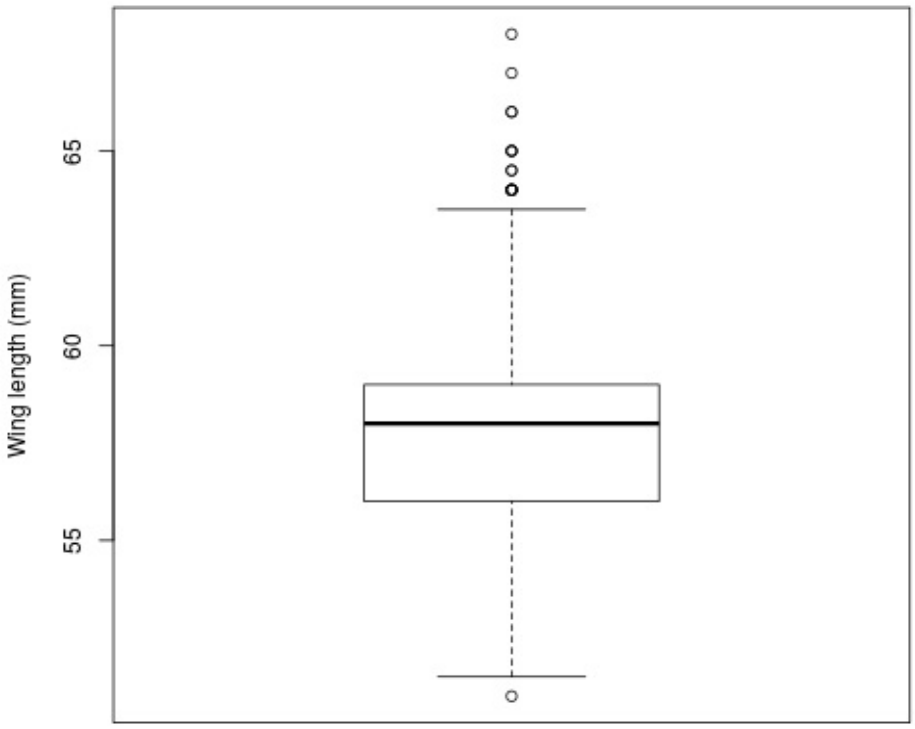
Os boxplot dentro do pacote `ggplot2` são conceituados para representar uma variável qualitativa e uma quantitativa. Portanto, é necessário instanciar os argumentos `x` e `y` na função `aes`. Quando o objetivo é criar um boxplot de uma variável quantitativa sem considerar outra qualitativa, devemos instanciar `x` de qualquer forma e posteriormente omitir a etiqueta e marca do respectivo eixo.

```
> ggplot(sparrows, aes(x = 1, y = wingcrd)) +
+   geom_boxplot() +
+   xlab('') +
+   ylab('Wing length (mm)') +
+   theme(axis.ticks.x = element_blank(),
+         axis.text.x = element_blank())
```



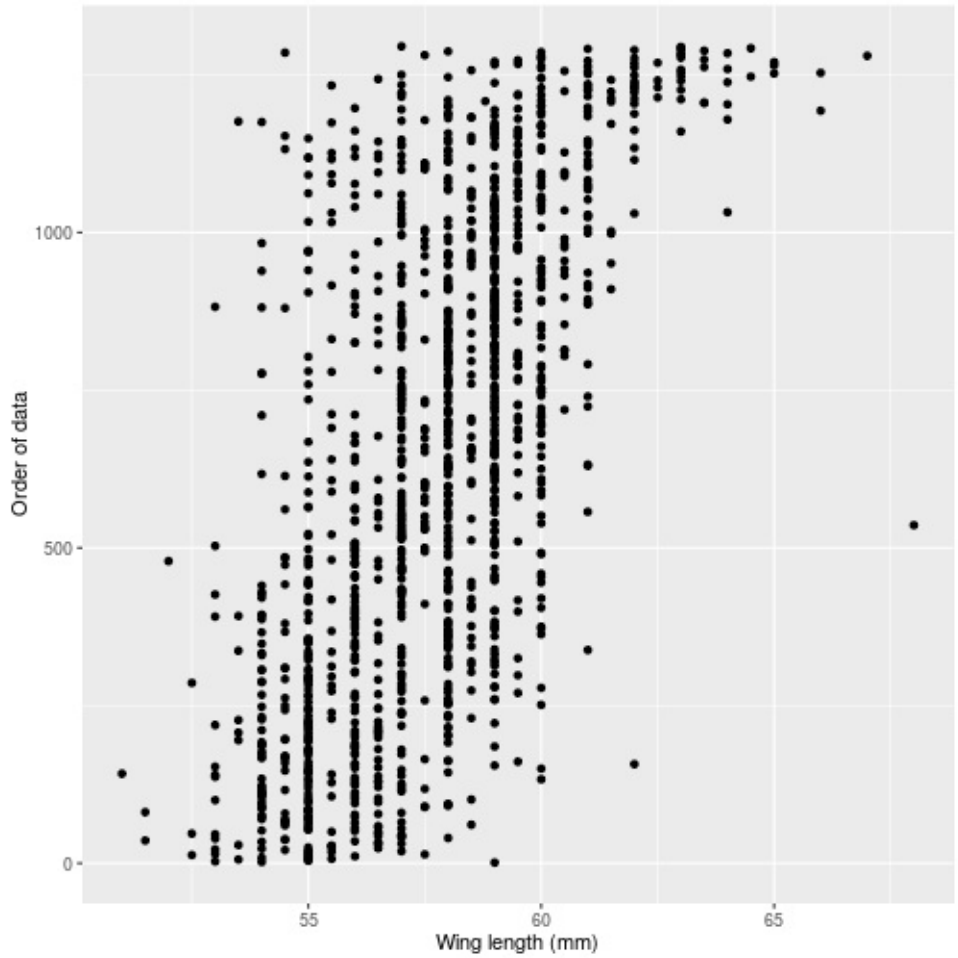
Nesses casos, a abordagem convencional do R é bem mais simples.

```
> boxplot(sparrows$wingcrd, ylab = 'Wing length (mm)')
```

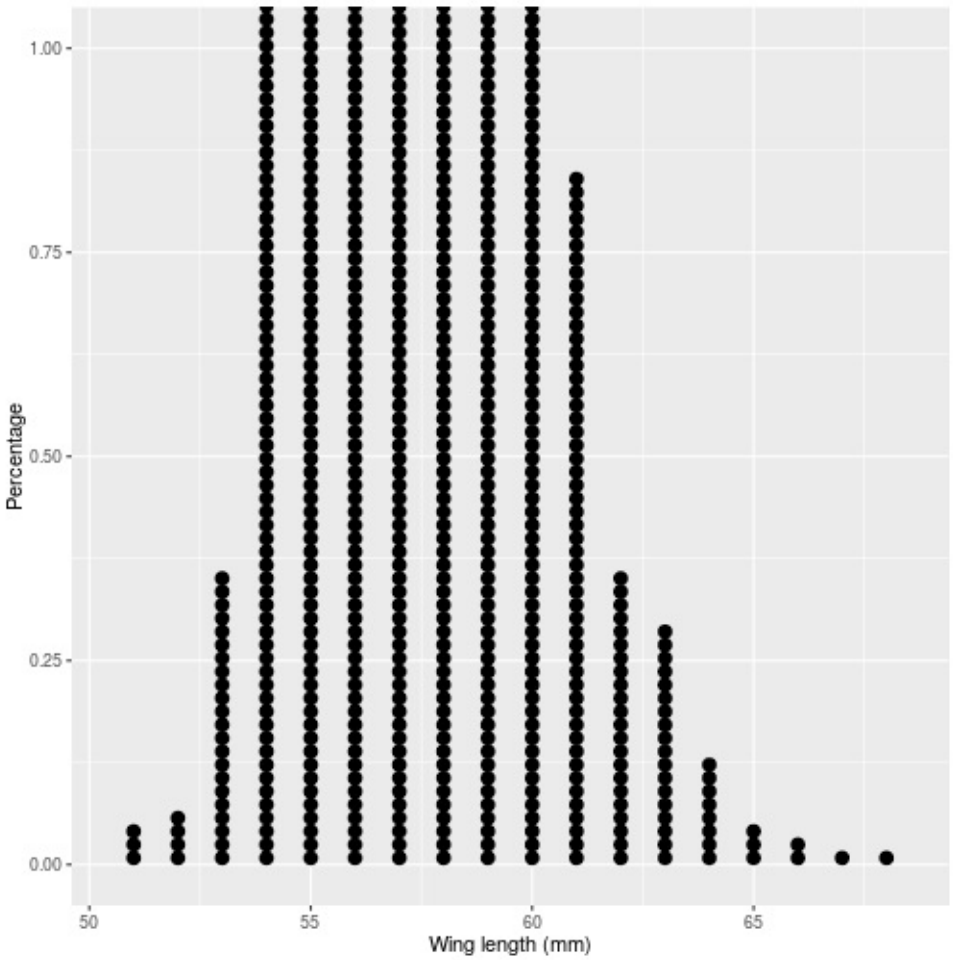


Dotplot

```
> ggplot(sparrows, aes(wingcrd, 1:nrow(sparrows))) +
+   geom_point() +
+   xlab('Wing length (mm)') +
+   ylab('Order of data')
```




```
> ggplot(sparrows, aes(wingcrd)) +
+   geom_dotplot(method = 'histodot', binwidth = 1, dotsize = .3) +
+   xlab('Wing length (mm)') +
+   ylab('Percentage')
```



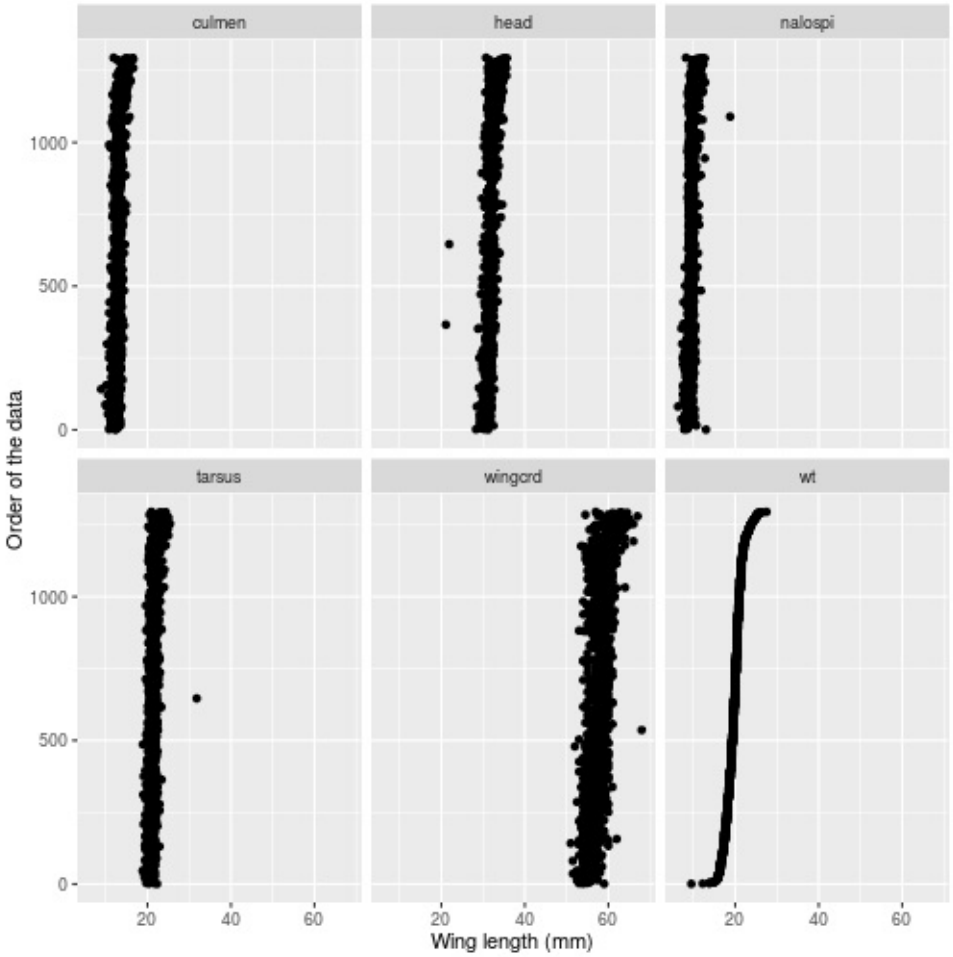

```
> sparrows2 <- sparrows %>%
+   mutate(order = 1:nrow(sparrows)) %>%
+   select(order, wingcrd, tarsus, head, culmen, nalospi, wt)
> head(sparrows2)
```

```
  order wingcrd tarsus head culmen nalospi   wt
1     1    59.0   22.3 31.2   12.3    13.0  9.5
2     2    54.0   20.3 28.3   10.8     7.8 12.2
3     3    53.0   21.6 30.2   12.5     8.5 13.8
4     4    55.0   19.7 30.4   12.1     8.3 13.8
5     5    55.0   20.3 28.7   11.2     8.0 14.1
6     6    53.5   20.8 30.6   12.8     8.6 14.8
```

```
> sparrows2 <- gather(sparrows2, variable, value, -order)
> head(sparrows2)
```

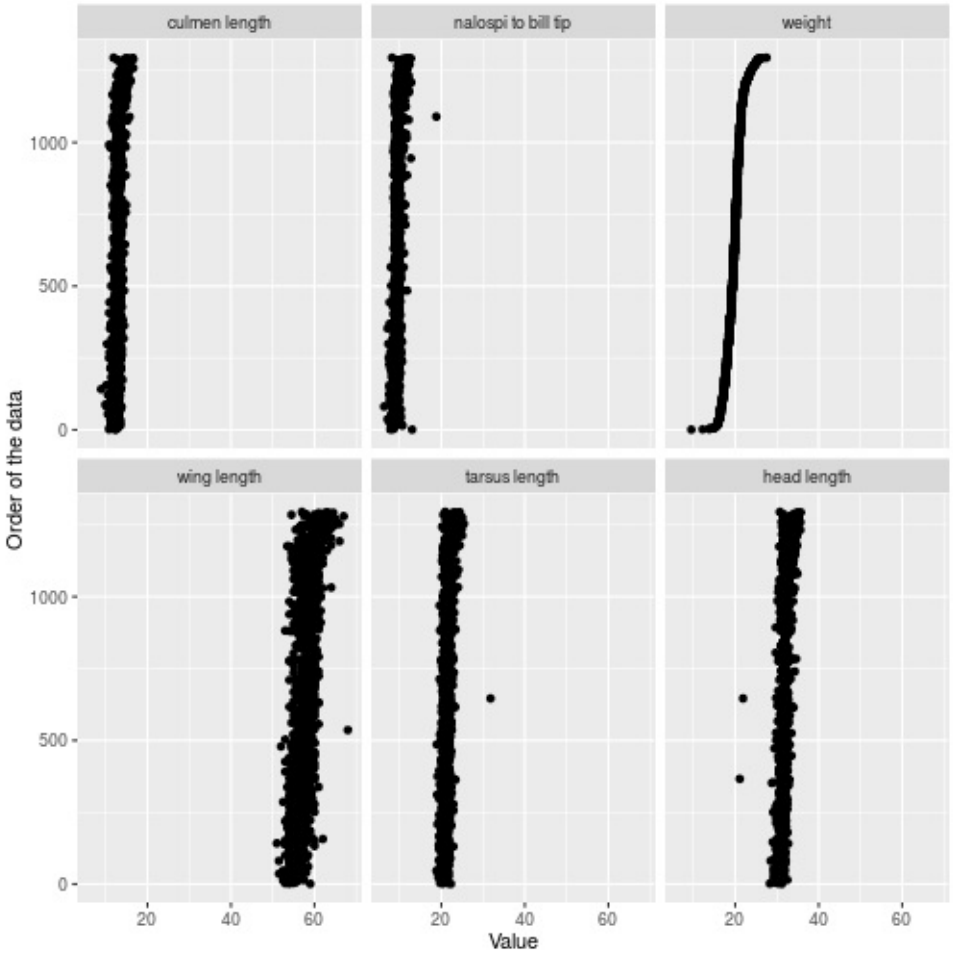
```
  order variable value
1     1  wingcrd  59.0
2     2  wingcrd  54.0
3     3  wingcrd  53.0
4     4  wingcrd  55.0
5     5  wingcrd  55.0
6     6  wingcrd  53.5
```

```
> ggplot(sparrows2, aes(value, order)) +
+   geom_point() +
+   facet_wrap(~ variable) +
+   xlab('Wing length (mm)') +
+   ylab('Order of the data')
```




```
> sparrows2$variable <- factor(sparrows2$variable,
+                              levels = c('culmen', 'nalospi', 'wt',
+                                         'wingcrd', 'tarsus', 'head'),
+                              labels = c('culmen length', 'nalospi to bill tip',
+                                         'weight', 'wing length',
+                                         'tarsus length', 'head length'))
```

```
> ggplot(sparrows2, aes(value, order)) +
+   geom_point() +
+   facet_wrap(~ variable) +
+   xlab('Value') +
+   ylab('Order of the data')
```



Homogeneidade de variâncias

Box plot

```
> godwits <- read.table('Godwits.txt', header = T)
> str(godwits)
```

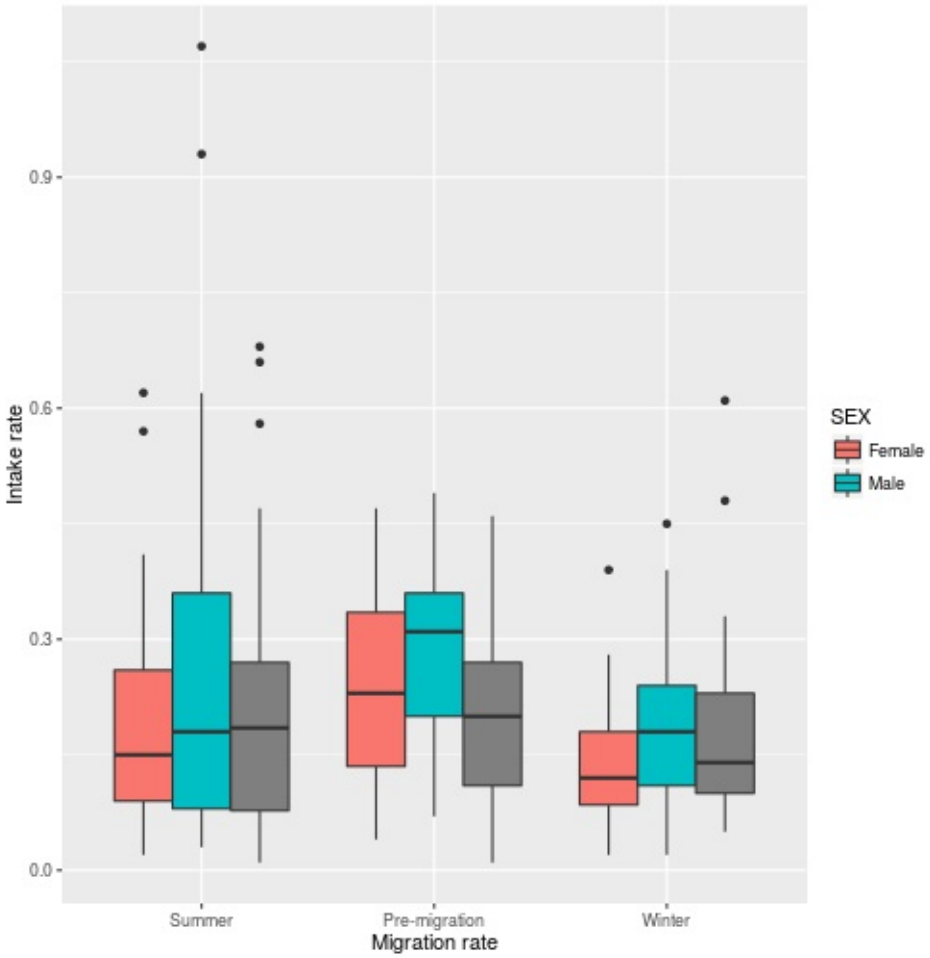
```
'data.frame':    330 obs. of  9 variables:
 $ RECORD    : int  1 2 3 4 5 6 7 8 9 10 ...
 $ DAY       : int  5 5 5 5 5 5 5 6 6 6 ...
 $ MONTH     : int  1 1 1 1 1 1 1 2 2 2 ...
 $ YEAR      : int  97 97 97 97 97 97 97 97 97 97 ...
 $ LOCATION  : int  0 0 0 0 0 0 0 1 0 ...
 $ AGE       : int  0 0 0 0 0 0 0 0 0 0 ...
 $ SEX       : int  0 0 0 0 0 0 0 0 0 0 ...
 $ PERIOD    : num  0 0 0 0 0 0 0 1 1 1 ...
 $ mgconsumed: num  0.07 0.16 0.25 0.07 0.14 0.26 0.1 0.21 0.11 0.09 ...
```

A variável `SEX` tem três valores diferentes (`0` , `1` e `2`).). O `1` representa as fêmeas e o `2` os machos. Para omitirmos o `0` e atribuir etiquetas transformaremos a variável em um fator só com os níveis de interesse (`1` e `2`).

```
> godwits <- godwits %>%
+   mutate(SEX = factor(SEX, levels = c(1, 2),
+                       labels = c('Female', 'Male')),
+          PERIOD = factor(PERIOD, levels = c(0, 1, 2),
+                          labels = c('Summer', 'Pre-migration', 'Winter')))
```

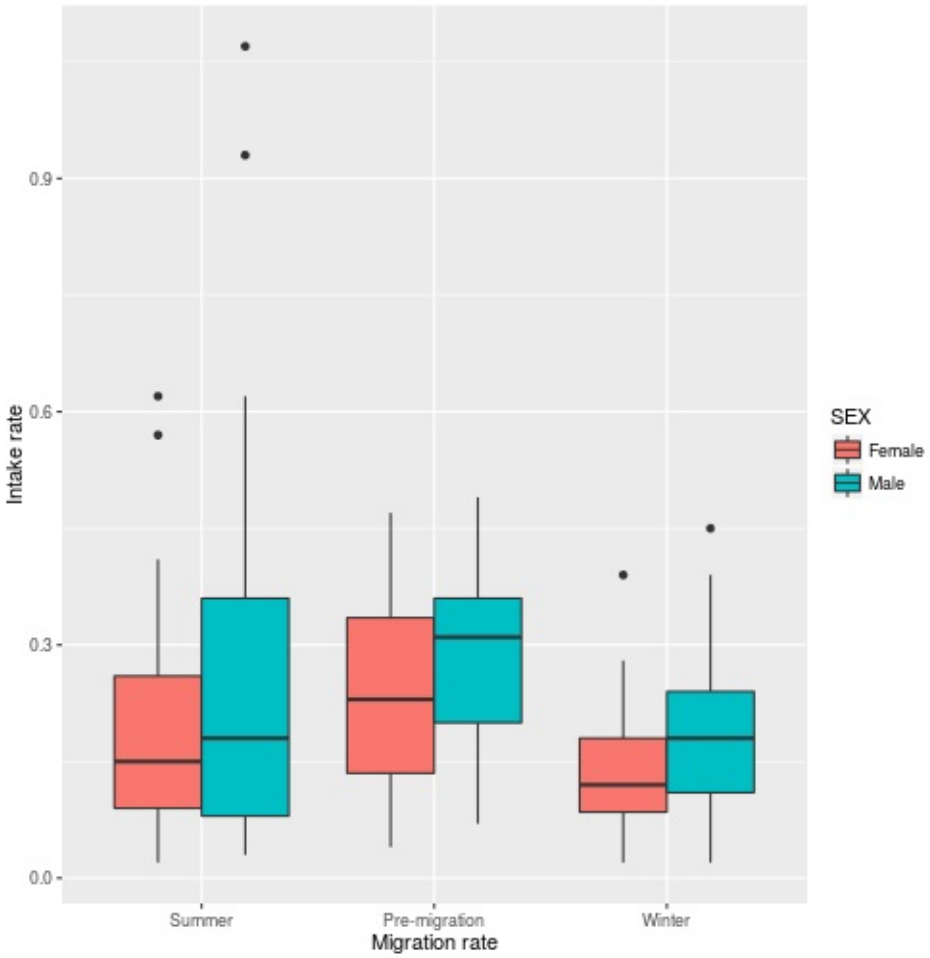
As observações com `0` são representadas com `NA` .

```
> ggplot(godwits, aes(PERIOD, mgconsumed, fill = SEX)) +
+   geom_boxplot() +
+   xlab('Migration rate') +
+   ylab('Intake rate')
```

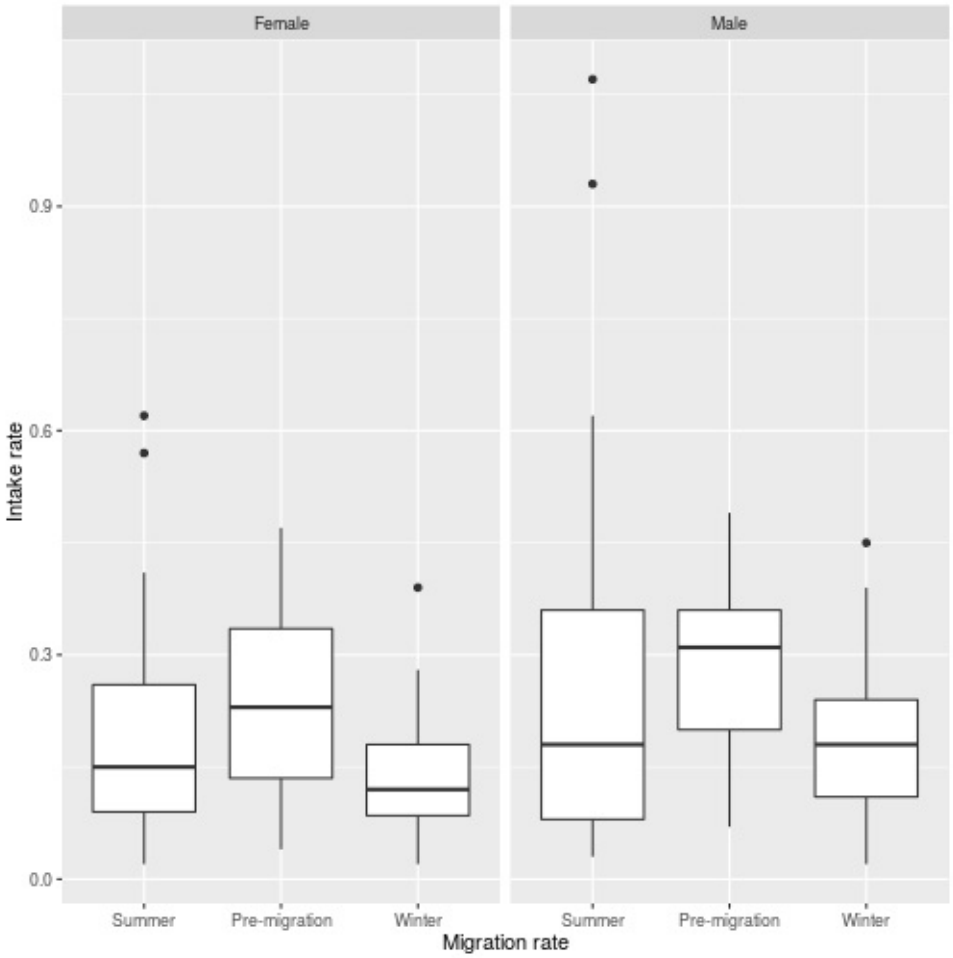


A função `na.omit` omite todas as observações com `NA`.

```
> ggplot(na.omit(godwits), aes(PERIOD, mgconsumed, fill = SEX)) +
+   geom_boxplot() +
+   xlab('Migration rate') +
+   ylab('Intake rate')
```




```
> ggplot(na.omit(godwits), aes(PERIOD, mgconsumed)) +
+   geom_boxplot() +
+   facet_wrap(~ SEX) +
+   xlab('Migration rate') +
+   ylab('Intake rate')
```



Distribuição normal

Histograma

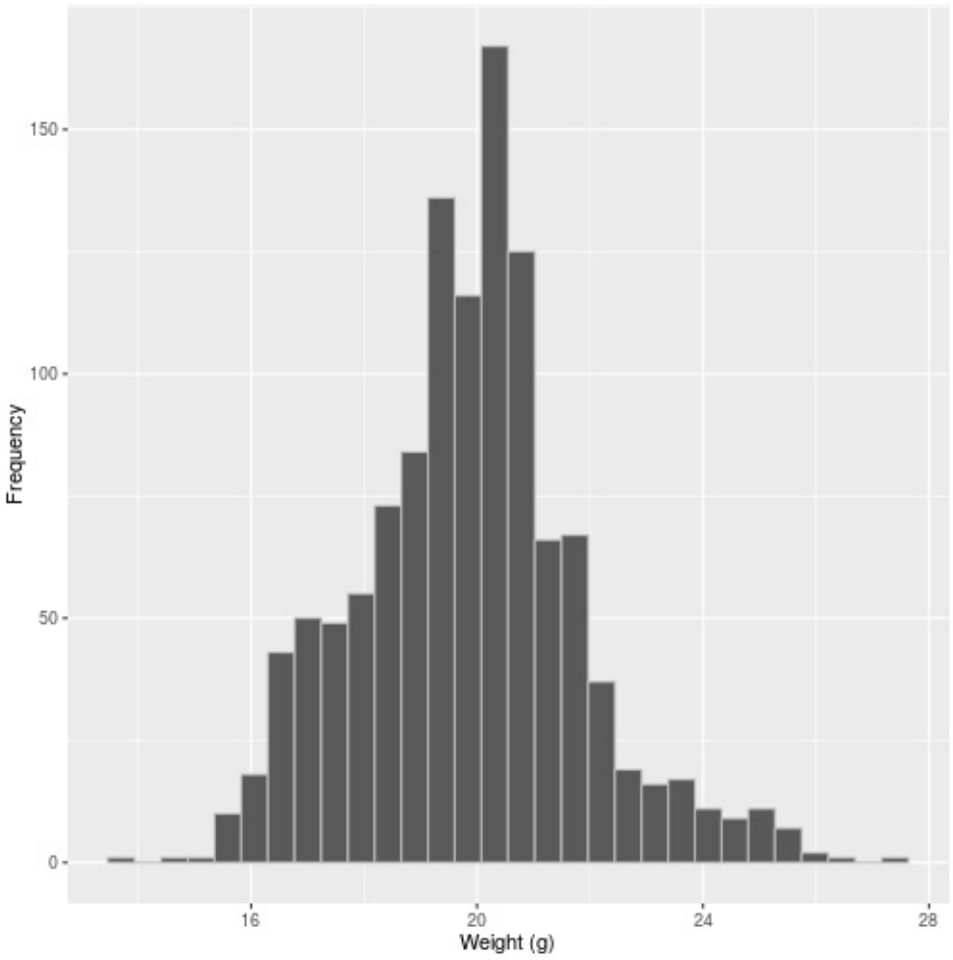
```
> str(sparrows)
```

```
'data.frame':    1295 obs. of  16 variables:
 $ wingcrd    : num  59 54 53 55 55 53.5 55.5 55 54 55 ...
 $ flatwing   : num  60 55 54 56 56 54.5 57 56 56 56 ...
 $ tarsus     : num  22.3 20.3 21.6 19.7 20.3 20.8 20.3 20.9 21.4 20.4 ...
 $ head       : num  31.2 28.3 30.2 30.4 28.7 30.6 29.5 30 29.8 29.8 ...
 $ culmen     : num  12.3 10.8 12.5 12.1 11.2 12.8 11.5 11.7 11 11.2 ...
 $ nalospi    : num  13 7.8 8.5 8.3 8 8.6 8.5 8.5 8.7 8.1 ...
 $ wt         : num  9.5 12.2 13.8 13.8 14.1 14.8 15 15 15.1 15.1 ...
 $ bandstat   : int  1 1 1 1 1 1 1 1 1 1 ...
 $ initials   : int  2 2 2 8 3 7 3 3 1 2 ...
 $ Year       : int  2002 2002 2002 2002 2002 2004 2002 2002 2002 2002 ...
 $ Month      : int  9 10 10 7 10 8 10 10 10 10 ...
 $ Day        : int  19 4 4 30 4 2 4 2 21 2 ...
 $ Location   : int  4 4 4 9 4 1 4 5 7 5 ...
 $ SpeciesCode: int  1 3 3 1 3 1 3 3 3 3 ...
 $ Sex        : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Age        : int  2 2 2 2 2 2 2 1 2 2 ...
```

```
> sparrows$Month <- factor(sparrows$Month,
+                          levels = c(5, 6, 7, 8, 9, 10),
+                          labels = c('May', 'June', 'July', 'August',
+                                     'Sep', 'Oct'))
```

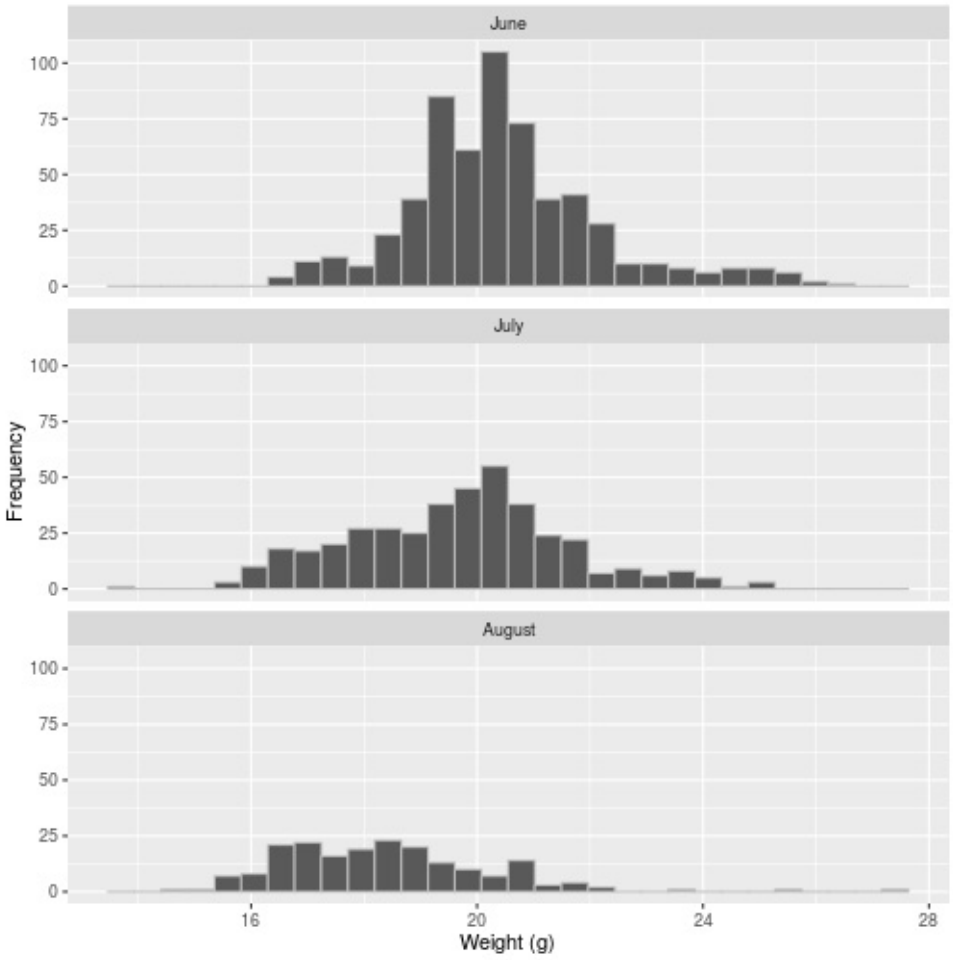
```
> ggplot(filter(sparrows, Month == 'June' | Month == 'July' | Month == 'August'),
+        aes(wt)) +
+   geom_histogram(color = 'gray') +
+   xlab('Weight (g)') +
+   ylab('Frequency')
```

```
`stat_bin()` using `bins = 30`. Pick better value with
`binwidth`.
```




```
> ggplot(filter(sparrows, Month == 'June' | Month == 'July' | Month == 'August'),
+        aes(wt)) +
+   geom_histogram(color = 'gray') +
+   facet_wrap(~ Month, ncol = 1) +
+   xlab('Weight (g)') +
+   ylab('Frequency')
```

```
`stat_bin()` using `bins = 30`. Pick better value with
`binwidth`.
```



Exceso de zeros

Proporção de zeros

```
> rice_field <- read.table(file='ElphickBirdData.txt', header = T)
```

Apos calcular a frequência de valores únicos do produto entre `AREA` e `AQBIRDS` com a função `table`,

```
> freqs <- table(round(rice_fieldAREA * rice_fieldAQBIRDS))
```

```
Error in table(round(rice_fieldAREA * rice_fieldAQBIRDS)): object 'rice_fieldAREA' not
 found
```

```
> freqs[1:5]
```

```
  0   1   2   3   4
718 131 108  46  52
```

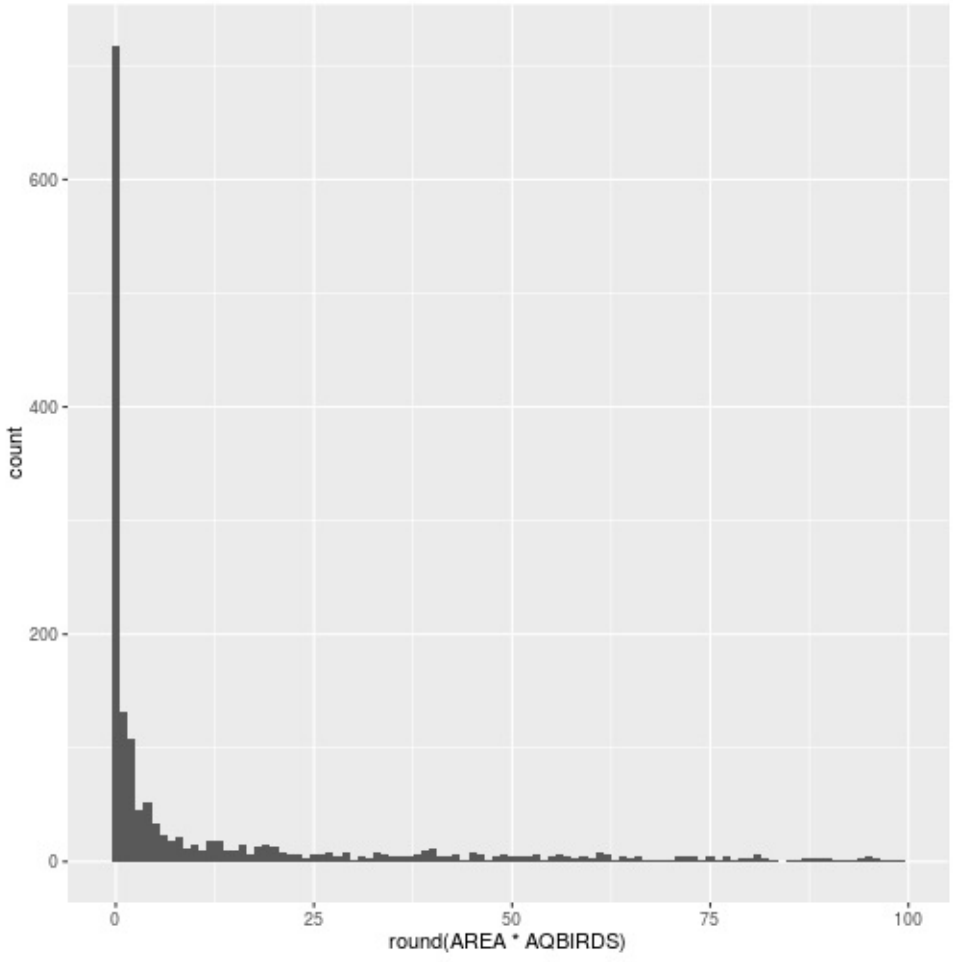
a proporção de zeros é a divisão do total de zeros pelo número observações no banco.

```
> freqs[1] / nrow(rice_field)
```

```
        0
0.3528256
```

```
> ggplot(rice_field, aes(round(AREA * AQBIRDS))) +
+   geom_histogram(binwidth = 1) +
+   xlim(c(-1, 100))
```

```
Warning: Removed 400 rows containing non-finite values (stat_bin).
```

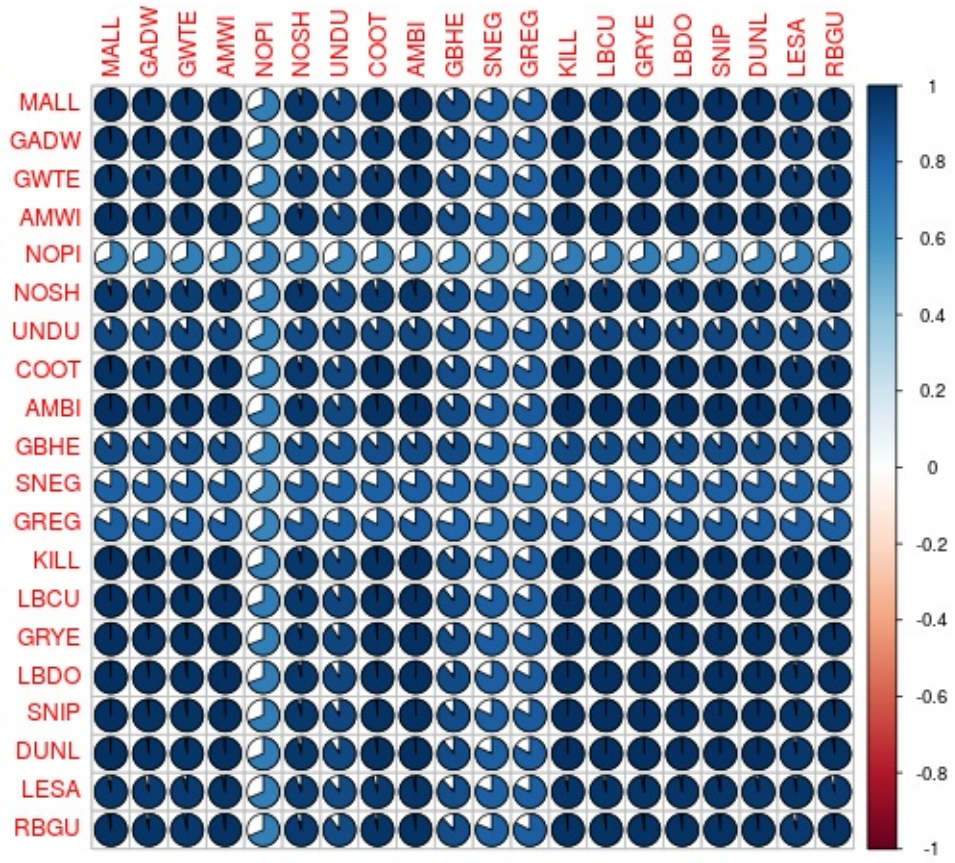


Correlograma

Os correlogramas são tipicamente usados para representar matrizes de correlação. Entretanto, servem para outros propósitos como ilustrado no manuscrito e a continuação.

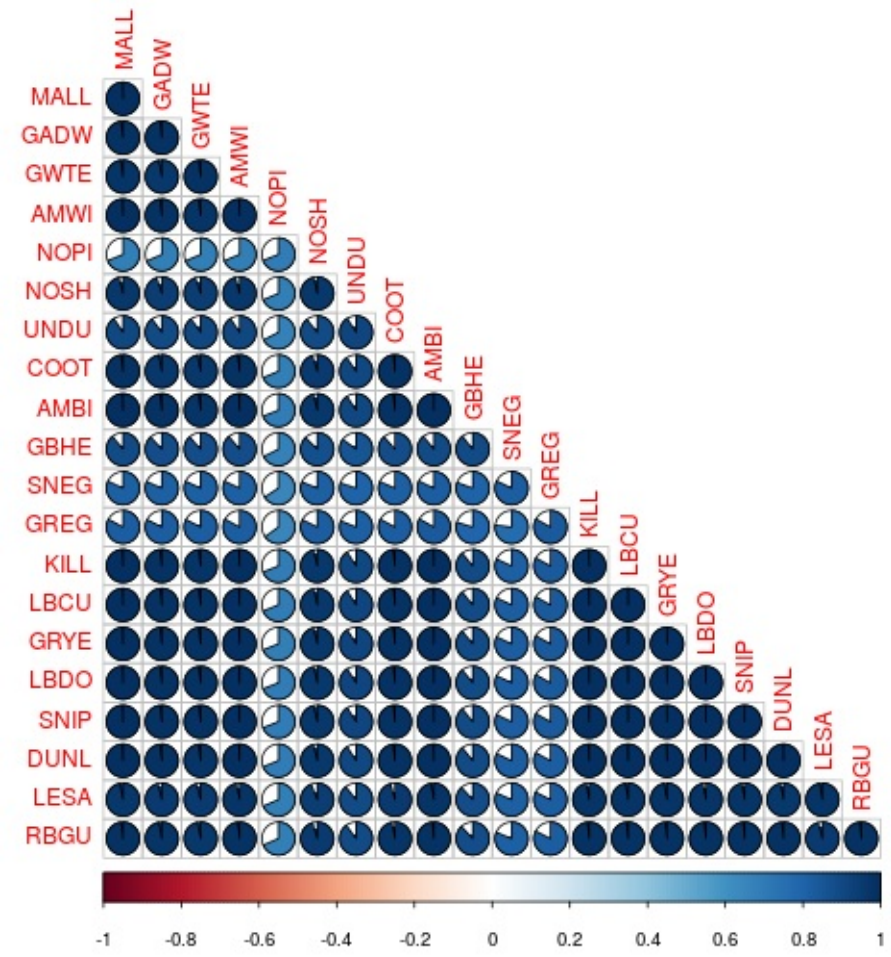
```
> especies <- rice_field[, 14:62]
> abundancia <- colSums(especies > 0, na.rm = T)
> especies2 <- especies[, abundancia > 40]
> cls <- ncol(especies2)
> prop_zeros <- matrix(ncol = cls, nrow = cls)
>
> # Para reproduzir a matriz que gera o gráfico do artigo, o laço de repetição
> # é baseado no objeto especies, mas pela lógica das linhas anteriores
> # (que reproduzem as linhas correspondentes do material suplementar),
> # acredito que deveria ser o obetjo especies2.
> for (i in 1:cls) {
+   for (j in 1:cls){
+     prop_zeros[i, j] <- sum(especies[, i] == 0 & especies[, j] == 0, na.rm = T)
+   }
+ }
>
> prop_zeros <- prop_zeros / nrow(especies2)
> rownames(prop_zeros) <- names(especies2)
> colnames(prop_zeros) <- names(especies2)
```

```
> library(corrplot)
> corrplot(prop_zeros, method = 'pie')
```



Como o triângulo inferior é uma imagem especular do superior, é desnecessário apresentar a matriz completa.

```
> corrplot(prop_zeros, method = 'pie', type = 'lower')
```



Colinearidade

```
> sparrows3 <- read.table(file = 'VegSamplesV1.txt', header = T)
```

VIF

```
> mod1 <- lm(Banded ~ Avgmaxht + Avgdens + ht.thatch + S.patens +
+               Distichlis + S.alternifloraShort + S.alternifloraTall +
+               Juncus + Bare + Other + Phragmites + Shrub + Tallsedge +
+                Water, data = sparrows3)
> summary(mod1)
```

```
Call:
lm(formula = Banded ~ Avgmaxht + Avgdens + ht.thatch + S.patens +
    Distichlis + S.alternifloraShort + S.alternifloraTall + Juncus +
    Bare + Other + Phragmites + Shrub + Tallsedge + Water, data = sparrows3)

Residuals:
    Min      1Q  Median      3Q     Max
-19.142  -5.431   1.011   5.240  18.634

Coefficients:
                      Estimate Std. Error t value Pr(>|t|)
(Intercept)         -1.612e+02  8.780e+01  -1.836   0.0730 .
Avgmaxht             4.195e-01  3.548e-01   1.183   0.2432
Avgdens              6.158e-02  1.719e-01   0.358   0.7219
ht.thatch            8.427e-04  6.241e-01   0.001   0.9989
S.patens             1.626e+00  8.562e-01   1.899   0.0640 .
Distichlis           1.703e+00  8.558e-01   1.990   0.0527 .
S.alternifloraShort  1.668e+00  8.462e-01   1.971   0.0549 .
S.alternifloraTall   1.449e+00  8.521e-01   1.700   0.0960 .
Juncus               2.009e+00  8.354e-01   2.405   0.0203 *
Bare                 1.620e+00  8.615e-01   1.880   0.0666 .
Other                1.800e+00  9.807e-01   1.836   0.0730 .
Phragmites           1.851e+00  1.003e+00   1.846   0.0715 .
Shrub                6.300e-02  1.248e+00   0.050   0.9600
Tallsedge            1.471e+00  1.172e+00   1.255   0.2160
Water                2.092e+00  1.070e+00   1.955   0.0568 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 10.45 on 45 degrees of freedom
Multiple R-squared:  0.393,    Adjusted R-squared:  0.2041
F-statistic: 2.081 on 14 and 45 DF,  p-value: 0.03214
```

```
> vif(mod1)
```

```
          Avgmaxht            Avgdens          ht.thatch
          6.120018           3.206401           1.671224
          S.patens         Distichlis S.alternifloraShort
        159.350658          53.754540         121.463782
 S.alternifloraTall             Juncus               Bare
        159.382860          44.995377          12.058665
             Other          Phragmites              Shrub
          5.817015           3.749027           2.781804
         Tallsedge              Water
          4.409398          17.067777
```

```
> cbind(vif(mod1))
```

```
                          [,1]
Avgmaxht              6.120018
Avgdens               3.206401
ht.thatch             1.671224
S.patens            159.350658
Distichlis           53.754540
S.alternifloraShort 121.463782
S.alternifloraTall  159.382860
Juncus               44.995377
Bare                 12.058665
Other                 5.817015
Phragmites            3.749027
Shrub                 2.781804
Tallsedge             4.409398
Water                17.067777
```

A seguinte estrutura de controle elimina sequencialmente a covariável com o maior VIF até que o maior VIF seja menor do que 3. `form` é um caractere que representa a fórmula da regressão e para incluí-lo como argumento de `lm` devemos usar a função `as.formula`. O código deve ser modificado antes de ser usado com variáveis explicativas qualitativas.

```
> vifs <- sort(vif(mod1), d = T)
> while(vifs[1] > 3) {
+     form <- paste('Banded', '~', paste0(names(vifs[-1]), collapse = ' + '))
+     mod2 <- lm(as.formula(form), sparrows3)
+     vifs <- sort(vif(mod2), d = T)
+ }
> summary(mod2)
```

```
Call:
lm(formula = as.formula(form), data = sparrows3)

Residuals:
     Min       1Q   Median       3Q      Max
-19.1448  -6.3075   0.4815   7.2374  20.0593

Coefficients:
                    Estimate Std. Error t value Pr(>|t|)
(Intercept)          9.93887    8.38424   1.185 0.241685
S.patens             0.08011    0.08162   0.982 0.331243
S.alternifloraShort  0.12045    0.11375   1.059 0.294972
Juncus               0.59087    0.14415   4.099 0.000159 ***
Distichlis           0.15929    0.13794   1.155 0.253875
ht.thatch            0.13163    0.59680   0.221 0.826366
Other                0.03050    0.45018   0.068 0.946264
Phragmites           0.65102    0.58768   1.108 0.273473
Bare                 0.03854    0.27935   0.138 0.890862
Shrub               -1.59457    0.81712  -1.951 0.056853 .
Tallsedge            0.48892    0.61606   0.794 0.431324
Water                0.11326    0.28638   0.395 0.694233
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 10.61 on 48 degrees of freedom
Multiple R-squared:  0.3331,    Adjusted R-squared:  0.1802
F-statistic: 2.179 on 11 and 48 DF,  p-value: 0.03169
```

Seleção por AIC (stepwise)

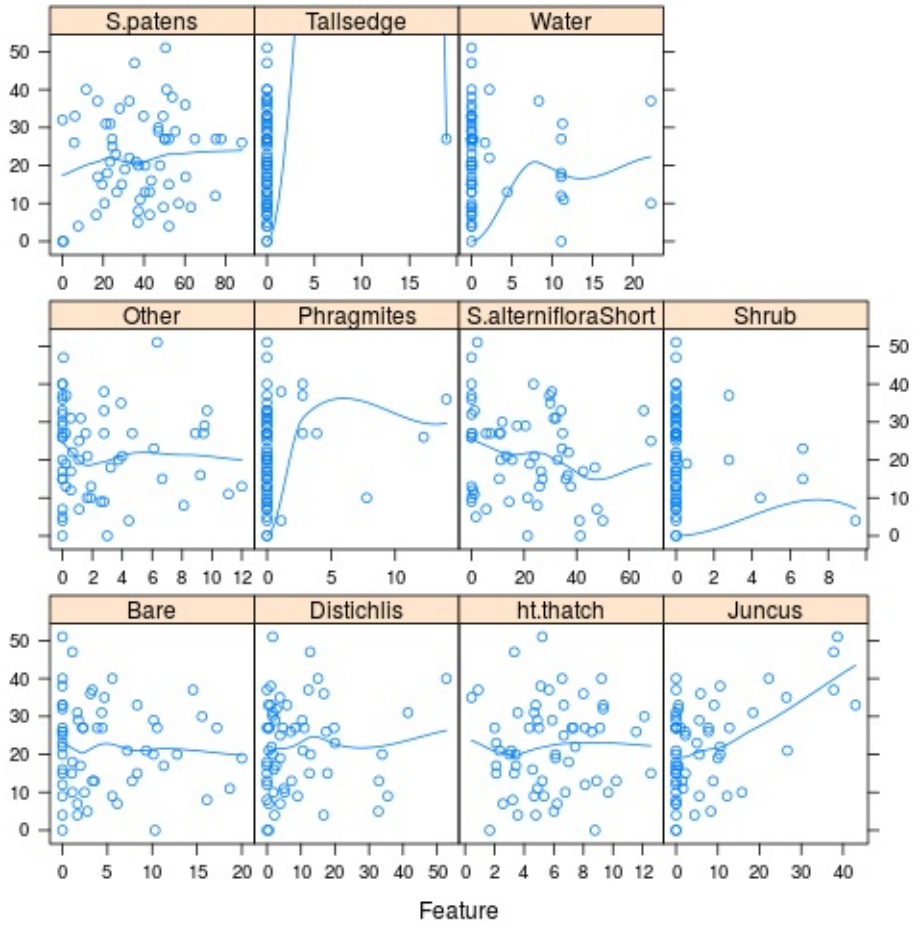
```
> mod3 <- stepAIC(mod2, trace = 0)
> summary(mod3)
```

```
Call:
lm(formula = Banded ~ Juncus + Shrub, data = sparrows3)

Residuals:
    Min      1Q  Median      3Q     Max
-19.022  -7.495   1.283   7.978  20.978

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  19.0220     1.5944  11.930  < 2e-16 ***
Juncus        0.5354     0.1204   4.445 4.11e-05 ***
Shrub        -1.3237     0.7240  -1.828   0.0727 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 10.1 on 57 degrees of freedom
Multiple R-squared:  0.2822,    Adjusted R-squared:  0.2571
F-statistic: 11.21 on 2 and 57 DF,  p-value: 7.858e-05
```

Relação entre variáveis

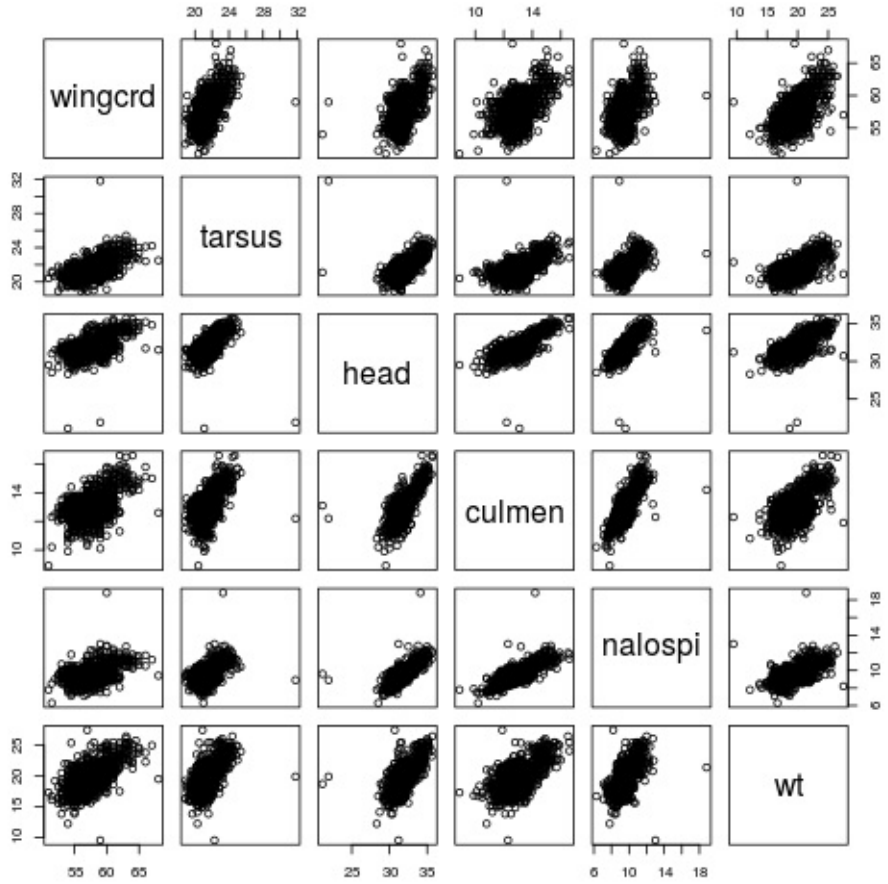
Gráfico de dispersão e linha de ajuste (modelo não linear)

```
> featurePlot(sparrows3[, names(vifs)], sparrows3$Banded,
+             type = c('p', 'smooth'))
```

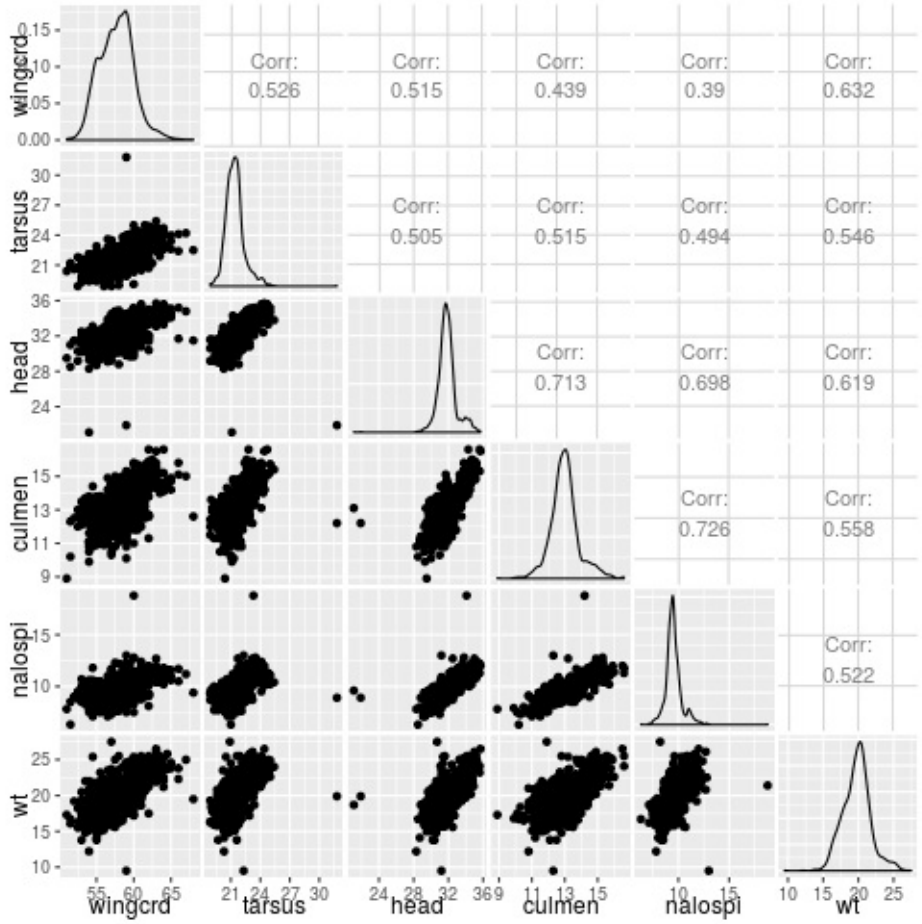


Correlograma

```
> pairs(sparrows[, c(1, 3, 4, 5, 6, 7)])
```



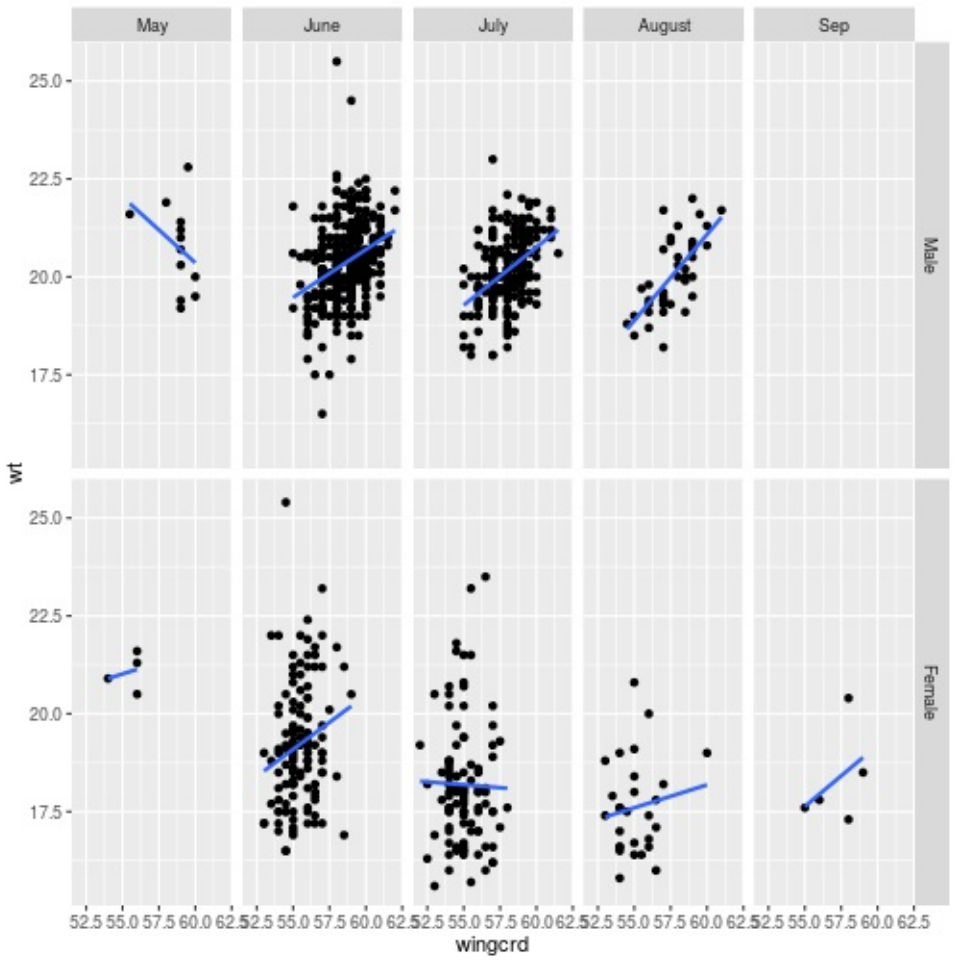

```
> ggpairs(sparrows[, c(1, 3, 4, 5, 6, 7)])
```



Interação

Gráfico de dispersão e linha de ajuste (modelo não linear)

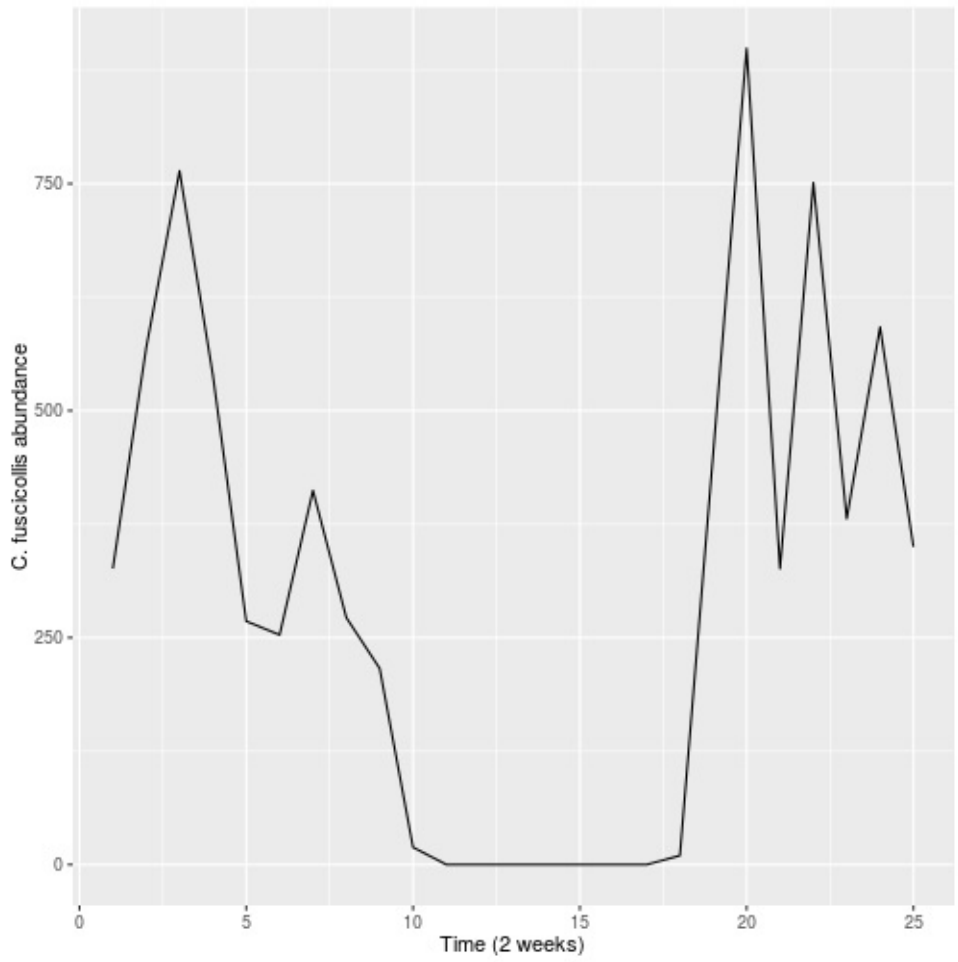
```
> sparrows4 <- filter(sparrows, SpeciesCode == 1 & Sex != '0' & wingcrd < 65)
> sparrows4$Sex <- factor(sparrows4$Sex, levels = c(4, 5),
+                         labels = c('Male', 'Female'))
>
> ggplot(sparrows4, aes(wingcrd, wt)) +
+   geom_point() +
+   stat_smooth(method = 'lm', se = F) +
+   facet_grid(Sex ~ Month)
```



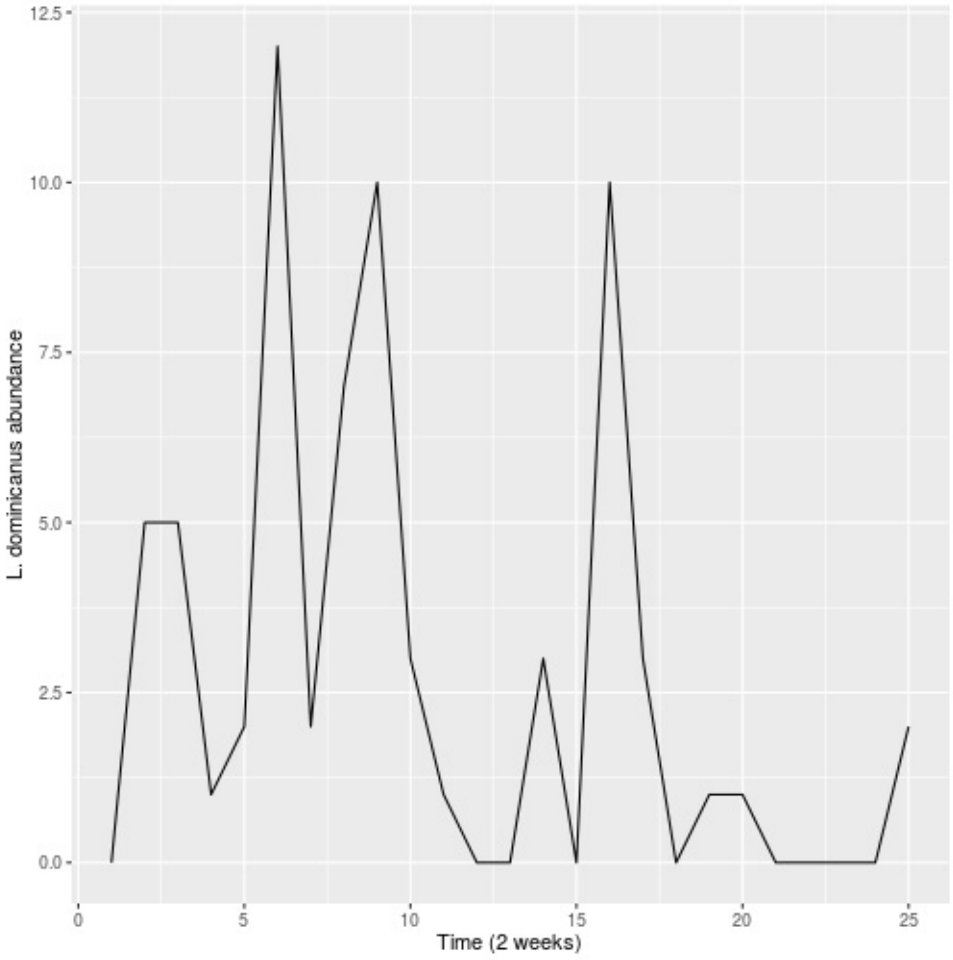
Independência temporal do desfecho

Linhas

```
> waders <- read.table(file = 'wader.txt', header = T)
>
> ggplot(waders, aes(1:25, C.fuscicolis)) +
+   geom_line() +
+   xlab('Time (2 weeks)') +
+   ylab('C. fuscicollis abundance')
```

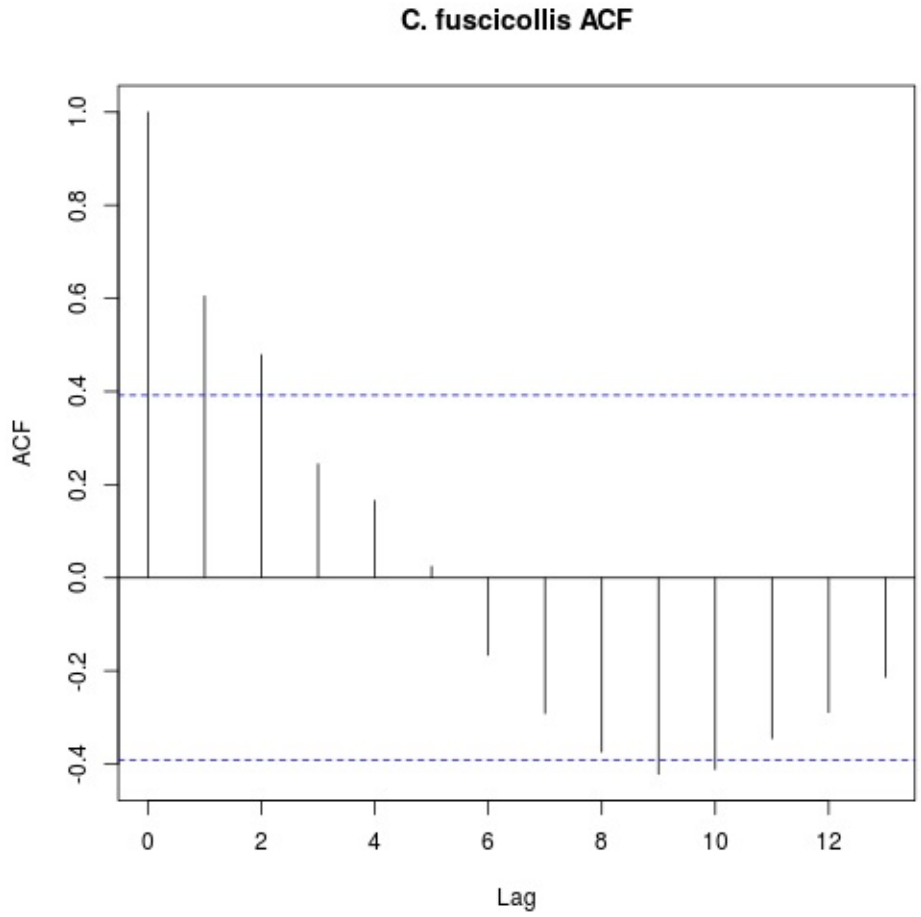



```
> ggplot(waders, aes(1:25, L.dominicanus)) +
+   geom_line() +
+   xlab('Time (2 weeks)') +
+   ylab('L. dominicanus abundance')
```

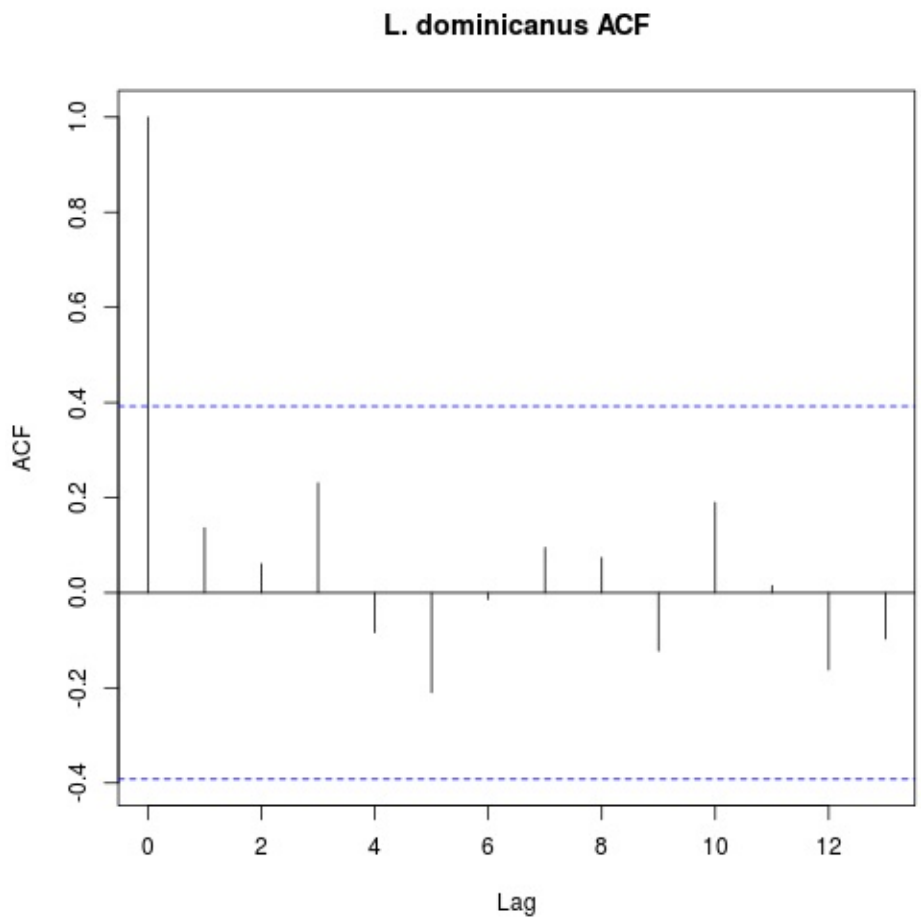


Gráficos de autocorrelação

```
> acf(waders$C.fuscicolis, main = 'C. fuscicollis ACF')
```




```
> acf(waders$L.dominicanus, main = 'L. dominicanus ACF')
```


L. dominicanus ACF
ACF
1.0
0.8
0.6
0.4
0.2
0.0
-0.2
-0.4
0
2
4
6
8
10
12
Lag

Referências

- Dasu, Tamraparni, and Theodore Johnson. Exploratory data mining and data cleaning. Vol. 479. John Wiley & Sons, 2003.
- Wickham, H. 2016. Tidy data. https://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html
- Wickham, H. 2016. Introduction to dplyr. https://cran.r-project.org/web/packages/dplyr/vignettes/introduction.html
- RStudio. https://www.rstudio.com/
- Robinson, D. 2016. Introduction to broom. https://cran.r-project.org/web/packages/broom/vignettes/broom.html
- Wilson, Greg, et al. "Best practices for scientific computing." PLoS Biol 12.1 (2014): e1001745. http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745
- Zuur, Alain F., Elena N. Ieno, and Chris S. Elphick. "A protocol for data exploration to avoid common statistical problems." Methods in Ecology and Evolution 1.1 (2010): 3-14. http://onlinelibrary.wiley.com/doi/10.1111/j.2041-210X.2009.00001.x/abstract