

Introdução à Ciência de Dados com R

Saulo Guerra e Paulo Felipe

2018-04-12

Conteúdo

Prefácio	7
1 Introdução	9
1.1 O que é Ciência de Dados?	9
1.2 Workflow da Ciência de Dados	9
1.3 Linguagens para Ciência de dados	10
1.4 O que é R e por quê devo aprender?	10
1.5 RStudio	10
1.6 Buscando Ajuda	12
1.7 Exercícios	12
2 Conceitos Básicos	13
2.1 Console	13
2.2 Scripts	13
2.3 Objetos (Variáveis)	14
2.4 Funções	15
2.5 Pacotes	16
2.6 Boas práticas	17
2.7 Tidyverse	17
2.8 Exercícios	17
3 Lendo os dados	19
3.1 Tipos de Estrutura dos Dados	19
3.2 Definindo o Local dos Dados	20
3.3 Pacote para leitura dos dados	20
3.4 Exercícios:	23
4 Manipulando os dados	25
4.1 Tipos de Variáveis e Colunas	25
4.2 If e Else	32
4.3 Loops	33
4.4 Manipulações com R base	36
4.5 Pacote dplyr	39
4.6 Exercícios	43
5 Limpando dados	45
5.1 O formato “ideal” dos dados	45
5.2 Pacote tidyverse	48
5.3 Manipulação de texto	51
5.4 Exercícios	53
6 Juntando dados	55
6.1 União de dados (Union)	55

6.2	Cruzamento de Dados (Join)	55
6.3	Exercícios	63
7	Escrevendo dados	67
7.1	Escrevendo csv	67
7.2	Rdata	67
7.3	Escrevendo outros tipos de arquivos	68
7.4	Exercícios	68
8	Obtendo dados	69
8.1	API	69
8.2	Web Scrapping	71
8.3	Exercícios	71
9	Visualizações de dados (ggplot2)	73
9.1	Mapeamento Estético	76
9.2	Objetos geométricos	77
9.3	Escalas	78
9.4	Subplots (facet)	92
9.5	Temas	97
9.6	Legendas	103
9.7	Escolhendo o tipo de gráfico	105
9.8	Gráfico de Dispersão (<code>geom_point()</code>)	105
9.9	Gráficos de Bolhas	111
9.10	Gráficos de Barras	112
9.11	Gráficos de linhas	119
9.12	Histogramas e freqpoly	121
9.13	Boxplots, jitterplots e violinplots	124
9.14	Anotações	129
9.15	Cleveland Dot Plot	131
9.16	Textos/Rótulos	135
9.17	Plotando funções	138
9.18	Mapas	141
9.19	Salvando Gráficos	147
9.20	Extensões do ggplot2	148
9.21	Exercícios	150
10	Visualizações Interativas	153
10.1	Introdução	153
10.2	Plotly	153
10.3	dygraphs	154
10.4	Leaflet	158
10.5	Exercícios	165
11	RMarkdown	167
11.1	Usos do RMarkdown	167
11.2	Estrutura de um RMarkdown	168
11.3	Renderizando um documento	170
11.4	Sintaxe	170
11.5	Opções de Chunk	173
11.6	Principais Formatos	174
11.7	Excercícios	177
12	Modelos	181
12.1	Modelo Linear	181

12.2 Exercícios	188
13 Revisão - Titanic	189
13.1 Objetivo	189
13.2 Carregando os Dados	189
13.3 Manipulando os dados	190
13.4 Idade	192
13.5 Visualizações	192
13.6 Modelo Preditivo	196
14 Referências	197

Prefácio

Esse livro foi criado por Saulo Guerra e Paulo Felipe para o curso Introdução à Ciéncia de Dados com R oferecido pelo IBPAD. Para construção desse livro, foi utilizado o Bookdown. Em alguns trechos, este livro foi baseado no material produzido por Robert McDonnell.



Capítulo 1

Introdução

1.1 O que é Ciência de Dados?

Trata-se de um termo cada vez mais utilizado para designar uma área de conhecimento voltada para o estudo e a análise de dados, onde se busca extrair conhecimento e criar novas informações. É uma atividade interdisciplinar que concilia principalmente duas grandes áreas: ciência da computação e estatística. A ciência de dados vem sendo aplicada como apoio em diferentes outras áreas de conhecimento, tais como: medicina, biologia, economia, publicidade, ciência política, etc. Apesar de não ser uma área nova, o tema vem se popularizando cada vez mais graças à explosão na produção de dados e crescente dependência dos dados para tomada de decisão.

1.2 Workflow da Ciência de Dados

Não existe apenas uma forma de estruturar e aplicar os conhecimentos da ciência de dados. A forma de aplicação vai variar bastante conforme a necessidade do projeto ou do objetivo que se busca alcançar. Neste curso utilizaremos um modelo de workflow bastante utilizado, apresentado no livro R for Data Science (Hadley Wickham, 2017).

Esse workflow propõe basicamente os seguintes passos:

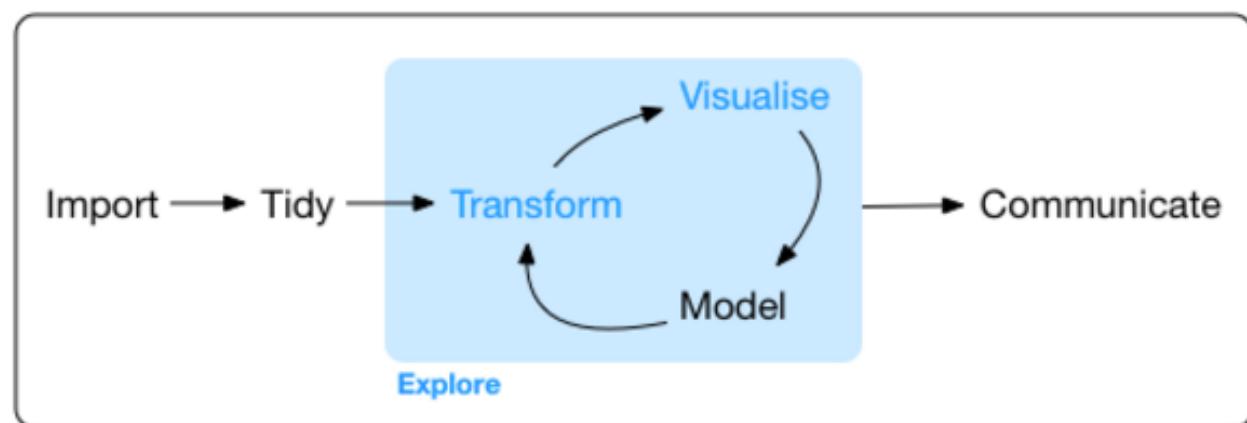


Figura 1.1: Workflow básico para ciência de dados

- Carregar os dados
- Limpar os dados
- Transformar, visualizar e modelar (fase exploratória)
- Comunicar o resultado

1.3 Linguagens para Ciência de dados

Para aplicação dessas atividades comuns da ciência de dados, é necessário dominar as ferramentas corretas. Existem diversas linguagens/ferramentas: R, Python, SAS, SQL, Matlab, Stata, Aplicações de BI, etc.

Cabe ao cientista de dados avaliar qual a ferramenta mais adequada para alcançar seus objetivos.

1.4 O que é R e por quê devo aprender?

R é uma linguagem de programação estatística que vem passando por diversas evoluções e se tornando cada vez mais uma linguagem de amplos objetivos. Podemos entender o R também como um conjunto de pacotes e ferramentas estatísticas, munido de funções que facilitam sua utilização desde a criação de simples rotinas até análises de dados complexas com visualizações bem acabadas.

Segue alguns motivos para aprender R:

- É completamente gratuito e de livre distribuição
- Curva de aprendizado bastante amigável, sendo muito fácil aprender
- Enorme quantidade de tutoriais e ajuda disponíveis gratuitamente na internet
- É excelente para criar rotinas e sistematizar tarefas repetitivas
- Amplamente utilizado pela comunidade acadêmica e pelo mercado
- Quantidade enorme de pacotes para diversos tipos de necessidades
- Ótima ferramenta para criar relatórios e gráficos

Apenas para exemplificar sua versatilidade, esse eBook e os slides das aulas foram todos feitos em R.

1.5 RStudio

O R puro se apresenta como uma simples “tela preta” com uma linha para inserir comandos. Isso é bastante assustador para quem está começando e bastante improdutivo para quem já faz uso intensivo da ferramenta. Felizmente existe o RStudio, ferramenta auxiliar que usaremos durante todo o curso.

Entenda o RStudio como uma interface gráfica com diversas funcionalidades que melhoram ainda mais o uso e aprendizado do R. Na prática, o RStudio facilita muito o dia a dia de trabalho. Portanto, desde já, ao falarmos em R, falaremos automaticamente no RStudio.

Essa é a “cara” do R studio:

Repare que, além da barra de menu superior, o RStudio é dividido em 4 partes principais:

1. Editor de Código

No editor de código, você poderá escrever e editar os scripts. Script nada mais é do que uma sequência de comandos/ordens que serão executados em sequência pelo R. O editor do RStudio oferece facilidades como organização dos comandos, “auto-complete” de comandos, destaque da sintaxe dos comandos, etc. Provavelmente é a parte que mais utilizaremos.

2. Console

É no console que o R irá mostrar a maioria dos resultados dos comandos. Também é possível escrever os comandos diretamente no console, sem uso do editor de código. Muito utilizado para testes e

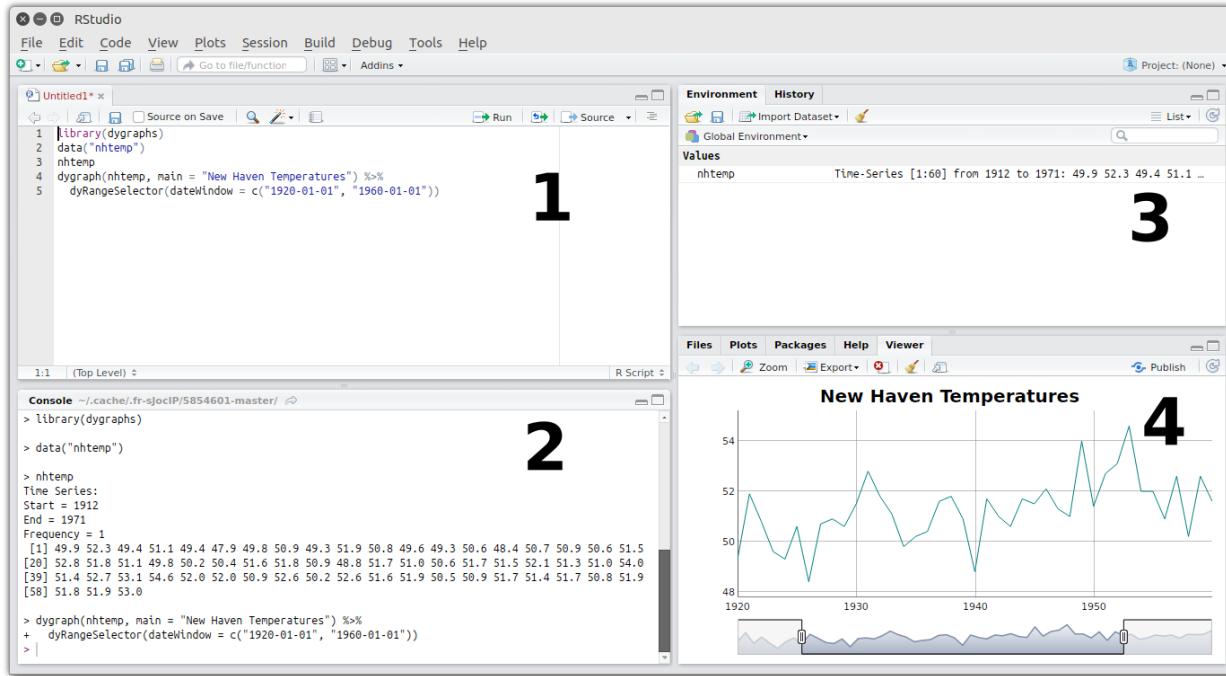


Figura 1.2: RStudio se divide em 4 partes

experimentos rápidos. Por exemplo, um uso rápido do console é chamar a ajuda do R usando o comando `?` (isso mesmo, a interrogação é um comando!). Voltaremos a falar desse comando `?` em breve..

3. Environment e History

No Environment ficarão guardados todos os objetos que forem criados na sessão do R. Entenda sessão como o espaço de tempo entre o momento em que você inicia o R e o momento que finaliza. Nesse período, tudo que você faz usa memória RAM e o processador do computador. E na aba History, como você deve imaginar, o RStudio cria um histórico de comandos utilizados.

4. Files, Plots, Packages, Help e Viewer

Nessa janela, estão várias funcionalidades do RStudio. Na aba Files, você terá uma navegação de arquivos do seu computador. Também será possível definir o diretório de trabalho (você também pode definir diretamente no código, mas isto será tratado posteriormente), ou seja, o R entende o seu diretório de trabalho como ponto de partida para localizar arquivos que sejam chamados no script.

4.1 Aba Plots

A aba Plots trará os gráficos gerados, possibilitando a exportação para alguns formatos diferentes, como png e pdf.

4.2 Aba Packages

Em Packages, estão listados os pacotes que estão instalados e você pode verificar quais estão carregados e, caso necessário, poderá carregar algum pacote necessário para a sua análise. Também é possível instalar e atualizar pacotes. Novamente, tudo isso é possível fazer diretamente no código.

4.3 Aba Help

Help o nome já diz tudo. Essa aba será bastante utilizada por você. Saber usar o help é fundamental para evitar desperdício de tempo. Os usuários de R, em geral, são bastante solícitos. Entretanto, uma olhadinha rápida no help pode evitar que você gaste “créditos” desnecessariamente.

4.4 Aba Viewer

Por fim, o Viewer. Essa funcionalidade é utilizada para visualizar localmente conteúdo web. O gráfico da figura está na aba Viewer porque é uma visualização em javascript, que pode ser adicionada a documentos htmls gerados usando o RMarkdown ou em aplicações web com suporte do Shiny.

1.6 Buscando Ajuda

Independente do seu nível de conhecimento, sempre haverá a necessidade de buscar ajuda. Ou seja, saber procurar ajuda é essencial para aprimorar seus conhecimentos em ciência de dados. Segue algumas dicas de como buscar ajuda para sanar dúvidas e resolver problemas em R:

- Sempre procure em inglês, apesar da comunidade R em língua portuguesa ser bem grande, a de língua inglesa é maior ainda. É muito provável que seus problemas e dúvidas já tenham sido sanados.
- Explore bem o help do próprio R.
- Conheça e aprenda a usar o stack overflow, a maior comunidade de ajuda técnica da internet.

Se mesmo explorando todas as dicas acima não conseguir resolver seu problema, procure por fóruns específicos.

Se você for realizar uma pergunta em algum fórum ou site de perguntas e respostas, é importante atentar para alguns pontos que deverão ser informados para que fique mais fácil de alguém te ajudar:

- Versão do R que está usando.
- Sistema Operacional.
- Forneça um exemplo replicável.
- Veja se a sua dúvida já não foi abordada em outro tópico.

1.7 Exercícios

1. Digite o comando `R.Version()`. O que acontece?
2. Encontre o item de menu `Help` e descubra como identificar a versão do seu RStudio.
3. Encontre o item de menu `Cheatsheets`. O que esse menu oferece?
4. Entre no site <https://stackoverflow.com> e digite `[r]` na caixa de busca. O que acontece?

Capítulo 2

Conceitos Básicos

Entenda o R como uma grande calculadora científica cheia de botões, mas, ao invés de apertar os botões, você irá escrever os comandos. Ou seja, “aprender R” significa se familiarizarizar com os comandos e saber quando usá-los. Todos os comandos são baseados em inglês e seus nomes normalmente dão dicas do seu uso.

2.1 Console

O console é uma das 4 partes principais do RStudio. Lá é onde você vai digitar suas ordens (comandos) e é onde o R vai “responder”. Para que o R possa interpretar correntemente, será necessário que você conheça a sintaxe da linguagem e a escrita correta dos comandos.

Olhando para o console você irá ver o símbolo `>`. Esse símbolo indica a linha onde você deve inserir os comandos. Clique nesse símbolo para posicionar o cursor na linha de comandos e digite seu primeiro comando em R: `2 * 3`. Digite e aperte *enter*. Você verá o seguinte resultado:

```
## [1] 6
```

Além de outras funcionalidades mais interessantes, o R é como uma grande calculadora científica. Para entender esse conceito melhor, vamos exercitar um pouco no console de comandos. Digite um por um dos seguintes comandos e acompanhe os resultados:

```
7 * 9 + 2 * 6  
2.5 * 4  
(50 + 7)/(8 * (3 - 5/2))  
3 ^ 4
```

Repare que, na medida em que for executando os comandos, o R vai respondendo. Esse é o comportamento básico do console. Muito utilizado para obter resultados rápidos de comandos específicos.

Um boa dica é que o R guarda um histórico dos comandos que você executou por último, basta apertar a seta para cima no teclado e ver os comandos executados anteriormente.

2.2 Scripts

Enquanto no console seus comandos são executados na medida que você os “envia” com o *enter*, em um script você ordena a execução de uma sequência de comandos escritos previamente, um seguido do outro. Scripts são escritos no editor de códigos do RStudio. Para entender melhor, localize o editor de códigos no RStudio e copie os mesmos comandos anteriores executados no console. No editor de códigos, a ordem para

execução dos comandos não é o *enter*. Para executá-los, clique em **Source**, no canto superior direito da área do editor de códigos. Repare bem pois há uma setinha escura que revelará duas opções de *Source* (execução do script): **Source**, e **Source With Echo**. A diferença entre as duas opções é que a primeira executa mas não exibe as respostas no console, a segunda executa mostrando as respostas. A primeira opção será útil em outros casos de scripts muito grandes ou outras situações que não convém “poluir” o console com um monte de mensagens.

Há um atalho de teclado para o **Source**: *ctrl + shift + enter*. Aprenda esse atalho, pois você usará muito mais o editor de códigos do que o console para executar os passos da sua análise.

Agora posicione o cursor do mouse com um clique em apenas um dos comandos do seu script. Em seguida, clique no ícone **Run** também no canto superior na área do editor de códigos. Repare que dessa vez o R executou apenas um comando, o que estava na linha selecionada. Esse tipo de execução também é bastante útil, mas esteja atento, pois é muito comum que comandos em sequência dependam da execução de comandos anteriores para funcionar corretamente.

Há um atalho de teclado para o **Run**: *ctrl + enter*. Aprenda esse atalho, ele também é muito importante e muito utilizado.

2.2.1 Salvando Scripts

Ao digitar seus comandos no console, o máximo que você consegue recuperar são os comandos anteriores usando a seta para cima. Já no editor de códigos, existe a possibilidade de salvar os seus scripts para continuar em outro momento ou em outro computador, preservar trabalhos passados ou compartilhar seus códigos com a equipe.

Um script em R tem a extensão (terminação) **.R**. Se você tiver o RStudio instalado e der dois clicks em um arquivo com extensão **.R**, o windows abrirá direto no RStudio.

Ainda utilizando os comandos digitados no editor de códigos, vá em **File > Save**. Escolha um local e um nome para seu script e confirme no botão **Save**. Lembre-se sempre de ser organizado utilizando pastas para os diferentes projetos e nomes explicativos. Para salvar mais rapidamente, utilize o atalho **ctrl + S**

2.2.2 Comentários de Código

Ao utilizar o símbolo **#** em uma linha, você está dizendo para o R ignorar aquela linha, pois trata-se de um comentário.

Clique na primeira linha do seu script, aperte *enter* para adicionar uma linha a mais e digite **# Meu primeiro comentário de código!**. Repare que a cor do comentário é diferente. Execute novamente seu script com o Source (*ctrl + shift + enter*) e veja que nada mudou na execução. A título de experimento, retire o símbolo **#** e mantenha o texto do comentário. Execute novamente. O R tenta interpretar essa linha como comando. Já que ele não conseguiu entender, será exibida uma mensagem de erro no console.

O símbolo de comentário também é muito útil para suprimir linhas de código para testar determinados comportamentos. Para exemplificar, adicione o símbolo **#** em qualquer uma das linhas com as operações e veja que ela não será mais executada, será ignorada pois foi entendida pelo R como um simples comentário de código.

2.3 Objetos (Variáveis)

Para que o R deixe de ser uma simples calculadora, será necessário aprender, dentre outras coisas, o uso de variáveis. Se você já tem alguma noção de estatística, provavelmente já tem a intuição do que vem a ser uma variável para uma linguagem de programação. No contexto do R, vamos entender variável como um

objeto, ou seja, como uma estrutura pré definida que vai “receber” algum valor. Sendo mais técnico, objeto (ou variável) é um pequeno espaço na memória do seu computador onde o R vai armazenar um valor ou resultado de um comando utilizando um nome que você mesmo definiu.

Conhecer os tipos de objetos do R é fundamental. Para criar objetos se utiliza o símbolo `<-`. Esse provavelmente é o símbolo que você mais vai ver daqui para frente.

Execute, no console ou no editor de códigos, o seguinte comando `x <- 15`. Pronto. Agora o nome `x` representa o valor 15. Para comprovar execute apenas o nome do objeto `x`, o R irá mostrar o conteúdo dele. A partir de então, você poderá utilizar esse objeto como se fosse o valor 15. Experimente os seguintes resultados:

```
x + 5
x * x / 2
2 ^ x
y <- x / 3
```

De uma boa lida em *Dicas e boas práticas para um código organizado* para aprender a organizar seus objetos e funções da melhor maneira possível.

Todos objetos que você criar estarão disponíveis na aba ***Environment***.

O RStudio possui a função de *auto complete*. Digite as primeiras letras de um objeto (ou função) que você criou e em seguida use o atalho ***ctrl + barra de espaço***. O RStudio irá listar tudo que começar com essas letras. Selecione alguma e aperte *enter* para escrevê-la no editor de códigos.

2.4 Funções

Entenda função como uma sequência de comandos preparados para serem usados de forma simples e facilitar sua vida. Funções são usadas para tudo que você possa imaginar: cálculos mais complexos, estatística, análise de dados, manipulação de dados, gráficos, relatórios, etc. Assim que você instala o R ele já vem com várias funções prontas para uso. A partir de agora chamaremos esse conjunto de funções que já vem por padrão com o R de *R Base*.

Claro que as funções do R base não serão suficientes para resolver todos os problemas que você vai encontrar pela frente. Nesse sentido o R também mostra outro ponto forte. Você pode instalar conjuntos extras de funções mais específicas de maneira muito simples: usando pacotes.

Funções do R base.

```
raiz.quadrada <- sqrt(16) # função para calcular raiz quadrada

round(5.3499999, 2) # função para arredondamento
```

Uma função tem dois elementos básicos: o nome e o(s) parâmetro(s) (também chamados de argumentos). Por exemplo, a função `log(10)` possui o nome `log()` e apenas um parâmetro que é o número que você quer calcular o log. Já a função `round()` possui dois parâmetros, o número que você quer arredondar e a quantidade de dígitos para arredondamento.

Quando você usa uma função, você pode informar os parâmetros de duas formas: sequencialmente sem explicitar o nome dos parâmetros, ou na ordem que quiser explicitando o nome dos parâmetros. Veja o exemplo a seguir:

```
round(5.3499999, 2)
# o mesmo que:
round(digits = 2, x = 5.3499999)
```

Para saber como informar os parâmetros corretamente, utilize o comando `?` (ou coloque o cursor no nome da função e pressione F1) para ver a documentação de funções, ou seja, conhecer para que ela serve, entender

cada um dos seus parâmetros e ver exemplos de uso.

```
?round
?rnorm
??inner_join # procurar ajuda de funções que não estão "instaladas" ainda
```

Vale comentar que é possível informar objetos nos parâmetros das funções.

```
x <- 3.141593
round(x, 3)

## [1] 3.142
ceiling(x)

## [1] 4
floor(x)

## [1] 3
```

Observe algumas das principais funções para estatísticas básicas no R:

Função R	Estatística
<code>sum()</code>	Soma de valores
<code>mean()</code>	Média
<code>var()</code>	Variancia
<code>median()</code>	Mediana
<code>summary()</code>	Resumo Estatístico
<code>quantile()</code>	Quantis

2.5 Pacotes

Como dito antes, pacotes são conjuntos extras de funções que podem ser instalados além do R base. Existe pacotes para auxiliar toda linha de estudo você imaginar: estatística, econometria, ciências sociais, medicina, biologia, gráficos, machine learning, etc.

Caso você precise de algum pacote mais específico, procure no google pelo tema que você precisa, encontre o nome do pacote e instale normalmente.

Nesse link você pode ver uma lista de todos os pacotes disponíveis no repositório central. Além desses, ainda existe a possibilidade de instalar pacotes “não oficiais”, que ainda não fazem parte de um repositório central.

Para instalar um pacote, execute o seguinte comando:

```
install.packages("dplyr") # instala um famoso pacote de manipulação de dados
```

Uma vez instalado, esse pacote estará disponível para uso sempre que quiser, sem necessidade de instalar novamente. Mas sempre que iniciar um código novo, você precisará carregá-lo na memória. Para isso, use o seguinte comando:

```
library(dplyr)
```

Para instalar um pacote você precisa informar o nome entre aspas `install.packages("readxl")`, caso contrário não vai funcionar. Porém, para carregar o pacote em memória, você pode usar com ou sem aspas `library(readxl)` ou `library("readxl")`, ambas as formas funcionam.

2.6 Boas práticas

Rapidamente você irá perceber que quanto mais organizado e padronizado você mantiver seus códigos, melhor para você e para sua equipe.

Existem dois guias de boas práticas bastante famosos na comunidade do R. Um sugerido pelo Hadley Wickham e outro por uma equipe do google.

Dentre as dicas de boa prática, algumas são as mais importantes, como por exemplo, não use acentos e caracteres especiais. Outro ponto importante, o R não aceita variáveis que comecem com números. Você pode até usar números no meio do nome, mas nunca começar com números.

O principal de tudo é, seja qual for o padrão que você preferir, use um padrão e seja consistente.

- Guia sugerido Hadley Wickham: <http://adv-r.had.co.nz/Style.html>
- _ Guia sugerido pelo Google: <https://google.github.io/styleguide/Rguide.xml>

2.7 Tidyverse

Como já dito, eventualmente as funções do R base não é suficiente ou simplesmente não fornecem maneira mais fácil de resolver um problema. Nesse curso utilizaremos o Tidyverse: uma coleção de pacotes R cuidadosamente desenhados para atuar no workflow comum da ciência de dados: importação, manipulação, exploração e visualização de dados. Uma vez carregado, esse pacote disponibiliza todo o conjunto de ferramentas de outros pacotes importantes: ggplot2, tibble, tidy, readr, purrr e dplyr. Oportunamente detalharemos cada um deles.

O Tidyverse foi idealizado, dentre outros responsáveis, por Hadley Wickham, um dos maiores colaboradores da comunidade R. Se você não o conhece e pretende seguir em frente com o R, certamente vai ouvir falar muito dele. Recomendamos segui-lo nas redes sociais para ficar por dentro das novidades do Tidyverse.

2.8 Exercícios

1. Quais as principais diferenças entre um script e o console?
2. Digite `?dplyr` o que acontece? E se digitar `??dplyr`? para que serve esse pacote?
3. Para que serve a função `rnorm()`? Quais os seus parâmetros/atributos?
4. Para que serve a função `rm()`? Quais os seus parâmetros/atributos?

Capítulo 3

Lendo os dados

Após o entendimento do problema/projeto que irá resolver com ciência de dados será necessário fazer com que o R leia os dados. Seja lá qual for o assunto do projeto, é muito importante garantir uma boa fonte de dados. Dados ruins, inconsistentes, não confiáveis ou mal formatados podem gerar muita dor de cabeça para o analista.

3.1 Tipos de Estrutura dos Dados

Os dados podem ser apresentados de diversas maneiras, não existe um padrão único para difusão ou divulgação. Sendo assim, é bom que você esteja preparado para lidar com qualquer tipo de estrutura de dados.

Existem diversas classificações de estrutura de dados. Vamos utilizar uma classificação mais geral que diz respeito a como os dados são disponibilizados. Sendo assim, podemos classificar os dados em 3 grandes tipos quanto à sua estrutura ou forma: dados estruturados, semiestruturados e não estruturados.

3.1.1 Dados Estruturados

Talvez seja o formato de dados mais fácil de se trabalhar no R. São conjuntos de informações organizadas em colunas (atributos, variáveis, features, etc.) e linhas (registros, itens, observações, etc.). São dados mais comumente encontrados diretamente em bancos de dados, arquivos com algum tipo de separação entre as colunas, Excel, arquivos com campos de tamanho fixo, etc.

3.1.2 Dados Não Estruturados

Como o nome diz, não tem um estrutura previsível, ou seja, cada conjunto e informações possui uma forma única. Geralmente são arquivos com forte teor textual. Não podemos dizer que são dados “desorganizados”, e sim que são organizações particulares para cada conjunto de informações. Podemos citar, por exemplo, emails, twitters, PDFs, imagens, vídeos, etc.

Analisar esse tipo de dado é muito mais complexo e exige conhecimento avançado em mineração de dados. Apesar disso, é o tipo de dados mais abundante na realidade.

3.1.3 Dados Semiestruturados

São dados que também possuem uma organização fixa, porém, não seguem o padrão de estrutura linha/coluna, ou seja, é uma estrutura mais complexa e flexível, geralmente hierárquica, estruturada em tags ou marcadores de campos. São exemplos de arquivos semiestruturados: JSON, XML, HTML, YAML, etc. É o formato mais usado em troca de dados pela internet e consumo de APIs (Application programming interface). Dados semiestruturados algumas vezes são facilmente transformados em dados estruturados.

3.2 Definindo o Local dos Dados

O R sempre trabalha com o conceito de *Working direcotry*, ou seja, uma pasta de trabalho onde ele vai “ler” e “escrever” os dados. Para verificar qual o diretório o R está “olhando”, utilize o seguinte comando:

```
getwd() #Get Working Directory
```

Para informar ao R em qual pasta ele deve ler os arquivos, utilizamos o comando *set working directory*, que muda o diretório padrão do R para leitura e escrita:

```
setwd('D:/caminho/do/arquivo/arquivo.csv')
```

3.3 Pacote para leitura dos dados

O R base possui funções para leitura dos principais tipos de arquivos. Um outro pacote específico e muito bom para isso é o **readr**. O Tidyverse inclui o carregamento do **readr**.

O R é capaz de ler diversos tipos de arquivos: csv (Comma-Separated Values), excel, arquivos separados por delimitadores, colunas de tamanho fixo, etc. Talvez o tipo de arquivo (estruturado) mais comum hoje em dia, e mais simples de trabalhar, seja o csv. Começaremos a importar dados com arquivos csv.

```
library(tidyverse) # já carrega o readr
#ou
library(readr)
```

Vamos importar um csv chamado **senado.csv**. Caso o arquivo esteja em seu working directory (`getwd()`), basta passar apenas o nome do arquivo para a função, caso contrário será necessário informar todo o caminho até a pasta do arquivo. Usamos o `read_csv()` para fazer isso.

```
senado <- read_csv("senado.csv")
```

Esse comando simplesmente carrega o conteúdo do arquivo **senado.csv** para o objeto (variável) **senado**. Após o carregamento, começaremos a investigar o conteúdo desse objeto: os dados.

O `head` e o `tail` são funções para ver a “cabeça” e o “rabo” dos seus dados, ou seja, o começo das amostras e fim das amostras. É muito importante sempre dar uma olhada na “cara” dos dados após o carregamento. Essa olhada ajuda a identificar erros básicos no carregamento, possibilitando ajustes o quanto antes, impedindo que esses erros se propaguem. Repare que na primeira linha temos os nomes das colunas e em seguida os registros.

```
head(senado)
```

```
## # A tibble: 6 x 15
##   VoteNumber  SenNumber    SenatorUpper  Vote Party GovCoalition State
##       <int>      <chr>          <chr> <chr> <chr>        <lgl> <chr>
## 1     2007001 PRS0002/07 FLEXA RIBEIRO      S  PSDB      FALSE    PA
## 2     2007001 PRS0002/07 ARTHUR VIRGILIO     S  PSDB      FALSE    AM
```

```

## 3 2007001 PRS0002/07 FLAVIO ARNS N PT TRUE PR
## 4 2007001 PRS0002/07 MARCELO CRIVELLA S PRB TRUE RJ
## 5 2007001 PRS0002/07 JOAO DURVAL N PDT FALSE BA
## 6 2007001 PRS0002/07 PAULO PAIM S PT TRUE RS
## # ... with 8 more variables: FP <int>, Origin <int>, Contentious <int>,
## # PercentYes <dbl>, IndGov <chr>, VoteType <int>, Content <chr>,
## # Round <int>

tail(senado)

## # A tibble: 6 x 15
##   VoteNumber SenNumber SenatorUpper Vote Party GovCoalition State
##       <int>      <chr>           <chr> <chr> <chr>    <lgl> <chr>
## 1 2010027 PLC0010/10 EDISON LOBAO     S  PMDB    TRUE   MA
## 2 2010027 PLC0010/10 EDUARDO SUPLICY  S  PT      TRUE   SP
## 3 2010027 PLC0010/10 JARBAS VASCONCELOS N  PMDB    TRUE   PE
## 4 2010027 PLC0010/10 MARISA SERRANO   S  PSDB    FALSE  MS
## 5 2010027 PLC0010/10 EPITACIO CAFETEIRA S  PTB     FALSE  MA
## 6 2010027 PLC0010/10 INACIO ARRUDA   S  PCdoB   TRUE   CE
## # ... with 8 more variables: FP <int>, Origin <int>, Contentious <int>,
## # PercentYes <dbl>, IndGov <chr>, VoteType <int>, Content <chr>,
## # Round <int>
```

Outros comandos muito importantes para começar a investigar os dados é o `str()` o `class()` e o `summary()`

Para verificar o tipo do objeto, ou seja, sua classe:

```
class(senado)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

Para verificar a estrutura do objeto, ou seja, seus campos (quando aplicável):

```
str(senado) #STRucture
```

```

## Classes 'tbl_df', 'tbl' and 'data.frame': 9262 obs. of 15 variables:
## $ VoteNumber : int 2007001 2007001 2007001 2007001 2007001 2007001 2007001 2007001 2007001 ...
## $ SenNumber  : chr "PRS0002/07" "PRS0002/07" "PRS0002/07" "PRS0002/07" ...
## $ SenatorUpper: chr "FLEXA RIBEIRO" "ARTHUR VIRGILIO" "FLAVIO ARNS" "MARCELO CRIVELLA" ...
## $ Vote       : chr "S" "S" "N" "S" ...
## $ Party      : chr "PSDB" "PSDB" "PT" "PRB" ...
## $ GovCoalition: logi FALSE FALSE TRUE TRUE FALSE TRUE ...
## $ State      : chr "PA" "AM" "PR" "RJ" ...
## $ FP         : int 2 2 2 2 2 2 2 2 2 ...
## $ Origin     : int 11 11 11 11 11 11 11 11 11 ...
## $ Contentious: int 0 0 0 0 0 0 0 0 0 ...
## $ PercentYes : num 85.5 85.5 85.5 85.5 85.5 ...
## $ IndGov     : chr "S" "S" "S" "S" ...
## $ VoteType   : int 1 1 1 1 1 1 1 1 1 ...
## $ Content    : chr "Creates the Senate Commission of Science, Technology, Innovation, Communication and Education"
## $ Round      : int 1 1 1 1 1 1 1 1 1 ...
## - attr(*, "spec")=List of 2
##   ..$ cols  :List of 15
##   ... .$ VoteNumber : list()
##   ... . .-$ SenNumber  : list()
##   ... . .-$ SenatorUpper: list()
```

```

## ... .- attr(*, "class")= chr "collector_character" "collector"
## ... $. Vote      : list()
## ... .- attr(*, "class")= chr "collector_character" "collector"
## ... $. Party     : list()
## ... .- attr(*, "class")= chr "collector_character" "collector"
## ... $. GovCoalition: list()
## ... .- attr(*, "class")= chr "collector_logical" "collector"
## ... $. State     : list()
## ... .- attr(*, "class")= chr "collector_character" "collector"
## ... $. FP        : list()
## ... .- attr(*, "class")= chr "collector_integer" "collector"
## ... $. Origin    : list()
## ... .- attr(*, "class")= chr "collector_integer" "collector"
## ... $. Contentious: list()
## ... .- attr(*, "class")= chr "collector_integer" "collector"
## ... $. PercentYes: list()
## ... .- attr(*, "class")= chr "collector_double" "collector"
## ... $. IndGov    : list()
## ... .- attr(*, "class")= chr "collector_character" "collector"
## ... $. VoteType   : list()
## ... .- attr(*, "class")= chr "collector_integer" "collector"
## ... $. Content    : list()
## ... .- attr(*, "class")= chr "collector_character" "collector"
## ... $. Round      : list()
## ... .- attr(*, "class")= chr "collector_integer" "collector"
## ... $. default: list()
## ... .- attr(*, "class")= chr "collector_guess" "collector"
## ... - attr(*, "class")= chr "col_spec"

```

Para verificar estatísticas básicas do objeto (média, mediana, quantis, mínimo, máximo... quando aplicável):

```
summary(senado)
```

```

##   VoteNumber      SenNumber      SenatorUpper
## Min.   :2007001  Length:9262       Length:9262
## 1st Qu.:2008006  Class  :character  Class  :character
## Median :2009003  Mode   :character  Mode   :character
## Mean   :2008483
## 3rd Qu.:2009048
## Max.   :2010027
## 
##   Vote          Party          GovCoalition      State
## Length:9262      Length:9262      Mode :logical  Length:9262
## Class :character Class :character FALSE:3480      Class :character
## Mode  :character Mode  :character TRUE :5782      Mode  :character
## 
## 
##   FP          Origin          Contentious      PercentYes
## Min.   :1.000  Min.   : 1.000  Min.   :0.00000  Min.   : 2.174
## 1st Qu.:2.000  1st Qu.: 1.000  1st Qu.:0.00000  1st Qu.: 66.667
## Median :2.000  Median : 2.000  Median :0.00000  Median : 96.078
## Mean   :1.878  Mean   : 2.595  Mean   :0.01781  Mean   : 82.509
## 3rd Qu.:2.000  3rd Qu.: 4.000  3rd Qu.:0.00000  3rd Qu.: 98.148
## Max.   :2.000  Max.   :11.000  Max.   :1.00000  Max.   :100.000
## 
##   IndGov      VoteType      Content      Round

```

```
##  Length:9262      Min.   :1.000  Length:9262      Min.   :1.000
##  Class :character 1st Qu.:1.000  Class :character 1st Qu.:1.000
##  Mode  :character Median :1.000  Mode  :character Median :1.000
##                           Mean   :1.159  Mean   :1.358
##                           3rd Qu.:1.000 3rd Qu.:2.000
##                           Max.   :2.000  Max.   :4.000
```

Acontece que nem sempre o separador será o ;, típico do csv. Nesse caso será necessário usar o `read_delim()`, onde você pode informar qualquer tipo de separador. Outro tipo de arquivo bastante comum é o de colunas com tamanho fixo (fixed width), ou também conhecido como colunas posicionais. Nesse caso será necessário usar o `read_fwf()` informando o tamanho de cada coluna.

Exemplo:

```
#lendo arquivo com delimitador #
read_delim('caminho/do/arquivo/arquivo_separado_por#.txt', delim = '#')

#lendo arquivo de coluna fixa
#coluna 1 de tamanho 5, coluna 2 de tamanho 2 e coluna 3 de tamanho 10
read_fwf('caminho/do/arquivo/arquivo_posicional.txt', col_positions = fwf_widths(c(5, 2, 10), c("col1",
```

No capítulo a seguir entenderemos melhor os tipos de objeto mais comuns no R.

3.4 Exercícios:

1. Leia o arquivo `TA_PRECOS_MEDICAMENTOS.csv` cujo separador é uma barra |.
2. Leia o arquivo de colunas fixas `fwf-sample.txt` cuja primeira coluna (nomes) tem tamanho 20, a segunda (estado) tem tamanho 10 e a terceira (codigo) tem tamanho 12.
3. Investigue os parâmetros das funções de leitura do R base: `read.csv()`, `read.delim()` e `read.fwf()`. Notou diferenças das funções do `readr`?

Capítulo 4

Manipulando os dados

Após obter uma boa fonte de dados e carregá-los para poder trabalhá-los no R, você certamente precisará realizar algumas limpezas e manipulações para que os dados estejam no ponto ideal para as fases finais de uma análise: execução de modelos econométricos, visualizações de dados, tabelas agregadas, relatórios, etc. A realidade é que na prática os dados nunca estarão do jeito que você de fato precisa. Portanto, é fundamental dominar técnicas de manipulação de dados.

Vamos entender a manipulação de dados como o ato de transformar, reestruturar, limpar, agregar, juntar dados. Para se ter uma noção da importância dessa fase, alguns estudiosos da área de ciência de dados costumam afirmar que 80% do trabalho é encontrar uma boa fonte de dados, limpar e preparar os dados, sendo que os 20% restantes seriam o trabalho de aplicar modelos e realizar alguma análise propriamente dita.

80% of data analysis is spent on the process of cleaning and preparing the data (Dasu and Johnson, 2003).

Data preparation is not just a first step, but must be repeated many over the course of analysis as new problems come to light or new data is collected (Hadley Wickham).

4.1 Tipos de Variáveis e Colunas

Existem diversos tipos de objetos, e cada tipo “armazena” um conteúdo diferente, desde tabelas de dados recém carregados, textos, números, ou simplesmente a afirmação de verdadeiro ou falso (booleano).

```
inteiro <- 928
outro.inteiro <- 5e2
decimal <- 182.93
caracter <- 'exportação'
logico <- TRUE
outro.logico <- FALSE
```

Repare nas atribuições acima. Vamos usar a função `class()` para ver o tipo de cada uma:

```
class(inteiro)

## [1] "numeric"
class(outro.inteiro)

## [1] "numeric"
```

```

class(decimal)

## [1] "numeric"

class(caracter)

## [1] "character"

class(logico)

## [1] "logical"

class(outro.logico)

## [1] "logical"

```

Esses são alguns dos tipos básicos de objetos/variáveis no R. `numeric` para valores inteiros ou decimais, `character` para valores textuais e `logical` para valores lógicos (verdadeiro ou falso). Ainda existe também o tipo `integer` que representa apenas números inteiros, sem decimais, porém, na maioria das vezes o R interpreta o `integer` como `numeric`, pois o `integer` também é um `numeric`.

Além dos tipos básicos, existem também os tipos “complexos”, que são `vector`, `array`, `matrix`, `list`, `data.frame` e `factor`.

Data frame é provavelmente o tipo de dado complexo mais utilizados em R. É nele que você armazena conjuntos de dados estruturados em linhas e colunas. Um data frame possui colunas nomeadas, sendo que todas as colunas possuem a mesma quantidade de linhas. Imagine o `dataframe` como uma tabela.

```

class(senado)

## [1] "tbl_df"     "tbl"        "data.frame"

dim(senado)

## [1] 9262    15

```

Outro tipo que já utilizamos bastante até agora, mas que não foi detalhado, é o `vector`, ou vetor. Vetores são sequências unidimensionais de valores de um mesmo tipo:

```

#faça as seguintes atribuições
vetor.chr <- c('tipo1', 'tipo2', 'tipo3', 'tipo4')
vetor.num <- c(1, 2, 5, 8, 1001)
vetor.num.repetidos <- c(rep(2, 50)) #usando função para repetir números
vetor.num.seq <- c(seq(from=0, to=100, by=5)) #usando função para criar sequências
vetor.logical <- c(TRUE, TRUE, TRUE, FALSE, FALSE)
##veja o conteúdo das variáveis
vetor.chr

## [1] "tipo1" "tipo2" "tipo3" "tipo4"
vetor.num

## [1] 1 2 5 8 1001
vetor.num.repetidos

## [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [36] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
vetor.num.seq

## [1] 0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80
## [18] 85 90 95 100

```

```
vetor.logical

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

Para criação de vetores, usamos a função de combinação de valores `c()` (combine). Essa função vai combinar todos os parâmetros em um único vetor. Lembre-se: vetores são sequências que contém apenas 1 tipo de dado.

Conhecendo o `data.frame` e o `vector` você será capaz de entender como os dois se relacionam. Cada coluna de um data frame é um vetor. Um data frame pode ter colunas de diferentes tipos, mas cada coluna só pode ter registros de um único tipo.

Ficará mais claro a seguir. Veja como se cria um `data.frame`:

```
#cria-se diferentes vetores
nome <- c('João', 'José', 'Maria', 'Joana')
idade <- c(45, 12, 28, 31)
adulito <- c(TRUE, FALSE, TRUE, TRUE)
uf <- c('DF', 'SP', 'RJ', 'MG')
#cada vetor é uma combinação de elementos de um MESMO tipo de dados
#sendo assim, cada vetor pode ser uma coluna de um data.frame
clientes <- data.frame(nome, idade, adulto, uf)
clientes

##      nome    idade  adulto   uf
## 1 João     45  TRUE  DF
## 2 José     12 FALSE  SP
## 3 Maria    28  TRUE  RJ
## 4 Joana    31  TRUE  MG

str(clientes)

## 'data.frame': 4 obs. of 4 variables:
## $ nome : Factor w/ 4 levels "Joana","João",...: 2 3 4 1
## $ idade : num 45 12 28 31
## $ adulto: logi TRUE FALSE TRUE TRUE
## $ uf    : Factor w/ 4 levels "DF","MG","RJ",...: 1 4 3 2
```

4.1.1 Conversões de tipos de variáveis

Quando é feito o carregamento de algum arquivo de dados no R, ele tenta “deduzir” os tipos de dados de cada coluna. Nem sempre essa dedução sai correta e eventualmente você precisará converter de um tipo para o outro. O R tem algumas funções para fazer essas conversões.

```
class("2015")

## [1] "character"
as.numeric("2015")

## [1] 2015
class(55)

## [1] "numeric"
as.character(55)

## [1] "55"
```

```

class(3.14)

## [1] "numeric"
as.integer(3.14)

## [1] 3
as.numeric(TRUE)

## [1] 1
as.numeric(FALSE)

## [1] 0
as.logical(1)

## [1] TRUE
as.logical(0)

## [1] FALSE

```

O R também tenta “forçar a barra” às vezes para te ajudar. Quando você faz uma operação entre dois tipos diferentes, ele tenta fazer algo chamado **coerção de tipos**, ou seja, ele tenta converter para que a operação faça sentido. Caso o R não consiga fazer a coerção, ele vai mostrar uma mensagem de erro.

Experimente os comandos a seguir no console:

```

7 + TRUE
2015 > "2016"
"2014" < 2017
# em alguns casos a coerção irá falhar ou dar resultado indesejado
6 > "100"
"6" < 5
1 + "1"

```

Recomendamos fortemente que sempre realize as conversões explicitamente com as funções apropriadas ao invés de confiar na coerção do R, a não ser que tenha certeza do resultado.

4.1.2 Outros tipos de variáveis

Existem outros tipos de variáveis também bastante utilizados. Vamos apenas citar alguns deles pois nesse curso utilizaremos muito pouco os demais tipos.

Tipo	Descrição	Dimensão	Homogêneo
vector	Coleção de elementos simétricos. Todos os elementos precisam ser do mesmo tipo e básico de dado.	1	Sim
array	Coleção que se parece com o vector, mas é multidimensional.	n	Sim
matrix	Tipo especial de array com duas dimensões.	2	Sim
list	Objeto com complexo com ele-mentos que podem ser diferentes tipos.	1	Não

Tipo	Descrição	Dimensão	Homogêneo
data.frame	especial de lista onde cada coluna é um vetor de apenas um tipo e todos as colunas têm o mesmo nú- mero de registros. É o tipo mais utilizado se trabalhar com dados	2	Não

Tipo	Descrição	Dimensão	Homogêneo
factor	<p>Tipo especial de vector que só contém valores pré-definidos (levels) e categoricos (characters). Não é possível adicionar novas categorias sem criação de novos levels</p>	1	Não

4.1.3 Valores faltantes e o ‘NA’

Em casos onde não existe valor em uma coluna de uma linha, o R atribui NA. É muito comum lidar com conjuntos de dados que tenham ocorrências de NA em alguns campos. É importante saber o que fazer em casos de NA, e nem sempre a solução será a mesma, vai variar de acordo com as suas necessidades.

Em algumas bases de dados, quem gera o dado atribui valores genéricos como 999 ou até mesmo um “texto vazio” ‘ ’. Nesse caso, você provavelmente terá que substituir esses valores “omissos” por NA. Imputar dados em casos de NA é uma das várias estratégias para lidar com ocorrência de missing no conjunto dos dados.

Seguem algumas funções úteis para lidar com NA.

- A função **summary()** pode ser usada para averiguar a ocorrência de NA.
- A função **is.na()** realiza um teste para saber se a variável/coluna possui um valor NA. retorna TRUE se for NA e FALSE se não for.
- A função **complete.cases()** retorna TRUE para as linhas em que todas as colunas possuem valores válidos (preenchidos) e FALSE para as linhas em que em alguma coluna existe um NA. Ou seja, essa função diz quais são as linhas (amostras) completas em todas suas características (campos).

- Algumas funções possuem o argumento `na.rm`, ou semelhantes, para desconsiderar NA no cálculo. É o caso da função `mean()` ou `sum()`.

Por exemplo:

```
data("airquality") # carrega uma base de dados pré carregada no R
```

```
summary(airquality) # verificando ocorrência de NA
```

```
##      Ozone          Solar.R          Wind          Temp
##  Min.   : 1.00   Min.   : 7.0   Min.   :1.700   Min.   :56.00
##  1st Qu.:18.00   1st Qu.:115.8  1st Qu.: 7.400  1st Qu.:72.00
##  Median :31.50   Median :205.0  Median : 9.700  Median :79.00
##  Mean   :42.13   Mean   :185.9  Mean   : 9.958  Mean   :77.88
##  3rd Qu.:63.25   3rd Qu.:258.8  3rd Qu.:11.500  3rd Qu.:85.00
##  Max.   :168.00  Max.   :334.0  Max.   :20.700  Max.   :97.00
##  NA's   :37     NA's   :7
##      Month         Day
##  Min.   :5.000   Min.   : 1.0
##  1st Qu.:6.000   1st Qu.: 8.0
##  Median :7.000   Median :16.0
##  Mean   :6.993   Mean   :15.8
##  3rd Qu.:8.000   3rd Qu.:23.0
##  Max.   :9.000   Max.   :31.0
##
##  is.na(airquality$Ozone)

##  [1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE FALSE
##  [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [23] FALSE FALSE TRUE TRUE TRUE FALSE FALSE FALSE FALSE TRUE TRUE
##  [34] TRUE TRUE TRUE TRUE FALSE TRUE FALSE FALSE TRUE TRUE FALSE
##  [45] TRUE TRUE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE
##  [56] TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE
##  [67] FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE
##  [78] FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE FALSE
##  [89] FALSE FALSE
##  [100] FALSE FALSE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
##  [111] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE
##  [122] FALSE FALSE
##  [133] FALSE FALSE
##  [144] FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
```

4.1.4 Estruturas de Controle de Fluxo

Para auxiliar no processo de manipulação de dados, você eventualmente precisará de algumas técnicas e estruturas de controle de fluxo. Estruturas para controle de fluxo nada mais são do que loops e condições. São estruturas fundamentais para qualquer linguagem de programação.

4.2 If e Else

A estrutura condicional é algo bastante intuitivo. A estrutura de `if (se)` e `else (então)` usa os operadores lógicos apresentados anteriormente. Se a condição do `if()` for verdadeira, executa-se uma tarefa específica, se for falsa, executa outra tarefa diferente. A estrutura parece com algo do tipo:

```
if( variavel >= 500 ) {
  #executa uma tarefa se operação resultar TRUE
} else {
  #executa outra tarefa se operação resultar FALSE
}
```

Da mesma forma existe uma função que gera o mesmo resultado, o `ifelse()` (e uma do pacote `dplyr` o `if_else()`).

```
ifelse(variavel >= 500, 'executa essa tarefa se TRUE', 'executa outra se FALSE')
```

Existe uma diferença entre as duas formas de `if else`: a estrutura `if() {} else {}` só opera variáveis, uma por uma, já a estrutura `ifelse()` opera vetores, ou seja, consegue fazer a comparação para todos os elementos. Isso faz com que a forma `if() {} else {}` seja mais usada para comparações fora dos dados, com variáveis avulsas, já a estrutura `ifelse()` é mais usada para comparações dentro dos dados, com colunas, vetores e linhas.

Qualquer uma dessas estruturas pode ser “aninhada”, ou seja, encadeada. Por exemplo:

```
a <- 9823
```

```
if(a >= 10000) {
  b <- 'VALOR ALTO'
} else if(a < 10000 & a >= 1000) {
  b <- 'VALOR MEDIO'
} else if(a < 1000) {
  b <- 'VALOR BAIXO'
}

b
```

```
## [1] "VALOR MEDIO"
```

Ou ainda:

```
a <- 839
c <- ifelse(a >= 10000, 'VALOR ALTO', ifelse(a < 10000 & a >= 1000, 'VALOR MEDIO', 'VALOR BAIXO'))
c

## [1] "VALOR BAIXO"
```

4.3 Loops

Trata-se de um dos conceitos mais importantes de qualquer linguagem de programação. Em R não é diferente. Loops (ou laços) repetem uma sequência de comando quantas vezes você desejar, ou até que uma condição aconteça, variando-se alguns aspectos entre uma repetição e outra.

Supondo que você tenha que ler 400 arquivos de dados que você obteve de um cliente. Você vai escrever 400 vezes a função de leitura? Nesse caso, basta fazer um loop para percorrer todos os arquivos da pasta e ir lendo um por um com a função de leitura.

4.3.1 For

O `for()` é usado para realizar uma série de ordens para uma determinada sequência ou índices (vetor). Sua sintaxe é bem simples:

```
for(i in c(1, 2, 3, 4, 5)) {
  print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

Para cada valor (chamamos esse valor de `i`) dentro do vetor `c(1, 2, 3, 4, 5)`, execute o comando `print(i^2)`. Qualquer outro comando dentro das chaves `{ ... }` seria executado para cada valor do vetor.

Para entender melhor, vamos repensar o exemplo das séries usando o `for()`.

```
lista.de.arquivos <- list.files('dados/dados_loop') #lista todos os arquivos de uma pasta
is.vector(lista.de.arquivos)
```

```
## [1] TRUE
for(i in lista.de.arquivos) {
  print(paste('Leia o arquivo:', i))
  #exemplo: read_delim(i, delim = "/")
}
```

```
## [1] "Leia o arquivo: arquivo1.txt"
## [1] "Leia o arquivo: arquivo10.txt"
## [1] "Leia o arquivo: arquivo11.txt"
## [1] "Leia o arquivo: arquivo12.txt"
## [1] "Leia o arquivo: arquivo13.txt"
## [1] "Leia o arquivo: arquivo2.txt"
## [1] "Leia o arquivo: arquivo3.txt"
## [1] "Leia o arquivo: arquivo4.txt"
## [1] "Leia o arquivo: arquivo5.txt"
## [1] "Leia o arquivo: arquivo6.txt"
## [1] "Leia o arquivo: arquivo7.txt"
## [1] "Leia o arquivo: arquivo8.txt"
## [1] "Leia o arquivo: arquivo9.txt"
```

Também é possível utilizar loop com `if`. No exemplo a seguir queremos ver todos os números, de 1 a 1000 que são divisíveis por 29 e por 3 ao mesmo tempo. Para isso utilizaremos o operador `%%` que mostra o resto da divisão. Se o resto for zero, é divisível.

```
for(i in 1:1000){
  if((i %% 29 == 0) & (i %% 3 == 0)){
    print(i)
  }
}
```

```
## [1] 87
## [1] 174
## [1] 261
## [1] 348
## [1] 435
## [1] 522
## [1] 609
## [1] 696
## [1] 783
```

```
## [1] 870
## [1] 957
```

4.3.2 While

O `while()` também é uma estrutura de controle de fluxo do tipo loop, mas, diferente do `for()` o `while` executa as tarefas repetidamente até que uma condição seja satisfeita e não percorrendo um vetor.

```
i <- 1
while(i <= 5){
  print(i)
  i <- i + 1
}
```

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

O uso do `while` é um pouco menos intuitivo mas não menos importante. O `while` é mais apropriado para eventos de automação ou simulação, onde tarefas serão executadas quando um “gatilho” for acionado. Um simples exemplo para ajudar na intuição de seu uso:

```
automatico <- list.files('dados/automatico/')
length(automatico) == 0
```

Temos uma pasta vazia. O loop abaixo vai monitorar essa pasta. Enquanto essa pasta estiver vazia, ele estará em execução. Quando você colocar um arquivo dentro dessa pasta, vai mudar a condição `length(automatico) == 0` de TRUE para FALSE e vai mudar a condição `length(automatico) > 0` de FALSE para TRUE, disparando todas as tarefas programadas. Usamos a função `Sys.sleep(5)` para o código esperar por mais 5 segundos antes de começar o loop novamente.

```
while (length(automatico) == 0) {
  automatico <- list.files('dados/automatico/')
  if(length(automatico) > 0) {
    print('0 arquivo chegou!')
    print('Inicia a leitura dos dados')
    print('Faz a manipulação')
    print('Envia email informando conclusão dos cálculos')
  } else {
    print('aguardando arquivo...')
    Sys.sleep(5)
  }
}
```

Faça o teste: execute o código acima, aguarde alguns segundos e perceba que nada aconteceu. Crie um arquivo qualquer dentro da pasta `dados/automatico/`. Imediatamente o loop será encerrado e as tarefas executadas. Observe o output em tela.

4.3.3 Funções

Funções “encapsulam” uma sequência de comandos e instruções. É uma estrutura nomeada que recebe parâmetros para iniciar sua execução e retorna um resultado ao final. Até o momento você já usou diversas funções. Veremos como criar uma função.

```
sua_funcao <- function(parametro1, parametro2){

  # sequência de tarefas

  return(valores_retornados)
}

# chamada da função
sua_funcao
```

Tente entender a seguinte função:

```
montanha_russa <- function(palavra) {
  retorno <- NULL
  for(i in 1:nchar(palavra)) {
    if(i %% 2 == 0) {
      retorno <- paste0(retorno, tolower(substr(palavra, i, i)))
    } else {
      retorno <- paste0(retorno, toupper(substr(palavra, i, i)))
    }
  }
  return(retorno)
}

montanha_russa('teste de função: letras maiúsculas e minúsculas')

## [1] "TeStE De fUnçãO: lEtRaS MaIúScUlAs e mInÚsCuLaS"
montanha_russa('CONSEGUIU ENTENDER?')

## [1] "CoNsEgUiU EnTeNdEr?"
montanha_russa('É Fácil Usar Funções!')

## [1] "É FáCiL UsAr fUnçõEs!"
```

4.4 Manipulações com R base

Dominar a manipulação de data frames e vetores é muito importante. Em geral, toda manipulação pode ser feita com o R base, mas acreditamos que utilizando técnicas do tidyverse a atividade fica bem mais fácil. Portanto, utilizaremos o `dplyr`, um dos principais pacotes do tidyverse.

Porém, alguns conceitos do R base são clássicos e precisam ser dominados.

4.4.1 Trabalhando com colunas de um data.frame

Para selecionar ou trabalhar separadamente com apenas um campo (coluna) do seu data.frame, deve-se utilizar o `$`. Repare nas funções abaixo e no uso do símbolo `$`.

```
head(airquality$Ozone)

## [1] 41 36 12 18 NA 28

tail(airquality$Ozone)
```

```
## [1] 14 30 NA 14 18 20
class(airquality$Ozone) # Informa o tipo da coluna

## [1] "integer"
is.vector(airquality$Ozone) # Apenas para verificar que cada coluna de um data.frame é um vetor

## [1] TRUE
unique(senado$Party) # Função que retorna apenas os valores únicos, sem repetição, de um vetor

## [1] "PSDB"      "PT"        "PRB"       "PDT"       "PR"        "PFL/DEM"   "PMDB"
## [8] "PP"         "PSB"       "PTB"       "PCdoB"     "PSOL"      "S/PART"    "PSC"
## [15] "PV1"
```

Lembre-se sempre: Cada coluna de um data.frame é um vetor, portanto todos os registros (linhas) daquela coluna devem ser do mesmo tipo. Um data.frame pode ser considerado um conjunto de vetores nomeados, todos do mesmo tamanho, ou seja, todos com a mesma quantidade de registros.

Usando termos mais técnicos, um data frame é um conjunto de dados HETEROGÊNEOS pois cada coluna pode ser de um tipo, e BIDIMENSIONAL pois possui apenas linhas e colunas. Já o vetor é um conjunto de dados HOMOGÊNEO pois só pode ter valores de um mesmo tipo, e UNIDIMENSIONAL.

Com esses conceitos em mente fica mais fácil entender o que mostraremos a seguir:

```
vetor <- c(seq(from=0, to=100, by=15)) #vetor de 0 a 100, de 15 em 15.
vetor #lista todos os elementos
```

```
## [1] 0 15 30 45 60 75 90
vetor[1] #mostra apenas o elemento na posição 1

## [1] 0
vetor[2] #apenas o elemento na posição 2

## [1] 15
vetor[7] #apenas o elemento na posição 7

## [1] 90
vetor[8] #não existe nada na posição 8...

## [1] NA
```

A notação [] é usada para selecionar o elemento em uma (ou mais) posição(ões) do vetor.

```
vetor[c(2,7)] #selecionando mais de um elemento no vetor
```

```
## [1] 15 90
```

Uma notação parecida é usada para selecionar elementos no data.frame. Porém, como já comentamos, data frames são BIDIMENSIONAIS. Então usaremos a notação [,] com uma vírgula separando qual a linha (antes da vírgula) e a coluna (após a vírgula) queremos selecionar.

```
senado[10, ] #linha 10, todas as colunas
```

```
## # A tibble: 1 x 15
##   VoteNumber  SenNumber SenatorUpper  Vote Party GovCoalition State     FP
##       <int>      <chr>          <chr> <chr> <chr>      <lgl> <chr> <int>
## 1    2007001  PRS0002/07     MAO SANTA      S  PMDB      TRUE    PI     2
## # ... with 7 more variables: Origin <int>, Contentious <int>,
```

```

## #   PercentYes <dbl>, IndGov <chr>, VoteType <int>, Content <chr>,
## #   Round <int>
senado[72, 3] #linha 72, coluna 3

## # A tibble: 1 x 1
##       SenatorUpper
##   <chr>
## 1 WELLINGTON SALGADO

senado[c(100, 200), c(2,3,4)] # selecionando mais de uma linha e coluna em um data.frame

## # A tibble: 2 x 3
##   SenNumber     SenatorUpper  Vote
##   <chr>          <chr>    <chr>
## 1 PLS0229/06  MARISA SERRANO   S
## 2 PLS0134/06  EPITACIO CAFETEIRA S

senado[c(10:20), ]

## # A tibble: 11 x 15
##   VoteNumber  SenNumber     SenatorUpper  Vote  Party GovCoalition
##   <int>        <chr>          <chr>    <chr>  <chr>      <lgl>
## 1 2007001    PRS0002/07    MAO SANTA      S    PMDB      TRUE
## 2 2007001    PRS0002/07    MAGNO MALTA    S    PR        TRUE
## 3 2007001    PRS0002/07    EDUARDO SUPLICY  S    PT        TRUE
## 4 2007001    PRS0002/07    GILVAM BORGES   S    PMDB      TRUE
## 5 2007001    PRS0002/07    RAIMUNDO COLOMBO S    PFL/DEM    FALSE
## 6 2007001    PRS0002/07    CICERO LUCENA   S    PSDB      FALSE
## 7 2007001    PRS0002/07    FRANCISCO DORNELLES S    PP        TRUE
## 8 2007001    PRS0002/07    OSMAR DIAS      N    PDT       FALSE
## 9 2007001    PRS0002/07    ALFREDO NASCIMENTO S    PR        TRUE
## 10 2007001   PRS0002/07    VALDIR RAUPP     S    PMDB      TRUE
## 11 2007001   PRS0002/07    GARIBALDI ALVES FILHO S    PMDB      TRUE
## # ... with 9 more variables: State <chr>, FP <int>, Origin <int>,
## #   Contentious <int>, PercentYes <dbl>, IndGov <chr>, VoteType <int>,
## #   Content <chr>, Round <int>

```

Repare na notação c(10:20), você pode usar : para criar sequências. Experimente 1:1000

Também é possível selecionar com o próprio nome da coluna:

```
senado[1:10, c('SenatorUpper', 'Party', 'State')]
```

```

## # A tibble: 10 x 3
##       SenatorUpper  Party State
##   <chr>          <chr> <chr>
## 1 FLEXA RIBEIRO   PSDB  PA
## 2 ARTHUR VIRGILIO PSDB  AM
## 3 FLAVIO ARNS     PT    PR
## 4 MARCELO CRIVELLA PRB   RJ
## 5 JOAO DURVAL    PDT   BA
## 6 PAULO PAIM      PT    RS
## 7 EXPEDITO JUNIOR PR    RO
## 8 EFRAIM MORAIS   PFL/DEM PB
## 9 ALOIZIO MERCADANTE PT    SP
## 10 MAO SANTA      PMDB  PI

```

Existem diversas outras formas de seleção e manipulação de dados, como por exemplo, seleção condicional:

```
head(senado[senado$Party == 'PDT', ])
```

```
## # A tibble: 6 x 15
##   VoteNumber SenNumber SenatorUpper Vote Party GovCoalition State
##       <int>     <chr>          <chr> <chr> <chr>      <lgl> <chr>
## 1    2007001 PRS0002/07 JOAO DURVAL     N  PDT      FALSE   BA
## 2    2007001 PRS0002/07 OSMAR DIAS      N  PDT      FALSE   PR
## 3    2007001 PRS0002/07 CRISTOVAM BUARQUE A  PDT      FALSE   DF
## 4    2007002 PLS0229/06   JOAO DURVAL     S  PDT      FALSE   BA
## 5    2007002 PLS0229/06   OSMAR DIAS      S  PDT      FALSE   PR
## 6    2007002 PLS0229/06 CRISTOVAM BUARQUE S  PDT      FALSE   DF
## # ... with 8 more variables: FP <int>, Origin <int>, Contentious <int>,
## #   PercentYes <dbl>, IndGov <chr>, VoteType <int>, Content <chr>,
## #   Round <int>
```

Em todas as comparações do R usamos operadores lógicos. São operações matemáticas em que o resultado ou é TRUE ou FALSE (tipo `logic`). Para entender melhor, seguem alguns operadores lógicos e seus significados:

- `==` igual a: compara dois objetos e se forem iguais, retorna TRUE, caso contrário, FALSE;
- `!=` diferente: compara dois objetos e se forem diferentes, retorna TRUE, caso contrário, FALSE;
- `|` ou (or): compara dois objetos, se um dos dois for TRUE, retorna TRUE, se os dois forem FALSE, retorna FALSE;
- `&` e (and): compara dois objetos, se os dois forem TRUE, retorna TRUE, se um dos dois ou os dois forem FALSE, retorna FALSE;
- `>, >=, <, <=` maior, maior ou igual, menor, menor ou igual: compara grandeza de dois números e retorna TRUE ou FALSE conforme a condição;

É possível fazer muita coisa com o R base, porém, vamos avançar com as manipulações utilizando o pacote `dplyr`, por ser mais simples e de mais rápido aprendizado.

4.5 Pacote dplyr

O forte do `dplyr` é a sintaxe simples e concisa, facilitando o aprendizado e tornando o pacote um dos preferidos para as tarefas do dia a dia. Também conta como ponto forte sua otimização de performance para manipulação de dados. Ao carregar o pacote `tidyverse` você já irá carregar automaticamente o pacote `dplyr`. Mas pode carregá-lo individualmente:

```
install.packages("dplyr")
library(dplyr)
?dplyr
```

4.5.1 Verbetes do dplyr e o operador %>%

O `dplyr` cobre praticamente todas as tarefas básicas da manipulação de dados: agregar, summarizar, filtrar, ordenar, criar variáveis, joins, dentre outras.

As funções do `dplyr` simplesmente reproduzem as principais tarefas da manipulação de forma bastante intuitiva. Veja só:

- `select()`
- `filter()`
- `arrange()`
- `mutate()`

- `group_by()`
- `summarise()`

Esses são os principais verbetes, mas existem outros disponíveis como, por exemplo, `slice()`, `rename()` e `transmute()`. Além de nomes de funções intuitivos, o `dplyr` também faz uso de um recurso disponível em boa parte dos pacotes do Hadley, o operador `%>%` (originário do pacote `magrittr`). Esse operador encadeia as chamadas de funções de forma que você não vai precisar ficar chamando uma função dentro da outra ou ficar fazendo atribuições usando diversas linhas para concluir suas manipulações. Aliás, podemos dizer que esse operador `%>%` literalmente cria um fluxo sequencial bastante claro e legível para todas as atividades de manipulação.

4.5.2 Select

O `select()` é a função mais simples de ser entendida. Ela é usada para selecionar variáveis (colunas, campos, features...) do seu data frame.

```
senadores.partido <- senado %>% select(SenatorUpper, Party)
head(senadores.partido)
```

```
## # A tibble: 6 x 2
##       SenatorUpper Party
##   <chr>     <chr>
## 1 FLEXA RIBEIRO  PSDB
## 2 ARTHUR VIRGILIO PSDB
## 3 FLAVIO ARNS    PT
## 4 MARCELO CRIVELLA PRB
## 5 JOAO DURVAL    PDT
## 6 PAULO PAIM     PT
```

Você pode também fazer uma “seleção negativa”, ou seja, escolher as colunas que não quer

```
senadores.partido <- senado %>% select(-SenatorUpper, -Party)
head(senadores.partido)
```

```
## # A tibble: 6 x 13
##   VoteNumber SenNumber Vote GovCoalition State    FP Origin Contentious
##   <int>      <chr>  <chr>    <lgl>  <chr>  <int>  <int>      <int>
## 1 2007001  PRS0002/07 S FALSE    PA    2    11      0
## 2 2007001  PRS0002/07 S FALSE    AM    2    11      0
## 3 2007001  PRS0002/07 N TRUE     PR    2    11      0
## 4 2007001  PRS0002/07 S TRUE     RJ    2    11      0
## 5 2007001  PRS0002/07 N FALSE    BA    2    11      0
## 6 2007001  PRS0002/07 S TRUE     RS    2    11      0
## # ... with 5 more variables: PercentYes <dbl>, IndGov <chr>,
## #   VoteType <int>, Content <chr>, Round <int>
```

4.5.3 Filter

Além de escolher apenas alguns campos, você pode escolher apenas algumas linhas utilizando alguma condição como filtragem. Para isso basta utilizar a função `filter`.

```
senadores.pdt.df <- senado %>%
  select(SenatorUpper, Party, State) %>%
  filter(State == 'RJ', Party == 'PMDB') %>%
  distinct() #semelhante ao unique(), traz registros únicos sem repetição
```

```
head(senadores.pdt.df)

## # A tibble: 2 x 3
##   SenatorUpper Party State
##   <chr>     <chr> <chr>
## 1 PAULO DUQUE  PMDB    RJ
## 2 REGIS FICHTNER PMDB    RJ
```

4.5.4 Mutate

Para criar novos campos, podemos usar o `mutate`:

```
senadores.pdt.df <- senado %>%
  select(SenatorUpper, Party, State) %>%
  filter(Party == 'PMDB') %>%
  distinct() #semelhante ao unique(), traz registros únicos sem repetição

head(senadores.pdt.df)

## # A tibble: 6 x 3
##   SenatorUpper Party State
##   <chr>     <chr> <chr>
## 1 MAO SANTA  PMDB    PI
## 2 GILVAM BORGES PMDB    AP
## 3 VALDIR RAUPP  PMDB    RO
## 4 GARIBALDI ALVES FILHO PMDB    RN
## 5 GERSON CAMATA PMDB    ES
## 6 JARBAS VASCONCELOS PMDB    PE
```

4.5.5 Group By e Summarise

O `group_by()` e o `summarise()` são operações que trabalham na agregação dos dados, ou seja, um dado mais detalhado passa a ser um dado mais agregado, agrupado, menos detalhado. Agrupamento de dados geralmente é trabalhado em conjunção com sumarizações, que usam funções matemáticas do tipo soma, média, desvio padrão, etc.

Enquanto o `group_by()` “separa” seus dados nos grupos que você selecionar, o `summarise()` faz operações de agregação de linhas limitadas a esse grupo.

Vale observar que operações de agrupamento e sumarização geralmente DIMINUEM a quantidade de linhas dos seus dados, pois está reduzindo o nível de detalhe. Ou seja, de alguma forma você está “perdendo” detalhe para “ganhar” agregação.

Como exemplo, utilizaremos os dados disponíveis no pacote `nycflights13` que disponibiliza

```
install.packages("nycflights13")
library(nycflights13)
data("flights")

str(flights)

## Classes 'tbl_df', 'tbl' and 'data.frame': 336776 obs. of 19 variables:
## $ year      : int 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
## $ month     : int 1 1 1 1 1 1 1 1 1 ...
## $ day       : int 1 1 1 1 1 1 1 1 1 ...
```

```
## $ dep_time      : int  517 533 542 544 554 554 555 557 557 558 ...
## $ sched_dep_time: int  515 529 540 545 600 558 600 600 600 600 ...
## $ dep_delay     : num  2 4 2 -1 -6 -4 -5 -3 -3 -2 ...
## $ arr_time      : int  830 850 923 1004 812 740 913 709 838 753 ...
## $ sched_arr_time: int  819 830 850 1022 837 728 854 723 846 745 ...
## $ arr_delay     : num  11 20 33 -18 -25 12 19 -14 -8 8 ...
## $ carrier       : chr  "UA" "UA" "AA" "B6" ...
## $ flight        : int  1545 1714 1141 725 461 1696 507 5708 79 301 ...
## $ tailnum       : chr  "N14228" "N24211" "N619AA" "N804JB" ...
## $ origin        : chr  "EWR" "LGA" "JFK" "JFK" ...
## $ dest          : chr  "IAH" "IAH" "MIA" "BQN" ...
## $ air_time      : num  227 227 160 183 116 150 158 53 140 138 ...
## $ distance      : num  1400 1416 1089 1576 762 ...
## $ hour          : num  5 5 5 5 6 5 6 6 6 ...
## $ minute         : num  15 29 40 45 0 58 0 0 0 0 ...
## $ time_hour     : POSIXct, format: "2013-01-01 05:00:00" "2013-01-01 05:00:00" ...
```

Gostaríamos de obter a média de atraso da chegada para cada mês. Primeiro agrupamos no nível necessário e depois sumarizamos.

```
media <- flights %>%
  group_by(month) %>%
  summarise(arr_delay_media = mean(arr_delay, na.rm=TRUE),
            dep_delay_media = mean(dep_delay, na.rm=TRUE))
```

```
media
```

```
## # A tibble: 12 x 3
##   month arr_delay_media dep_delay_media
##   <int>       <dbl>           <dbl>
## 1     1       6.1299720    10.036665
## 2     2       5.6130194    10.816843
## 3     3       5.8075765    13.227076
## 4     4      11.1760630    13.938038
## 5     5       3.5215088    12.986859
## 6     6      16.4813296    20.846332
## 7     7      16.7113067    21.727787
## 8     8       6.0406524    12.611040
## 9     9      -4.0183636     6.722476
## 10    10      -0.1670627     6.243988
## 11    11      0.4613474     5.435362
## 12    12      14.8703553    16.576688
```

4.5.6 Arrange

A função `arrange()` serve para organizar os dados em sua ordenação. Costuma ser uma das últimas operações, normalmente usada para organizar os dados e facilitar visualizações ou criação de relatórios. Utilizando o exemplo anterior, gostaríamos de ordenar os meses pelas menores médias de decolagem (para ordens decrescentes basta usar o sinal de menos -)

```
media <- flights %>%
  group_by(month) %>%
  summarise(arr_delay_media = mean(arr_delay, na.rm=TRUE),
            dep_delay_media = mean(dep_delay, na.rm=TRUE)) %>%
  arrange(dep_delay_media)
```

```
media

## # A tibble: 12 x 3
##   month arr_delay_media dep_delay_media
##   <int>        <dbl>          <dbl>
## 1     11      0.4613474    5.435362
## 2     10     -0.1670627    6.243988
## 3      9     -4.0183636    6.722476
## 4      1      6.1299720   10.036665
## 5      2      5.6130194   10.816843
## 6      8      6.0406524   12.611040
## 7      5      3.5215088   12.986859
## 8      3      5.8075765   13.227076
## 9      4     11.1760630   13.938038
## 10     12     14.8703553   16.576688
## 11      6     16.4813296   20.846332
## 12      7     16.7113067   21.727787
```

4.5.7 O operador %>%

Observe novamente as manipulações feitas acima. Repare que apenas fomos acrescentando verbetes e encadeando a manipulação com o uso de %>%.

A primeira parte `serie.orig %>%` é a passagem onde você informa o data.frame que você irá trabalhar na sequência de manipulação. A partir daí, as chamadas seguintes `select()` %>%, `filter()` %>%, `mutate()` %>% etc, são os encadeamentos de manipulação que você pode ir fazendo sem precisar atribuir resultados ou criar novos objetos.

Em outras palavras, usando o operador %>% , você estará informando que um resultado da operação anterior será a entrada para a nova operação. Esse encadeamento facilita muito as coisas, tornando a manipulação mais legível e intuitiva.

4.6 Exercícios

Utilizando os dados em `senado.csv`, tente usar da manipulação de dados para responder as perguntas a seguir.

1. Verifique a existência de registros NA em `State`. Caso existam, crie um novo data.frame `senado2` sem esses registros e utilize-o para os próximos exercícios. Dica: `is.na(State)`
2. Quais partidos foram parte da coalizão do governo? E quais não foram? Dica: `filter()`
3. Quantos senadores tinha cada partido? Qual tinha mais? Quais tinham menos? Dica: `group_by()`, `summarise()` e `n_distinct()`
4. Qual partido votou mais sim? E qual voltou menos sim? Dica: `sum(Vote == 'S')`
5. Qual região do país teve mais votos sim? Primeiro será necessário criar uma coluna região, para depois contabilizar o total de votos por região.

Dica: `mutate(Regiao = ifelse(State %in% c("AM", "AC", "TO", "PA", "RO", "RR"), "Norte", ifelse(State %in% c("SP", "MG", "RJ", "ES"), "Sudeste", ifelse(State %in% c("MT", "MS", "GO", "DF"), "Centro-Oeste", ifelse(State %in% c("PR", "SC", "RS"), "Sul", "Nordeste")))))`

Capítulo 5

Limpando dados

No dia a dia de quem trabalha com dados infelizmente é muito comum se deparar com dados formatados de um jeito bastante complicado de se manipular. Isso acontece pois a forma de se trabalhar com dados é muito diferente da forma de se apresentar ou visualizar dados. Resumindo: “olhar” dados requer uma estrutura bem diferente do que “mexer” com dados. Limpeza de dados também é considerada parte da manipulação de dados.

5.1 O formato “ideal” dos dados

É importante entender um pouco mais sobre como os dados podem ser estruturados antes de entrarmos nas funções de limpeza de dados. O formato ideal para analisar dados visualmente é diferente do formato ideal para analisar dados de forma sistemática. Observe as duas tabelas a seguir:

A primeira tabela é mais intuitiva para análise visual, pois faz uso de cores e propõe uma leitura natural, da esquerda para a direita. Ela usa elementos e estruturas que guiam seus olhos por uma análise de forma simples. Já a segunda tabela é um pouco árida para se interpretar “no olho”.

Há uma “regra geral” que diz que um dado bem estruturado deve conter uma única variável em uma coluna e uma única observação em uma linha.

Observando a primeira tabela com essa regra em mente, podemos perceber que as observações de ano estão organizadas em colunas. Apesar de estar num formato ideal para análise visual, esse formato vai dificultar bastante certas análises sistemáticas. O melhor a se fazer é converter a primeira tabela para um modelo mais próximo possível da segunda tabela.

Infelizmente não temos como apresentar um passo a passo padrão para limpeza de dados, pois isso vai depender completamente do tipo de dado que você receber, da análise que você quer fazer e da sua criatividade em manipulação de dados. Mas conhecer os pacotes certos ajuda muito nessa tarefa.

Lembre-se: é muito mais fácil trabalhar no R com dados “bem estruturados”, onde **cada coluna deve ser uma única variável** e **cada linha deve ser uma única observação**.

Na contramão da limpeza de dados, você provavelmente terá o problema contrário no final da sua análise. Supondo que você organizou seus dados perfeitamente, conseguiu executar os modelos que gostaria, gerou diversos gráficos interessantes e está satisfeita com o resultado. Você ainda precisará entregar relatórios finais da sua análise em forma de tabelas sumarizadas e explicativas, de modo que os interessados possam entender facilmente apenas com uma rápida análise visual. Nesse caso, que tipo de tabela seria melhor produzir? Provavelmente quem for ler seus relatórios entenderá mais rapidamente as tabelas mais próximas do primeiro exemplo mostrado.

É importante aprender a estruturar e desestruturar tabelas de todas as formas possíveis.

	2014		2015	
Produtos	US\$ FOB	Kg. Líquido	US\$ FOB	Kg. Líquido
A1	9.193	1.019.483	10.923	1.983.124
A2	8.381	2.003.984	9.819	2.839.218
A3	9.102	192.801	9.382	203.938
A4	7.181	3.093.029	8.192	3.183.902
Total da categoria A	33.857	6.309.297	38.316	8.210.182
B1	10.293	1.831	11.238	1.931
B2	9.839	2.938	10.928	3.823
B3	8.910	983	9.192	1.923
Total da categoria B	29.042	5.752	31.358	7.677

Figura 5.1: Tabela wide

PRODUTO	ANO	FOB	KG
A1	2014	9193	1019483
A1	2015	10923	1983124
A2	2014	8381	2003984
A2	2015	9819	2839218
A3	2014	9102	192801
A3	2015	9382	203938
A4	2014	7181	3093029
A4	2015	8192	3183902
B1	2014	10293	1831
B1	2015	11238	1931
B2	2014	9839	2938
B2	2015	10928	3823
B3	2014	8910	983
B3	2015	9192	1923

Figura 5.2: Tabela long

Para ilustrar, veja algumas tabelas disponíveis no pacote `tidyverse` ilustrando os diferentes tipos de organização nos formatos wide e long. Todas as tabelas possuem os mesmos dados e informações:

```
library(tidyverse)
```

```
table1
```

```
## # A tibble: 6 x 4
##       country   year   cases population
##       <chr>     <int>    <int>      <int>
## 1 Afghanistan 1999      745 19987071
## 2 Afghanistan 2000     2666 20595360
## 3      Brazil  1999    37737 172006362
## 4      Brazil  2000    80488 174504898
## 5      China   1999   212258 1272915272
## 6      China   2000   213766 1280428583
```

```
table2
```

```
## # A tibble: 12 x 4
##       country   year     type   count
##       <chr>     <int>    <chr>    <int>
## 1 Afghanistan 1999   cases      745
## 2 Afghanistan 1999 population 19987071
## 3 Afghanistan 2000   cases     2666
## 4 Afghanistan 2000 population 20595360
## 5      Brazil  1999   cases    37737
## 6      Brazil  1999 population 172006362
## 7      Brazil  2000   cases    80488
## 8      Brazil  2000 population 174504898
## 9      China   1999   cases   212258
## 10     China   1999 population 1272915272
## 11     China   2000   cases   213766
## 12     China   2000 population 1280428583
```

```
table3
```

```
## # A tibble: 6 x 3
##       country   year         rate
##       <chr>     <int>    <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3      Brazil  1999 37737/172006362
## 4      Brazil  2000 80488/174504898
## 5      China   1999 212258/1272915272
## 6      China   2000 213766/1280428583
```

```
table4a
```

```
## # A tibble: 3 x 3
##       country `1999` `2000`
##       <chr>    <int>   <int>
## 1 Afghanistan     745    2666
## 2      Brazil    37737   80488
## 3      China   212258  213766
```

```
table4b
```

```
## # A tibble: 3 x 3
```

```

##       country    `1999`    `2000`
## *     <chr>      <int>      <int>
## 1 Afghanistan 19987071 20595360
## 2      Brazil 172006362 174504898
## 3      China 1272915272 1280428583
table5

## # A tibble: 6 x 4
##       country century year      rate
##     <chr>     <chr> <chr>    <chr>
## 1 Afghanistan    19    99 745/19987071
## 2 Afghanistan    20    00 2666/20595360
## 3      Brazil     19    99 37737/172006362
## 4      Brazil     20    00 80488/174504898
## 5      China      19    99 212258/1272915272
## 6      China      20    00 213766/1280428583

```

5.2 Pacote `tidyverse`

Apesar das diversas possibilidades de situações que necessitem de limpeza de dados, a conjugação de 3 pacotes consegue resolver a grande maioria dos casos: `dplyr`, `tidyverse`, `stringr`.

O pacote `tidyverse` é mais um dos pacotes criados pelo Hadley. Esse fato por si só já traz algumas vantagens: ele se integra perfeitamente com o `dplyr` usando o conector `%>%` e tem a sintaxe de suas funções bastante intuitiva.

```

install.packages("tidyverse")
library(tidyverse)
?tidyverse

```

O `tidyverse` também tem suas funções organizadas em pequenos verbetes, onde cada um representa uma tarefa para organizar os dados. Os verbetes básicos que abordaremos serão os seguintes:

- `gather()`
- `separate()`
- `spread()`
- `unite()`

Vale lembrar que tudo que for feito usando o `tidyverse` é possível de ser feito também usando o R base, mas é uma forma um pouco menos intuitiva. Caso queira entender como usar o R base pra isso, procure mais sobre as funções `melt()` e `cast()`.

5.2.1 Gather

A função `gather()` serve para agrupar duas ou mais colunas e seus respectivos valores (conteúdos) em pares de valores. Assim, o resultado após o agrupamento são sempre duas colunas. A primeira delas possui observações cujos valores chave eram as colunas antigas e a segunda, possui os valores respectivos relacionados com as colunas antigas. Na prática, a função `gather` diminui o número de colunas e aumenta o número de linhas de nossa base de dados.

Usaremos dados disponíveis no R base para exemplificar:

```

data("USArrests")
str(USArrests)

```

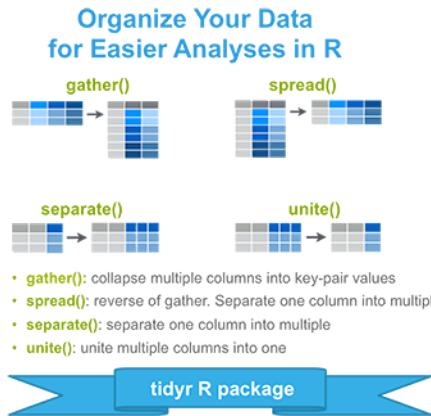


Figura 5.3: Tabela long

```
## 'data.frame':   50 obs. of  4 variables:
## $ Murder : num  13.2 10 8.1 8.8 9 7.9 3.3 5.9 15.4 17.4 ...
## $ Assault : int  236 263 294 190 276 204 110 238 335 211 ...
## $ UrbanPop: int  58 48 80 50 91 78 77 72 80 60 ...
## $ Rape    : num  21.2 44.5 31 19.5 40.6 38.7 11.1 15.8 31.9 25.8 ...
head(USArrests)
```

```
##           Murder Assault UrbanPop Rape
## Alabama     13.2     236      58 21.2
## Alaska      10.0     263      48 44.5
## Arizona      8.1     294      80 31.0
## Arkansas     8.8     190      50 19.5
## California    9.0     276      91 40.6
## Colorado      7.9     204      78 38.7
```

```
# Transformando o nome das linhas em colunas
USArrests$State <- rownames(USArrests)
head(USArrests)
```

```
##           Murder Assault UrbanPop Rape      State
## Alabama     13.2     236      58 21.2    Alabama
## Alaska      10.0     263      48 44.5    Alaska
## Arizona      8.1     294      80 31.0   Arizona
## Arkansas     8.8     190      50 19.5  Arkansas
## California    9.0     276      91 40.6 California
## Colorado      7.9     204      78 38.7 Colorado
```

```
usa.long <- USArrests %>%
  gather(key = "tipo_crime", value = "valor", -State)

head(usa.long)
```

```
##       State tipo_crime valor
## 1    Alabama    Murder  13.2
## 2    Alaska    Murder  10.0
## 3  Arizona    Murder   8.1
## 4  Arkansas    Murder   8.8
## 5 California    Murder   9.0
## 6 Colorado    Murder   7.9
```

```
tail(usa.long)

##           State tipo_crime valor
## 195      Vermont      Rape  11.2
## 196  Virginia      Rape  20.7
## 197 Washington      Rape  26.2
## 198 West Virginia      Rape   9.3
## 199 Wisconsin      Rape 10.8
## 200    Wyoming      Rape 15.6
```

No primeiro parâmetro do `gather()` nós informamos a “chave”, ou seja, a coluna que guardará o que antes era coluna. No segundo parâmetro informamos o “value”, ou seja, a coluna que guardar os valores para cada uma das antigas colunas. Reparem que agora você pode afirmar com certeza que cada linha é uma observação e que cada coluna é uma variável.

5.2.2 Spread

É a operação antagônica do `gather()`. Ela espalha os valores de duas colunas em diversos campos para cada registro: os valores de uma coluna viram o nome das novas colunas, e os valores de outra virão valores de cada registro nas novas colunas. Usaremos a `table2` para exemplificar:

```
head(table2)

## # A tibble: 6 x 4
##       country   year     type   count
##       <chr>     <int>   <chr>   <int>
## 1 Afghanistan 1999   cases     745
## 2 Afghanistan 1999 population 19987071
## 3 Afghanistan 2000   cases     2666
## 4 Afghanistan 2000 population 20595360
## 5      Brazil 1999   cases     37737
## 6      Brazil 1999 population 172006362

table2.wide <- table2 %>%
  spread(key = type, value = count)

head(table2.wide)

## # A tibble: 6 x 4
##       country   year   cases population
##       <chr>     <int>   <int>      <int>
## 1 Afghanistan 1999     745  19987071
## 2 Afghanistan 2000    2666  20595360
## 3      Brazil 1999   37737  172006362
## 4      Brazil 2000   80488  174504898
## 5      China 1999  212258 1272915272
## 6      China 2000  213766 1280428583
```

5.2.3 Separate

O `separate()` é usado para separar duas variáveis que estão em uma mesma coluna. Lembr-se: “cada coluna deve ser apenas uma única variável”. É muito normal virem variáveis juntas em uma única coluna, mas nem sempre isso é prejudicial, cabe avaliar quando vale a pena separar.

Usaremos o exemplo da `table3` para investigar:

```
table3.wide <- table3 %>%
  separate(rate, into = c("cases", "population"), sep='/')

head(table3.wide)

## # A tibble: 6 x 4
##       country   year cases population
##       <chr>     <int>  <chr>      <chr>
## 1 Afghanistan 1999    745 19987071
## 2 Afghanistan 2000   2666 20595360
## 3      Brazil 1999 37737 172006362
## 4      Brazil 2000 80488 174504898
## 5      China 1999 212258 1272915272
## 6      China 2000 213766 1280428583
```

5.2.4 Unite

A operação `unite()` é o oposto da `separate()`, ela pega duas colunas (variáveis) e transforma em uma só. Muito utilizada para montar relatórios finais ou tabelas para análise visual. Vamos aproveitar o exemplo em `table2` e montar uma tabela final comparando a case e population de cada país em cada ano.

```
table2.relatorio <- table2 %>%
  unite(type_year, type, year) %>%
  spread(key = type_year, value = count, sep = '_')

table2.relatorio

## # A tibble: 3 x 5
##       country type_year_cases_1999 type_year_cases_2000
##       <chr>          <int>           <int>
## 1 Afghanistan            745            2666
## 2      Brazil           37737           80488
## 3      China           212258          213766
## # ... with 2 more variables: type_year_population_1999 <int>,
## #   type_year_population_2000 <int>
```

O primeiro parâmetro é a coluna que desejamos criar, os próximos são as colunas que desejamos unir e por fim temos o `sep`, que é algum símbolo opcional para ficar entre os dois valores na nova coluna.

5.3 Manipulação de texto

Manipulação de texto também é algo importante em ciência de dados, pois nem tudo são números, existem variáveis categóricas que são baseadas em texto. Mais uma vez, esse tipo de manipulação depende do tipo de arquivo que você receber.

```
a <- 'texto 1'
b <- 'texto 2'
c <- 'texto 3'
paste(a, b, c)

## [1] "texto 1 texto 2 texto 3"
```

O `paste()` é a função mais básica para manipulação de textos usando o R base. Ela simplesmente concatena todas as variáveis textuais que você informar. Existe um parâmetro, extra (`sep`) cujo valor padrão é espaço

```
```
paste(a, b, c, sep = '-')
[1] "texto 1-texto 2-texto 3"
paste(a, b, c, sep = ';')
[1] "texto 1;texto 2;texto 3"
paste(a, b, c, sep = '---%---')
[1] "texto 1---%---texto 2---%---texto 3"
```

### 5.3.1 Pacote stringr

Texto no R é sempre do tipo `character`. No universo da computação, também se referem a texto como `string`. E é daí que vem o nome desse pacote, também criado por Hadley Wickham. Por um acaso esse pacote não está incluído no pacote `tidyverse`.

```
install.packages('stringr')
library(stringr)
?stringr
```

Começaremos pela função `str_sub()`, que extrai apenas parte de um texto.

```
cnae.texto <- c('10 Fabricação de produtos alimentícios', '11 Fabricação de bebidas',
 '12 Fabricação de produtos do fumo', '13 Fabricação de produtos têxteis',
 '14 Confecção de artigos do vestuário e acessórios',
 '15 Preparação de couros e fabricação de artefatos de couro, artigos para viagem e calçados',
 '16 Fabricação de produtos de madeira',
 '17 Fabricação de celulose, papel e produtos de papel')
cnae <- str_sub(cnae.texto, 0, 2)
texto <- str_sub(cnae.texto, 4)

cnae

[1] "10" "11" "12" "13" "14" "15" "16" "17"

texto

[1] "Fabricação de produtos alimentícios"
[2] "Fabricação de bebidas"
[3] "Fabricação de produtos do fumo"
[4] "Fabricação de produtos têxteis"
[5] "Confecção de artigos do vestuário e acessórios"
[6] "Preparação de couros e fabricação de artefatos de couro, artigos para viagem e calçados"
[7] "Fabricação de produtos de madeira"
[8] "Fabricação de celulose, papel e produtos de papel"
```

Temos também a função `str_replace()` e `str_replace_all()`, que substitui determinados caracteres por outros. Tal como no exemplo a seguir:

```
telefones <- c('9931-9512', '8591-5892', '8562-1923')
str_replace(telefones, '-', '')

[1] "99319512" "85915892" "85621923"

cnpj <- c('19.702.231/9999-98', '19.498.482/9999-05', '19.499.583/9999-50', '19.500.999/9999-46', '19.500.999/9999-46')
str_replace_all(cnpj, '\\.|/-', '')
```

```
[1] "19702231999998" "19498482999905" "19499583999950" "19500999999946"
[5] "19501139999990"
```

O que são esses símbolos no segundo exemplo? São símbolos especiais utilizados em funções textuais para reconhecimento de padrão. Esses símbolos são conhecidos como **Expressões Regulares** ou o famoso **Regex**.

### 5.3.2 Regex

Trata-se de um assunto bastante complexo e avançado. Não é fácil dominar regex e provavelmente você vai precisar sempre consultar e experimentar a montagem dos padrões de regex. Infelizmente não é possível aprender regex rápido e de um jeito fácil, só existe o jeito difícil: errando muito, e com muita prática e experiências reais.

A seguir, uma lista dos principais mecanismos de regex:

| regex                      | correspondência                   |
|----------------------------|-----------------------------------|
| ^                          | começa do string (ou uma negação) |
| .                          | qualquer caracter                 |
| \$                         | fim da linha                      |
| [maça]                     | procura as caracteres m, a, ç     |
| maça                       | maça                              |
| [0-9]                      | números                           |
| [A-Z]                      | qualquer letra maiúscula          |
| \w                         | uma palavra                       |
| \W                         | não é palavra                     |
| (é pontuação, espaço etc.) |                                   |
| \s                         | um espaço (tab, newline, space)   |

A seguir, alguns bons sites para aprender mais sobre regex. É um assunto interessante e bastante utilizado para tratamento textual.

<http://turing.com.br/material/regex/introducao.html>

<https://regextester.com/>

## 5.4 Exercícios

- Utilizando senado.csv monte uma tabela mostrando a quantidade de votos sim e não por coalisão, no formato wide (sim e não são linhas e coalisão ou não coalisão são colunas) Dica: `mutate(tipo_coalisao = ifelse(GovCoalition, 'Coalisão', 'Não Coalisão'))`

```
A tibble: 2 x 3
Tipo_voto Coalisão `Não Coalisão`
<chr> <int> <int>
1 Votos Não 702 657
2 Votos Sim 5002 2739
```

- Utilizando o dataframe abaixo, obtenha o resultado a seguinte. Dica: `separate()`, `str_replace_all()`, `str_trim()`, `str_sub()`

```
cadastrados <- data.frame(
 email = c('joaodasilva@gmail.com', 'rafael@hotmail.com', 'maria@uol.com.br', 'juliana.moraes@outlook.com.br')
```

```
 telefone = c('(61)99831-9482', '32 8976 2913', '62-9661-1234', '15-40192.5812')
)

cadastros

email telefone
1 joaodasilva@gmail.com (61)99831-9482
2 rafael@hotmail.com 32 8976 2913
3 maria@uol.com.br 62-9661-1234
4 juliana.morais@outlook.com 15-40192.5812

login dominio telefone dd
1 joaodasilva gmail 99831-9482 61
2 rafael hotmail 8976-2913 32
3 maria uol 9661-1234 62
4 juliana.morais outlook 40192-5812 15
```

# Capítulo 6

## Juntando dados

Existem duas grandes formas de junção de dados: **UNIÃO** e **CRUZAMENTO**.

Para que uma união seja possível, os dois conjuntos de dados precisam ter os mesmos campos. Para que um cruzamento seja possível, os dois conjuntos precisam ter pelo menos um campo em comum.

### 6.1 União de dados (Union)

A união de dados é mais intuitiva. Basta ter a mesma quantidade de campos e que os campos estejam “alinhados”. A função mais usada para isso é o famoso `rbind()` (Row Bind). Caso os campos tenham exatamente os mesmos nomes e tipo, o `rbind()` consegue fazer a união perfeitamente.

```
dados2016 <- data.frame(ano = c(2016, 2016, 2016),
 valor = c(938, 113, 1748),
 produto = c('A', 'B', 'C'))

dados2017 <- data.frame(valor = c(8400, 837, 10983),
 produto = c('H', 'Z', 'X'),
 ano = c(2017, 2017, 2017))

dados.finais <- rbind(dados2016, dados2017)

dados.finais

ano valor produto
1 2016 938 A
2 2016 113 B
3 2016 1748 C
4 2017 8400 H
5 2017 837 Z
6 2017 10983 X
```

União de dados é a forma mais simples de juntar dados.

### 6.2 Cruzamento de Dados (Join)

O cruzamento de dados é um pouco mais complexo, mas nem por isso chega a ser algo difícil.

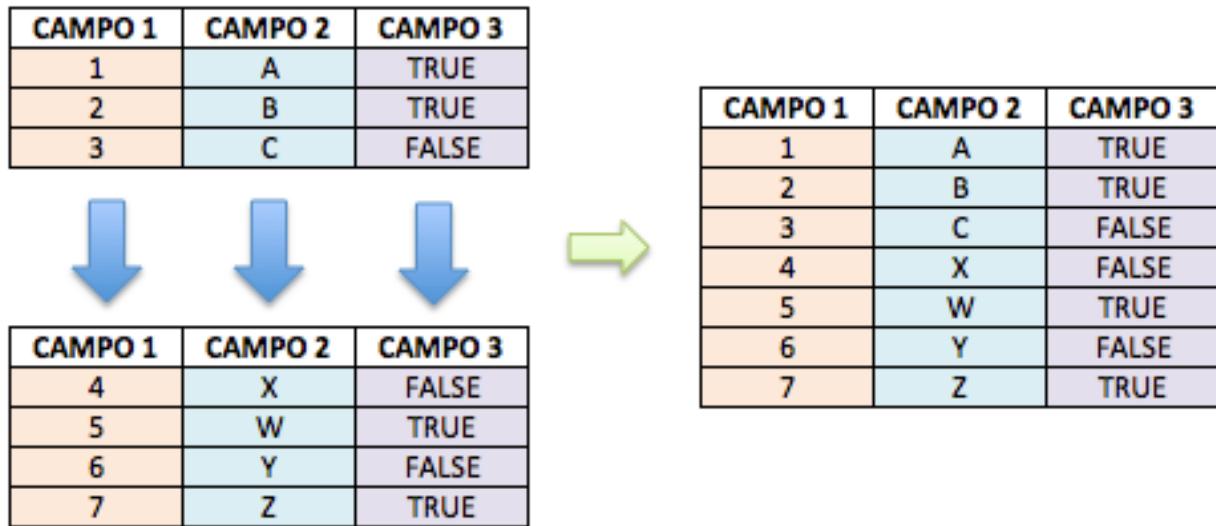


Figura 6.1: União de tabelas

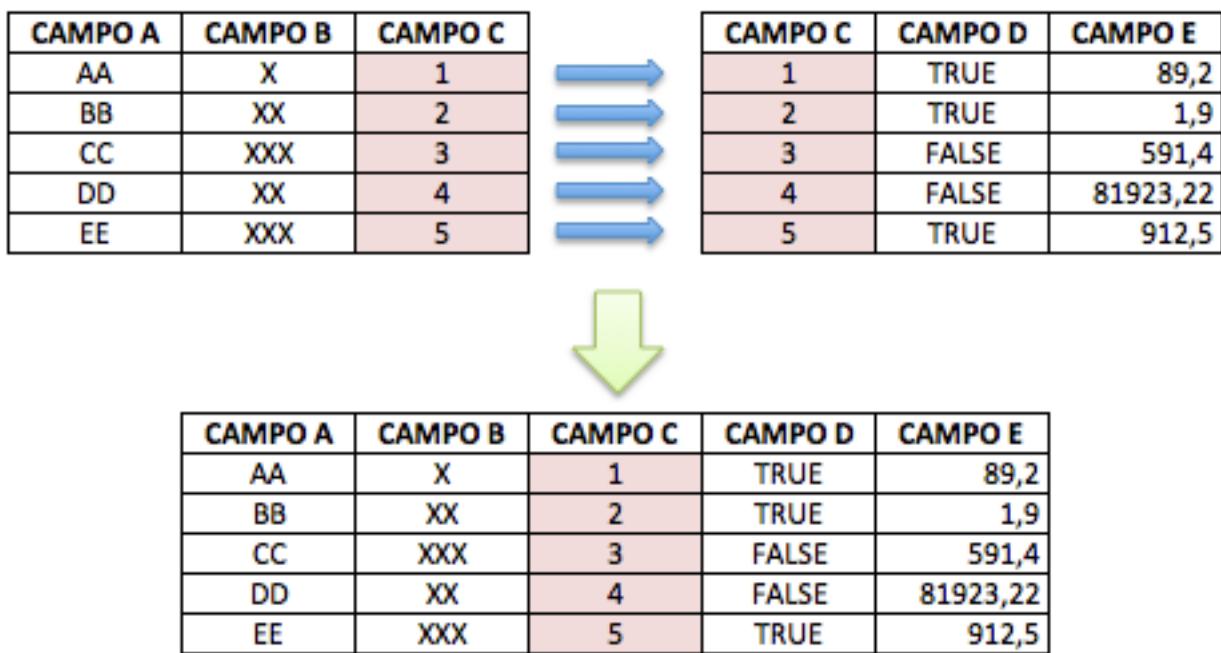


Figura 6.2: Cruzamento de tabelas

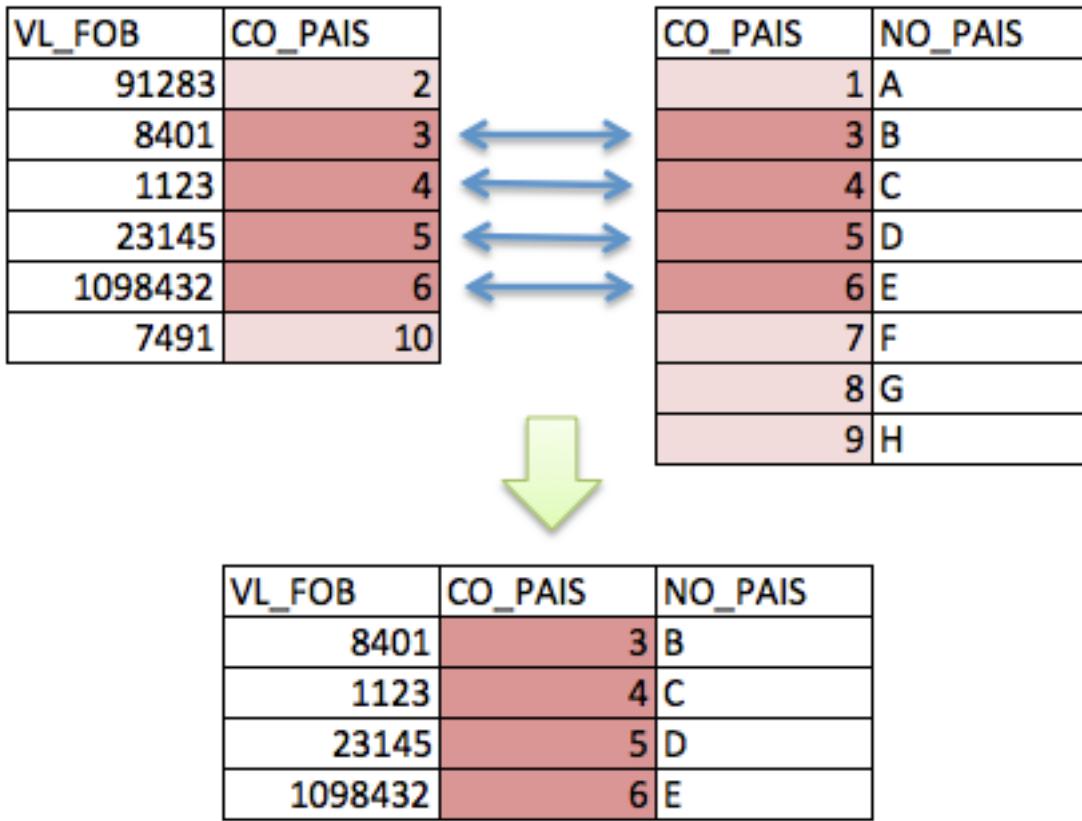


Figura 6.3: Cruzamento de tabelas

Para entender como fazer joins (cruzamentos) é preciso entender o conceito de **chave**. Entenda chave como uma coluna que está presente da mesma forma em dois conjuntos de dados distintos. O conceito completo de chave é bem mais complexo que isso, mas para começar a entender e usar os joins, basta usar essa intuição.

Tendo o conceito de chave em mente, a primeira coisa que deve fazer quando precisar cruzar dois conjuntos de dados é tentar identificar quais os campos chaves, ou seja, quais os campos estão presentes nos dois grupos.

O que acontece quando nem todos os códigos de um grupo estão no outro? E quando um grupo tem códigos repetidos em várias linhas? Para responder essas e outras perguntas precisamos conhecer os diferentes tipos de joins. Existe pelo menos uma dezena de tipos de joins, mas 90% das vezes você precisará apenas dos tipos básicos que explicaremos a seguir. Usaremos o pacote `dplyr` para aplicar os joins. O R base possui a função `merge()` para joins, se tiver curiosidade procure mais sobre ela depois.

### 6.2.1 Inner Join (ou apenas Join)

Trata-se do join mais simples, mais básico e mais usado dentre todos os outros tipos. O seu comportamento mantém no resultado apenas as linhas que estão presentes nos dois conjuntos de dados que estão sendo cruzados. O inner join funciona da seguinte forma:

A tabela final após o cruzamento conterá as linhas com as chaves que estiverem em AMBOS os conjuntos de dados. As linhas com chaves que não estão em ambos serão descartadas. Essa característica torna o inner join muito útil para fazer filtros.

Vamos utilizar dados já disponíveis no `dplyr` para testar os joins:

```

band_members

A tibble: 3 x 2
name band
<chr> <chr>
1 Mick Stones
2 John Beatles
3 Paul Beatles

band_instruments

A tibble: 3 x 2
name plays
<chr> <chr>
1 John guitar
2 Paul bass
3 Keith guitar

str(band_members)

Classes 'tbl_df', 'tbl' and 'data.frame': 3 obs. of 2 variables:
$ name: chr "Mick" "John" "Paul"
$ band: chr "Stones" "Beatles" "Beatles"

str(band_instruments)

Classes 'tbl_df', 'tbl' and 'data.frame': 3 obs. of 2 variables:
$ name : chr "John" "Paul" "Keith"
$ plays: chr "guitar" "bass" "guitar"

#vamos juntar os dois conjuntos com um join

band_members %>% inner_join(band_instruments)

A tibble: 2 x 3
name band plays
<chr> <chr> <chr>
1 John Beatles guitar
2 Paul Beatles bass

#o dplyr "adivinhou" a coluna chave pelo nome

```

Repare que nesse caso a chave é a coluna `name`. Repare também que os dois conjuntos tem 3 registros. Então por que o resultado final só tem 2 registros? Pois o comportamento do join é justamente retornar apenas as linhas em que as chaves coincidiram (efeito de filtro).

Vamos fazer o mesmo experimento com `band_instruments2`

```

band_instruments2

A tibble: 3 x 2
artist plays
<chr> <chr>
1 John guitar
2 Paul bass
3 Keith guitar

str(band_instruments2) #o nome da coluna é diferente

Classes 'tbl_df', 'tbl' and 'data.frame': 3 obs. of 2 variables:

```

```
$ artist: chr "John" "Paul" "Keith"
$ plays : chr "guitar" "bass" "guitar"
band_members %>% inner_join(band_instruments2, by = c('name' = 'artist'))
```

```
A tibble: 2 x 3
name band plays
<chr> <chr> <chr>
1 John Beatles guitar
2 Paul Beatles bass
```

Repare que dessa vez tivemos que especificar qual a coluna chave para que o join aconteça.

Mais um exemplo:

```
setwd('dados')

empregados <- read_csv('dados/Employees.csv')
departamentos <- read_csv('dados/Departments.csv')

str(empregados)

Classes 'tbl_df', 'tbl' and 'data.frame': 6 obs. of 4 variables:
$ Employee : int 1 2 3 4 5 6
$ EmployeeName: chr "Alice" "Bob" "Carla" "Daniel" ...
$ Department : int 11 11 12 12 13 21
$ Salary : int 800 600 900 1000 800 700
- attr(*, "spec")=List of 2
..$ cols :List of 4
...$ Employee : list()
... ..- attr(*, "class")= chr "collector_integer" "collector"
...$ EmployeeName: list()
... ..- attr(*, "class")= chr "collector_character" "collector"
...$ Department : list()
... ..- attr(*, "class")= chr "collector_integer" "collector"
...$ Salary : list()
... ..- attr(*, "class")= chr "collector_integer" "collector"
..$ default: list()
...- attr(*, "class")= chr "collector_guess" "collector"
..- attr(*, "class")= chr "col_spec"

str(departamentos)

Classes 'tbl_df', 'tbl' and 'data.frame': 4 obs. of 3 variables:
$ Department : int 11 12 13 14
$ DepartmentName: chr "Production" "Sales" "Marketing" "Research"
$ Manager : int 1 4 5 NA
- attr(*, "spec")=List of 2
..$ cols :List of 3
...$ Department : list()
... ..- attr(*, "class")= chr "collector_integer" "collector"
...$ DepartmentName: list()
... ..- attr(*, "class")= chr "collector_character" "collector"
...$ Manager : list()
... ..- attr(*, "class")= chr "collector_integer" "collector"
..$ default: list()
...- attr(*, "class")= chr "collector_guess" "collector"
```

```
..- attr(*, "class")= chr "col_spec"
empregados

A tibble: 6 x 4
Employee EmployeeName Department Salary
<int> <chr> <int> <int>
1 1 Alice 11 800
2 2 Bob 11 600
3 3 Carla 12 900
4 4 Daniel 12 1000
5 5 Evelyn 13 800
6 6 Ferdinand 21 700

departamentos

A tibble: 4 x 3
Department DepartmentName Manager
<int> <chr> <int>
1 11 Production 1
2 12 Sales 4
3 13 Marketing 5
4 14 Research NA

final <- empregados %>%
 inner_join(departamentos, by = c('Employee' = 'Manager'))

final

A tibble: 3 x 6
Employee EmployeeName Department.x Salary Department.y DepartmentName
<int> <chr> <int> <int> <int> <chr>
1 1 Alice 11 800 11 Production
2 4 Daniel 12 1000 12 Sales
3 5 Evelyn 13 800 13 Marketing
```

Novamente tivemos o mesmo efeito, listamos apenas os empregados que são Gerentes de departamento.

Acontece que existem situações em que esse descarte de registro do inner join não é interessante. Nesses casos usamos outros tipos de join: os Outer Joins. Existem 3 tipos básicos de outer join: left outer join (ou só left join), o right outer join (ou só right join) e full outer join (ou apenas full join)

## 6.2.2 Left Outer Join

Chama-se **LEFT** outer join pois todos os registros do “conjunto da esquerda” estarão presentes no resultado final, além dos registros da direita que coincidirem na chave. Podemos usar no caso a seguir.

```
band_members %>% left_join(band_instruments2, by = c('name' = 'artist'))
```

```
A tibble: 3 x 3
name band plays
<chr> <chr> <chr>
1 Mick Stones <NA>
2 John Beatles guitar
3 Paul Beatles bass

band_instruments2
```

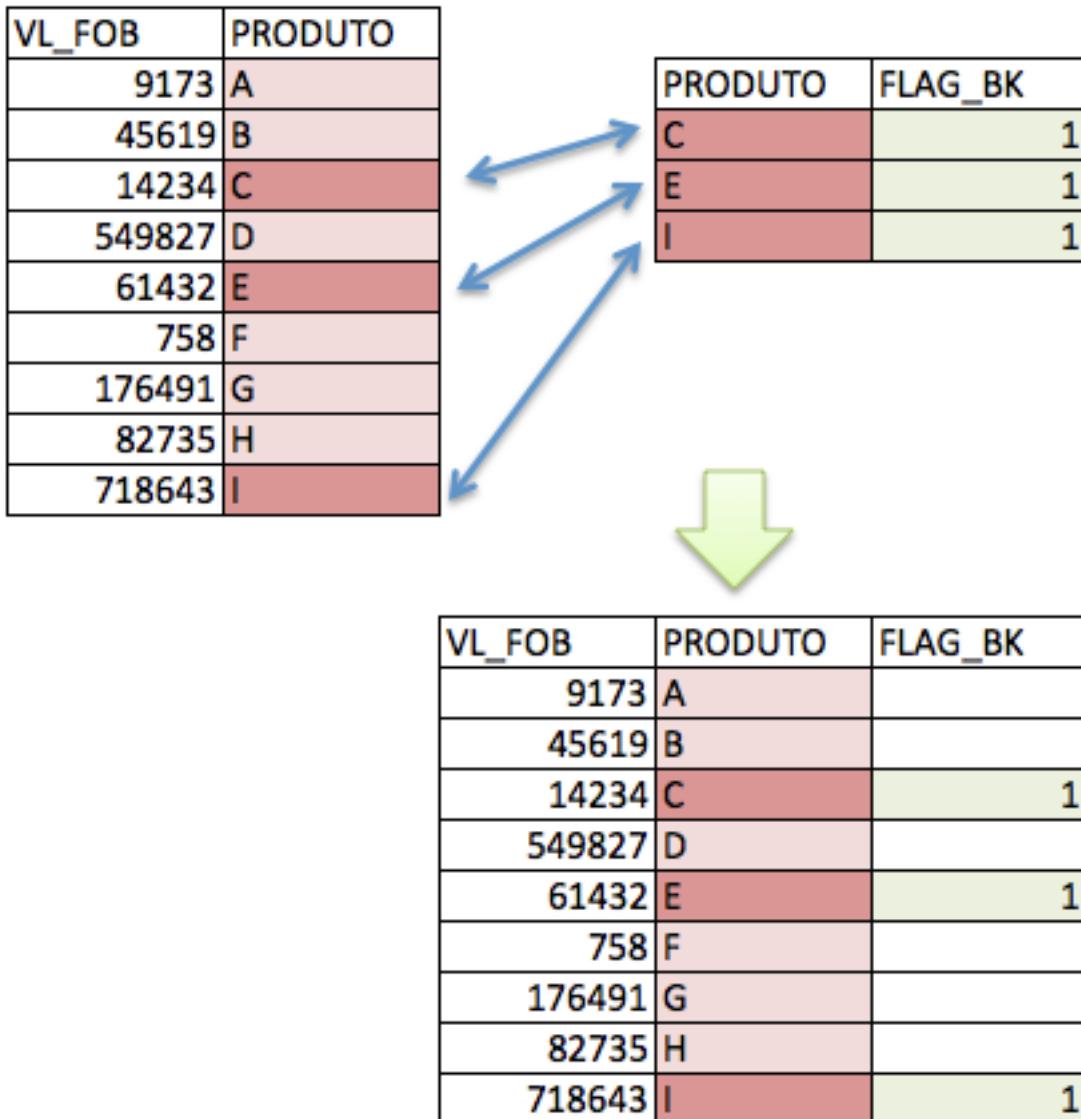


Figura 6.4: Cruzamento de tabelas

```
A tibble: 3 x 2
artist plays
<chr> <chr>
1 John guitar
2 Paul bass
3 Keith guitar
```

Reparem no efeito: mesmo Mick não tendo referência no conjunto de dados “da direita” (band\_instruments2) ele apareceu no registro final com NA no campo que diz respeito ao conjunto da direita. Da mesma forma, Keith não está presente no conjunto final pois não tem referência no conjunto da esquerda.

Repare que a “posição” das tabelas faz diferença. No caso da nossa manipulação de exemplo, aplicamos o left join pois a tabela que queríamos preservar estava “à esquerda” na manipulação.

```
final2 <- empregados %>%
 left_join(departamentos, by = c('Employee' = 'Manager'))
```

```
final2
```

```
A tibble: 6 x 6
Employee EmployeeName Department.x Salary Department.y DepartmentName
<int> <chr> <int> <int> <int> <chr>
1 1 Alice 11 800 11 Production
2 2 Bob 11 600 NA <NA>
3 3 Carla 12 900 NA <NA>
4 4 Daniel 12 1000 12 Sales
5 5 Evelyn 13 800 13 Marketing
6 6 Ferdinand 21 700 NA <NA>
```

### 6.2.3 Right Outer Join

O princípio é EXATAMENTE o mesmo do left join. A única diferença é a permanência dos registros do conjunto da direita. Podemos chegar ao mesmo resultado anterior apenas mudando os data frames de posição na manipulação.

```
final3 <- departamentos %>%
 right_join(empregados, by = c('Manager'='Employee'))
```

```
final3
```

```
A tibble: 6 x 6
Department.x DepartmentName Manager EmployeeName Department.y Salary
<int> <chr> <int> <chr> <int> <int>
1 11 Production 1 Alice 11 800
2 NA <NA> 2 Bob 11 600
3 NA <NA> 3 Carla 12 900
4 12 Sales 4 Daniel 12 1000
5 13 Marketing 5 Evelyn 13 800
6 NA <NA> 6 Ferdinand 21 700
```

```
final2
```

```
A tibble: 6 x 6
Employee EmployeeName Department.x Salary Department.y DepartmentName
<int> <chr> <int> <int> <int> <chr>
1 1 Alice 11 800 11 Production
2 2 Bob 11 600 NA <NA>
3 3 Carla 12 900 NA <NA>
4 4 Daniel 12 1000 12 Sales
5 5 Evelyn 13 800 13 Marketing
6 6 Ferdinand 21 700 NA <NA>
```

A escolha entre right join e left join vai depender completamente da ordem que você escolher realizar as operações. Via de regra, um pode ser substituído pelo outro desde que a posição dos data frames se ajuste na sequência das manipulações

### 6.2.4 Full Outer Join

Existem ainda as situações em que é necessário preservar todos os registros de ambos os conjuntos de dados. O full join tem essa característica. Nenhum dos conjuntos de dados perderá registros no resultado final, isto

é, quando as chaves forem iguais, todos os campos estarão preenchidos. Quando não houver ocorrência das chaves em ambos os lados, será informado NA em qualquer um dos “lados”.

```
band_members %>% full_join(band_instruments2, by = c('name' = 'artist'))
```

```
A tibble: 4 x 3
name band plays
<chr> <chr> <chr>
1 Mick Stones <NA>
2 John Beatles guitar
3 Paul Beatles bass
4 Keith <NA> guitar
```

Reparam que dessa vez não perdemos nenhum registro de nenhum dos conjuntos de dados, apenas teremos NA quando a ocorrência da chave não acontecer em alguns dos conjuntos.

O full join funciona da seguinte forma:

```
final4 <- departamentos %>%
 full_join(empregados, by = c('Manager'='Employee'))
```

```
final4
```

```
A tibble: 7 x 6
Department.x DepartmentName Manager EmployeeName Department.y Salary
<int> <chr> <int> <chr> <int> <int>
1 11 Production 1 Alice 11 800
2 12 Sales 4 Daniel 12 1000
3 13 Marketing 5 Evelyn 13 800
4 14 Research NA <NA> NA NA
5 NA <NA> 2 Bob 11 600
6 NA <NA> 3 Carla 12 900
7 NA <NA> 6 Ferdinand 21 700
```

Do resultado desse full join, por exemplo, podemos concluir que não tem nenhum *Manager* no departamento *Resarch*, da mesma forma os empregados Bob, Carla e Ferdinand não são *managers* de departamento nenhum.

## 6.3 Exercícios

- Utilizando as bases de dados do pacote `nycflights13`, encontre a tabela abaixo que mostra quais aeroportos (Origem e Destino) tiveram mais voos. Será necessário utilizar o dataframe `flights` e `airports`. Dica: primeiro descubra as chaves.

```
A tibble: 217 x 3
Groups: Origem [3]
Origem Destino qtd
<chr> <chr> <int>
1 John F Kennedy Intl Los Angeles Intl 11262
2 La Guardia Hartsfield Jackson Atlanta Intl 10263
3 La Guardia Chicago Ohare Intl 8857
4 John F Kennedy Intl San Francisco Intl 8204
5 La Guardia Charlotte Douglas Intl 6168
6 Newark Liberty Intl Chicago Ohare Intl 6100
7 John F Kennedy Intl General Edward Lawrence Logan Intl 5898
8 La Guardia Miami Intl 5781
9 John F Kennedy Intl Orlando Intl 5464
```

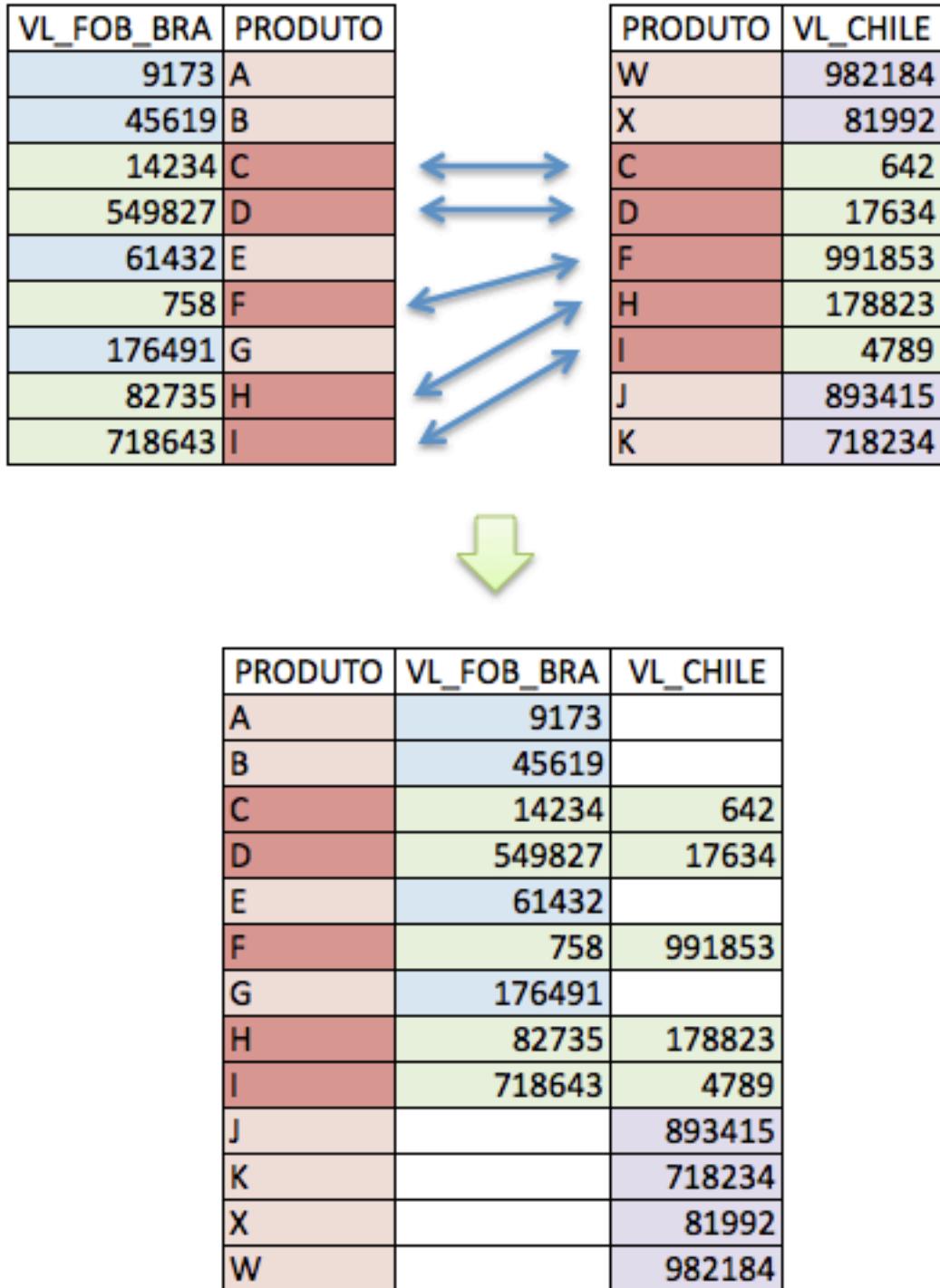


Figura 6.5: Cruzamento de tabelas

```
10 Newark Liberty Intl General Edward Lawrence Logan Intl 5327
... with 207 more rows
```

2. Utilizando os dataframes abaixo, chege no resultado a seguir.

```
participantes <- data.frame(
 Nome = c('Carlos', 'Maurício', 'Ana Maria', 'Rebeca', 'Patrícia'),
 Cidade = c('Brasília', 'Minas Gerais', 'Goiás', 'São Paulo', 'Ceará'),
 Idade = c(23, 24, 22, 29, 28)
)

aprovados <- data.frame(
 Nome = c('Carlos', 'Patrícia'),
 Pontuacao = c(61, 62)
)

eliminados <- data.frame(
 Nome = c('Maurício', 'Ana Maria', 'Rebeca'),
 Pontuacao = c(49, 48, 48)
)

participantes

Nome Cidade Idade
1 Carlos Brasília 23
2 Maurício Minas Gerais 24
3 Ana Maria Goiás 22
4 Rebeca São Paulo 29
5 Patrícia Ceará 28

aprovados

Nome Pontuacao
1 Carlos 61
2 Patrícia 62

eliminados

Nome Pontuacao
1 Maurício 49
2 Ana Maria 48
3 Rebeca 48

Warning: Column `Nome` joining factors with different levels, coercing to
character vector

Warning: Column `Nome` joining character vector and factor, coercing into
character vector

Nome Cidade Idade Pontuacao Resultado
1 Carlos Brasilia 23 61 Aprovado
2 Maurício Minas Gerais 24 49 Eliminado
3 Ana Maria Goiás 22 48 Eliminado
4 Rebeca São Paulo 29 48 Eliminado
5 Patrícia Ceará 28 62 Aprovado
```



# Capítulo 7

## Escrevendo dados

Já na fase final da sua análise, pode ser que apareça a necessidade de gerar arquivos: gráficos, relatórios, planilhas, pdf, arquivos de dados, etc.

Da mesma forma que você consome dados e relatórios, talvez você precise produzir e divulgar dados e relatórios para outras pessoas analisarem, ou para publicar.

### 7.1 Escrevendo csv

O formato mais básico e mais usado mundialmente para envio e recebimento de dados entre instituições é o `csv`. Para escrever um arquivo de dados em `csv` é muito simples. Utilizaremos uma função do R base para isso: `write.table()`

### 7.2 Rdata

Caso seja necessário salvar um ou vários objetos para passar para alguém ou até mesmo para continuar seu trabalho a partir de certo ponto, pode-se utilizar o formato de dados próprios do R: `Rdata`.

Veja o seguinte exemplo:

```
participantes <- data.frame(
 Nome = c('Carlos', 'Maurício', 'Ana Maria', 'Rebeca', 'Patrícia'),
 Cidade = c('Brasília', 'Minas Gerais', 'Goiás', 'São Paulo', 'Ceará'),
 Idade = c(23, 24, 22, 29, 28)
)

save(participantes, file = 'participantes.Rdata')

rm(participantes) # removendo o objeto
```

Pronto você salvou o objeto `participantes` no arquivo `participantes.Rdata`. Esse arquivo é específico para ser lido pelo R e interpretado como objeto. Como excluímos o arquivo, tente exibi-lo para ver o que acontece: erro. Agora vejamos como carregá-lo novamente no R utilizando o arquivo.

```
load('participantes.Rdata')

str(participantes)
```

```
'data.frame': 5 obs. of 3 variables:
$ Nome : Factor w/ 5 levels "Ana Maria","Carlos",...: 2 3 1 5 4
$ Cidade: Factor w/ 5 levels "Brasília","Ceará",...: 1 4 3 5 2
$ Idade : num 23 24 22 29 28
```

## 7.3 Escrevendo outros tipos de arquivos

Outra forma bastante importante de escrever dados é em planilhas: o famoso Excel. Recomendo conhecer o pacote `openxlsx`. É um pacote que lê e escreve excel sem nenhuma dependência de Java, que pode acabar dando muita dor de cabeça para manter e normalmente consome bastante memória. Para windows o `openxlsx` precisa do `Rtools`: <https://cran.r-project.org/bin/windows/Rtools/>. Recomendamos conhecer e experimentar esse pacote, com ele é possível criação de planilhas bem acabadas, com cores e formatações complexas.

Outra forma de escrita de dados é utilizando o RMarkdown, mas esse formato merece um capítulo específico para detalhar seu uso.

## 7.4 Exercícios

- .1 Escolha qualquer dataframe já trabalhado até agora e escreva-o em csv.
- .2 Experimente algo semelhante ao exemplo: escolha qualquer dataframe, save-o como Rdata, remova-o com o `rm()` em seguida carregue novamente com o `load()`

# Capítulo 8

## Obtendo dados

A base da ciência de dados é, obviamente, o DADO. Portanto, é fundamental sempre ter boas fontes de dados. Se você der sorte, conseguirá dados estruturados para iniciar sua análise. Porém, eventualmente precisará recorrer a fontes de dados não estruturados ou semi-estruturados.

Muito provavelmente você algum dia precisará recorrer a uma API de dados, ou até mesmo utilizar técnicas de Web Scrapping para obter dados diretamente em um próprio site.

### 8.1 API

API (*Application Programming Interface*), é uma forma de comunicação de dados mais apropriada para trocas de informações entre softwares. Normalmente APIs trocam dados em formato hierárquico. Os dois formatos hierárquicos mais comuns são JSON (Javascript Object Notation) e XML (eXtensible Markup Language).

Para obter e utilizar dados de API em R recomendamos a utilização do pacote `jsonlite`.

```
library(jsonlite)
```

A seguir apresentaremos alguns exemplos de APIs e seu uso. Existem diversas APIs e formas de consumí-las, portanto não iremos exaurir nesse texto todas as possibilidades de uso de APIs. O principal é entender APIs como uma fonte rica de dados que pode ser explorada em suas análises.

No exemplo a seguir utilizamos a API do github (portal para repositórios) e veremos quais os repositórios do Hadley Wickham

```
hadley.rep <- jsonlite::fromJSON("https://api.github.com/users/hadley/repos")
```

```
dim(hadley.rep)
```

```
[1] 30 71
```

```
head(hadley.rep[,c('name', 'description')], 15)
```

```
name
1 15-state-of-the-union
2 15-student-papers
3 500lines
4 adv-r
5 appdirs
6 assertthat
```

```

7 babynames
8 beautiful-data
9 bench
10 bigvis
11 bigvis-infovis
12 boxplots-paper
13 broom
14 builder
15 building-permits
description
1 <NA>
2 Graphics & computing student paper winners @ JSM 2015
3 500 Lines or Less
4 Advanced R programming: a book
5 A small Python module for determining appropriate platform-specific dirs, e.g. a "user data dir".
6 User friendly assertions for R
7 An R package contain all baby names data from the SSA
8 Book chapter for beautiful data
9 Benchmarking tools for R
10 Exploratory data analysis for large datasets (10-100 million observations)
11 Paper describing the bigvis package and framework submitted to Infovis 2013
12 <NA>
13 Convert statistical analysis objects from R into tidy format
14 Provide a simple way to create XML markup and data structures.
15 Code & data accompanying "whole-game" youtube video

```

Outra exemplo de API muito interessante é o portal de dados abertos da câmara dos deputados, eles possuem diversas APIs para consultar os dados do processo legislativo. Veja o exemplo a seguir que resgata as proposições utilizando API:

```

proposicoes <- jsonlite:::fromJSON("https://dadosabertos.camara.leg.br/api/v2/proposicoes")

head(proposicoes$dados %>% select(siglaTipo, numero, ano, ementa))

```

```

siglaTipo numero ano
1 PEC 454 1997
2 PL 6 1995
3 PL 125 1999
4 PL 220 1995
5 PL 227 1995
6 PL 246 1995
##
1 Altera a Lei nº 8.666, de 21 de junho de 1993
2 que "regulamenta o artigo 37, inciso I
3 da Constituição Federal"
4 Altera dispositivos da Lei nº 8.666, de 21 de junho de 1993, que "regulamenta o artigo 37, inciso I
5 Altera dispositivos da Lei nº 8.666, de 21 de junho de 1993, que "regulamenta o artigo 37, inciso I
6 Altera dispositivos da Lei nº 8.666, de 21 de junho de 1993, que "regulamenta o artigo 37, inciso I

```

Hoje em dia todas as redes sociais possuem APIs para consumir os dados dos usuários e postagens. Normalmente essas APIs pedem um cadastro anterior (apesar de gratuitas em sua maior parte). O R possui diversos pacotes para consumir APIs interessantes:

- Quandl: pacote que fornece diversos dados econômicos de diversos países
- Rfacebook: pacote que facilita o uso da API do facebook (requer cadastro prévio)
- twitterR: pacote que facilita o uso da API do twitter (requer cadastro prévio)

- ggmap: pacote que facilita o uso da API do google maps

Como dito, a lista não é exaustiva. Sempre procure por APIs para obter dados que possam enriquecer suas análises.

## 8.2 Web Scrapping

Eventualmente você não terá facilmente dados estruturados, nem terá uma API com os dados que procura. Nesses casos pode ser que um próprio site da internet seja sua fonte de dados. Para isso utiliza-se técnicas chamadas de Web Scrapping.

Sites da internet são construídos utilizando uma linguagem que é interpretada pelos browsers: HTML (*HyperText Markup Language*). É uma linguagem que trabalha com tags de forma hierárquica. Nesse site você pode aprender um pouco mais o que é HTML [http://www.w3schools.com/html/tryit.asp?filename=tryhtml\\_basic\\_document](http://www.w3schools.com/html/tryit.asp?filename=tryhtml_basic_document)

Existe um pacote em R que facilita muito o consumo de dados em HTML: `rvest`, criado também por Hadley Wickham. O `rvest` mapeia os elementos HTML (tags) de uma página web e facilita a “navegação” do R por esses nós da árvore do HTML. Veja o exemplo a seguir:

```
library(rvest)

html <- read_html("https://pt.wikipedia.org/wiki/Lista_de_redes_de_tv_no_Brasil")

html$table <- html %>% html_nodes("table")
dados <- html$table[[1]] %>% html_table()

dados <- dados %>%
 select(-`Lista de emissoras`)

head(dados)
```

Otivemos todo o HTML da página, mapeamos os nós de tabela (table) e pegamos seu conteúdo. A partir daí trata-se de um dataframe normal que pode ser manipulado com o `dplyr`.

## 8.3 Exercícios

1. Obtenha a tabela exibida em <http://globoesporte.globo.com/futebol/brasileirao-serie-a/> e chegue no seguinte resultado:
2. Escolha um site do seu interesse e faça um dataframe com uma parte do seu conteúdo (tabelas, listas, etc...)



## Capítulo 9

# Visualizações de dados (ggplot2)

O `ggplot2` é mais um pacote desenvolvido pelo Hadley Wickham, o criador, por exemplo, do `tidyverse` e do `dplyr`. A ideia do pacote, ainda que com algumas modificações, vem de uma obra chamada *The Grammar of Graphics*, que é uma maneira de descrever um gráfico a partir dos seus componentes. Dessa forma, teoricamente, ficaria mais fácil entender a construção de gráficos mais complexos.

O `ggplot2` é estruturado de forma que a “gramática” seja utilizada para um gráfico a partir de múltiplas camadas. As camadas serão formadas por dados, mapeamentos estéticos, transformações estatísticas dos dados, objetos geométricos (pontos, linhas, barras etc.) e ajuste de posicionamento. Além disso, existem outros componentes como os sistemas de coordenadas (cartesiano, polar, mapa etc.) e, se for o caso, divisões do gráfico em subplots (`facet`). Um simples exemplo de múltiplas camadas seria um gráfico de pontos adicionado de uma curva de ajustamento.

Uma forma geral (template) para entender a estrutura do `ggplot2`, segundo o próprio Hadley Wickham no livro *R for Data Science*, é a seguinte:

```
ggplot(data = <DATA>) +
 <GEOM_FUNCTION>(
 mapping = aes(<MAPPINGS>),
 stat = <STAT>,
 position = <POSITION>
) +
 <COORDINATE_FUNCTION> +
 <FACET_FUNCTION> # dividir o gráfico em subplots
```

A ideia é que todo gráfico pode ser representado por essa forma. No entanto, na criação de um gráfico, não é necessário especificar todas as partes acima. O `ggplot2` já oferece um padrão para o sistema de coordenadas, para o `stat` e `position`. O `facet` (subplot) só será utilizado quando necessário.

Além disso, existem as escalas que são utilizadas para controlar o mapeamento dos dados em relação aos atributos estéticos do gráfico. Por exemplo, suponha que no seu gráfico existe uma coluna que é uma variável categórica com 3 classes possíveis, e as cores do objeto geométrico estão associadas a essa variável. Automaticamente, o `ggplot2` vai definir uma cor pra cada classe. No entanto, você pode alterar a escala de cores para ter controle sobre elas. O mesmo vale para os valores apresentados nos eixos x e y.

Uma observação importante é que, apesar de os dados estarem na função `ggplot()` (<DATA>), eles também podem ser incluídos diretamente em cada objeto geométrico. Isso será útil quando for necessário criar uma nova camada a partir de dados diferentes daqueles que estão inicialmente nos gráficos.

Dessa forma, incorporando essas observações, um template estendido seria o abaixo:

```
ggplot(data = <DATA>) +
```

```
<GEOM_FUNCTION>(
 mapping = aes(<MAPPINGS>),
 stat = <STAT>,
 position = <POSITION>,
 data = <DATA> # pode receber os dados diretamente
) +
<SCALE_FUNCTION> + # uma para cada elemento estético
<COORDINATE_FUNCTION> +
<FACET_FUNCTION> # dividir o gráfico em subplots
```

Também é importante ressaltar que, como todo sistema de gráficos, é possível alterar todos os títulos e rótulos do gráfico, além de controle sobre características do tema do gráfico (cor do fundo, estilo da fonte, tamanho da fonte etc).

Para quebrar a barreira inicial, vamos criar um exemplo por partes:

```
library(ggplot2)
data("mtcars")

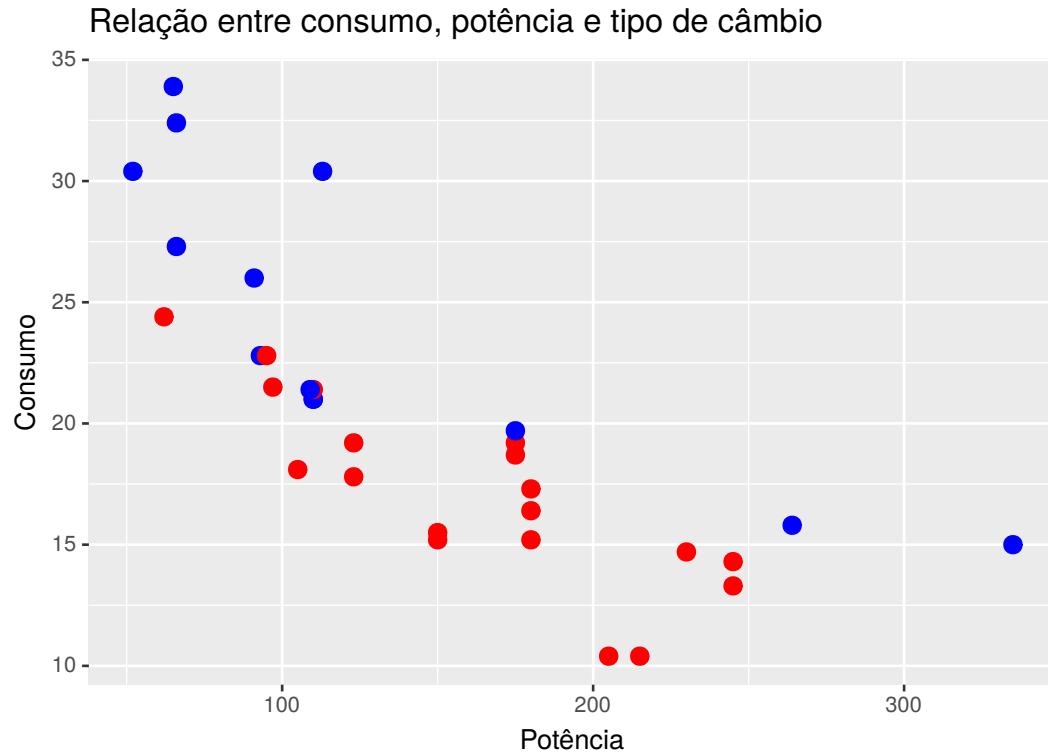
Inicia o plot
g <- ggplot(mtcars)

Adicionar pontos (geom_point) e
vamos mapear variáveis a elemertos estéticos dos pontos
Size = 3 define o tamanho de todos os pontos
g <- g +
 geom_point(aes(x = hp, y = mpg, color = factor(am)),
 size = 3)

Altera a escala de cores
g <- g +
 scale_color_manual("Automatic",
 values = c("red", "blue"),
 labels = c("No", "Yes"))

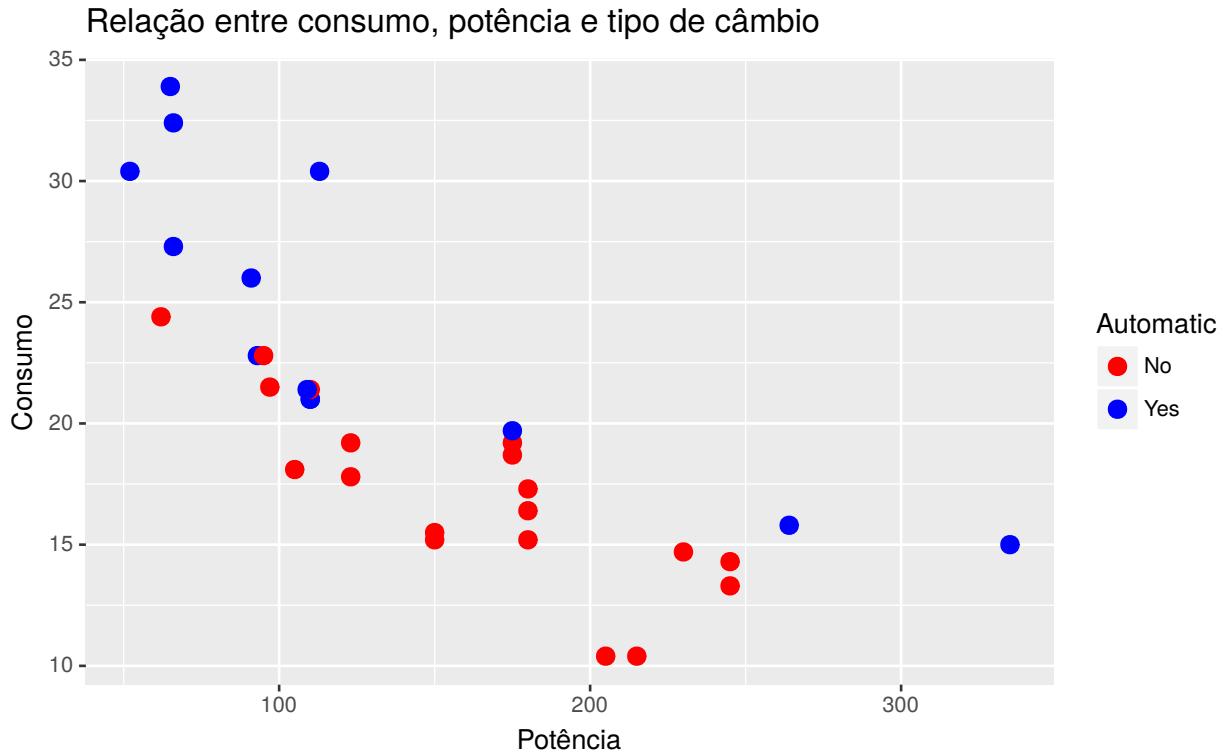
Rótulos (títulos)
g <- g +
 labs(title = 'Relação entre consumo, potência e tipo de câmbio',
 y = 'Consumo',
 x = 'Potência')

g
```



Note que o gráfico poderia ser criado com um bloco único de código:

```
ggplot(mtcars) +
 geom_point(aes(x = hp, y = mpg, color = factor(am)),
 size = 3) +
 scale_color_manual("Automatic",
 values = c("red", "blue"),
 labels = c("No", "Yes")) +
 labs(title = 'Relação entre consumo, potência e tipo de câmbio',
 y = 'Consumo',
 x = 'Potência')
```



Iremos detalhar cada parte do gráfico, mas vale falar rapidamente sobre o código acima. Primeiramente, passamos um conjunto de dados para o ggplot. Depois, adicionamos uma camada de pontos, mapeando as variáveis `hp` e `mpg` para as posições de cada ponto nos eixos `x` e `y`, respectivamente, e a variável `am` para a cor de cada ponto. Adicionalmente, alteramos a escala de cor, definindo seu título, os rótulos (`labels`) e os valores (`values`) para as cores. Por fim, definimos os títulos/rótulos do gráfico.

Nas próximas seções, iremos falar com mais detalhes de cada componente. Começando pelo mapeamento estético.

## 9.1 Mapeamento Estético

O mapeamento estético é o mapeamento de variáveis dos dados às características visuais dos objetos geométricos (pontos, barras, linhas etc.). Isso é feito a partir da função `aes()`. E quais são as características visuais de um objeto geométrico? Abaixo segue uma lista não exaustiva:

- Posição (`x` e `y`);
- Cor (`color`);
- Tamanho (`size`);
- Preenchimento (`fill`);
- Transparência (`alpha`);
- Texto (`label`);

Como vimos no exemplo acima, mapeamos três variáveis para três características visuais de cada ponto: posição `x`, posição `y` e cor. Nos próximos exemplos, outros elementos estéticos serão utilizados, conforme o objeto geométrico selecionado.

## 9.2 Objetos geométricos

Os objetos geométricos começam com a expressão `geom_` e são seguidos pelo tipo de objeto. Por exemplo, `geom_point()` para pontos e `geom_bar()` para barras. A tabela abaixo apresenta os tipos de objetos geométricos utilizados para criar alguns tipos de gráficos populares.

| Tipo                    | Objeto Geométrico                                 |
|-------------------------|---------------------------------------------------|
| Dispersão (scatterplot) | <code>geom_point()</code>                         |
| Gráfico de bolhas       | <code>geom_point()</code>                         |
| Gráfico de barras       | <code>geom_bar()</code> e <code>geom_col()</code> |
| Histograma              | <code>geom_histogram()</code>                     |
| Boxplot                 | <code>geom_boxplot()</code>                       |
| Densidade               | <code>geom_density()</code>                       |
| Gráfico de linhas       | <code>geom_line()</code>                          |

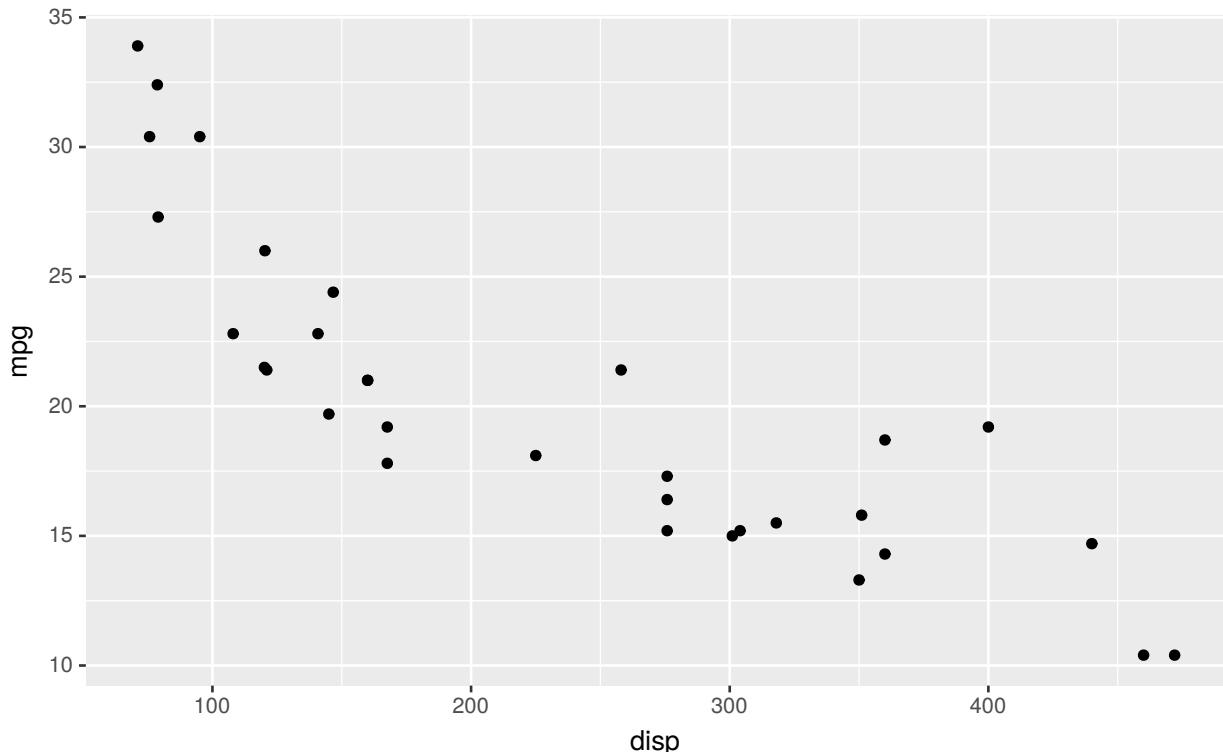
Nesse material, os principais tipos de objetos geométricos serão demonstrados a partir de exemplos. A lista completa de objetos geométricos e as descrições dos argumentos estão na documentação do `ggplot2`.

É importante saber que um gráfico do `ggplot2` pode ter mais de um objeto geométrico, cada um formando uma camada. Por exemplo, uma camada pontos e outra de linhas que conectam os pontos.

Vamos primeiramente criar uma gráfica com pontos a partir dos dados `mtcars`. Use `?mtcars` para mais detalhes.

```
g1 <- ggplot(mtcars, aes(y = mpg, x = disp)) +
 geom_point()

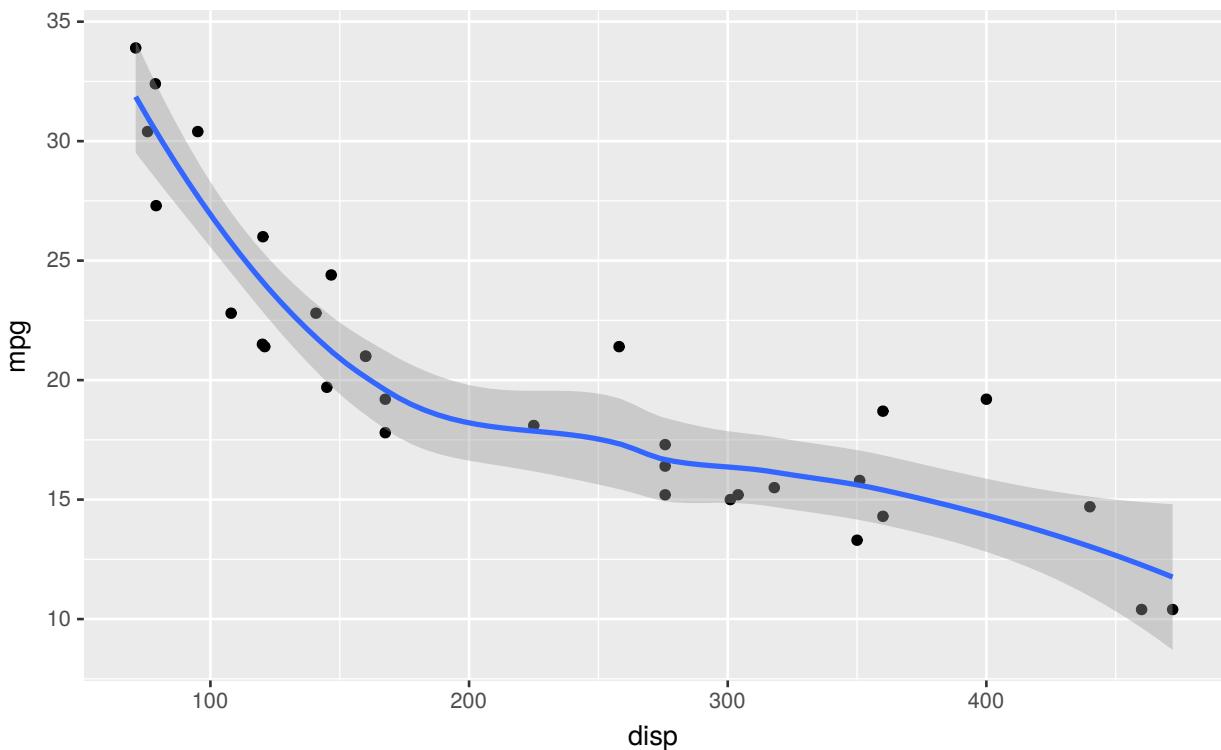
g1
```



Note que o `aes()` está sendo usado diretamente na função `ggplot()`, e não no objeto geométrico. O que isso significa? O mapeamento estético definido na função `ggplot()` é global. Ou seja, é aplicado para todos os objetos geométricos daquele gráfico, a menos que seja explicitado novamente em alguma camada.

Para finalizar essa breve introdução a objetos geométricos, vamos adicionar mais uma camada ao gráfico:

```
library(dplyr)
mtcars <- mtcars %>%
 mutate(name = rownames(mtcars))
ggplot(mtcars, aes(y = mpg, x = disp)) +
 geom_point() +
 geom_smooth()
```



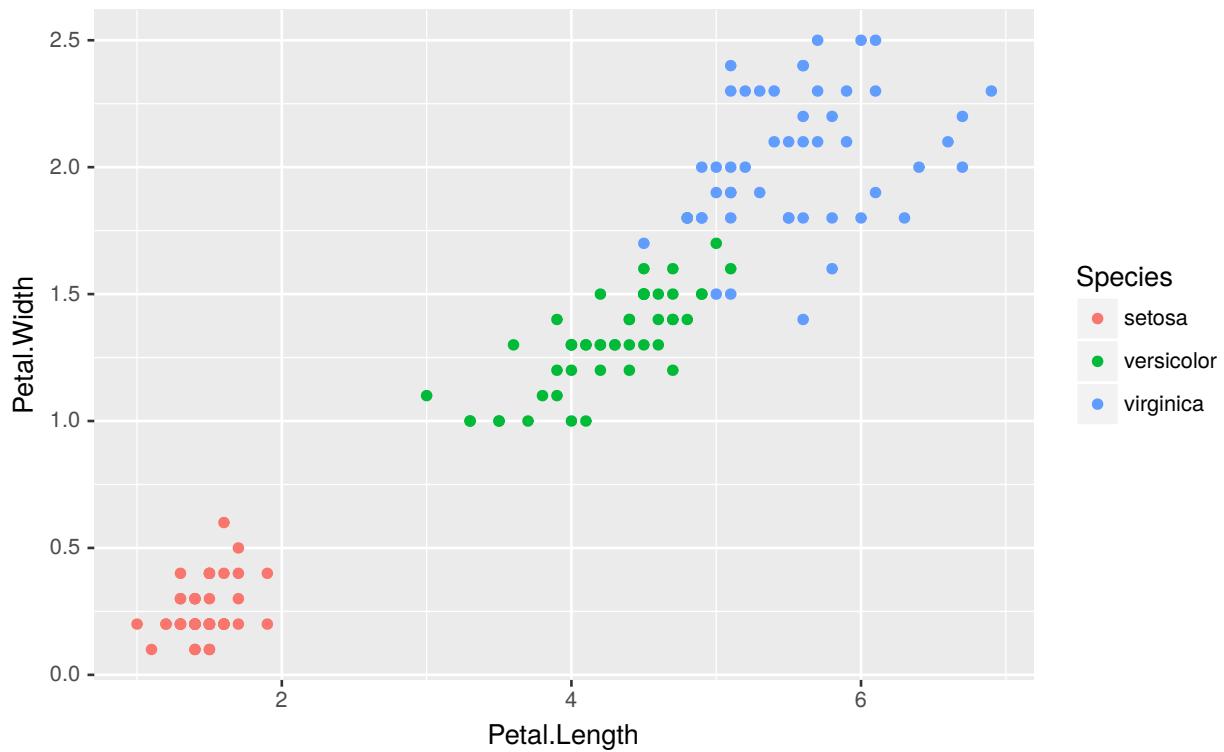
No caso, adicionamos uma curva de ajustamento aos dados que tem o objetivo de evidenciar um padrão nos dados.

### 9.3 Escalas

O controle sobre as escalas do gráfico é fundamental no ajuste de um gráfico. Em geral, o ggplot2, como outros pacotes gráficos, fornecem as escalas automaticamente, não sendo necessário o entendimento de como controlar esse componente. No entanto, se o interesse é ter controle sobre todos os aspectos de um gráfico, esse componente é fundamental.

Veja o gráfico abaixo:

```
ggplot(iris, aes(x = Petal.Length, y = Petal.Width, color = Species)) +
 geom_point()
```

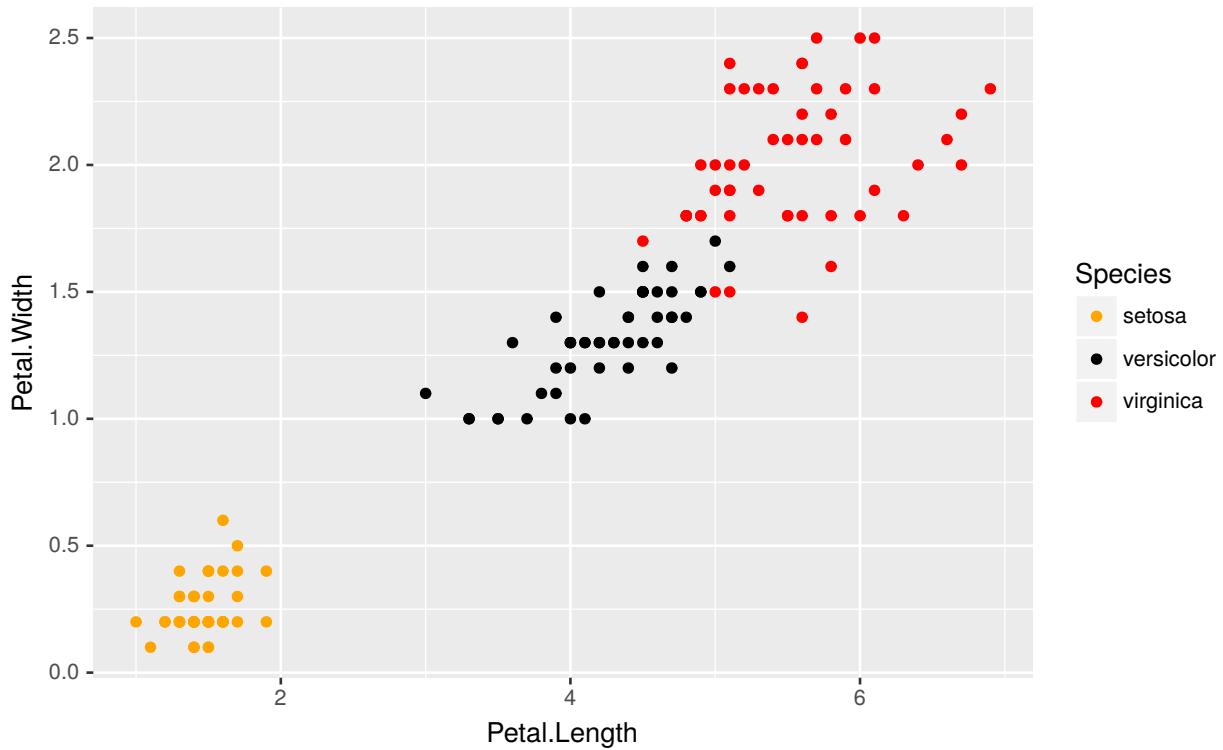


Note que a cor está mapeada para a variável `Species`. O `ggplot2` automaticamente criou a seguinte escala:

| Species    | Cor      |
|------------|----------|
| setosa     | Vermelho |
| versicolor | Verde    |
| virginica  | azul     |

Todavia, é comum haver interesse em alterar essas cores, ou seja, alterar a escala de cor. Como fazer isso no `ggplot2`? Podemos usar, por exemplo, a função `scale_color_manual()`.

```
ggplot(iris, aes(x = Petal.Length, y = Petal.Width, color = Species)) +
 geom_point() +
 scale_color_manual(values = c("orange", "black", "red"))
```

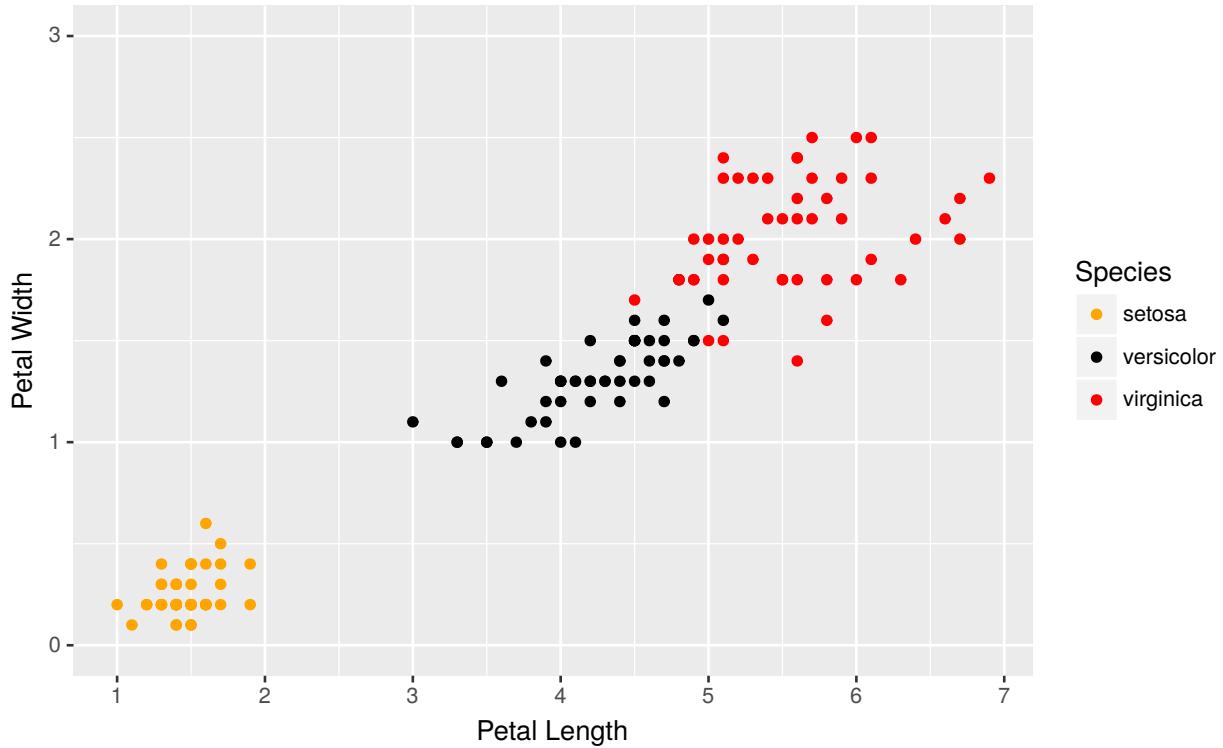


Usamos a função `scale_color_manual()` em razão da variável `Species` ser categórica. Para o ggplot2, dados categóricos são discretos, e a função citada permite criar uma escala discreta customizada. No entanto, essa não é a única função para controlar escala de cor. Existem outras como `scale_color_discrete()`, `scale_color_continuous()`, `scale_color_gradient()` etc. A utilização de cada função depende do tipo de dado que está associado ao elemento estético `color`. Adiante, entraremos em mais detalhes sobre os tipos de dados.

As funções utilizadas para controlar as escalas dos elementos de um gráfico do ggplot2 seguem um padrão. Todas iniciam-se com `scale_`, depois o nome do elemento estético (color, fill, x etc.) e, por fim, o tipo/nome da escala que será aplicada.

Abaixo, continuamos o exemplo anterior, alterando as escalas dos eixos x e y. Note que as variáveis `Petal.Length` e `Petal.Width` são variáveis numéricas/contínuas. Dessa forma, iremos utilizar as funções `scale_x_continuous()` e `scale_y_continuous()`:

```
ggplot(iris, aes(x = Petal.Length, y = Petal.Width, color = Species)) +
 geom_point() +
 scale_color_manual(values = c("orange", "black", "red")) +
 scale_x_continuous(name = "Petal Length", breaks = 1:7) +
 scale_y_continuous(name = "Petal Width", breaks = 0:3, limits = c(0, 3))
```



No gráfico acima, definimos quais seriam os pontos em que rótulos deveriam ser exibidos em cada eixo. Além disso, no eixo y, definimos que o limite seriam 0 e 3.

### 9.3.1 Tipos de Variáveis

Para melhor uso das escalas, é preciso saber o tipo de variável que foi mapeado para cada elemento estético. Vamos rapidamente montar essa relação:

| Classe    | Exemplo                                                 | Tipo no ggplot2        |
|-----------|---------------------------------------------------------|------------------------|
| numeric   | seq(0, 1, length.out = 10)                              | continuous             |
| integer   | 1L:10L                                                  | continuous ou discrete |
| character | c("Sim", "Não")                                         | discrete               |
| factor    | factor(c("Sim", "Não"))                                 | discrete               |
| date      | seq(as.Date("2000/1/1"), by = "month", length.out = 12) | date                   |

Lembre-se que o padrão do ggplot é `scale_`, depois o nome do elemento estético (color, fill, x etc.) e, por fim, o tipo/nome da escala que será aplicada. É importante que o usuário saiba o tipo de dado, pois assim saberá com mais facilidade qual é o tipo de escala que deve ser escolhido.

Vamos na sequência entrar em mais detalhes para escalas dos eixos (x e y) e de cores. Espera-se que a intuição desenvolvida a partir dos exemplos das escalas para esses elementos estéticos seja útil para os demais elementos estéticos.

### 9.3.2 Eixos

#### 9.3.2.1 Variáveis Contínuas

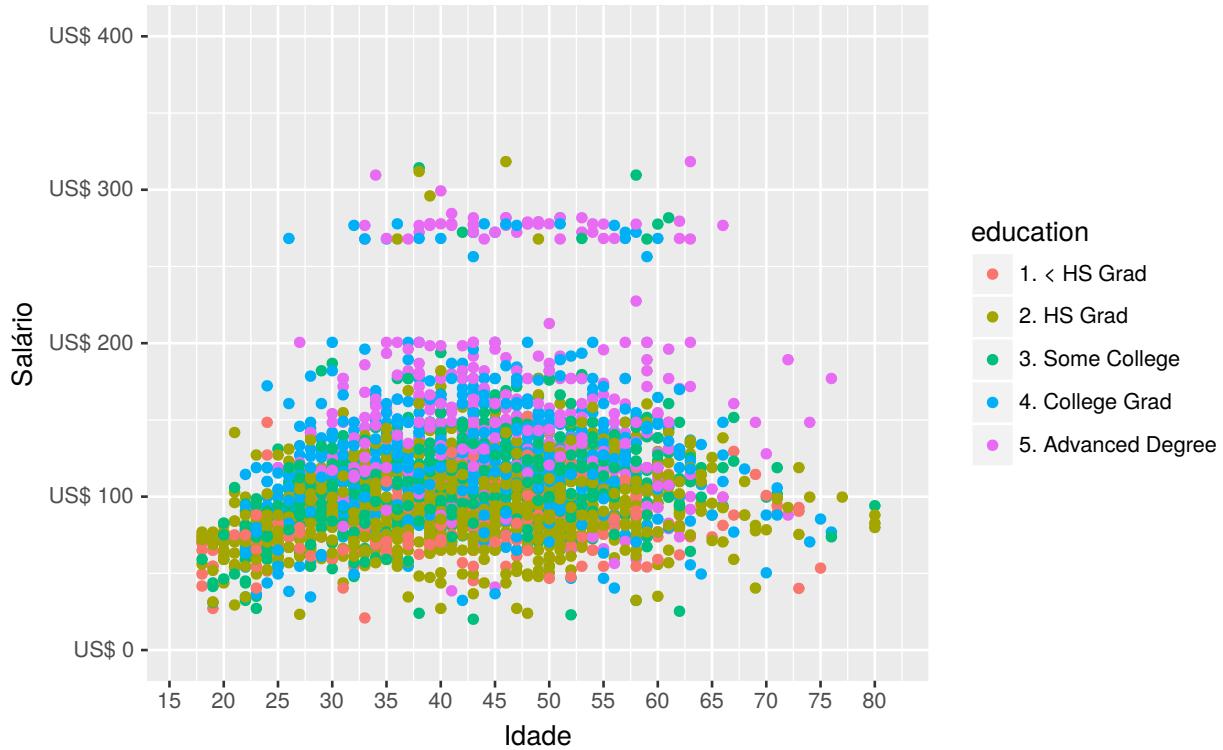
```
scale_x_continuous(name = waiver(), breaks = waiver(), minor_breaks = waiver(),
 labels = waiver(), limits = NULL, expand = waiver(),
 oob = censor, na.value = NA_real_, trans = "identity")

scale_y_continuous(name = waiver(), breaks = waiver(), minor_breaks = waiver(),
 labels = waiver(), limits = NULL, expand = waiver(),
 oob = censor, na.value = NA_real_, trans = "identity")
```

Vamos começar editando os valores dos eixos x e y. Anteriormente, já demos uma pequena amostra sobre a edição dos eixos x e y. Abaixo, será apresentado mais um exemplo:

```
library(ISLR)

ggplot(Wage, aes(x = age, y = wage, color = education)) +
 geom_point() +
 scale_x_continuous("Idade", breaks = seq(0, 80, 5),
 expand = c(0, 5)) +
 scale_y_continuous("Salário", labels = function(x) paste0("US$ ", x),
 limits = c(0, 400))
```



Para o eixo x, determinamos que as quebras (`breaks`) acontecem a cada 5 anos. O `expand(0,0)` é um argumento que controla os espaços adicionais no final do gráfico. É preciso fornecer um vetor de tamanho 2, com uma constante multiplicativa e outra aditiva. No exemplo acima, eliminamos a expansão.

Para o eixo y, informamos que o nome do eixo é Salário, que os limites inferior e superior são 0 e 400, e

alteramos os rótulos. No caso, passamos uma função que concatena US\$ com o valor que já seria exibido. Note que, sabendo de antemão todos os breaks, é possível definir manualmente os labels. Veremos isso no exemplo com variáveis categóricas. No entanto, para variáveis contínuas, o uso de funções parece mais apropriado.

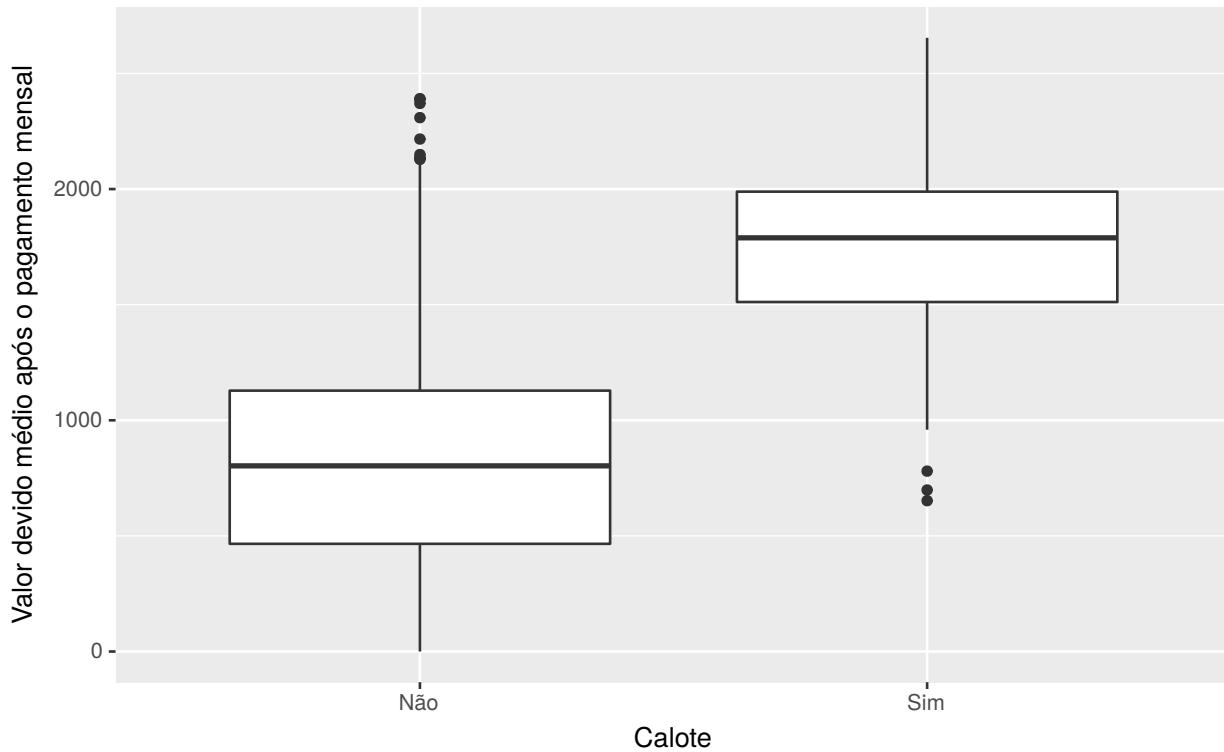
### 9.3.2.2 Variáveis discretas

Apesar do *help* não apresentar todos os argumentos para as escalas discretas, podemos usar quase todos que foram listados para escala contínua.

```
scale_x_discrete(..., expand = waiver(), position = "bottom")
scale_y_discrete(..., expand = waiver(), position = "left")
```

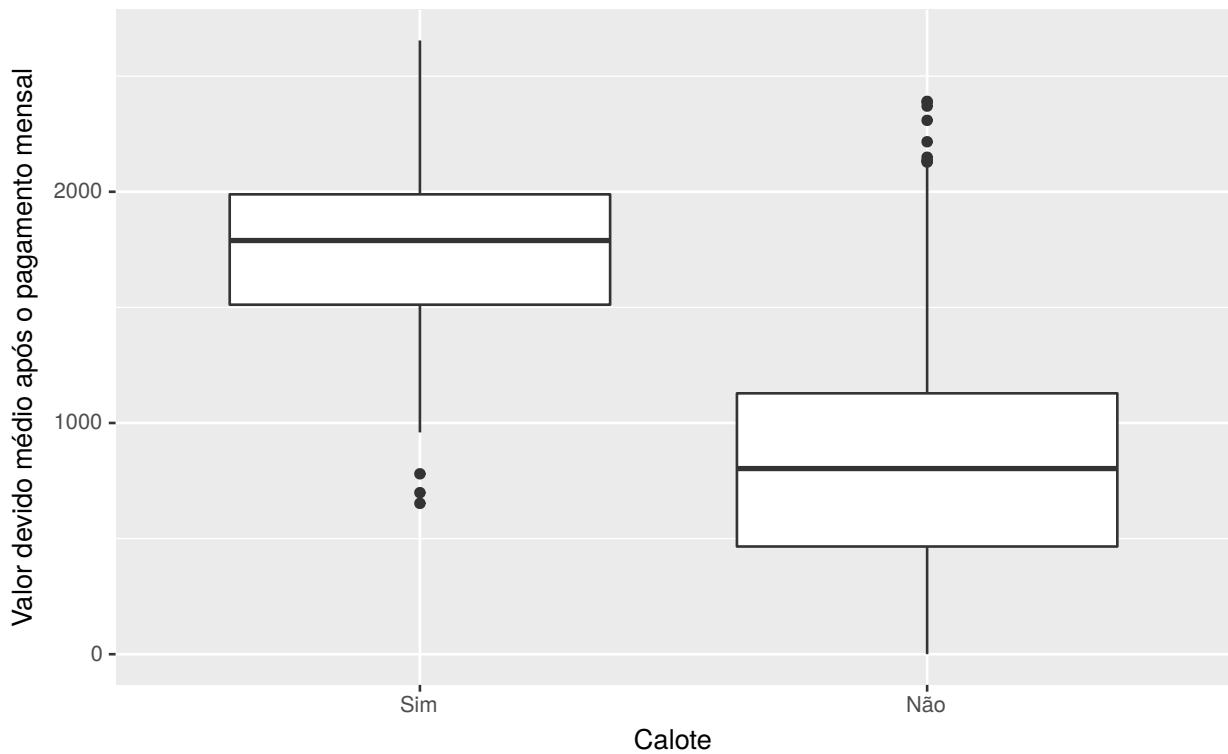
No exemplo abaixo, iremos alterar os rótulos para uma escala discreta que originalmente contém os valores Yes e No.

```
ggplot(Default, aes(x = default, y = balance)) +
 geom_boxplot() +
 scale_x_discrete("Calote", labels = c("Não", "Sim")) +
 labs(y = "Valor devido médio após o pagamento mensal")
```



Pode-se usar o argumento `limits` para alterar a ordem das categorias.

```
ggplot(Default, aes(x = default, y = balance)) +
 geom_boxplot() +
 scale_x_discrete("Calote", limits = c("Yes", "No"),
 labels = c("Sim", "Não")) +
 labs(y = "Valor devido médio após o pagamento mensal")
```



Também pode-se alterar a ordem de variáveis categóricas alterando a ordem dos níveis (`levels`) da variável no data.frame original, ou usando as funções `ylim()` e `xlim()`. Experimente.

### 9.3.2.3 Variáveis de Datas

Quando estamos trabalhando com séries temporais é comum que datas sejam associadas a algum eixo do gráfico, geralmente ao eixo `x`. As funções padrões para controle de escalas dos eixos para variáveis de datas são as seguintes:

```
scale_x_date(name = waiver(), breaks = waiver(), date_breaks = waiver(),
 labels = waiver(), date_labels = waiver(), minor_breaks = waiver(),
 date_minor_breaks = waiver(), limits = NULL, expand = waiver())

scale_y_date(name = waiver(), breaks = waiver(), date_breaks = waiver(),
 labels = waiver(), date_labels = waiver(), minor_breaks = waiver(),
 date_minor_breaks = waiver(), limits = NULL, expand = waiver())

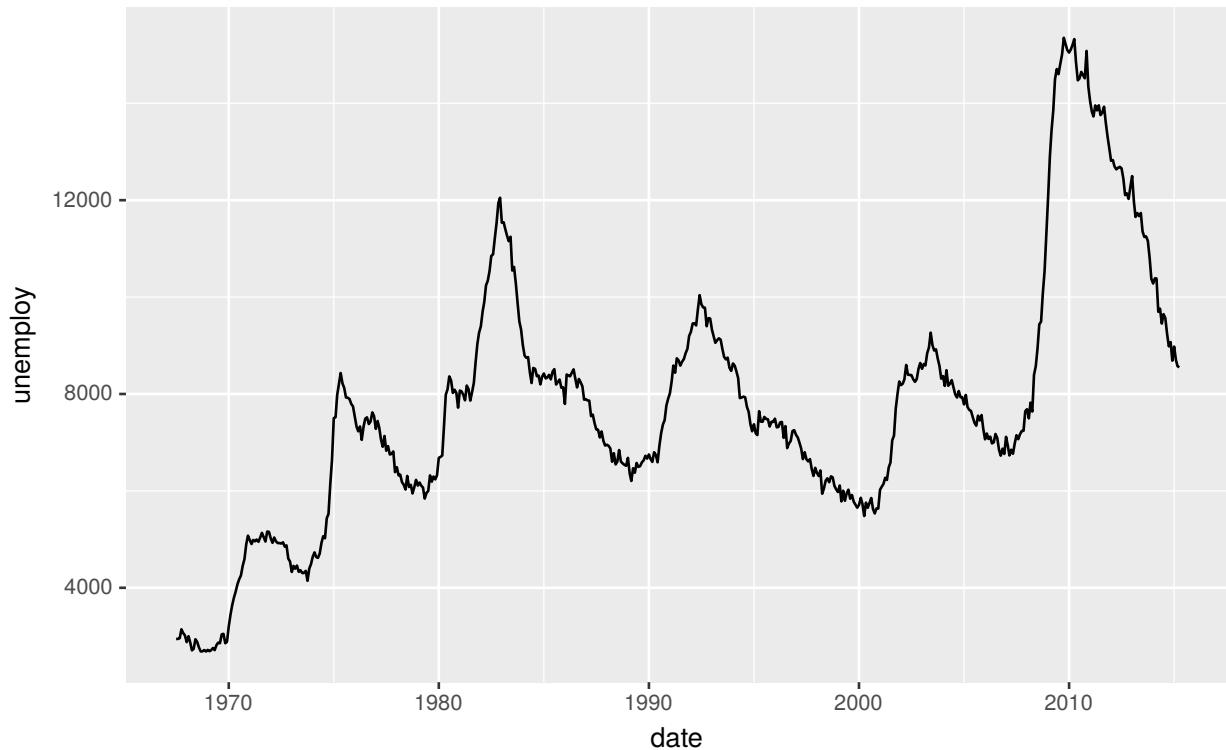
scale_x_datetime(name = waiver(), breaks = waiver(), date_breaks = waiver(),
 labels = waiver(), date_labels = waiver(), minor_breaks = waiver(),
 date_minor_breaks = waiver(), limits = NULL, expand = waiver())

scale_y_datetime(name = waiver(), breaks = waiver(), date_breaks = waiver(),
 labels = waiver(), date_labels = waiver(), minor_breaks = waiver(),
 date_minor_breaks = waiver(), limits = NULL, expand = waiver())
```

`scale_*_date` é utilizado para variáveis do tipo `Date` e `scale_*_datetime` para variáveis do tipo `POSIXct`. A classe `POSIXct` aceita informações relacionadas a tempo/horário e a classe `Date` aceita apenas dia, mês e ano.

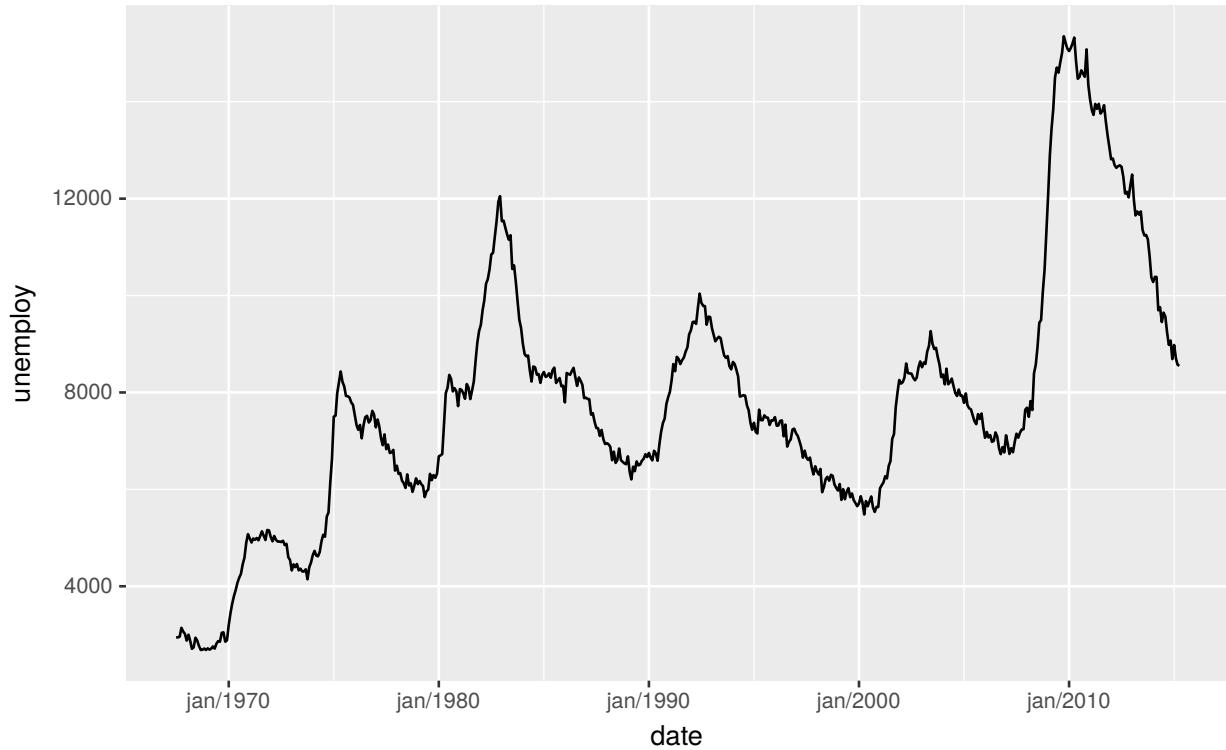
O mais importante é a possibilidade de alterar como as datas são apresentadas a partir do argumento `date_labels`. Para isso, vamos utilizar um exemplo a partir dos dados `economics`. Primeiro, observa-se o resultado padrão do ggplot2:

```
ggplot(economics, aes(x = date, y = unemploy)) +
 geom_line()
```



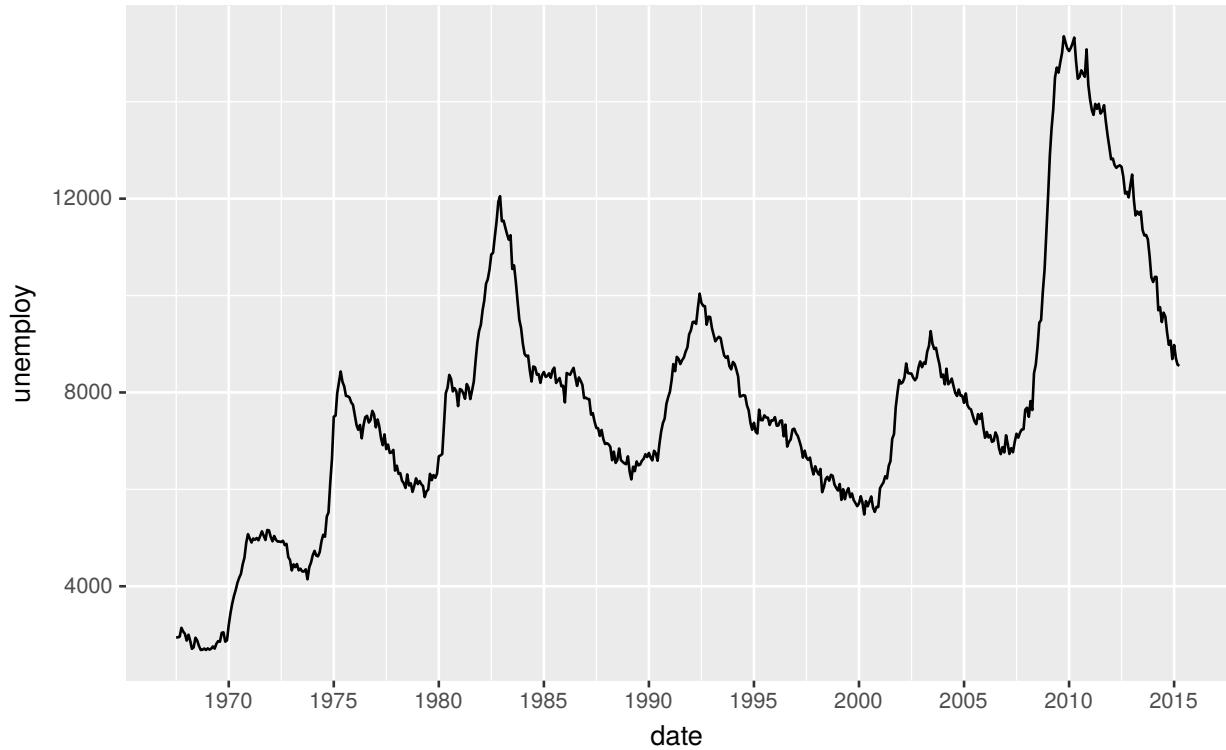
Agora, suponha que queremos alterar o gráfico para o formato “Jan/1970”:

```
ggplot(economics, aes(x = date, y = unemploy)) +
 geom_line() +
 scale_x_date(date_labels = "%b/%Y")
```

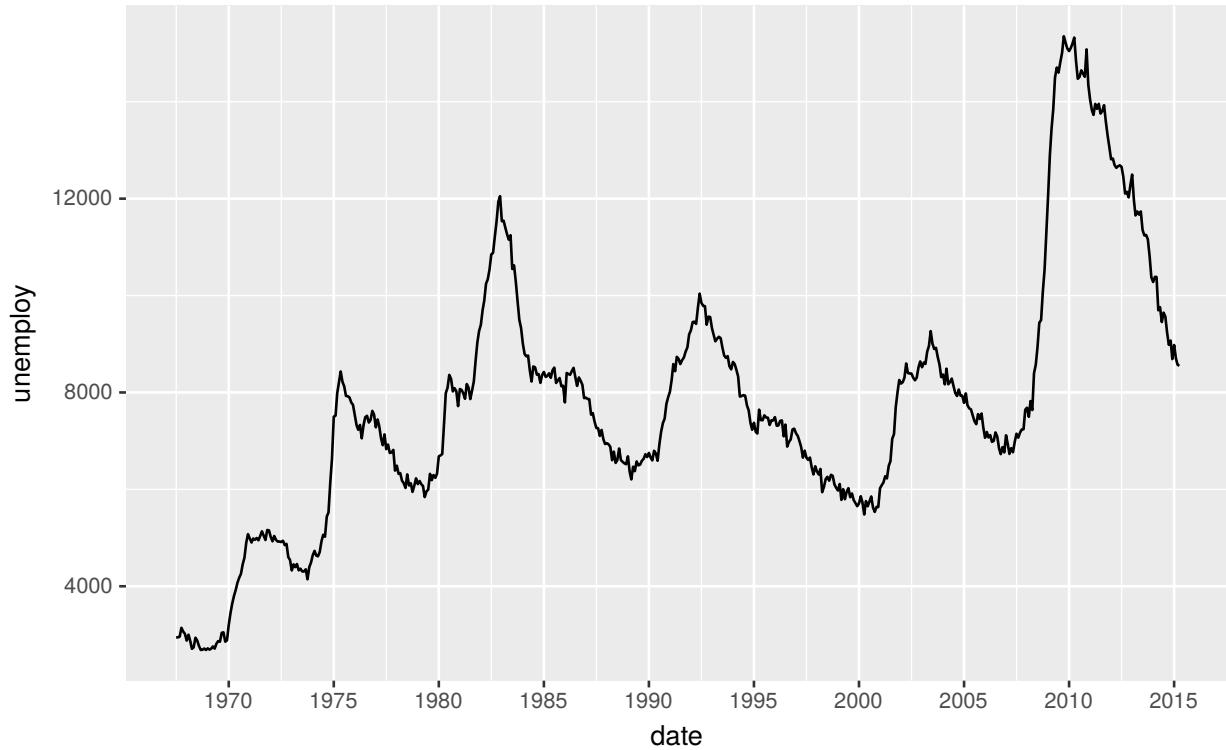


O uso do `%b/%Y` é usado para definir o formato de data desejado. Para ver a lista de formatos, use `help(strptime)`. Para os `breaks`, temos duas opções. A primeira é utilizar o argumento `breaks` informando um vetor de datas ou usando o argumento `date_breaks`, em que se informa a frequência dos breaks (por exemplo, “1 month” e “5 years”). Veja os exemplos abaixo:

```
ggplot(economics, aes(x = date, y = unemploy)) +
 geom_line() +
 scale_x_date(date_breaks = "5 years", date_labels = "%Y")
```



```
seq_datas <- seq.Date(as.Date('1970-01-01'),
 as.Date('2015-04-01'),
 by = '5 years')
ggplot(economics, aes(x = date, y = unemploy)) +
 geom_line() +
 scale_x_date(breaks = seq_datas, date_labels = "%Y")
```



### 9.3.3 Escalas de Cores (color) e Preenchimento (fill)

Como nos casos dos eixos x e y, o tipo da variável utilizada define qual o tipo de escala.

| Tipo da Variável | Escala          | Descrição                                                                                                  |
|------------------|-----------------|------------------------------------------------------------------------------------------------------------|
| Discreta         | <b>hue</b>      | escolhe n cores igualmente espaçadas em um disco de cores. É possível editar a luminosidade e a saturação. |
|                  | grey            | escala de cinza                                                                                            |
|                  | brewer          | ver pacote RColorBrewer                                                                                    |
|                  | identity        | usa as cores inseridas na própria variável                                                                 |
|                  | manual          | escolher as cores manualmente                                                                              |
|                  | <b>gradient</b> | cria um gradiente de duas cores (low-high)                                                                 |
| Contínua         | gradient2       | cria um gradiente de cores divergentes (low-mid-high)                                                      |
|                  | gradientn       | cria um gradiente com n cores                                                                              |

A opção **hue** usa a seguinte roda de cores:

A opção brewer pode usar as paletas de cores disponíveis no pacote RColorBrewer.

```
library(RColorBrewer)
display.brewer.all(n=NULL, type="all", select=NULL, exact.n=TRUE,
 colorblindFriendly=FALSE)
```

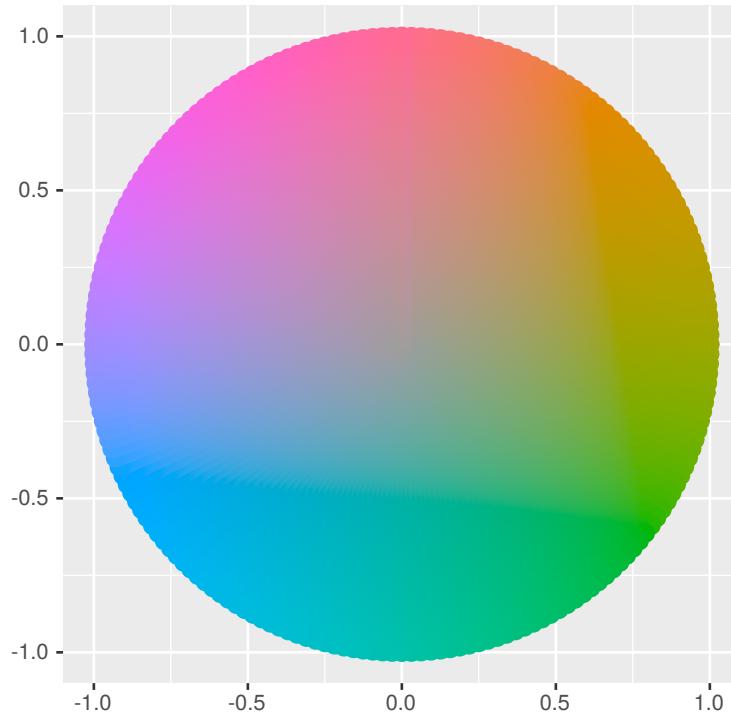
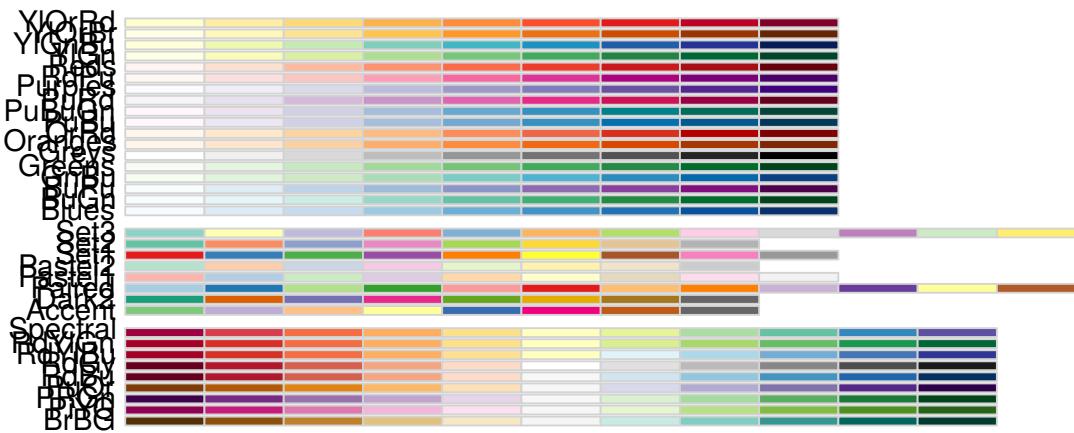


Figura 9.1: Roda de Cores - hue



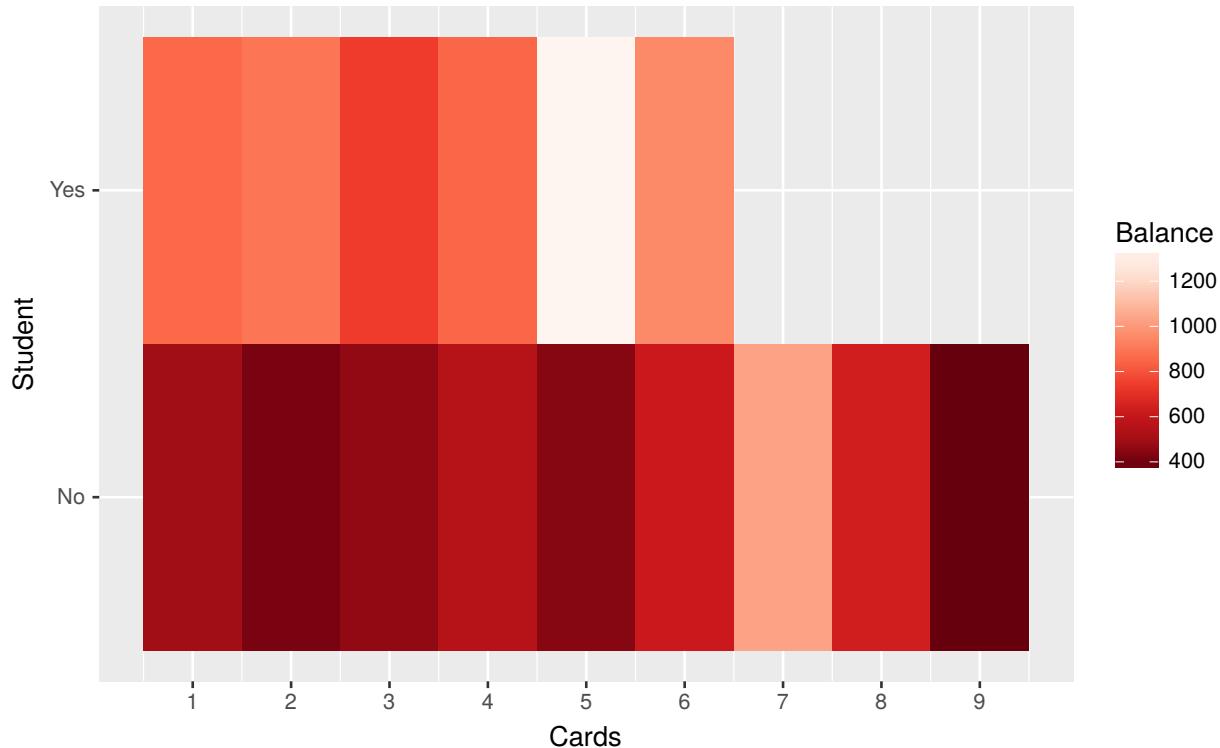
No exemplo abaixo, vamos utilizar a função `brewer.pal` que retorna um vetor de cores de alguma paleta do

pacote RColorBrewer. O objeto `paleta.gradientn` recebe 9 cores da paleta `Reds`. Essas cores são utilizadas na função `scale_fill_gradientn()`.

```
paleta.gradientn <- brewer.pal(n = 10, name = 'Reds')
```

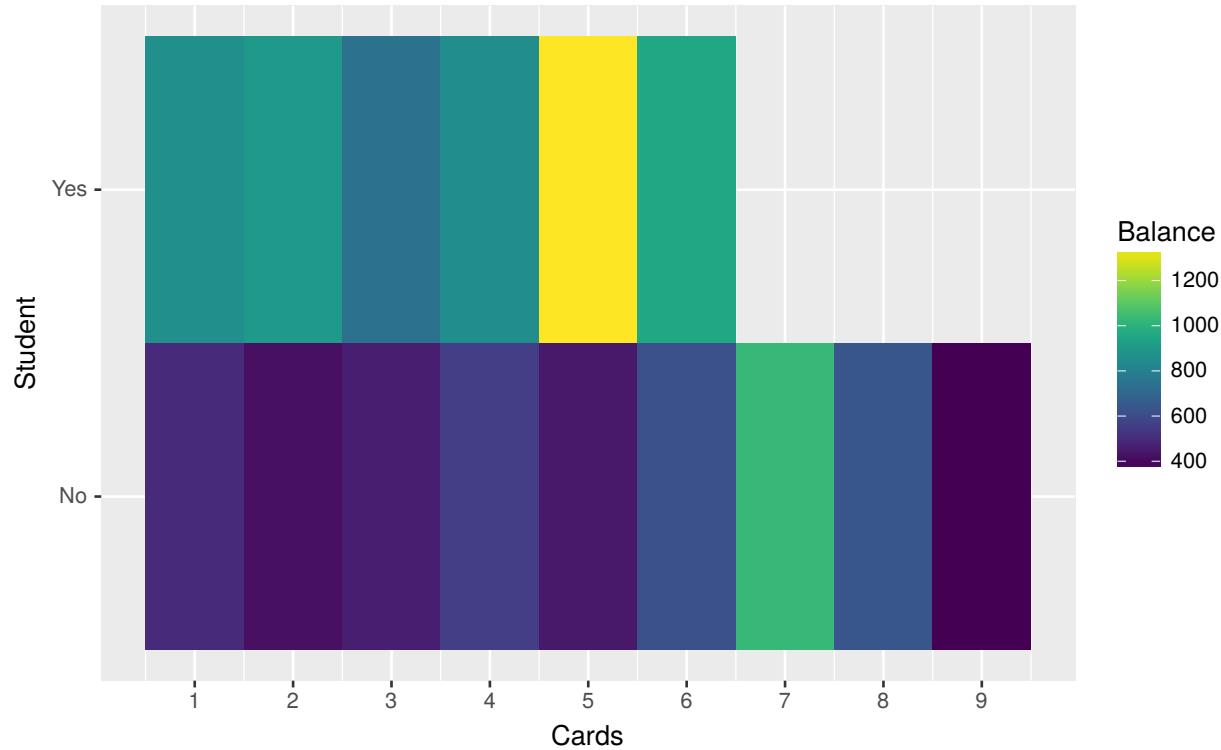
```
Warning in brewer.pal(n = 10, name = "Reds"): n too large, allowed maximum for palette Reds is 9
Returning the palette you asked for with that many colors
```

```
Credit %>%
 group_by(Cards, Student) %>%
 summarise(Balance = mean(Balance), n = n()) %>%
 ggplot(aes(x = Cards, y = Student, fill = Balance)) +
 geom_tile() +
 scale_fill_gradientn(colors = rev(paleta.gradientn)) +
 scale_x_continuous(breaks = 1:9)
```



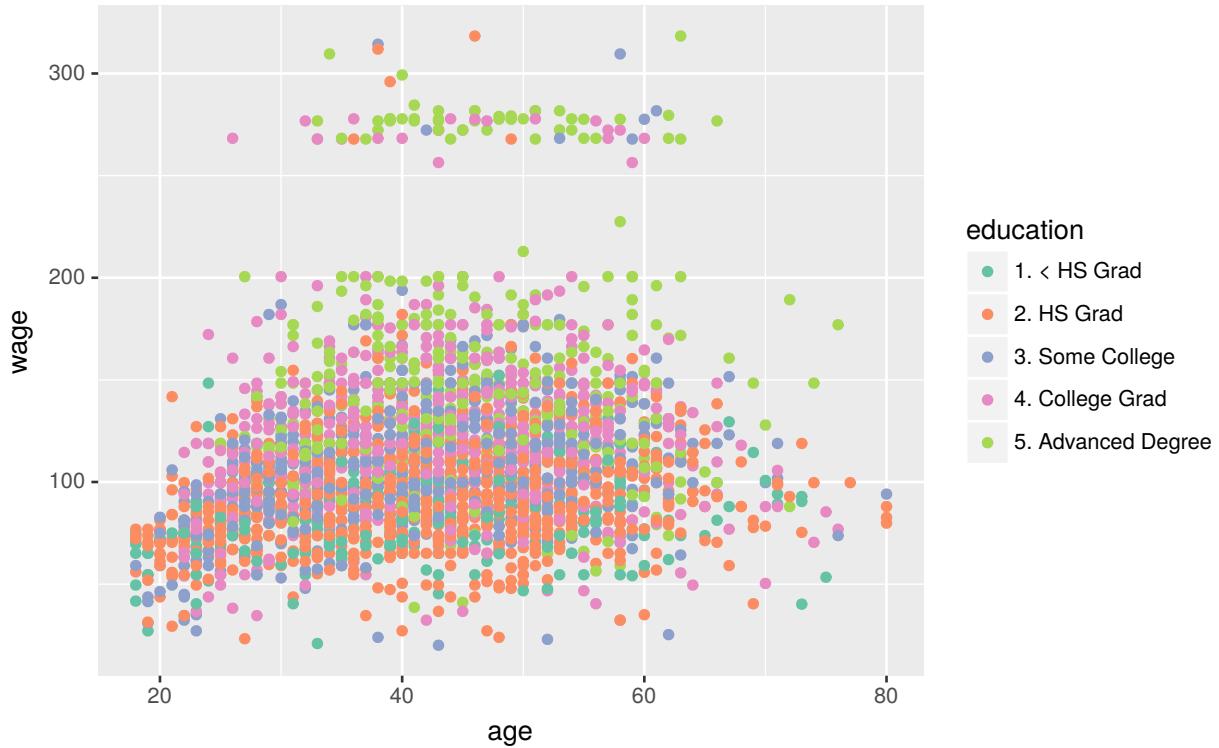
Alguns pacotes também fornecem escalas de cores próprias, como é o caso do pacote `viridis`.

```
Credit %>%
 group_by(Cards, Student) %>%
 summarise(Balance = mean(Balance), n = n()) %>%
 ggplot(aes(x = Cards, y = Student, fill = Balance)) +
 geom_tile() +
 viridis::scale_fill_viridis() +
 scale_x_continuous(breaks = 1:9)
```



Agora, vamos fazer um exemplo usando o `scale_color_manual()`.

```
ggplot(Wage, aes(y = wage, x = age, color = education)) +
 geom_point() +
 scale_color_manual(values = c("#66C2A5", "#FC8D62", "#8DA0CB", "#E78AC3", "#A6D854"))
```



Aqui fizemos uma introdução ao componente de escalas. Não tratamos de todas as funções de escalas. A ideia é passar a lógica geral para se utilizar escalas, e não que o usuário decore todas as funções, o que contraproducente.

## 9.4 Subplots (facet)

O ggplot2 facilita a criação de subplots no caso em que se deseja replicar o mesmo gráfico para um conjunto de valores de outra variável. Por exemplo, criar um gráfico da série temporal de desemprego para cada Unidade da Federação. As duas principais funções são `facet_wrap()` e `facet_grid()`.

- Painéis em formato de grid:

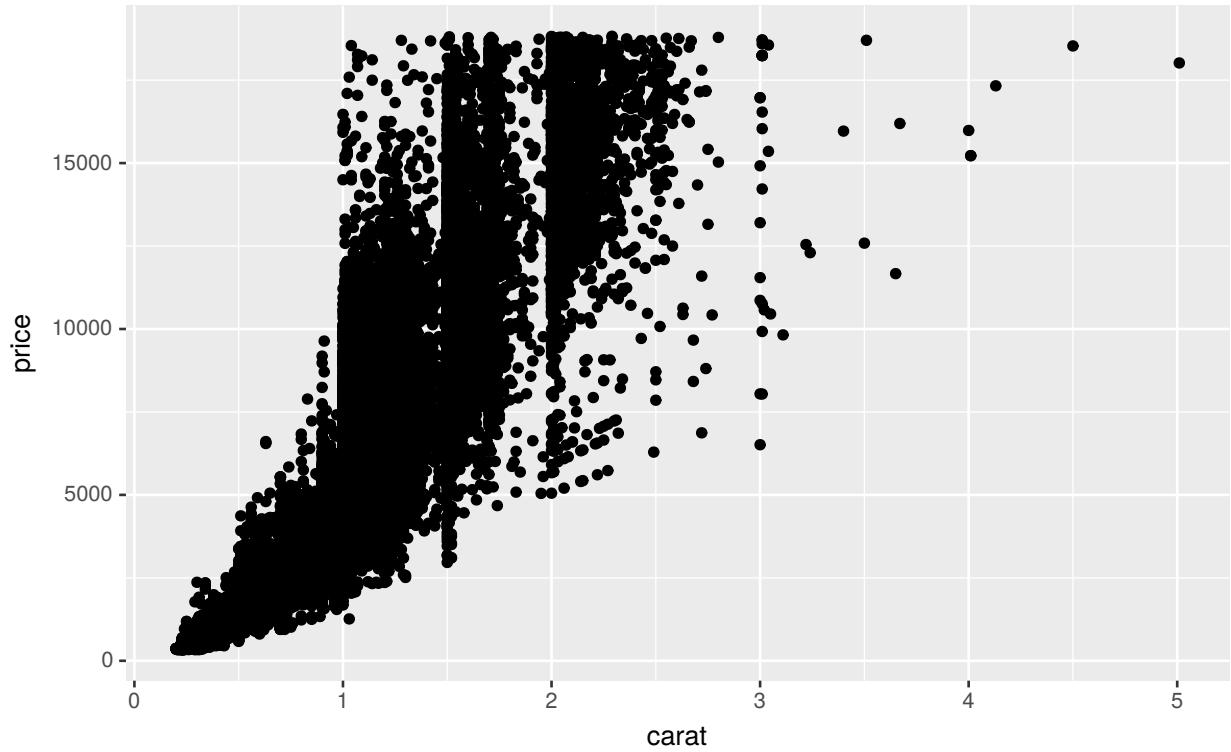
```
facet_grid(facets, margins = FALSE, scales = "fixed", space = "fixed", shrink = TRUE,
 labeller = "label_value", as.table = TRUE, switch = NULL, drop = TRUE)
```

- Converte painéis de uma dimensão para duas dimensões:

```
facet_wrap(facets, nrow = NULL, ncol = NULL, scales = "fixed", shrink = TRUE,
 labeller = "label_value", as.table = TRUE, switch = NULL, drop = TRUE,
 dir = "h")
```

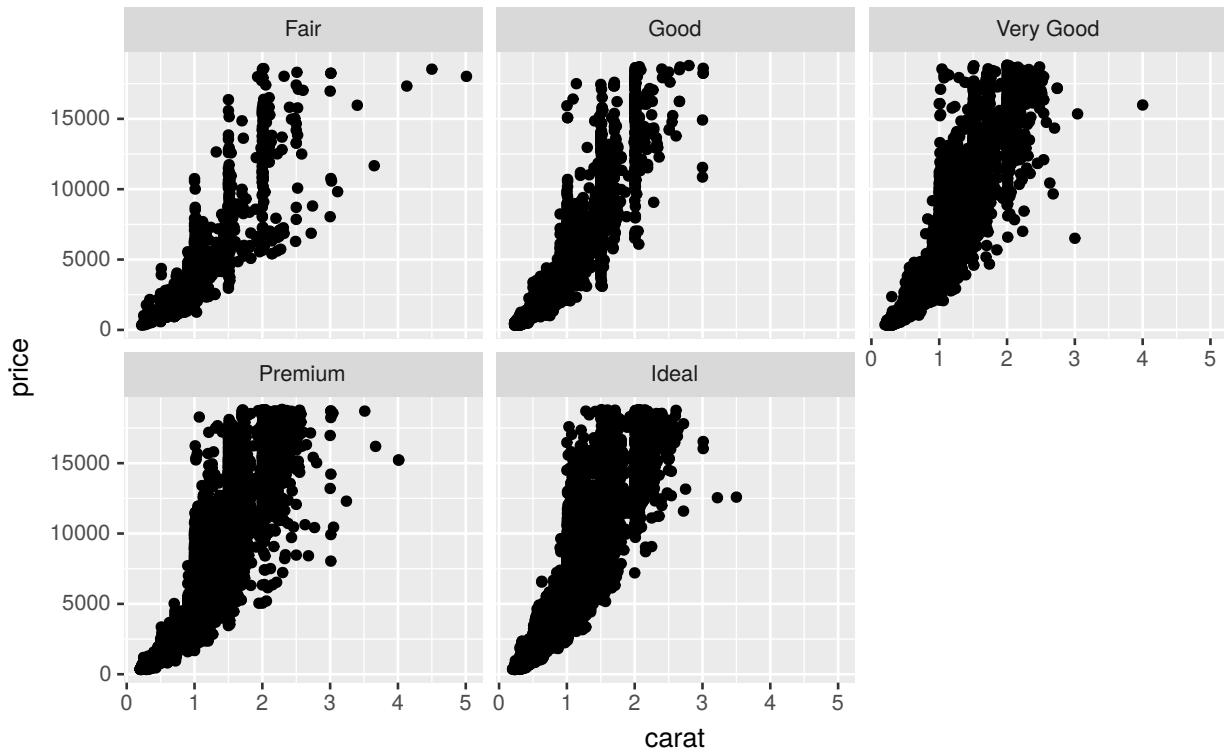
Primeiramente, vamos criar um exemplo para o `facet_wrap()`:

```
ggplot(diamonds, aes(x = carat, y = price)) +
 geom_point()
```



E se o objetivo for comparar essas relações para diferentes grupos de lapidação?

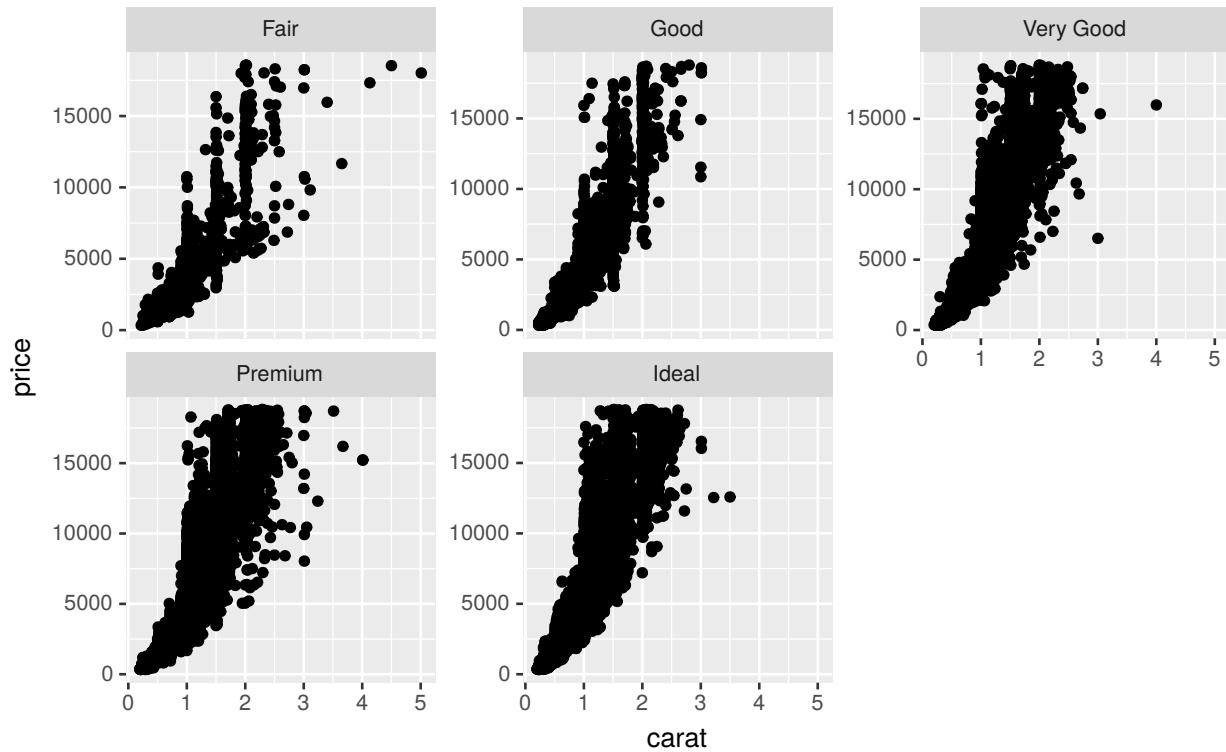
```
ggplot(diamonds, aes(x = carat, y = price)) +
 geom_point() +
 facet_wrap(~ cut)
```



Usamos a fórmula `~ cut`. Ou seja, indicamos que queremos quebrar os gráficos pela variável `cut`.

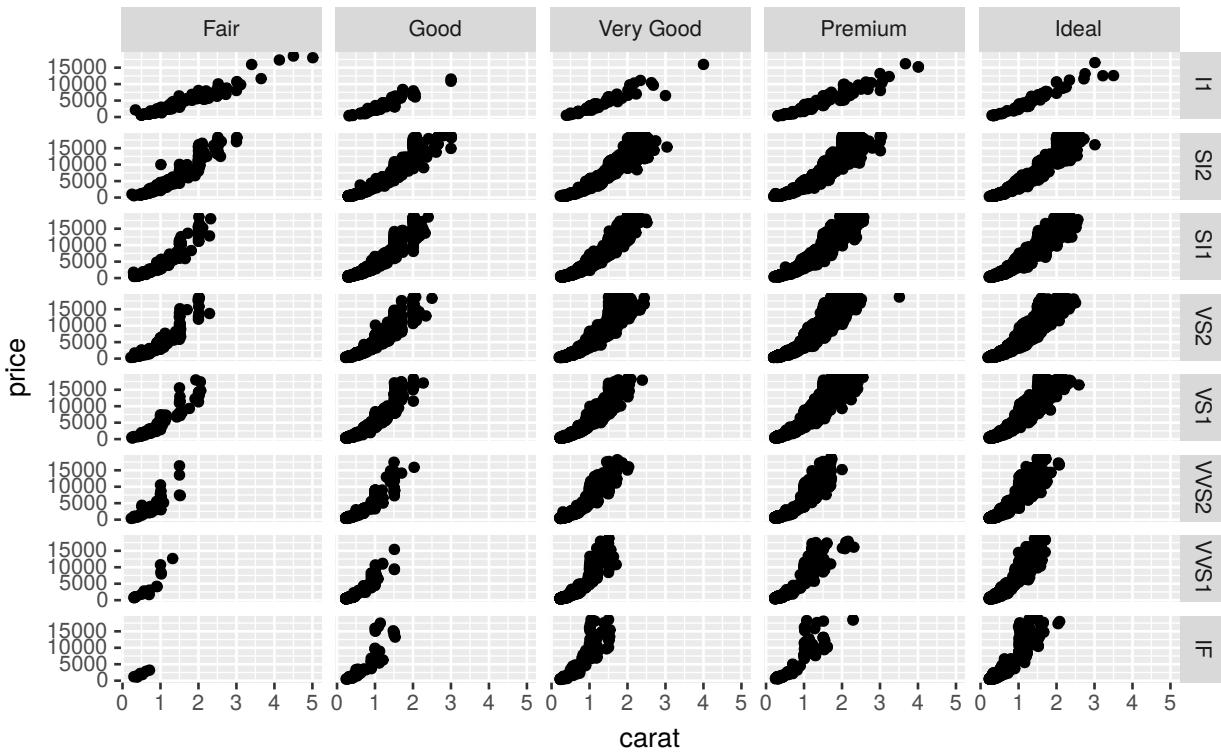
O `ggplot2` determinou automaticamente o número de colunas e linhas e fixou as escalas dos eixos. No entanto, podemos definir o número de linha ou colunas a partir dos argumentos `nrow` e `ncol`. Também é possível definir que cada gráfico tenha sua escala. No exemplo abaixo, vamos deixar a escala do eixo y livre.

```
ggplot(diamonds, aes(x = carat, y = price)) +
 geom_point() +
 facet_wrap(~ cut, scales = "free_y")
```



Já o uso do `facet_grid()` é indicado para o cruzamento de variáveis. No exemplo abaixo, a relação entre as variáveis `price` e `carat` será “quebrada” para grupos formados pelas variáveis `cut` e `clarity`:

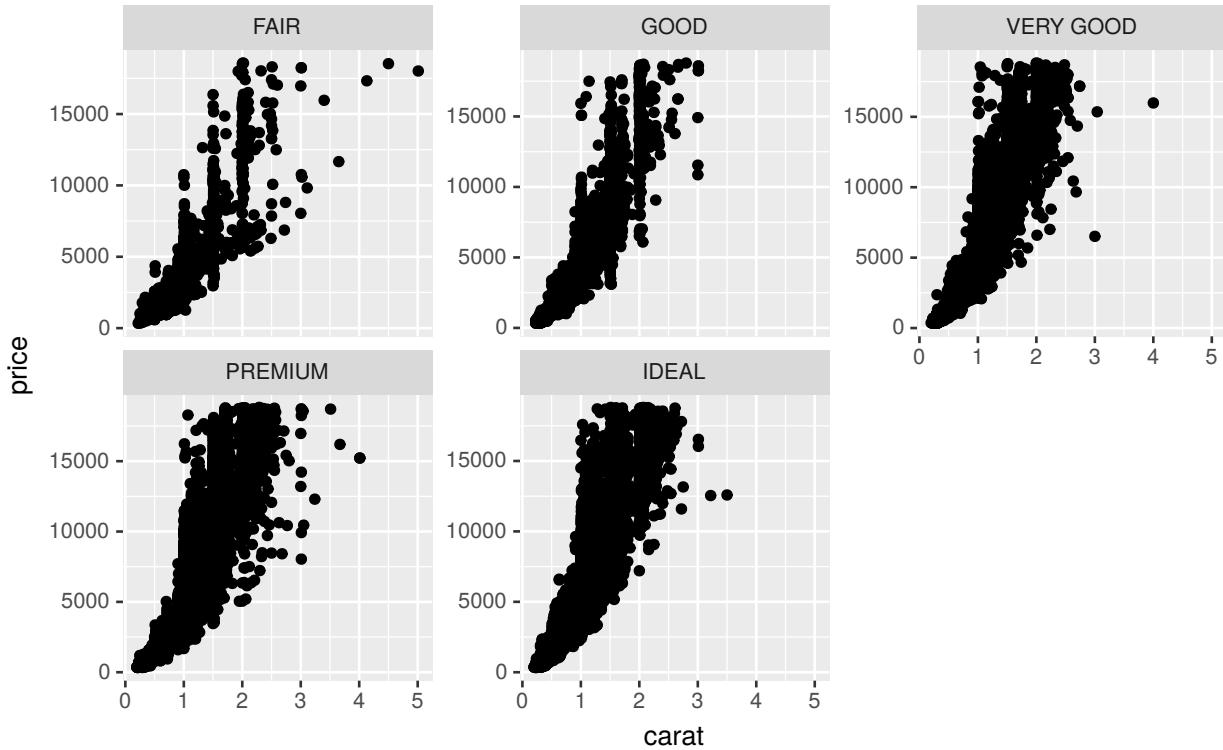
```
ggplot(diamonds, aes(x = carat, y = price)) +
 geom_point() +
 facet_grid(clarity ~ cut)
```



Note que para cada categoria existe um rótulo. Para alterar esse relatório, pode-se alterar diretamente o `data.frame` ou usar a função `labeller()`.

```
nomes_cut <- c(
 Fair = "FAIR",
 Good = "GOOD",
 `Very Good` = "VERY GOOD",
 Premium = "PREMIUM",
 Ideal = "IDEAL"
)

ggplot(diamonds, aes(x = carat, y = price)) +
 geom_point() +
 facet_wrap(~ cut, scales = "free_y",
 labeller = labeller(cut = nomes_cut))
```



## 9.5 Temas

O `ggplot2` fornece alguns temas prontos. Todavia, o usuário pode alterar manualmente cada detalhe de um gráfico ou criar um tema que será utilizado em outras visualizações. Para editar o tema, será usada a função `theme()`. Nessa função, poderão ser alterados os elementos do tema, como a cor de fundo do painel, o tamanho da fonte do eixo x, a posição da legenda etc. A lista de elementos estão disponíveis neste link.

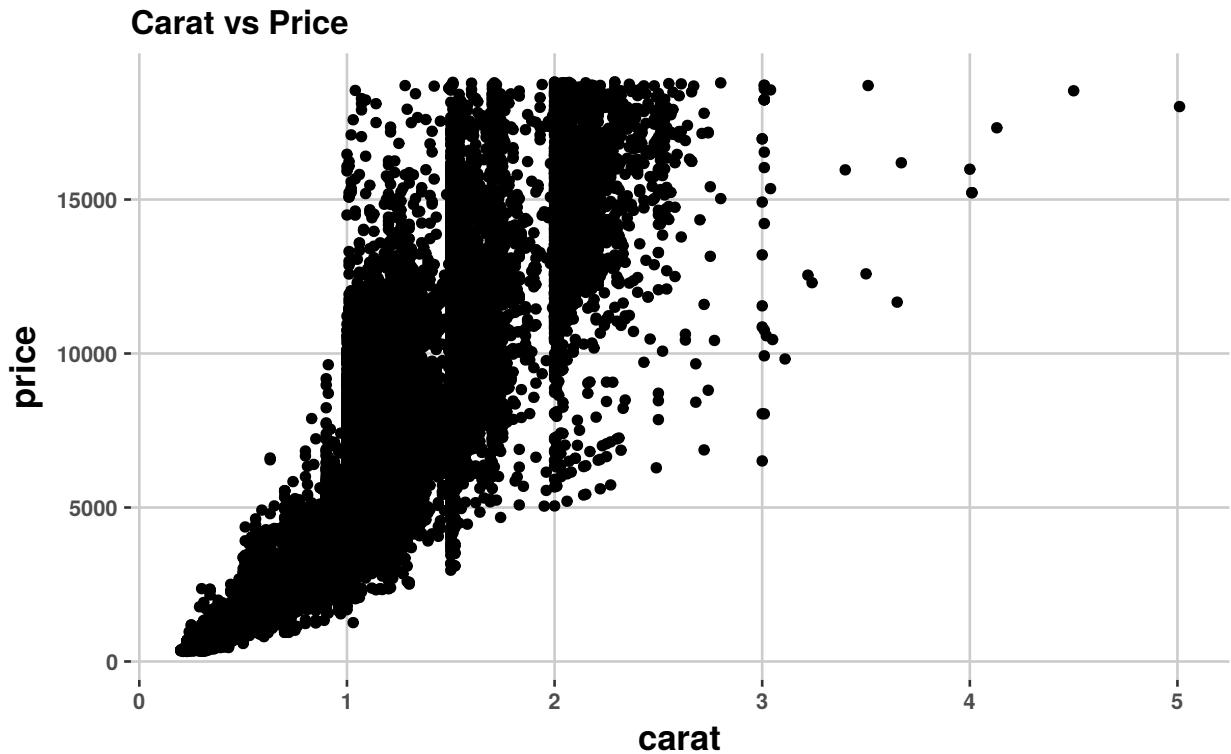
Para cada elemento do tema, um tipo de objeto é esperado para realizar alterações. Por exemplo, para alterar o estilo do título do eixo x (`axis.title.x`) é preciso passar a função `element_text()`, que possui diversos parâmetros (família da fonte, tipo da fonte, cor, tamanho, alinhamento etc.). Além do `element_text()`, as principais funções para alterar elementos do tema são `element_line()`, `element_rect()` e `element_blank()`. O `element_blank()` é usado para que nada seja desenhado para o elemento que recebe essa função.

Em um primeiro momento, pode parecer complicado ir alterando o tema via código. Porém, conforme o usuário for praticando, essas alterações vão ficar mais intuitivas. De toda forma, existe um addin para o RStudio que ajuda a customizar um gráfico do `ggplot2` a partir de uma interface de *point and click*. Para instalá-lo, faça o seguinte:

```
install.packages('ggThemeAssist')
```

Para exemplificar a alteração do tema manualmente,

```
ggplot(diamonds, aes(x = carat, y = price)) +
 geom_point() +
 labs(title = "Carat vs Price") +
 theme(text = element_text(face = "bold"),
 panel.grid.major = element_line(colour = "gray80"),
 axis.title = element_text(size = 14),
 panel.background = element_rect(fill = "gray100"))
```



#### 9.5.1 Temas disponíveis no ggplot2

```
p <- ggplot(diamonds, aes(x = carat, y = price)) +
 geom_point()
p + theme_gray() +
 labs(title = "theme_gray()")

p + theme_bw() +
 labs(title = "theme_bw()")

p + theme_linedraw() +
 labs(title = "theme_linedraw()")

p + theme_light() +
 labs(title = "theme_light()")

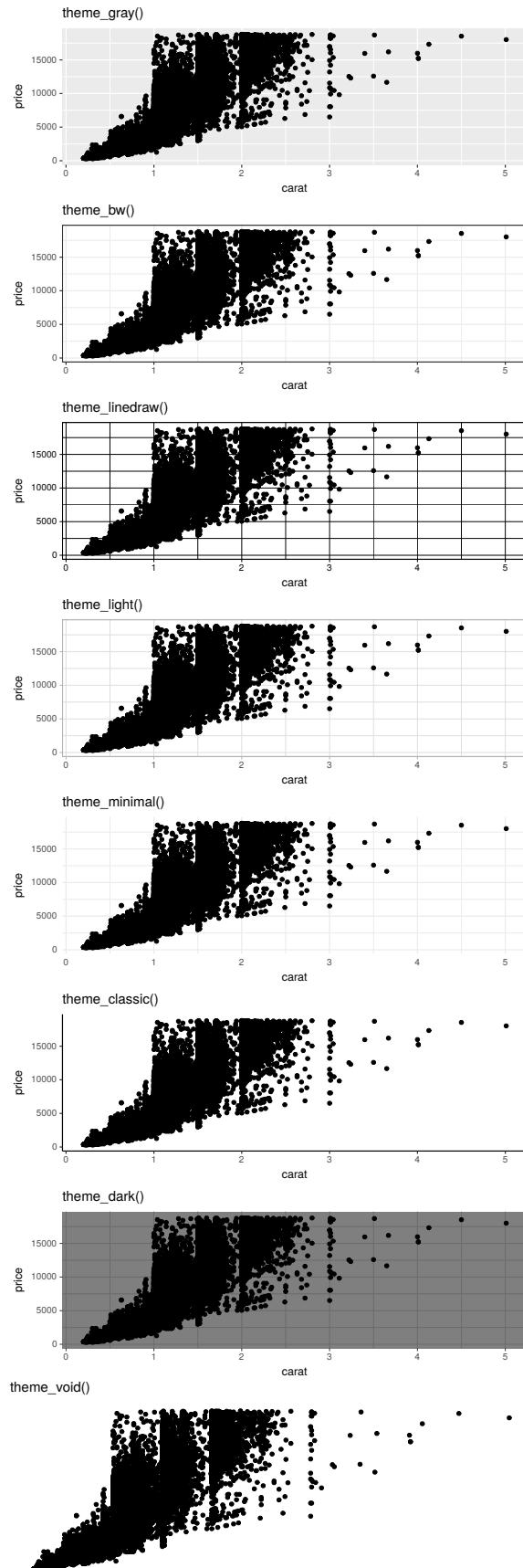
p + theme_minimal() +
 labs(title = "theme_minimal()")

p + theme_classic() +
 labs(title = "theme_classic()")

p + theme_dark() +
 labs(title = "theme_dark()")

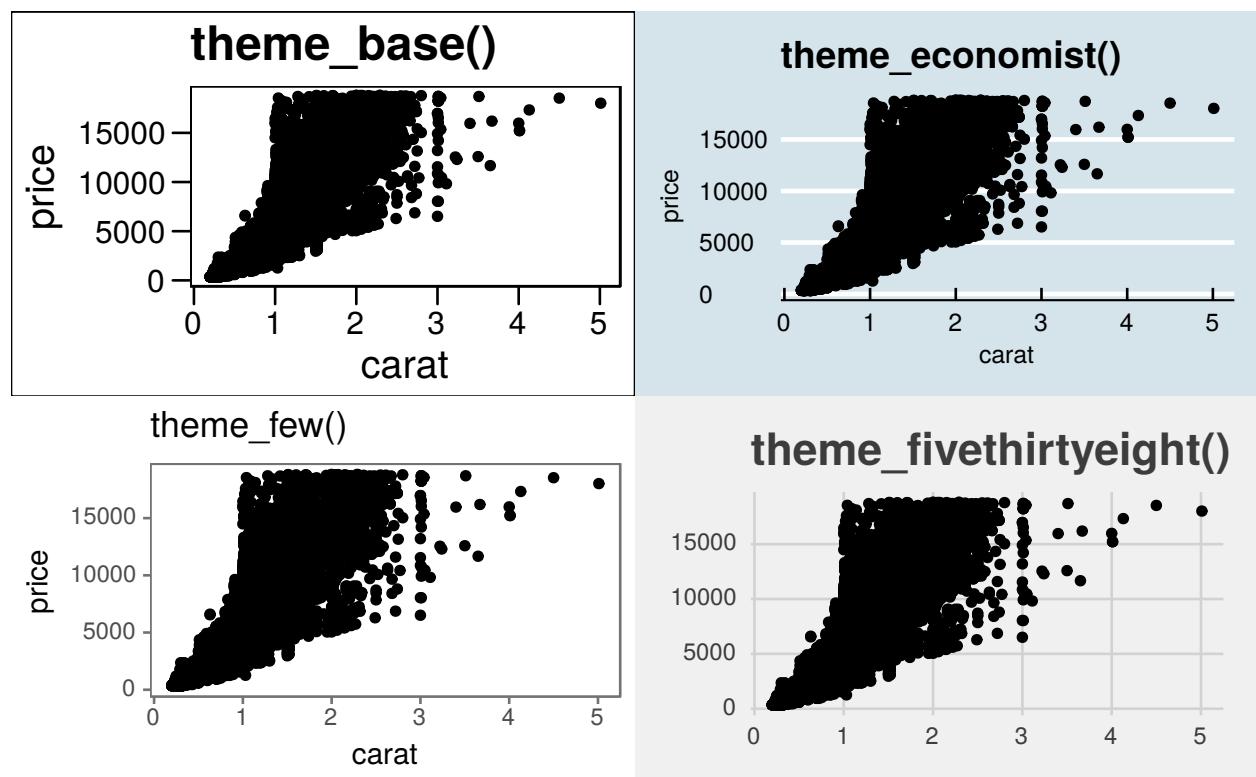
p + theme_void() +
```

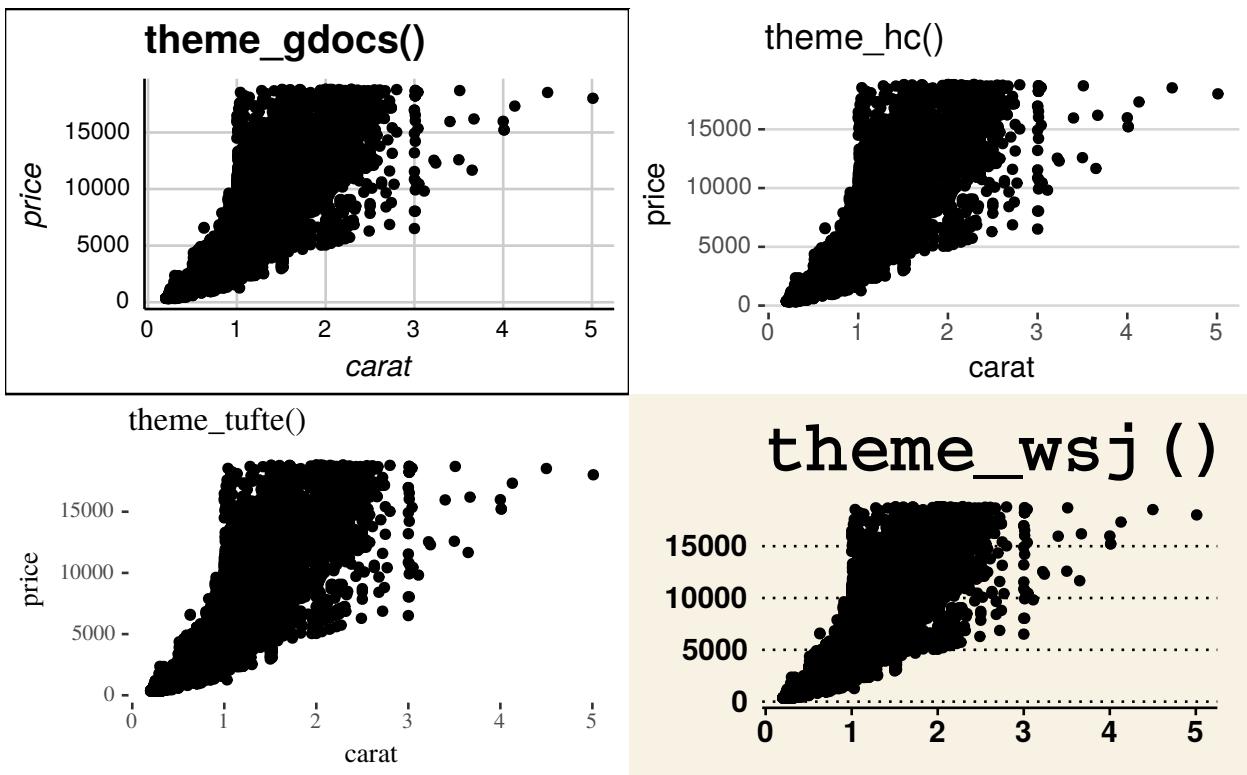
```
labs(title = "theme_void()")
```



### 9.5.2 Temas no pacote ggthemes

O pacote `ggthemes` disponibiliza um conjuntos de temas e escalas de cores. Vamos apresentar alguns temas disponíveis.



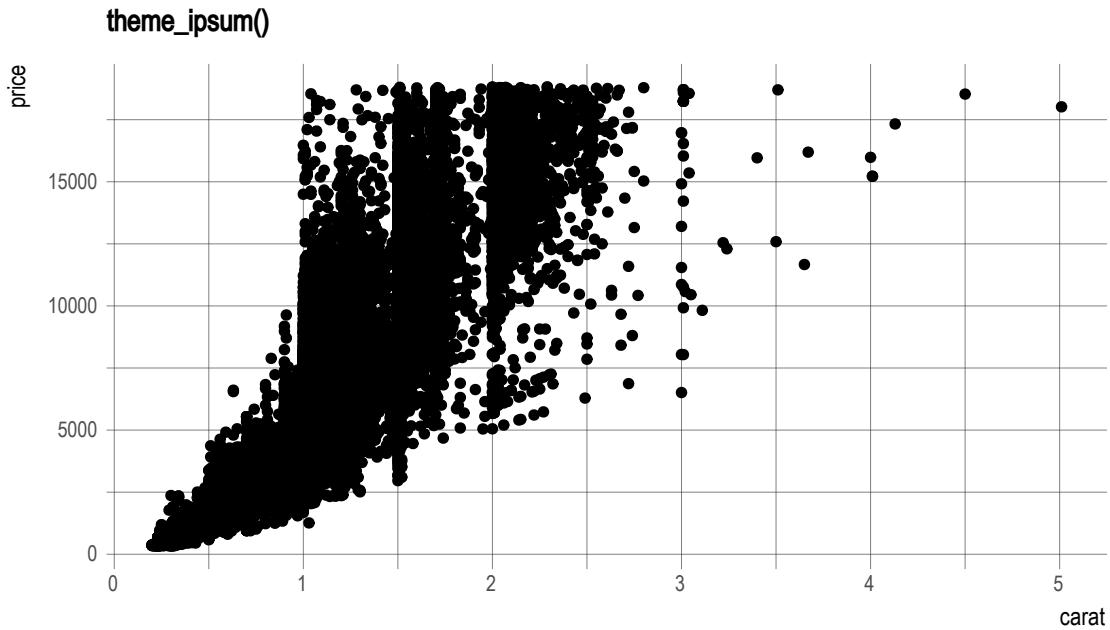


### 9.5.3 hrbrthemes

Alguns pacotes fornecem seus próprios temas. É o caso do `hrbrthemes`. Esse pacote fornece um tema bastante interessante e será usado no resto desse capítulo. Como qualquer outro tema, se for necessário, você pode editar esse tema com a função `theme()`

```
install.packages("hrbrthemes")

library(hrbrthemes)
ggplot(diamonds, aes(x = carat, y = price)) +
 geom_point() +
 labs(title = "theme_ipsum()") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



#### 9.5.4 Setando o tema globalmente

Com o comando abaixo todos os gráficos do seu script terão o mesmo tema:

```
theme_set(theme_ipsum(plot_title_size = 12,
 axis_title_size = 10) +
 theme(text = element_text(angle = 0)))
```

## 9.6 Legendas

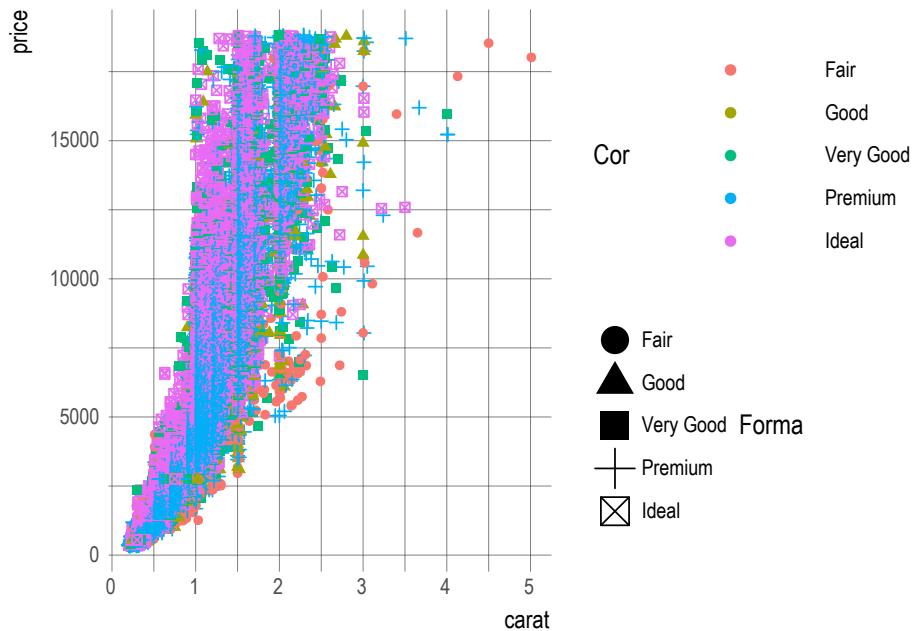
Parte das alterações das legendas pode ser feita via `theme()`. Essas alterações são gerais para todas as legendas. Se o interesse for em mudanças pontuais na legenda de algum elemento estético, serão utilizadas as funções `guides()`, `guide_legend()` e `guide_colorbar()`.

```
guide_legend(title = waiver(), title.position = NULL, title.theme = NULL,
 title.hjust = NULL, title.vjust = NULL, label = TRUE,
 label.position = NULL, label.theme = NULL, label.hjust = NULL,
 label.vjust = NULL, keywidth = NULL, keyheight = NULL,
 direction = NULL, default.unit = "line", override.aes = list(),
 nrow = NULL, ncol = NULL, byrow = FALSE, reverse = FALSE, order = 0, ...)

guide_colourbar(title = waiver(), title.position = NULL, title.theme = NULL,
 title.hjust = NULL, title.vjust = NULL, label = TRUE,
 label.position = NULL, label.theme = NULL, label.hjust = NULL,
 label.vjust = NULL, barwidth = NULL, barheight = NULL, nbin = 20,
 raster = TRUE, ticks = TRUE, draw.ulim = TRUE, draw.llim = TRUE,
 direction = NULL, default.unit = "line", reverse = FALSE, order = 0, ...)
```

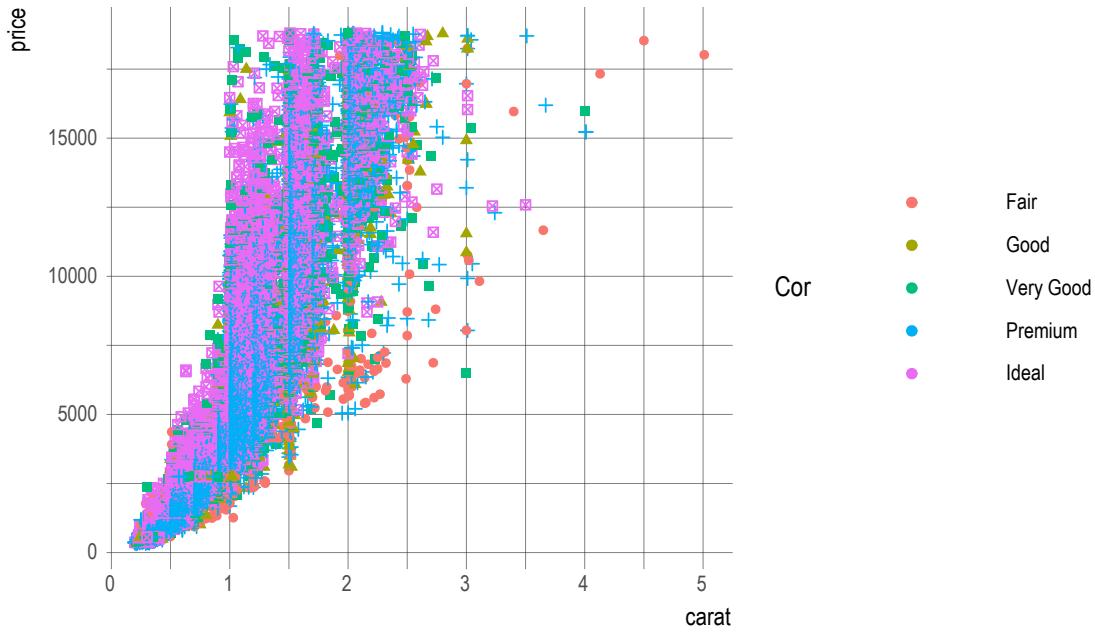
O código abaixo exemplifica o uso do `guide_legend()`.

```
ggplot(diamonds, aes(x = carat, y = price, color = cut, shape = cut)) +
 geom_point() +
 guides(color = guide_legend(title = "Cor", title.position = "left", keywidth = 5),
 shape = guide_legend(title = "Forma", title.position = "right", override.aes = aes(size = 5)))
```



Pode-se fazer uso do “none” para omitir a legenda de um elemento estético:

```
ggplot(diamonds, aes(x = carat, y = price, color = cut, shape = cut)) +
 geom_point() +
 guides(color = guide_legend(title = "Cor", title.position = "left", keywidth = 5),
 shape = "none")
```



## 9.7 Escolhendo o tipo de gráfico

Antes de decidir qual gráfico você irá utilizar, é preciso saber o que se deseja representar. O objetivo guiará qual o tipo de gráfico é mais adequado. A imagem abaixo apresenta uma lista bastante completa de possibilidades de gráficos, dos mais simples aos mais complexos.

Entre neste link para visualizar a imagem com zoom.

Os gráficos mais tradicionais podem ser facilmente criados a partir da lógica de camadas do ggplot2 e dos objetos geométricos disponíveis no pacote. Para gráficos mais complexos, alguns pacotes estão disponíveis. Por exemplo, para criação de treemaps, existe o pacote `treemapify`.

```
library(treemapify)

ggplot(G20, aes(area = gdp_mil_usd, fill = hdi, label = country)) +
 geom_treemap() +
 geom_treemap_text(fontface = "italic", colour = "white", place = "centre",
 grow = TRUE) +
 theme(legend.position = 'bottom')
```

## 9.8 Gráfico de Dispersão (`geom_point()`)

```
geom_point(mapping = NULL, data = NULL, stat = "identity", position = "identity",
..., na.rm = FALSE, show.legend = NA, inherit.aes = TRUE)
```

O gráfico de dispersão é bastante usado para verificar relações entre duas variáveis quantitativas. Para

## THE GRAPHIC CONTINUUM

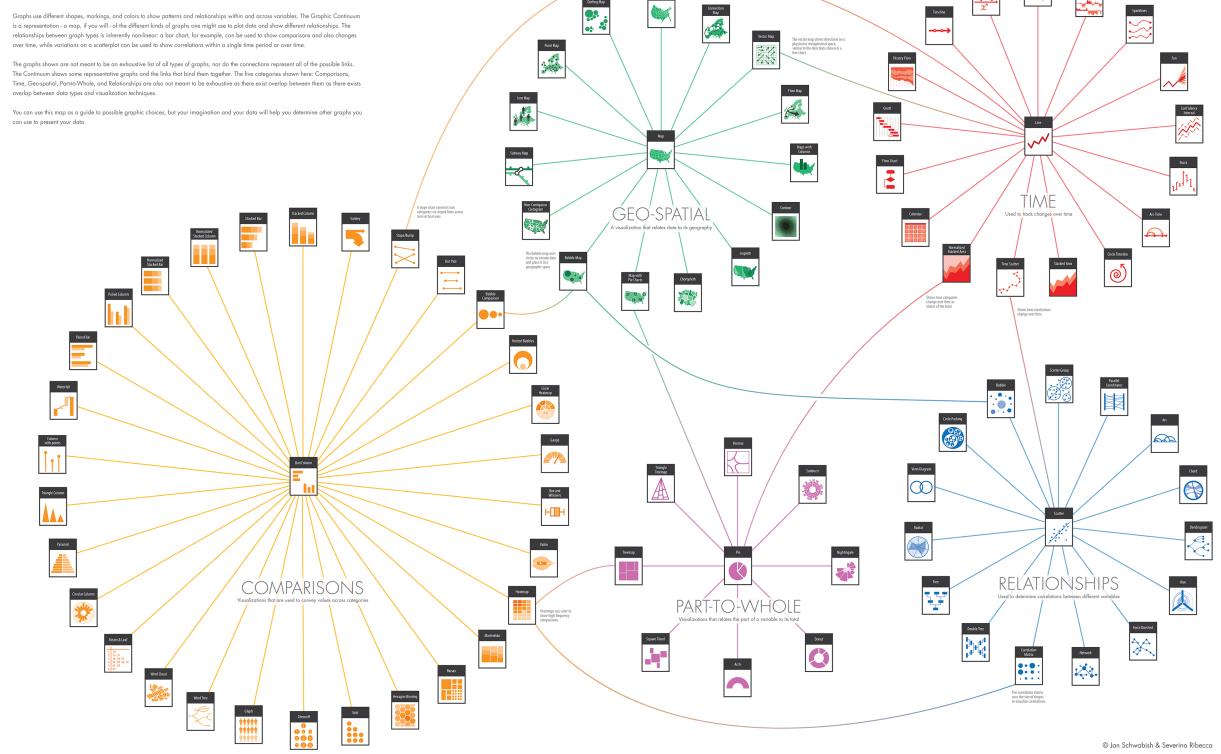


Figura 9.2:

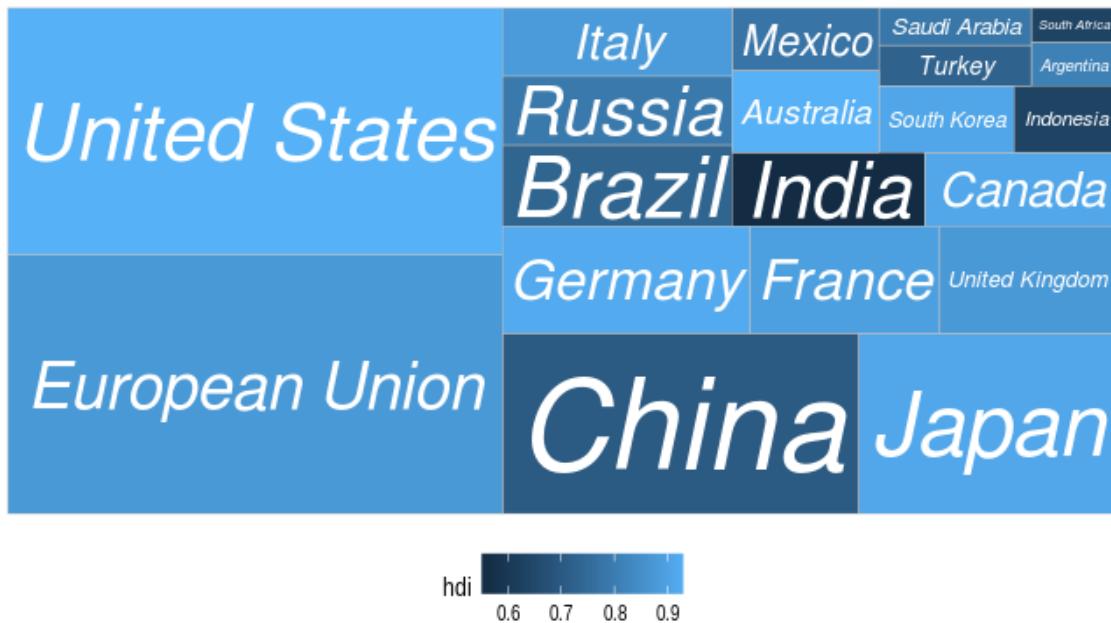


Figura 9.3:

exemplificar, vamos utilizar a base disponível no pacote `gapminder`. Nessa base, existem uma variável de expectativa de vida e outra de renda per capita.

Como queremos um gráfico de pontos, o objeto geométrico natural é o `geom_point()`. Esse objeto geométrico tem os seguintes elementos estéticos:

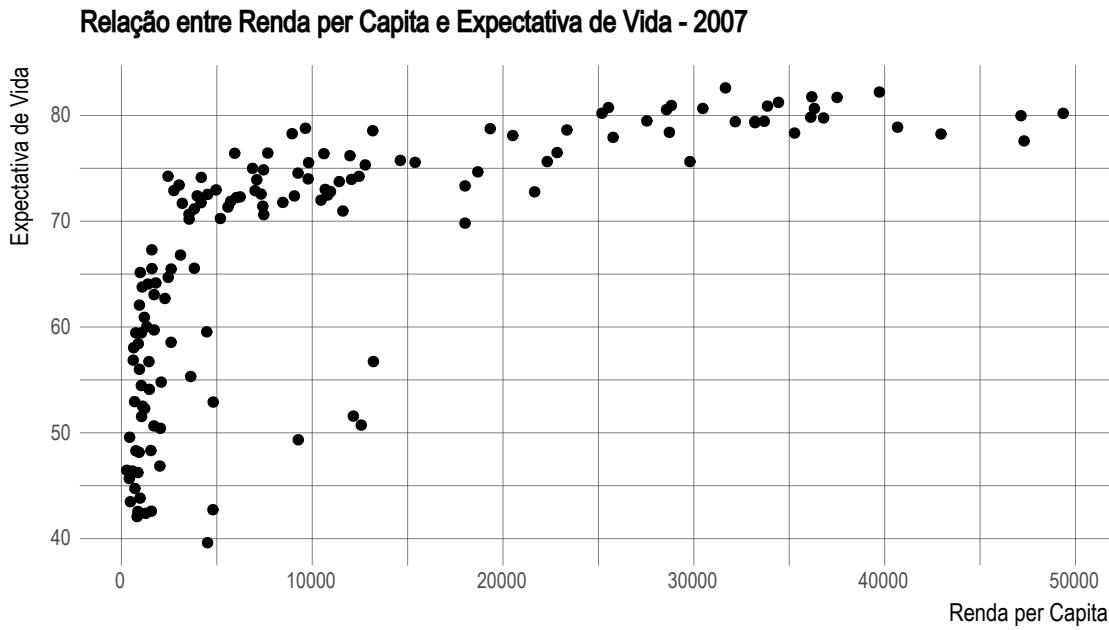
Os parâmetros estéticos (aes) são:

- `x`
- `y`
- `alpha`
- `colour`
- `fill`
- `shape`
- `size`
- `stroke`

Vamos verificar qual é a relação entre essas duas variáveis:

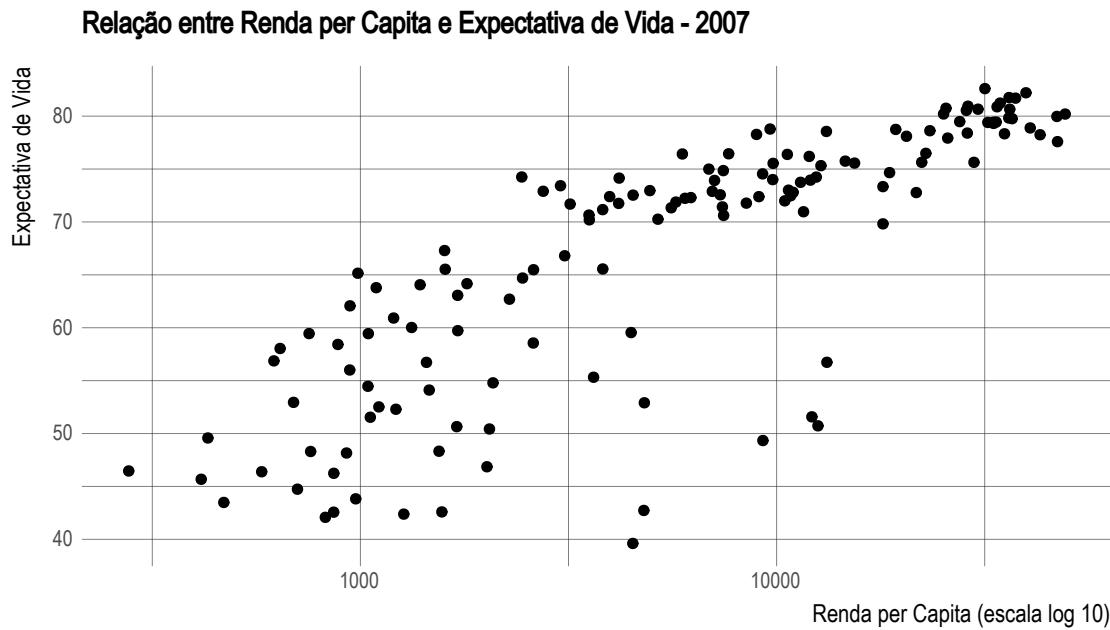
```
library(hrbrthemes)
library(gapminder)
gapminder %>%
 filter(year == max(year)) %>%
 ggplot(aes(x = gdpPercap, y = lifeExp)) +
 geom_point() +
 labs(title = "Relação entre Renda per Capita e Expectativa de Vida - 2007",
 x = "Renda per Capita",
 y = "Expectativa de Vida") +
 theme_ipsum(plot_title_size = 12,
```

```
axis_title_size = 10
```



Note que a expectativa de vida cresce muito rápido para níveis de renda baixos, mas o incremento decresce conforme o nível de renda aumenta. Esse fato pode ser melhor representado utilizando uma escala logarítmica para a variável renda per capita. É assim que usualmente essa relação é apresentada. Para isso, poderíamos aplicar a função `log10()` na variável de renda per capita, ou utilizar a função `scale_x_log10()`:

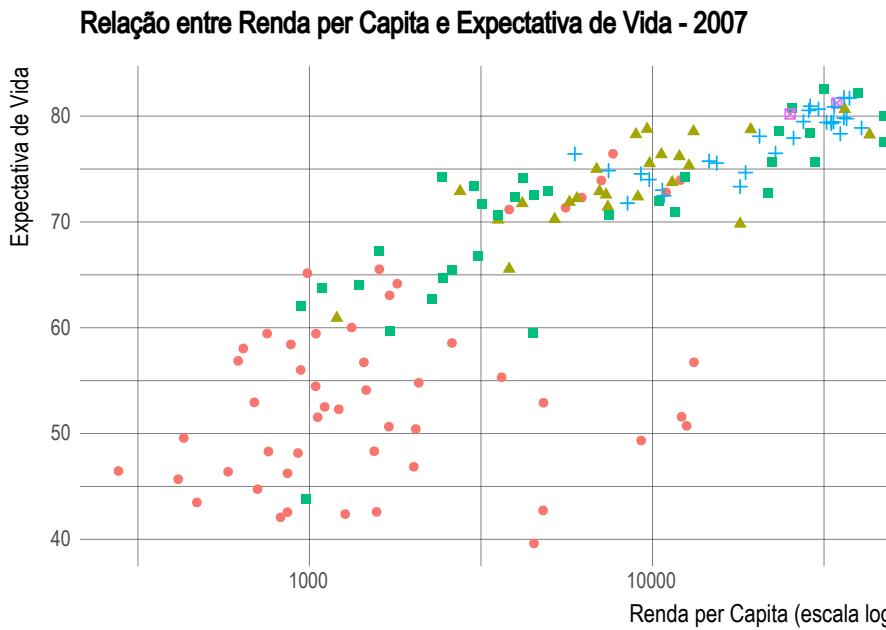
```
gapminder %>%
 filter(year == max(year)) %>%
 ggplot(aes(x = gdpPercap, y = lifeExp)) +
 geom_point() +
 scale_x_log10() +
 labs(title = "Relação entre Renda per Capita e Expectativa de Vida - 2007",
 x = "Renda per Capita (escala log 10)",
 y = "Expectativa de Vida") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



Com essa escala, valores incrementados na ordem de 10 vezes serão igualmente espaçados. Nesse caso, a relação parece ser mais linear. Ou seja, ao aumentar a renda 10 vezes, espera-se que a expectativa de vida cresça a uma taxa constante.

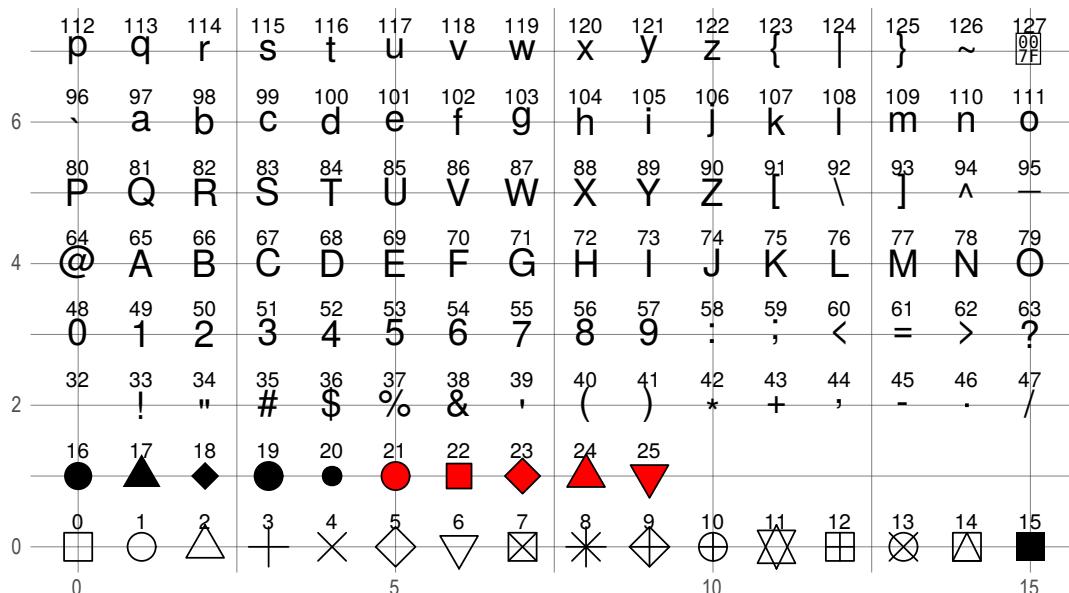
Vamos mapear a variável `continent` ao elemento estético color e shape:

```
gapminder %>%
 filter(year == max(year)) %>%
 ggplot(aes(x = gdpPercap, y = lifeExp,
 color = continent, shape = continent)) +
 geom_point() +
 scale_x_log10() +
 scale_color_discrete("Continente") +
 scale_shape_discrete("Continente") +
 labs(title = "Relação entre Renda per Capita e Expectativa de Vida - 2007",
 x = "Renda per Capita (escala log 10)",
 y = "Expectativa de Vida") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



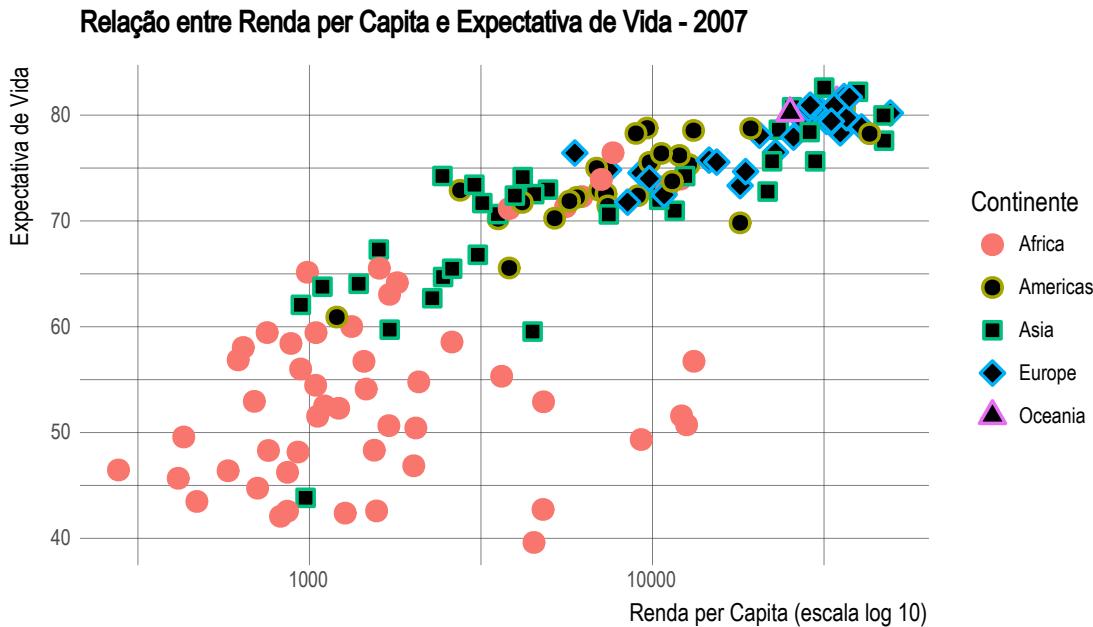
Automaticamente o ggplot2 criou uma escala para as cores e formatos dos pontos. O usuário pode alterar esse mapeamento utilizando as funções `scale_*_*`.

Por fim, fica aqui a lista com os tipos de shapes:



Perceba que os formatos de 21 a 24 possuem preenchimento (`fill`). Assim, no código abaixo iremos definir o preenchimento, o tamanho do ponto e a espessura para aqueles formatos que possuem contornos.

```
gapminder %>%
 filter(year == max(year)) %>%
 ggplot(aes(x = gdpPercap, y = lifeExp,
 color = continent, shape = continent)) +
 geom_point(fill = "black", size = 3, stroke = 1) +
 scale_x_log10() +
 scale_color_discrete("Continente") +
 scale_shape_manual("Continente", values = c(19, 21, 22, 23, 24)) +
 labs(title = "Relação entre Renda per Capita e Expectativa de Vida - 2007",
 x = "Renda per Capita (escala log 10)",
 y = "Expectativa de Vida")
```

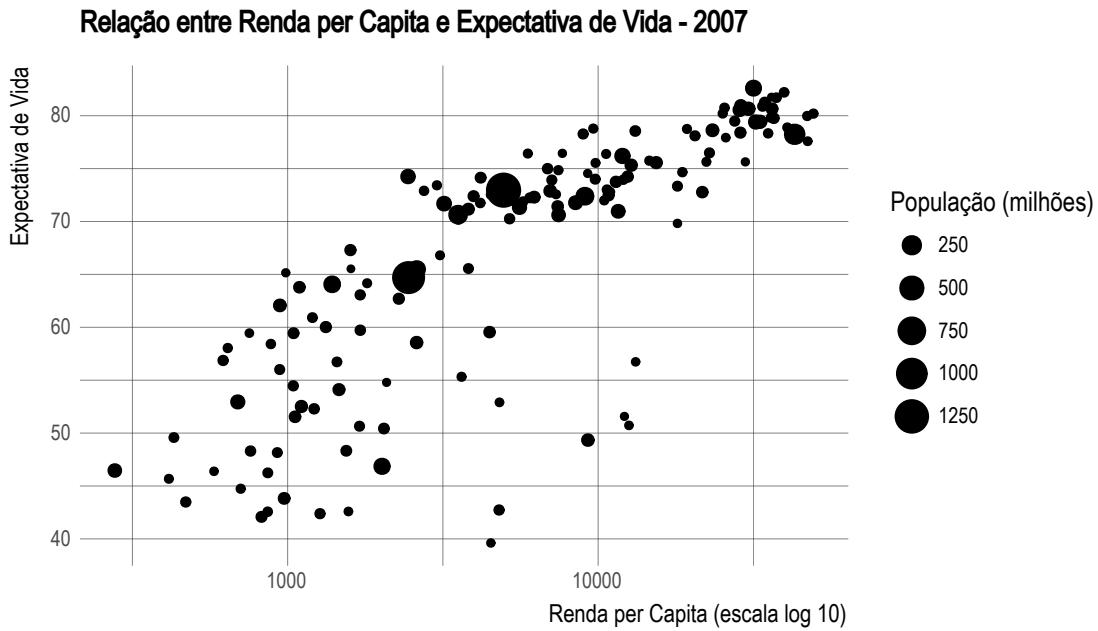


## 9.9 Gráficos de Bolhas

O gráfico de bolha é uma extensão natural do gráfico de pontos. Nele é possível observar possíveis relações entre as três variáveis. Para esse tipo de gráfico, são necessárias três variáveis. Duas para indicarem as posições x e y, e uma terceira para definir o tamanho do ponto (`size`). Vamos utilizar a variável `pop` (população):

```
gapminder %>%
 filter(year == max(year)) %>%
 ggplot(aes(x = gdpPercap, y = lifeExp,
 size = pop)) +
 geom_point()
```

```
scale_size_continuous("População (milhões)", labels = function(x) round(x/1e6)) +
 scale_x_log10() +
 labs(title = "Relação entre Renda per Capita e Expectativa de Vida - 2007",
 x = "Renda per Capita (escala log 10)",
 y = "Expectativa de Vida") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



## 9.10 Gráficos de Barras

Os gráficos de barras/colunas são geralmente utilizados para comparações entre categorias (variáveis qualitativas). No ggplot2, podemos usar dois objetos geométricos distintos:

```
geom_bar(mapping = NULL, data = NULL, stat = "count", position = "stack", ...,
 width = NULL, binwidth = NULL, na.rm = FALSE, show.legend = NA,
 inherit.aes = TRUE)

geom_col(mapping = NULL, data = NULL, position = "stack", ...,
 width = NULL, na.rm = FALSE, show.legend = NA, inherit.aes = TRUE)
```

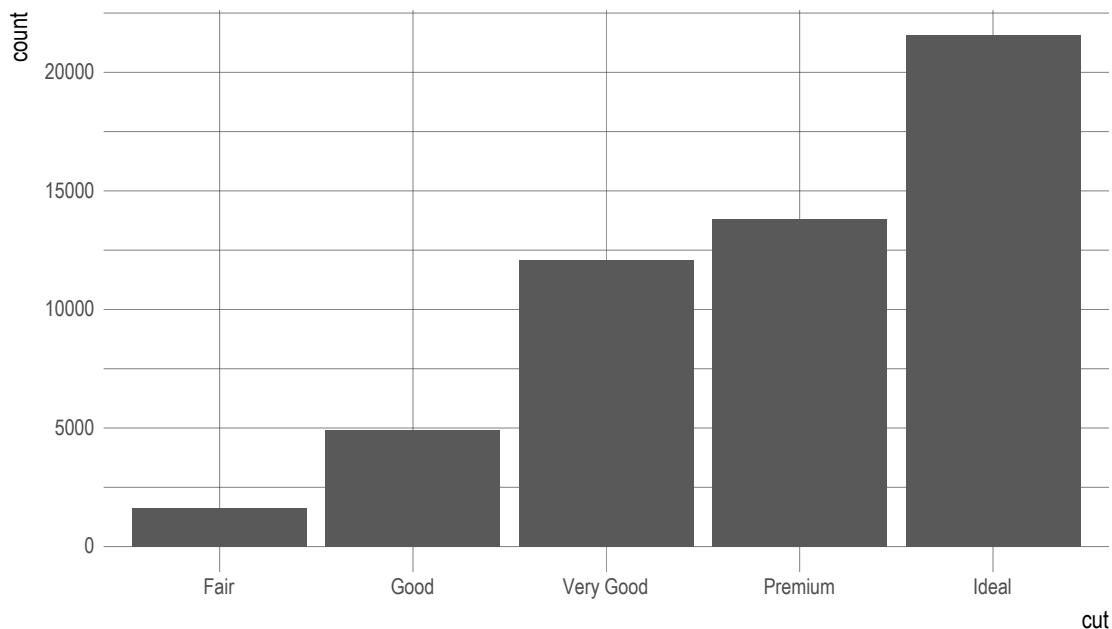
Os parâmetros estéticos (aes) são:

- x
- y (somente com stat=identity)
- alpha
- colour
- fill

- `linetype`
- `size`

Há um detalhe importante para o `geom_bar()`, o argumento `stat`. Por padrão, para esse objeto geométrico, o valor de `stat` é `count`, o que significa que ele irá fazer uma contagem dos elementos dos eixo x. Essa contagem será usada no eixo y. Por exemplo:

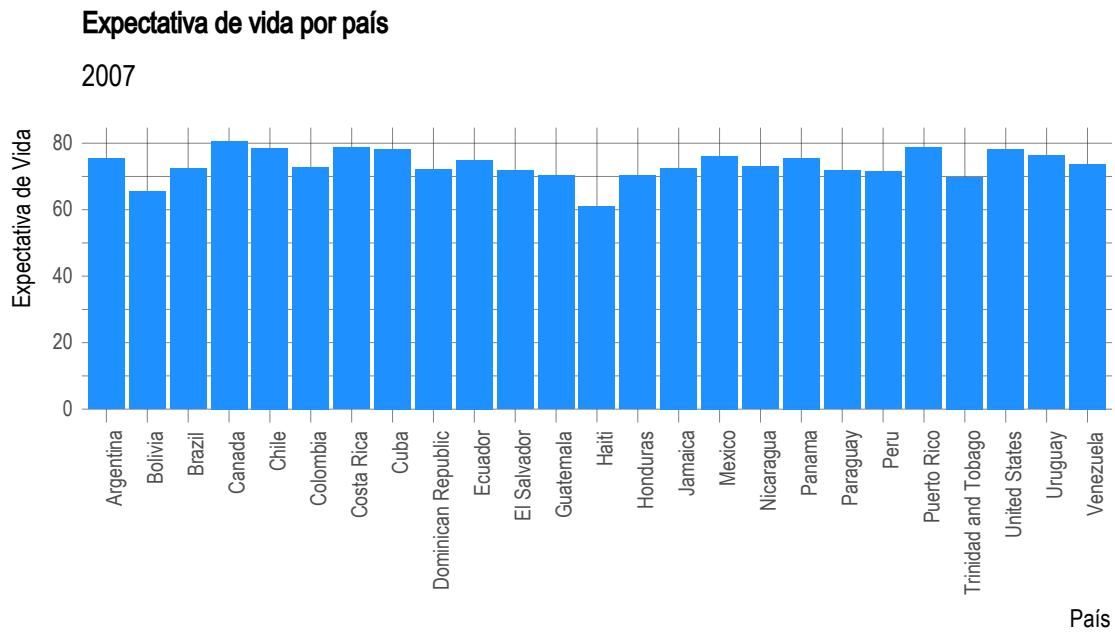
```
ggplot(diamonds, aes(x = cut)) +
 geom_bar() +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



Para cada valor da variável `cut`, o `ggplot2` calculou o número de observações no data.frame `diamonds`.

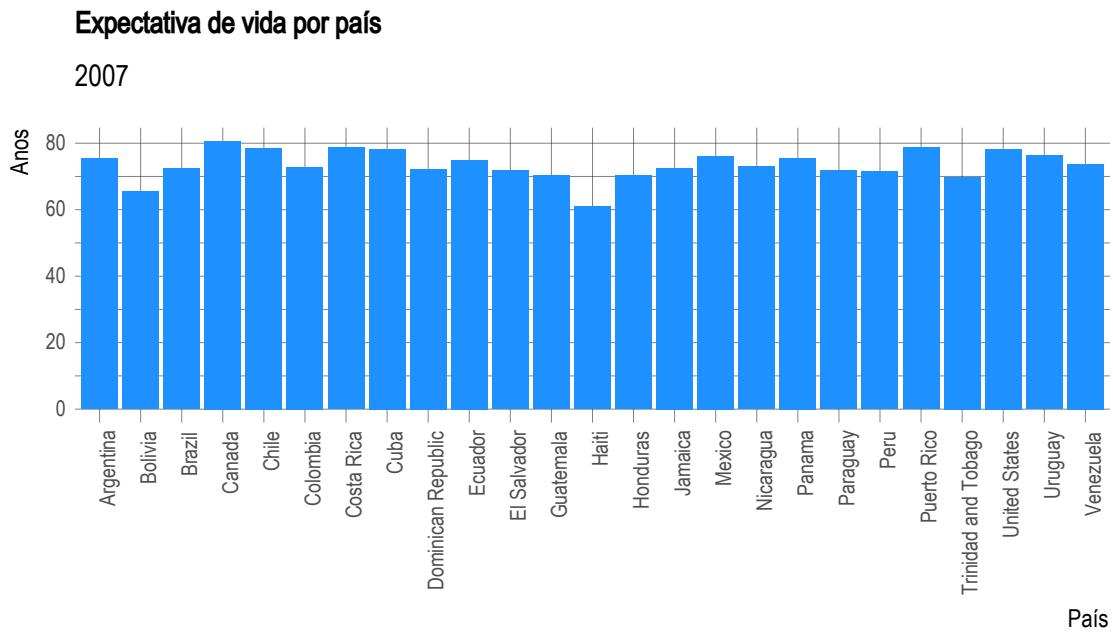
Para que o y seja mapeado para uma variável do data.frame, é necessário definir `stat = identity`.

```
gapminder %>%
 filter(year == max(year),
 continent == "Americas") %>%
 ggplot(aes(x = country, y = lifeExp)) +
 geom_bar(stat = "identity", fill = "dodgerblue") +
 labs(title = "Expectativa de vida por país",
 subtitle = "2007",
 x = "País",
 y = "Expectativa de Vida") +
 theme(axis.text.x = element_text(angle = 90, hjust = 1))
```



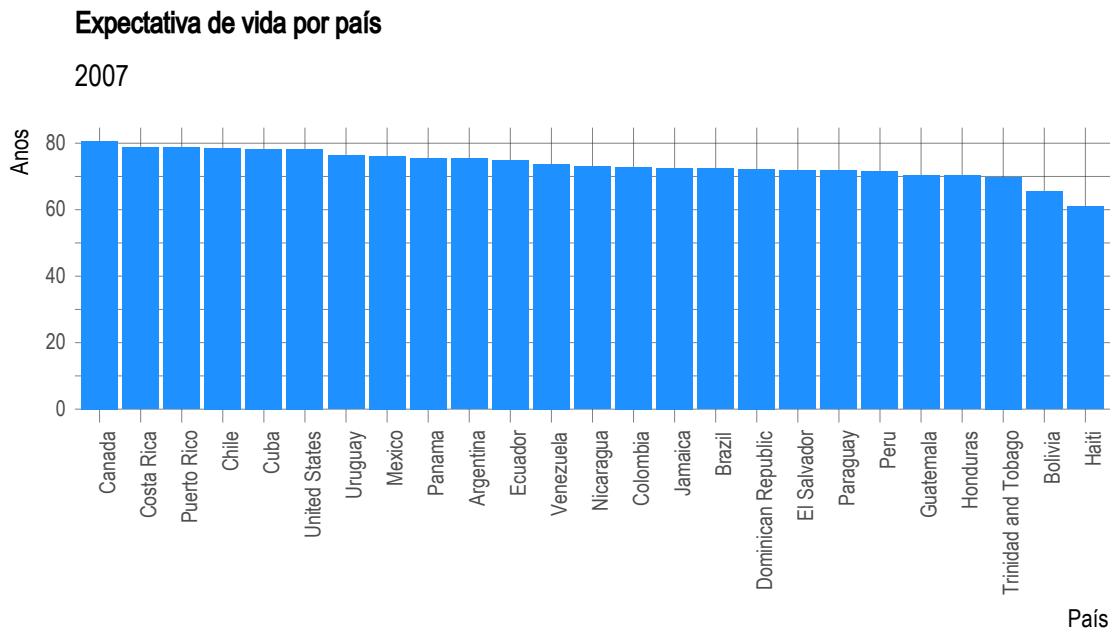
Usando o `geom_col()`:

```
gapminder %>%
 filter(year == max(year),
 continent == "Americas") %>%
 ggplot(aes(x = country, y = lifeExp)) +
 geom_col(fill = "dodgerblue") +
 labs(title = "Expectativa de vida por país",
 subtitle = "2007",
 x = "País",
 y = "Anos") +
 theme(axis.text.x = element_text(angle = 90, hjust = 1))
```



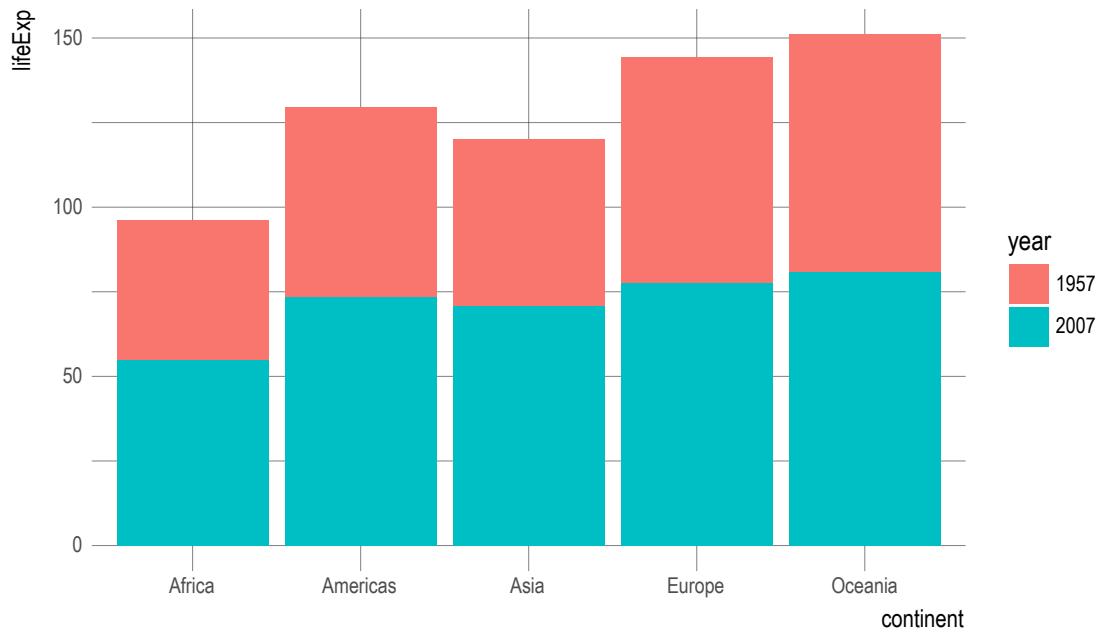
Uma pergunta recorrente é: Como ordenar as barras em ordem crescente/decrescente? Para isso, pode-se usar a função `reorder()` no momento do mapeamento. Fica mais claro com um exemplo:

```
gapminder %>%
 filter(year == max(year),
 continent == "Americas") %>%
 ggplot(aes(x = reorder(country, -lifeExp), y = lifeExp)) +
 geom_col(fill = "dodgerblue") +
 labs(title = "Expectativa de vida por país",
 subtitle = "2007",
 x = "País",
 y = "Anos") +
 theme(axis.text.x = element_text(angle = 90, hjust = 1))
```



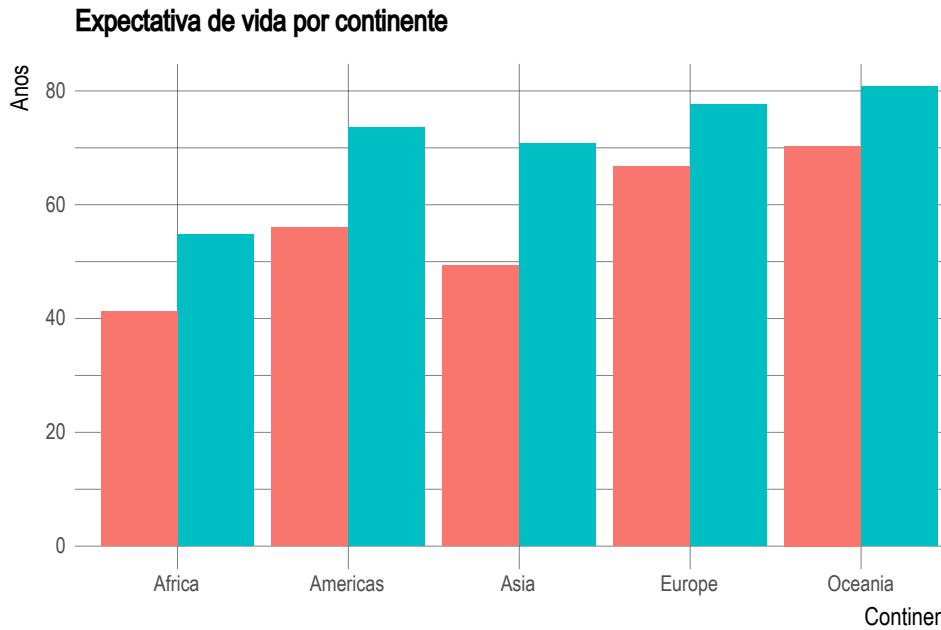
Vamos agora criar um gráfico em que se compara a expectativa de vida média por continente em 1957 e 2007:

```
gapminder %>%
 filter(year %in% c(1957, 2007)) %>%
 # Converte o ano para factor - será categoria no gráfico
 mutate(year = factor(year)) %>%
 group_by(continent, year) %>%
 summarise(lifeExp = mean(lifeExp)) %>%
 ggplot(aes(x = continent, y = lifeExp, fill = year)) +
 geom_col() +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



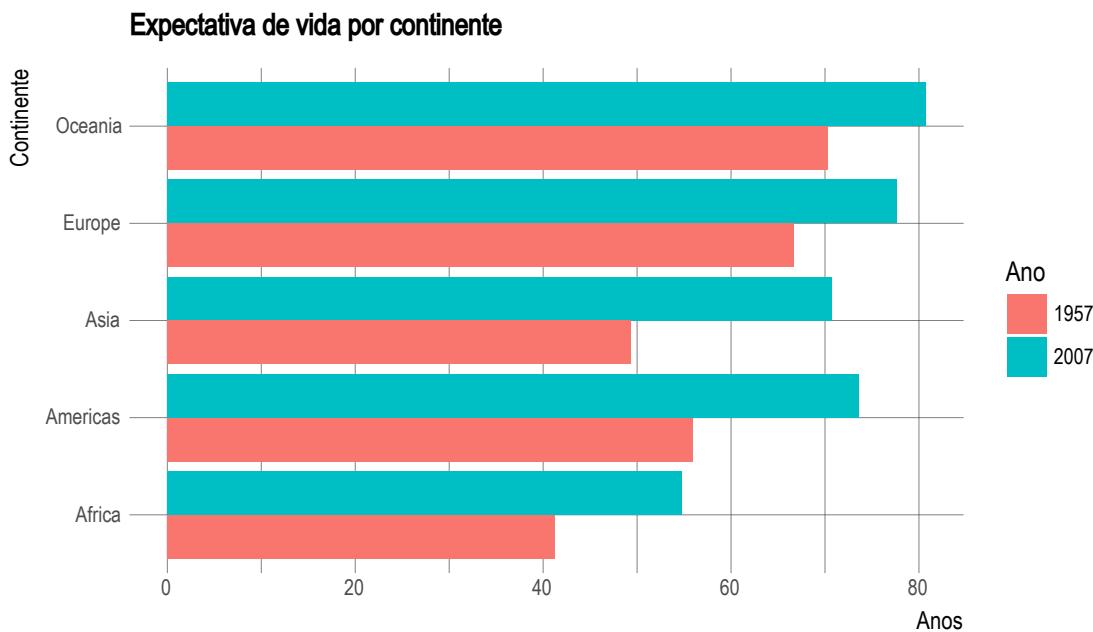
Para continente, o gráfico empilhou as barras. Isto se deve ao argumento `position = stack`. Para colocar as barras lado a lado, utilizamos o valor “dodge”:

```
gapminder %>%
 filter(year %in% c(1957, 2007)) %>%
 # Converte o ano para factor - será categoria no gráfico
 mutate(year = factor(year)) %>%
 group_by(continent, year) %>%
 summarise(lifeExp = mean(lifeExp)) %>%
 ggplot(aes(x = continent, y = lifeExp, fill = year)) +
 geom_col(position = "dodge") +
 labs(title = "Expectativa de vida por continente",
 x = "Continente",
 y = "Anos",
 fill = "Ano") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



Também é comum representar as barras horizontalmente. Basta usar a função `coord_flip()`:

```
gapminder %>%
 filter(year %in% c(1957, 2007)) %>%
 # Converte o ano para factor - será categoria no gráfico
 mutate(year = factor(year)) %>%
 group_by(continent, year) %>%
 summarise(lifeExp = mean(lifeExp)) %>%
 ggplot(aes(x = continent, y = lifeExp, fill = year)) +
 geom_col(position = "dodge") +
 coord_flip() +
 labs(title = "Expectativa de vida por continente",
 x = "Continente",
 y = "Anos",
 fill = "Ano") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



## 9.11 Gráficos de linhas

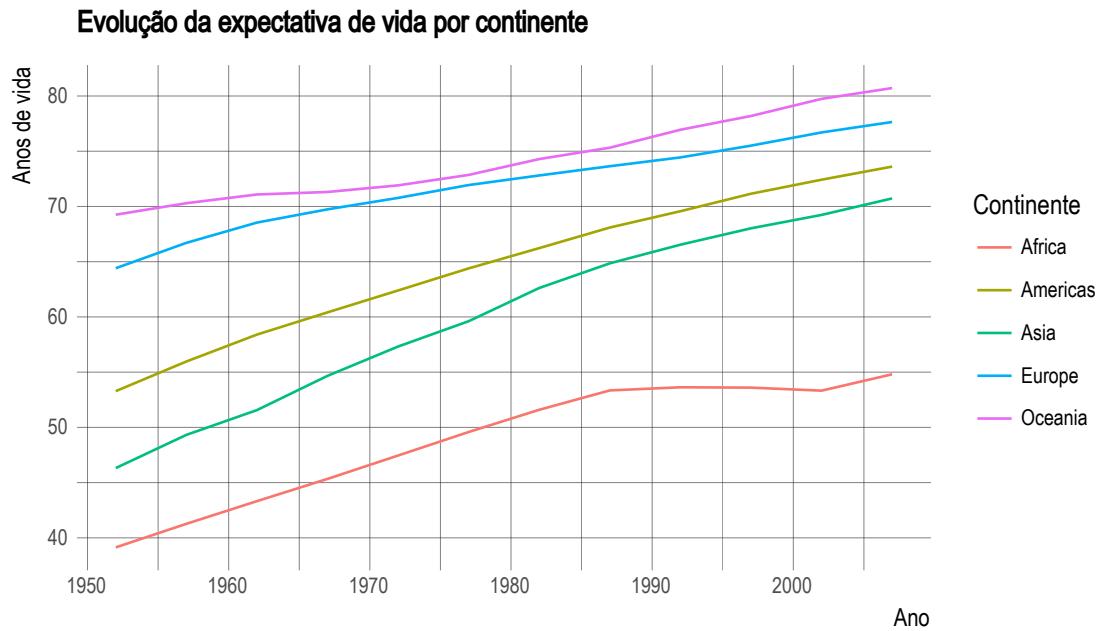
Os gráficos de linhas são geralmente utilizados para apresentar a evolução de uma variável quantitativa em um intervalo de tempo.

```
geom_line(mapping = NULL, data = NULL, stat = "identity", position = "identity",
na.rm = FALSE, show.legend = NA, inherit.aes = TRUE, ...)
```

Os parâmetros estéticos (aes) são:

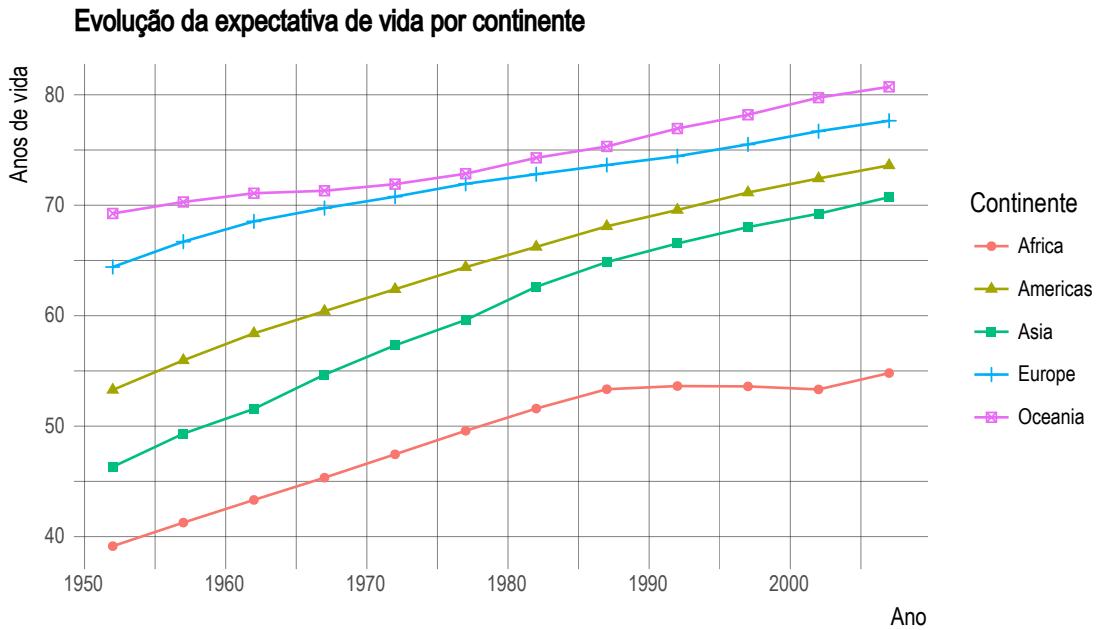
- x
- y
- alpha
- colour
- linetype
- size

```
gapminder %>%
 group_by(continent, year) %>%
 summarise(lifeExp = mean(lifeExp)) %>%
 ggplot(aes(x = year, y = lifeExp, color = continent)) +
 geom_line() +
 labs(title = "Evolução da expectativa de vida por continente",
 x = "Ano",
 y = "Anos de vida",
 color = "Continente") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



É bastante comum que gráficos de linhas apresentem marcações para os períodos em que realmente existem os dados. Para isso, podemos adicionar uma camada de pontos:

```
gapminder %>%
 group_by(continent, year) %>%
 summarise(lifeExp = mean(lifeExp)) %>%
 ggplot(aes(x = year, y = lifeExp, color = continent)) +
 geom_line() +
 geom_point(aes(shape = continent)) +
 labs(title = "Evolução da expectativa de vida por continente",
 x = "Ano",
 y = "Anos de vida",
 color = "Continente",
 shape = "Continente") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



## 9.12 Histogramas e freqpoly

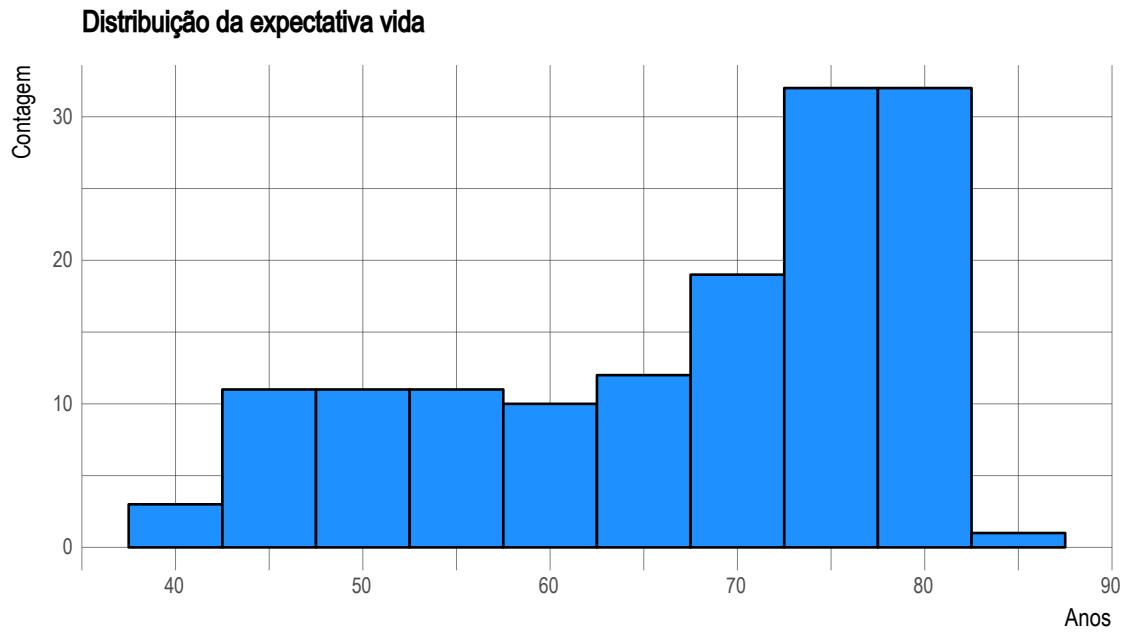
```
geom_freqpoly(mapping = NULL, data = NULL, stat = "bin",
 position = "identity", ..., na.rm = FALSE, show.legend = NA,
 inherit.aes = TRUE)

geom_histogram(mapping = NULL, data = NULL, stat = "bin",
 position = "stack", ..., binwidth = NULL, bins = NULL, na.rm = FALSE,
 show.legend = NA, inherit.aes = TRUE)
```

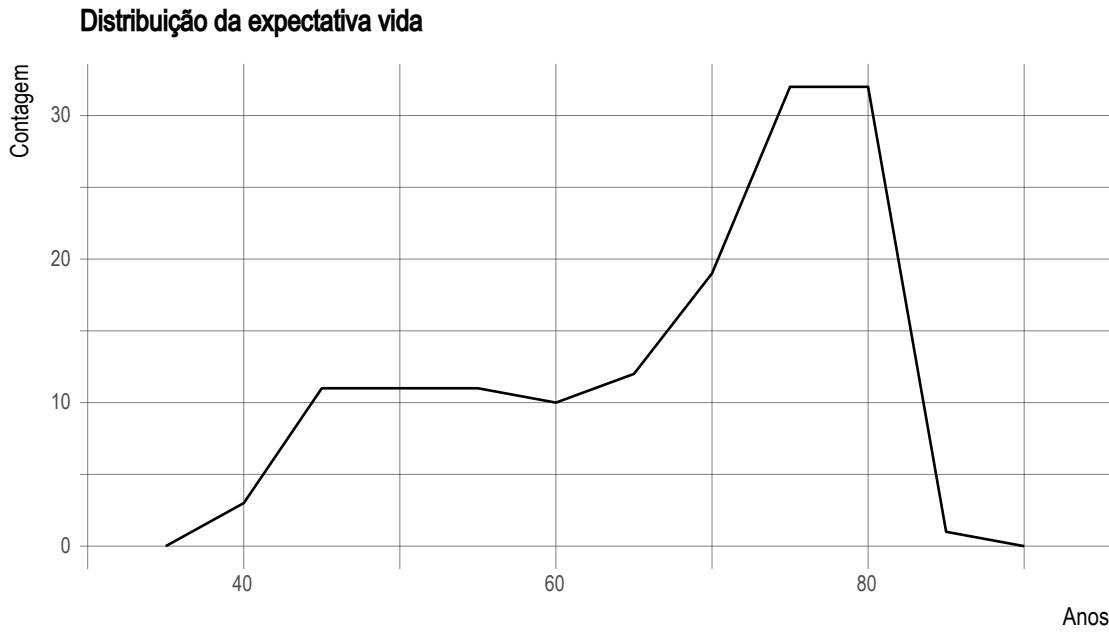
Os histogramas são utilizados para representar a distribuição de dados de uma variável quantitativa em intervalos contínuos. Esses intervalos são chamados de `bins`. Para cada bin, será apresentado a quantidade de valores que estão naquele intervalo. A diferença para o `geom_freqpoly` é que este utiliza linhas para construir polígonos, enquanto o `geom_histogram` utiliza barras.

Conforme a documentação do `ggplot2`, o `geom_histogram()` utiliza os mesmos elementos estéticos do `geom_bar()`. Já o `geom_freqpoly()` utiliza os mesmos do `geom_line()`.

```
gapminder %>%
 filter(year == 2007) %>%
 ggplot(aes(x = lifeExp)) +
 geom_histogram(binwidth = 5, fill = 'dodgerblue', color = 'black') +
 labs(title = "Distribuição da expectativa vida",
 x = "Anos",
 y = "Contagem") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```

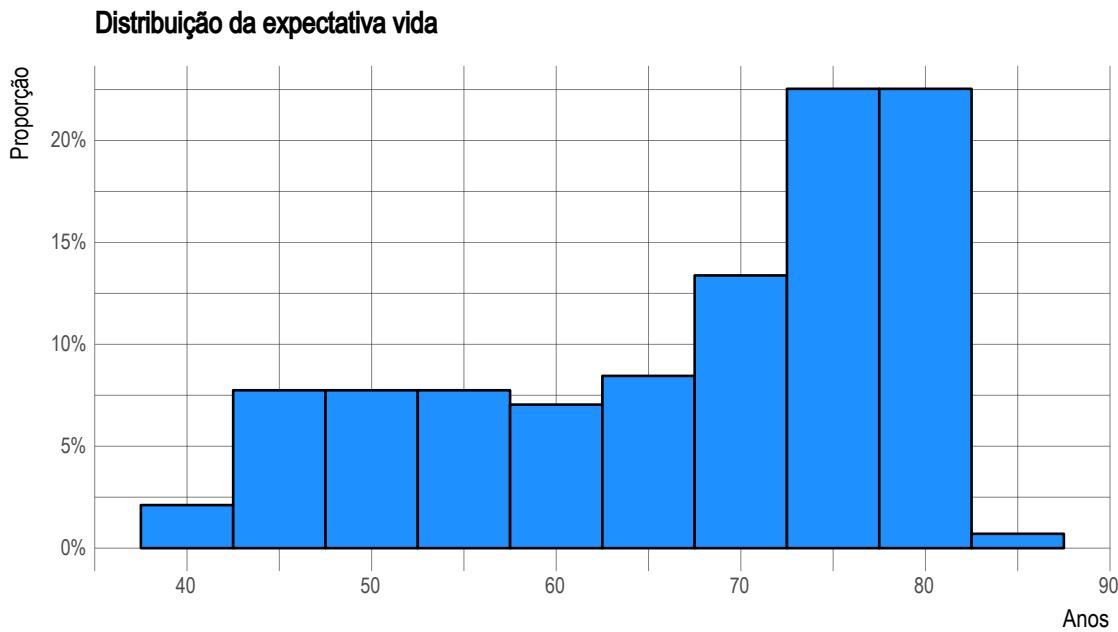


```
gapminder %>%
 filter(year == 2007) %>%
 ggplot(aes(x = lifeExp)) +
 geom_freqpoly(binwidth = 5) +
 labs(title = "Distribuição da expectativa vida",
 x = "Anos",
 y = "Contagem") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



Transformando em proporção:

```
gapminder %>%
 filter(year == 2007) %>%
 ggplot(aes(x = lifeExp)) +
 geom_histogram(aes(y = ..count../sum(..count..))), binwidth = 5, fill = 'dodgerblue', color = 'black') +
 labs(title = "Distribuição da expectativa vida",
 x = "Anos",
 y = "Proporção") +
 scale_y_continuous(labels = scales::percent_format()) +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



O `ggplot2` internamente criou a variável `..count...`. Dessa forma, podemos utilizá-la para criar as proporções.

## 9.13 Boxplots, jitterplots e violinplots

O boxplot é uma representação comum para apresentar a distribuição de uma variável a partir de seus quantis. A imagem abaixo detalha como um boxplot é formado.

O boxplot também pode ser usado para verificar a distribuição de variável para um conjunto de valores de um segunda variável. Por exemplo, qual é a distribuição da expectativa de vida por ano?

```
ggplot(gapminder, aes(x = factor(year), y = lifeExp)) +
 geom_boxplot(fill = "dodgerblue") +
 labs(y = "Anos de vida",
 x = "Ano",
 title = "Distribuição da expectativa de vida por ano") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```

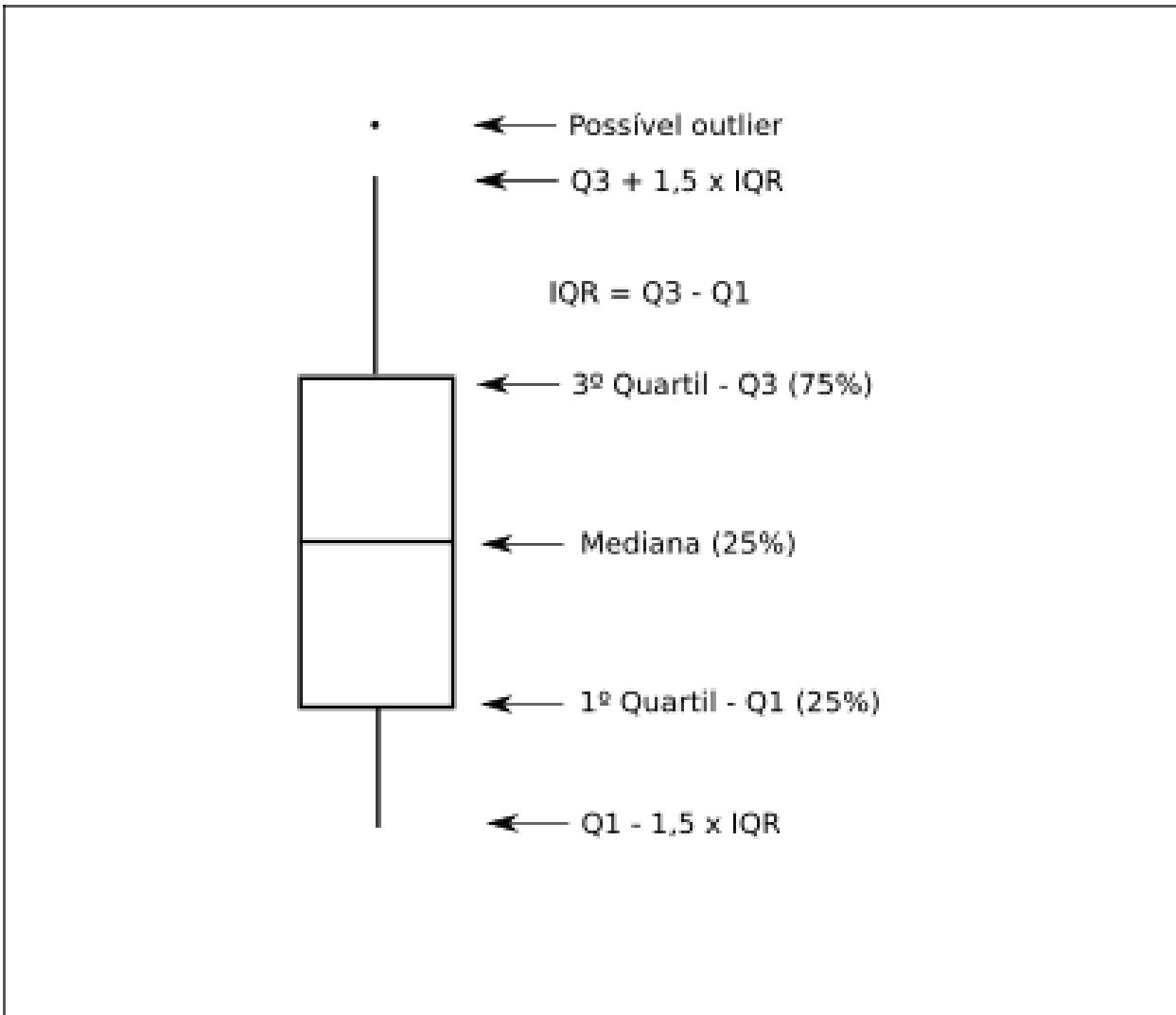
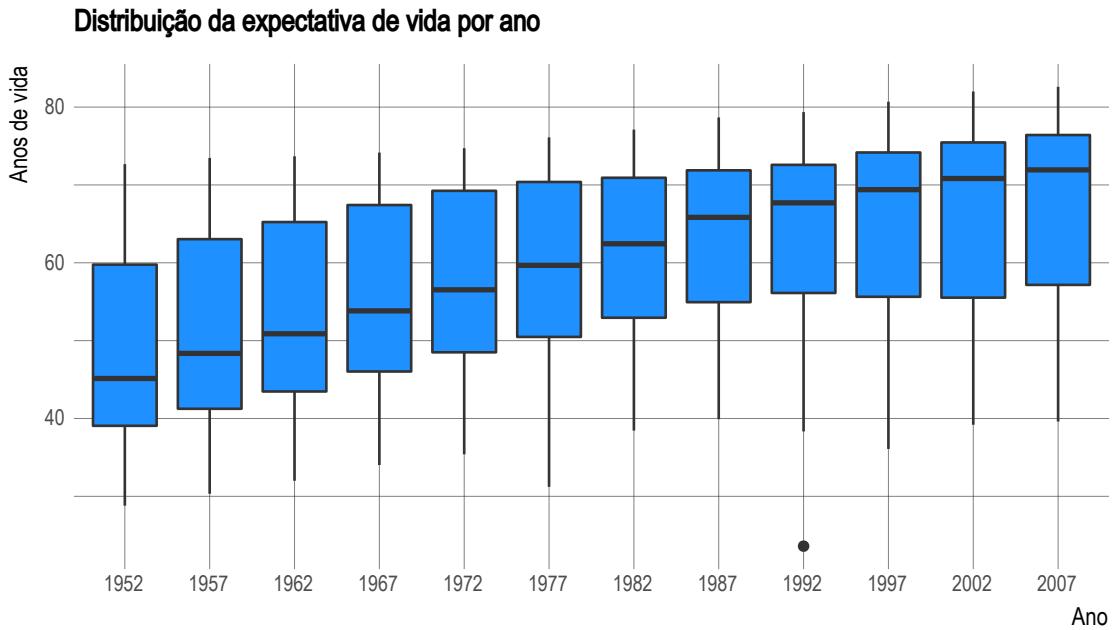


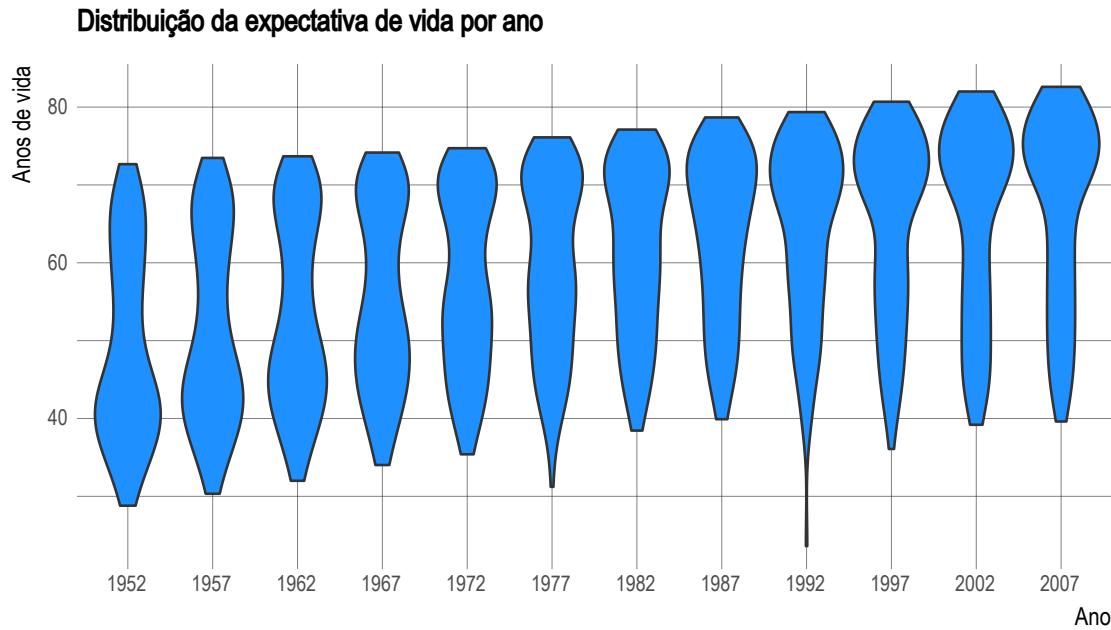
Figura 9.4: Detalhes sobre o boxplot



Vemos que existe um possível outlier em 1992. Quando falarmos sobre anotações, voltaremos a esse gráfico.

Para termos uma visão da distribuição geral dos valores por ano, podemos usar o `geom_violin()`. O violinplot se baseia na densidade de probabilidade de uma variável contínua. Assim, é possível verificar em quais intervalos existe uma maior chance de ocorrência. Isto é representado pela parte mais larga do objeto.

```
ggplot(gapminder, aes(x = factor(year), y = lifeExp)) +
 geom_violin(fill = "dodgerblue") +
 labs(y = "Anos de vida",
 x = "Ano",
 title = "Distribuição da expectativa de vida por ano") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



O jitterplot é utilizado para evitar o problema do overplotting em um gráfico. Note no gráfico abaixo que não sabemos se a marcação de um ponto representa uma única observação ou várias.

```
ggplot(gapminder, aes(x = factor(year), y = lifeExp)) +
 geom_point() +
 labs(y = "Anos de vida",
 x = "Ano",
 title = "Distribuição da expectativa de vida por ano") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



Para observarmos a real distribuição, é necessário adicionar um pouco de ruído a fim de que os pontos se afastem um pouco:

```
ggplot(gapminder, aes(x = factor(year), y = lifeExp)) +
 geom_jitter() +
 labs(y = "Anos de vida",
 x = "Ano",
 title = "Distribuição da expectativa de vida por ano") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



## 9.14 Anotações

Para criar anotações no ggplot2, pode-se utilizar a função `annotate()`.

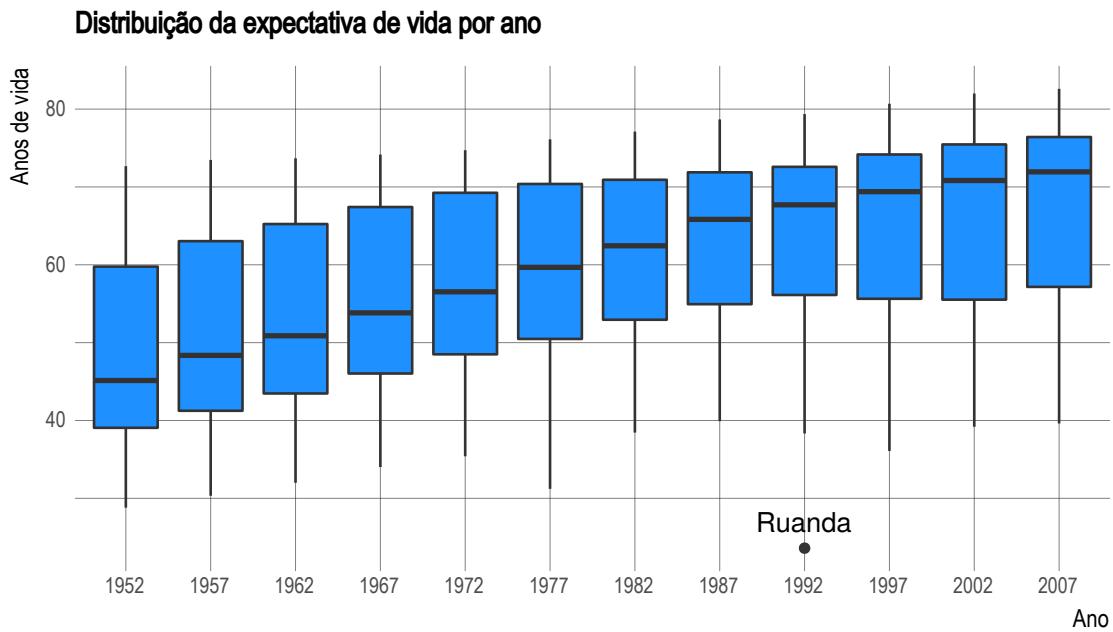
Primeiro, vamos manipular os dados para saber qual é aquele ponto:

```
gapminder %>%
 filter(year == 1992, lifeExp == min(lifeExp))

A tibble: 1 x 6
country continent year lifeExp pop gdpPercap
<fctr> <fctr> <int> <dbl> <int> <dbl>
1 Rwanda Africa 1992 23.599 7290203 737.0686
```

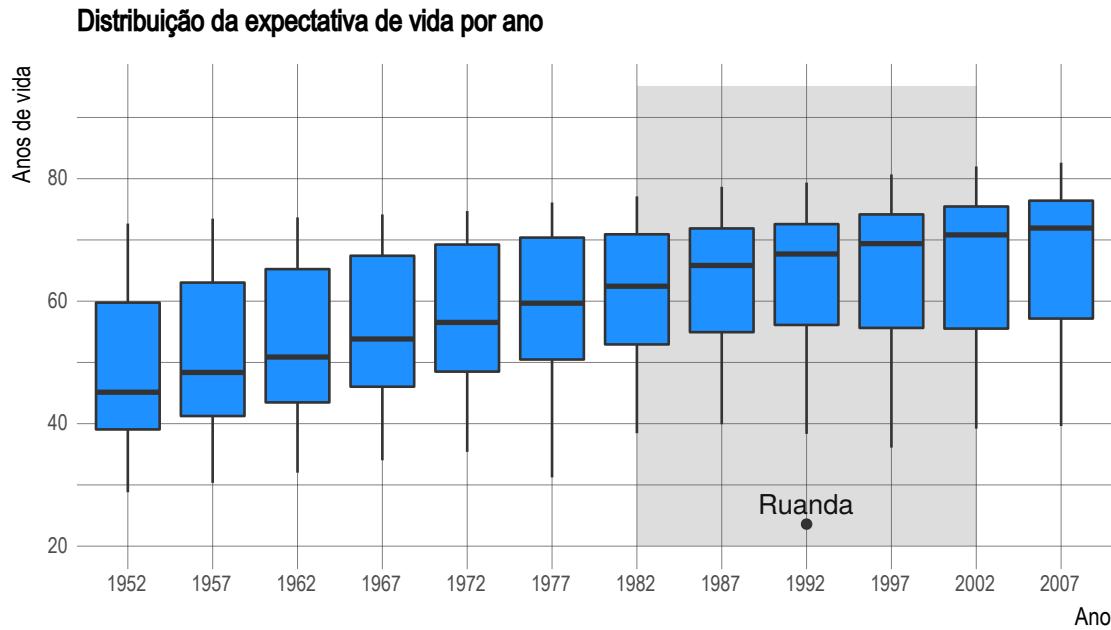
Com essas informações, podemos adicionar uma anotação ao gráfico:

```
ggplot(gapminder, aes(x = factor(year), y = lifeExp)) +
 geom_boxplot(fill = "dodgerblue") +
 annotate("text", x = "1992", y = 27, label = "Ruanda") +
 labs(y = "Anos de vida",
 x = "Ano",
 title = "Distribuição da expectativa de vida por ano") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



Também podemos adicionar segmentos e retângulos com o `annotate()`. Vamos marcar o período de 1982 a 2002 com um retângulo:

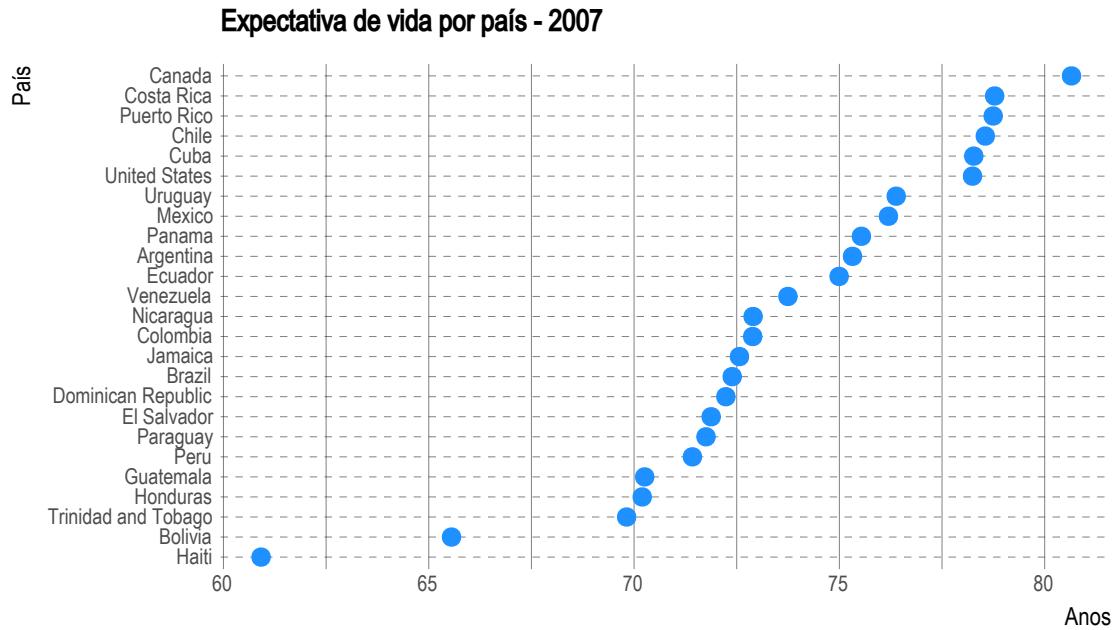
```
ggplot(gapminder, aes(x = factor(year), y = lifeExp)) +
 annotate("text", x = "1992", y = 27, label = "Ruanda") +
 annotate("rect", xmin = "1982", ymin = 20,
 xmax = "2002", ymax = 95, alpha = 0.2) +
 geom_boxplot(fill = "dodgerblue") +
 labs(y = "Anos de vida",
 x = "Ano",
 title = "Distribuição da expectativa de vida por ano") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



## 9.15 Cleveland Dot Plot

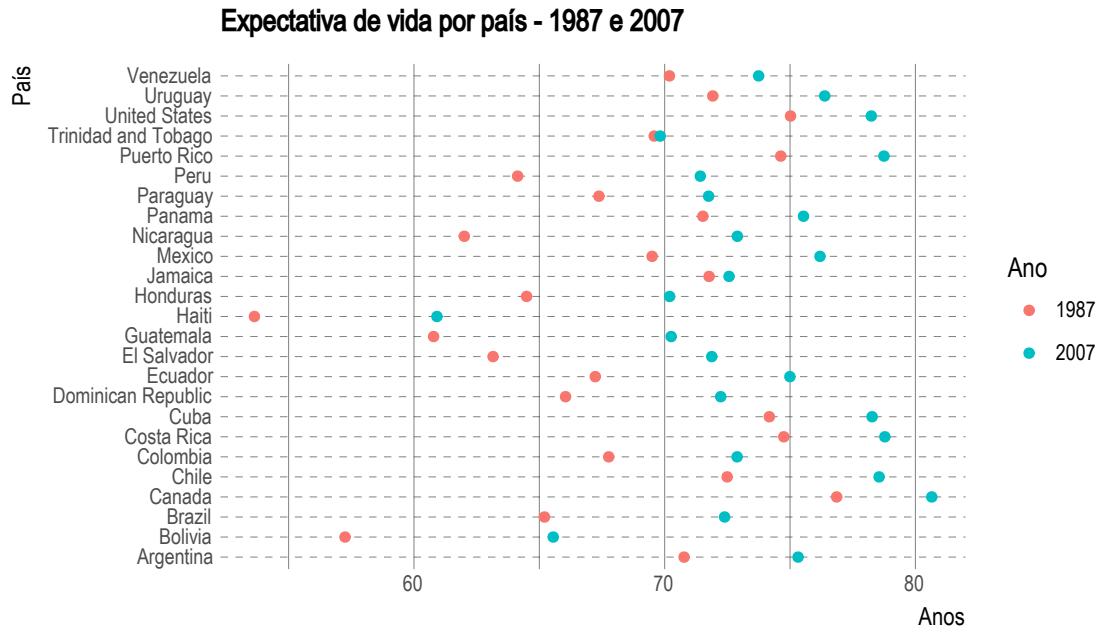
O cleveland dot plot é uma visualização que pode substituir os gráficos de barras. A ideia é que o gráfico fica menos poluído com os pontos, fazendo que o leitor foque no que é importante. Vamos criar um gráfico para comparar as expectativas de vida no ano de 2007 para os países das Américas:

```
gapminder %>%
 filter(year == 2007,
 continent == "Americas") %>%
 ggplot(aes(x = lifeExp, y = reorder(country, lifeExp))) +
 geom_point(size = 3, color = "dodgerblue") +
 labs(title = "Expectativa de vida por país - 2007",
 y = "País",
 x = "Anos") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10) +
 theme(panel.grid.major.y = element_line(linetype = "dashed"))
```



Esse tipo de gráfico também pode apresentar mais de um ponto para cada valor da variável categórica (país):

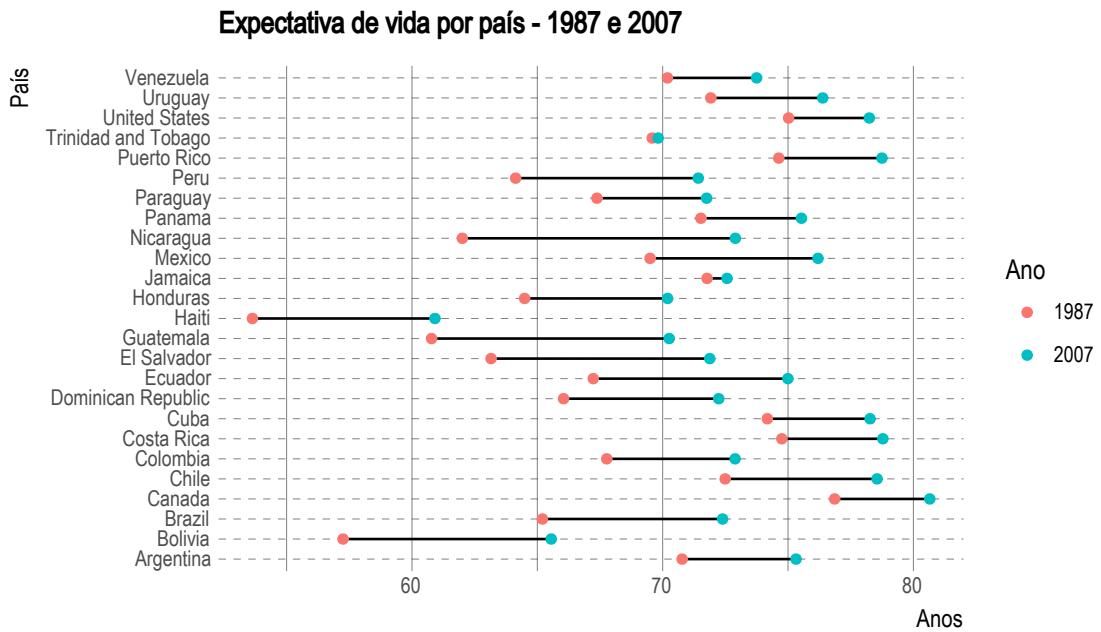
```
gapminder %>%
 filter(year %in% c(1987, 2007),
 continent == "Americas") %>%
 ggplot(aes(x = lifeExp, y = country, color = factor(year))) +
 geom_point(aes(color = factor(year))) +
 labs(title = "Expectativa de vida por país - 1987 e 2007",
 y = "País",
 x = "Anos",
 color = "Ano") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10) +
 theme(panel.grid.major.y = element_line(linetype = "dashed"))
```



No gráfico acima, vemos dois pontos para cada país. Um para representar 1987 e outro para 2007.

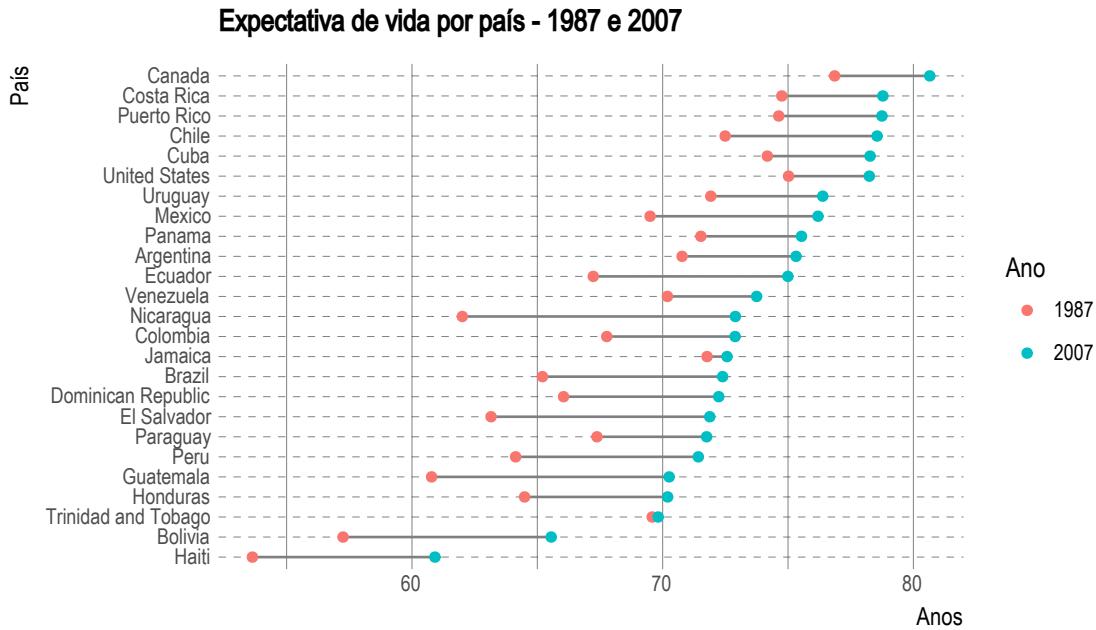
Para completar o gráfico, precisamos adicionar uma linha conectando esses dois pontos. Esse gráfico é chamado de *connected dot plot*. Um detalhe importante é que queremos criar uma linha por país. Assim, usaremos o elemento estético `group` para obter o resultado esperado:

```
gapminder %>%
 filter(year %in% c(1987, 2007),
 continent == "Americas") %>%
 ggplot(aes(x = lifeExp, y = country)) +
 geom_line(aes(group = country)) +
 geom_point(aes(color = factor(year))) +
 labs(title = "Expectativa de vida por país - 1987 e 2007",
 y = "País",
 x = "Anos",
 color = "Ano") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10) +
 theme(panel.grid.major.y = element_line(linetype = "dashed"))
```



Para finalizar, vamos ordenar o eixo y pela expectativa de vida:

```
gapminder %>%
 filter(year %in% c(1987, 2007),
 continent == "Americas") %>%
 ggplot(aes(x = lifeExp, y = reorder(country, lifeExp, max))) +
 geom_line(aes(group = country), color = "grey50") +
 geom_point(aes(color = factor(year))) +
 labs(title = "Expectativa de vida por país - 1987 e 2007",
 y = "País",
 x = "Anos",
 color = "Ano") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10) +
 theme(panel.grid.major.y = element_line(linetype = "dashed"))
```



## 9.16 Textos/Rótulos

```
geom_label(mapping = NULL, data = NULL, stat = "identity", position = "identity",
..., parse = FALSE, nudge_x = 0, nudge_y = 0,
label.padding = unit(0.25, "lines"), label.r = unit(0.15, "lines"),
label.size = 0.25, na.rm = FALSE, show.legend = NA, inherit.aes = TRUE)

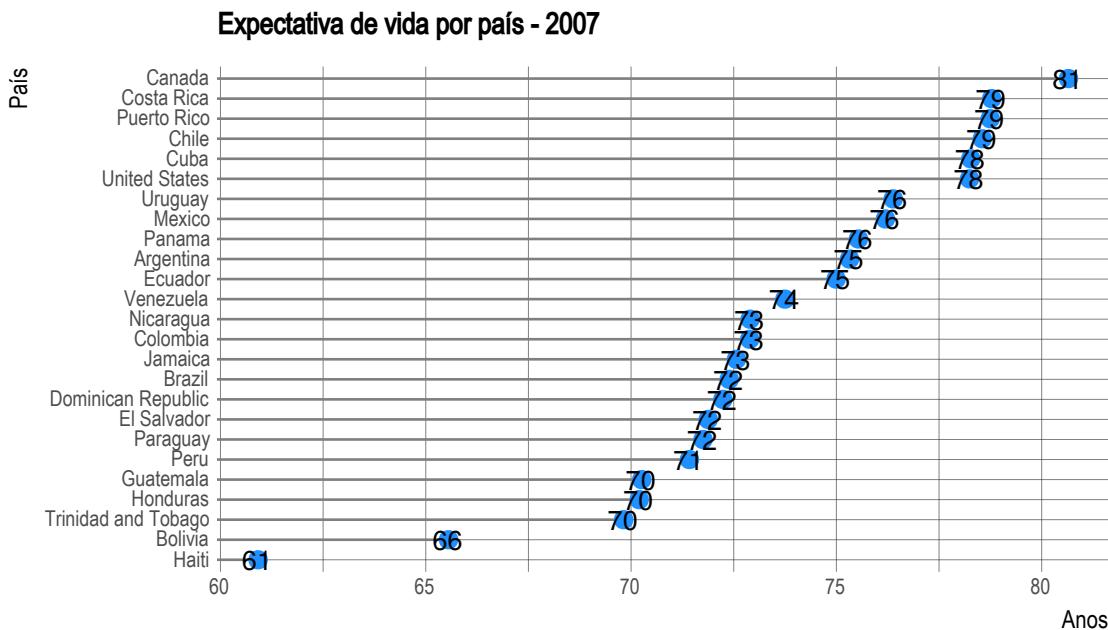
geom_text(mapping = NULL, data = NULL, stat = "identity", position = "identity",
..., parse = FALSE, nudge_x = 0, nudge_y = 0, check_overlap = FALSE,
na.rm = FALSE, show.legend = NA, inherit.aes = TRUE)
```

Os parâmetros estéticos (**aes**) são:

- **label**
- **x**
- **y**
- **alpha**
- **angle**
- **colour**
- **family**
- **fontface**
- **hjust**
- **lineheight**
- **size**
- **vjust**

Para adicionar textos ou rótulos usamos, respectivamente, o `geom_text()` ou `geom_label()`. Eles se diferenciam na formatação. Isto ficará mais claro nos exemplos.

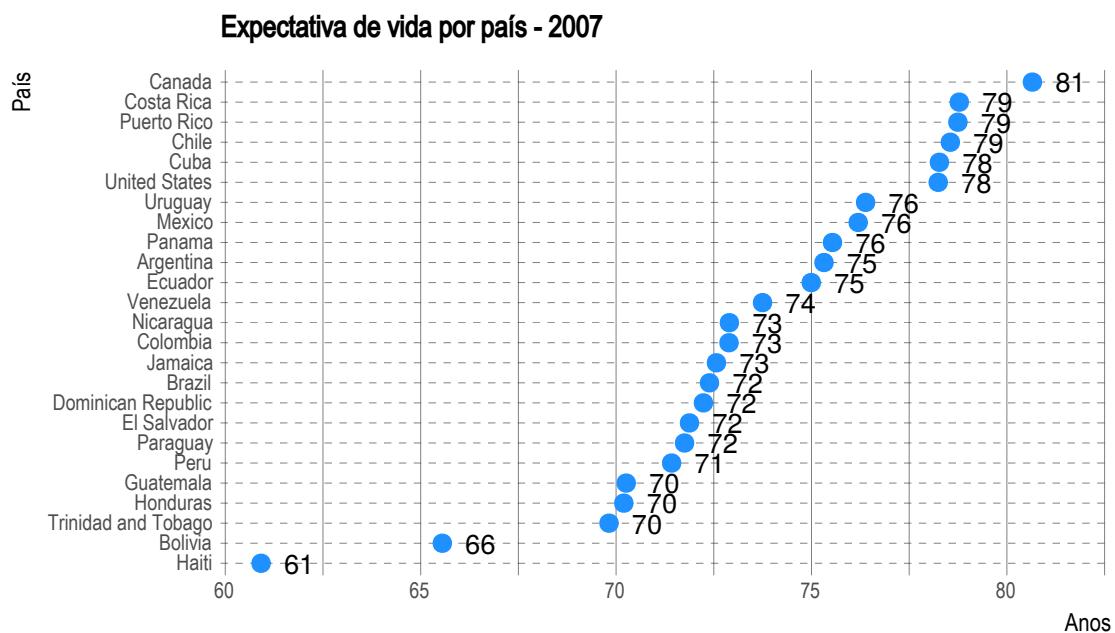
```
gapminder %>%
 filter(year == 2007,
 continent == "Americas") %>%
 ggplot(aes(x = lifeExp, y = reorder(country, lifeExp))) +
 geom_segment(x = 0, aes(xend = lifeExp, yend = country),
 color = "grey50") +
 geom_point(size = 3, color = "dodgerblue") +
 geom_text(aes(label = round(lifeExp))) +
 labs(title = "Expectativa de vida por país - 2007",
 y = "País",
 x = "Anos") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



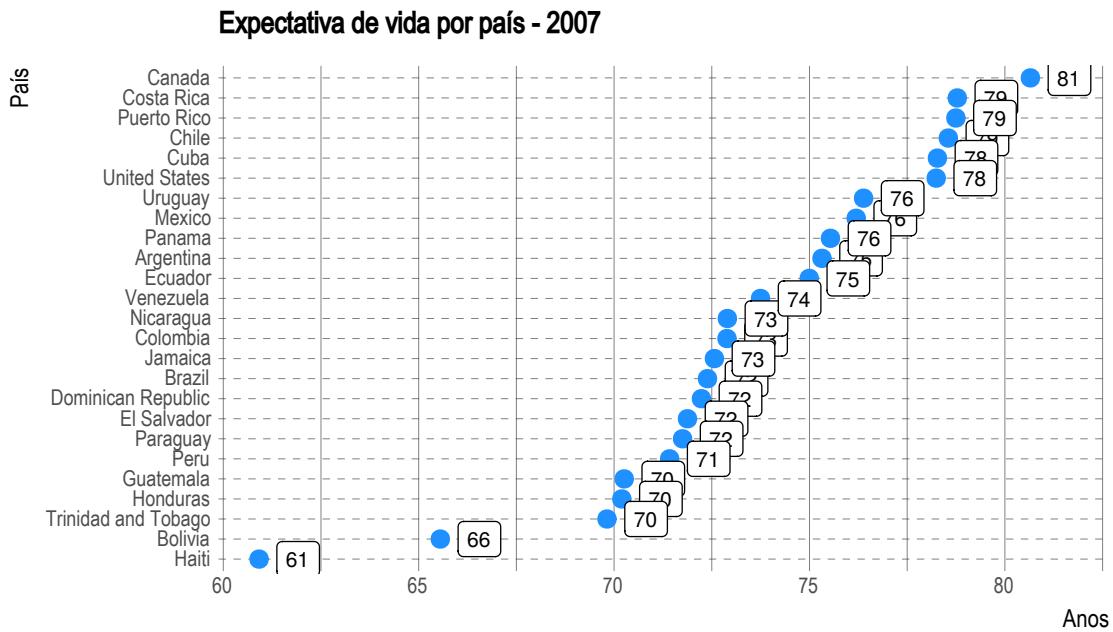
O resultado ficou bem ruim. O texto foi inserido exatamente na posição (x, y). Para alterar a posição podemos usar os argumentos `hjust`, `vjust`, `nudge_x` e `nudge_y`. O `hjust` e `vjust` podem assumir valor entre 0 (direita/embaixo) e 1(esquerda/em cima) ou “left”, “middle”, “right”, “bottom”, “center” e “top”, além de “inward” e “outward”.

```
gapminder %>%
 filter(year == 2007,
 continent == "Americas") %>%
 ggplot(aes(x = lifeExp, y = reorder(country, lifeExp))) +
 geom_point(size = 3, color = "dodgerblue") +
 geom_text(aes(label = round(lifeExp)), nudge_x = 1) +
 labs(title = "Expectativa de vida por país - 2007",
 y = "País",
 x = "Anos") +
```

```
theme_ipsum(plot_title_size = 12,
 axis_title_size = 10) +
theme(panel.grid.major.y = element_line(linetype = "dashed"))
```



```
gapminder %>%
 filter(year == 2007,
 continent == "Americas") %>%
 ggplot(aes(x = lifeExp, y = reorder(country, lifeExp))) +
 geom_point(size = 3, color = "dodgerblue") +
 geom_label(aes(label = round(lifeExp)), nudge_x = 1, size = 3) +
 labs(title = "Expectativa de vida por país - 2007",
 y = "País",
 x = "Anos") +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10) +
 theme(panel.grid.major.y = element_line(linetype = "dashed"))
```

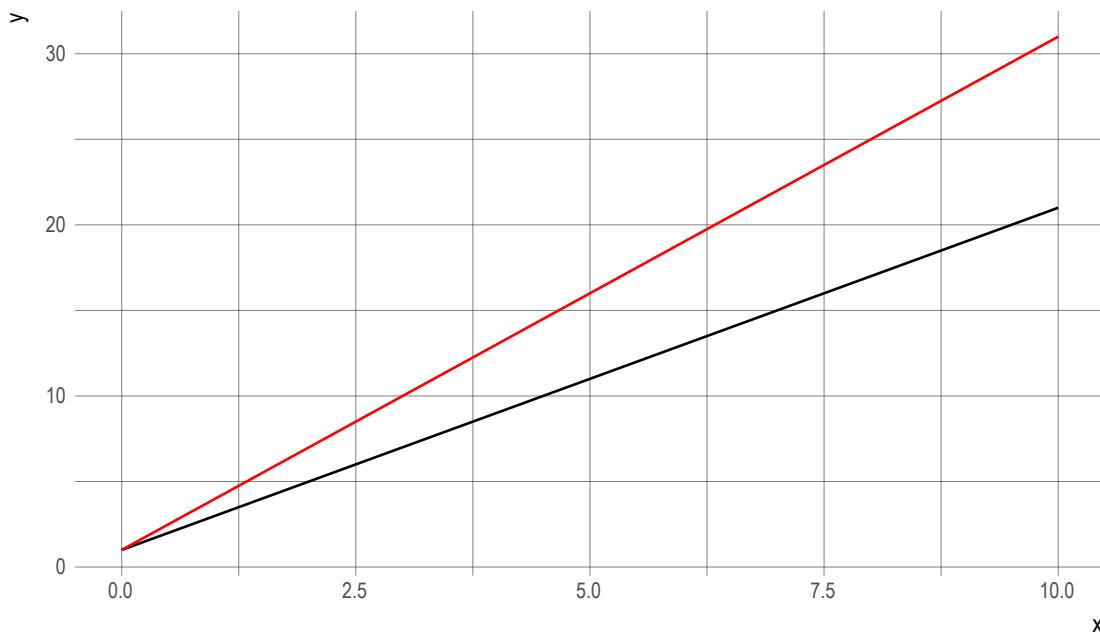


## 9.17 Plotando funções

```

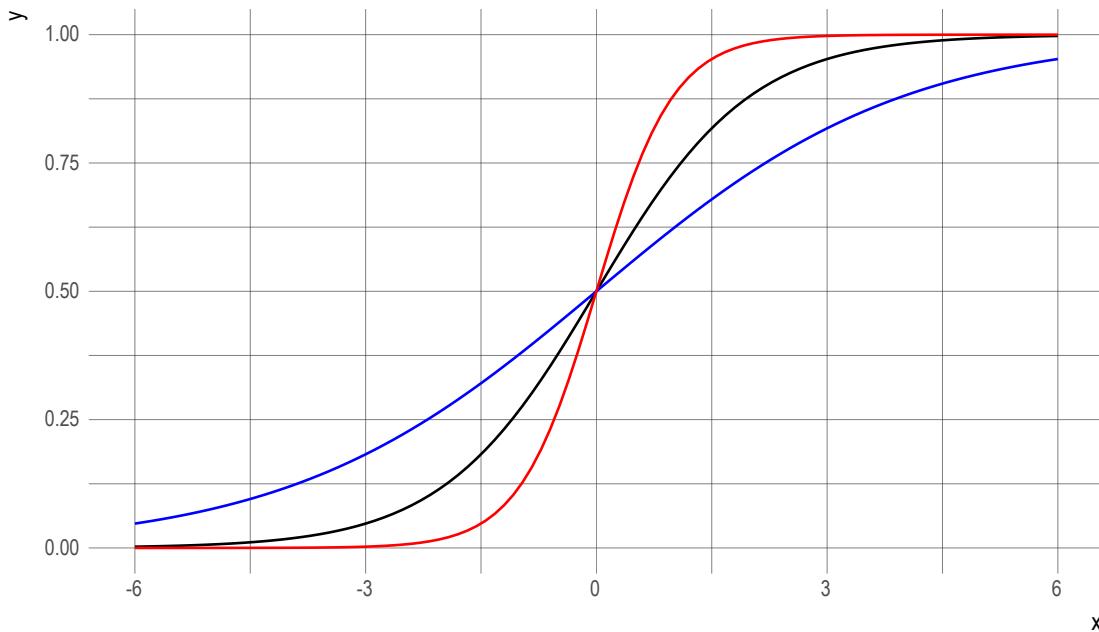
reta <- function(a, b, x){
 a + b * x
}
data <- data.frame(x = seq(0, 10, by = 0.1))
ggplot(data, aes(x = x)) +
 stat_function(fun = reta, args = list(a = 1, b = 2)) +
 stat_function(fun = reta, args = list(a = 1, b = 3), col = 'red')

```



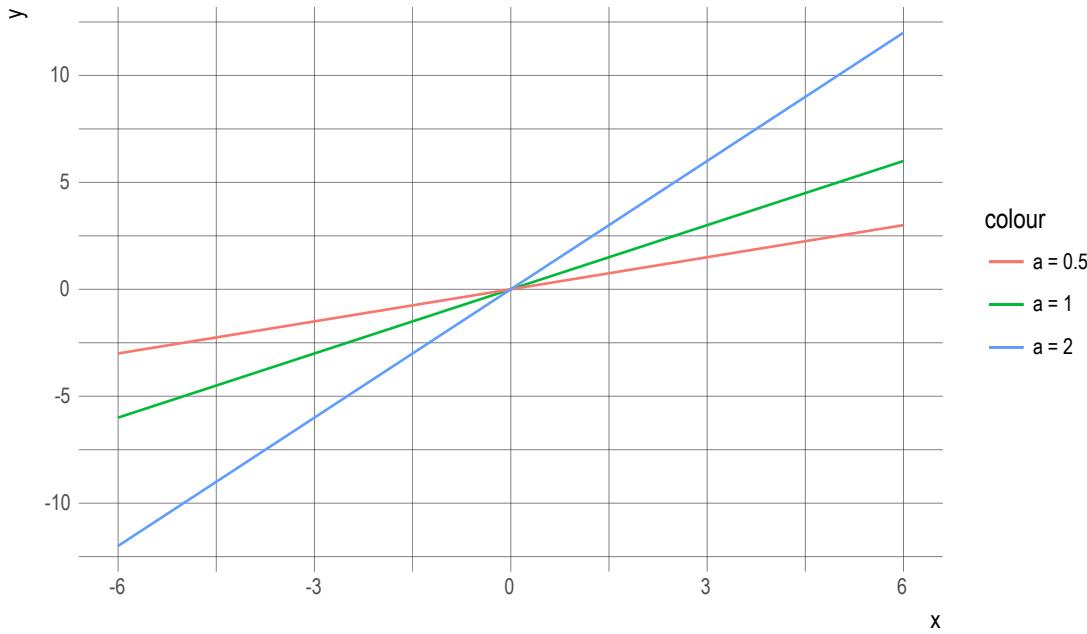
```
sigmoid <- function(a = 1,z){
 1/(1 + exp(-a * z))
}

data <- data.frame(x = -6:6)
ggplot(data, aes(x = x)) +
 stat_function(fun = sigmoid, args = list(a = 1)) +
 stat_function(fun = sigmoid, args = list(a = 0.5), color = "blue") +
 stat_function(fun = sigmoid, args = list(a = 2), color = "red")
```



```
logit <- function(a, z){
 log(sigmoid(a, z)/(1 - sigmoid(a, z)))
}

data <- data.frame(x = -6:6)
ggplot(data, aes(x = x)) +
 stat_function(fun = logit, args = list(a = 1), aes(color = "a = 1")) +
 stat_function(fun = logit, args = list(a = 0.5), aes(color = "a = 0.5")) +
 stat_function(fun = logit, args = list(a = 2), aes(color = "a = 2"))
```



## 9.18 Mapas

No primeiro exemplo, usaremos um arquivo de dados que já possui as informações necessárias para criar o mapa. No segundo, será mostrado o processo a partir de um shapefile (\*.shp) que é um formato utilizado para o armazenamento de dados geoespaciais.

Os dados para criação do mapa estão disponíveis no arquivo `world_map.csv`. Nesse arquivo, existem 177 regiões e, onde possível, estão correlacionadas com o nome disponível na base do MDIC.

```
library(readr)
world_map <- read_delim('dados/world_map.csv', delim = ";",
 locale = locale(encoding = "ISO-8859-1",
 decimal_mark = ","))

Parsed with column specification:
cols(
long = col_double(),
lat = col_double(),
order = col_integer(),
hole = col_logical(),
piece = col_integer(),
id = col_character(),
group = col_character(),
NO_PAIS_POR = col_character()
)
head(world_map)

A tibble: 6 x 8
```

```

long lat order hole piece id group
<dbl> <dbl> <int> <lgl> <int> <chr> <chr>
1 61.21082 35.65007 1 FALSE 1 Afghanistan Afghanistan.1
2 62.23065 35.27066 2 FALSE 1 Afghanistan Afghanistan.1
3 62.98466 35.40404 3 FALSE 1 Afghanistan Afghanistan.1
4 63.19354 35.85717 4 FALSE 1 Afghanistan Afghanistan.1
5 63.98290 36.00796 5 FALSE 1 Afghanistan Afghanistan.1
6 64.54648 36.31207 6 FALSE 1 Afghanistan Afghanistan.1
... with 1 more variables: NO_PAIS_POR <chr>

```

As colunas long e lat serão mapeadas para os eixos x e y, e será utilizado o objeto geométrico polígono (`geom_polygon()`). A coluna group será utilizada para que seja criado um polígono para cada região, evitando sobreposições das linhas.

Para criação do gráfico, iremos remover a região Antarctica e realizar um `left_join()` com os dados de exportações por país de destino no ano de 2015, disponível no arquivo EXP\_2015\_PAIS.txt. Também iremos criar classes para os valores exportados.

```

Remover Antarctica
world_map <- world_map %>%
 filter(id != "Antarctica")

exp.2015 <- read_delim('dados/EXP_2015_PAIS.csv', delim = ";",
 locale = locale(encoding = "ISO-8859-1"),
 col_types = 'ccd')

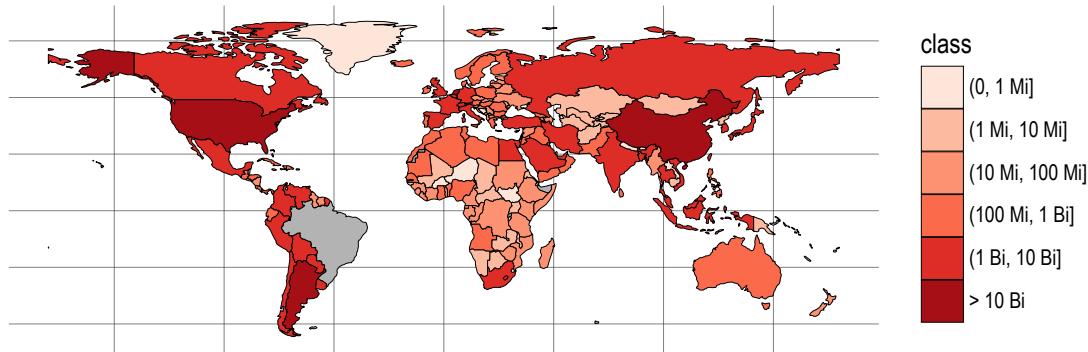
world_map <- left_join(world_map, exp.2015, by = "NO_PAIS_POR") %>%
 mutate(class = cut(VL_FOB, breaks = c(0, 1e6, 10e6, 100e6, 1e9, 10e9, Inf)))

ggplot(world_map, aes(x = long, y = lat, group = group)) +
 geom_polygon(aes(fill = class), col = 'black', size = 0.1) +
 scale_fill_brewer(palette = "Reds", breaks = levels(world_map$class),
 labels = c("(0, 1 Mi]", "(1 Mi, 10 Mi]", "(10 Mi, 100 Mi]",
 "(100 Mi, 1 Bi]", "(1 Bi, 10 Bi]", "> 10 Bi"),
 na.value = 'grey70') +
 labs(title = "Exportações Brasileiras - 2015",
 subtitle = 'Valor FOB') +
 coord_quickmap() +
 theme(axis.text.x = element_blank(),
 axis.text.y = element_blank(),
 axis.title.x = element_blank(),
 axis.title.y = element_blank())

```

### Exportações Brasileiras - 2015

Valor FOB



Como dito, o segundo exemplo será realizado a partir arquivos do tipo shapefile. O IBGE disponibiliza shapefiles para o Brasil em diferentes níveis, UF's, municípios, mesorregiões e microrregiões. Os arquivos estão disponíveis neste link.

Para leitura do shapefile, serão necessários alguns pacotes: `rgdal`, `maptools`, `rgeos` e `broom`. A função `ogrListLayers()` lista as camadas disponíveis em um shapefile. Para ler o arquivo para o R, utiliza-se a função `readOGR()`. Após ler o shapefile para o R, é preciso transformá-lo para um formato em que o `ggplot2` possa gerar a visualização. A função `tidy()` realiza essa tarefa. Essa função também transforma objetos de outras classes para `data.frames`.

```
library(rgdal)
library(maptools)
library(rgeos)
library(broom)

ogrListLayers('dados/mapas/mg_municípios/31MUE250GC_SIR.shp')

[1] "31MUE250GC_SIR"
attr(),"driver")
[1] "ESRI Shapefile"
attr(),"nlayers")
[1] 1

mg_mapa <- readOGR('dados/mapas/mg_municípios/31MUE250GC_SIR.shp',
 layer = '31MUE250GC_SIR')

OGR data source with driver: ESRI Shapefile
Source: "dados/mapas/mg_municípios/31MUE250GC_SIR.shp", layer: "31MUE250GC_SIR"
with 853 features
It has 2 fields
```

Os arquivos shapefile também fornecem dados sobre as regiões disponíveis no arquivo. No caso do exemplo

acima, são fornecidos o código e o nome do município.

```
mg_mapa_data <- data.frame(mg_mapa)

head(mg_mapa_data)

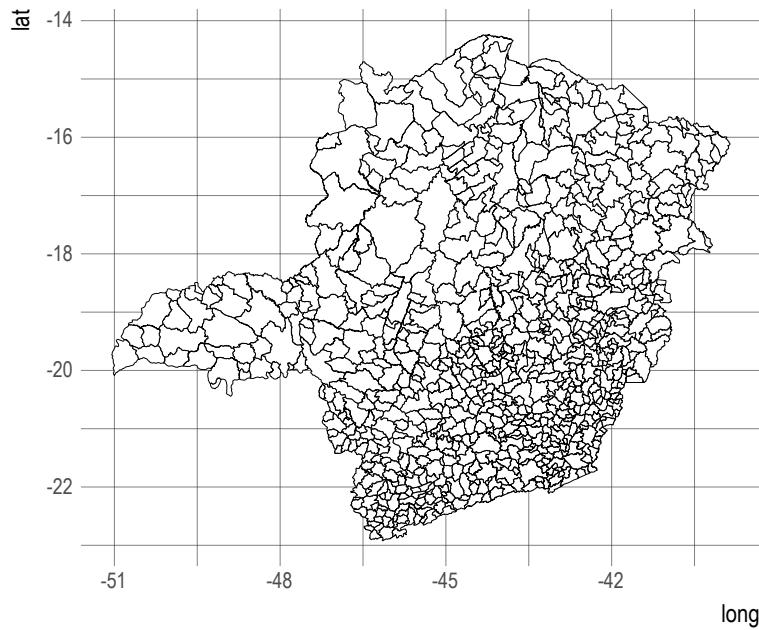
NM_MUNICIP CD_GEOCMU
0 ABADIA DOS DOURADOS 3100104
1 ABAETÉ 3100203
2 ABRE CAMPO 3100302
3 ACAIACA 3100401
4 AÇUCENA 3100500
5 ÁGUA BOA 3100609
```

Na criação do data.frame com uso da função `fortify()`, é preciso informar a variável que identificará a região. No exemplo, `CD_GEOCMU`.

```
#mg_mapa <- fortify(mg_mapa, region = "CD_GEOCMU")
mg_mapa <- tidy(mg_mapa, region = "CD_GEOCMU")

Para adicionar os nomes dos municípios
mg_mapa <- left_join(mg_mapa, mg_mapa_data, by = c("id" = "CD_GEOCMU"))

Warning: Column `id`/`CD_GEOCMU` joining character vector and factor,
coercing into character vector
ggplot(mg_mapa, aes(x = long, y = lat, group = group)) +
 geom_polygon(color = 'black', fill = 'white', size = 0.1) +
 coord_quickmap()
```



Com o mapa pronto, faltam os dados que serão plotados.

```
REM_RAIS_MG_2015 <- read_delim('dados/REM_RAIS_MG_2015.csv',
 delim = ";",
 col_types = 'cd',
 skip = 1,
 col_names = c("Mun.Trab", "mediana"),
 locale = locale(encoding = 'ISO-8859-1'))

REM_RAIS_MG_2015 <- REM_RAIS_MG_2015 %>%
 mutate(mediana = ifelse(mediana > 1500, 1500, mediana))
head(REM_RAIS_MG_2015)
```

```
A tibble: 6 x 2
Mun.Trab mediana
<chr> <dbl>
1 315820 788.0
2 313865 827.4
3 311880 831.0
4 313980 831.0
5 312290 831.0
6 315760 831.0

mg_mapa <- mg_mapa %>%
 mutate(Mun.Trab = substr(id, 1, 6))

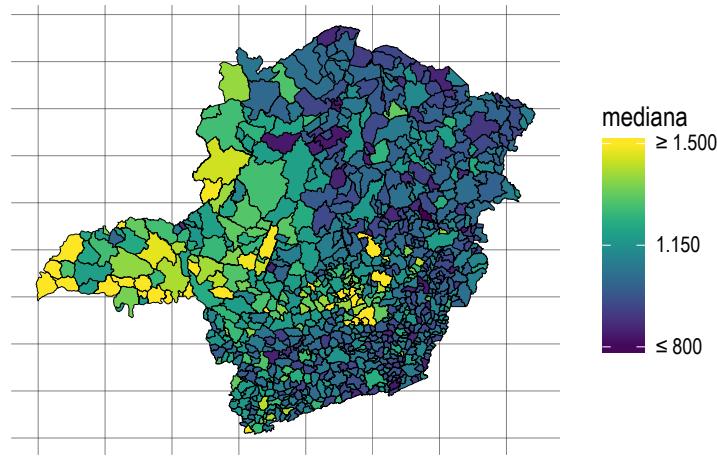
mg_mapa <- left_join(mg_mapa, REM_RAIS_MG_2015, by = "Mun.Trab")
```

Criar a visualização:

```
cores.viridis <- viridis::viridis(n = 20)
ggplot(mg_mapa, aes(x = long, y = lat, group = group, fill = mediana)) +
 geom_polygon(color = 'black', size = 0.1) +
 scale_fill_gradientn(colours = cores.viridis,
 breaks = c(800, 1150, 1500),
 labels = c("\u2264 800", "1.150", "\u2265 1.500")) +
 labs(title = "Mediana da Remuneração ",
 subtitle = "Minais Gerais - Dezembro/2015. Valores em R$",
 caption = "Fonte: RAIS/MTE") +
 coord_quickmap() +
 theme(axis.text.x = element_blank(),
 axis.text.y = element_blank(),
 axis.title.x = element_blank(),
 axis.title.y = element_blank())
```

### Mediana da Remuneração

Minais Gerais - Dezembro/2015. Valores em R\$



Fonte: RAIS/MTE

#### 9.18.1 Simplificando o shapefile

Algumas vezes o shapefile pode ser bastante grande, tornando a renderização e o arquivo final relativamente pesado. É possível simplificar o shapefile com a função `gSimplify()`. Veja o exemplo abaixo.

```
library(rgeos)
mg_mapa <- readOGR('dados/mapas/mg_municípios/31MUE250GC_SIR.shp',
 layer = '31MUE250GC_SIR')

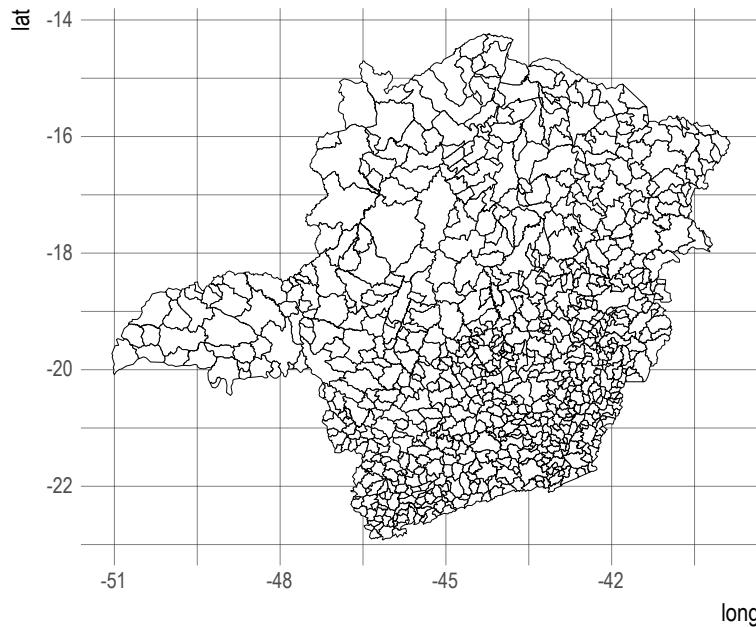
OGR data source with driver: ESRI Shapefile
Source: "dados/mapas/mg_municípios/31MUE250GC_SIR.shp", layer: "31MUE250GC_SIR"
with 853 features
It has 2 fields

df <- data.frame(mg_mapa)
mg_mapa <- gSimplify(mg_mapa, tol=0.01, topologyPreserve = TRUE)
mg_mapa = SpatialPolygonsDataFrame(mg_mapa, data=df)

mg_mapa <- tidy(mg_mapa, region = "CD_GEOCMU")

Para adicionar os nomes dos municípios
mg_mapa <- left_join(mg_mapa, mg_mapa_data, by = c("id" = "CD_GEOCMU"))

Warning: Column `id`/`CD_GEOCMU` joining character vector and factor,
coercing into character vector
ggplot(mg_mapa, aes(x = long, y = lat, group = group)) +
 geom_polygon(color = 'black', fill = 'white', size = 0.1) +
 coord_quickmap()
```



### 9.18.2 Encoding dos dados do shapefile

É possível que os nomes das regiões apresentem problemas de codificações se o arquivo estiver em UTF-8 e o usuário esteja usando o Windows. Nesse caso, pode-se usar a função `iconv()` para converter o encoding.

```
mg_mapa <- readOGR('dados/mapas/mg_municípios/31MUE250GC_SIR.shp',
 layer = '31MUE250GC_SIR')
mg_mapa@data$NM_MUNICIP <- iconv(mg_mapa@data$NM_MUNICIP,
 from = "UTF-8",
 to = "ISO-8859-1")
```

## 9.19 Salvando Gráficos

Há duas formas de salvar o gráfico gerado pelo ggplot2. A primeira delas é clicando no botão **Export** na aba **Plots**. Lá existirão três opções: exportar para o clipboard, salvar como PDF ou salvar como imagem.

```
knitr::include_graphics('images/salvar_plot.gif')
```

A outra opção é usar a função `ggsave()`:

```
ggsave(filename, plot = last_plot(), device = NULL, path = NULL,
 scale = 1, width = NA, height = NA, units = c("in", "cm", "mm"),
 dpi = 300, limitsize = TRUE, ...)
```

O argumento `device` é usado para escolher o tipo de arquivo (“eps”, “ps”, “tex” (pictex), “pdf”, “jpeg”, “tiff”, “png”, “bmp”, “svg” or “wmf”).

## 9.20 Extensões do ggplot2

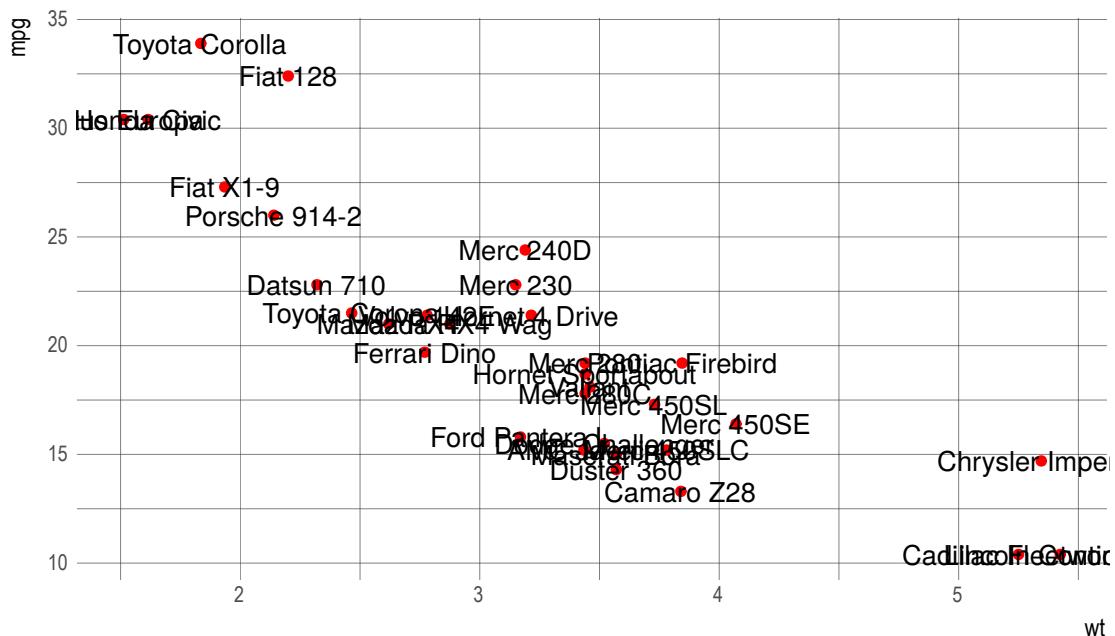
Um conjunto de pacotes fornece extensões ao ggplot2. Ou seja, criam funcionalidades não existentes no pacote original.

Veja neste link uma galeria com as extensões. Aqui, vamos somente exemplificar algumas.

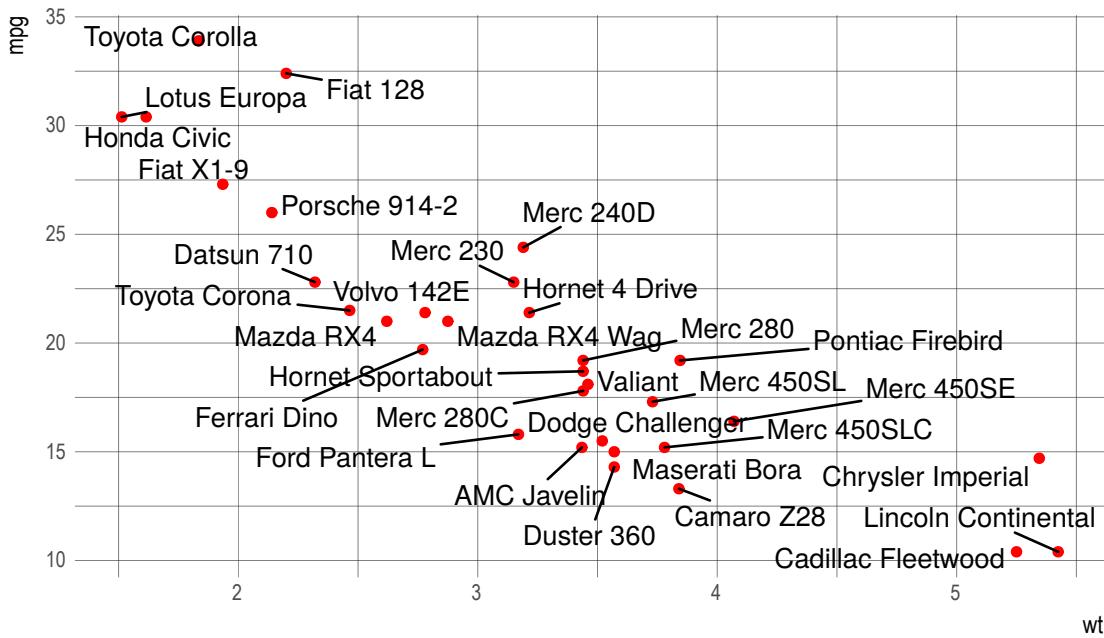
### 9.20.1 ggrepel

O `ggrepel` é importante para evitar que textos (`geom_text`) e rótulos (`geom_label`) se sobreponham. Veja o mesmo gráfico sem e com o `ggrepel`:

```
library(ggrepel)
data(mtcars)
ggplot(mtcars, aes(wt, mpg)) +
 geom_point(color = 'red') +
 geom_text(aes(label = rownames(mtcars))) +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



```
library(ggrepel)
data(mtcars)
ggplot(mtcars, aes(wt, mpg)) +
 geom_point(color = 'red') +
 geom_text_repel(aes(label = rownames(mtcars))) +
 theme_ipsum(plot_title_size = 12,
 axis_title_size = 10)
```



### 9.20.2 gganimate

Outra extensão muito interessante é o `gganimate`. Com essa extensão é possível criar gifs (animações) de gráficos do `ggplot2`. O ponto fundamental é definir uma variável que irá controlar os `frames`, ou seja, as imagens que serão sobrepostas para compor a animação.

Veja o exemplo abaixo:

```
O pacote não está no cran
devtools::install_github("dgrtwo/gganimate")
library(gganimate)

p <- ggplot(gapminder, aes(x = gdpPercap, y = lifeExp,
 size = pop, color = continent,
 frame = year)) +
 geom_point() +
 scale_x_log10()

animation::ani.options(interval = 1)
x <- gganimate(p, filename = 'images/gapminder1.gif',
 ani.width = 750,
 ani.height = 450)
```

É possível suavizar a animação com o pacote `tweenr`. É um pouco complicado, mas fica o exemplo abaixo retirado desse post:

```
library(tweenr)

years <- unique(gapminder$year)
```

```

gapminder_list <- list()
for(i in 1:length(years)){
 j <- years[i]
 gapminder_list[[i]] <- gapminder %>%
 filter(year == j)
}

tf <- tween_states(gapminder_list,
 tweenlength = 2,
 statelength = 0,
 ease = rep("linear", length(gapminder_list)),
 nframes = 308)

tf2 <- expand.grid(y = 80, x = 10^(2.5), year = seq(1957, 2007, 5))
tf2 <- split(tf2, tf2$year)
tf2 <- tween_states(tf2,
 tweenlength = 2,
 statelength = 0,
 ease = rep("linear", length(tf2)),
 nframes = 308)
tf2 <- tf2 %>%
 mutate(year = rep(seq(1957, 2007, 5), each = 29)[1:310])

p2 <- ggplot(tf,
 aes(x=gdpPerCap, y=lifeExp, frame = .frame)) +
 geom_point(aes(size=pop, color=continent), alpha=0.8) +
 geom_text(data = tf2, aes(x = x, y = y, label = year)) +
 xlab("GDP per capita") +
 ylab("Life expectancy at birth") +
 scale_x_log10()
animation::ani.options(interval = 1/20)
x <- gganimate(p2, filename = 'images/gapminder2.gif',
 ani.width = 750,
 ani.height = 450,
 title.frame = FALSE)

```

## 9.21 Exercícios

1. Crie um gráfico de dispersão usando a base `gapminder` para o ano de 2007. Mapeie uma variável para o tamanho do ponto e adicione os títulos do gráfico e dos eixos.
2. No gráfico anterior, como você faria para que o ponto do Brasil fosse identificado com uma cor adicional?  
Dica: se uma nova camada usar um segundo conjunto de dados é preciso informar usando o argumento `data`. Exemplo, `geom_point(data = novo_data_frame, ...)`
3. Crie um histograma da variável `wage` a partir da base de dados `Wage` do pacote `ISLR`. Crie uma visualização distribuição da variável `wage` por nível da variável `education`. Mapeie a variável `education` para o elemento estético `fill`.
4. Crie um `data.frame` chamado `Wage2` em que a variável `education` é removida. Adicione a seguinte camada no início do código:

```
geom_histogram(data = Wage2, fill = "grey50", alpha = 0.5)
```

Veja e interprete o resultado.

5. A partir do código abaixo crie um gráfico que apresenta os 10 países que tiveram maior crescimento do PIB em 2016 e aqueles 10 que tiveram o menor. Use o `facet_wrap` para quebrar a visualização em dois gráficos. Escolha o objeto geométrico `geom_col` ou `geom_point()`.

```
library(WDI)
```

```
Loading required package: RJSONIO
##
Attaching package: 'RJSONIO'
##
The following objects are masked from 'package:jsonlite':
fromJSON, toJSON
library(dplyr)
gdp_growth <- WDI(indicator = "NY.GDP.MKTP.KD.ZG",
 start = 2016,
 end = 2016,
 extra = TRUE)

Remove regiões - ISO's com números
gdp_growth <- gdp_growth %>%
 filter(!is.na(region) & region != "Aggregates" & !is.na(NY.GDP.MKTP.KD.ZG))
```

6. Utilize o mapa mundi disponível no pacote `choroplethrMaps` e os dados que criamos no exercício anterior para plotar as variações do PIB em um mapa.

```
library(choroplethrMaps)
data("country.map")

Dados - Crescimento do PIB em 2016
library(WDI)
library(dplyr)
gdp_growth <- WDI(indicator = "NY.GDP.MKTP.KD.ZG",
 start = 2016,
 end = 2016,
 extra = TRUE)

Remove regiões - ISO's com números
gdp_growth <- gdp_growth %>%
 filter(!is.na(region) & region != "Aggregates" & !is.na(NY.GDP.MKTP.KD.ZG))

Continue fazendo o join entre country.map e gdp_growth
chaves: wb_a3 e iso3c

country.map <- country.map %>%
 left_join(gdp_growth, by = c("wb_a3" = "iso3c"))

ggplot(country.map, aes(x = long, y = lat, group = group)) +
 geom_polygon(aes(fill = NY.GDP.MKTP.KD.ZG)) +
 scale_fill_continuous("Var. % GDP") +
 coord_quickmap()
```

7. Utilizando o código “NY.GDP.PCAP.KD” e o pacote `WDI`, crie um gráfico do tipo connected dot plot comparando a renda per capita do Brasil e mais 5 países nos anos de 1990 e 2010.

# Capítulo 10

## Visualizações Interativas

### 10.1 Introdução

No R, as visualizações interativas são geralmente criadas a partir de pacotes que utilizam um framework chamado `htmlwidgets`. O `htmlwidgets` é um pacote/framework que facilita o uso de bibliotecas javascript de visualizações para o ambiente R, sendo possível usá-las no console, no RMarkdown e no Shiny. Basicamente, javascript é uma linguagem *client-side*, ou seja, o processamento ocorre no lado do cliente e é utilizada, principalmente, para alterar códigos HTML e CSS (estilos) interativamente.

O `htmlwidgets` busca fazer a ponte entre o R e alguma biblioteca javascript de visualização, fazendo com que o usuário do R consiga utilizar essas bibliotecas sem necessariamente precisar escrever uma linha de código em javascript. Apesar de não ser necessário, pode ser que em algum momento, a fim de customizar uma visualização, seja necessário algum código em javascript, mas isso não será tratado aqui.

Abaixo estão listados alguns projetos de visualizações de dados em javascript:

1. D3
2. D3plus
3. Highcharts
4. C3
5. Leaflet
6. Plotly

Existe uma quantidade significativa de pacotes que utilizam o `htmlwidgets`. Para se ter uma noção, visite a esta galeria.

O objetivo deste capítulo é fazer uma pequena apresentação sobre alguns pacotes. Cada pacote possui um conjunto de detalhes, que torna inviável apresentá-los nesse curso. Dessa forma, vamos fazer um breve tour por alguns pacotes, começando com o `plotly`.

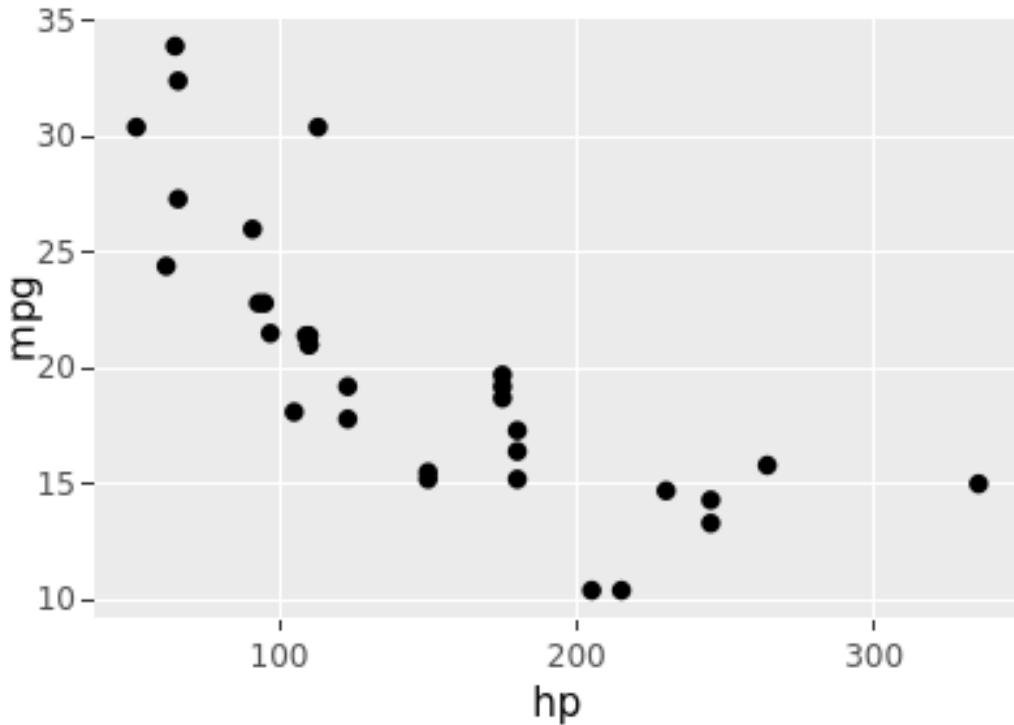
### 10.2 Plotly

O `plotly` é um pacote interessante, uma vez que ele se integra com o `ggplot2`:

```
library(plotly)
library(tidyverse)

p <- ggplot(mtcars, aes(x = hp, y = mpg)) +
```

```
geom_point()
ggplotly(p)
```



Basta que o objeto criado com o ggplot2 seja passado para a função `ggplotly()`. Note que a visualização possui uma série de funcionalidades, como *tooltips*, possibilidade de zoom, download da imagem etc.

### 10.3 dygraphs

O dygraphs é uma biblioteca para visualizações de séries temporais. Os detalhes do pacote estão disponíveis neste link.

Antes dos exemplos, será necessário falar sobre objetos de séries de tempo no R.

Para criar um objeto de séries de tempo usaremos a função `ts()`:

```
ts(data = NA, start = 1, end = numeric(), frequency = 1,
 deltat = 1, ts.eps = getOption("ts.eps"), class = , names =)
```

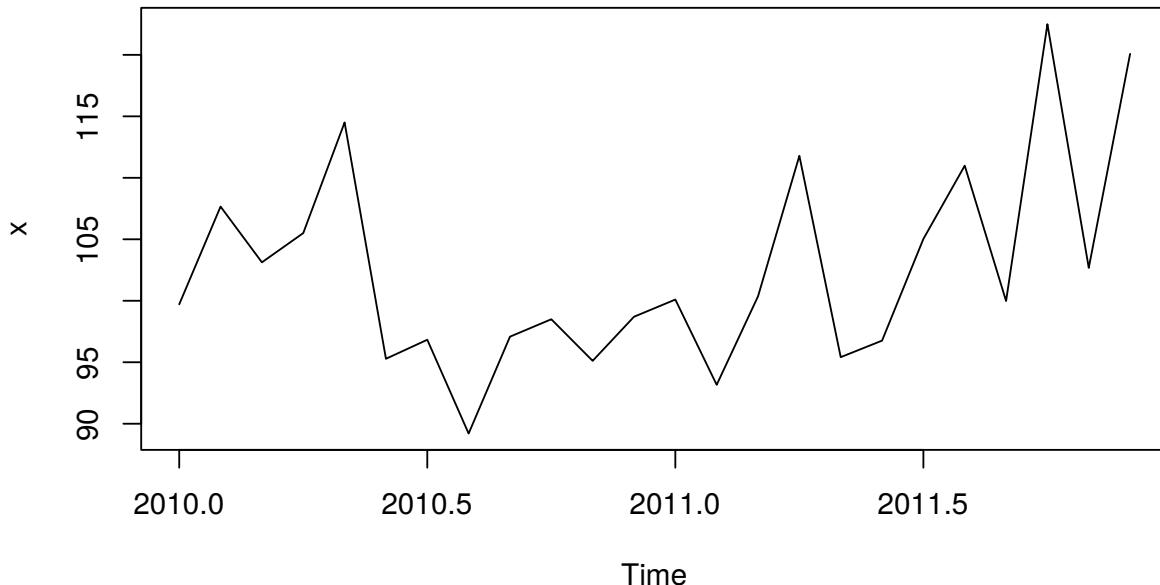
Os parâmetros relevantes são:

- `data`: um vetor ou uma matriz de valores da(s) série(s) de tempo. Um data.frame é transformado automaticamente em uma matriz;

- **start**: o período da primeira observação. Pode ser um valor único ou um vetor de dois inteiros. Geralmente, utiliza-se a segunda opção. Por exemplo, Janeiro de 1997: `start = c(1997, 1)`;
- **end**: o período da última observação. Similar ao **start**, porém não é obrigatório;
- **frequency**: frequência dos dados. Mensal(12), Trimestral (4), Anual(1) etc.

Exemplo de criação de uma série de tempo:

```
x <- rnorm(24, mean = 100, sd = 10)
Transformando em série mensal a partir de janeiro de 2010
x <- ts(x, freq = 12, start = c(2010, 1))
plot(x)
```



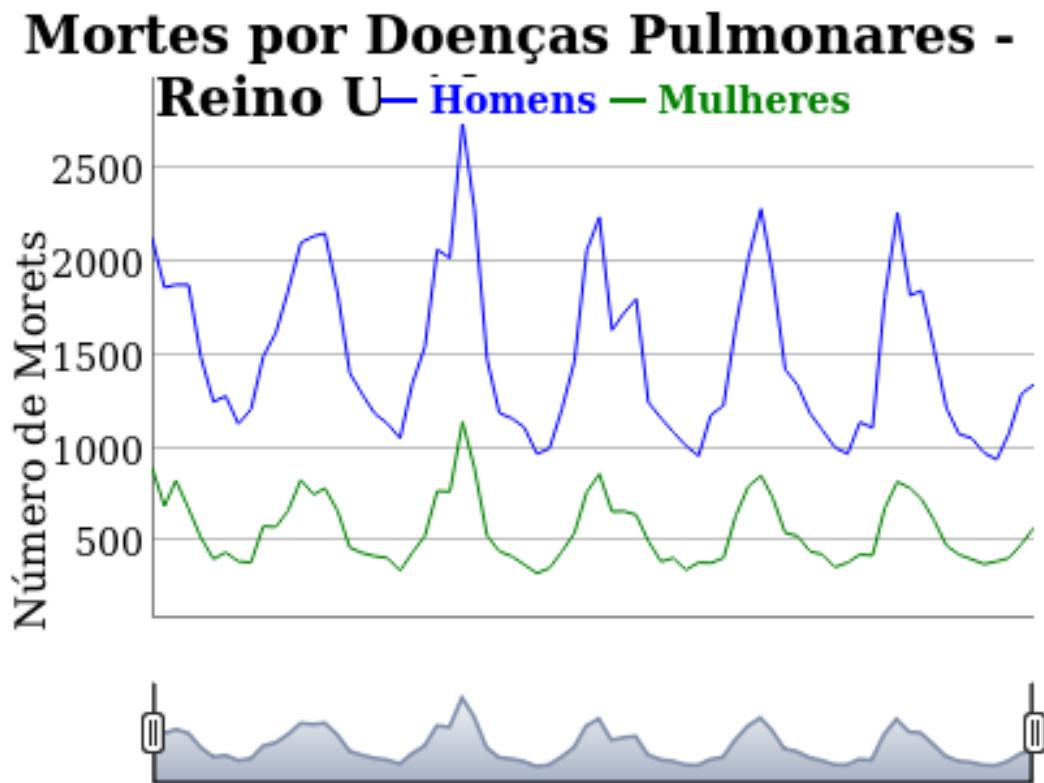
Outra maneira de declarar um objeto de série de tempo é usando a função `xts()` do pacote de mesmo nome. No entanto, para essa função precisamos de um vetor ordenado de datas do tipo `Date`, `POSIXct`, `timeDate`, `yearmon` e `yearqtr`.

```
library(xts)
xts_df <- data.frame(y = rnorm(365, 100, 10))
xts_df$data <- seq.Date(as.Date("2011-01-01"), length.out = 365,
 by = "1 day")
xts_df <- xts(x = xts_df[, "y"], order.by = xts_df[, "data"])

head(xts_df)

[,1]
2011-01-01 93.21508
2011-01-02 91.29554
2011-01-03 99.29232
2011-01-04 99.57950
2011-01-05 111.34360
```

```
2011-01-06 91.37527
library(dygraphs)
lungDeaths <- cbind(mdeaths, fdeaths)
dygraph(lungDeaths,
 main = "Mortes por Doenças Pulmonares - Reino Unido - 1874-1979",
 ylab = "Número de Mortes") %>%
dySeries("mdeaths", color = "blue", label = "Homens") %>%
dySeries("fdeaths", color = "green", label = "Mulheres") %>%
dyRangeSelector()
```



Aqui fica o código para alterar os padrões dos números e datas:

```
Alterar rótulos do eixo x e a legenda
axlabform <- "function(date, granularity, opts, dygraph) {
 var months = ['Janeiro', 'Fevereiro', 'Março', 'Abril', 'Maio',
 'Junho', 'Julho', 'Agosto', 'Setembro', 'Outubro', 'Novembro', 'Dezembro'];
 return months[date.getMonth()] + \" \"+ date.getFullYear()}"
```

```
valueform <- "function(ms) {
 var months = ['Janeiro', 'Fevereiro', 'Março', 'Abril', 'Maio',
 'Junho', 'Julho', 'Agosto', 'Setembro',
 'Outubro', 'Novembro', 'Dezembro'];
```

```

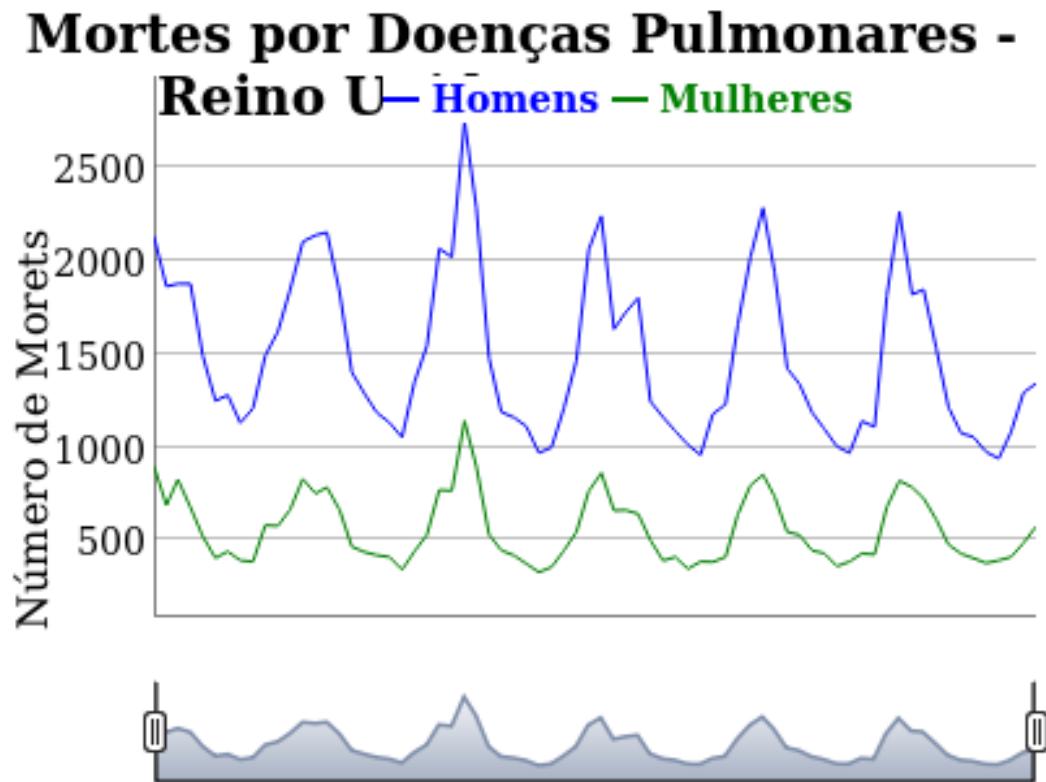
var ms = new Date(ms);
return months[ms.getMonth()] + '/' + ms.getFullYear()}"
```

```

valueformy <- "function(value) {
 return (Math.round(value * 100)/100).toString()
.replace('.', ',')
.replace(/\B(?=(\d{3})+(?!\d))/g, '.')}"
```

```

dygraph(lungDeaths,
 main = "Mortes por Doenças Pulmonares - Reino Unido - 1874-1979",
 ylab = "Número de Mortes") %>%
dySeries("mdeaths", color = "blue", label = "Homens") %>%
dySeries("fdeaths", color = "green", label = "Mulheres") %>%
dyAxis("y", valueFormatter = valueformy) %>%
dyAxis("x", axisLabelFormatter = axlabform, valueFormatter = valueform) %>%
dyRangeSelector()
```



## 10.4 Leaflet

O leaflet é, provavelmente, a principal biblioteca JavaScript para visualizações interativas de mapas. Mais informações sobre o pacote estão disponíveis neste link. Para iniciar uma visualização com leaflet, basta executar o código abaixo. A função `addTiles()` adiciona uma camada de mapas ao leaflet que foi inicializado.

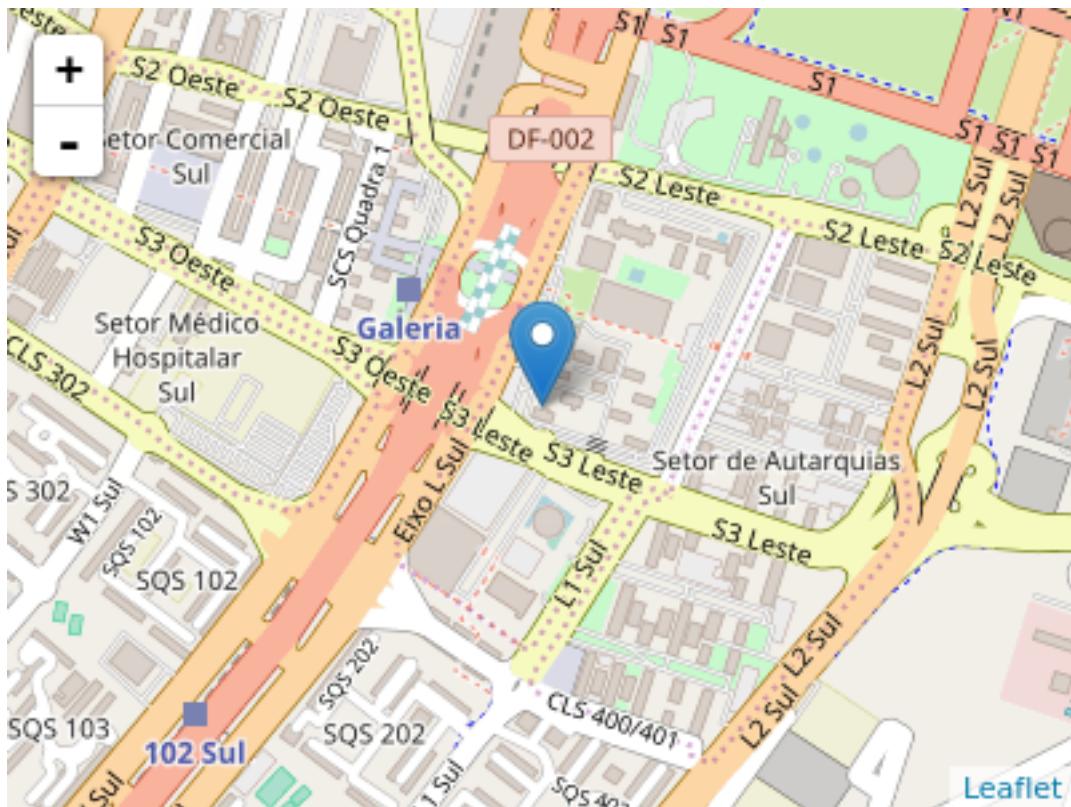
```
library(dplyr)
library(leaflet)
leaflet() %>%
 addTiles()
```



### 10.4.1 Primeiro exemplo

No primeiro exemplo, vamos incluir um marcador na localização do IBPAD. Para isso, vamos obter a latitude e longitude usando a função `geocode()` do pacote `ggmap`. Além disso, foi incluída uma coluna chamada `popup` que receberá um texto que será mostrado no mapa.

```
library(ggmap)
Pegar Localização do ibpad (Google desatualizado)
#loc.ibpad <- geocode("IBPAD")
loc.ibpad <- data.frame(lon = -47.8838813, lat = -15.8010146)
loc.ibpad$popup <- "Estamos aqui! (teoricamente)"
leaflet(loc.ibpad) %>%
 addTiles() %>%
 addMarkers(lat = ~lat, lng = ~lon, popup = ~popup)
```

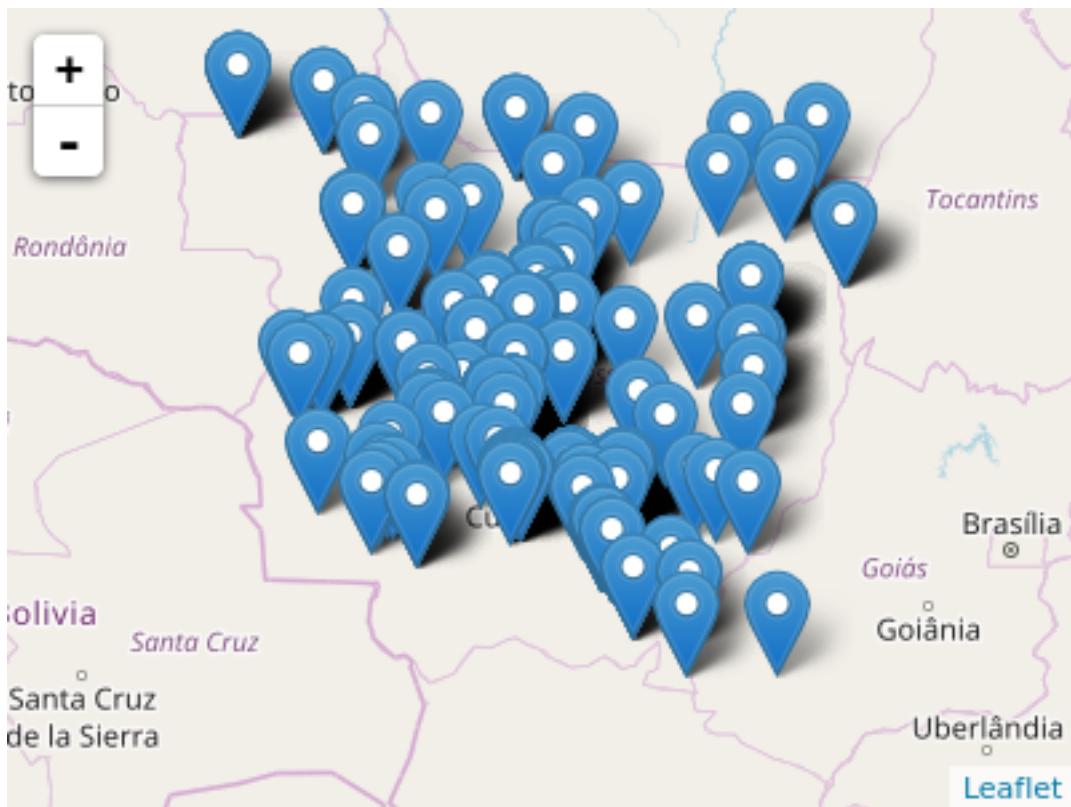


#### 10.4.2 Marcadores

No exemplo abaixo, vamos criar uma visualização com a posição de algumas empresas exportadoras de Mato Grosso a partir de dados disponibilizados pelo MDIC. Como a busca da localização foi feita usando o endereço, nem sempre a localização estará perfeitamente correta. No entanto, para exemplificar o uso do pacote, não há problemas.

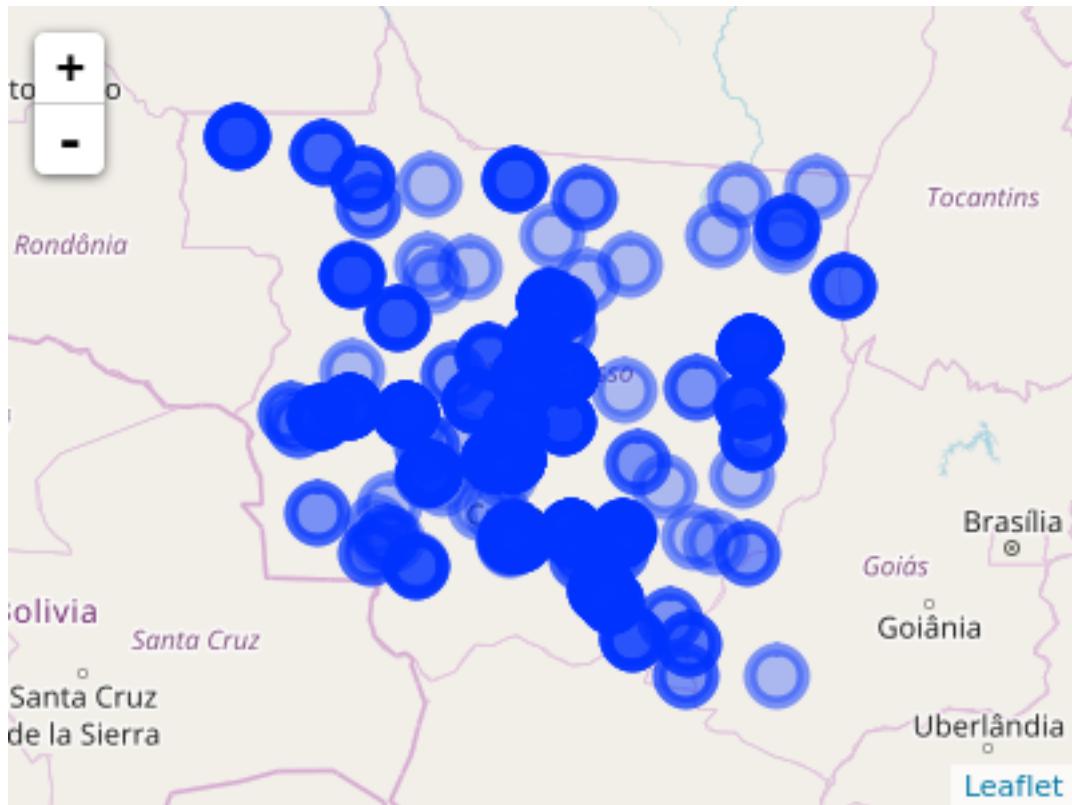
```
dados.empresas.mt <- read_delim('dados/empresas_exp_mt.csv',
 delim = ";",
 locale = locale(encoding = 'ISO-8859-1',
 decimal_mark = ","))

leaflet(dados.empresas.mt) %>%
 addTiles() %>%
 addMarkers(lat = ~lat, lng = ~lon, popup = ~EMPRESA)
```



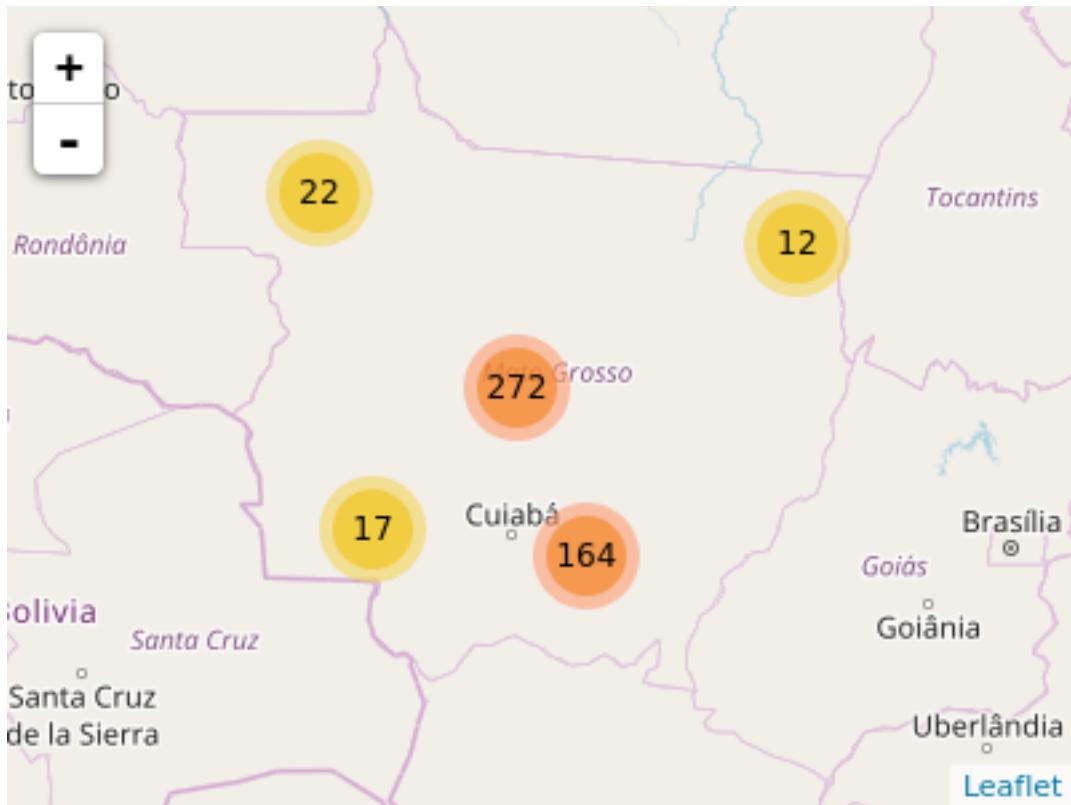
Podemos também adicionar outros tipos de marcadores, como círculos:

```
leaflet(dados.empresas.mt) %>%
 addTiles() %>%
 addCircleMarkers(lat = ~lat, lng = ~lon, popup = ~EMPRESA, fillOpacity = 0.3)
```



Adicionalmente, é possível agrupar pontos próximos em clusters.

```
leaflet(dados.empresas.mt) %>%
 addTiles() %>%
 addCircleMarkers(lat = ~lat, lng = ~lon, popup = ~EMPRESA, fillOpacity = 0.3,
 clusterOptions = markerClusterOptions())
```



#### 10.4.3 Polígonos

Também é possível criar polígonos a partir de shapefiles.

```
library(rgdal)
library(maptools)
library(rgeos)
library(ggplot2)

ogrListLayers('dados/mapas/mg_municípios/31MUE250GC_SIR_simplificado.shp')

mg_mapa <- readOGR('dados/mapas/mg_municípios/31MUE250GC_SIR_simplificado.shp',
 layer = '31MUE250GC_SIR_simplificado')

mg_mapa$Mun.Trab <- substr(mg_mapa$CD_GEOCMU, 1, 6)

REM_RAIS_MG_2015 <- read_delim('dados/REM_RAIS_MG_2015.csv',
 delim = ";",
 locale = locale(encoding = "ISO-8859-1"),
 col_types = 'cd')

colnames(REM_RAIS_MG_2015)[1] <- "Mun.Trab"
summary(REM_RAIS_MG_2015$mediana)
REM_RAIS_MG_2015 <- REM_RAIS_MG_2015 %>%
 mutate(mediana.original = mediana,
```

```

 mediana = ifelse(mediana > 1500, 1500, mediana))
head(REM_RAIS_MG_2015)

mg_mapa@data <- left_join(mg_mapa@data, REM_RAIS_MG_2015, by = "Mun.Trab")

mg_mapa$POPUP <- paste0(mg_mapa$NM_MUNICIP, ": R$ ",
 format(round(mg_mapa$mediana.original, 2),
 big.mark = ".", decimal.mark = ","))

viridis.colors<- viridis::viridis(n = 20)

pal <- colorNumeric(viridis.colors, domain = mg_mapa$mediana)

leaflet(mg_mapa) %>%
 addTiles() %>%
 addPolygons(weight = 0.8,
 fillColor = ~pal(mediana), fillOpacity = 0.9,
 popup = ~POPUP)

```

```

[1] "31MUE250GC_SIR_simplificado"
attr(),"driver")
[1] "ESRI Shapefile"
attr(),"nlayers")
[1] 1

```

```

OGR data source with driver: ESRI Shapefile
Source: "dados/mapas/mg_municípios/31MUE250GC_SIR_simplificado.shp", layer: "31MUE250GC_SIR_simplifi"
with 853 features
It has 2 fields

```

```

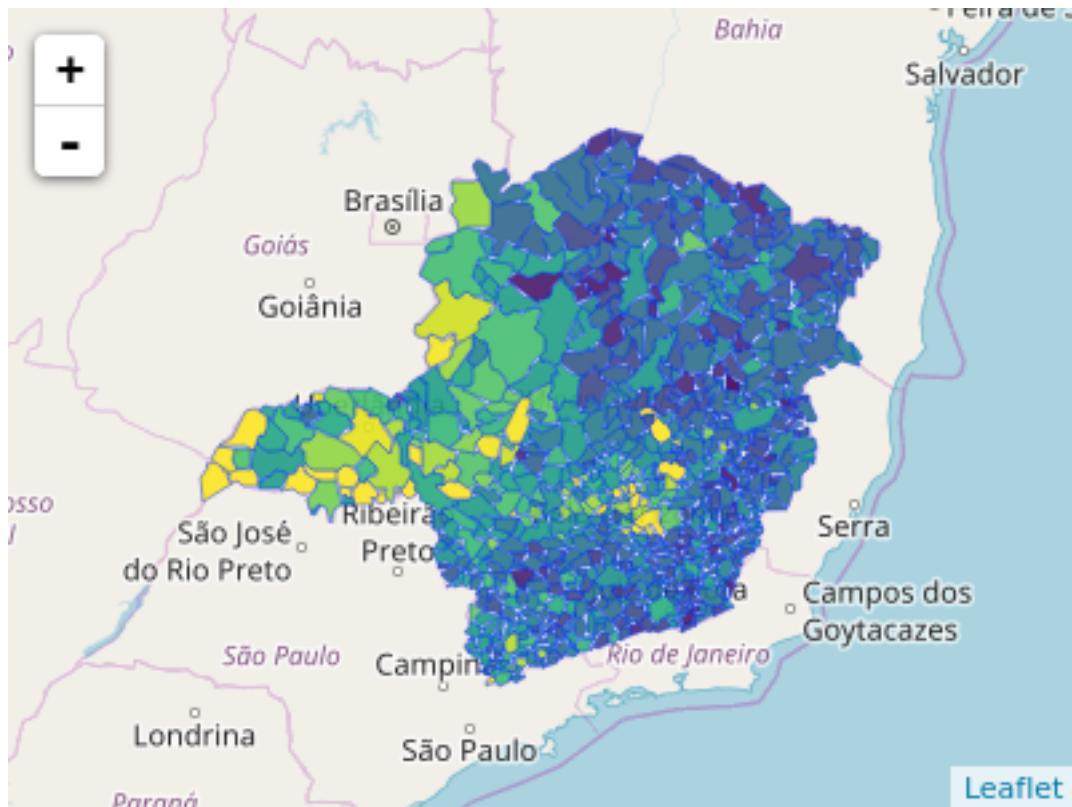
Min. 1st Qu. Median Mean 3rd Qu. Max.
788 1007 1086 1124 1182 3675

```

```

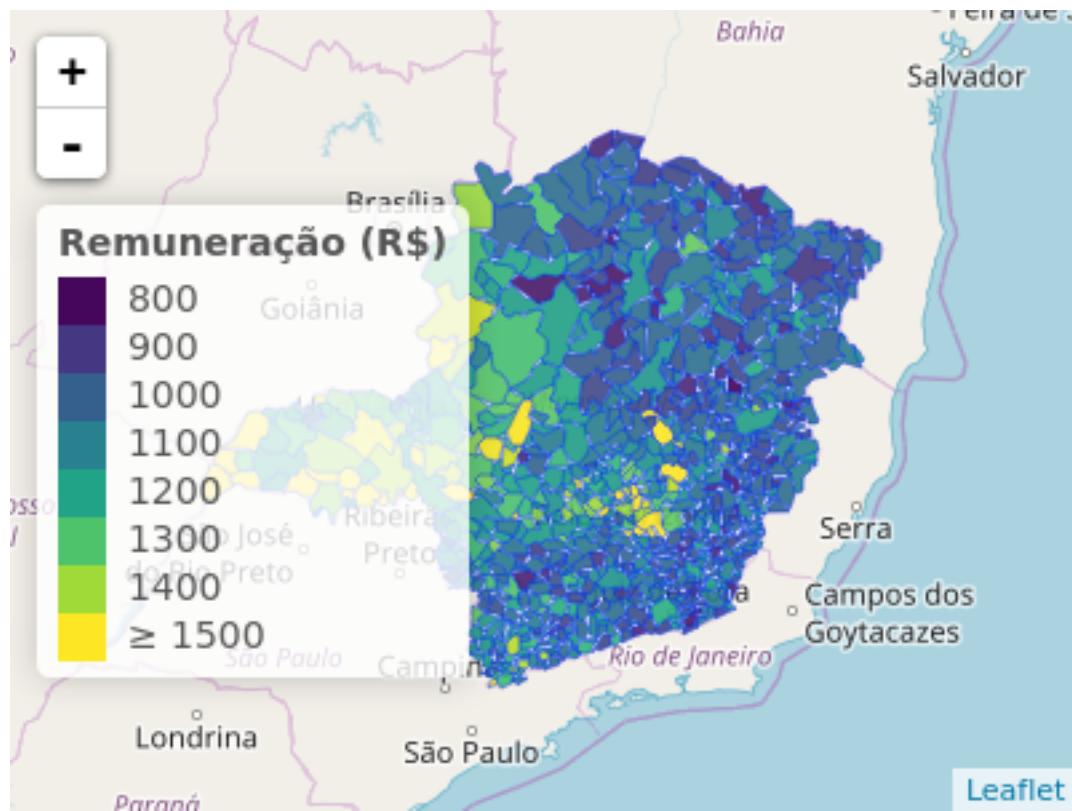
A tibble: 6 x 3
Mun.Trab mediana mediana.original
<chr> <dbl> <dbl>
1 315820 788.0 788.0
2 313865 827.4 827.4
3 311880 831.0 831.0
4 313980 831.0 831.0
5 312290 831.0 831.0
6 315760 831.0 831.0

```



A função `colorNumeric()` criou uma função que é usada para definir uma cor para cada valor usado como input, no caso `mediana`. Adicionalmente, pode-se uma legenda usando a função `addLegend()`.

```
leaflet(mg_mapa) %>%
 addTiles(urlTemplate = 'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png') %>%
 addPolygons(weight = 0.8,
 fillColor = ~pal(mediana), fillOpacity = 0.9,
 popup = ~POPUP) %>%
 addLegend("topleft", title = "Remuneração (R$)",
 colors = pal(seq(800, 1500, 100)),
 values = seq(800, 1500, 100),
 labels = c(seq(800, 1400, 100), "\u2265 1500"),
 opacity = 1)
```



## 10.5 Exercícios

1. Usando os pacotes `ggmap` e `leaflet`, crie uma visualização marcando 5 localizações de Brasília.
2. Utilizando a base `gapminder` (`library(gapminder)`), crie uma visualização usando o pacote `plotly`.
3. A partir da base `economics` do pacote `ggplot2`, escolha uma variável e plote a série histórica utilizando o pacote `dygraphs`.
4. Entre na galeria de `htmlwidgets` [neste link], escolha alguma `htmlwidget` que você tenha achado interessante e replique um exemplo.



# Capítulo 11

## RMarkdown

Antes de falar sobre o que é RMarkdown, é interessante discutir sobre o Markdown.

Markdown é uma linguagem de marcação, ou seja, não é uma linguagem de programação. Linguagens de marcação dizem como algo deve ser entendido, mas não têm capacidade de processamento e execução de funções. Por exemplo, HTML é uma linguagem de marcação. Ela apenas diz como uma página web está estruturada, mas não executa nenhum processamento. O Markdown, da mesma forma, apenas informa como um documento está estruturado.

No entanto, a vantagem do Markdown é a sua simplicidade e a possibilidade de utilização de uma linguagem comum para criação de vários tipos de documentos. Por exemplo, um mesmo código Markdown pode ser convertido para HTML, LaTeX (gera pdf's), docx etc. Para isso, é necessário de um conversor, que lê um código em Markdown e, considerando a escolha do *output* desejado, converte o arquivo para a linguagem desejada. Isso ficará mais claro com os exemplos.

E o que é o RMarkdown? Nada mais é do que a possibilidade de executar scripts em R (além de outras linguagens) e incorporá-lo ao um arquivo Markdown (extensão .md). O pacote `knitr` irá executar “pedaços” (*chunk*) de códigos e gerará um arquivo .md com os códigos e os seus resultados. Na sequência, o `pandoc` que é um conversor, converte para a linguagem desejada gerando os arquivos nos formatos escolhidos (.html, .docx, .pdf, .odt). A figura abaixo ilustra o processo:

Entre neste link para ver os tipos de formatos disponíveis no RMarkdown. É possível gerar desde documentos no formato Word, pdf formatados para revistas científicas, apresentações, dashboards etc.

### 11.1 Usos do RMarkdown

A seção anterior já deu algumas dicas sobre a utilidade do RMarkdown. Aqui vamos elaborar um pouco mais.



Figura 11.1: Processo - RMarkdown

1. **Reprodutibilidade.** Isso é importante quando um estudo é realizado. Pode ser que em algum momento após a realização do seu estudo, outro analista/pesquisador deseje replicá-lo. Um documento que une o código às explicações pode ser fundamental nesse momento.
2. **Compartilhamento de informação.** Pode ser que você tenha aprendido a usar um novo pacote e ache que ele pode ser interessante para outros colegas. Com o RMarkdown, você poderá criar um documento com exemplos de uso do pacote que pode ser facilmente compartilhado.
3. **Documentação de Rotinas.** A criação de rotinas para realização de tarefas repetitivas é cada vez mais comum. No entanto, é importante que haja uma boa documentação da rotina para que você ou outro colega possa entender e dar manutenção à rotina no futuro.
4. **Relatórios parametrizados.** O RMarkdown facilita a criação de relatórios, inclusive dashboards, em que a estrutura é padrão, mas os dados dependem de um parâmetro. Por exemplo, relatórios de balança comercial por Unidade da Federação. No caso, a UF seria um parâmetro que variaria de relatório para relatório.

## 11.2 Estrutura de um RMarkdown

```

title: "Primeiro Exemplo para o Curso de R"
author: "Paulo"
date: "`r format(Sys.time(), '%d de %B de %Y')`"
output:
 html_document: default
 pdf_document:
 fig_caption: yes
 fig_height: 3.5
 fig_width: 7
 number_sections: yes
lang: pt-br

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
options(OutDec = ",")
```

Cabeçalho de primeiro nível
Cabeçalho de segundo nível
Cabeçalho de terceiro nível

Hello World
```

Este é um primeiro exemplo de \*RMarkdown\* para o \*\*Curso de Introdução ao `R`\*\*.

## Outra Seção

Vamos executar um código:

```
```{r, fig.cap="Exemplo de Figura", collapse=TRUE}
library(ggplot2)
x <- rnorm(100)
y <- rnorm(100)
```

```

```{r}
library(ggplot2)
x <- rnorm(100)
y <- rnorm(100)
dados <- data.frame(x, y)
ggplot(dados, aes(x = x, y = y)) +
 geom_point()
```

```

Figura 11.2:

```

dados <- data.frame(x, y)
ggplot(dados, aes(x = x, y = y)) +
  geom_point()
```

```

A média de x é `r mean(x)`.

Vamos agora entender qual é a função de cada parte desse código.

#### YAML (Configurações):

```

title: "Primeiro Exemplo para o Curso de R"
author: "Paulo"
date: "12 de abril de 2018"
output:
 html_document: default
 pdf_document:
 fig_caption: yes
 fig_height: 3.5
 fig_width: 7
 number_sections: yes
lang: pt-br

```

O **YAML** é o responsável pelas configurações dos documentos. Basicamente, estamos informando qual é o título do documento, a data de criação, o nome do autor e qual é o tipo de output que desejamos. No exemplo, está definido como output **pdf\_document** e foram adicionadas opções para que as seções fossem numeradas e de tamanho das figuras. Caso quiséssemos um arquivo no formato Word (.docx), o output poderia ser modificado para **word\_document**. Cada formato, possui um conjunto de opções disponíveis. Nesse caso, é importante olhar a página de cada formato disponível na documentação do RMarkdown.

#### Code Chunks:

Os *Code Chunks* são pedaços de código em R que podem ser executados para gerar resultados que serão incorporados ao documento. Você pode inserir um *chunk* manualmente ou com o atalho CTRL + ALT + I. Dentro de {} é possível incluir uma série de opções relacionadas à execução do código. Abaixo, iremos falar sobre algumas dessas opções.

#### Textos e Markdown:

```
Hello World

Este é um primeiro exemplo de _RMarkdown_ para o *Curso de Introdução ao `R`*.

Outra Seção
```

Figura 11.3:

Na figura acima, é mostrado como o texto se mistura com o código de markdown. Esse código markdown, quando convertido, irá gerar a formatação desejada. Na próxima seção, iremos detalhar o que cada marcação faz.

### 11.3 Renderizando um documento

Há duas formas de renderizar um documento .Rmd. A primeira é via função `render()` do pacote `rmarkdown`.

```
render(input, output_format = NULL, output_file = NULL, output_dir = NULL,
 output_options = NULL, intermediates_dir = NULL,
 runtime = c("auto", "static", "shiny"),
 clean = TRUE, params = NULL, knit_meta = NULL, envir = parent.frame(),
 run_pandoc = TRUE, quiet = FALSE, encoding = getOption("encoding"))
```

Veja no help a função de cada argumento. Essa função é especialmente importante quando a renderização de um RMarkdown está inserida dentro de uma rotina. Por exemplo, pode-se usar a função `render()` dentro de um loop para criar vários pdf's a partir de um relatório parametrizado.

A outra opção é usando o botão `knit` disponível na interface do RStudio.

### 11.4 Sintaxe

Abaixo estão os principais elementos de sintaxe do RMarkdown. Vários são autoexplicativos. Para alguns, iremos fazer alguns comentários.

- **Cabeçalhos:**

```
Cabeçalho de primeiro nível
Cabeçalho de segundo nível
Cabeçalho de terceiro nível
```

O resultado em um documento com `output` definido como `pdf_document` seria:

- **Estilo de texto:**

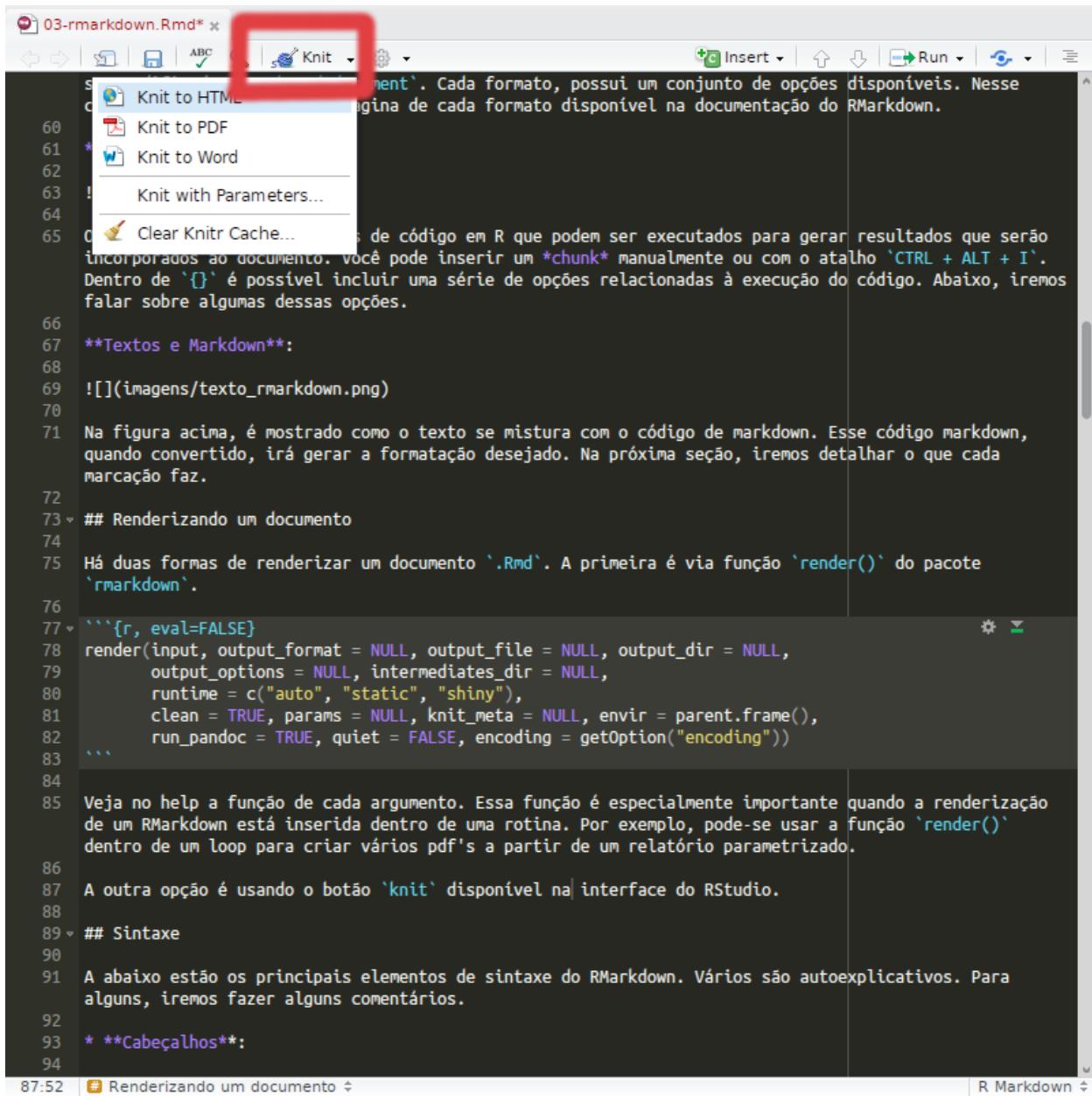
**\*Itálico\*** e **\*\*Negrito\*\***

**Itálico e Negrito**

- **Citações:**

> Aqui vai um texto para citação

Aqui vai um texto para citação



A screenshot of the RStudio interface showing an R Markdown file named '03-rmarkdown.Rmd'. The code editor displays several lines of R Markdown syntax. A red box highlights the 'Knit' button in the toolbar, which is part of a dropdown menu. The dropdown menu is open, showing options: 'Knit to HTML', 'Knit to PDF', 'Knit to Word', and 'Knit with Parameters...'. Below the menu, the R Markdown code continues with sections like '\*\*Textos e Markdown\*\*', '## Renderizando um documento', and '### Sintaxe'.

```

60 # knit to HTML
61 # knit to PDF
62 # knit to Word
63 # knit with parameters
64 # clear knitr cache
65
66 **Textos e Markdown**:
67
68
69
70 Na figura acima, é mostrado como o texto se mistura com o código de markdown. Esse código markdown,
71 quando convertido, irá gerar a formatação desejada. Na próxima seção, iremos detalhar o que cada
72 marcação faz.
73 ## Renderizando um documento
74
75 Há duas formas de renderizar um documento '.Rmd'. A primeira é via função `render()` do pacote
76 `rmarkdown`.
77 ```{r, eval=FALSE}
78 render(input, output_format = NULL, output_file = NULL, output_dir = NULL,
79 output_options = NULL, intermediates_dir = NULL,
80 runtime = c("auto", "static", "shiny"),
81 clean = TRUE, params = NULL, knit_meta = NULL, envir = parent.frame(),
82 run_pandoc = TRUE, quiet = FALSE, encoding = getOption("encoding"))
83 ...
84
85 Veja no help a função de cada argumento. Essa função é especialmente importante quando a renderização
86 de um RMarkdown está inserida dentro de uma rotina. Por exemplo, pode-se usar a função `render()`
87 dentro de um loop para criar vários pdf's a partir de um relatório parametrizado.
88
89 A outra opção é usando o botão `knit` disponível na interface do RStudio.
90
91 ## Sintaxe
92
93 * **Cabeçalhos**:
94

```

Figura 11.4: Botão Knit

# 1 Cabeçalho de primeiro nível

## 1.1 Cabeçalho de segundo nível

### 1.1.1 Cabeçalho de terceiro nível

Figura 11.5:

- Código no texto:

```
`mean(x)`
```

```
mean(x)
```

- Código processado no texto:

A opção abaixo é importante para que resultados do R possam ser incorporados diretamente ao texto do documento. Para demonstração aqui, foi necessário dar um espaço entre a aspa e o código, mas o correto é não haver esse espaço.

```
` r mean(c(2, 3, 4)) `
```

```
3
```

- Imagens:

```
! [] (images/code_chunk.png)
```

```
! [Título Opcional] (images/code_chunk.png)
```

- Listas não ordenadas:

- \* Item a
- \* Item b
  - + Subitem b1
  - + Subitem b2

- Item a
- Item b
  - Subitem b1
  - Subitem b2

- Listas ordenadas:

1. Item 1
2. Item 2
3. Item 3
  - i. Item 3a
  - ii. Item 3b
1. Item 1
2. Item 2
3. Item 3
  - i. Item 3a
  - ii. Item 3b

- Tabelas:

-----: indica que a Coluna 1 está alinhada à direita. :-----: indica que a coluna está centralizada. Alguns pacotes do R fornecem funções para geração de tabelas a partir de `data.frames` e matrizes. Veja

```
Coluna 1 | Coluna 2
-----: | :-----:
10 | Brasil
20 | China
Fonte: MDIC.
```

Coluna 1	Coluna 2
10	Brasil
20	China

Fonte: MDIC.

Column 1	Column 2
I want the contents of this cell to fit into one line	Word1 Word2
Column 1	Column 2
I want the contents of this cell to fit into one line	Word1 Word2

```
x <- letters[1:3]
y <- LETTERS[1:3]
knitr::kable(data.frame(x, y), align = 'cc')
```

x	y
a	A
b	B
c	C

- **Links:**

Site do [MDIC] ([www.mdic.gov.br](http://www.mdic.gov.br))

Site do MDIC

- Linha horizontal ou Quebra de Página:

\*\*\*

---



- **Equação:**

Modelo linear simples:  $\$y_i = \alpha + \beta x_i + e_i\$$

Modelo linear simples:  $y_i = \alpha + \beta x_i + e_i$

- **Equação em Bloco:**

Modelo linear simples:  $\$\$y_i = \alpha + \beta x_i + e_i\$\$$

Modelo linear simples:

$$y_i = \alpha + \beta x_i + e_i$$

## 11.5 Opções de Chunk

Os chunks de códigos (`{r, ...}`) possuem uma série de opções. Vamos elencar aqui as principais. Para a lista completa, veja esse link.

Opção	Valor Padrão	Descrição
eval	TRUE	Indica se o código deve ser executado
include	TRUE	Indica se o código deve ser exibido no documento final. Os resultados não serão apresentados.

Opção	Valor Padrão	Descrição
collapse	FALSE	Indica se o código e os resultados do chunk devem ser colapsados em um bloco único
echo	FALSE	Indica se o código será exibido no documento final. Os resultados serão apresentados.
results	markup	Se <code>hide</code> , os resultados não serão exibidos. Se <code>hold</code> , os resultados serão exibidos ao final do chunk. Se <code>asis</code> , os resultados não serão formatados, sendo mostrados os resultados “brutos” (código html, tex, ...).
error	TRUE	Indica se mensagens de erros serão exibidas.
message	TRUE	Indica se mensagens geradas pelo código serão exibidas.
warning	TRUE	Indica se avisos gerados pelo código serão exibidos.
fig.cap	NULL	Título da gráfico referente ao chunk.
fig.height	7	Altura para gráficos criados pelo código (em polegadas).
fig.width	7	Largura para gráficos criados pelo código (em polegadas).

## 11.6 Principais Formatos

### 11.6.1 HTML

A HTML é a linguagem de marcação para construção de páginas web. Assim, se criarmos um documento e escolhermos como opção de output `html_document`, o resultado será uma página a ser aberta em browsers. Outros formatos do markdown, como `flexdashboard` e `ioslides`, também geram páginas html. Cada tipo de formato tem um conjunto de aspectos específicos. Abaixo listamos os principais para html's:

- A aparência e o estilo são definidas por um arquivo no formato `css`. Isso impõe uma dificuldade adicional para formatação do documento. O Rmarkdown fornece alguns temas e pacotes também podem fornecer documentos com algum formatação de estilo (ver `prettydoc`);
- Único formato que aceita htmlwidgets (o próprio nome indica isso).

O código markdown incluído no arquivo `.Rmd` é convertido pelo pandoc gerando um documento estruturado com código html. Vejamos abaixo o código em RMarkdown:

Após o processamento, será gerado o seguinte código html:

Esse código é interpretado pelo navegador e gera o seguinte resultado:

### 11.6.2 PDF

Para criação de PDF's pelo RMarkdown, utiliza-se o LaTeX (pronuncia-se: *Lah-tech* or *Lay-tech*) que é um sistema de preparação de documentos muito utilizado pela comunidade científica. Inicialmente, o RMarkdown abstrai para o usuário a necessidade de saber essa linguagem. No entanto, como no HTML, se você quiser avançar na estrutura do documento e nos estilos vai ser necessário aprender essa linguagem, pelo menos o suficiente para resolver o seu problema.

Entre as vantagens do LaTeX estão:

- Numeração automática de seções (e os demais níveis) e de equações;
- Criação automática de legendas com base em arquivos `.bib`;
- Facilidade de referências cruzadas no documento.

```

1 ---
2 title: "r paste0("Exportações de ", params$prod)"
3 author: "MDIC"
4 date: "5 de novembro de 2016"
5 output:
6 html_document:
7 fig_caption: yes
8 fig_height: 3.5
9 fig_width: 10
10 self_contained: false
11 params:
12 prod: Minérios de ferro e seus concentrados
13 ...
14
15 ````{r setup, include=FALSE}
16 knitr::opts_chunk$set(echo = TRUE, warning=FALSE, message=FALSE)
17
18 ...
19 |
20 # Dados
21
22 Os dados desse exemplo foram extraídos das séries históricas disponíveis no site do [MDIC](www.mdic.gov.br).
23 ````{r, echo=FALSE}
24 options(OutDec = ",", scipen = 999)
25 library(readxl)
26 library(dplyr)
27 library(ggplot2)
28 library(ggthemes)
29 library(knitr)
30 library(forecast)
31 library(tidyr)
32 library(DT)
33
34 dados <- read_excel('dados/FAT_PDF_PPT.xlsx', sheet = 1)

```

Figura 11.6: Código RMarkdown para gerar um arquivo html

```

<div class="fluid-row" id="header">

<h1 class="title toc-ignore">Exportações de Minérios de ferro e seus concentrados</h1>
<h4 class="author">MDIC</h4>
<h4 class="date">5 de novembro de 2016</h4>

</div>

<div id="dados" class="section level1">
<h2>Dados</h2>
<p>Os dados desse exemplo foram extraídos das séries históricas disponíveis no site do MDIC. </p>
<p>Amostra dos dados:</p>
<pre class="r"><code>kable(dados %>%
 select(DATA, VL_FOB, KG_LIQUIDO, PRECO) %>%
 mutate(DATA = format(DATA, "%b/%Y")) %>%
 head(10), format.args = list(big.mark = "."))</code></pre>
<table>
<thead>
<tr class="header">
<th align="left">DATA</th>
<th align="right">VL_FOB</th>
<th align="right">KG_LIQUIDO</th>
<th align="right">PRECO</th>
</tr>
</thead>
<tbody>
<tr class="odd">
<td align="left">Jan/1997</td>
<td align="right">266.869.782</td>
<td align="right">12.190.272.878</td>
<td align="right">0,0218920</td>
</tr>
<tr class="even">
<td align="left">Fev/1997</td>
<td align="right">172.468.713</td>
<td align="right">7.845.212.443</td>
<td align="right">0,0219839</td>
</tr>

```

Figura 11.7: Exemplo de código HTML

# Exportações de Minérios de ferro e seus concentrados

MDIC

5 de novembro de 2016

## Dados

Os dados desse exemplo foram extraídos das séries históricas disponíveis no site do [MDIC](#).

Amostra dos dados:

```
kable(dados %>%
 select(DATA, VL_FOB, KG_LIQUIDO, PRECO) %>%
 mutate(DATA = format.Date(DATA, "%b/%Y")) %>%
 head(10), format.args = list(big.mark = "."))
```

DATA	VL_FOB	KG_LIQUIDO	PRECO
Jan/1997	266.869.782	12.190.272.878	0,0218920
Fev/1997	172.468.713	7.845.212.443	0,0219839
Mar/1997	265.033.886	13.041.094.690	0,0203230
Abr/1997	229.954.303	10.368.844.046	0,0221774
Mai/1997	222.365.501	10.515.503.370	0,0211464
Jun/1997	281.409.102	13.278.045.526	0,0211936
Jul/1997	261.705.472	12.452.278.986	0,0210167
Ago/1997	207.029.373	9.810.599.506	0,0211026

Figura 11.8: Exemplo de formato HTML

Como no caso do HTML, é possível usar templates. Isso é bastante útil para criação de artigos científicos, que devem ser padronizados. Além disso, relatórios de instituições podem ser padronizados. Assim, basta que alguém crie um template e os demais poderão criar documentos com a mesma estrutura usando apenas o RMarkdown.

LaTeX é uma versão mais amigável de TeX. Ou seja, LaTeX é uma linguagem em um nível maior do que TeX. Como no R, existem diversos pacotes em LaTeX que fornecem comandos para facilitar a edição de alguma parte do documento. Por exemplo, o pacote `fancyhdr` fornece comandos que facilitam a construção de cabeçalhos e rodapé. Abaixo, está um pequeno código para se ter uma pequena noção sobre essa linguagem:

```
\documentclass{article}
\title{Exemplo 1}
\author{Nome do Autor}
\date{\today}
\begin{document}
 \maketitle
 \newpage
 \section{Introdução}
 Aqui vai o texto!
\end{document}
```

Basicamente, o código acima define a classe do documento como artigo, o título, o nome do autor, a data para o dia em que o documento for compilado. Depois, inicia-se o documento, criando o título (inclui título, autor e data), definindo uma quebra de página e a seção introdução.

### 11.6.2.1 Instalações necessárias

Para criar documentos PDF no RMarkdown é preciso ter uma instalação **TeX** disponível. Para isso, é preciso baixar uma distribuição compatível com o seu sistema operacional. Nessa página estão listadas as distribuições disponíveis por sistema operacional. No windows, é comum usar a distribuição MiKTeX.

### 11.6.2.2 Exemplo

Como no HTML, vamos mostrar primeiramente o código em RMarkdown:

Esse código irá gerar um arquivo intermediário com a extensão `.tex`. Esse arquivo terá o seguinte código:

Após a compilação, o seguinte documento é gerado:

### 11.6.3 Word

A geração de word segue a geração dos demais formatos. No entanto, templates apenas funcionam para definir estilos que serão usados no documento. Atualmente, devido a limitações do conversor (pandoc) é difícil ter total acesso à formatação do documento.

## 11.7 Exercícios

1. Crie um RMarkdown com o formato HTML para output. Neste documento, faça um mini tutorial sobre algum pacote que você aprendeu no curso ou outro pacote que você tenha conhecido. Crie as seções adequadamente e, se for o caso, crie tabelas.
2. Crie um novo RMarkdown e explique o processo de criação de um gráfico usando o ggplot2. O ideal é criar um novo gráfico, mas fique livre para utilizar algum exemplo desse material.

```
1^- ---
2 title: ``r paste0('Exportações de ', params$prod)`"
3 author: "Paulo Alencar"
4 date: "5 de novembro de 2016"
5 lang: "pt-br"
6 output:
7 pdf_document:
8 fig_caption: yes
9 fig_height: 3.5
10 fig_width: 10
11 number_sections: yes
12 keep_tex: yes
13 includes:
14 in_header: preamble.tex
15 params:
16 prod: Soja mesmo triturada
17 ---
18 ```{r setup, include=FALSE}
19 knitr::opts_chunk$set(echo = FALSE, warning=FALSE, message=FALSE)
20 ````
21
22
23 # Dados
24
25 Os dados desse exemplo foram extraídos das séries históricas disponíveis no site do [MDIC](www.mdic.gov.br).
26 ```{r, echo=FALSE}
27 options(OutDec = ",", scipen = 999, xtable.comment = FALSE)
28 library(readxl)
29 library(dplyr)
30 library(ggplot2)
31 library(ggthemes)
32 library(forecast)
33 library(tidyr)
34 library(xtable)
```

Figura 11.9: Código RMarkdown para gerar um arquivo html

```

100 \makeatletter
101 \setlength{\@fptop}{0pt}
102 \makeatother
103
104 \newcommand{\source}[1]{
105 \begin{flushleft}
106 \scriptsize{Fonte: #1}
107 \end{flushleft}
108 }
109
110 \begin{document}
111 \maketitle
112
113 \section{Dados}\label{dados}
114
115 Os dados desse exemplo foram extraídos das séries históricas disponíveis
116 no site do \href{www.mdic.gov.br}{MDIC}.
117
118 Amostra dos dados:
119
120 \begin{table}[ht]
121 \centering
122 \caption{Amostra de dados}
123 \begin{tabular}{crrr}
124 \hline
125 DATA & VL_FOB & KG_LÍQUIDO & PRECO \\
126 \hline
127 Jan/1997 & 4.118.740 & 14.030.000 & 0,29 \\
128 Feb/1997 & 4.896.148 & 17.095.000 & 0,29 \\
129 Mar/1997 & 156.260.973 & 550.157.554 & 0,28 \\
130 Abr/1997 & 462.887.400 & 1.598.886.449 & 0,29 \\
131 Mai/1997 & 504.098.587 & 1.697.312.680 & 0,30 \\
132 Jun/1997 & 395.752.981 & 1.345.808.797 & 0,29 \\
133 Jul/1997 & 494.893.812 & 1.670.251.481 & 0,30 \\
134 Ago/1997 & 291.057.057 & 986.597.810 & 0,30 \\
135 Set/1997 & 104.112.390 & 348.746.010 & 0,30 \\
136 Out/1997 & 32.397.210 & 105.540.943 & 0,31 \\
137 \hline \multicolumn{4}{l}{\scriptsize{Fonte: MDIC}} \\ \end{tabular}
138 \end{table}
139
140 A figura \ref{fig:fig1} apresenta a evolução do valor exportado de Soja
141 mesmo triturada. O recorde de valor aconteceu em maio de 2013, sendo

```

Figura 11.10: Código tex gerado

3. Utilizando o pacote WDI crie uma visualização com o pacote dygraphs para a série de renda per capita do Brasil (BRA). Coloque um título no seu documento e faça com que o código não seja apresentado ao leitor.
4. A partir do exemplo anterior, crie um parâmetro chamado `country_code` no cabeçalho de configuração (`yaml`). Esse parâmetro será usado para selecionar o país base para a visualização.
5. Crie um vetor com cinco códigos de países. A partir desse vetor, use a função `render` para criar um arquivo para cada país.

# Exportações de Soja mesmo triturada

*Paulo Alencar*

*5 de novembro de 2016*

## 1 Dados

Os dados desse exemplo foram extraídos das séries históricas disponíveis no site do MDIC.

Amostra dos dados:

Tabela 1: Amostra de dados

DATA	VL_FOB	KG_LIQUIDO	PRECO
Jan/1997	4.118.740	14.030.000	0,29
Fev/1997	4.896.148	17.095.000	0,29
Mar/1997	156.200.973	550.157.554	0,28
Abr/1997	462.887.400	1.598.886.449	0,29
Mai/1997	504.098.587	1.697.312.680	0,30
Jun/1997	395.752.981	1.345.808.797	0,29
Jul/1997	494.893.812	1.670.251.481	0,30
Ago/1997	291.057.057	986.597.810	0,30
Set/1997	104.112.390	348.746.010	0,30
Out/1997	32.397.210	105.540.943	0,31

Fonte: MDIC

Figura 11.11: Exemplo de output pdf

```

title: "r paste0("Exportações de ", params$prod)"
author: "MDIC"
date: "5 de novembro de 2016"
output:
 word_document: default
params:
 prod: Minérios de ferro e seus concentrados

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE, warning=FALSE, message=FALSE)
```

Dados

Os dados desse exemplo foram extraídos das séries históricas disponíveis no site do [MDIC](www.mdic.gov.br).
```{r, echo=FALSE}
options(OutDec = ",", scipen = 999)
library(readxl)
library(dplyr)
library(ggplot2)
library(ggthemes)
library(knitr)
library(forecast)
library(tidyr)
library(DT)

dados <- read_excel('../dados/FAT_PPE_PPI.xlsx', sheet = 1)
```

Figura 11.12: Código RMarkdown para gerar um arquivo word

Capítulo 12

Modelos

O objetivo desse capítulo é dar uma visão geral sobre a estrutura de modelos no R. Isto é, quais são as funções básicas, como especificar um modelo, recuperar resíduos, realizar previsões, etc. Esse processo é parte fundamental de análises mais aprofundadas. Os modelos podem ser usados, de maneira não exclusiva, para exploração de dados, gerar previsões e análise de causalidade. Por exemplo:

- Descritivo: relação entre salários, idade, experiência e anos de estudo;
- Previsão: modelo para identificar risco de fraude em uma transação bancária, classificação de imagens, previsão do PIB para o ano que vem;
- Causalidade: aumento de imposto sobre cigarro e redução no consumo.

12.1 Modelo Linear

Vamos introduzir a estrutura de modelos no R a partir de modelos lineares. Trataremos do modelo linear para regressão e o modelo de regressão logística para classificação. O modelo de regressão é utilizado quando a variável de interesse (dependente ou target) é uma variável quantitativa contínua. Por exemplo, salários, preços, nota em uma exame etc. Por outro lado, modelos de classificação são utilizados quando a variável de interesse é categórica. Por exemplo, uma pessoa tem ou não tem a doença X, o cliente pagou ou não o cartão de crédito, o usuário X é um robô ou uma pessoa.

12.1.1 Regressão

Vamos começar com o modelo linear de regressão:

$$y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \dots + \beta_k x_{ki} + \epsilon_i, \quad i = 1, \dots, N,$$

onde y é a variável dependente, x_k é a k-ésima variável explicativa, β_k é o parâmetro estimado para k-ésima variável e ϵ é o termo de erro.

A função `lm()` estima esse modelo pelo método denominado de mínimos quadrados ordinários (MQO). Antes de exemplificar o uso da função, vamos falar sobre a representação simbólica do modelo, ou seja, como especificar o modelo no R. Em geral, o modelo terá argumentos `x` e `y` em que o usuário passa os dados nesses argumentos, ou terá a estrutura de fórmula. Por ser menos o método usado no modelo linear, detalharemos a estrutura de fórmula. Na função `lm()` é obrigatório passar um objeto da classe `fórmula` ou algum objeto que possa ser convertido para uma fórmula. Por exemplo, para o modelo linear com duas variáveis (y e x) e uma constante, a fórmula correspondente é:

```
f <- 'y ~ x'
class(f)

## [1] "character"
class(as.formula(f))

## [1] "formula"
```

Para mostrar as possibilidades de uso da fórmula de especificação do modelo, vamos utilizar a base `mtcars`. Essa base traz o consumo de gasolina (`mpg`) e algumas características do veículo. Iremos detalhar cada variável explicativa conforme ela é usada. No entanto, você pode olhar o help dessa base: `?mtcars`. Para iniciar, iremos utilizar a variável `mpg` (miles per gallon) e a variável `hp` (Gross horsepower).

```
data(mtcars)
lm(mpg ~ hp, data = mtcars)
```

```
##
## Call:
## lm(formula = mpg ~ hp, data = mtcars)
##
## Coefficients:
## (Intercept)          hp
##   30.09886     -0.06823
```

Note que não houve especificação de uma constante. Automaticamente o R inclui a constante. Você pode incluí-la explicitamente ou retirá-la:

```
lm(mpg ~ hp + 1, data = mtcars)
lm(mpg ~ hp + 0, data = mtcars)
```

Já temos uma pista de como incluir mais variáveis: basta “adicioná-las” com o símbolo `+`. Isto é, vamos incluir a variável `am` (Transmission (0 = automatic, 1 = manual)) no modelo:

```
lm(mpg ~ hp + am, data = mtcars)
```

Se quiséssemos incluir todas as variáveis explicativas:

```
lm(mpg ~ ., data = mtcars)
```

Interações:

```
lm(mpg ~ hp + am + hp:am, data = mtcars)
```

Transformações:

```
lm(log(mpg) ~ log(hp) + am, data = mtcars)
```

```
##
## Call:
## lm(formula = log(mpg) ~ log(hp) + am, data = mtcars)
##
## Coefficients:
## (Intercept)      log(hp)          am
##       5.1196     -0.4591        0.1954
```

No entanto, algumas transformações podem se confundir com símbolos quem são usados na fórmula. No exemplo abaixo, abstraia os dados e foque no efeito resultante da fórmula:

```
lm(mpg ~ (am + hp)^2 + hp^2, data = mtcars)
```

```
##
## Call:
## lm(formula = mpg ~ (am + hp)^2 + hp^2, data = mtcars)
##
## Coefficients:
## (Intercept)      am          hp          am:hp
## 26.6248479    5.2176534   -0.0591370    0.0004029
```

$(am + hp)^2$ em termos simbólicos retorna $am + hp + am*hp$ e hp^2 retorna hp . No caso em que um símbolo não pode ser usado diretamente, ele deve ser usado dentro da função `I()`:

```
lm(mpg ~ hp + I(hp^2), data = mtcars)
```

```
##
## Call:
## lm(formula = mpg ~ hp + I(hp^2), data = mtcars)
##
## Coefficients:
## (Intercept)      hp      I(hp^2)
## 40.4091172   -0.2133083   0.0004208
```

Variáveis categóricas são convertidas automaticamente para dummies. Por exemplo, vamos adicionar uma variável fictícia chamada `cat`, que receberá valores `a`, `b` e `c` ao data.frame `mtcars`.

```
library(dplyr)
mtcars <- mutate(mtcars,
                 cat = sample(c("a", "b", "c"),
                              size = nrow(mtcars), replace = TRUE))
lm(mpg ~ hp + cat, data = mtcars)
```

```
##
## Call:
## lm(formula = mpg ~ hp + cat, data = mtcars)
##
## Coefficients:
## (Intercept)      hp      catb      catc
## 32.11893     -0.07146    -1.88098    -2.65980
```

Falta agora discutir os principais argumentos da função `lm()`:

```
lm(formula, data, subset, weights, na.action,
  method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
  singular.ok = TRUE, contrasts = NULL, offset, ...)
```

O argumento `formula` já foi discutido anteriormente. É nesse argumento que o modelo é especificado. O argumento `data` recebe o (opcionalmente) um data.frame com os dados. O parâmetro `data` é opcional, porque você pode passar diretamente os vetores de dados. Por exemplo:

```
lm(log(mtcars$mpg) ~ log(mtcars$hp))
```

```
##
## Call:
## lm(formula = log(mtcars$mpg) ~ log(mtcars$hp))
##
## Coefficients:
## (Intercept)  log(mtcars$hp)
##             5.5454        -0.5301
```

Continuando, há possibilidade de estimar o modelo para um subconjunto dos dados, sendo necessário

informar um vetor que seleciona as observações que entrarão na estimação no argumento `subset`. No exemplo que estamos utilizando, suponha que você queira estimar o modelo apenas para os carros automáticos:

```
lm(mpg ~ hp, data = mtcars, subset = (am == 0))

##
## Call:
## lm(formula = mpg ~ hp, data = mtcars, subset = (am == 0))
##
## Coefficients:
## (Intercept)          hp
##    26.62485     -0.05914

lm(mpg ~ hp, data = mtcars, subset = (am == 1))

##
## Call:
## lm(formula = mpg ~ hp, data = mtcars, subset = (am == 1))
##
## Coefficients:
## (Intercept)          hp
##    31.84250     -0.05873
```

Há também a possibilidade de utilizar um vetor de pesos no argumento `weight` para estimação de mínimos quadrados ordinários.

Para ver um sumário dos resultados da estimação, utiliza-se a função `summary()`:

```
summary(lm(mpg ~ hp, data = mtcars))

##
## Call:
## lm(formula = mpg ~ hp, data = mtcars)
##
## Residuals:
##      Min       1Q   Median       3Q      Max 
## -5.7121 -2.1122 -0.8854  1.5819  8.2360 
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 30.09886   1.63392 18.421 < 2e-16 ***
## hp          -0.06823   0.01012 -6.742 1.79e-07 ***
## ---        
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.863 on 30 degrees of freedom
## Multiple R-squared:  0.6024, Adjusted R-squared:  0.5892 
## F-statistic: 45.46 on 1 and 30 DF,  p-value: 1.788e-07
```

12.1.1.1 Acessando os resultados

Além do resumo, é possível acessar uma série de objetos gerados pela função `lm()`, como coeficientes, resíduos, valores preditos (dentro do conjunto de estimação) etc. Primeiro, vamos listar esses elementos:

```
fit <- lm(mpg ~ hp, data = mtcars)
is.list(fit)
```

```
## [1] TRUE
ls(fit)

## [1] "assign"      "call"        "coefficients" "df.residual"
## [5] "effects"     "fitted.values" "model"       "qr"
## [9] "rank"        "residuals"    "terms"       "xlevels"
```

Como se trata de uma lista, podemos acessar os objetos usando o \$.

```
fit$coefficients

## (Intercept)      hp
## 30.09886054 -0.06822828

fit$residuals[1:10]

##          1         2         3         4         5         6
## -1.5937500 -1.5937500 -0.9536307 -1.1937500  0.5410881 -4.8348913
##          7         8         9        10
##  0.9170676 -1.4687073 -0.8171741 -2.5067823
```

Também existem funções para acessar esses resultados:

```
coefficients(fit)

## (Intercept)      hp
## 30.09886054 -0.06822828

residuals(fit)[1:5]

##          1         2         3         4         5
## -1.5937500 -1.5937500 -0.9536307 -1.1937500  0.5410881
```

12.1.1.2 Predições

No R, para realizar predições utiliza-se a função `predict()`, que é uma função genérica. Isso significa que os seus argumentos e os valores retornados dependem da classe do objeto que estamos passando. No caso de um objeto da classe `lm`, é suficiente passar o próprio objeto.

Abaixo está um exemplo do seu uso:

```
set.seed(13034) # para replicação
# 70% dos dados
idx <- sample(nrow(mtcars), size = 0.7*nrow(mtcars), replace = FALSE)
train <- mtcars[idx, ]
test <- mtcars[-idx, ]

# 2 Modelos
fit1 <- lm(mpg ~ hp, data = train)
fit2 <- lm(mpg ~ hp + am + disp, data = train)

# Predições
pred1 <- predict(fit1, newdata = test[,-1])
pred2 <- predict(fit2, newdata = test[,-1])

# Comparando Root Mean Square Errors
library(ModelMetrics)
rmse(pred1, test[, "mpg"])
```

```
## [1] 3.785694
rmse(pred2, test[, "mpg"])
## [1] 3.004418
```

12.1.2 Classificação

Como já mencionado, quando a variável de interesse é categórica utilizamos modelos de classificação. O modelo linear mais conhecido é o chamado *Regressão Logística*.

Suponha que queremos prever se uma pessoa irá ou não pagar a fatura do cartão de crédito. Definimos como p a probabilidade da pessoa não pagar e como razão de chance (*_odds ratio*) o valor $\frac{p}{1-p}$. A função logit, por sua vez, é definida como:

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right)$$

Sendo y a nossa variável dependente, vamos definir que ela recebe valor 1 se o cliente não paga e 0 caso contrário. Logo, o modelo linear para o logit é definido como:

$$\text{logit}(p(y=1|X)) = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \dots + \beta_k x_{ki}$$

Os parâmetros β 's são obtidos a partir de métodos de otimização em que o objetivo minimizar é uma função de perda determinada. Note que a probabilidade de ocorrência do evento pode ser calculada como:

$$p(y=1|X) = \frac{e^{\beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \dots + \beta_k x_{ki}}}{1 + e^{\beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \dots + \beta_k x_{ki}}}$$

Um detalhe importante sobre a regressão logística é que esse modelo de enquadra na classe de modelos lineares generalizados (generalized linear models - glm). Logo, esse modelo pode ser estimado a partir da função `glm()`, escolhendo a família binomial no argumento `family`.

O exemplo a seguir vem do livro An Introduction to Statistical Learning with Application in R. Vamos utilizar o pacote `ISLR` e o conjunto de dados `Smarket` (`?Smarket`). Essa base traz informações sobre as variações do índice S&P 500 entre 2001 e 2005. Esse índice é composto por 500 ativos negociados na NYSE ou Nasdaq.

```
library(ISLR)
data("Smarket")
head(Smarket)

##   Year   Lag1   Lag2   Lag3   Lag4   Lag5 Volume Today Direction
## 1 2001  0.381 -0.192 -2.624 -1.055  5.010 1.1913  0.959      Up
## 2 2001  0.959  0.381 -0.192 -2.624 -1.055 1.2965  1.032      Up
## 3 2001  1.032  0.959  0.381 -0.192 -2.624 1.4112 -0.623     Down
## 4 2001 -0.623  1.032  0.959  0.381 -0.192 1.2760  0.614      Up
## 5 2001  0.614 -0.623  1.032  0.959  0.381 1.2057  0.213      Up
## 6 2001  0.213  0.614 -0.623  1.032  0.959 1.3491  1.392      Up
```

A base consiste em nove variáveis. A variável de interesse é `Direction` e outras cinco variáveis serão usadas como variáveis explicativas ou preditores. Inicialmente, vamos separar nossos dados em treino e teste. Como se trata de um problema de série temporal, vamos utilizar a variável `Year` para separar os dados.

```
train <- Smarket %>%
  filter(Year <= 2004) %>%
  select(-Year)
test <- Smarket %>%
  filter(Year == 2005) %>%
  select(-Year)
```

Agora vamos estimar o modelo:

```
fit <- glm(Direction ~ . - Today, data = train, family = binomial())
summary(fit)

##
## Call:
## glm(formula = Direction ~ . - Today, family = binomial(), data = train)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max 
## -1.302  -1.190   1.079   1.160   1.350 
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)    
## (Intercept) 0.191213  0.333690  0.573   0.567    
## Lag1        -0.054178  0.051785 -1.046   0.295    
## Lag2        -0.045805  0.051797 -0.884   0.377    
## Lag3         0.007200  0.051644  0.139   0.889    
## Lag4         0.006441  0.051706  0.125   0.901    
## Lag5        -0.004223  0.051138 -0.083   0.934    
## Volume      -0.116257  0.239618 -0.485   0.628    
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 1383.3 on 997 degrees of freedom
## Residual deviance: 1381.1 on 991 degrees of freedom
## AIC: 1395.1
##
## Number of Fisher Scoring iterations: 3
```

As predições são realizadas com a função `predict()`, mas com o detalhe que temos que escolher o tipo de predição. O default, `link`, irá passar o logit. Isto é, o valor da predição linear. Já `response`, irá estimar a probabilidade da observação do evento de interesse. Por fim, `terms` retorna uma matriz com a predição linear para cada variável explicativa. O nosso interesse é na probabilidade do mercado ter subido. Logo, usaremos o tipo `response` e iremos transformar a probabilidade em Up e Down.

```
pred <- predict(fit, test, type = 'response')
pred <- ifelse(pred > 0.5, "Up", "Down")
pred <- factor(pred, levels = c("Down", "Up"))
```

Abaixo avaliamos o erro de classificação, que é de, aproximadamente, 52%. Ou seja, pior do que um chute aleatório.

```
# Taxa de erro
ce(test$Direction, pred)
```

```
## [1] 0.5198413
```

Os autores então sugerem estimar o modelo com apenas duas variáveis.

```

fit <- glm(Direction ~ Lag1 + Lag2, data = train, family = binomial())
pred <- predict(fit, test, type = 'response')
pred <- ifelse(pred > 0.5, "Up", "Down")
pred <- factor(pred, levels = c("Down", "Up"))
# Taxa de erro
ce(test$Direction, pred)

## [1] 0.4404762

```

Nesse caso, o modelo acertaria 56% das vezes.

12.2 Exercícios

1. Usando a base de dados `Wage` do pacote `ISLR`, crie dois data.frames, um com 70% dos dados (`train`) e outro com 30% (`test`).
2. Crie um novo objeto chamado `fit` a partir da função `lm()`. Use como variável dependente (Y) a coluna `logwage` e escolha outras três colunas como variáveis explicativas.
3. Compute as previsões desse modelo utilizando a função `predict()`.
4. Compute a raiz do erro quadrático médio (rmse). (`ModelMetrics::rmse()`).
5. Inclua outras variáveis e cheque o que acontece com o `rmse`.

```

library(ISLR)
library(ModelMetrics)

idx <- sample(nrow(Wage), 0.7 * nrow(Wage))
train <- Wage[idx, ]
test <- Wage[-idx, ]

fit <- lm(logwage ~ age + education + maritl + health_ins, data = train)
pred <- predict(fit, test)

rmse(actual = test$logwage, predicted = pred)

## [1] 0.2812615

```

Capítulo 13

Revisão - Titanic

13.1 Objetivo

O objetivo desse capítulo é fazer uma breve revisão do que foi ensinado no curso. Para isso, será usada a base disponível no pacote `titanic`. É esperado que o aluno consiga realizar manipulações nos dados, visualizações e um modelo preditivo. A análise deve ser documentada em um documento criado com o RMarkdown.

13.2 Carregando os Dados

```
library(tidyverse)
library(titanic)
data("titanic_train")
# Base de treinamento
head(titanic_train)

##   PassengerId Survived Pclass
## 1            1        0     3
## 2            2        1     1
## 3            3        1     3
## 4            4        1     1
## 5            5        0     3
## 6            6        0     3
##                                     Name      Sex Age SibSp
## 1           Braund, Mr. Owen Harris    male  22     1
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) female 38     1
## 3          Heikkinen, Miss. Laina female 26     0
## 4       Futrelle, Mrs. Jacques Heath (Lily May Peel) female 35     1
## 5           Allen, Mr. William Henry    male 35     0
## 6           Moran, Mr. James        male  NA     0
##   Parch      Ticket      Fare Cabin Embarked
## 1    0        A/5 21171  7.2500          S
## 2    0         PC 17599 71.2833        C85      C
## 3    0 STON/O2. 3101282  7.9250          S
## 4    0        113803 53.1000       C123      S
## 5    0        373450  8.0500          S
## 6    0        330877  8.4583          Q
```

Variável	Descrição
PassengerId	Identificador do Passageiro
Survived	Variável de indicadora de sobrevivência (0 = Não Sobreviveu, 1 = Sobreviveu)
Pclass	Classe do passageiro
Name	Nome do passageiro
Sex	Sexo do passageiro
Age	Idade do passageiro
SibSp	Número de irmãos/cônjuge no navio
Parch	Número de pais e filhos no navio
Ticket	Número da passagem
Fare	Preço da passagem
Cabin	Código da cabine
Embarked	Porto de embarque

13.3 Manipulando os dados

Nesta seção, faremos alguma alteração nos dados. Veja o summary do dataset

```
summary(titanic_train)
```

```
##   PassengerId      Survived      Pclass      Name
##   Min.   : 1.0   Min.   :0.0000   Min.   :1.000   Length:891
##   1st Qu.:223.5  1st Qu.:0.0000  1st Qu.:2.000   Class  :character
##   Median :446.0  Median :0.0000  Median :3.000   Mode   :character
##   Mean   :446.0  Mean   :0.3838  Mean   :2.309
##   3rd Qu.:668.5  3rd Qu.:1.0000  3rd Qu.:3.000
##   Max.   :891.0  Max.   :1.0000  Max.   :3.000
##
##           Sex          Age      SibSp      Parch
##   Length:891      Min.   : 0.42  Min.   :0.0000  Min.   :0.00000
##   Class  :character 1st Qu.:20.12  1st Qu.:0.0000  1st Qu.:0.00000
##   Mode   :character Median :28.00  Median :0.000   Median :0.00000
##                   Mean   :29.70  Mean   :0.523   Mean   :0.3816
##                   3rd Qu.:38.00  3rd Qu.:1.000   3rd Qu.:0.0000
##                   Max.   :80.00  Max.   :8.000   Max.   :6.0000
##                   NA's   :177
##           Ticket      Fare      Cabin      Embarked
##   Length:891      Min.   : 0.00  Length:891      Length:891
##   Class  :character 1st Qu.: 7.91  Class  :character  Class  :character
##   Mode   :character Median :14.45  Mode   :character  Mode   :character
##                   Mean   :32.20
##                   3rd Qu.:31.00
##                   Max.   :512.33
```

13.3.1 Variável Survived

A variável `Survived` está definida como indicadora (1 ou 0). Como ela será usada em um modelo de classificação, é interessante que ela seja transformada ou criada uma nova variável que a torne uma variável do tipo `factor` ou `character`.

```
titanic_train <- titanic_train %>%
  mutate(Survived = factor(Survived))
levels(titanic_train$Survived) <- c("Não", "Sim")
```

13.3.2 Variável Name

Na variável `Name`, percebe-se que os passageiros possuíam títulos: Mr., Miss., Mrs. etc. Seria interessante criar uma nova variável que possui apenas o título do passageiro. Para isso, precisaremos usar a função `str_extract()` do pacote `stringr` e um pouco de regex.

```
titanic_train <- titanic_train %>%
  mutate(title = str_extract(tolower(Name), '[a-z]{1,}\.\.'))
```

Quais são os títulos mais comuns?

```
titanic_train %>%
  group_by(title) %>%
  summarise(n = n()) %>%
  arrange(-n)
```

```
## # A tibble: 17 x 2
##       title     n
##       <chr> <int>
## 1      mr.    517
## 2     miss.    182
## 3     mrs.    125
## 4   master.     40
## 5      dr.      7
## 6     rev.      6
## 7     col.      2
## 8   major.      2
## 9    mlle.      2
## 10   capt.      1
## 11 countess.    1
## 12     don.      1
## 13  jonkheer.    1
## 14     lady.      1
## 15     mme.      1
## 16      ms.      1
## 17     sir.      1
```

Vamos fazer mais uma modificação. Pode ser interessante agregar os títulos menos frequentes e uma única categoria.

```
classes_de_interesse <- c("mr.", "miss", "mrs.", "master.")
titanic_train <- titanic_train %>%
  mutate(title = ifelse(title %in% classes_de_interesse,
                        title,
                        "other"))
```

13.4 Idade

Como vimos no summary, há alguns valores faltantes para a variável `Age`. Alguns modelos conseguem tratar internamente os missing values, outros não. Para o modelo que vamos usar, não podemos ter missings. Assim, podemos eliminar essas observações ou atribuir um valor. Vamos utilizar a segunda opção.

Para imputação, existem inúmeros métodos, podendo até mesmo ser utilizado o modelo auxiliar. Aqui, vamos inserir a mediana da idade, separando por título e sexo.

```
titanic_train <- titanic_train %>%
  group_by(Sex, title) %>%
  mutate(Age = ifelse(is.na(Age), median(Age, na.rm = TRUE), Age))

summary(titanic_train$Age)
```

```
##      Min. 1st Qu. Median      Mean 3rd Qu.      Max.
##      0.42    22.00   30.00    29.43   35.00    80.00
```

13.4.1 Exercício

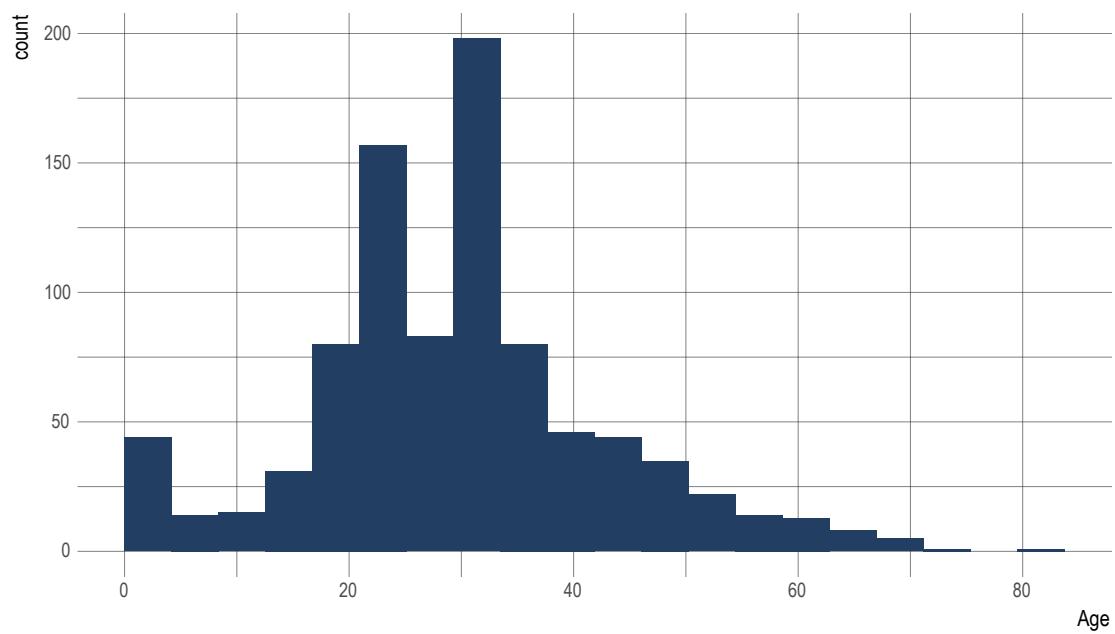
1. Crie mais duas variáveis

13.5 Visualizações

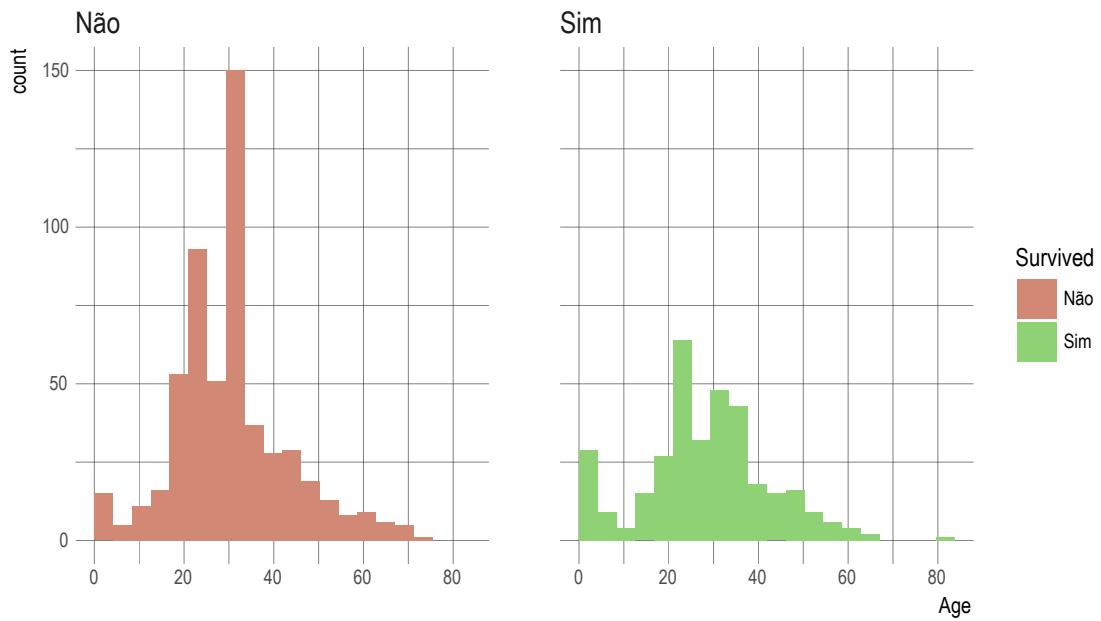
Abaixo, criamos algumas visualizações iniciais. Explore as demais variáveis da base e mostre relações com a variável `Survived`.

```
library(hrbrthemes)
theme_set(theme_ipsum(base_size = 10))

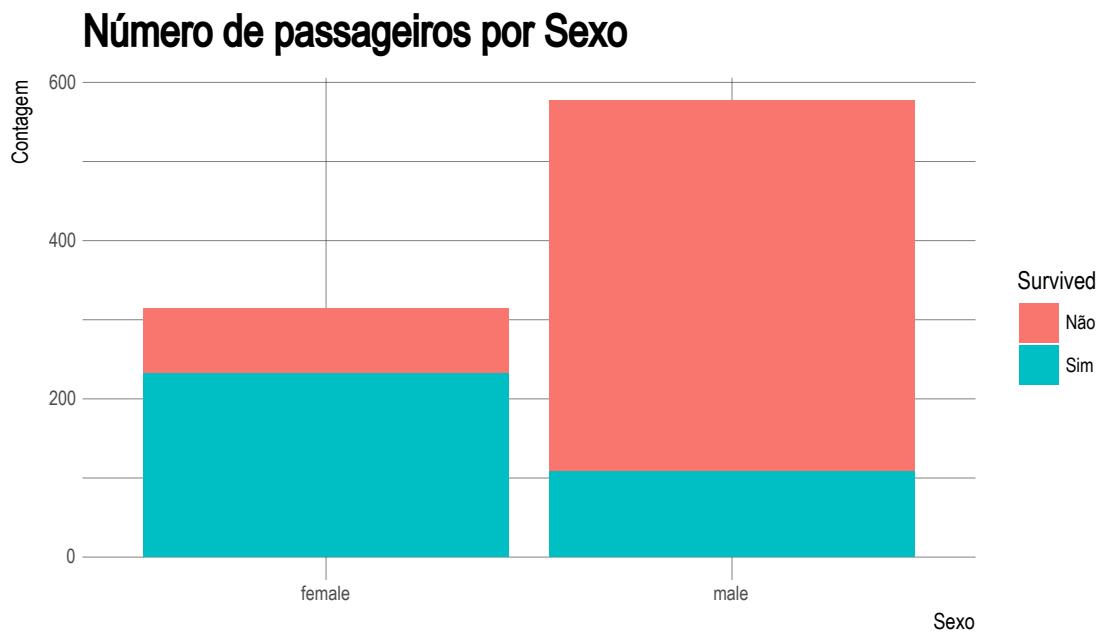
ggplot(titanic_train, aes(x = Age)) +
  geom_histogram(boundary = 0, fill = "#223e63", bins = 20)
```



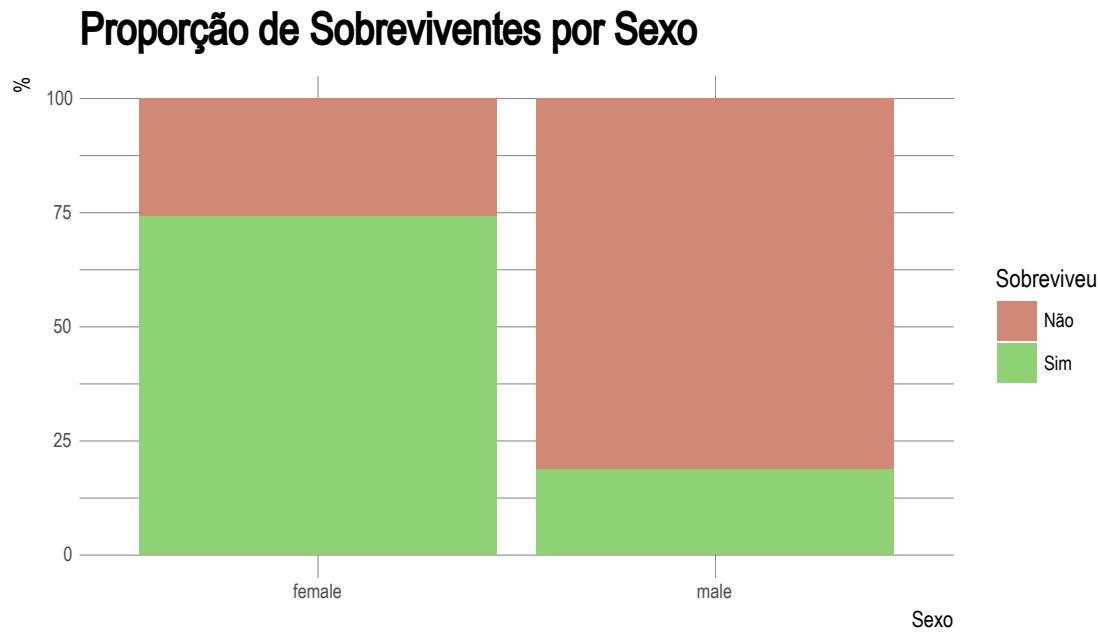
```
ggplot(titanic_train, aes(x = Age, fill = Survived)) +  
  geom_histogram(boundary = 0, bins = 20) +  
  facet_wrap(~ Survived) +  
  scale_fill_ipsum()
```



```
ggplot(titanic_train, aes(x = Sex, fill = Survived)) +  
  geom_bar() +  
  labs(title = "Número de passageiros por Sexo",  
       y = "Contagem",  
       x = "Sexo")
```



```
titanic_train %>%
  group_by(Sex, Survived) %>%
  summarise(n = n()) %>%
  group_by(Sex) %>%
  mutate(prop = n/sum(n) * 100) %>%
  ggplot(aes(x = Sex, y = prop, fill = Survived)) +
  geom_col() +
  labs(title = "Proporção de Sobreviventes por Sexo",
       y = "%",
       x = "Sexo") +
  scale_fill_ipsum("Sobreviveu")
```



13.5.1 Exercício

1. Crie mais duas visualizações

13.6 Modelo Preditivo

1. Crie um modelo preditivo a partir da base de treinamento. Para isso, selecione um subconjunto de variáveis que você irá usar como input (features).
2. Divida a base de treinamentos em duas. 70% para treinamento e 30% para a validação.
3. Utilize a função `glm()` para estimar um modelo de regressão logística.
4. Calcule a acurácia do modelo.
5. Treine o modelo na base `titanic_train` completa
6. Realize previsões para base `titanic_test`.

Capítulo 14

Referências

Bibliografia

Hadley Wickham, G. G. (2017). *R for Data Science*.