Ciência de Dados com R - Introdução

Saulo Guerra, Paulo Felipe de Oliveira e Robert McDonnell

27/05/2018

Conteúdo

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 busca-se 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, Comunicação, Ciências Políticas 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 a 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 varia bastante conforme a necessidade do projeto ou do objetivo que se busca alcançar. Neste curso, usaremos um modelo de workflow bastante utilizado, apresentado no livro R for Data Science (Hadley Wickham, 2017).

Esse workflow propõe basicamente os seguintes passos:

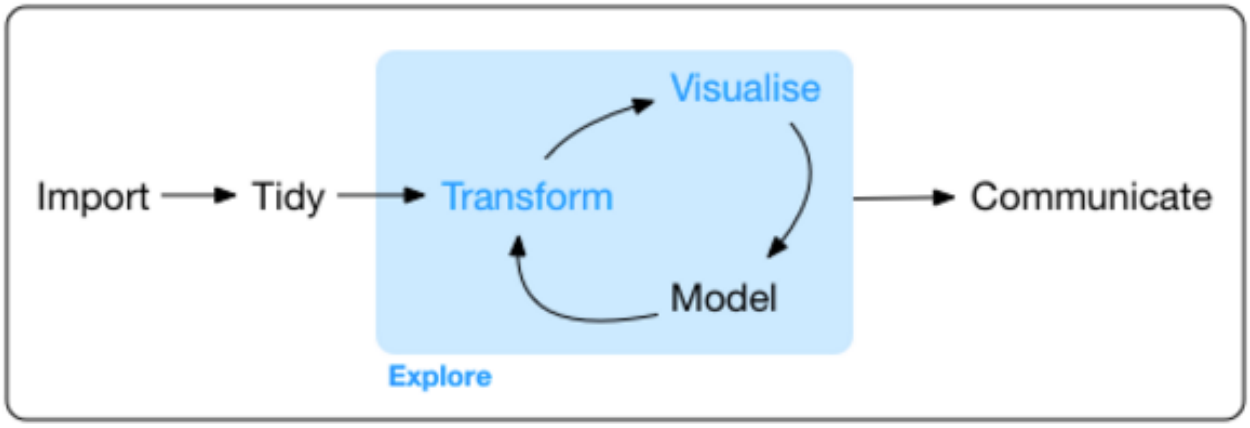


Figura 1.1: Workflow básico para ciência de dados

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

1.3 Linguagens para Ciência de dados

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

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

1.4 O que é R e por que devo aprendê-lo?

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-se R:

- É completamente gratuito e de livre distribuição;
- Curva de aprendizado bastante amigável, sendo muito fácil de se 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-se sua versatilidade, este 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 RStudio:

Repare que, além da barra de menu superior, o RStudio é divido em quatro 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.

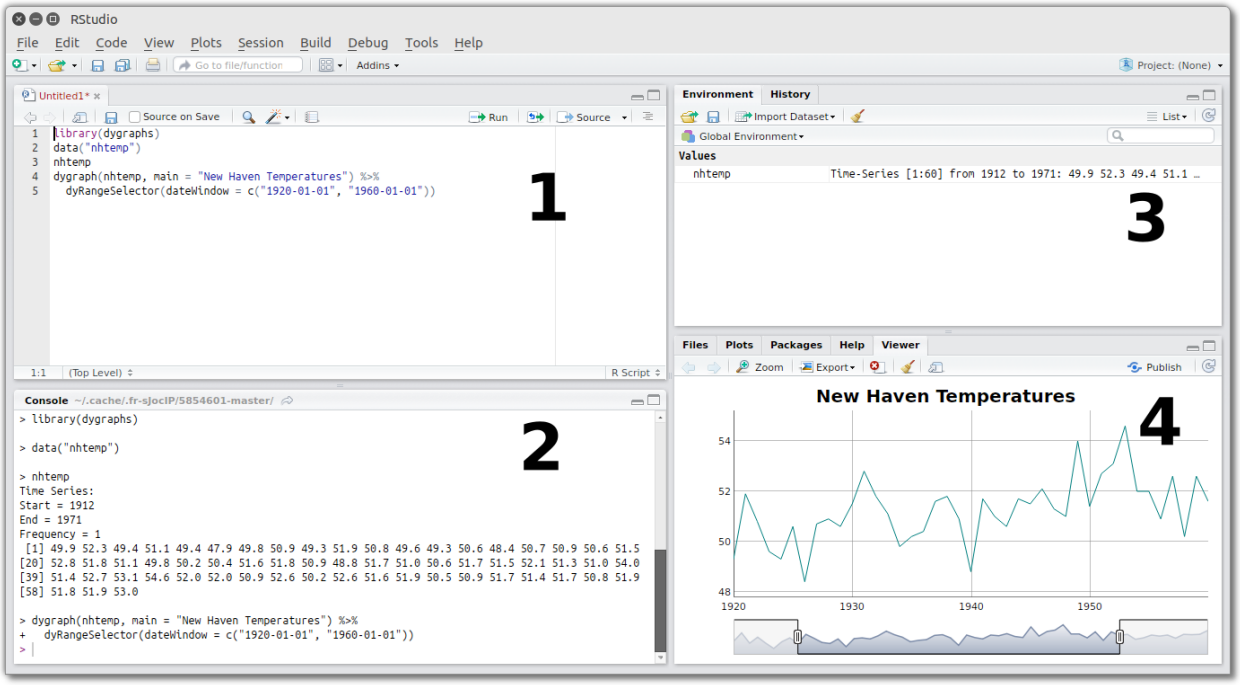


Figura 1.2: RStudio se divide em 4 partes

2. Console
   É no console que o R mostrará a maioria dos resultados dos comandos. Também é possível escrever os comandos diretamente no console, sem o uso do editor de código. É muito utilizado para testes e experimentos rápidos. Um uso rápido do console é, por exemplo, chamar a ajuda do R usando o comando `?` (isso mesmo, a interrogação é um comando!). Voltaremos a falar deste 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 em que finaliza. Neste 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.
   Nesta 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 instalados. 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 isto é possível fazer diretamente no código.

4.3 Aba Help

O nome já diz tudo. Esta 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. Esta funcionalidade é utilizada para visualizar-se localmente conteúdo web. O gráfico da figura está na aba Viewer porque é uma visualização em javascript, que pode ser adicionada a documentos html gerados usando o RMarkdown ou em aplicações web com suporte do Shiny.

1.6 Buscando Ajuda

Independentemente 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 este 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 familiarizar com os comandos e saber quando usá-los. Todos os comandos são baseados em inglês e seus nomes, normalmente, dão dicas a respeito do seu uso.

2.1 Console

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

Olhando para o console, você verá o símbolo `>`. Este 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, como apontado anteriormente. Para entender melhor esse conceito, vamos exercitar um pouco o console de comandos. Digite um por um os 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 executa os comandos, o R vai respondendo. Esse é o comportamento básico do console, muito utilizado para obter-se resultados rápidos de comandos específicos.

Uma funcionalidade importante é o fato de que o R guarda um histórico dos últimos comandos executados e para acessá-los basta apertar a seta para cima no teclado.

2.2 Scripts

Enquanto no console seus comandos são executados na medida em que você os "envia" com a ordem *enter*, em um script você ordena a execução de uma sequência de comandos, escritos previamente, um seguido do outro. Esses 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 a 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, já a segunda executa mostrando as respostas no mesmo. A primeira opção será útil em outros casos de scripts muito grandes, ou em situações nas quais não convenha "poluir" o console com um monte de mensagens.

Há um atalho de teclado para o ***Source***: *ctrl + shift + enter*. Aprenda este 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, aquele 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 funcionarem corretamente.

Há, também, um atalho de teclado para o ***Run***: *ctrl + enter*. Este é outro comando importante de ser lembrado, pois é muito importante e será muito utilizado.

2.2.1 Salvando Scripts

Ao digitar seus comandos no console, o máximo que você consegue recuperar são os comandos imediatamente 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 extesão `.R`, o windows abrirá esse arquivo diretamente 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 na hora de armazenar os seus arquivos. Utilize pastas para os diferentes projetos e dentro delas escolha nomes explicativos para os seus trabalhos. 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 e já que ele não consegue entendê-lo, exibe uma mensagem de erro no console.

O símbolo de comentário também é muito útil para suprimir linhas de código que servem 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 tem alguma noção de estatística, provavelmente já tem a intuição do que é uma variável para uma linguagem de programação. No contexto do R, vamos entender variável como um objeto, ou seja, como uma estrutura predefinida que vai "receber" algum valor. Utilizando uma linguagem mais técnica, objeto (ou variável) é um pequeno espaço na memória do seu computador onde o R armazenará um valor ou o resultado de um comando, utilizando um nome que você mesmo definiu.

Conhecer os tipos de objetos do R é fundamental. Para criar objetos, utiliza-se o símbolo `<-`. Este provavelmente é o símbolo que você mais verá 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 mostrará 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
```

Dê 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 os objetos que você criar estarão disponíveis na aba ***Environment***.

O RStudio possui a função *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 listará tudo que começar com essas letras no arquivo. Selecione algum item e aperte *enter* para escrevê-lo no editor de códigos.

2.4 Funções

Entenda função como uma sequência de comandos preparados para serem usados de forma simples e, assim, 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 o instala, o R já vem configurado 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 encontrará pela frente. Nesse sentido, o R também mostra outro ponto forte. Você pode instalar conjuntos extras de funções específicas de maneira muito simples: usando pacotes.

Funcões 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 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
`sum()`	Soma de valores
`mean()`	Média
`var()`	Variância
`median()`	Mediana
`summary()`	Resumo Estatístico
`quantile()`	Quantis

2.5 Pacotes

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

Caso você precise de algum pacote específico, procure no Google pelo tema que necessita. Você encontrará o nome do pacote e o instalará normalmente.

Nesse link você pode ver uma lista de todos os pacotes disponíveis no repositório central. Além desses, ainda existe a possibilidade de instalar-se 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 a necessidade de instalá-lo 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 o pacote não funcionará. 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 perceberá que quanto mais organizado e padronizado mantiverem-se os 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 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, escolha apenas um padrão e seja consistente a ele.

- 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 são suficientes ou simplesmente não fornecem a maneira mais fácil de resolver-se um problema. Neste curso utilizaremos o Tidyverse: uma coleção de pacotes R cuidadosamente desenhados para atuarem 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, tidyr, 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 ouvirá 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 se resolverá com a 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 a 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 generalista, que diz respeito a como os dados são disponibilizados. Sendo assim, podemos classificar os dados em três 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, estes dados não têm uma estrutura previsível, ou seja, cada conjunto de 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, e-mails, twitters, PDF, imagens, vídeos etc.

Analisar este tipo de dado é muito mais complexo e exige conhecimento avançado em mineração de dados. Apesar disso, é o tipo de dado 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, seguem 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 Application programming interface (API). 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 vai "ler" e "escrever" os dados. Para verificar qual o diretório que 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 a leitura dos principais tipos de arquivos. Um outro pacote específico, e muito bom para isso, é o `readr`. O Tidyverse inclui o carregamento do pacote `readr`.

Diversos tipos de arquivos são lidos pelo R: Comma-Separated Values (csv), 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 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 e o fim das amostras. É muito importante sempre observar a "aparência" dos dados após o carregamento. Essa observação 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 são o `str()`, o `class()` e o `summary()`.

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

```
class(senado)
```

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

Já para verificar a estrutura do objeto, ou seja, seus campos (quando aplicável), insira:

```
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 200700
##  $ 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 2 ...
##  $ Origin      : int  11 11 11 11 11 11 11 11 11 11 ...
##  $ Contentious : int  0 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 1 ...
##  $ Content     : chr  "Creates the Senate Commission of Science, Technology, Innovation, Communicati
##  $ Round       : int  1 1 1 1 1 1 1 1 1 1 ...
##  - attr(*, "spec")=List of 2
##   ..$ cols   :List of 15
##   .. ..$ VoteNumber  : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ SenNumber   : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
```

```
##   .. ..$ 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 etc.), quando aplicáveis:

```
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), 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 exploraremos melhor os tipos de objetos 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 vinte, a segunda (estado) tem tamanho dez e a terceira (código) tem tamanho doze.
3. Investigue os parâmetros das funções de leitura do R base: `read.csv()`, `read.delim()` e `read.fwf()`. Notou as 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.

Entendamos a manipulação de dados como o ato de transformar, reestruturar, limpar, agregar e juntar os 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 a textos, números, ou simplesmente a afirmação de verdadeiro ou falso (Boleano).

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

Repare nas atribuições acima. Usaremos 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. Para valores inteiros ou decimais, `numeric` , `character` para valores textuais e `logical` para valores lógicos (verdadeiro ou falso). 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 utilizado 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 funcão para repetir números
vetor.num.sequencia <- 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 2 2 2 2 2 2
## [36] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

```
vetor.num.sequencia
```

```
##  [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 a criação de vetores, usamos a função de combinação de valores `c()` (combine). Esta função vai combinar todos os parâmetros em um único vetor. Lembre-se: vetores são sequências que contêm apenas um 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)
adulto <- 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 os dados 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 se realize as conversões explicitamente com as funções apropriadas ao invés de confiar na coerção do R, a não ser que se tenha certeza do resultado.

4.1.2 Outros tipos de variáveis

Existem outros tipos de variáveis bastante utilizados. Citaremos alguns deles, pois nesse curso utilizaremos muito pouco os demais tipos.

Tipo	Descrição	Dimensões	Homogêneo
vector	Coleção de elementos simples. Todos os elementos precisam ser do mesmo tipo 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 complexo com elementos que podem ser de diferentes tipos	1	Não

Tipo	Descrição	Dimensões	Homogêneo
data.frame	Tipo especial de lista, onde cada coluna é um vetor de apenas um tipo e todas as colunas têm o mesmo número de registros. É o tipo mais utilizado para se trabalhar com dados	2	Não

Tipo	Descrição	Dimensões	Homogêneo
factor	Tipo especial de vector, que só contém valores predefinidos (levels) e categóricos (characters). Não é possível adicionar novas categorias sem criação de novos levels	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 se fazer em casos de `NA`, e nem sempre a solução será a mesma: varia 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", `' '`. Neste caso, você provavelmente terá que substituir esses valores "omissos" por `NA`. Imputar dados em casos de `NA` é uma das várias estratégias para lidar-se com ocorrência de missing no conjunto dos dados.

Seguem algumas funções úteis para lidar-se 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, esta

função diz quais são as linhas (amostras) completas em todas as 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 FALSE  TRUE FALSE
##  [12] FALSE 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 FALSE  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 FALSE FALSE FALSE FALSE FALSE FALSE FALSE 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 FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE 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-se uma tarefa diferente. A estrutura parece 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 utilizada 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 funcão de leitura? Nesse caso, basta fazer um loop para percorrer todos os arquivos da pasta e ler 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 entendermos 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, diferentemente do `for()`, o while executa as tarefas repetidamente até que uma condição seja satisfeita, 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 é:

```
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 que o código espere por mais cinco segundos antes de começar o loop novamente.

```
while (length(automatico) == 0) {
  automatico <- list.files('dados/automatico/')
  if(length(automatico) > 0) {
    print('O 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. Vejamos então como criar uma função:

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

  # sequência de tarefas

  return(valores_retornados)
}

# chamada da função
sua_funcao
```

Agora 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 sifrão.

```
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 vector
```

```
## [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, por possuir 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 posições 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 (posição antes da vírgula) e a coluna (posição após a vírgula) que 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 o item desejado utilizando 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 é `TRUE` ou `FALSE` (tipo `logic`). Para melhor entendimento, selecionamos 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, por isso, de mais rápido aprendizado.

4.5 Pacote dplyr

O forte do pacote `dplyr` é a sintaxe simples e concisa, o que facilita o aprendizado e torna 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 carregará automaticamente o pacote `dplyr`, mas você também 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, sumarizar, filtrar, ordenar, criar variáveis, joins, dentre outras.

As funções do `dplyr` 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`). Este 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. 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 e agrupado, em consequência disso, menos detalhado. O 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:

```
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 2013 ...
##  $ month         : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ day           : int  1 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 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. Para isso, 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 acrescentamos verbetes e encadeamos a manipulação com o uso de `%>%`.

A primeira parte `serie.orig %>%` é a passagem onde você informa o data.frame que utilizará na sequência de manipulação. A partir daí, as chamadas seguintes `select() %>%`, `filter() %>%`, `mutate() %>%` etc, são os encadeamentos de manipulação que você pode fazer 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 às 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 de "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. O formato ideal para analisar dados, visualmente, é diferente do formato ideal para analisá-los 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. Utiliza, ainda, 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 espécie de regra geral a qual diz que um dado bem estruturado deve conter uma única variável em uma coluna e uma única observação em uma linha.

Observando-se 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 dificulta bastante certas análises sistemáticas. O melhor a se fazer é converter a primeira tabela a um modelo mais próximo o possível da segunda tabela.

Infelizmente, não temos como apresentar um passo a passo padrão para limpeza de dados, pois isso depende 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 ao final da sua análise. Supondo que você organizou seus dados perfeitamente, conseguiu executar os modelos que gostaria, gerou diversos gráficos interessantes e está satisfeito 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. Neste 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 exemplificar, 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 tidyr

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

O pacote `tidyr` é mais um dos pacotes criados por Hadley Wickham. Este 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("tidyr")
library(tidyr)
?tidyr
```

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

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

Vale lembrar que tudo que for feito usando o `tidyr` é possível executar também usando o R base, mas de 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. Assim, o resultado após o agrupamento é 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)
```

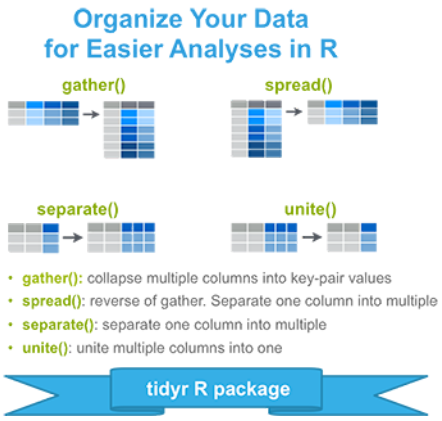


Figura 5.3: Tabela long

```
## '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 guardará os valores para cada uma das antigas colunas. Repare 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 viram 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. Lembre-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 separá-las.

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. É muito utilizada para montar relatórios finais ou tabelas para análise visual. Aproveitemos o exemplo em `table2` para montarmos 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 representa 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 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 acaso, este pacote não está incluído no `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 substituem 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', '19.5
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, com muita prática e experiências reais.

A seguir, uma lista dos principais mecanismos de regex:

regex	correspondência
`^`	começa do string (ou uma negação)
`.`	qualquer caractere
`$`	fim da linha
`[maça]`	procura os caracteres `m`, `a`, `ç`
`maça`	`maça`
`[0-9]`	números
`[A-Z]`	qualquer letra maiúscula
`\\w`	uma palavra
`\\W`	não é palavra (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://regexone.com/

5.4 Exercícios

1. 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
```

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

```
cadastros <- data.frame(
  email = c('joaodasilva@gmail.com', 'rafael@hotmail.com', 'maria@uol.com.br', 'juliana.morais@outlook.
  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 estes 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
```

A união de dados é a forma mais simples de juntá-los.

6.2 Cruzamento de Dados (Join)

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

CAMPO 1	CAMPO 2	CAMPO 3
1	A	TRUE
2	B	TRUE
3	C	FALSE

CAMPO 1	CAMPO 2	CAMPO 3
4	X	FALSE
5	W	TRUE
6	Y	FALSE
7	Z	TRUE

CAMPO 1	CAMPO 2	CAMPO 3
1	A	TRUE
2	B	TRUE
3	C	FALSE
4	X	FALSE
5	W	TRUE
6	Y	FALSE
7	Z	TRUE

Figura 6.1: União de tabelas

CAMPO A	CAMPO B	CAMPO C
AA	X	1
BB	XX	2
CC	XXX	3
DD	XX	4
EE	XXX	5

CAMPO C	CAMPO D	CAMPO E
1	TRUE	89,2
2	TRUE	1,9
3	FALSE	591,4
4	FALSE	81923,22
5	TRUE	912,5

CAMPO A	CAMPO B	CAMPO C	CAMPO D	CAMPO E
AA	X	1	TRUE	89,2
BB	XX	2	TRUE	1,9
CC	XXX	3	FALSE	591,4
DD	XX	4	FALSE	81923,22
EE	XXX	5	TRUE	912,5

Figura 6.2: Cruzamento de tabelas

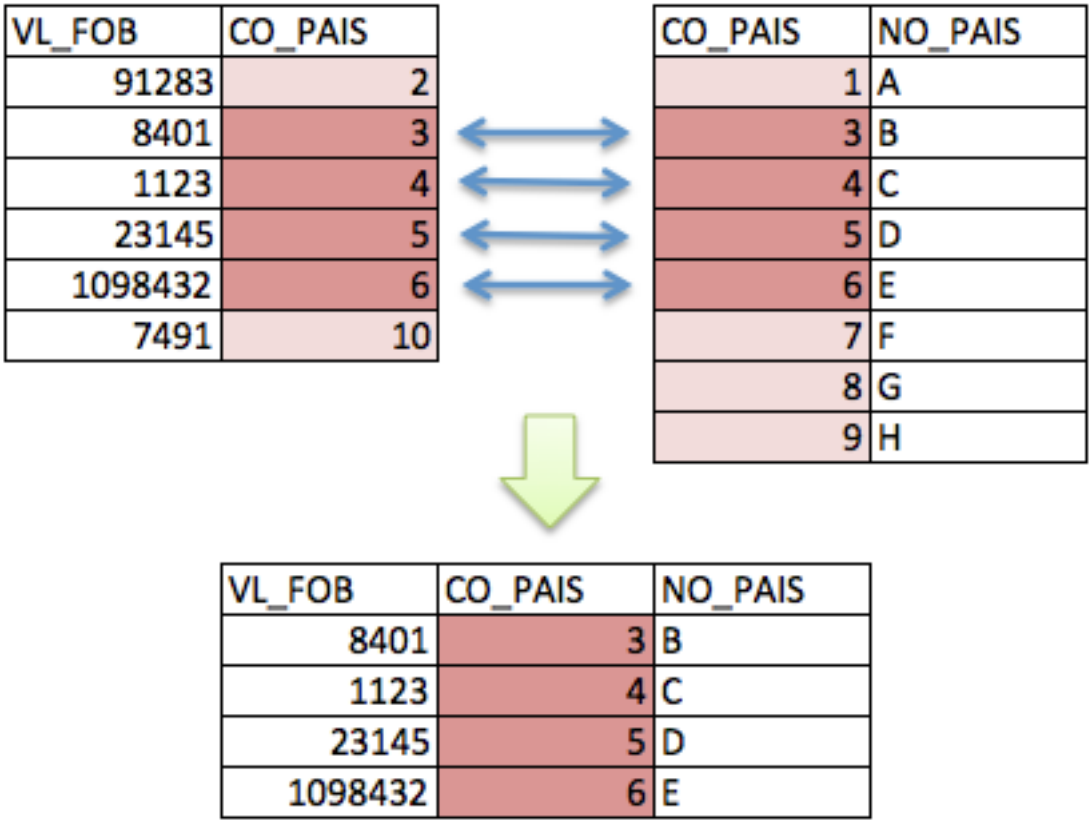


Figura 6.3: Cruzamento de tabelas

Para entender-se como fazer "joins" (cruzamentos), é preciso compreender-se 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 isto, mas, para começarmos a entender e usar os joins, basta usar essa intuição.

Tendo esse conceito simplificado de chave em mente, a primeira coisa que se deve fazer quando for preciso cruzar dois conjuntos de dados é tentar identificar quais os campos chaves, ou seja, quais 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 a 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 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. Esta característica torna o inner join muito útil para fazer-se 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 têm três registros. Então, por que o resultado final só tem dois registros? A resposta é simples: porque 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_intruments2`:

```
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 três tipos básicos de outer join: left outer join (ou só left join), 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 à esquerda" estarão presentes no resultado final, além dos registros à 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
```

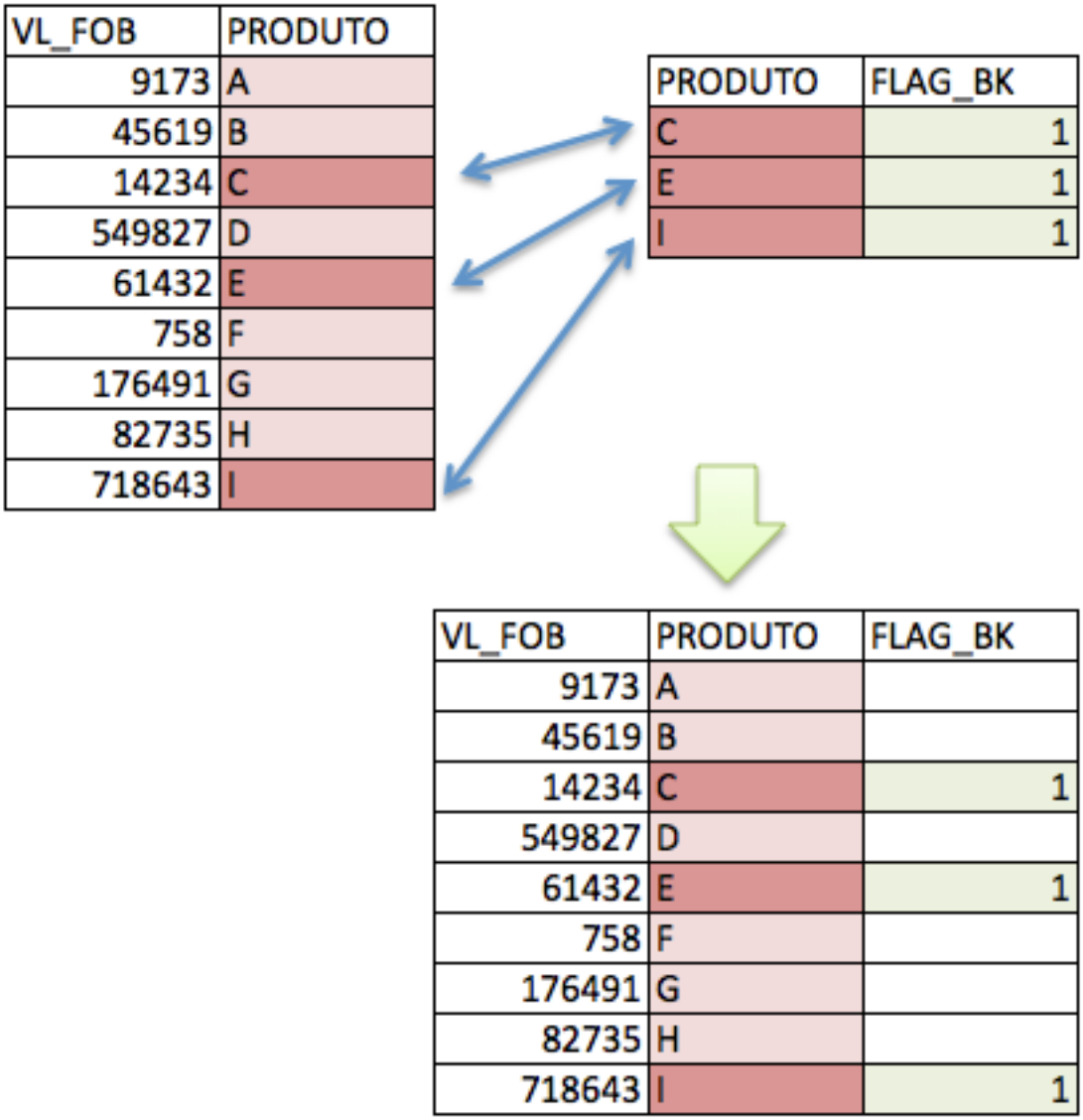


Figura 6.4: Cruzamento de tabelas

```
## 3  Paul Beatles   bass
```

```
band_instruments2
```

```
## # 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 “à direita” (band_instruments2), ele apareceu no registro final com `NA`, no campo que diz respeito ao conjunto à direita. Da mesma forma, Keith não está presente no conjunto final, pois não tem referência no conjunto à esquerda.

Repare que a “posição” das tabelas faz diferença. No caso da nossa manipulação de exmeplo, 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 à 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 depende completamente da ordem em 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 deles.

```
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
```

Reparem que, dessa vez, não perdemos nenhum registro, de nenhum conjunto 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
```

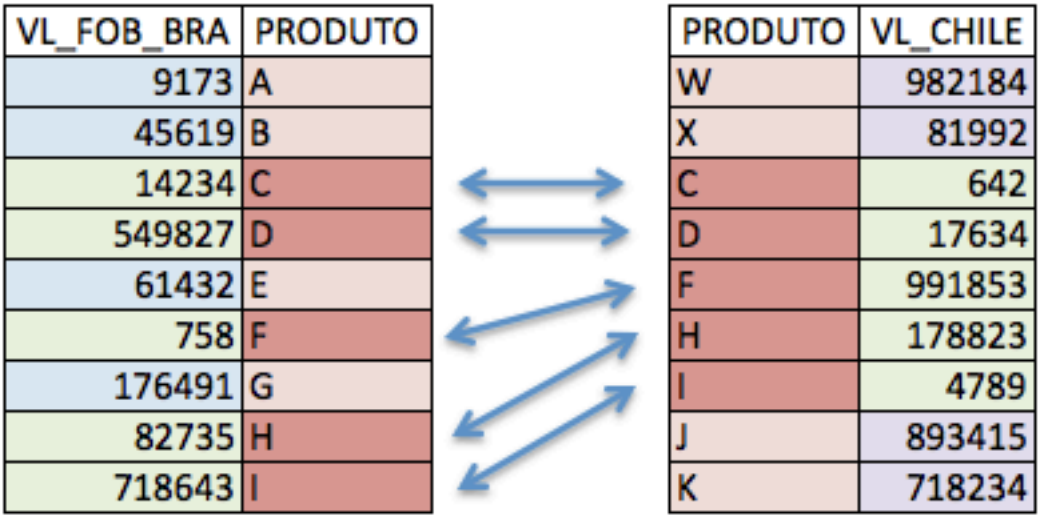
```
## # A tibble: 7 x 6
##   Department.x DepartmentName Manager EmployeeName Department.y Salary
##          <int>          <chr>   <int>        <chr>        <int>  <int>
## 1           11     Production       1        Alice           11    800
## 2           12          Sales       4       Daniel           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

1. 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
```



VL_FOB_BRA	PRODUTO
9173	A
45619	B
14234	C
549827	D
61432	E
758	F
176491	G
82735	H
718643	I

PRODUTO	VL_CHILE
W	982184
X	81992
C	642
D	17634
F	991853
H	178823
I	4789
J	893415
K	718234

PRODUTO	VL_FOB_BRA	VL_CHILE
A	9173	
B	45619	
C	14234	642
D	549827	17634
E	61432	
F	758	991853
G	176491	
H	82735	178823
I	718643	4789
J		893415
K		718234
X		81992
W		982184

Figura 6.5: Cruzamento de tabelas

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

2. Utilizando os dataframes abaixo, chegue no resultado a seguir:

```
participantes <- data.frame(
  Nome = c('Carlos', 'Maurício', 'Ana Maria', 'Rebeca', 'Patrícia'),
  Estado = 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       Estado 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       Estado Idade Pontuacao Resultado
## 1    Carlos     Brasília    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
```