Linguagem R

Primeiros passos para análise de dados

Wagner Hugo Bonat & Walmes Marques Zeviani

Contents

Prefácio **5**
Contexto e motivação 5
Objetivos e público alvo 6
Seleção dos tópicos 6

1 Introdução a Ciência de Dados 2: Manipulando dados no R **9**
1.1 Importando dados 10
1.2 Arrumando dados 13
1.3 Manipulando dados 17
1.4 Combinando dados 25
1.5 Exportando dados 28
1.6 Praticando: Mortalidade por causas externas 29

Prefácio

**Essa é uma versão preliminar do que imaginamos um dia se tornar um livro. Nesta versão ainda não foram incluídas as referências bibliográficas e algumas referências cruzadas estão indefinidas. Se você achar erros de digitação, concordância etc e puder nos informar pelo e-mail wbonat@gmail.com, agradecemos muito! IMPORTANTE: este material não foi revisado por pares até o momento 03/12/2021.

Contexto e motivação

O desenvolvimento da tecnologia da informação atrelado à disponibilidade cada vez maior de dispositivos como *smartphones* e *tablets* conectados à internet vêm trazendo uma verdadeira revolução em praticamente todas as áreas do conhecimento. A chamada **Era da Informação** vem alterando drasticamente a forma como vivemos e interagimos como sociedade.

O grande volume de pessoas conectadas está revolucionando todas as formas de negócio. Esta revolução é baseada na capacidade cada vez maior de armazenar e gerenciar dados de como as pessoas se comportam em praticamente todos os aspectos de sua vida *online*. Todo esse fluxo de dados vem se tornando o principal valor de muitos negócios e, em muitos casos, vital para o seu sucesso. Um exemplo clássico é a empresa Google: detentora do buscador mais famoso do mundo, é uma das maiores empresas de tecnologia da atualidade e está presente em nossas vidas 24 horas por dia sete dias por semana.

Com toda esta geração de dados resultante do acesso à internet, empresas de todos os ramos estão buscando extrair algum tipo de valor destas informações com o objetivo de atrair, manter ou ainda melhorar a forma de se relacionar com seus clientes. Neste contexto fica evidente que empresas que sabem lidar de forma efetiva com dados tem uma grande vantagem competitiva. Empresas são formadas por pessoas, e assim surgiu a profissão denominada de **Cientista de Dados**. Trata-se de uma profissão para a qual ainda muito se discute sobre suas atribuições, contudo é composta minimamente de três habilidades chaves:

i) Conhecimento de ferramentas para análise e resumo de dados. Estas ferramentas em geral são herdadas de uma área mais antiga chamada de **estatística** que vem há anos sendo o principal motor da ciência como um todo.

ii) Conhecimento de ferramentas computacionais capazes de lidar com a quantidade e complexidade crescente de dados. Nesta área entram as chamadas ferramentas de **big data** que em geral são dominadas por profissionais da ciência da computação.

iii) E por fim, de nada serve conhecer métodos estatísticos e ferramentas computacionais sem saber exatamente o que fazer com elas para resolver problemas relevantes e gerar valor por meio da análise de dados. Isso só é possível tendo um conhecimento razoável da área de negócio em que se está trabalhando.

Neste cenário, o uso de ferramentas computacionais que permitam analisar quantidades massivas de dados usando métodos estatísticos cada vez mais complexos é de suma importância. A linguagem de programação R pela sua flexibilidade, poder, sofisticação e expressividade vem se tornando uma das principais ferramentas do cientista de dados em todo o mundo.

O R (como vamos chamar o software daqui em diante) é uma linguagem e ambiente para computação estatística e gráficos. É um projeto da GNU similar a linguagem S desenvolvida nos Laboratórios Bell por John Chambers e seus coautores. O R surgiu em 1993 e é de autoria de Ross Ihaka e Robert Gentleman, do Departamento de Estatística da Universidade de Auckland, na Nova Zelândia. O R é um software livre e de código aberto sob os termos da *Free Software Foundation's GNU General Public License*. Pode ser facilmente instalado em uma grande variedade de plataformas UNIX incluindo FreeBSD e Linux, Windows e MacOS.

Objetivos e público alvo

O objetivo deste livro é oferecer um texto introdutório e acessível para profissionais de diversas áreas sobre os fundamentos da linguagem de programação R. É importante ressaltar que o texto explica a linguagem R em sua essência ao invés de focar em alguns *frameworks* como, por exemplo, o *tidyverse*. O objetivo é justamente pavimentar o caminho de um profissional que busca uma formação completa e sólida para analisar grandes volumes de dados utilizando a linguagem R e todo o seu potencial, sem particularizar em nenhum *framework*.

O público-alvo são profissionais que desejam uma introdução acessível aos fundamentos da linguagem R de forma rápida e padronizada. Tais fundamentos são necessários para o profundo entendimento da linguagem e uma maior desenvoltura no uso de *frameworks* específicos de ciência de dados e modelagem estatística avançada. Formações usuais destes profissionais incluem, mas não está limitada à: engenharia, ciência da computação, economia, física, administração, gestão da informação e tecnologia em análise e desenvolvimento de sistemas.

Seleção dos tópicos

Selecionar os tópicos para este livro não foi uma tarefa trivial dado a quantidade enorme de recursos disponíveis na linguagem R. Optamos por uma descrição sucinta da sua instalação e configuração padrão. Seguida de uma discussão das principais IDEs (*Integrated Development Environment*) disponíveis. Buscamos orientar o leitor iniciante sobre qual IDE pode ser a mais adequada dependendo do seu perfil. Neste material optamos por usar o RStudio por ser provavelmente a IDE mais popular para R e considerada adequada para iniciantes em programação.

Seguimos com os primeiros passos na linguagem apresentando o código, a área de trabalho, como acessar a documentação (tanto interna como a vasta documentação disponível na internet) e os tipos de arquivos que o R gera para armazenar os resultados de uma análise. O R é um software colaborativo; isso significa que existe uma comunidade unida e ativa desenvolvendo e contribuindo com uma série de métodos e ferramentas para melhorar e expandir as capacidades da linguagem. Tudo isso é disponibilizado para os usuários por meio de pacotes. Assim, apresentamos como instalar, usar e gerenciar pacotes adicionais.

Passada esta ambientação inicial, seguimos para a apresentação das operações matemáticas e lógicas da linguagem. Após as operações passamos para as estruturas de controle como *if*, *else*, *switch* e *cases*, bem como as estruturas de repetição *for*, *foreach*, *while*, *repeat* e *do-until*. O conhecimento de tais estruturas levam a um domínio maior da linguagem, principalmente para implementação de solução de problemas não explorados.

Um aspecto importante do trabalho do cientista de dados é estar apto a construir grandes algoritmos encadeados com estruturas bem definidas e ordenadas. Para isto, a construção de funções é primordial. Nós apresentamos uma descrição detalhada da implementação de funções em `R` com toda a especificidade da linguagem focada para a implementação computacional de técnicas estatísticas e matemáticas. A partir deste momento você não pergunta mais se o `R` faz alguma coisa ou se tem alguma técnica implementada. Você será capaz de implementar o que desejar mesmo que a linguagem não tenha a funcionalidade nativa. Por fim, apresentamos as principais estratégias para importação de arquivos de dados em diversos formatos para o `R`, bem como a sua manipulação usando a classe *data frame*.

O texto foi escrito utilizando o sistema para análise reproduzível `knitr`. Por meio da linguagem de marcação markdown e a estrutura de livro fornecida pelo pacote `bookdown` dentro da linguagem `R`. É uma iniciativa da **Ômega Data Science** para formar uma geração de Cientistas de Dados com forte fundamentação matemática e de computação estatística baseada em `R`.

W.H.B e W.M.Z

Curitiba, Outubro, 2021.

Chapter 1

Introdução a Ciência de Dados 2: Manipulando dados no R

Neste Capítulo serão discutidas as tarefas básicas de importação, arrumação, manipulação e exportação de dados usando o `R`. Serão adotados os princípios do *tidy data* (dados arrumados). Vamos ver por meio de exemplos as cinco principais ações de manipulação de dados: filtrar, ordenar, selecionar, criar/transformar e resumir. Além disso, tarefas usuais de arrumação de dados como disposição do formato longo para o curto e vice-versa e união de diferentes conjuntos de dados via um código indexador serão discutidas e exemplificadas. Por fim, veremos algumas estratégias e cuidados para lidar com dados perdidos.

Ao trabalhar com análise de dados lidamos com dados de todos os tipos e formatos. O `R` nativamente apresenta um conjunto completo de ferramentas para importar, manipular e exportar conjuntos de dados. Também existem pacotes especializados para estas tarefas, tais como os pacotes do framework `tidyverse`: `readr` para leitura, `tidyr` para arrumação e o `dplyr` para manipulação. Neste Capítulo vamos discutir os aspectos básicos da manipulação de dados usando os princípios do `tidy data` ou dados arrumados. Este termo ficou famoso após o artigo *Tidy Data*. O chamado *tidy data* segue três principios:

- Cada variável está em uma coluna.
- Cada observação está em uma linha.
- Cada tipo de unidade observacional está em uma célula.

A Figura 1.1 mostra uma representação de um conjunto de dados em que os 3 princípios do *tidy data* são atendidos.

A ideia parece simples mas na prática pode ser confuso. A melhor forma de entender esses conceitos é por meio da prática e exemplos. Neste Capítulo vamos lidar com conjuntos de dados simples para ilustrar os principais aspectos da importação, manipulação e exportação de dados.

A estrutura deste Capítulo segue a ordem lógica de uma análise de dados:

- **Importação de dados**: Nesta etapa o objetivo é a partir de uma fonte primária de dados importá-los para o ambiente de programação `R`. Fontes de dados como o datasus e similares são as mais populares. Em geral os arquivos serão obtidos por meio de algum sítio eletrônico em um formato de texto pleno como .csv, .tsv, .txt entre outros. Outra forma usual é os dados serem disponibilizados em planilhas eletrônicas. Por fim, menos popular é a opção de acessar um banco de dados relacional como MySQL, MariaDB e similares.

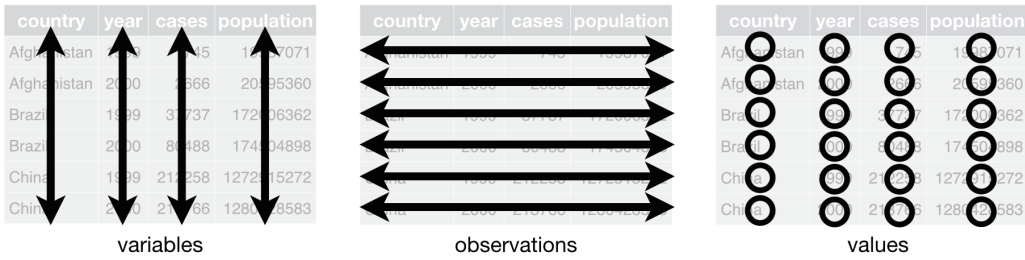


Figure 1.1: Representação dos 3 princípios do tidy data.

- **Arrumação de dados**: Após importar a base de dados é comum que ela não esteja organizada como desejamos para análise. Assim, o objetivo desta etapa é fazer com que os três princípios do *tidy data* sejam respeitados.
- **Manipulação de dados**: Com os dados arrumados podemos começar a conhecê-los. Isso envolve uma série de tarefas que podem ser resumidas em: ordenar, selecionar, filtrar, criar/transformar e resumir registros ou observações.
- **Combinação de dados**: Em situações mais complexas pode ser necessário lidar com diferentes fontes de dados de forma conjunta. Assim, ter ferramentas para combinar diferentes conjuntos de dados é muito importante.
- **Exportação de dados**: Por fim, realizada a análise pode ser de interesse exportar o conjunto de dados organizado ou mesmo diferentes versões com diferentes resumos para serem consumidos por outros aplicativos. Exemplos incluem softwares de visualização de dados e as populares planilhas eletrônicas.

Nas próximas seções cada uma destas etapas será discutida com mais detalhes e diversos exemplos.

1.1 Importando dados

No universo da análise de dados existem diversos tipos de arquivos, tais como: arquivos de texto pleno, planilhas eletrônicas e banco de dados relacionais. Provavelmente, a forma mais popular de armazenar um pequeno conjunto de dados é em uma planilha eletrônica. Todas as planilhas eletrônicas podem facilmente exportar uma tabela no formato retangular em um arquivo de texto pleno (*plain text*). Para salvar uma planilha em formato texto pleno na maioria das planilhas eletrônicas basta selecionar o Menu -> Arquivo (*File*) -> Salvar como (*Save as*) e selecionar o formato desejado. Um cuidado importante é deixar a sua planilha o mais simples possível, ou seja, sem formatações, cores, células mescladas, etc. Lembre-se: cada coluna deve ser uma variável e cada linha uma observação. Entre os diversos formatos disponíveis os mais populares são: `.csv` (*comma-separated value*) e `.tsv` (*tab separated values*) ou versões destes.

A principal diferença entre esses formatos é o caracter ou estratégia usada para separar as colunas da base de dados. Os nomes são auto-explicativos, por exemplo, no formato `.csv` as colunas são separadas pelo símbolo `,` (vírgula). Em português usamos a `,` para representar decimais, por isso usamos o `;` (ponto e vírgula) para separar as colunas. Na prática tanto faz o formato desde que você saiba qual é o caracter ou estratégia usada para delimitar as colunas.

Você pode estar pensando: por que não ler a planilha eletrônica diretamente? Isso é uma

possibilidade, porém a extensão das planilhas está atrelada ao sistema operacional. Por exemplo, você não terá planilhas do famoso Microsoft Excel em um sistema operacional Linux. Da mesma forma, você dificilmente terá arquivos do tipo `.odf` do pacote OpenOffice, popular em ambientes Linux, em sistemas operacionais Windows. Assim, optamos por usar arquivos de texto pleno porque são mais portáveis e, de certa forma, agnósticos ao sistema operacional e/ou software fabricante.

Um arquivo no formato `.csv` quando visualizado em um editor de texto simples como o bloco de notas, *wordpad* ou *gedit* tem o seguinte formato.

```
Sigla;Código;Município;idh;renda;alfab;
AC;1200013;Acrelândia;;136,539;73,309;
AC;1200054;Assis Brasil;;115,16;70,954;
AC;1200104;Brasiléia;;132,383;75,493;
AC;1200138;Bujari;;118,815;61,23;
AC;1200179;Capixaba;;108,164;62,283;
AC;1200203;Cruzeiro do Sul;;140,199;71,548;
AC;1200252;Epitaciolândia;;134,364;75,381;
```

Note que o caracter `;` separa as colunas. Isso significa que `;` é o caracter delimitador de campos. De forma similar um arquivo `.tsv` tem a seguinte estrutura

```
circuito    data    rank    corredor    equipe
Monza   03 September 1950   1   Giuseppe Farina Alfa Romeo
Monza   03 September 1950   2   Dorino Serafini Ferrari
Monza   03 September 1950   2   Alberto Ascari  Ferrari
Monza   03 September 1950   3   Luigi Fagioli   Alfa Romeo
Monza   03 September 1950   4   Louis Rosier    Lago
Monza   03 September 1950   5   Philippe Etancelin  Lago
```

Neste caso o espaço em branco é quem delimita as colunas. Em `R` a função `read.table()` oferece uma forma simples de importar a maioria dos formatos de dados em texto pleno. Além desta função nativa existem pacotes especializados na leitura de dados, tal como o `readr`.

O objetivo do pacote `readr` é fornecer uma maneira rápida e amigável de ler dados retangulares. Este pacote foi criado para lidar com os mais diversos tipos de formatos de dados encontrados na prática e dispõe das funções `read_csv()`, `read_tsv()`, `read_delim()`, `read_fwf()`, `read_table()` e `read_log()`. Além de serem funções bastante simples do ponto de vista do usuário, elas possuem um recurso interessante: por meio de um argumento `col_types` é possível fornecer uma lista que informa já na leitura o tipo da variável, o que poupa esforços na fase de tratamento dos dados para análise. Abaixo carregamos o pacote `readr` e visualizamos a sua documentação.

```
library(readr)
help(readr)
```

Para leitura do conjunto de dados é importante saber qual é o diretório dentro do computador onde o arquivo de interesse está gravado. Neste Capítulo leremos a base de dados direto de um endereço *web*. Porém, se não for o seu caso, basta você especificar o caminho do seu arquivo no argumento `file`. Uma recomendação é deixar o conjunto de dados no mesmo diretório do seu arquivo `.R`. Assim, você pode simplesmente fixar o diretório de trabalho como o diretório atual usando a função `setwd()` e ler o arquivo apenas especificando o nome. Caso você esteja usando o RStudio IDE na Guia Session -> Set Working Directory -> To Source File Location fará está tarefa.

Para começar, vamos importar um arquivo no formato `.txt` que está no seguinte endereço:

http://leg.ufpr.br/~wagner/ms_livro/anovareg.txt. Você pode acessar este endereço e ver que o caracter separador neste exemplo é um *tab*. Por este motivo usaremos a função do pacote `readr` destinada a conjuntos de dados separados por tabulação: a `read_tsv()`. Ao abrir o arquivo .txt nota-se que cada coluna tem um nome, por isso o argumento `col_names = TRUE` deve ser usado. O Código 1.1 ilustra como importar esta base de dados para o `R` e mostra as primeiras seis linhas.

Código 1.1. *Importando um arquivo do tipo .txt usando a função `read_tsv()`.*

```
library(readr)
url <- "http://leg.ufpr.br/~wagner/ms_livro/anovareg.txt"
dados <- read_tsv(url, col_names = TRUE)
head(dados)
```

```
## # A tibble: 6 x 4
##   cultivar  dose bloco indice
##   <chr>    <dbl> <chr>  <dbl>
## 1 Ag-1002      0 I         46
## 2 Ag-1002      0 II        48
## 3 Ag-1002      0 III       44
## 4 Ag-1002      0 IV        46
## 5 Ag-1002     60 I         48
## 6 Ag-1002     60 II        47
```

Vamos importar um outro conjunto de dados agora no formato `.csv` que está no seguinte endereço *web*: http://leg.ufpr.br/~wagner/ms_livro/reglinear.csv. Novamente, você pode abrir o arquivo em um editor de texto e ver que o caracter separador é um espaço em branco. Neste caso a função do pacote `readr` para leitura é a `read_table()`.

Código 1.2. *Importando um arquivo do tipo .csv usando a função `read_table()`.*

```
url <- "http://leg.ufpr.br/~wagner/ms_livro/reglinear.csv"
dados <- read_table(url, col_names = TRUE)
head(dados)
```

```
## # A tibble: 6 x 2
##     `"y"` `"x"`
##     <dbl> <dbl>
## 1 207318.    55
## 2 250846.    69
## 3 165755.    46
## 4 219817.    61
## 5 268582.    73
## 6 229060.    63
```

Se a leitura foi executada com sucesso você não deve receber nenhuma mensagem. O erro mais comum neste ponto é não especificar o caminho de diretórios corretamente, neste caso você vai receber uma mensagem similar a esta

```
dados <- read_table("reglinear.csv", col_names = TRUE)
```

```
## Error: 'reglinear.csv' does not exist in current working directory ('/home/wagner/Drop
```

O `R` está dizendo que não conseguiu encontrar o arquivo que você deseja ler no diretório que ele está procurando. Lembre-se: para saber onde o `R` está trabalhando use o comando `getwd()`.

Com a leitura bem sucedida, é hora de inspecionar o que foi carregado. Recomendamos usar a

função `glimpse()` para ter uma visão geral do arquivo. Assim, vamos carregar o pacote `dplyr` que contém a função `glimpse` e inspecionar o objeto `dados`.

```
library(dplyr)
glimpse(dados)
```

```
## Rows: 20
## Columns: 2
## $ `"y"` <dbl> 207318, 250846, 165755, 219817, 268582, 229060, 179098~
## $ `"x"` <dbl> 55, 69, 46, 61, 73, 63, 50, 46, 69, 62, 63, 60, 55, 70~
```

A função `glimpse()` começa mostrando quantas observações (*rows*) e quantas colunas (*cols*) a base de dados contém. Neste caso temos 20 observações (linhas) e 2 variáveis (colunas). Na sequência é apresentado o nome de cada coluna e o tipo de dado que ela representa. Neste exemplo, temos variáveis do tipo `dbl` que é a abreviação para `double`. Com o conjunto de dados disponível é o momento de arrumá-los para posterior análise.

1.2 Arrumando dados

Para analisar um conjunto de dados é importante que ele esteja arrumado, ou seja, cada coluna deve ser uma variável e cada linha uma observação. Na maioria das situações práticas os dados não estarão neste formato e você precisará trabalhar para deixá-lo desta forma. Um dos pacotes `R` mais famosos para arrumação de dados é o `tidyr`. O Código abaixo carrega o pacote `tidyr` e acessa sua documentação.

```
library(tidyr)
help(tidyr)
```

Uma situação comum é quando os dados vêm em muitas colunas. Este caso ocorre muito na tabulação de dados de experimentos nos quais se tem medidas realizadas ao longo do tempo. Por exemplo, suponha que o conjunto de dados consiste no número de casos de uma doença rara em seis grands centros urbanos nos anos de 2011, 2012 e 2013.

```
tb1 <- data.frame("cidade" = c("Curitiba", "São Paulo", "Rio de Janeiro"),
                  '2011' = c(5, 7, 3),
                  '2012' = c(6, 2, 5),
                  '2013' = c(6, 9, 7), check.names = FALSE)

tb1
```

```
##           cidade 2011 2012 2013
## 1       Curitiba    5    6    6
## 2      São Paulo    7    2    9
## 3 Rio de Janeiro    3    5    7
```

Apenas um pequeno esclarecimento, no código acima o argumento `check.names = FALSE` é apenas para permitir que o `R` use números como nomes para colunas. Em `R` essa não é uma prática recomendável. Porém, optei por utilizar neste momento por acreditar que o benefício didático é compensador.

Neste exemplo temos que as observações da variável de interesse estão em três colunas (`2011`, `2012` e `2013`). Além disso, a variável `ano`, que deveria ser uma única coluna, está no título de três outras colunas. Este é um exemplo típico de dados desorganizados que estão no formato amplo (*wide*). Nós queremos transformá-los para o formato longo (*long*), de modo que cada variável esteja em uma coluna e cada observação em uma linha.

Podemos realizar esta operação utilizando o pacote `tidyr`. Para o caso de empilhar colunas temos a função `pivot_longer()`.

```
library(magrittr)

tb1_long <- tb1 %>% pivot_longer(names_to = 'ano',
                                 values_to = 'resposta',
                                 cols = -cidade)

tb1_long
```

```
## # A tibble: 9 x 3
##   cidade         ano   resposta
##   <chr>          <chr>    <dbl>
## 1 Curitiba       2011         5
## 2 Curitiba       2012         6
## 3 Curitiba       2013         6
## 4 São Paulo      2011         7
## 5 São Paulo      2012         2
## 6 São Paulo      2013         9
## 7 Rio de Janeiro 2011         3
## 8 Rio de Janeiro 2012         5
## 9 Rio de Janeiro 2013         7
```

A sintaxe é bastante simples:

- O argumento `names_to` nomeia a coluna que vai converter os nomes das colunas para um fator.
- O argumento `values_to` nomeia a coluna que vai receber os valores empilhados das colunas de interesse.
- Por fim especificamos que nada deve ser feito com a coluna `cidade` por meio do argumento `cols = -cidade`.

Note que introduzimos um novo operador: o `%>%`, chamado de *pipe*. O objetivo do operador deste operador é encaminhar um valor para uma expressão ou chamada de função. Neste caso, utilizamos o `%>%` para dizer que na função `pivot_longer()` o objeto de interesse é o nosso *data.frame* `tb1`. É importante ressaltar que dentro da função `pivot_longer()` podemos especificar diretamente o *data.frame* por meio do argumento `data`, contudo o código fica mais organizado e elegante quando usamos o `%>%`.

Podemos também fazer a operação inversa, ou seja, desempilhar caso seja de interesse. Isso pode ser feito com a função `pivot_wider()` do `tidyr`.

```
tb1_long %>% pivot_wider(names_from =  'ano',
                         values_from = 'resposta')
```

```
## # A tibble: 3 x 4
##   cidade         `2011` `2012` `2013`
##   <chr>           <dbl>  <dbl>  <dbl>
## 1 Curitiba            5      6      6
## 2 São Paulo           7      2      9
## 3 Rio de Janeiro      3      5      7
```

Outra operação que pode ser necessária é separar uma variável em várias outras. Isso é comum quando um campo de texto é a união de várias informações. Considere o seguinte *data.frame*.

```
tb <- data.frame(US = c("US1", "US2", "US3"),
                 cidade_ano = c("Curitiba/2012", "Santos/2012", "Viçosa/2016"),
                 local = c("Curitiba-PR", "Santos-SP", "Viçosa-MG"))

tb
```

```
##    US    cidade_ano       local
## 1 US1 Curitiba/2012 Curitiba-PR
## 2 US2   Santos/2012   Santos-SP
## 3 US3   Viçosa/2016   Viçosa-MG
```

A coluna `cidade_ano` é a mistura das variáveis cidade e ano. De forma similar, a coluna `local` é a mistura de duas variáveis: município e unidade federativa (UF). Assim, precisamos separar estas informações. Note que o caracter `/` é quem separa as informações na coluna `cidade_ano` enquanto que na coluna `local` o caracter `-` é o separador.

Uma alternativa simples e que exige pouco tratamento para separação de variáveis é a função `separate()` do pacote `tidyr`. Para esta função passamos como argumentos o *data.frame*, a coluna de interesse, quais colunas novas serão criadas e qual o separador. Por exemplo, o seguinte Código

```
tb_nova1 <- tb %>% separate(col = cidade_ano,
                            into = c('Cidade', 'Ano'),
                            sep = '/')

tb_nova1
```

```
##    US   Cidade  Ano       local
## 1 US1 Curitiba 2012 Curitiba-PR
## 2 US2   Santos 2012   Santos-SP
## 3 US3   Viçosa 2016   Viçosa-MG
```

separa a coluna `cidade_ano` em outras duas colunas chamadas de `Cidade` e `Ano`, usando como caracter para separar as informações a `/`. De forma similar podemos separar as informações da coluna `local`.

```
tb_nova2 <- tb_nova1 %>% separate(col = local,
                                  into = c('Município', 'UF'),
                                  sep = '-')

tb_nova2
```

```
##    US   Cidade  Ano Município UF
## 1 US1 Curitiba 2012  Curitiba PR
## 2 US2   Santos 2012    Santos SP
## 3 US3   Viçosa 2016    Viçosa MG
```

Da mesma forma que podemos separar variáveis que estão em uma mesma coluna pode ser necessário criar novas variáveis unindo os valores de duas ou mais colunas. Por exemplo, suponha que em um *data.frame* os campos dia, mês e ano estão cada um em uma coluna. Porém, você precisa de apenas uma coluna que represente a data da observação. Assim, precisamos juntar essas três colunas em uma nova.

```
tb <- data.frame(dia = c(1, 5, 23, 16),
                 mes = c(3, 6, 2, 9),
                 ano = 2018)
```

Esta tarefa pode ser realizada com a função `unite()` do pacote `tidyr` em que especifica-se o *data.frame*, o nome da coluna que receberá a união das outras, o separador de interesse e quais colunas serão unidas. Neste caso, com o intuito de manter as variáveis originais acrescentamos o argumento `remove = FALSE`.

```
tb <- tb %>% unite(col = 'data',
                   sep = '/',
                   c('dia', 'mes', 'ano'),
                   remove = FALSE)

tb
```

```
##        data dia mes  ano
## 1  1/3/2018   1   3 2018
## 2  5/6/2018   5   6 2018
## 3 23/2/2018  23   2 2018
## 4 16/9/2018  16   9 2018
```

Podemos ainda converter esta coluna para o tipo data.

```
tb$data <- as.Date(tb$data, format =  "%d/%m/%Y")
tb
```

```
##         data dia mes  ano
## 1 2018-03-01   1   3 2018
## 2 2018-06-05   5   6 2018
## 3 2018-02-23  23   2 2018
## 4 2018-09-16  16   9 2018
```

Por fim, teremos frequentemente que lidar com dados faltantes. Algumas vezes podemos simplesmente querer remover todas as linhas que tenham algum `NA`. Outras vezes queremos preencher com algum valor específico. O pacote `tidyr` dispõe de funções bastante úteis para tratamento de dados ausentes. Veja o seguinte conjunto de dados

```
tb <- data.frame(Paciente = 1:5,
                 N_consultas = c(0, 1, 3, 1, 2),
                 N_tratamentos = c(NA, 0, 0, 2, 1),
                 N_remedios = c(NA, 1, 1, 0, 0))
```

Para excluir todas as linhas que contenham pelo menos um `NA` temos a função `drop_na()` do pacote `tidyr`.

```
drop_na(tb)
```

```
##   Paciente N_consultas N_tratamentos N_remedios
## 1        2           1             0          1
## 2        3           3             0          1
## 3        4           1             2          0
## 4        5           2             1          0
```

Para substituir os campos iguais a `NA` por algum valor o pacote `tidyr` dispõe da função `replace_na()`. Esta função recebe como argumento o *data.frame* e uma lista com o nome da variável no *dataframe* e que valor um dado ausente encontrado naquela variável deve assumir.

```
tb %>% replace_na(list(N_tratamentos = 0,
                       N_remedios = 0))
```

```
##   Paciente N_consultas N_tratamentos N_remedios
## 1        1           0             0          0
## 2        2           1             0          1
## 3        3           3             0          1
## 4        4           1             2          0
## 5        5           2             1          0
```

É importante salientar que tratar dados ausentes de forma estatisticamente coerente nem sempre é uma tarefa fácil. Assim, pense muito bem o que os dados faltantes significam no contexto que você está trabalhando e se faz sentido substituir por algum número ou mesmo excluir da base de dados.

1.3 Manipulando dados

Uma vez que os dados estejam arrumados é a hora de começar a conhecê-los. Seguindo os princípios do *tidy data* para manipular um conjunto de dados temos cinco ações essenciais:

- **Ordenar** observações de acordo com os seus valores.
- **Selecionar** variáveis de acordo com o seu nome e/ou características.
- **Filtrar** observações de acordo com os seus valores.
- **Criar/transformar** variáveis a partir de variáveis já existentes.
- **Resumir** ou **sumarizar** dados. Essa ação é a mais complexa e em geral envolve o uso de medidas estatísticas como média, mediana, desvio-padrão entre outros para criar resumo dos dados.

Existem diversas alternativas em `R` para manipulação de dados. Uma delas é o pacote `dplyr`. No código abaixo carregamos o pacote `dplyr` e acessamos a sua documentação.

```
library(dplyr)
help(dplyr)
```

Para ilustrar o uso de cada uma das ações de manipulação vamos considerar os dados apresentados na Figura 1.2. O exemplo consiste em um estudo longitudinal composto por sete pacientes de duas unidades de saúde que foram avaliados em três consultas com relação a severidade de alguma doença. A severidade foi avaliado como um nota entre 0 e 10. Por fim, a variável número de medicamentos que o paciente usou durante o período avaliado é registrado na coluna `medicamentos`. Obviamente este é um exemplo simples e puramente didático. Porém, com base nestes dados é possível ilustrar as principais tarefas da etapa de manipulação de dados.

Podemos entrar com esses dados diretamente em `R` conforme abaixo.

```
df1 <- data.frame(
    codigo = c(256, 487, 965,
                  125, 458, 874, 963),
    paciente = c("João", "Vanessa", "Tiago",
              "Luana", "Gisele", "Pedro",
              "André"),
    US = c("US1", "US1", "US2", "US2",
             "US2", "US1", "US2"),
    severidade1 = c(8, 7.5, 9.5, 7, 4.5, 5.5, 3),
    severidade2 = c(9, 7.5, 8, 8.5, 5, 7.5, NA),
```

	codigo	paciente	US	severidade1	severidade2	severidade3	medicamentos
1	256	João	US1	8	9	8	4
2	487	Vanessa	US1	7,5	7,5	7,5	4
3	965	Tiago	US2	9,5	8	7,5	0
4	125	Luana	US2	7	8,5	5	8
5	458	Gisele	US2	4,5	5		2
6	874	Pedro	US1	5,5	7,5	9	0
7	963	André	US2	3		3	5

Figure 1.2: Uma tabela com dados fictícios sobre um experimento longitudinal.

```
    severidade3 = c(8, 7.5, 7.5, 5, NA, 9, 3),
    medicamentos = c(4, 4, 0, 8, 2, 0, 5))
```

A ação mais simples é a de ordenação que pode ser feita de acordo com os valores de uma ou mais variáveis.

Por exemplo, podemos ordenar os pacientes pelo seu código. No `dplyr` a função destinada para ordenação é a `arrange()`.

```
df1 %>% arrange(codigo)
```

```
##   codigo paciente  US severidade1 severidade2 severidade3
## 1    125    Luana US2         7.0         8.5         5.0
## 2    256     João US1         8.0         9.0         8.0
## 3    458   Gisele US2         4.5         5.0          NA
## 4    487  Vanessa US1         7.5         7.5         7.5
## 5    874    Pedro US1         5.5         7.5         9.0
## 6    963    André US2         3.0          NA         3.0
## 7    965    Tiago US2         9.5         8.0         7.5
##   medicamentos
## 1            8
## 2            4
## 3            2
## 4            4
## 5            0
## 6            5
## 7            0
```

Note que a variável `codigo` foi ordenada de forma crescente. Para ordenar de maneira descrescente:

```
df1 %>% arrange(desc(codigo))
```

```
##   codigo paciente  US severidade1 severidade2 severidade3
## 1    965    Tiago US2         9.5         8.0         7.5
## 2    963    André US2         3.0          NA         3.0
## 3    874    Pedro US1         5.5         7.5         9.0
## 4    487  Vanessa US1         7.5         7.5         7.5
## 5    458   Gisele US2         4.5         5.0          NA
```

```
## 6    256     João US1         8.0         9.0         8.0
## 7    125    Luana US2         7.0         8.5         5.0
##   medicamentos
## 1            0
## 2            5
## 3            0
## 4            4
## 5            2
## 6            4
## 7            8
```

Podemos ordenar por duas ou mais colunas.

```
df1 %>% arrange(US, severidade1)
```

```
##   codigo paciente  US severidade1 severidade2 severidade3
## 1    874    Pedro US1         5.5         7.5         9.0
## 2    487  Vanessa US1         7.5         7.5         7.5
## 3    256     João US1         8.0         9.0         8.0
## 4    963    André US2         3.0          NA         3.0
## 5    458   Gisele US2         4.5         5.0          NA
## 6    125    Luana US2         7.0         8.5         5.0
## 7    965    Tiago US2         9.5         8.0         7.5
##   medicamentos
## 1            0
## 2            4
## 3            4
## 4            5
## 5            2
## 6            8
## 7            0
```

A próxima ação é de seleção e pode ser feita de várias formas:

- Pelo nome das colunas.
- Pela posição das linhas ou das colunas.
- Condicional ao valor de uma ou mais variáveis.
- Por extremidades cabeça (*head*) e cauda (*tail*).

Suponha que temos interesse apenas nas colunas `paciente`, `US` e `medicamentos`. Podemos selecionar as colunas pelos seus nomes. Utilizando a função `select` do `dplyr`, temos

```
df1 %>% select(c("paciente", "US", "medicamentos"))
```

```
##   paciente  US medicamentos
## 1     João US1            4
## 2  Vanessa US1            4
## 3    Tiago US2            0
## 4    Luana US2            8
## 5   Gisele US2            2
## 6    Pedro US1            0
## 7    André US2            5
```

Pelas posições das colunas.

```
df1 %>% select(c(2,3,7))
```

```
##   paciente  US medicamentos
## 1     João US1            4
## 2  Vanessa US1            4
## 3    Tiago US2            0
## 4    Luana US2            8
## 5   Gisele US2            2
## 6    Pedro US1            0
## 7    André US2            5
```

Podemos selecionar apenas os paciente que usam mais de dois medicamentos usando a função `filter()` do `dplyr`.

```
df1 %>% filter(medicamentos > 2 )
```

```
##   codigo paciente  US severidade1 severidade2 severidade3
## 1    256     João US1         8.0         9.0         8.0
## 2    487  Vanessa US1         7.5         7.5         7.5
## 3    125    Luana US2         7.0         8.5         5.0
## 4    963    André US2         3.0          NA         3.0
##   medicamentos
## 1            4
## 2            4
## 3            8
## 4            5
```

Podemos também selecionar pelo número das linhas usando a função `slice()` do `dplyr`.

```
df1 %>% slice(3:5)
```

```
##   codigo paciente  US severidade1 severidade2 severidade3
## 1    965    Tiago US2         9.5         8.0         7.5
## 2    125    Luana US2         7.0         8.5         5.0
## 3    458   Gisele US2         4.5         5.0          NA
##   medicamentos
## 1            0
## 2            8
## 3            2
```

Além disso podemos selecionar apenas as extremidades. Cabeça (*head*) ou cauda (*tail*). O número de observações pode ser alterado pelo argumento `n`.

```
# Cabeça
df1 %>% slice_head(n = 3)
```

```
##   codigo paciente  US severidade1 severidade2 severidade3
## 1    256     João US1         8.0         9.0         8.0
## 2    487  Vanessa US1         7.5         7.5         7.5
## 3    965    Tiago US2         9.5         8.0         7.5
##   medicamentos
## 1            4
## 2            4
## 3            0
```

```
# Cauda
df1 %>% slice_tail(n = 3)
```

```
##   codigo paciente  US severidade1 severidade2 severidade3
## 1    458   Gisele US2         4.5         5.0          NA
## 2    874    Pedro US1         5.5         7.5           9
## 3    963    André US2         3.0          NA           3
##   medicamentos
## 1            2
## 2            0
## 3            5
```

Entre as diversas ações possíveis, a de filtrar observações de acordo com suas características é uma das mais importantes. Já vimos a função `filter()` do `dplyr`, que permite fazer filtros ou obte r subconjuntos que tenham características de interesse. Similar a ação de seleção, a ação de filtro pode ser realizada por uma ou mais características. Por exemplo, podemos selecionar só os pacientes da unidade de saúde `US1`.

```
df1 %>% filter(US == "US1")
```

```
##   codigo paciente  US severidade1 severidade2 severidade3
## 1    256     João US1         8.0         9.0         8.0
## 2    487  Vanessa US1         7.5         7.5         7.5
## 3    874    Pedro US1         5.5         7.5         9.0
##   medicamentos
## 1            4
## 2            4
## 3            0
```

Outro exemplo é selecionar pacientes em que a soma das severidades das consultas 1, 2 e 3 seja maior que 18.

```
df1 %>% filter(US == "US1" &  (severidade1 + severidade2 + severidade3) > 18)
```

```
##   codigo paciente  US severidade1 severidade2 severidade3
## 1    256     João US1         8.0         9.0         8.0
## 2    487  Vanessa US1         7.5         7.5         7.5
## 3    874    Pedro US1         5.5         7.5         9.0
##   medicamentos
## 1            4
## 2            4
## 3            0
```

Pacientes que usam algum medicamento.

```
df1 %>% filter(medicamentos != 0)
```

```
##   codigo paciente  US severidade1 severidade2 severidade3
## 1    256     João US1         8.0         9.0         8.0
## 2    487  Vanessa US1         7.5         7.5         7.5
## 3    125    Luana US2         7.0         8.5         5.0
## 4    458   Gisele US2         4.5         5.0          NA
## 5    963    André US2         3.0          NA         3.0
##   medicamentos
## 1            4
## 2            4
```

```
## 3            8
## 4            2
## 5            5
```

Pelo nome dos pacientes.

```
df1 %>% filter(paciente %in% c("Aline", "Vanessa"))
```

```
##   codigo paciente  US severidade1 severidade2 severidade3
## 1    487  Vanessa US1         7.5         7.5         7.5
##   medicamentos
## 1            4
```

O operador `%in%` implica na seleção de qualquer um que se encaixar em `c("Aline", "Vanessa")`. Como não temos nenhuma Aline, a saída retornou apenas a linha da paciente Vanessa.

As operações de criar/transformar nos permitem por exemplo renomear as colunas do conjunto de dados. Para renomear as colunas de um *data.frame* basta atribuir novos nomes. Para esta tarefa o `dplyr` possui a função `rename()`.

```
df1 <- df1 %>% rename(cod = 'codigo',
                      pac = 'paciente',
                      US = 'US',
                      s1 = 'severidade1',
                      s2 = 'severidade2',
                      s3 = 'severidade3',
                      med = 'medicamentos')
```

Trocar a ordem das colunas para que as severidades apareçam primeiro pode ser feito tanto pelo nome como pelos números associados às colunas.

```
df1[,c(4,5,6,1,2,3)]
```

```
##    s1  s2  s3 cod     pac  US
## 1 8.0 9.0 8.0 256    João US1
## 2 7.5 7.5 7.5 487 Vanessa US1
## 3 9.5 8.0 7.5 965   Tiago US2
## 4 7.0 8.5 5.0 125   Luana US2
## 5 4.5 5.0  NA 458  Gisele US2
## 6 5.5 7.5 9.0 874   Pedro US1
## 7 3.0  NA 3.0 963   André US2
```

A ação de transformação consiste em criar novas variáveis a partir das variáveis existentes. Por exemplo, podemos calcular a severidade média dos pacientes. Para isso é possível utilizar a função `mutate()` do `dplyr`

```
df1 %>% mutate(media = (s1 + s2 + s3)/3)
```

```
##   cod     pac  US  s1  s2  s3 med media
## 1 256    João US1 8.0 9.0 8.0   4  8.33
## 2 487 Vanessa US1 7.5 7.5 7.5   4  7.50
## 3 965   Tiago US2 9.5 8.0 7.5   0  8.33
## 4 125   Luana US2 7.0 8.5 5.0   8  6.83
## 5 458  Gisele US2 4.5 5.0  NA   2    NA
## 6 874   Pedro US1 5.5 7.5 9.0   0  7.33
## 7 963   André US2 3.0  NA 3.0   5    NA
```

Note que, onde os pacientes não tem severidade, o cálculo da média resultou em `NA`. Vamos supor que se o paciente não realizou a consulta a severidade dele será zero.

```
df1 <- df1 %>% replace_na(list(s1 = 0,
                               s2 = 0,
                               s3 = 0))

df1
```

```
##   cod     pac  US  s1  s2  s3 med
## 1 256    João US1 8.0 9.0 8.0   4
## 2 487 Vanessa US1 7.5 7.5 7.5   4
## 3 965   Tiago US2 9.5 8.0 7.5   0
## 4 125   Luana US2 7.0 8.5 5.0   8
## 5 458  Gisele US2 4.5 5.0 0.0   2
## 6 874   Pedro US1 5.5 7.5 9.0   0
## 7 963   André US2 3.0 0.0 3.0   5
```

```
df1 <- df1 %>% mutate(media = (s1 + s2 + s3)/3)

df1
```

```
##   cod     pac  US  s1  s2  s3 med media
## 1 256    João US1 8.0 9.0 8.0   4  8.33
## 2 487 Vanessa US1 7.5 7.5 7.5   4  7.50
## 3 965   Tiago US2 9.5 8.0 7.5   0  8.33
## 4 125   Luana US2 7.0 8.5 5.0   8  6.83
## 5 458  Gisele US2 4.5 5.0 0.0   2  3.17
## 6 874   Pedro US1 5.5 7.5 9.0   0  7.33
## 7 963   André US2 3.0 0.0 3.0   5  2.00
```

Podemos criar uma coluna para classificar a severidaed. Por exemplo, se o paciente tem severidade (média > 7) classificamos como `alta`, como `média` se ($40 <$ média < 70) ou `baixa` (média < 40) usando a função `cut()` temos

```
breaks <- c(0, 4, 7, 10)

df1$classificacao <- cut(df1$media,
                    breaks = breaks,
                    labels = c("Baixa", "Média", "Alta"))
df1
```

```
##   cod     pac  US  s1  s2  s3 med media classificacao
## 1 256    João US1 8.0 9.0 8.0   4  8.33          Alta
## 2 487 Vanessa US1 7.5 7.5 7.5   4  7.50          Alta
## 3 965   Tiago US2 9.5 8.0 7.5   0  8.33          Alta
## 4 125   Luana US2 7.0 8.5 5.0   8  6.83         Média
## 5 458  Gisele US2 4.5 5.0 0.0   2  3.17         Baixa
## 6 874   Pedro US1 5.5 7.5 9.0   0  7.33          Alta
## 7 963   André US2 3.0 0.0 3.0   5  2.00         Baixa
```

A última ação que vamos discutir é a de resumo ou sumarização. Esta ação está muito ligada com a ideia de dados no formato amplo. No exemplo dos pacientes o conjunto de dados está no formato amplo. Se desejarmos calcular a média da severidade para cada consulta precisamos percorrer as colunas `severidade1`, `severidade2` e `severidade3` uma a uma calculando a média. Se os dados estiverem no formato longo poderíamos mais facilmente usar a ideia de resumir

uma variável condicional aos valores de uma outra, neste caso: a severidade.

Usando a função `pivot_longer()` podemos facilmente passar um conjunto de dados do formato amplo para longo. Note que vamos remover as colunas `cod`, `rem``,mediaeclassificacao‘ antes de fazer a manipulação.

```
df1_temp <- df1 %>% select(-c(cod, med, media, classificacao))

df1_long <- df1_temp %>% pivot_longer(names_to = 'consulta',
                                      values_to = 'valor',
                                      cols = -c(pac, US))

df1_long
```

```
## # A tibble: 21 x 4
##    pac     US    consulta valor
##    <chr>   <chr> <chr>    <dbl>
##  1 João    US1   s1         8
##  2 João    US1   s2         9
##  3 João    US1   s3         8
##  4 Vanessa US1   s1         7.5
##  5 Vanessa US1   s2         7.5
##  6 Vanessa US1   s3         7.5
##  7 Tiago   US2   s1         9.5
##  8 Tiago   US2   s2         8
##  9 Tiago   US2   s3         7.5
## 10 Luana   US2   s1         7
## # ... with 11 more rows
```

Agora suponha que queremos saber qual é a severidade média por consulta. Estamos fazendo um resumo da coluna `valor` porém de acordo com os valores da coluna `consulta`. Informalmente, dizemos que estamos agrupando pela coluna `consulta` ou fazendo um *group by* por consulta. Este resultado pode ser obtido por meio da função `summarise()` do `dplyr`

```
df1_long %>%
    group_by(consulta) %>%
    summarise(media = mean(valor))
```

```
## # A tibble: 3 x 2
##   consulta media
##   <chr>    <dbl>
## 1 s1        6.43
## 2 s2        6.5
## 3 s3        5.71
```

Podemos agrupar por mais de uma coluna. Por exemplo, a severidade média de cada unidade de saúde para cada consulta.

```
df1_long %>%
    group_by(US, consulta) %>%
    summarise(media = mean(valor))
```

```
## # A tibble: 6 x 3
## # Groups:   US [2]
##   US    consulta media
##   <chr> <chr>    <dbl>
```

```
## 1 US1   s1        7
## 2 US1   s2        8
## 3 US1   s3        8.17
## 4 US2   s1        6
## 5 US2   s2        5.38
## 6 US2   s3        3.88
```

Outra opção é calcular mais do que uma medida resumo. No caso da função `summarise()` basta acrescentar uma linha especificando que temos interesse em obter outra medida.

```
df1_long %>%
    group_by(US, consulta) %>%
    summarise(media = mean(valor),
              sd = sd(valor))
```

```
## # A tibble: 6 x 4
## # Groups:   US [2]
##   US    consulta media    sd
##   <chr> <chr>    <dbl> <dbl>
## 1 US1   s1         7    1.32
## 2 US1   s2         8    0.866
## 3 US1   s3         8.17 0.764
## 4 US2   s1         6    2.86
## 5 US2   s2         5.38 3.90
## 6 US2   s3         3.88 3.17
```

Este tipo de tarefa é também chamada de *split-apply-combine*. Primeiro nós dividimos (*split*) o conjunto de dados pelas variáveis unidade de saúde e consulta. Para cada pedaço foram calculadas as medidas resumo (*apply*). Finalmente, combinamos os resultados em um novo *data.frame* (*combine*).

1.4 Combinando dados

Em algumas situações pode ser necessário combinar diversos conjuntos de dados. A situação mais comum é quando você já tem um conjunto de dados, como por exemplo o nosso `df1`, e um novo conjunto de dados é disponibilizado. Assim, precisamos combinar os dados que já temos com o novo. Suponha que o novo conjunto de dados é o seguinte.

```
df2 <- data.frame(
    codigo = c(505, 658, 713),
    paciente = c("Bia", "Carlos", "Cris"),
    unidade_saude = c("US3", "US3", "US3"),
    severidade1 = c(6.5, 7.5, 7.5),
    severidade2 = c(8.5, 8.0, 9.0),
    medicamentos = c(0, 0, 2))
```

Para combinar o novo conjunto de dados com o antigo a primeira coisa a fazer é deixá-los compatíveis. Isso significa que devem ter os mesmos dados e as colunas devem ter os mesmos nomes. Como fizemos diversos cálculos com o `df1`, vamos recriá-lo.

```
df1 <- data.frame(
    codigo = c(256, 487, 965,
                   125, 458, 874, 963),
    paciente = c("João", "Vanessa", "Tiago",
```

```
            "Luana", "Gisele", "Pedro",
            "André"),
    unidade_saude = c("US1", "US1", "US2", "US2",
             "US2", "US1", "US2"),
    severidade1 = c(8, 7.5, 9.5, 7, 4.5, 5.5, 3),
    severidade2 = c(9, 7.5, 8, 8.5, 5, 7.5, NA),
    severidade3 = c(8, 7.5, 7.5, 5, NA, 9, 3),
    medicamentos = c(4, 4, 0, 8, 2, 0, 5))
```

Note que no conjunto `df2` não temos uma coluna chamada de `severidade3`. Assim, primeiro precisamos criar esta coluna.

```
df2$severidade3 <- NA
```

Uma vez que os conjuntos de dados são compatíveis, podemos simplesmente concatenar suas linhas usando a função `rbind()`.

```
df <- rbind(df1, df2)
df
```

```
##    codigo paciente unidade_saude severidade1 severidade2 severidade3
## 1     256     João           US1         8.0         9.0         8.0
## 2     487  Vanessa           US1         7.5         7.5         7.5
## 3     965    Tiago           US2         9.5         8.0         7.5
## 4     125    Luana           US2         7.0         8.5         5.0
## 5     458   Gisele           US2         4.5         5.0          NA
## 6     874    Pedro           US1         5.5         7.5         9.0
## 7     963    André           US2         3.0          NA         3.0
## 8     505      Bia           US3         6.5         8.5          NA
## 9     658   Carlos           US3         7.5         8.0          NA
## 10    713     Cris           US3         7.5         9.0          NA
##    medicamentos
## 1             4
## 2             4
## 3             0
## 4             8
## 5             2
## 6             0
## 7             5
## 8             0
## 9             0
## 10            2
```

Por fim, uma outra operação que aparece com frequência ao analisar dados é a de juntar/parear duas tabelas que possuem uma chave primária. Essas operações são geralmente chamadas de **junções** (*join*) e existem diversas versões, as mais comuns são:

- Junção por interseção (*inner join*).
- Junção por união (*full join*).
- Junção à esquerda (*left join*).
- Junção à direita (*right join*).
- Existem também os chamados *exclusive joins*.

Cada uma destas versões é ilustrada na Figura **??**.

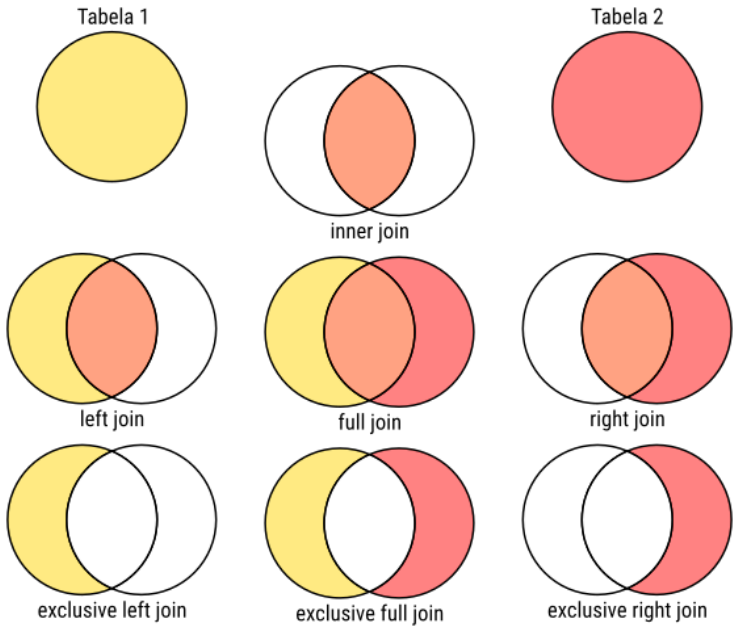


Figure 1.3: Tipos de junções de tabelas ilustrado com diagramas de Veen.

Para praticarmos este tipo de operação considere o seguinte conjunto de dados.

```
df_extra <- data.frame(
    "codigo" = c(256, 965, 285, 125, 874, 321, 669, 963),
    "paciente" = c("João", "Tiago", "Tiago", "Luana",
                "Pedro", "Mia", "Luana", "André"),
    "idade" = c(18, 18, 22, 21, 19, 18, 19, 20),
    "assist_social" = c("S", "N", "N", "S", "N", "N", "S", "N"))
```

O pacote `dplyr()` possui funções específicas para cada tipo de junção

```
# Left join
left_join(df1, df_extra, by = "codigo")

# Right join
right_join(df1, df_extra, by = "codigo")

# inner join
inner_join(df1, df_extra, by = "codigo")

# full join
full_join(df1, df_extra, by = "codigo")
```

A diferença entre os diferentes *join* é bastante simples:

- *left join*: mantém todos os registros da tabela a esquerda (`df1`) e completa com os correspondentes da tabela da direita (`df_extra`).
- *right join*: mantém todos os registros da tabela da direita (`df_extra`) e completa com os dados da tabela da esquerda (`df1`).

- *inner join*: mantém apenas os registros que estão em ambas tabelas.
- *full join*: mantém todos os registros e preenche com NA onde não tem correspondência entre as tabelas.

Assim, passamos pelas principais tarefas de manipulação de dados. Com os dados organizados e devidamente manipulado ou resumido o último passo é exportar para consumo em alguma outra ferramenta computacional. Esse é o assunto da nossa próxima seção.

1.5 Exportando dados

Após realizar as análises é bastante provável que você precise exportar um conjunto de dados organizado ou mesmo uma tabela com estatísticas. O `R` tem uma longa lista de funções para exportar dados. Assim como leitura, o pacote `readr` dispõe de uma série de funções de escrita.

A forma mais comum é exportar um arquivo de texto pleno nos formatos `.csv` ou `.txt`. Para esta tarefa as funções `write_tsv()` e `write_csv()` podem ser usadas. Podemos exportar o resultado da junção das tabelas `df1` e `df_extra` como o resultado final de nossa análise, conforme ilustra o Código 1.3.

Código 1.3. *Exemplo de uso da função `write_csv()` para exportar arquivos de texto pleno.*

```
final <- inner_join(df1,
                    df_extra,
                    by = "matricula")

write_csv(final,
          file = "Nome_do_arquivo.csv")
```

Os argumentos básicos são o *data.frame* que você quer salvar em disco e o nome desejado. A maioria dos argumentos de escrita são iguais ou muito similares aos argumentos de leitura. Assim, exportação de dados não gera mais complexidade e tudo que você já aprendeu para leitura se aplica imediatamente.

Sem dúvida, exportar dados em texto pleno é muito fácil e simples. No entanto, existem algumas desvantagens. A principal é que as classes das colunas podem ser perdidas ao ler novamente o conjunto de dados em `R`. Por ser um texto pleno o arquivo não carrega metadados que foram criados em `R`. Caso seja de interesse, é possível salvar o próprio objeto `R` que contém o seu *data.frame* usando a função `save()`.

```
save(final,
     file = "Nome_do_arquivo.RData")
```

A principal vantagem é que tudo que você fez no objeto `final` será mantido. Outra vantagem é que você pode salvar outros formatos de dados como listas, matrizes e arrays. Para carregar novamente o *data.frame* você deve usar a função `load()`.

```
load("Nome_do_arquivo.RData")
```

Caso você tenha um objeto mais complicado, como por exemplo uma série de listas, a função `dput()` é uma opção interessante. Ela permite salvar um objeto `R` como um texto simples que pode ser copiado e colado.

```
dput(final)
```

É possível também escrever um arquivo com extensão `.R` que pode ser posteriormente carregado no `R`.

```
dput(final,
     file = "Nome_do_arquivo.R")
```

Para carregar os dados basta usar a função `dget()`.

```
dget("Nome_do_arquivo.R")
```

Essa estratégia pode ser usada para múltiplos objetos usando a função `dump()`.

```
dump(c("final", "df1"),
     file = "Nome_do_arquivo.R")
```

Que agora deve ser carregado em `R` usando a função `source()`.

```
source("Nome_do_arquivo.R")
```

Um outro formato que o `R` exporta é o formato binário com a extensão `.rda`. As vantagens deste formato são que ocupa pouco espaço em disco e preserva as propriedades do objeto `R`.

```
save(final,
     file = "Nome_do_arquivo.rda")
```

Neste caso, podemos usar novamente a função `load()` para fazer a leitura em `R`.

Por fim, você pode salvar todos os objetos e instruções que você usou em sua sessão `R` usando a função `save.image()`.

Lembre-se: sempre que você não especificar o caminho de escrita explicitamente, o `R` vai escrever no diretório de trabalho atual. Algumas funções úteis para você ter controle de onde o `R` está escrevendo os arquivos são:

- `file.info()`: mostra o tamanho do arquivo, data de criação e outros detalhes do arquivo.
- `dir()`: mostra todos os arquivos presentes em um diretório.
- `file.exists()`: verifica se um arquivo já existe (`TRUE`) ou não (`FALSE`) no diretório de trabalho.
- `getwd()` e `setwd()`: verifica e altera o diretório de trabalho, respectivamente.

Agora que temos todos os ingredientes para lidar com dados em `R` vamos praticar em um exemplo com dados reais extraídos do site do Datasus.

1.6 Praticando: Mortalidade por causas externas

Para fecharmos este Capítulo vamos apresentar uma análise completa que vai desde a leitura, organização, manipulação e resumo de dados até a apresentação por meio de tabelas.

Nosso exemplo corresponde a um conjunto de dados coletado do site [tabsus] (http://tabnet.datasus.gov.br/cgi/menu_tabnet_php.htm), mantido pelo Ministério da Saúde. Os dados referem-se a mortalidade por causas externas por faixa etária para todos os municipios brasileiros no ano de 2019. O conjunto de dados está disponível no link no formato `.csv` (*comma-separated value*).

Os dados foram mantidos da forma mais crua possível para que você vivencie todos os desafiosd das etapas de importar, manipular, analisar e apresentar os resultados de uma análise de dados.

1.6.1 Importando o conjunto de dados

O arquivo com a conjunto de dados que desejamos importar é chamado de `dados.csv`. Sugiro que o leitor abra o arquivo no navegador para identificar que o caracter separador de colunas é o ";" Além disso, um aspecto interessante é que municipios em que não teve eventos em uma determinada faixa etária apresenta o caracter "-." Assim, devemos informar ao `R` minimamente:

- Qual é o caracter separador de colunas, argumento `delim = ";"`.
- Qual é o caracter indica dados ausentes, argumento `na = "-"`.
- Se o conjunto de dados contém nomes das colunas, argumento `col_names = TRUE`.

Novamente vamos usar o pacote `readr` e a função `read_delim` que nos permite facilmente especificar esses argumentos. O Código 1.4 ilustra como importar o conjunto de dados para o `R`.

Código 1.4. *Importando o conjunto de dados.*

```
dados <- read_delim("data/dados.csv", col_names = TRUE, delim = ";", na = "-")
```

Se a leitura foi executada com sucesso você deve receber uma mensagem indicando a especificação das colunas, bem como o tipo e nome de cada uma. Assim, é hora de inspecionar o que foi carregado. Recomendamos usar a função `glimpse()` para ter uma visão geral do arquivo de dados.

```
glimpse(dados)
```

```
## Rows: 5,598
## Columns: 15
## $ Município         <chr> "110001 Alta Floresta D'Oeste", "110037 Alt~
## $ `Menor 1 ano`     <dbl> NA, NA, NA, NA, NA, NA, NA, NA, 1, NA, 2, N~
## $ `1 a 4 anos`      <dbl> NA, NA, NA, NA, 3, NA, NA, NA, 4, NA, NA, N~
## $ `5 a 9 anos`      <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
## $ `10 a 14 anos`    <dbl> 1, NA, 2, NA, NA, NA, NA, NA, NA, NA, NA, N~
## $ `15 a 19 anos`    <dbl> 2, 1, 2, 1, 7, 4, 1, NA, 3, NA, 2, NA, NA, ~
## $ `20 a 29 anos`    <dbl> 4, 3, 5, NA, 26, 8, NA, NA, 15, NA, 3, NA, ~
## $ `30 a 39 anos`    <dbl> 2, 3, NA, 5, 20, 3, NA, 2, 11, 1, 6, NA, 2,~
## $ `40 a 49 anos`    <dbl> 7, NA, 2, NA, 15, 6, 1, 1, 15, 3, 6, NA, NA~
## $ `50 a 59 anos`    <dbl> 1, 1, 3, 3, 6, 3, NA, NA, 6, NA, 5, NA, 5, ~
## $ `60 a 69 anos`    <dbl> NA, 1, 3, 1, 10, 4, NA, 1, 3, 1, 2, NA, 1, ~
## $ `70 a 79 anos`    <dbl> 2, 1, NA, 1, 4, NA, 1, NA, 4, NA, NA, NA, N~
## $ `80 anos e mais`  <dbl> 1, 1, NA, NA, 3, NA, 1, NA, 7, NA, NA, NA, ~
## $ `Idade ignorada`  <dbl> NA, NA, NA, NA, NA, 1, NA, NA, 1, NA, 1, NA~
## $ Total             <dbl> 20, 11, 17, 11, 94, 29, 4, 4, 70, 5, 27, NA~
```

A função `glimpse()` começa mostrando quantas observações e quantas colunas a base de dados contém. Neste caso temos 5598 observações (linhas) e 15 variáveis (colunas). Na sequência é apresentado o nome de cada coluna e o tipo de dado que ela representa. Neste exemplo, temos variáveis do tipo `chr` e `dbl` que são abreviações dos tipos de vetores `character` e `double`. Apesar do nome das colunas ser auto explicativo é sempre interessante manter um dicionário da sua base de dados. Um dicionário nada mais é do que uma lista com o nome de cada coluna e o que ela representa. Por exemplo,

- `Municipio`: código e nome do município onde o evento ocorreu.
- `Menor de 1 ano`: Número de mortes por causas externas na faixa etária Menor de 1 ano.
- `1 a 4 anos`: Número de mortes por causas externas na faixa etária 1 a 4 anos.
- `5 a 9 anos`: Número de mortes por causas externas na faixa etária 5 a 9 anos.
- `10 a 14 anos`: Número de mortes por causas externas na faixa etária 10 a 14 anos.

- `15 a 19 anos`: Número de mortes por causas externas na faixa etária 15 a 19 anos.
- `20 a 29 anos`: Número de mortes por causas externas na faixa etária 20 a 29 anos.
- `30 a 39 anos`: Número de mortes por causas externas na faixa etária 30 a 39 anos.
- `40 a 49 anos`: Número de mortes por causas externas na faixa etária 40 a 49 anos.
- `50 a 59 anos`: Número de mortes por causas externas na faixa etária 50 a 59 anos.
- `60 a 69 anos`: Número de mortes por causas externas na faixa etária 60 a 69 anos.
- `70 a 79 anos`: Número de mortes por causas externas na faixa etária 70 a 79 anos.
- `80 anos e mais`: Número de mortes por causas externas na faixa etária 80 anos e mais.
- `Idade ignorada`: Número de mortes por causas externas onde a Idade foi ignorada.
- `Total`: Soma do número de mortes por causas externas em todas as faixas etárias.

Esse é um conjunto de dados bastante típico e ao se deparar com um conjunto de dados como este precisamos nos perguntar o que queremos mostrar/saber. Nem sempre o objetivo é claro e precisamos investigar os aspectos dos dados para saber quais são as possibilidades. É neste momento que precisamos saber como arrumar e manipular a base de dados para entendê-la e investigar possíveis análises.

1.6.2 Preparando a base de dados

Dado a estrutura da base de dados vemos claramente que os principios do *tidy data* não são respeitados em ao menos dois pontos:

1. A coluna `Municipio` contém duas informações, o código e nome do município. Assim, precisamos particionar essa informação.
2. A variável faixa etária está como nome de colunas ao invés de em um coluna. Novamente precisamos arrumar essa informação para tê-la em apenas uma coluna. Neste sentido estamos com um conjunto de dados no formato amplo.

Para começar vamos particionar a informação da coluna `Municipio`. Note que neste exemplo temos um desafio adicional uma vez que o separador das informações código e nome do município é o espaço em branco. Porém, para municípios com nome composto como por exemplo, Alto Alegre dos Parecis se usarmos o argumento `sep = ' '` da função `separate` como fizemos na Subsecção XXXX não teremos o resultado desejado.

Assim, vamos precisar lidar com algumas operações relacionadas com *strings*. De forma geral o termo *strings* será usado para se referir a campos com texto. O *tidyverse* tem um pacote especifico para lidar com *strings* o `stringr`. A ideia é simples: primeiro quebramos a *string* completa nos espaços em branco e depois especificamos quantas partes retornar. Neste caso queremos duas partes, a primeira será o código do município e a segunda o nome do município.

A função `str_split` do pacote `stringr` é adequada para esta tarefa. Por *default* a função retorna uma lista com os valores. Como queremos construir um novo *data.frame* com os valores do código e nome do município vamos usar o argumento `simplify = TRUE` que é uma forma de ao invés de retornar uma lista já retornar os valores do código e nome do município cada um em uma coluna.

Porém, o objeto retornado será da classe `matrix`. Assim, devemos convertê-lo para *data.frame* e renomear suas colunas para então juntá-lo com o conjunto de dados original. O Código 1.5 faz a separação da coluna `Município`, guarda o resultado em um objeto chamado `dados_temp` e renomeia as suas colunas.

Código 1.5. *Separando uma string em duas partes, criando um novo data.frame e renomeando suas colunas.*

```
library(stringr)
dados_temp <- data.frame(str_split(dados$Município, " ", 2, simplify = TRUE))
```

```
dados_temp <- dados_temp %>% rename("Codigo" = X1,
                                    "Nome" = X2)
```

Por fim, podemos juntar as colunas `Codigo` e `Nome` como o *data frame* original. A função `bind_cols` faz essa tarefa.

```
dados <- bind_cols(dados_temp, dados)
dados %>% slice_head(n = 5)
```

```
##   Codigo                     Nome                       Município
## 1 110001   Alta Floresta D'Oeste   110001 Alta Floresta D'Oeste
## 2 110037 Alto Alegre dos Parecis 110037 Alto Alegre dos Parecis
## 3 110040            Alto Paraíso            110040 Alto Paraíso
## 4 110034        Alvorada D'Oeste        110034 Alvorada D'Oeste
## 5 110002               Ariquemes               110002 Ariquemes
##   Menor 1 ano 1 a 4 anos 5 a 9 anos 10 a 14 anos 15 a 19 anos
## 1          NA         NA         NA            1            2
## 2          NA         NA         NA           NA            1
## 3          NA         NA         NA            2            2
## 4          NA         NA         NA           NA            1
## 5          NA          3         NA           NA            7
##   20 a 29 anos 30 a 39 anos 40 a 49 anos 50 a 59 anos 60 a 69 anos
## 1            4            2            7            1           NA
## 2            3            3           NA            1            1
## 3            5           NA            2            3            3
## 4           NA            5           NA            3            1
## 5           26           20           15            6           10
##   70 a 79 anos 80 anos e mais Idade ignorada Total
## 1            2              1             NA    20
## 2            1              1             NA    11
## 3           NA             NA             NA    17
## 4            1             NA             NA    11
## 5            4              3             NA    94
```

O próximo passo é transformar o nosso conjunto de dados do formato amplo para longo empilhando as colunas relacionadas com as faixas etárias.

```
dados_long <- dados %>% pivot_longer(names_to = 'faixa_etaria',
                                     values_to = 'contagem',
                                     cols = -c(Codigo, Nome, Município))
dados_long %>% slice_head(n = 5)
```

```
## # A tibble: 5 x 5
##   Codigo Nome                  Município         faixa_etaria contagem
##   <chr>  <chr>                 <chr>             <chr>           <dbl>
## 1 110001 Alta Floresta D'Oeste 110001 Alta Flor~ Menor 1 ano        NA
## 2 110001 Alta Floresta D'Oeste 110001 Alta Flor~ 1 a 4 anos         NA
## 3 110001 Alta Floresta D'Oeste 110001 Alta Flor~ 5 a 9 anos         NA
## 4 110001 Alta Floresta D'Oeste 110001 Alta Flor~ 10 a 14 anos        1
## 5 110001 Alta Floresta D'Oeste 110001 Alta Flor~ 15 a 19 anos        2
```

Por fim, municípios onde a contagem de mortes por causas externas foi zero estão registrados como `NA`. Assim, é interessante substituir os `NA`s por zero para facilitar a obtenção de estatísticas descritivas e tabelas de frequência.

```
dados_long <- dados_long %>%
  replace_na(list(contagem = 0))
dados_long %>% slice_head(n = 5)
```

```
## # A tibble: 5 x 5
##   Codigo Nome                  Município         faixa_etaria contagem
##   <chr>  <chr>                 <chr>             <chr>           <dbl>
## 1 110001 Alta Floresta D'Oeste 110001 Alta Flor~ Menor 1 ano         0
## 2 110001 Alta Floresta D'Oeste 110001 Alta Flor~ 1 a 4 anos          0
## 3 110001 Alta Floresta D'Oeste 110001 Alta Flor~ 5 a 9 anos          0
## 4 110001 Alta Floresta D'Oeste 110001 Alta Flor~ 10 a 14 anos        1
## 5 110001 Alta Floresta D'Oeste 110001 Alta Flor~ 15 a 19 anos        2
```

1.6.3 Manipulando e resumindo dados

Uma vez que temos o conjunto de dados organizado podemos começar a explorá-lo. É muito importante ter objetivos claros ao começar a analisar um conjunto de dados. Vamos supor que o nosso interesse inicial é obter a soma de mortes por faixa etária. Essa é uma tarefa de manipulação e resumo de dados. Usando o pacote `dplyr` podemos facilmente realizá-la.

```
dados_long %>%
  group_by(faixa_etaria) %>%
  summarise("Total" = sum(contagem))
```

```
## # A tibble: 14 x 2
##    faixa_etaria    Total
##    <chr>           <dbl>
##  1 1 a 4 anos       2418
##  2 10 a 14 anos     3098
##  3 15 a 19 anos    23670
##  4 20 a 29 anos    64200
##  5 30 a 39 anos    50374
##  6 40 a 49 anos    37548
##  7 5 a 9 anos       1472
##  8 50 a 59 anos    29294
##  9 60 a 69 anos    22586
## 10 70 a 79 anos    18622
## 11 80 anos e mais  27670
## 12 Idade ignorada   2686
## 13 Menor 1 ano      1962
## 14 Total          285600
```

A tarefa foi realizada, mas tem alguns aspectos que podem ser melhorados:

1. A ordem das faixas etárias não está na ordem natural. Por exemplo, a faixa Menor que 1 ano apareceu como antepenúltima.
2. Podemos querer que a faixa com idade ignorada e total não sejam incluídos na nossa tabela.

Assim, precisamos primeiro selecionar quais os níveis da coluna `faixa_etaria` queremos manter e depois especificar qual a ordem que queremos que eles sejam exibidos. Para selecionar quais as faixas etárias queremos usamos a função `filter`.

```
dados_long_faixa <- dados_long %>%
  filter(faixa_etaria %in% c("Menor 1 ano", "1 a 4 anos" , "5 a 9 anos",
                             "10 a 14 anos", "15 a 19 anos" , " 20 a 29 anos" ,
                             "30 a 39 anos" , "40 a 49 anos", "50 a 59 anos" ,
                             "60 a 69 anos" , "70 a 79 anos" , "80 anos e mais"))
```

Agora verificamos a classe da coluna `faixa_etaria`.

```
class(dados_long_faixa$faixa_etaria)
```

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

Note que a classe é `character`. Assim, precisamos convertê-la para `factor` e informar ao `R` qual é a ordem que as faixas devem ser apresentadas.

```
faixa <-  c("Menor 1 ano", "1 a 4 anos" , "5 a 9 anos",
            "10 a 14 anos", "15 a 19 anos" , " 20 a 29 anos" ,
            "30 a 39 anos" , "40 a 49 anos", "50 a 59 anos" ,
            "60 a 69 anos" , "70 a 79 anos" , "80 anos e mais")
dados_long_faixa$faixa_etaria <- factor(dados_long_faixa$faixa_etaria,
                                        levels = faixa)
```

Agora podemos obter a tabela novamente.

```
dados_long_faixa %>%
  group_by(faixa_etaria) %>%
  summarise("Total" = sum(contagem))
```

```
## # A tibble: 11 x 2
##    faixa_etaria   Total
##    <fct>          <dbl>
##  1 Menor 1 ano     1962
##  2 1 a 4 anos      2418
##  3 5 a 9 anos      1472
##  4 10 a 14 anos    3098
##  5 15 a 19 anos   23670
##  6 30 a 39 anos   50374
##  7 40 a 49 anos   37548
##  8 50 a 59 anos   29294
##  9 60 a 69 anos   22586
## 10 70 a 79 anos   18622
## 11 80 anos e mais 27670
```

Uma outra forma de resumir os dados que pode ser de interesse é por município. Suponha que queremos saber quantas mortes por causas externas aconteceram na cidade de Curitiba, porém independente da faixa etária. Neste caso, os passos são: primeiro filtrar apenas os dados de Curitiba e depois somá-los, conforme ilustrado no Código abaixo.

```
dados_long_faixa %>%
  filter(Nome == "Curitiba") %>%
  summarize("Total" = sum(contagem))
```

```
## # A tibble: 1 x 1
##   Total
##   <dbl>
## 1   905
```

Usando as operações de filtrar e resumir é possível fazer uma infinidade de análises similares as apresentadas nesta subseção.

1.6.4 Combinando conjuntos de dados

Suponha agora que o interesse é obter uma tabela com o número absoluto e a taxa de mortalidade por causas externas padronizada pela população residente para cada estado. Neste caso precisamos de um conjunto de dados adicional com a população residente em cada estado. Além disso, precisamos saber a qual estado cada município pertence. O IBGE dispõe de bases de dados com a população residente por município com a informação adicional de a qual estado cada município pertence. Uma exemplo deste tipo de base de dados é o arquivo `Pop_2019.csv` disponível no endereço www.leg.ufpr.br/~wagner/data. Vamos ler esta base de dados em `R` e explorá-la.

```
pop2019 <- read_delim("data/Pop_2019.csv")
pop2019 %>%
  slice_head(n = 5)
```

```
## # A tibble: 5 x 5
##   UF      COD COD_MUNI MUNICIPIO                POP
##   <chr> <dbl> <chr>    <chr>                  <dbl>
## 1 RO       11 00015    Alta Floresta D Oeste  22945
## 2 RO       11 00023    Ariquemes             107863
## 3 RO       11 00031    Cabixi                  5312
## 4 RO       11 00049    Cacoal                 85359
## 5 RO       11 00056    Cerejeiras             16323
```

Inspecionando o conjunto de dados vemos que este está em um formato já organizado com as colunas `UF`, `COD`, `COD_MUNI`, `MUNICIPIO` e `POP`. Precisamos juntar a base de dados da população com a das mortes por causas externas. Para isso precisamos de uma forma de pareamento, ou seja, um código que nos informe como fazer a junção. Comparando a coluna `Código` do conjunto de dados `dados_long_faixa` vemos que ele é a combinação das colunas `COD` e `COD_MUNI` do conjunto de dados `pop2019`, porém o último dígito da coluna `COD_MUNI` deve ser ignorado. Teremos que criar uma coluna `Código` no conjunto de dados `pop2019` para então poder fazer a junção com o conjunto de dados `dados_long_faixa`. A estratégia será a seguinte: primeiro juntamos as colunas `COD` e `COD_MUNI` em uma unica *string* usando a função `str_c` e depois pegamos apenas os seis primeiros digitos com a função `str_sub`. Por fim, criamos uma nova coluna chamada de `Código` com o resultado destas operações.

```
pop2019 <- pop2019 %>%
  mutate("Codigo" = str_sub(str_c(COD, COD_MUNI), 1, 6))

pop2019 %>%
  slice_head(n = 5)
```

```
## # A tibble: 5 x 6
##   UF      COD COD_MUNI MUNICIPIO                POP Codigo
##   <chr> <dbl> <chr>    <chr>                  <dbl> <chr>
## 1 RO       11 00015    Alta Floresta D Oeste  22945 110001
## 2 RO       11 00023    Ariquemes             107863 110002
## 3 RO       11 00031    Cabixi                  5312 110003
## 4 RO       11 00049    Cacoal                 85359 110004
## 5 RO       11 00056    Cerejeiras             16323 110005
```

Agora fazemos a junção usando a função *left_join*. O *left_join* foi utilizado porque queremos

manter apenas os códigos de municípios que estão no conjunto de dados `pop2019`. Você pode inspecionar a coluna `Município` do conjunto de dados `dados_long_faixa` para identificar que existem alguns códigos para indicar que o município do óbito foi ignorado. Assim, não desejamos pegar esses casos.

```
completo <- left_join(pop2019, dados_long_faixa, by = "Codigo")
completo %>%
  slice_head(n = 5)
```

```
## # A tibble: 5 x 10
##   UF      COD COD_MUNI MUNICIPIO               POP Codigo Nome  Município
##   <chr> <dbl> <chr>    <chr>                 <dbl> <chr>  <chr> <chr>
## 1 RO       11 00015    Alta Floresta D Oeste 22945 110001 Alta~ 110001 A~
## 2 RO       11 00015    Alta Floresta D Oeste 22945 110001 Alta~ 110001 A~
## 3 RO       11 00015    Alta Floresta D Oeste 22945 110001 Alta~ 110001 A~
## 4 RO       11 00015    Alta Floresta D Oeste 22945 110001 Alta~ 110001 A~
## 5 RO       11 00015    Alta Floresta D Oeste 22945 110001 Alta~ 110001 A~
## # ... with 2 more variables: faixa_etaria <fct>, contagem <dbl>
```

Para atingirmos nosso objetivo que é ter uma tabela com as taxas de mortalidade por estado precisamos primeiro somar os óbitos por faixa etária dentro de cada municipio. Após podemos somar por estado e então padronizar pela população.

```
municipio <- completo %>%
  group_by(Codigo) %>%
  summarize("Obitos" = sum(contagem),
            "Nome" = unique(Nome),
            "Populacao" = unique(POP),
            "UF" = unique(UF))
```

Agora que temos a base com a soma dos óbitos por município podemos agregar por `UF`.

```
UF <- municipio %>%
  group_by(UF) %>%
  summarize("Obitos" = sum(Obitos),
            "Populacao" = sum(Populacao))
UF %>%
  slice_head(n = 5)
```

```
## # A tibble: 5 x 3
##   UF    Obitos Populacao
##   <chr>  <dbl>     <dbl>
## 1 AC       426    881935
## 2 AL      1712   3337357
## 3 AM      1962   4144597
## 4 AP       421    845731
## 5 BA      9024  14873064
```

Finalmente, calculamos a taxa de mortalidade, digamos, por cem mil e organizamos em ordem decrescente.

```
UF <- UF %>%
  mutate("Taxa" = (Obitos/Populacao)*100000) %>%
  arrange(desc(Taxa))
UF %>%
  slice_head(n = 5)
```

```
## # A tibble: 5 x 4
##   UF    Obitos Populacao  Taxa
##   <chr>  <dbl>     <dbl> <dbl>
## 1 RR       456    605761  75.3
## 2 TO      1062   1572866  67.5
## 3 ES      2625   4018650  65.3
## 4 PE      6017   9557071  63.0
## 5 GO      4349   7018354  62.0
```

Para visualizar a tabela completa no RStudio você pode usar o seguinte código.

```
View(UF)
```

Neste Capítulo passamos por todas as principais etapas de uma análise de dados usando o `R` e os pacotes do `tidyverse`.