Data frames - leitura e manipulação.

André Moura Gomes da Costa - andmgdc@gmail.com
Maria das Gracas Braga Ceccato

14/08/2019

Projetos do RStudio

Uma funcionalidade muito útil do RStudio é o uso de projetos. Com eles, é possível manter uma boa organização do trabalho, facilitando a escrita de programas e a continuidade do trabalho.

Um projeto permite manter salvos:

- Quais *scripts* estão abertos.
- As variáveis no ambiente global.
- O histórico de comandos.
- O espaço de trabalho. (Mas o que é isto?)

O espaço de trabalho

Quando utilizamos o R, muitas vezes queremos interagir com outros arquivos.

Podemos criar e ler arquivos de dados ou de *scripts*.

Uma maneira de acessar esses arquivos é expecificando um caminho global (e.g.: "C:/users/andre/projeto/dados.csv").

Devemos usar / ou \\ para especificar o caminho no R, pois \ é um character especial.

```
"hello\U1F30D"
```

A função `file.choose()` pode ser útil.

O espaço de trabalho

Para facilitar a usabilidade, e permitir o uso de um mesmo código em máquinas diferentes, podemos acessar arquivos utilizando um caminho relativo.

O espaço de trabalho é a pasta de referência para um caminho relativo.

Se o espaço de trabalho estiver na pasta "C:/users/andre/projeto", podemos acessar o mesmo arquivo anterior apenas referenciando "./dados.csv".

Se quisermos referenciar a pasta pai do ambiente de trabalho, podemos usar "../".

O espaço de trabalho

Quando abrimos um projeto do RStudio, o ambiente de trabalho passa a ser a pasta em que temos o arquivo **.Rproj**

Para verificar em qual pasta está o ambiente de trabalho, usamos a função `getwd()`

Para alterar o local do ambiente de trabalho, usamos a função `setwd()`

```
getwd()
```

```
>> [1] "/home/andre/Dropbox/Curso_R_Farmacia/Aula2"
```

```
# Definindo espaço de trabalho com caminho absoluto
setwd("/home/andre")
getwd()
```

```
>> [1] "/home/andre"
```

```
# Definindo espaço de trabalho com caminho relativo
setwd("./2019")
getwd()
```

```
>> [1] "/home/andre/2019"
```

```
setwd("../")
getwd()
```

```
>> [1] "/home/andre"
```

Criando um projeto

Vamos trabalhar, neste curso, utilizando projetos, por isso ser considerado uma boa prática.

file - new project

Arquivos externos

Lendo arquivos em formato textual

Usamos a função `read.table()` para lermos arquivos que podem ser lidos em um editor de texto (como bloco de notas).

- Flexível e robusta.
- Parâmetros mais importantes: *file*, *header*, *sep*, *skip*.
- `read.csv()` e `read.csv2()` são baseadas nela.
  - `read.csv()` é parametrizada para ler **.csv** com separador , .
  - `read.csv2()` é parametrizada para ler **.csv** com separador ; .

Para salvar algum data frame em um arquivo de texto, podemos usar funções como `write.table()` ,`write.csv()` e `write.csv()`

Lendo arquivos de outros programas

Uma maneira possível de se ler planilhas do excel no R é salvando estes arquivos como **.csv** e utilizando as funções descritas acima.

Alguns pacotes permitem a manipulação direta de arquivos do excel, como o **xlsx**.

O pacote **xlsx** possui as funções `write.xlsx()` e `read.xlsx()`, similares às apresentadas anteriormente.

Também, podemos ler arquivos **.sav** do *SPSS* :

```
library(foreign)
dataset = read.spss(
"/home/andre/Dados/Penalti_resposta.SAV",
        to.data.frame=TRUE)
```

Processando um arquivo do R: `source()`

Muitas vezes, queremos executar comandos que estão registrados em arquivos do R que não é o que digitamos.

Ao invés de usar *ctrl+v* e *ctrl+c*, é mais interessante lermos diretamente o que há nestes arquivos.

Códigos mais fáceis de ler e uma consistência maior entre os arquivos.

```
source("./scriptexterno.r")
```

Deixando seus dados *Tidy*

O que são dados *Tidy*?

Tidy: Arrumado; Limpo; Ordenado; Asseado; Ben-arranjado;

1. Cada variável deve ter um uma própria coluna.
2. Cada observação deve ter uma própria linha.
3. Deve haver uma tabela para cada tipo de variável
4. Cada valor deve estar em uma própria célula.

Muitas funções no R são projetadas para serem usadas com dados *tidy*.

O artigo de Hadley Wickham explica o por quê de se utilizar dados *tidy*.

Documentação dos dados

O *code book*

Um documento, como em word, texto, markdown, etc. Que descreve seus dados

- Informações sobre variáveis, como unidades.
- Informações de escolhas de sumarizações.
- Informação sobre o *design* do experimento.

A lista de instrução

Preferencialmente, *script* que:

- Abre os arquivos originais, crus.
- Processa os dados.
- Salva uma saída *tidy*.

Caso não seja possível salvar um script, use um texto, bem detalhado (versões do *software*, parâmetros, etc.).

O pacote *tidyr*

Muitas vezes, entramos em contato com dados que não estão em formato *tidy*.

O pacote *tidyr* nos ajuda a transformá-los em *tidy*, com as funções principais.

- `gather()` transforma multiplas colunas em duas : chave e valor.
- `spread()` faz a operação reversa, transformando um par chave-valor em múltiplas colunas.
- `unite()` une duas colunas em uma.
- `separate()` separa uma coluna em duas.

Praticando com *tidyr*

```
set.seed(1)
df1 <- data.frame(
        # Número do participante
        Participante = 1L:12L,
        # Informações (c=controle, e=experimental,
        #              f= feminino, m = masculino)
        Informação = rep(c("cm", "cm", "cf",
                           "ef", "em", "ef"),2),
        # Resultados das medições nos 3 dias.
        ObservacaoDia1 = rnorm(n = 12,
                               mean = 80, sd = 15),
        ObservacaoDia2 = rnorm(n = 12,
                               mean = 88, sd = 8),
        ObservacaoDia3 = rnorm(n = 12,
                               mean = 90, sd = 7)
)
```

Praticando com *tidyr*

Crie o data frame da página anterior e o visualize.

Descuta em pares:

- O que precisamos fazer para transformá-lo em um *tidy*?
- Quais funções vocês acham que será necessário utilizar?

Gather

Note que a observação para cada dia está em uma coluna diferente, mas em um conjunto de dados tidy, cada variável deve ter sua própria coluna.

As observações em diferentes dias devem estar na mesma coluna.

```
library(tidyr)
df2 <-  gather(df1, key = "Dia", value = "Observação",
               ObservacaoDia1,
               ObservacaoDia2,
               ObservacaoDia3)

# ou df2 <- df1 %>% gather( "Dia"...)
```

Separate

Pode-se observar que a coluna Informações contém mais de um valor: O grupo ao qual o partipante pertence e o seu sexo. Vamos separar estas colunas:

```
df3 <- separate(df2, col = Informação,
                into = c('Grupo','Sexo'),
                #Separar depois do 1o digito.
                sep = 1)
```

Participante	Grupo	Sexo	Dia	Observação
1	c	m	ObservacaoDia1	70.60319
2	c	m	ObservacaoDia1	82.75465
3	c	f	ObservacaoDia1	67.46557
4	e	f	ObservacaoDia1	103.92921
5	e	m	ObservacaoDia1	84.94262
6	e	f	ObservacaoDia1	67.69297

O operador %>%

O operador %>% (*pipe*) passa uma variável como primeiro argumento da função. É muito útil para economizar linhas de código e gerar um *script* de leitura mais fácil.
(\texttt{x %>% fun(y)} é equivalente a `fun(x,y)`)

```
df3b <- df1 %>%
        gather(key = "Dia", value = "Observação",
               ObservacaoDia1,
               ObservacaoDia2,
               ObservacaoDia3) %>%
        separate(col = Informação,
                 into = c('Grupo','Sexo'),
                 sep = 1)
```

Exemplo 2

O que podemos fazer para transformar estes dados em *tidy*?

```
dfEx2.1 <- data.frame(
        nome = c("João", "Maria", "João"),
        sobrenome = c("da Silva", "da Silva",
                      "Antunes"),
        variavel = c(rep(c("altura"), 3),
                     rep(c("peso"), 3),
                     rep(c("idade"), 3)),
        valor = c(180, 170, 170,
                  80, 70, 70,
                  30, 25, 20)
)
```

Spread

Podemos observar que a coluna **valor** possui variáveis de mais de uma variável.

```
dfEx2.2 <- dfEx2.1 %>% spread(key = variavel,
                              value = valor)
```

nome	sobrenome	altura	idade	peso
João	Antunes	170	20	70
João	da Silva	180	30	80
Maria	da Silva	170	25	70

Unite

Podemos considerar que o nome e sobrenome são uma variável só.

```
dfEx2.3 <- dfEx2.2 %>% unite (col = nomeCompleto,
                              nome, sobrenome,
                              sep = " ",
                              remove = TRUE)
```

nomeCompleto	altura	idade	peso
João Antunes	170	20	70
João da Silva	180	30	80
Maria da Silva	170	25	70

Exercícios:

A partir de **df3** chegue a um dataframe equivalente ao **df1**

A partir de **df2Ex.3** chegue a um dataframe equivalente ao **df2Ex.1**

Manipulando DFs com *dplyr*

dplyr

Um pacote muito útil para transformar e sumarizar dados é o *dplyr*.

Este pacote apresenta as vantagens de ser intuitivo e de apresentar um bom desempenho computacional.

Suas principais funções são:

- `select()` → seleciona colunas
- `filter()` → filtra linhas
- `arrange()` → ordena linhas
- `mutate()` → altera uma coluna
- `summarise()` → sumariza valores
- `group_by()` → agrupa
- funções *join* para juntar *data frames*.

A seguir, alguns exemplos de uso serão mostrados. Um bom tutorial pode ser encontrado aqui.

select()

Seleciona colunas de interesse para análise. Execute os códigos a seguir, mas antes de executar uma linha, tente prever qual será a saída.

```
dfdp <- dfEx2.3
```

```
dfdp %>% select(peso, idade)
dfdp %>% select(altura:peso)
dfdp %>% select(-peso)
dfdp %>% select(-peso, -idade, -altura)
```

Suas operações poderiam ser feitas com colchetes, mas o **dplyr** possui uma
sintaxe muito mais amigável.

filter()

Seleciona as linhas que satisfazem alguma condição.

```
dfdp %>% filter (altura == 180)
dfdp %>% filter (altura < 180)
dfdp %>% filter (altura < 180 & idade < 25)
dfdp %>% filter (altura == 180 | idade < 25)
dfdp %>% filter (nomeCompleto == "João Antunes")
dfdp %>% filter (idade %in% c(20,30))
```

arrange()

Ordena o data frame. A ordenação passada primeiro tem preferência e, em caso de empate, segue para as seguintes. Se não for especificado, a ordem é crescente.

```
dfdp %>% arrange(altura)
dfdp %>% arrange(desc(altura))
dfdp %>% arrange(nomeCompleto)
dfdp %>% arrange(altura, desc(idade))
```

mutate()

Altera ou cria uma coluna. Pode ser baseada em operações encolvendo: colunas do *data frame*, vetores externos, números externos.

```
dfdp %>% mutate(IMC = peso/(altura/100)^2)
dfdp %>% mutate(altura = altura/100)
dfdp %>% mutate(sexo = factor(c("M","M","F")))

dfdp %>% mutate(altura = altura/100,
                IMC = peso/altura^2,
                sexo = factor(c("M","M","F")))
```

summarise()

Sumariza variáveis do *data frame*.

```
dfdp %>% summarise(altura.media = mean(altura),
                   altura.std = sd(altura),
                   n = n(),
                   altura.ndist = n_distinct(altura),
                   peso.min = min(peso),
                   peso.max = max(peso),
                   nome.first = first(nomeCompleto)
                   )
```

group_by()

Agrupa variáveis por uma variável de referência ou a combinação de várias.

Usada principalmente em conjunbto com summarise.

```
dfdp <- dfdp %>% mutate(sexo = factor(c("M","M","F")))

dfdp %>% group_by(sexo) %>%
        summarise(altura.media = mean(altura),
                  altura.std = sd(altura),
                  n = n(),
                  altura.ndist = n_distinct(altura),
                  peso.min = min(peso),
                  peso.max = max(peso),
                  nome.first = first(nomeCompleto)
        ) %>% ungroup()
```

group_by()

Também afeta outras funções como mutate, e pode afetar outras, se especificado, como arrange.

```
dfdp %>% mutate(desvioAltura = altura - mean(altura))
dfdp %>% group_by(sexo) %>%
        mutate(desvioAltura = altura - mean(altura))
```

Unindo data frames

Concatenando

Há situações em que, por algum motivo, uma mesma tabela está dividida em mais de um *data frame*. Nesses casos podemos usar as funções `cbind()`, para concatenar colunas, e `rbind()`, para concatenar linhas.

```
ColsExtra <- data.frame(
        nomeCompleto = c("Ana Souza", "Pedro Moura"),
        altura = c(160, 165), idade = c(30, 18),
        peso = c(63, 75), sexo = c("F","M")
)
RowsExtra <- data.frame(
        pontos = rnorm(5, 6, 2),
        cidade = factor(c("BH", "BH", "BH",
                          "SP", "SP"))
)
DF <- dfdp %>% rbind(ColsExtra) %>% cbind(RowsExtra)
```

Funções *Join*

Em dados *Tidy*, cada tipo de variável deve estar em uma tabela diferente. Para juntar informações de tabelas distintas usa-se funções *join*:

- *Mutating Join*
  - `left_join(x,y)` :
    Todas as linhas de x; Todas colunas de x e y.
  - `righ_join(x,y)` :
    Todas as linhas de y; Todas colunas de x e y.
  - `full_join(x,y)` :
    Todas as linhas de x e y; Todas as colunas de x e y.
  - `inner_join(x,y)` :
    Linhas de x que correspondem a uma linha de y;
    Todas colunas de x e y.

Funções *Join*

- *Filtering Join*
  - `semi_join(x,y)` :
    Linhas de x que correspondem a uma linha de y;
    Colunas de x.
  - `anti_join(x,y)` :
    Linhas de x que **não** correspondem a uma linha de y;
    Colunas de x.

Praticando

Vamos praticar um pouco para compreender estas funções.

1. Crie um data frame com três colunas: cidade, população e área.
   a. A coluna cidade deve ter os valores "SP"; "RJ"; "BH".
   b. A coluna população deve ter os valores 12.2; 6.7; e 2.5. [milhões de habitantes]
   c. A coluna população deve ter os valores 1500; 1200; e 300. [km^2]
2. A partir deste data frame, crie uma coluna de densidade populacional, com unidade habitantes por km^2
3. Utilize cada função *join* considerando este data frame e o **DF**, criado durante a aula. (considere ambos os casos, em que **DF** é o primeiro argumento ou o segundo).

Tenho que saber tudo isso de cór?

Quanto mais sabemos utilizar uma feramenta sem consulta, melhor!

Não há como saber todas as funções e parametros.

Quando souber a função, accessar seu arquivo de ajuda é esclarecedor.

Cheat sheets (papéis de cola), podem ser muito úteis no processo de aprendizado!

Há diversos tutoriais na *internet* sobre uma grande variedades de pacote.

Stack Overflow possui perguntas e respostas. Se não achar algo que procura (o que é raro), faça sua pergunta.

Atividades

Faça os exercícios a seguir em um Script de R, que deve ser enviado ao professor. E envie o arquivo .csv final também.

Comente o seu código para que você lembre do que fez, e para que o professor possa corrigir apropriadamente.

Exploraremos o *data frame* **iris**. Digite `View(iris)` para verificar que este *data frame* pode ser acessado.

- Crie irisMolten, uma versão *molten* de iris, isto é, que tenha uma coluna com multiplas variáveis. Com as colunas *Species*, *Variable* e *Value*.
- A partir de irisMolten, crie um *data frame Tidy*, como o original.

- Crie iris2, que seja iguala iris, mas com novas colunas:
  - **Sepal.Ratio**: Igual à divisão de **Sepal.Length** e **Sepal.Width**.
  - **Petal.Ratio**: Igual à divisão de **Petal.Length** e **Petal.Width**.
- Verifique a média das colunas criadas.
- Verifique a média das colunas criadas, extratificando por **Species**. Discuta se há diferença.
- Verifique o tamanho da amostra para cada **Species**.
- Carregue o arquivo disponível na url: "https://raw.githubusercontent.com/andremgc/Curso_R_Farmacia/master/Aula2/Data/irisAux.csv"
- Crie um dataset iris4 que inclua iris2 e uma colunoa a informação do sinonimo da esécie, contida na coluna 'Alef' do dataframe disponibilizado.
- Salve o dataframe iris4 arquivo em um '.csv'.

swirl

Faça os caítulos de 1 a 3 do curos "Getting and Cleaning Data"

```
library(swirl)
install_from_swirl("Getting and Cleaning Data")
```

Obrigado!