



Introdução ao Software R

CEDE - UFMG

Dezembro – 2016

Universidade Federal de Minas Gerais

Conteúdo

- Módulo I: Motivação
- Módulo II: Introdução
- Módulo III: Estrutura de dados
- Módulo IV: Loops e condições
- Módulo V: Importando e exportando arquivo texto
- Módulo VI: Funções
- Módulo VII: Estatística elementar
- Módulo VIII: Visualização de dados
- Módulo IX: Testes de hipóteses
- Módulo X: Modelos estatísticos
- Módulo XI: Tratamento de dados
- Módulo XII: Manipulação de dados
- Módulo XIII: Integração R, Python, Oracle, D3

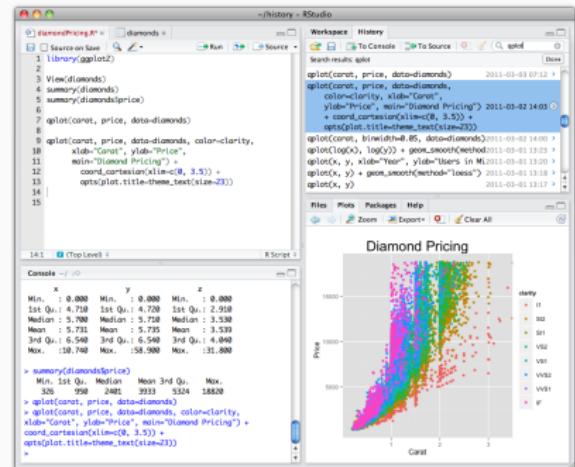
Módulo I: Motivação

R - Linguagem Franca da Ciência dos Dados

O que é o R?

O que é o R?

- Ambiente para computação estatística e gráficos.
- Gratuito!
- Linguagem de código aberto.
- Construído por estatísticos.



O que é o R?

O que é o R?

- Ambiente para computação estatística e gráficos.
- Gratuito!
- Linguagem de código aberto.
- Construído por estatísticos.



Figure 1: Ross Ihaka e Robert Gentleman

O que é o R?

O que é o R?

- Ambiente para computação estatística e gráficos.
- Gratuito!
- Linguagem de código aberto.
- Construído por estatísticos.



Figure 2: Ross Ihaka e Robert Gentleman

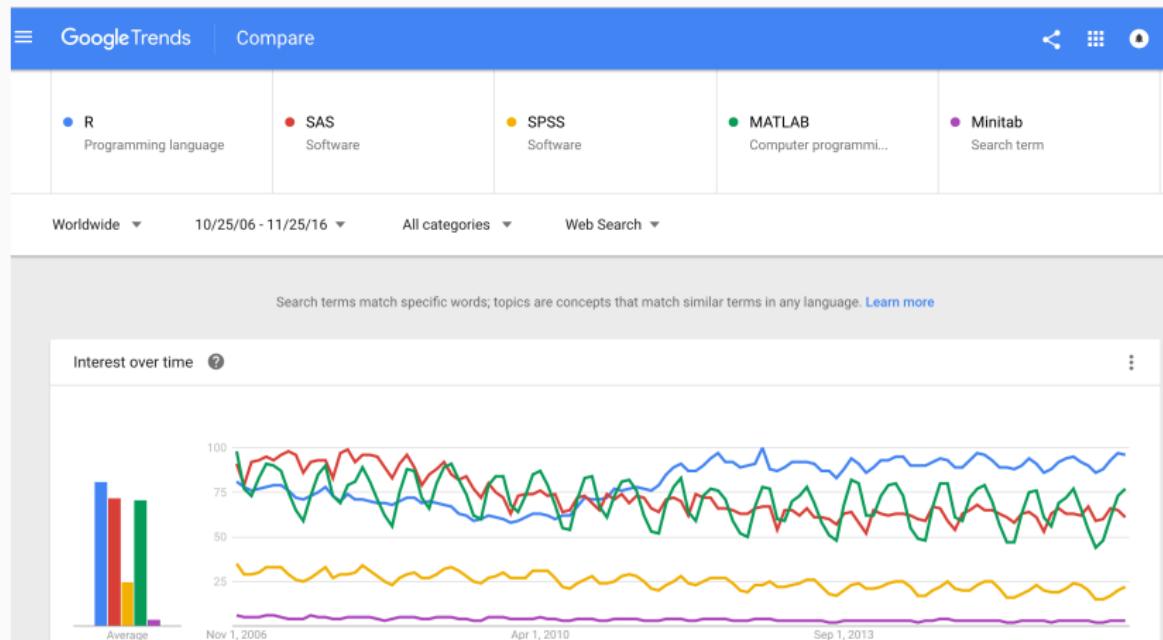
Porque usar o R?



Quem usa R?



Quem usa R?

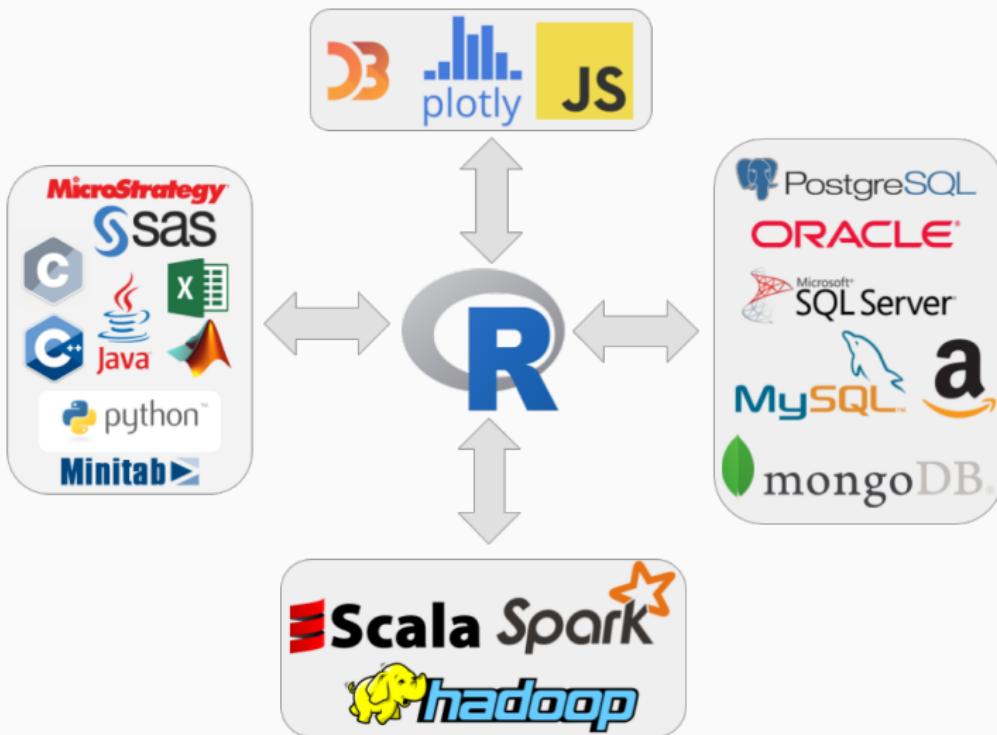






Cap. 5: Spatial Analysis

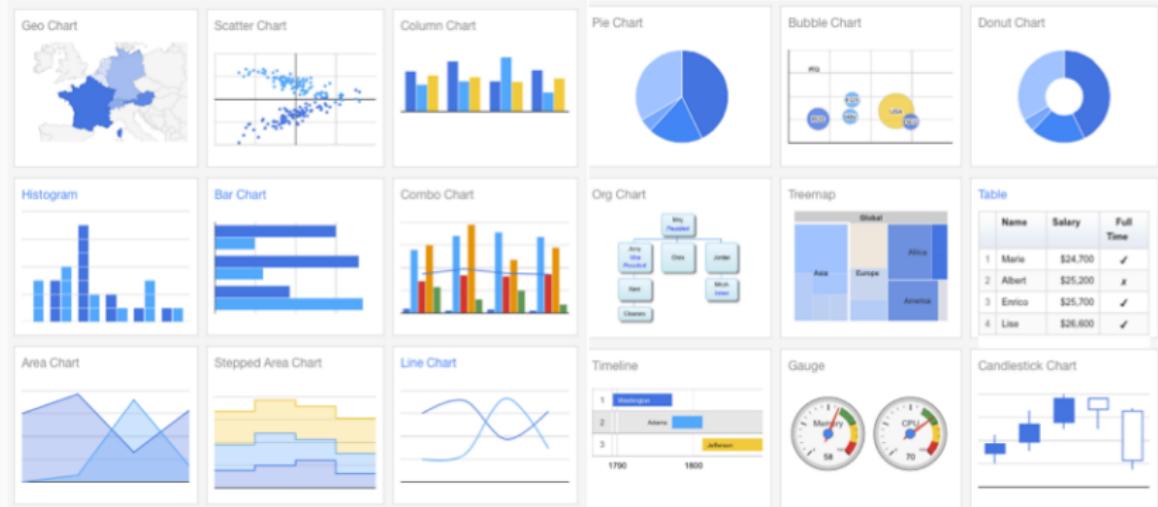
Integração



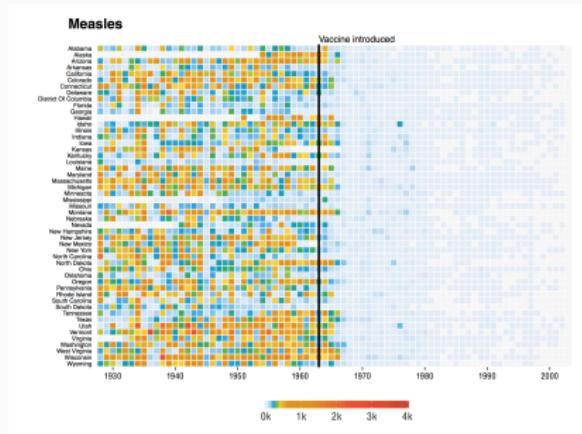
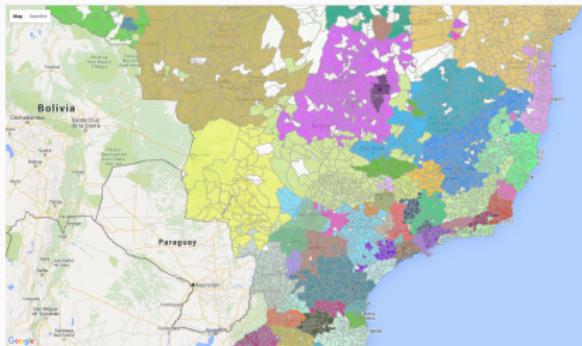
Data Visualization



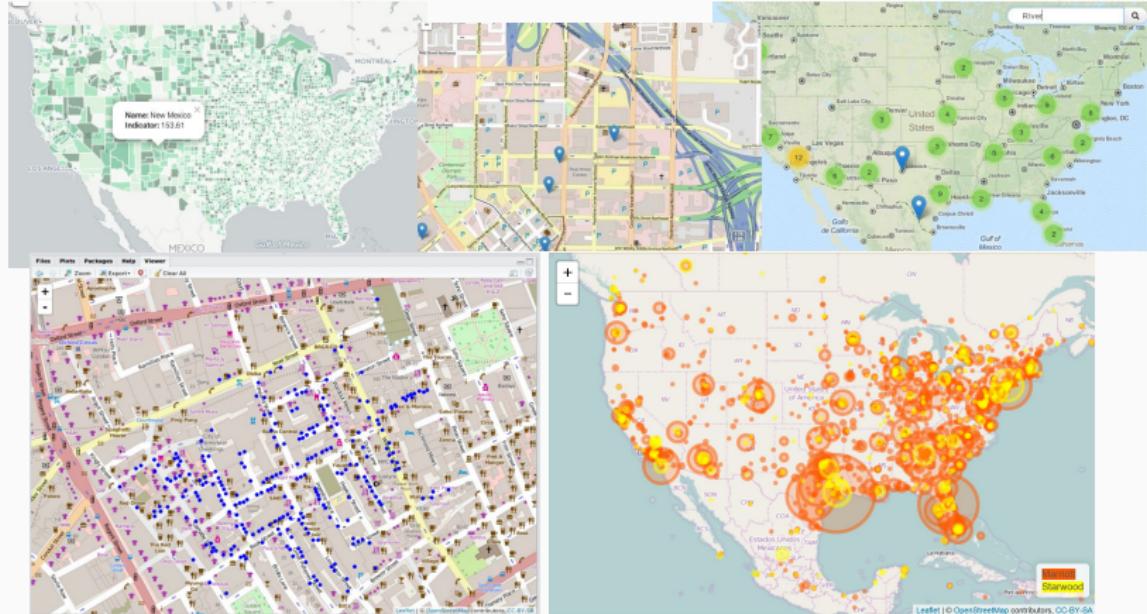
Data Visualization



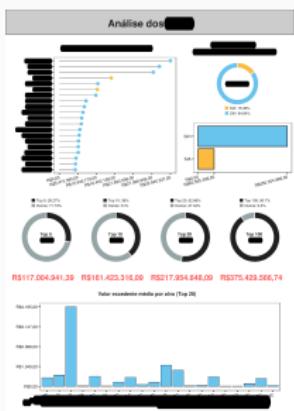
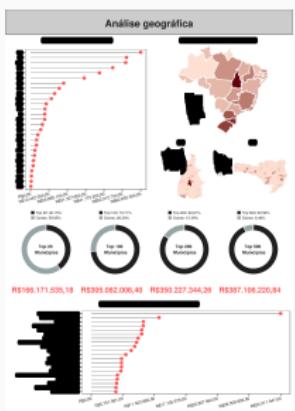
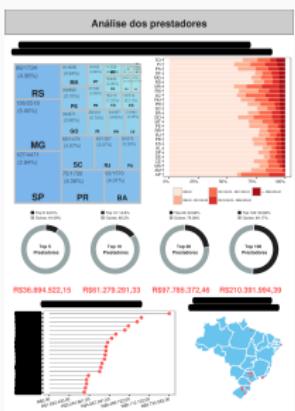
Data Visualization



Mapas Interativos



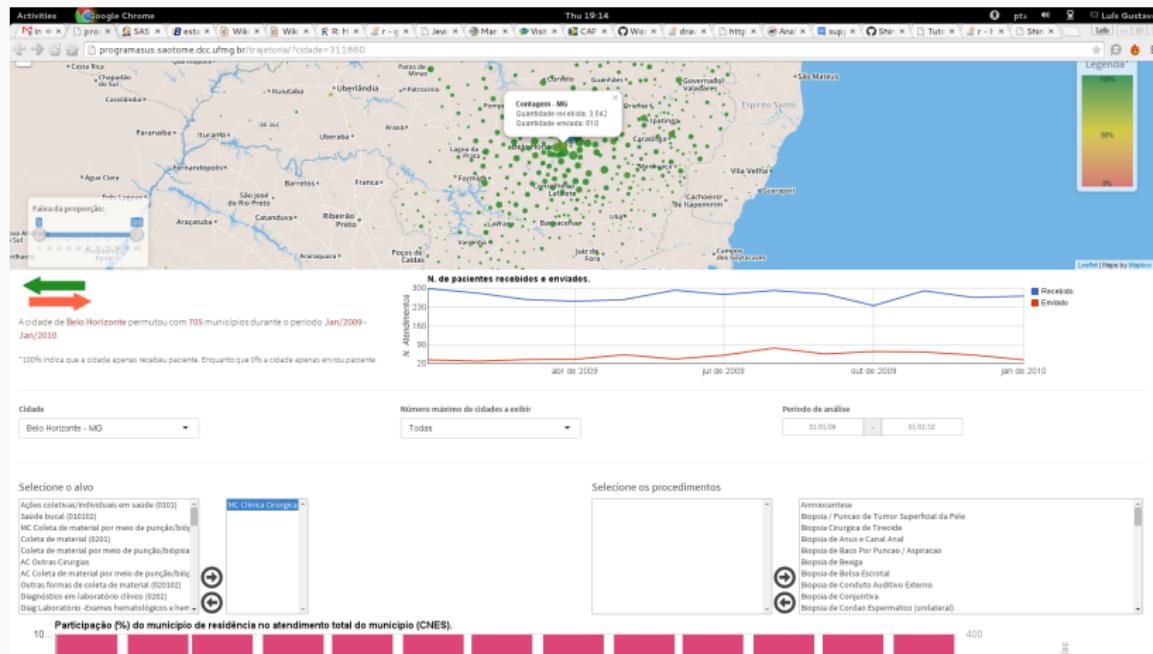
Relatórios Dinâmicos



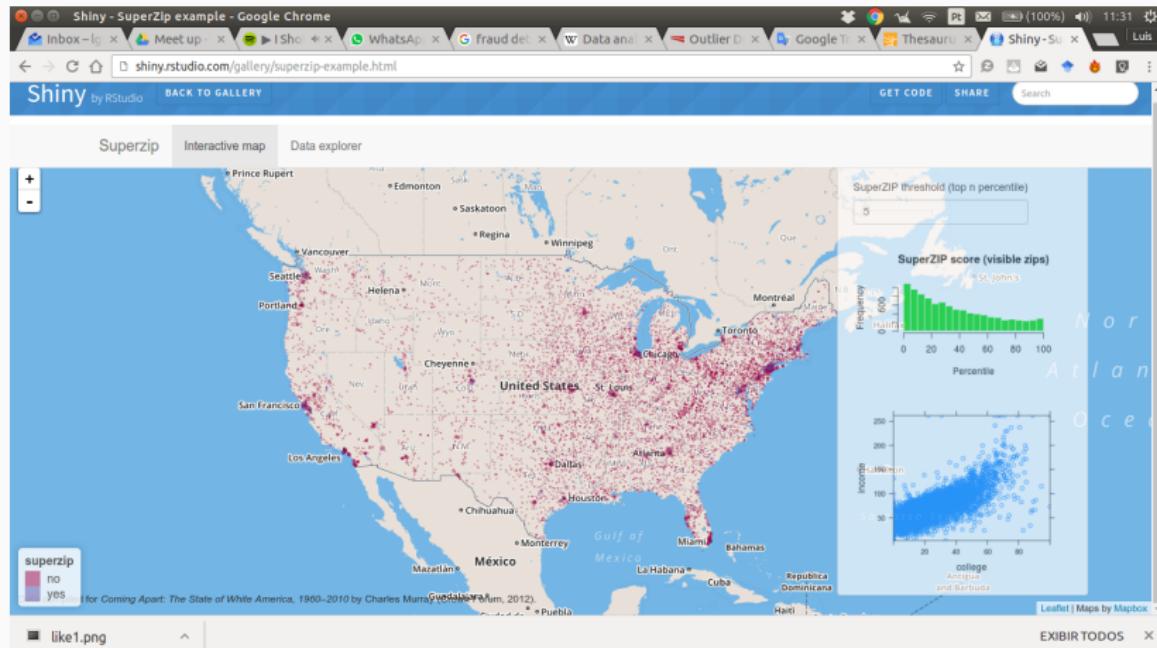
Relatórios Dinâmicos



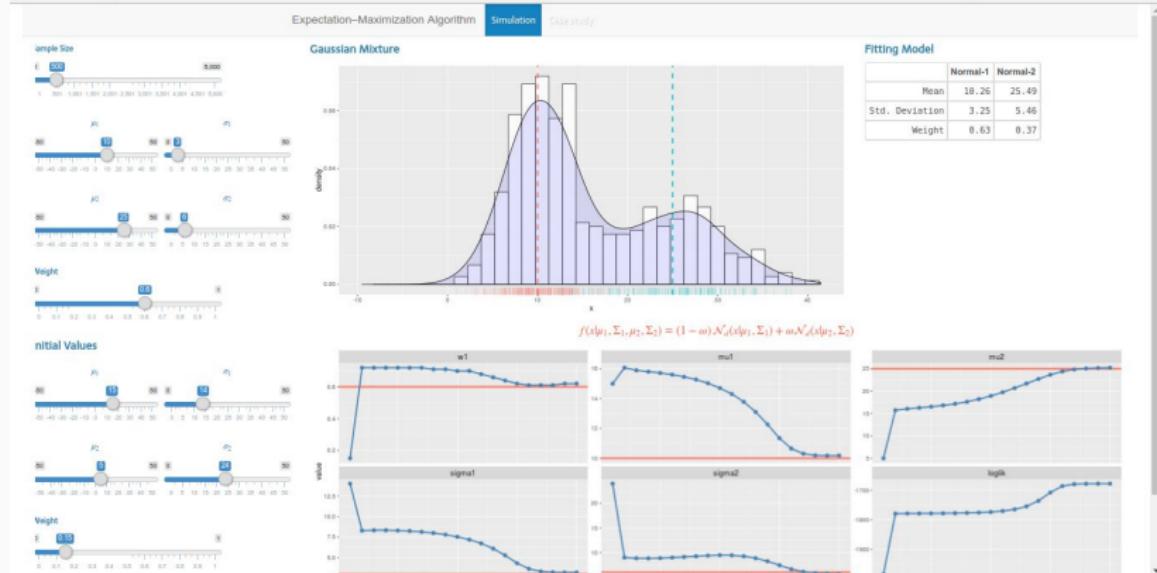
Interatividade – app



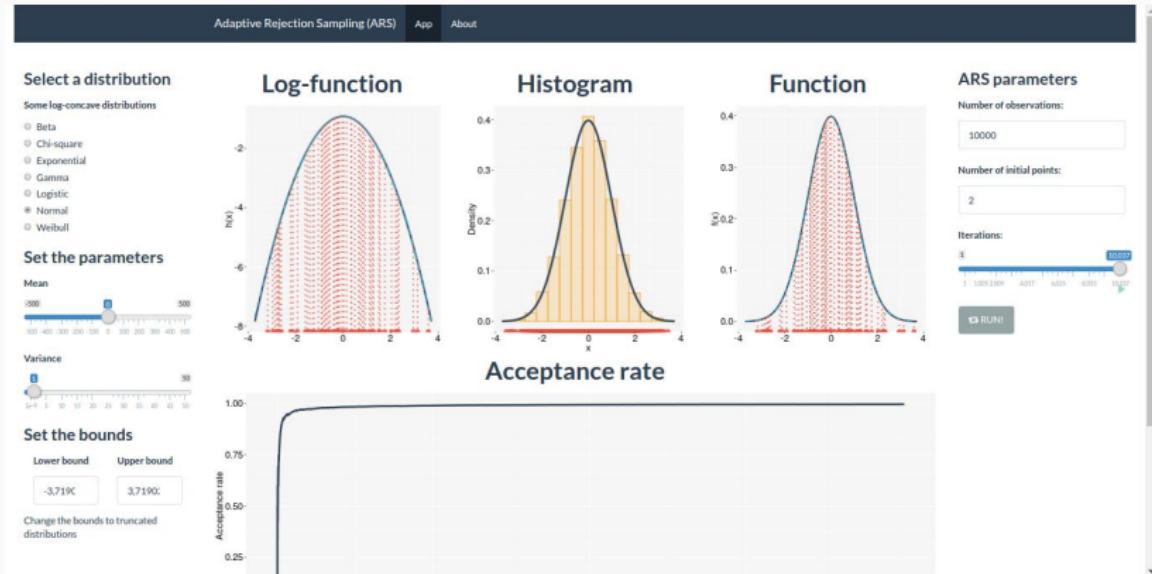
Interatividade – app



Interatividade – app



Interatividade – app



- NBA - Shot Plot
- Market Explore
- Crime Map
- Geo Explorer

Resumindo

- Estatística Espacial, Geoestatística
- Análise de confiabilidade (Tempo até ocorrência de uma falha)
- Classificação, Previsão dinâmica (Filtro de Kalman)
- Machine Learning, Deep Learning, Data Mining, Big Data
- Manipulação de grandes bases de dados
- Gráficos e Mapas interativos
- Aplicativos Web (Shiny)
- Relatórios Dinâmicos
- Multiplataforma - Microsoft, Apple, Linux



Módulo II: Conceitos Básicos

Introdução

Baixando e Instalando o R.

- Link para o site oficial do R: www.r-project.org.

Download

1. Clique no link CRAN (Comprehensive R Archive Network) na seção *Download, Packages*;
2. Escolha um repositório, por exemplo, UFPR;
3. Escolha o link de acordo com o sistema operacional do seu computador, (Ex.: Windows);
4. Escolha a opção *base*, pois as demais são para desenvolvimento de pacotes R;
5. Finalmente clique para baixar o R.

A instalação do R pode ser realizada escolhendo sempre as configurações padrões.

Pacotes, manual e demonstrações

- A versão **base** do R possui uma coleção enorme de funções:
 - Modelos Estatísticos
 - Algoritmos Computacionais
 - Métodos Matemáticas
 - Visualização de Dados

- A versão **base** do R possui uma coleção enorme de funções:
 - Modelos Estatísticos
 - Algoritmos Computacionais
 - Métodos Matemáticas
 - Visualização de Dados

Mas as vezes não é suficiente =/!

Pacotes, manual e demonstrações

- A versão **base** do R possui uma coleção enorme de funções:
 - Modelos Estatísticos
 - Algoritmos Computacionais
 - Métodos Matemáticas
 - Visualização de Dados

Mas as vezes não é suficiente =/!

Pacotes!



Pacotes

- Assim como alguns softwares estatísticos, o R também é extensível através de "módulos". Em R estes módulos são chamados de **pacotes**, **bibliotecas** ou **packages**.

Pacotes

Uma coleção de funções que podem ser escritas em R, C++, Fortran e C e que são chamadas diretamente de dentro do R.

- Um pacote inclui: as funções, dados para exemplificar as funcionalidades do pacote, arquivo com ajuda (help) para cada função, e uma descrição do pacote.
- Qualquer pessoa pode desenvolver seus pacotes e então submeter ao **CRAN**, disponibilizar através do **GitHub** ou **standalone**.

Pacotes

- As funcionalidades do R, podem ser ampliadas carregando estes pacotes, tornando um software ainda mais poderoso, capaz de realizar inúmeras tarefas:
 - Análise multivariada;
 - Análise Bayesiana;
 - Manipulação de dados;
 - Gráficos a nível de publicação;
 - Big Data, Deep Learning;
 - Processamento de imagens.

Alguns pacotes

- `maptools`: Funções para leitura, exportação e manipulação de estruturas espaciais.
 - `cluster`: Funções para análise de clusters.
 - `ggplot2`: Criação de gráficos elegantes.
 - `rmarkdown`: criação de documentos (dinâmicos) em PDF, Word, HTML.
 - `nlme`: Modelos lineares e não-lineares de efeitos mistos.
-
- O R possui mais de ???? pacotes, e milhares de funções.

Instalando pacotes

- Para instalar um pacote do R que já esteja no CRAN basta usar o comando:

```
> install.packages('ggplot2')
```

- Além da opção de comando, também podemos instalar pacotes utilizando os menus do R (Pacotes -> Instalar pacotes), ou do RStudio (Tools -> Install Packages ...).
- Temos também a opção de instalar pacotes a partir de arquivos .zip ou tar.gz (Pacotes -> Instalar pacotes a partir de zip locais) ou utilizando o Rstudio (Tools -> Install Packages ... -> Install From)

Carregando pacotes

- Uma vez que o pacote foi instalado não há mais a necessidade de instalar sempre que for utilizar as suas funcionalidades, basta carregar o pacote com os comandos: `library()` ou `require()`.

```
> library(cluster) # ou  
> require(cluster)
```

Help?

- Para conhecer quais as funções disponíveis no pacote, faça:

```
> help(package = "survey")
```

- Para pedir ajuda de uma determinada função:

```
> ?glm #forma mais comum de acessar o manual da função  
> help("glm")
```

- Obtendo ajuda na internet:

```
> help.search("t.test")
```

- Procurando por alguma função, mas esqueci o nome:

```
> apropos("lm")
> ??lm #??: procurar em todos os pacotes instalados no R
```

- Para todas as outras coisas existe o **Google!**
- Para algumas demonstrações da capacidade gráfica do R:

```
> demo(graphics)
> demo(persp)
> demo(Hershey)
> demo(plotmath)
```

Linguagem compilada vs interpretada

Compilada ou interpretada?

Compilada ou interpretada?

- Essencialmente o R é uma linguagem de programação interpretada.

Compilada ou interpretada?

- Essencialmente o R é uma linguagem de programação interpretada.
- Porém...

Compilada ou interpretada?

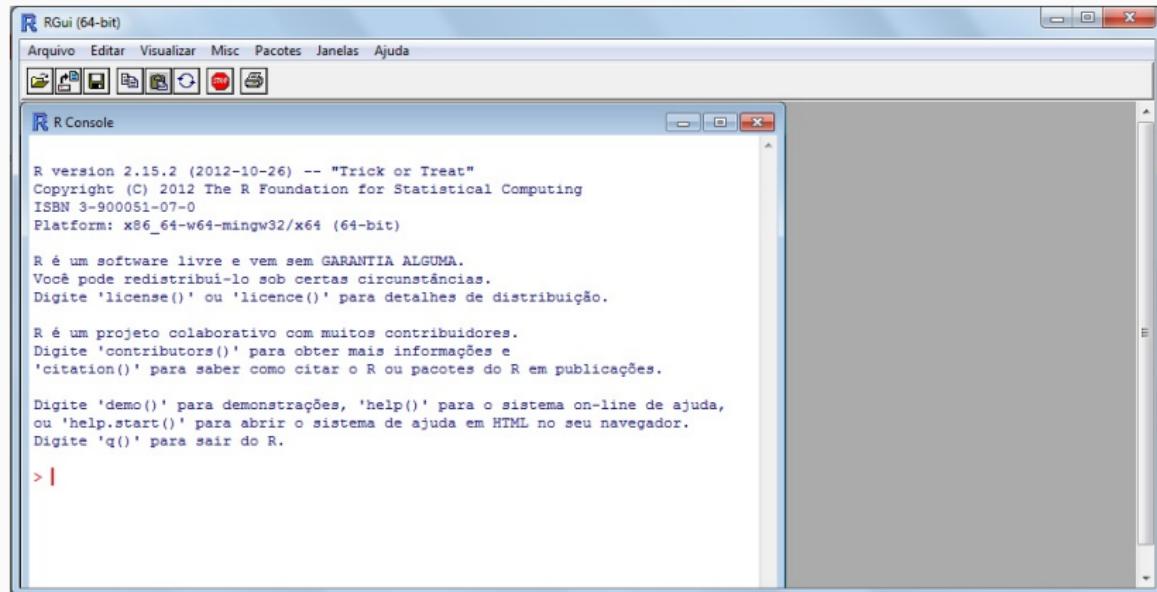
- Essencialmente o R é uma linguagem de programação interpretada.
- Porém...
- É mais correto vê-lo como uma interface para código compilado.

Compilada ou interpretada?

- Essencialmente o R é uma linguagem de programação interpretada.
- Porém...
- É mais correto vê-lo como uma interface para código compilado.
- As principais rotinas são executadas em código compilado (.C, .Call., .Internal, .Primitive)

Ambiente R

Interface R



IDE Rstudio

-/history - RStudio

diamondPricing.R* diamonds

Source on Save Run Source

```
1 library(ggplot2)
2
3 View(diamonds)
4 summary(diamonds)
5 summary(diamonds$price)
6
7 qplot(carat, price, data=diamonds)
8
9 qplot(carat, price, data=diamonds, color=clarity,
10       xlab="Carat", ylab="Price",
11       main="Diamond Pricing") +
12       coord_cartesian(xlim=c(0, 3.5)) +
13       opts(plot.title=theme_text(size=23))
14
15
```

Search results: qplot

```
qplot(carat, price, data=diamonds) 2011-03-03 07:12 >
qplot(carat, price, data=diamonds,
      color=clarity, xlab="Carat",
      ylab="Price", main="Diamond Pricing") 2011-03-02 14:03 >
+ coord_cartesian(xlim=c(0, 3.5)) +
  opts(plot.title=theme_text(size=23))
qplot(carat, binwidth=0.05, data=diamonds) 2011-03-02 14:00 >
qplot(log(x), log(y)) + geom_smooth(method=2011-03-01 13:23 >
qplot(x, y, xlab="Year", ylab="Users in MI") 2011-03-01 13:20 >
qplot(x, y) + geom_smooth(method="loess") 2011-03-01 13:18 >
qplot(x, y) 2011-03-01 13:17 >
```

Workspace History

Files Plots Packages Help

Console -/ ↻

```
x          y          z
Min. : 0.000  Min. : 0.000  Min. : 0.000
1st Qu.: 4.710  1st Qu.: 4.720  1st Qu.: 2.910
Median : 5.700  Median : 5.710  Median : 3.530
Mean   : 5.731  Mean   : 5.735  Mean   : 3.539
3rd Qu.: 6.540  3rd Qu.: 6.540  3rd Qu.: 4.040
Max.  :10.740  Max.  :58.900  Max.  :31.800
```

> summary(diamonds\$price)

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
326	950	2401	3933	5324	18820	

> qplot(carat, price, data=diamonds)
> qplot(carat, price, data=diamonds, color=clarity,
 xlab="Carat", ylab="Price", main="Diamond Pricing") +
 coord_cartesian(xlim=c(0, 3.5)) +
 opts(plot.title=theme_text(size=23))
>

Diamond Pricing

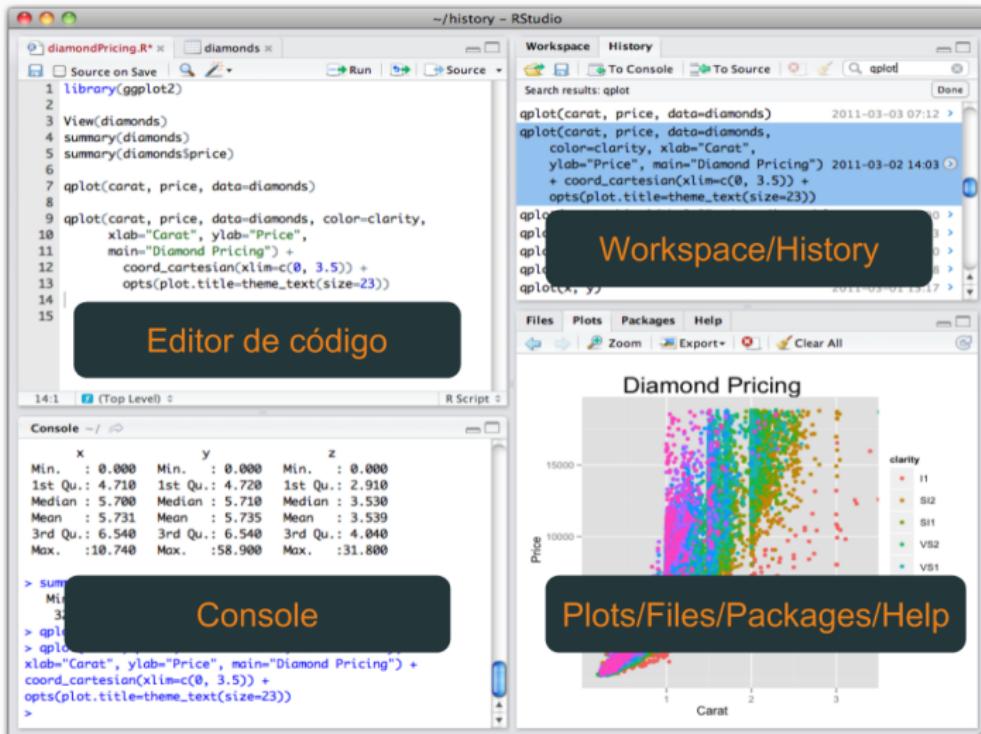
clarity

- I1
- SI2
- SI1
- VS2
- VS1
- VVS2
- VVS1
- IF

Price

Carat

IDE Rstudio



Outras IDE's

IDE's para o R

IDE's para o R

- Emacs Speaks Statistics - ESS
- StatET: plugin para o Eclipse.
- TINN-R

Manipulação simples no prompt

A forma mais direta de interagir com o R é através das linhas de comandos.

- Os comandos são digitados no prompt >.
- Continuação da linha é indicado por +.
- Para submeter os comandos pressione Enter.
- Para inserir vários comandos na mesma linha, utilize ;.

Manipulação simples no prompt

```
> 2 + 3  
> 1 - 8  
> 4 * 5  
> 3 / 5  
> 2 ^ 3
```

Use os parênteses para calcular expressões, por exemplo, $\left(\frac{20+7}{3}\right)^2$.

```
> ((20 + 7)/3)^2  
[1] 81
```

Manipulação simples no prompt

- O R ignora os espaços em brancos excessivos.
- String/caracteres devem ser inseridos entre aspas simples ou dupla: ' ' ou " "

```
> 18           /      3  
[1] 6
```

```
> 25 *  
+  
+   5  
[1] 125
```

```
> "Eu sou uma  
+ string quebrada e entre aspas duplas"  
[1] "Eu sou uma\nstring quebrada e entre aspas duplas"
```

- Note o símbolo de quebra de linha \n.
- Note o símbolo + que indica a continuação do comando.

Criando objetos/variáveis

Princípio 1: Todo que existe no R é um **objeto**.

- Para atribuir valores a objetos, basta usar o operador `<-`, o qual é a combinação do operador `<` com `-`. Como alternativa, podemos utilizar o operador `=`.

```
> objeto1 <- 3*9  
> objeto2 = 8+2
```

- Visualizar o valor armazenado em um objeto, basta digitar o nome do objeto no prompt então apertar **Enter**. Ou usar a função `print()`, ou ainda entre parênteses (`objeto1`)

```
> objeto1  
[1] 27
```

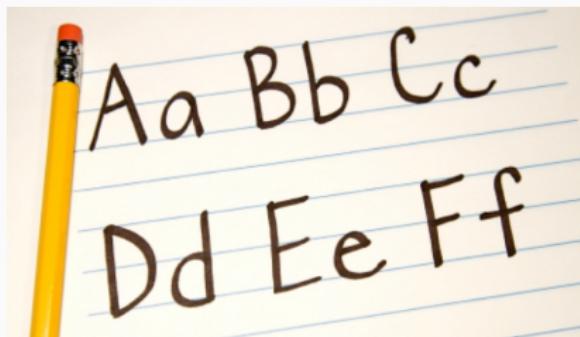
```
> print(objeto1)  
[1] 27
```

Case-sensitive

- Assim como a maioria das linguagens de programação o R também é sensível à letras minúsculas e maiúsculas.

```
> (foo <- "todas as letras sao minusculas")
[1] "todas as letras sao minusculas"
```

```
> (FOO <- "todas as letras sao maiusculas")
[1] "todas as letras sao maiusculas"
```

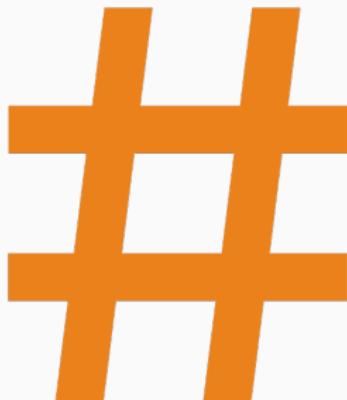


Comentários

- Comentários em R podem ser inseridos depois do caractere **#**. Desta forma, qualquer comando após o caractere **#** não será executado.
 - Ex.

```
> 4*2  
[1] 8
```

```
> #2*2 Olá eu sou um comentário =)
```



Agora é a sua vez

Volume de um tubo

Seja um tubo com raio de 10 cm, com 1,5 metros de comprimento e com uma espessura de 1 cm. Qual o volume deste cubo?



Criando um objeto

Volume de um tubo

Seja um tubo com raio de 10 cm, com 1,5 metros de comprimento e com uma espessura de 1 cm. Qual o volume deste cubo?



Dica

$$\text{Volume} = \pi \times \text{raio}^2 \times \text{altura}$$

$$\pi = 3.14$$

Criando um objeto

Volume de um tubo

Seja um tubo com raio de 10 cm, com 70 cm de comprimento e com uma espessura de 1 cm. Qual o volume deste cubo?

```
> raio <- 10
> espessura <- 1
> comprimento <- 70
> volume <- pi*(raio - espessura)^2*comprimento #calcula o volume do cubo
> volume
[1] 17812.83
```

Criando um objeto

Volume de um tubo

Seja um tubo com raio de 10 cm, com 70 cm de comprimento e com uma espessura de 1 cm. Qual o volume deste cubo?

```
> raio <- 10
> espessura <- 1
> comprimento <- 70
> volume <- pi*(raio - espessura)^2*comprimento #calcula o volume do cubo
> volume
[1] 17812.83
```

- Notaram alguma coisa diferente no cálculo do volume?

Criando um objeto

Volume de um tubo

Seja um tubo com raio de 10 cm, com 70 cm de comprimento e com uma espessura de 1 cm. Qual o volume deste cubo?

```
> raio <- 10
> espessura <- 1
> comprimento <- 70
> volume <- pi*(raio - espessura)^2*comprimento #calcula o volume do cubo
> volume
[1] 17812.83
```

- Notaram alguma coisa diferente no cálculo do volume?
- Onde o **objeto** π foi declarado?

Criando um objeto

Volume de um tubo

Seja um tubo com raio de 10 cm, com 70 cm de comprimento e com uma espessura de 1 cm. Qual o volume deste cubo?

```
> raio <- 10
> espessura <- 1
> comprimento <- 70
> volume <- pi*(raio - espessura)^2*comprimento #calcula o volume do cubo
> volume
[1] 17812.83
```

- Notaram alguma coisa diferente no cálculo do volume?
- Onde o **objeto** π foi declarado?
- O **R** armazena algumas quantidades importantes.

Constantes armazenadas no R

```
> pi  
[1] 3.141593
```

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

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

```
> month.abb  
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

```
> month.name  
[1] "January"    "February"   "March"       "April"        "May"         "June"  
[7] "July"        "August"      "September"  "October"     "November"    "December"
```

```
> Inf  
[1] Inf
```

Operadores lógicos

Operadores lógicos

Operadores lógicos: são operados binários para realização de testes entre duas variáveis (objetos). Estas operações retornam o valor **TRUE** (1) ou **FALSE** (0).

Operadores	Descrição
<	Menor que
<=	Menor ou igual a
>	Maior que
>=	Maior ou igual a
==	Igual a
!=	Diferente de
!x	Não x
x y	x OU y
x & y	x E y

Table 1: Tabela de operadores lógicos.

Usando os operadores lógicos

Exemplos:

```
> x <- 10 #atribuindo o valor 10 ao objeto x  
> y <- 20 #atribuindo o valor 20 ao objeto y  
> x < y  #x é menor que y?  
> x < x  #x é menor que x?  
> x <= x #x é menor igual que x?  
> x > y  #x é maior que x?  
> x >= y #x é maior igual que x?  
> x == y  #x é igual a y?  
> x != y  #x é diferente de y?
```

Dica

Note que há um **espaço em branco** entre os operadores lógicos. Estes espaços não são obrigatórios, porém tornam o **código mais legível**.

Agora é a sua vez

Teste lógico com string

Crie dois objetos em R: um que amarzene a primeira letra do seu primeiro nome e outro com a primeira letra do seu segundo nome. Agore compare estes objetos usando alguns dos operadores lógicos. Por exemplo o operador <=.

Teste lógico com string

Crie dois objetos em R: um que amarzene a primeira letra do seu primeiro nome e outro com a primeira letra do seu segundo nome. Agore compare estes objetos usando alguns dos operadores lógicos. Por exemplo o operador <=.

```
> #Luís Gustavo
> primeira_letra_do_meu_primeiro_nome <- 'L'
> primeira.letra.do.meu.segundo.nome <- 'G'
> primeira_letra_do_meu_primeiro_nome <= primeira.letra.do.meu.segundo.nome
```

Teste lógico com string

Crie dois objetos em R: um que amarzene a primeira letra do seu primeiro nome e outro com a primeira letra do seu segundo nome. Agore compare estes objetos usando alguns dos operadores lógicos. Por exemplo o operador <=.

```
> #Luís Gustavo  
> primeira_letra_do_meu_primeiro_nome <- 'L'  
> primeira.letra.do.meu.segundo.nome <- 'G'  
> primeira_letra_do_meu_primeiro_nome <= primeira.letra.do.meu.segundo.nome
```

Assimilando

- Strings são tratadas com aspas.
- O nome do objeto pode ser tão longo quanto você quira.
- Podemos usar os caracteres _ e . nos nomes dos objetos.
- Não podem iniciar o números: 1luis < - 2.

Classes de objetos

- Em uma análise estatística existem diferentes **tipos de dados**: numéricos, categóricos, ordinais, univariados, bivariados, multivariados, etc.
- R possui diferentes **classes** para acomodar estas diferentes natureza dos dados.

Classes

- `numeric()`: números com casas decimais (double). Ex.: 2.1.
- `integer()`: números inteiros. Ex.: 5L.
- `logical()`: TRUE ou FALSE.
- `character()`: caracteres/strings. Ex.: "Hello!"

Classe: `numeric()`

- Para saber a classe de um objeto devemos utilizar a função `class()`. É recomendado ler o help das funções que iremos aprender neste treinamento. `?class()`.

```
> x <- 12.5  
> class(x)  
[1] "numeric"
```

```
> y <- 10  
> class(y)  
[1] "numeric"
```

```
> ?class()  
> ?numeric()
```

- Podemos declarar um vetor da classe númeroico usando a função `numeric()`. Mais adiante iremos entender o conceito de vetor.

```
> vetor_numerico <- numeric(length = 10)
```

Classe: `integer()`

- Para criar um objeto da classe `integer` devemos utilizar o operador `L`.

```
> inteiro <- 50L  
> class(inteiro)  
[1] "integer"
```

```
> ?integer()
```

Classe: `logical()`

- Objetos da classe `logical` podem ser obtidos através da comparação entre variáveis (objetos) e assumem apenas os valores `TRUE` (T) ou `FALSE` (F).

```
> logico <- 2 < 3 #dois é menor que três?  
> class(logico)  
[1] "logical"
```

```
> (logico <- F) #posso sobrescrever o objeto  
[1] FALSE
```

- Operadores lógicos também podem ser aplicados a objetos da classe `logical`.

```
> u <- TRUE; v <- FALSE      #criando objetos  
> u & v                      #u E v  
> u | v                      #u OU v  
> !v                          #negação de v
```

Classe: `character()`

- `character()`: objetos do tipo `character` são utilizados para representar $\{strings\}$ no `R`. Ou seja, variáveis de natureza textual.

```
> nome <- "Gov. Valadares" #criando objeto  
> class(nome)           #classe do objeto nome  
[1] "character"
```

```
> toupper(nome)          #todas maiusculas  
[1] "GOV. VALADARES"
```

```
> tolower(nome)          #todas minusculas  
[1] "gov. valadares"
```

Classe: character()

- No R existe uma infinidade de funções para manipular strings, além de conseguir interpretar as famosas expressões regulares.

Algumas funções para string

- `paste()`: concatena strings.
- `grep()`: números inteiros. Ex.: `5L`.
- `gsub()`: `TRUE` ou `FALSE`.
- `substr()`: caracteres/strings. Ex.: "Hello!"

```
> gv_uf <- paste('Gov. Valadares', 'MG', sep = ' - ')
> gv_uf <- sub(" +", " ", gv_uf)
> gv_longo <- gsub(pattern = 'Gov.', replacement = 'Governador', x = gv_uf)
> posicao <- regexpr(pattern = ' - ', text = gv_longo)
> gv_sem_uf <- substr(x = gv_longo, start = 1, stop = posicao - 1)
```

Classes de objetos

- As funções do tipo `as.CLASS` são utilizadas para atribuir uma classe ao objeto. Ou seja, elas tentam forçar um obejto ser da classe `CLASSE`.
- Já as funções `is.CLASS` testam se o objeto é da classe `CLASSE`. Neste caso, o retorno desta função um objeto da classe `logical()`.

```
> as.integer(pi)  
[1] 3
```

```
> is.integer(3.14)  
[1] FALSE
```

```
> as.integer("5.45")  
[1] 5
```

```
> as.integer("Minas Gerais")  
[1] NA
```

```
> is.integer("Brasil")  
[1] FALSE
```

```
> as.integer(TRUE)  
[1] 1
```

Agora é a sua vez

Teste lógico com string

- Faça `time <- 'Democr56ata'`
- Qual a classe do objeto `time`?
- Utilize a função `substr()` para obter o número que está no objeto `time` e atribua este valor em um outro objeto. Ex.: `numero <- substr(???)`.
- Qual a classe do objeto `numero`? Transforme-o para a classe adequada.

Teste lógico com string

- Faça `time <- 'Democr56ata'`
- Qual a classe do objeto `time`?
- Utilize a função `substr()` para obter o número que está no objeto `time` e atribua este valor em um outro objeto. Ex.: `numero <- substr(???)`.
- Qual a classe do objeto `numero`? Transforme-o para a classe adequada.

```
> time <- 'Democr56ata'  
> class(time)  
> numero <- as.numeric(substr(time, 7, 8))  
> class(numero)
```

Módulo III: Estrutura de dados

Estrutura de dados

- Até este momento estamos apenas trabalhando com objetos escalares, ou seja, com um único valor. Agora em diante iremos conhecer estruturas de armazenamento de dados em R.
- **Dados:** são as informações obtidas de uma *unidade experimental ou observacional*.
- **Exemplo:** ‘Estes são tubos de aços produzidos na empresa XYZ com espessura de 1 cm e comprimento 80 cm.’

```
> dados
  id fabricante espressura comprimento
1 0001          XYZ      1.000      80.001
2 0002          ABC      0.988      79.999
3 0003          KML      1.001      81.001
```

No R existem várias estruturas para armazenar dados. Desde de dados que podem ser modelados em uma tabela, quando dados de natureza textual ou espacial. Abaixo é listado as estruturas mais comuns para iniciar na linguagem R.

- **Vetores:** `c()`, estrutura unidimensional;
- **Matrizes:** `matrix()`, estrutura bidimensional;
- **Arranjos (arrays):** `array()`, é uma generalização de matriz, por exemplo, um cubo;
- **Listas:** `list()`, a estrutura de dados mais genérica do R;
- **Data frames:** `data.frame()`, caso especial de uma lista;

Vetores

- **vetores:** um vetor é uma sequência de dados do mesmo tipo ou classe. Ou seja, um vetor só pode conter valores numéricos, ou lógicos, ou de caracteres.
- **Atenção:** nunca um vetor será composto por um valor numérico e lógico ao mesmo tempo.
- **Exemplo:** usando a função `c()` para criar um vetor. Esta função tem como argumento os elementos que irão compor o nosso vetor, e sua tarefa é concatenar todos os elementos em um único objeto.

```
> (numerico.vet <- c(1, 3, 1, 9))
[1] 1 3 1 9
```

```
> (logico.vet <- c(T, TRUE, FALSE, T))
[1] TRUE TRUE FALSE TRUE
```

```
> (caractere.vet <- c('a', 'cc', 'dd'))
[1] "a"  "cc" "dd"
```

```
> ?c()
```

- **Tarefa:** encontra a classe de cada um dos objetos criados acima.

Vetores: atributos

- Estas novas estruturas possuem alguns atributos que são de nosso interesse. Por exemplo, sua **classe**, **comprimento** do vetor, **nomes** de cada posição do vetor, etc.
- Para sabermos a classe de um objeto, já sabemos que basta aplicar a função **class()** no objeto.
- O comprimento de um vetor, ou quantos elementos este vetor possui, pode ser obtido através da função **length()**.

```
> length(numerico.vet)
[1] 4
```

```
> ?length #consulte o help desta função
```

Vetores: atributos

- O atributo de nomes de cada elemento pode ser acessado com a função `names()`.

```
> xx <- c(1, 2, 3)
> names(xx) #atributo vazio
NULL
```

```
> names(xx) <- c('Posição 1', 'Posição 2', 'Posição 3')
> xx
Posição 1 Posição 2 Posição 3
      1          2          3
```

Combinando vetores

- Podemos combinar/concatenar vetores usando a mesma função `c()`.
- **Exemplo:** vamos combinar os vetores `vet1` e `vet2` e armazenar em um terceiro objeto `vet3`.

```
> vet1 <- c( 10, 20, 30, 40)
> vet2 <- c( 60, 70, 80, 90, 100, 110)
> (vet3 <- c(vet1, vet2))
[1] 10 20 30 40 60 70 80 90 100 110
```

- **Tarefa:** tente `c(vet2, vet1)`.

Criando vetores longos

- A entrada de dados diretamente no R não é recomendada, porém em alguns momentos é necessário criar alguns vetores grandes. Abaixo listamos algumas funções:

Funções rep() e seq()

- **rep()**: replica os valores passados para a função.
- **seq()**: cria uma sequência, sendo possível controlar a que passo a sequência cresce.
- : atalho para função **seq()**, quando queremos criar uma sequência que é incrementada por uma unidade.

```
> rep(x = c(1, 2, 3), each = 3)
[1] 1 1 1 2 2 2 3 3 3
```

```
> rep(x = c(1, 2, 3), times = 3)
[1] 1 2 3 1 2 3 1 2 3
```

- **each = XX**, indica que cada elemento será repetido **XX** vezes.
- **times = XX**, indica que o vetor será repetido **XX** vezes.

Criando vetores longos

- Criando um vetor usando a função `seq()`.

```
> seq(from = 10, to = 110, by = 10) #lembra do vet3 <- c(vet1, vet2) ?  
[1] 10 20 30 40 50 60 70 80 90 100 110
```

```
> seq(from = 0, to = 1, length.out = 10)  
[1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667 0.7777778  
[9] 0.8888889 1.0000000
```

- A função `seq()`, cria um vetor que inicia `from = INICIO` e termina `to = FIM`.
- `by = XX`, indica que a sequência será construída de `XX` em `XX`.
- `length.out = XX`, indica que o vetor terá exatamente o comprimento igual a `XX`.

Agora é a sua vez

Operações com vetores

- Experimente digitar no R:
 - `x <- 1:5; y <- c(2:4, 1, 2)`
 - `x == seq(1, 5, by = 1)`
 - `x < y`

Operações com vetores

- Experimente digitar no R:
 - `x <- 1:5; y <- c(2:4, 1, 2)`
 - `x == seq(1, 5, by = 1)`
 - `x < y`

O que aconteceu?

Operações com vetores

- O R também é conhecido por ser uma linguagem vetorizada. Por exemplo, no exercício anterior, quando comparamos $x < y$, o teste foi realizado para todo o vetor. Com isso, cada elemento do vetor x foi comparado com o seu respectivo par do vetor y .
- Outras operações também podem ser executadas, por exemplo:

```
> a <- 1:5; b <- 3:7  
> a + b #somando elemento a elemento  
> a - b #subtraindo elemento a elemento  
> a * b #multiplicando elemento a elemento  
> a / b #dividindo elemento a elemento  
> a ^ b #exponeciando elemento a elemento
```

- Note que o resultado é sempre um vetor de mesmo comprimento que os vetores a e b .
- E se os vetores tivessem comprimentos diferentes?

Operações com vetores

- **Atenção:** quando realizamos as mesmas operações anteriores, porém com vetores de comprimentos diferentes.

```
> u <- c(10, 20, 30)
> v <- 1:9
> u + v
[1] 11 22 33 14 25 36 17 28 39
```

```
> w <- 1:10
> u + w
Warning in u + w: longer object length is not a multiple of shorter object length
[1] 11 22 33 14 25 36 17 28 39 20
```

Acessando elementos do vetor

- O acesso aos elementos de um vetor é realizado através do operador colchetes: [].

vetor[INDICE]

- Por exemplo, para acessar o índice (posição) 4 do vetor x, basta fazer x[4]

```
> x <- 10:1  
> x[4]  
[1] 7
```

- Índice **negativo**: quando o sinal negativo é usado na frente do índice, o resultado é um vetor com o membro referente a este índice removido do vetor:

vetor[-INDICE]

```
> x[-4]  
[1] 10 9 8 6 5 4 3 2 1
```

Acessando elementos do vetor

Nota: o vetor resultado da consulta pode ser armazenado e utilizado em outras análises, ou seja, a saída (*output*) do R também pode ser utilizada como dados de entrada!

```
> novo_x <- x[-5]
> length(x)
[1] 10
```

```
> length(novo_x)
[1] 9
```

Acessando elementos do vetor

- Uma forma ainda mais interessante de acessar elementos de um vetor é utilizando os operadores lógicos.
- Esta forma também é chamada de filtro. Lembra que ao realizar uma operação lógica sobre um vetor, nós temos como output um outro vetor de **TRUE** e **FALSE**.
- Suponha que temos um vetor de idades e queremos selecionar apenas as idades acima de 35 anos.

```
> idade <- c(34, 27, 20, 28, 32, 43, 31, 18, 45, 36)
> idade > 35
[1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE
```

- Alguma sugestão?

Acessando elementos do vetor

- Uma forma ainda mais interessante de acessar elementos de um vetor é utilizando os operadores lógicos.
- Esta forma também é chamada de filtro. Lembra que ao realizar uma operação lógica sobre um vetor, nós temos como output um outro vetor de **TRUE** e **FALSE**.
- Suponha que temos um vetor de idades e queremos selecionar apenas as idades acima de 35 anos.

```
> idade <- c(34, 27, 20, 28, 32, 43, 31, 18, 45, 36)
> idade > 35
[1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE
```

- Alguma sugestão?
- Basta fazermos **vetor[operacao_logica]**.

```
> idade[idade > 35]
[1] 43 45 36
```

Agora é a sua vez

Acessando vetores.

- Temos o vetor `altura` com a altura de 15 mulheres. Queremos criar mais três vetores:
 - Um com as mulheres com altura menor ou igual a 160.
 - Um com as mulheres com altura maior que 160 e menor ou igual a 170.
 - Um com as mulheres com altura maior que 170.
- Remover a altura 180 do vetor `altura`.

```
> altura <- c(150, 152, 145, 157, 167, 172, 175, 170, 165, 177, 162, 180, 160, 155, 147)
```

Acessando vetores.

- Temos o vetor `altura` com a altura de 15 mulheres. Queremos criar mais três vetores:
 - Um com as mulheres com altura menor ou igual a 160.
 - Um com as mulheres com altura maior que 160 e menor ou igual a 170.
 - Um com as mulheres com altura maior que 170.
- Remover a altura 180 do vetor `altura`.

```
> altura <- c(150, 152, 145, 157, 167, 172, 175, 170, 165, 177, 162, 180, 160, 155, 147)

> altura <- c(150, 152, 145, 157, 167, 172, 175, 170, 165, 177, 162, 180, 160, 155, 147)
> menor160      <- altura[altura <= 160]
> entre160e170 <- altura[altura > 160 & altura <= 170]
> maior170     <- altura[altura > 170]
>
> #removendo 180
> altura[altura != 180]
[1] 150 152 145 157 167 172 175 170 165 177 162 160 155 147
```

Algumas funções que são aplicadas sobre vetores

- `mean(x)`: média
- `sd(x)`: desvio padrão
- `min(x)`: mínimo
- `max(x)`: máximo
- `range(x)`: vetor com mínimo e máximo
- `sum(x)`: soma todos os elementos
- `exp(x)`: exponencia todos os elementos
- `sqrt(x)`: raiz quadrada
- `log(x)`: logarítmico natural

Matrizes

- **matriz:** é uma coleção de vetores lado a lado, em que cada vetor tem exatamente o mesmo comprimento e são da mesma classe. Cada linha ou coluna de uma matriz individualmente será um vetor.
- Exemplo de uma matriz com 2 linhas e 3 colunas:

$$A = \begin{bmatrix} 2 & 4 & 3 \\ 1 & 5 & 7 \end{bmatrix}$$

Matrizes

- **matriz:** é uma coleção de vetores lado a lado, em que cada vetor tem exatamente o mesmo comprimento e são da mesma classe. Cada linha ou coluna de uma matriz individualmente será um vetor.
- Exemplo de uma matriz com 2 linhas e 3 colunas:

$$A = \begin{bmatrix} 2 & 4 & 3 \\ 1 & 5 & 7 \end{bmatrix}$$

- Representando esta matriz no R:

```
> A <- matrix(  
+ c(2,4,3,1,5,7), # elementos de dados  
+ nrow = 2,          # numero de linhas  
+ ncol = 3,          # numero de colunas  
+ byrow = TRUE)     # preencher matriz pelas linhas  
> A  
      [,1] [,2] [,3]  
[1,]    2     4     3  
[2,]    1     5     7
```

Acessando elementos de uma matriz

- O elemento na linha m da coluna n da matriz A pode ser acessado pela expressão $A[m, n]$:

```
> A[2, 3] # elemento na linha 2, coluna 3  
[1] 7
```

- A linha m inteira de A pode ser extraída por $A[m,]$

```
> A[2, ] # acessando a linha 2 completa  
[1] 1 5 7
```

- A coluna n inteira de A pode ser extraída por $A[, n]$

```
> A[, 3] # acessando a coluna 3 completa  
[1] 3 7
```

- Podemos também extrair mais de uma linha ou coluna por vez:

```
> A[, c(1, 3)] # as colunas 1 e 3  
[,1] [,2]  
[1,]    2    3  
[2,]    1    7
```

Matriz: atributos

- As matrizes também possuem seus atributos: nomes das linhas, nomes das colunas, número de linha e colunas.
- `rownames()`: atribui e acessa os nomes das linhas
- `colnames()`: atribui e acessa os nomes das colunas
- `nrow()`: retorna o número de linhas da matriz
- `ncol()`: retorna o número de colunas da matriz
- `dim()`: retorna o número de linhas e colunas da matriz

```
> dim(A)  
[1] 2 3
```

```
> nrow(A)  
[1] 2
```

```
> ncol(A)  
[1] 3
```

```
> rownames(A) <- letters[1:nrow(A)]  
> colnames(A) <- LETTERS[1:ncol(A)]  
> A  
A B C  
a 2 4 3  
b 1 5 7
```

Combinando matrizes

- Assim como os vetores, as matrizes também podem ser concatenadas/combinadas usando as funções:
- `cbind()`: concatena matrizes lado a lado, ou por colunas
- `rbind()`: concatena matrizes empilhando-as, ou por linhas
- Ambas funções podem concatenar vetores e então resultar em uma matriz, como no exemplo abaixo.

```
> Segunda <- c(23, 29, 27, 28, 25)
> Terça   <- c(21, 24, 29, 31, 21)
> Quarta  <- c(24, 26, 25, 27, 31)
> Quinta  <- c(21, 27, 32, 21, 21)
> Sexta   <- c(22, 33, 27, 24, 33)
> Sábado  <- c(30, 28, 25, 24, 20)
> Domingo <- c(21, 21, 20, 30, 26)
>
> dias_uteis <- cbind(Segunda, Terça, Quarta, Quinta, Sexta)
> final_de_semana <- cbind(Sábado, Domingo)
```

Combinando matrizes

- Agora vamos concatenar as duas matrizes com a função `cbind()`:

```
> dias_uteis
      Segunda Terça Quarta Quinta Sexta
[1,]     23    21    24    21    22
[2,]     29    24    26    27    33
[3,]     27    29    25    32    27
[4,]     28    31    27    21    24
[5,]     25    21    31    21    33
```

```
> final_de_semana
      Sábado Domingo
[1,]     30     21
[2,]     28     21
[3,]     25     20
[4,]     24     30
[5,]     20     26
```

```
> (mes <- cbind(dias_uteis, final_de_semana))
      Segunda Terça Quarta Quinta Sexta Sábado Domingo
[1,]     23    21    24    21    22    30    21
[2,]     29    24    26    27    33    28    21
[3,]     27    29    25    32    27    25    20
[4,]     28    31    27    21    24    24    30
[5,]     25    21    31    21    33    20    26
```

Combinando matrizes

- Podemos concatenar (empilhar) usando a função `rbind()`:

```
> matriz1 <- matrix(1:8, nrow = 2, ncol = 4)
> matriz2 <- matrix(1:16, nrow = 4, ncol = 4)
> (matriz_rbind <- rbind(matriz1, matriz2))
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
[3,]    1    5    9   13
[4,]    2    6   10   14
[5,]    3    7   11   15
[6,]    4    8   12   16
```

Acessando matrizes: `rownames()` e `colnames()`

- As matrizes também podem ser acessadas usando os nomes das linhas e colunas:

```
> rownames(mes) <- paste('Semana', 1:nrow(mes), sep = '')  
> mes['Semana1', ] # o output desta consulta é um vetor  
Semana1   Semana2   Semana3   Semana4   Semana5   Semana6  
      23        21        24        21        22        30        21
```

```
> mes[1:2, 'Quinta']  
Semana1  Semana2  
      21        27
```

```
> mes[c('Semana1', 'Semana3'), c('Terça', 'Quarta')]  
Terça  Quarta  
Semana1    21    24  
Semana3    29    25
```

Agora é a sua vez

Acessando matrizes

- Considere a matriz `mes` e calcule:
 - A média da coluna `Quarta`.
 - A desvio-padrão da `Semana3`.
 - A média de cada uma dos dias da semana.
- Dica: `colMeans()`

```
> mean(mes[, 'Quarta'])  
[1] 26.6
```

```
> sd(mes['Semana3', ])  
[1] 3.735289
```

```
> colMeans(mes)  
Segunda   Terça   Quarta   Quinta   Sexta   Sábado Domingo  
    26.4     25.2     26.6     24.4     27.8     25.4     23.6
```

Algumas funções que são aplicadas sobre matrizes

- Todas as funções que são aplicadas em vetores.
- `colMeans(x)`: média de cada coluna
- `rowMeans(x)`: média de cada linha
- `colSums(x)`: soma de cada coluna
- `rowSums(x)`: soma de cada linha

Listas: `list()`

- Lista é a estrutura de dados mais genérica do R, pois ela comporta vetores e matrizes de diferentes classes.

```
> num  <- c(2, 3, 5)
> car   <- letters[1:5]
> logi <- c(TRUE, FALSE, TRUE, FALSE, FALSE)
> x <- list(num, car, logi, 10)
> x
[[1]]
[1] 2 3 5

[[2]]
[1] "a" "b" "c" "d" "e"

[[3]]
[1] TRUE FALSE TRUE FALSE FALSE

[[4]]
[1] 10
```

- Note que os vetores são de classes e tamanhos diferentes.

Acessando elementos de uma lista

- Para acessar uma parte da lista usa-se o operador colchetes simples [].

```
> x[2]  
[[1]]  
[1] "a" "b" "c" "d" "e"
```

```
> class(x[2])  
[1] "list"
```

- Note que usar o colchetes simples, o objeto retornado é da classe `list()`.
- Para acessar os valores armazenados em uma posição da lista temos que usar os colchetes duplos [[]]

```
> x[[2]]  
[1] "a" "b" "c" "d" "e"
```

```
> class(x[[2]])  
[1] "character"
```

Acessando elementos de uma lista

- Podemos modificar seu conteúdo diretamente:

```
> x[[2]][1] <- "WW"  
> x[[2]]  
[1] "WW" "b" "c" "d" "e"
```

```
> car  
[1] "a" "b" "c" "d" "e"
```

- Note que o vetor `car` foi apenas utilizado para construir a lista `x`, com isso ao alterar `x`, não estamos alterando os valores de `car`.

Nomes de membros de listas

- Podemos atribuir nomes aos membros de uma lista.

```
> compras <- list(pc = c("notebook", "desktop"), ano = c(1998, 2005, 2012))
> compras
$pc
[1] "notebook" "desktop"

$ano
[1] 1998 2005 2008 2012
```

```
> names(compras) #nome de cada elemento da lista
[1] "pc"   "ano"
```

- Note o símbolo **\$** no output acima. Este mesmo símbolo pode ser utilizado para acessar uma posição da lista.

```
> compras$pc
[1] "notebook" "desktop"
```

Outras formas de acessar uma lista

- Usando o nome para acessar:

```
> compras[["pc"]]
[1] "notebook" "desktop"
```

- Usando um vetor para acessar multiplas posições:

```
> x[c(1, 3)]
[[1]]
[1] 2 3 5

[[2]]
[1] TRUE FALSE TRUE FALSE FALSE
```

Outras formas de acessar uma lista

- Atribuindo nomes ao membros da lista:

```
> names(x)  
NULL
```

```
> names(x) <- c('numeros', 'caractere', 'logico', 'escalar')  
> x[c('numeros', 'caractere')]  
$numeros  
[1] 2 3 5  
  
$caractere  
[1] "WW"  "b"   "c"   "d"   "e"
```

Data frames: `data.frame()`

- Um `data frame` é uma lista de vetores de igual comprimento.

```
> nomes <- c('Joao', 'Lara', 'Manoel', 'Pedro', 'Denise')
> idade <- c(30, 43, 21, 34, 25)
> sexo <- c('M', 'F', 'M', 'M', 'M')
> (df <- data.frame(nomes, idade, sexo, stringsAsFactors = F))
   nomes  idade sexo
1    Joao     30    M
2    Lara     43    F
3 Manoel     21    M
4  Pedro     34    M
5 Denise     25    M
```

Acessando data.frame

- Como `data.frame` é um caso especial de uma lista, logo todas as formas de acessar uma lista podem ser usadas com um `data.frame`.

```
> df$nomes      #acessando direto os elementos  
[1] "Joao"     "Lara"      "Manoel"    "Pedro"    "Denise"
```

```
> df[['nomes']] #acessando direto os elementos  
[1] "Joao"     "Lara"      "Manoel"    "Pedro"    "Denise"
```

```
> df['nomes']   #acessando uma coluna, tente class(df['nomes'])  
nomes  
1  Joao  
2  Lara  
3 Manoel  
4 Pedro  
5 Denise
```

Acessando data.frame

- Podemos usar também a mesma forma de acessar matrizes.

```
> df[, 1]          #acessando a coluna 1  
[1] "Joao"    "Lara"    "Manoel"   "Pedro"   "Denise"
```

```
> df[, 'nomes']  #acessando a coluna nome  
[1] "Joao"    "Lara"    "Manoel"   "Pedro"   "Denise"
```

```
> df[2, ]          #acessando a linha 2  
  nomes idade sexo  
2  Lara     43     F
```

```
> df[2:4, 2:3]    #linhas 2,3,4 e colunas 2 e 3  
  idade sexo  
2    43     F  
3    21     M  
4    34     M
```

Algumas funções que são aplicadas sobre `data.frame`

- `head(x)`: apresenta as primeiras 6 linhas
- `str(x)`: estrutura do `data.frame`
- `summary(x)`: resumo do `data.frame`

data.frame: funções

```
> head(df)
  nomes idade sexo
1  Joao     30   M
2  Lara     43   F
3 Manoel    21   M
4 Pedro     34   M
5 Denise    25   M
```

```
> str(df)
'data.frame': 5 obs. of 3 variables:
 $ nomes: chr "Joao" "Lara" "Manoel" "Pedro" ...
 $ idade: num 30 43 21 34 25
 $ sexo : chr "M" "F" "M" "M" ...
```

```
> summary(df)
    nomes           idade          sexo
Length:5           Min.   :21.0   Length:5
Class :character  1st Qu.:25.0   Class :character
Mode  :character  Median :30.0   Mode  :character
                  Mean   :30.6
                  3rd Qu.:34.0
                  Max.   :43.0
```

Agora é a sua vez

Trabalhando com `data.frame`

- Carregue o `data.frame mtcars` usando a função `data(mtcars)`.
- `?mtcars` conheça a base de dados.
- Utilize as funções `str()`, `summary()` e `head()`.
- Cacule a média para cada uma das colunas.

Módulo IV: Loops e condições

Loops: `for()`

- Loops são replicações de uma mesma tarefa para diferentes valores.
- O R possui três opções de loops: `for`, `while`, `repeat`
- **Sintaxe:**

Estrutura do `for`

```
for(var in seq) {  
  tarefa que depende de var  
  ...  
}
```

- A ideia é que o valor do objeto `var` vai variar de acordo com vetor `seq`.
Por exemplo:

```
> for(i in 1:3) {  
+   print(i)  
+ }  
[1] 1  
[1] 2  
[1] 3
```

Loops: `for()`

- Exemplo calculando o desvio-padrão de cada variável do `mtcars`.

```
> desvio <- numeric(ncol(mtcars)) #definindo um vetor numerico
> for(cc in 1:ncol(mtcars)) {
+   desvio[cc] <- sd(mtcars[, cc])
+ }
```

- Ou, podemos passar um vetor de caracteres.

```
> desvio2 <- numeric(ncol(mtcars)) #definindo um vetor numerico
> names(desvio2) <- names(mtcars)
>
> for(cc in names(mtcars)) {
+   desvio2[cc] <- sd(mtcars[, cc])
+ }
> desvio == desvio2
mpg cyl disp hp drat wt qsec vs am gear carb
TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Loops: while

- Sintaxe:

Estrutura do while

```
while(VERDADEIRO) {  
  executa a tarefa  
  ...  
}
```

- Exemplo:

```
> resposta <- ""  
> while(resposta != "R") {  
+   resposta <- readline(prompt = "Qual a é a linguagem franca da ciência dos dados?")  
+ }
```

Loops: repeat

- Sintaxe:

Estrutura do repeat

```
repeat {  
  executa a tarefa  
  ...  
}
```

- Exemplo:

```
> x <- 1  
> repeat {  
+   print(x)  
+   x = x+1  
+   if (x == 6){  
+     break  
+   }  
+ }
```

- Note o comando `if()`

Condições

- No R existe duas principais formas de usar condições: `if(){}` e `ifelse()`.
- **Sintaxe:**

Estrutura do `if()`

```
if(condicao) {  
  
executa ESTA tarefa se a condicao for verdadeira  
  
} else {  
  
caso contrario execute este OUTRA tarefa  
  
}
```

- **Exemplo:**

```
> for(i in 1:10) {  
+   if(i > 7) {  
+     print(i)  
+   }  
+ }  
[1] 8  
[1] 9  
[1] 10
```

- A função `ifelse(teste, sim, nao)` avalia o teste para cada entrada do vetor e executa uma tarefa se for verdadeira ou outra caso contrário.
- Versão vetorializada do tradicional `if(){}``else{}`.
- **Exemplo:**

```
> nomes <- c('Joao', 'Lara', 'Manoel', 'Pedro', 'Denise')
> idade <- c(30, 43, 21, 34, 25)
> sexo <- c('M', 'F', 'M', 'M', 'M')
> df <- data.frame(nomes, idade, sexo, stringsAsFactors = F)
> (df$sexo <- ifelse(df$sexo == 'M', 'Masculino', 'Feminino'))
[1] "Masculino" "Feminino"  "Masculino" "Masculino" "Masculino"
```

Módulo V: Lendo dados de arquivos no formato texto

Lendo arquivo texto

- **Arquivo de texto:** arquivo de texto plano, sem qualquer formatação especial, e pode ser visualizado em qualquer editor de texto simples.
- Para leitura de arquivo texto iremos usar a função `read.table()`.

```
> args(read.table)
function (file, header = FALSE, sep = "", quote = "\"\"", dec = ".",
  numerals = c("allow.loss", "warn.loss", "no.loss"), row.names,
  col.names, as.is = !stringsAsFactors, na.strings = "NA",
  colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
  fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
  comment.char = "#", allowEscapes = FALSE, flush = FALSE,
  stringsAsFactors = default.stringsAsFactors(), fileEncoding = "",
  encoding = "unknown", text, skipNul = FALSE)
NULL
```

```
> ?read.table()
```

Lendo arquivo texto

- Ler um banco de dados com colunas separadas por vírgulas e ponto como separador decimal.

```
> dados <- read.table(file = "../data/reg3.csv",
+                      sep = ",",
+                      dec = ".",
+                      header = TRUE) #cabecalho com o nome das variaveis
```

- **file**: caminho com o nome do arquivo a ser importado.
- **sep**: caractere separador das variáveis (colunas) (Ex.: vírgula)
- **dec**: caractere para casas decimais (Ex.: ponto)
- **header**: se o arquivo contem o nome das variáveis (TRUE)

Agora é a sua vez

Importando dados de arquivo texto

- Importar o banco de dados "`data/Dados_VSB/Dados_Fic_Enf.csv`", onde os campos estão separados por ; e . como separador decimal.
- Quantas colunas e linhas tem este `data.frame`?

Diferentes caracteres representando missing

- Suponha que temos um banco de dados, onde os missing são representados por 99 e 9999 e queremos que o R entenda estes são NA. "Missing" é tratado como uma constante própria no R, esta constante é NA.
- Ler o banco de dados sem avisar o R quem so os missings.

```
> idh99 <- read.table("../data/idh.csv", sep = ";", dec = ".", header = TRUE)
```

- Ler o banco de dados utilizando o parâmetro na.strings.

```
> idh99 <- read.table("../data/idh.csv", sep = ";", dec = ".",  
+ header = TRUE, na.strings = c(99, 9999))
```

Exportando arquivos de texto

- Para salvar um arquivo texto no R tem a mesma lógica da leitura.
- Usaremos a função `write.table()`.

```
> write.table(idh99, "idh99.csv",
+               sep = ";", dec = ".",
+               row.names = FALSE,
+               quote=FALSE)
```

- `row.names`: não inserir o nome das linhas no arquivo de saída.
- `quote`: não colocar aspas nas variáveis do tipo character.

Casos particulares da função `read.table()`

- A função `read.csv()` são para bases de dados em que os campos são separados por vírgula e as casas decimais separadas por ponto.

```
> reg3 <- read.csv("../data/reg3.csv")
```

- A função `read.csv2()` são para bancos de dados em que os campos são separados por ponto e vírgula e as casas decimais separadas por vírgula.
- Para exportar os arquivos também temos casos particulares para a função `write.table()`. Sendo elas, `write.csv()` e `write.csv2()`.

Importando planilha do Excel

- Para importar planilhas do Excel para o R devemos utilizar o pacote `readxl`.

```
> #install.package(readxl)
> library(readxl)
> dados_excel <- read_excel(path = 'data/datasets.xlsx', sheet = 1)
> ?read_excel
```

- `path`: caminho para a planilha de dados
- `sheet`: nome ou número da aba na planilha

Módulo VI: Funções

Definições básicas

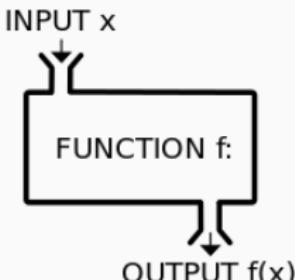
Para que serve uma função?

- Existem procedimentos que são repetidos diversas vezes em um *script*
 - Encontrar o maior/menor valor em uma lista
 - Encontrar a média de cada coluna de uma matriz
 - Fazer o resumo estatístico de uma variável
 - ...
- Nestes casos é possível automatizar o processo através de uma função
- Com isso o código fica mais enxuto, simples e legível
- Outro ponto positivo é que dada uma alteração na função não é necessário alterar o código em diversos pontos

Estrutura de uma função

Estrutura de uma função

```
NomeDaFuncao <- function(arg1, arg2, ...){  
  procedimento1  
  ...  
  procedimenton  
  return(resultado)  
}
```



Componentes de uma função

Uma função é composta basicamente de três partes:

Partes de uma função

- **body**: Representa os procedimentos que a função executa
- **formals**: Representa os argumentos de input da função
- **environment**: Representa o ambiente em que a função está definida

Considere a seguinte função de exemplo:

```
> produto <- function(x, n){  
+   resultado <- n*x  
+   return(resultado)  
+ }
```

Componentes de uma função

```
> formals(produto)
$x

$n
```

```
> body(produto)
{
  resultado <- n * x
  return(resultado)
}
```

```
> environment(produto)
<environment: R_GlobalEnv>
```

Ambiente global, pacotes e funções primitivas

Funções do ambiente global (criadas pela usuário)

```
> environment(produto)          # Ambiente global  
<environment: R_GlobalEnv>
```

Funções relacionadas à algum pacote.

```
> environment(ggplot2::ggplot)    # Pacote ggplot2  
<environment: namespace:ggplot2>
```

Funções primitivas não possuem um ambiente dentro do R.

```
> environment(sum)              # Função primitiva  
NULL
```

Exemplo

```
> divisores <- function(x = 10){  
+   resposta <- rep(0, x-1)  
+   for(i in 2:(x-1)){  
+     resposta[i-1] <- ifelse(x%%i == 0, 1, 0)}  
+   divisores <- which(resposta == 1)+1  
+   if(resposta[1] == 1){  
+     par_impar <- "Par"  
+   }else{  
+     par_impar <- "Ímpar"  
+   }  
+   return(list(div = divisores, par_impar = par_impar))  
+ }
```

Exemplo

```
> divisores(x = 4)
$div
[1] 2

$par_impar
[1] "Par"
```

```
> divisores(x = 10)
$div
[1] 2 5

$par_impar
[1] "Par"
```

```
> environment(produto)
<environment: R_GlobalEnv>
```

Lista de todas as funções do R (base):

▶ Funções do R-base

- A base do R conta com muitas funções matemáticas
- Além disso existem funções para:
 - Manipulação de dados numéricos
 - Manipulação de dados textuais
 - Manipulação de datas
 - ...
- Grande parte das funções são vetorizadas
- Exemplos:
 - `log()`, `sqrt()`, `cos()`, `sin()`, `factorial()`, ...

Variáveis locais e globais

Variáveis criadas dentro de uma função não são acessadas no ambiente global

```
> f1 <- function(a){  
+   b <- 10  
+   resultado <- b-a  
+   return(resultado)  
+ }  
>  
> f1(a = 8)  
[1] 2
```

```
> b  
Error in eval(expr, envir, enclos): object 'b' not found
```

Ou seja, as variáveis estão definidas apenas dentro do escopo da função

Variáveis locais e globais

Qual o resultado da seguinte função:

```
> b <- 10
>
> f1 <- function(a){
+   resultado <- b-a
+   return(b-a)
+ }
>
> f1(a = 8)
```

Variáveis locais e globais

Qual o resultado da seguinte função:

```
> b <- 10
>
> f1 <- function(a){
+   resultado <- b-a
+   return(b-a)
+ }
>
> f1(a = 8)
```

```
[1] 2
```

Variáveis locais e globais

Qual o resultado da seguinte função:

```
> b <- 10
>
> f1 <- function(a){
+   resultado <- b-a
+   return(b-a)
+ }
>
> f1(a = 8)
```

```
[1] 2
```

Uma variável não definida mas utilizada dentro de uma função é procurada no ambiente global!!!

Variáveis locais e globais

Para atualizar um objeto do escopo global de dentro da uma função utiliza-se o operador '`<<-`'

```
> b <- 10
> val <- 10
>
> f1 <- function(a){
+   val <<- 20
+   return(b-a)
+ }
>
> f1(a = 10)
[1] 0
```

```
> val
[1] 20
```

Agora é a sua vez

Criando uma função

Crie uma função que recebe um vetor numérico de qualquer tamanho e retorna uma lista contendo:

- O tamanho do vetor (função `length()`)
- A soma do vetor (função `sum()`)
- Se o tamanho do vetor for par:
 - Retornar o primeiro elemento
- Se o tamanho do vetor for ímpar:
 - Retornar o último elemento

Solução

```
> RetornaLista <- function(x){  
+   tam <- length(x)  
+   soma <- sum(x)  
+   par <- (tam%/%2 == 0)  
+  
+   if(par){  
+     elemento <- x[1]  
+   } else{  
+     elemento <- x[tam]  
+   }  
+  
+   resultado <- list(tamanho = tam,  
+                      soma = soma,  
+                      elemento = elemento)  
+   return(resultado)  
+ }
```

Solução

```
> RetornaLista(x = c(1, 3, 5, 7, 9))
$tamanho
[1] 5

$soma
[1] 25

$elemento
[1] 9
```

```
> RetornaLista(x = c(11, 3, 54, 7, 99, 22))
$tamanho
[1] 6

$soma
[1] 196

$elemento
[1] 11
```

Operadores

Operadores

Operadores também são funções, porém podem ser utilizadas de duas formas diferentes:

```
> 2 + 2*(8/4)
[1] 6
```

```
> '+'(2, '*'(2, '/'(8, 4)))
[1] 6
```

```
> 3 > 4
[1] FALSE
```

```
> '>'(3, 4)
[1] FALSE
```

Operadores

O mesmo vale para outras funções básicas

```
> i <- 1
> x <- letters[1:3]
>
> 'if'(x[i] == "a", print("Sim!"), print("Não"))
[1] "Sim!"
```

```
> 'for'(i, x, print(i))
[1] "a"
[1] "b"
[1] "c"
```

```
> '['(x, 3)
[1] "c"
```

Criando operadores

Para criar um operador basta criar uma função entre percentuais '%funcao%'

```
> '%soma_mult%' <- function(x, y){  
+   z <- x + y  
+   result <- x*z  
+   return(result)  
+ }  
>  
> 4 %soma_mult% 6  
[1] 40
```

```
> 10 %soma_mult% 2  
[1] 120
```

Tipos de *inputs*

Listas e funções

As funções em R podem receber qualquer tipo de estrutura (argumentos, caracteres, listas e até mesmo outras funções)

```
> ApplyFunc2List <- function(lista, funcao){  
+   dimensao <- length(lista)  
+   resultado <- rep(0, dimensao)  
+  
+   for(i in 1:dimensao){  
+     resultado[i] <- funcao(lista[[i]])  
+   }  
+  
+   return(resultado)  
+ }  
>  
> lista <- list(a = c(1, 2, 3), b = c(4, 5, 6))  
>  
> ApplyFunc2List(lista = lista, funcao = function(x) x[1])  
[1] 1 4
```

Argumentos *default*

Quando criamos uma função com diversos parâmetros é necessário definir o valor de cada um deles...

```
> raiz <- function(pNum, pRaiz)
+ {
+   pNum^(1/pRaiz)
+ }
>
> raiz(pNum = 4)
Error in raiz(pNum = 4): argument "pRaiz" is missing, with no default
```

Argumentos *default*

Porém, nem sempre necessitamos alterar todos os parâmetros de uma função. Nestes casos, podemos criar parâmetros padrões que só são alterados quando o usuário solicita.

```
> raiz <- function(pNum, pRaiz = 2)
+ {
+   pNum^(1/pRaiz)
+ }
>
> raiz(pNum = 4)
[1] 2
```

Argumentos *default*

```
> raiz(pNum = 4)
[1] 2
```

```
> raiz(pNum = 4, pRaiz = 2)
[1] 2
```

```
> raiz(pNum = 4, pRaiz = 15)
[1] 1.096825
```

```
> raiz(pNum = 4, pRaiz = 100)
[1] 1.013959
```

O argumento ‘...’

Muitas vezes utilizamos funções secundárias para criação de uma função. Quando o interesse não são os parâmetros da função secundária podemos utilizar o argumento ‘...’

```
> SalvaBase <- function(file, ...){  
+   data(cars)  
+   write.table(x = cars, file = file, ...)  
+   return("A base foi salva com sucesso")  
+ }  
>  
> SalvaBase(file = "../data/cars.csv",  
+             sep = ";", dec = ".")  
[1] "A base foi salva com sucesso"
```

Desta forma o código fica mais enxuto e mais flexível para o usuário.

Tipos de *outputs*

Tipos de *outputs*

Os *outputs* podem ser os mais variados possíveis, podendo retornar um único resultado ou diversos (através de uma lista). Além disso, os resultados podem ser de diferentes tipos (listas, matrizes, funções, ...).

```
> RetornaObjetos <- function(ncol, nrow){  
+   matriz <- matrix(0, ncol = ncol, nrow = nrow)  
+   lista <- list(ncol = ncol, nrow = nrow)  
+   funcao <- function(ncol, nrow){ncol*nrow}  
+  
+   return(list(matriz = matriz,  
+               lista = lista,  
+               funcao = funcao))  
+ }
```

Tipos de *outputs*

```
> RetornaObjetos(ncol = 4, nrow = 1)
$matriz
 [,1] [,2] [,3] [,4]
[1,]    0    0    0    0

$list
$list$ncol
[1] 4

$list$nrow
[1] 1

$funcao
function (ncol, nrow)
{
  ncol * nrow
}
<environment: 0x5080e38>
```

Mensagens de erro

A função `warning()`

A função `warning()` serve para auxiliar o programador nos casos em que ocorrem pequenos imprevistos. Esses imprevistos não param o programa, porém é importante que o usuário tenha atenção e saiba que os resultados subsequentes podem conter erros.

```
> log(-1)
Warning in log(-1): NaNs produced
[1] NaN
```

Uso:

```
> calcula_razao <- function(x, y){
+   if(y == 0){
+     warning("y = 0 -> Divisão por zero!")
+   }
+   +
+   return(x/y)
+ }
```

A função *warning()*

```
> calcula_razao(x = 10, y = 10)
[1] 1
```

```
> calcula_razao(x = 10, y = 5)
[1] 2
```

```
> calcula_razao(x = 10, y = 10e-10)
[1] 1e+10
```

```
> calcula_razao(x = 10, y = 0)
Warning in calcula_razao(x = 10, y = 0): y = 0 -> Divisão por zero!
[1] Inf
```

A função `stop()`

A função `stop` pode ser utilizada para tornar suas funções mais robustas e trazer mais informações aos usuários. O objetivo desta função é parar o processo dado algum evento e retornar uma mensagem de informação.

Exemplo:

```
> nchar_function <- function(char){  
+   if(!is.character(char)){  
+     stop("O input deve ser um caracter")  
+   }  
+   return(nchar(char))  
+ }  
>  
> nchar_function(char = "VSB")  
[1] 3
```

```
> nchar_function(char = 10)  
Error in nchar_function(char = 10): O input deve ser um caracter
```

Criando um log de erros

Em alguns problemas práticos erros são esperados, pois, nem sempre é possível prever todos as possibilidades de erros. Neste caso salvar um arquivo contendo os erros e *warnings* obtidos é muito útil e interessante. Podemos configurar o R para salvar um log de erros:

```
> funcao_erro <- function() {  
+   cat(geterrmessage(), file = "../data/error.txt",  
+       append = T)  
+ }  
>  
> options("error" = funcao_erro)  
>  
> 1 + "2"  
Error in 1 + "2": non-numeric argument to binary operator
```

Agora é a sua vez

Criando uma função complexa

Crie uma função com um argumento padrão e que recebe o argumento "...", essa função deve:

- Receber três vetores e montar uma matriz (cbind ou rbind)
- Caso a dimensão dos vetores seja diferente o programa deve parar
- Retornar a matriz construída
- Retornar um vetor com a soma de cada coluna

Solução

```
> RetornaSoma <- function(vet1 = rep(1, 5), vet2 = vet1,
+                           vet3 = vet1,
+                           ...){
+   if(!(length(vet1) == length(vet2) &
+        length(vet2) == length(vet3))){
+     stop("\n Os vetores não tem a mesma dimensão")
+   }
+   matriz <- cbind(vet1, vet2, vet3)
+   nCols <- ncol(matriz)
+   somaCols <- rep(0, nCols)
+
+   for(i in 1:nCols){
+     somaCols[i] <- sum(matriz[,i])
+   }
+
+   return(list(matriz = matriz,
+              soma = somaCols))
+ }
```

Solução

```
> RetornaSoma(vet1 = c(1,2,3), vet2 = c(4, 5, 6),
+               vet3 = c(7, 8, 9), sep = ";", dec = ".")
$matriz
      vet1  vet2  vet3
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

$soma
[1]  6 15 24
```

```
> RetornaSoma(vet1 = c(1,2,3), vet2 = c(4, 5, 6),
+               vet3 = c(7, 8), sep = ";", dec = ".")
Error in RetornaSoma(vet1 = c(1, 2, 3), vet2 = c(4, 5, 6), vet3 = c(7, :
  Os vetores não tem a mesma dimensão
```

Documentação de funções

Documentação de funções

Para entender uma função, seus argumentos e seus *outputs* pode-se consultar a documentação das funções.

Normalmente estas documentações possuem exemplos que auxiliam no entendimento da função.

```
> help(merge)
>
> ?merge
>
> ??merge
```

Veja por exemplo a função '[??expand.grid](#)'

Módulo VII: Estatística elementar

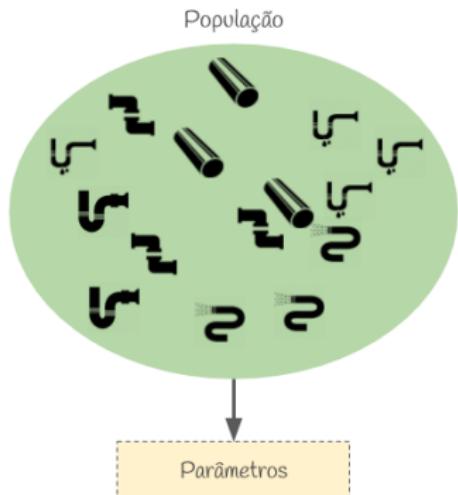
Revisão

A estatística é composta por um conjunto de métodos destinados à coleta, à apresentação, à análise e à interpretação de dados, tendo como objetivo compreender fenômenos de natureza aleatória e ajudar o usuário na tomada de decisão.

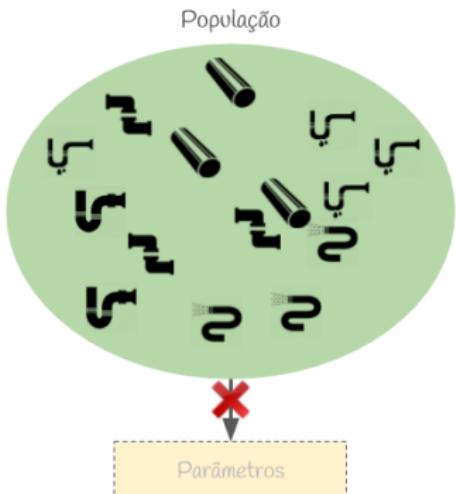
Palavras-chave

- **Variável:** O objeto/fenômeno/característica de interesse em estudo
- **População:** Conjunto de todas as observações da variável em estudo
- **Amostra:** Subconjunto de observações retiradas da população
- **Parâmetro:** Uma característica da população
- **Inferência:** Com base numa amostra inferimos os parâmetros associados à população

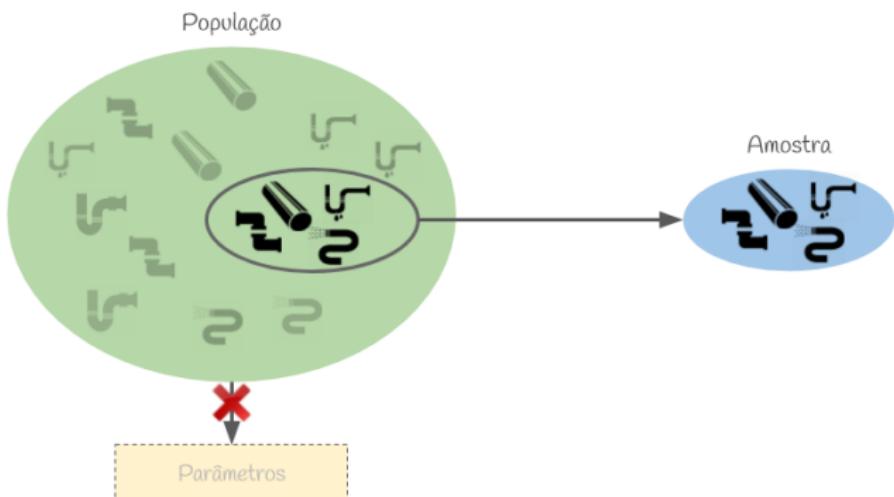
Estatística em um *slide*



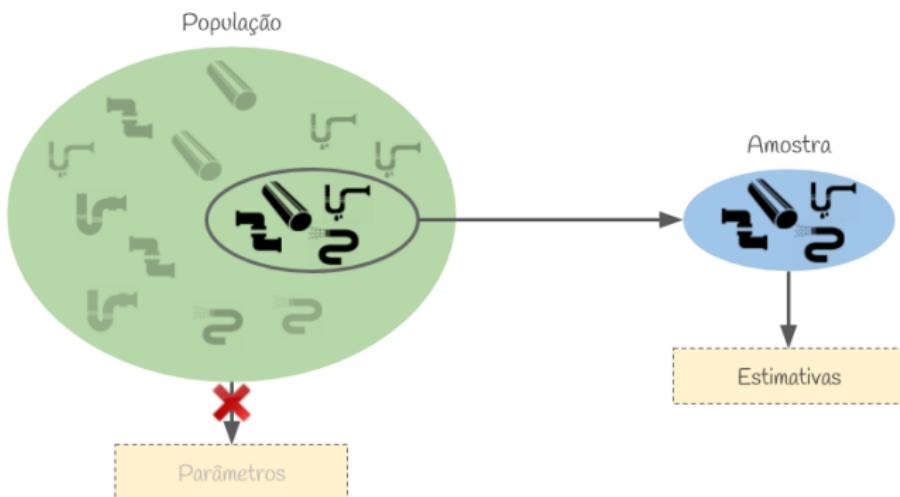
Estatística em um *slide*



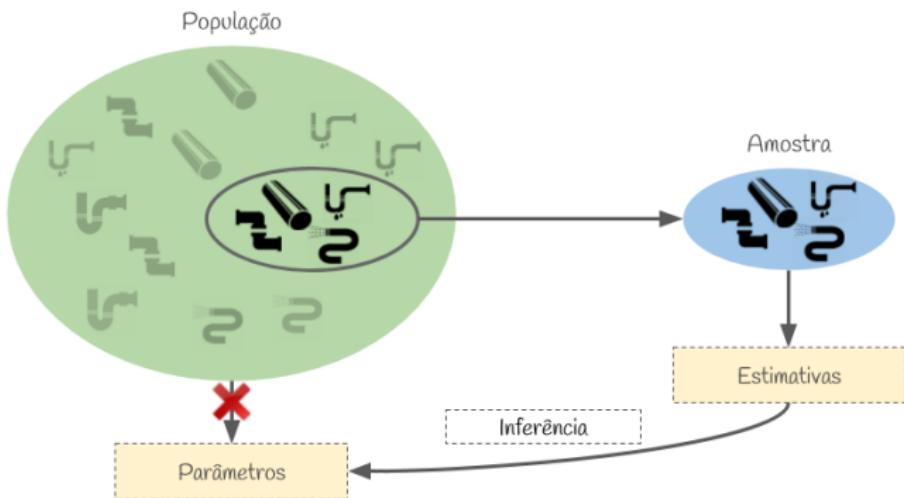
Estatística em um slide



Estatística em um slide



Estatística em um slide



Tipos de variáveis

Variáveis quantitativas

Contínuas



Discretas



Variáveis qualitativas

Ordinais



Nominais



Para cada tipo de variável, diferentes tipos de análises (e gráficos) são indicados.

Estatística descritiva

A estatística descritiva é uma área da estatística responsável por descrever e sumarizar um conjunto de dados.

Estatística descritiva

- **Medidas de tendência central:** Média, moda, mediana, ...
- **Medidas de variabilidade:** Variância, desvio padrão, amplitude, ...
- **Gráficos:** Boxplot, histograma, gráfico de barras, ...

Base para exemplificação

Vamos trabalhar com a base `dadosForno` e principalmente com as variáveis `PRS_GAS` e `PRS_FORNO`.

```
> load("/home/douglas/Documents/Dropbox/Cursos/Curso_R/data/Dados_VSB/dadosForno.RData")
> PRS_GAS <- dadosForno$PRS_GAS
> head(PRS_GAS)
[1] 100.1 99.5 100.1 99.9 99.5 99.9
```

```
> PRS_FORNO <- dadosForno$PRS_FORNO
> head(PRS_FORNO)
[1] 0.011 0.012 0.014 0.007 0.006 0.007
```

```
> miniBase <- cbind(PRS_FORNO,
+                     PRS_GAS)
```

Medidas de tendência central

As medidas de tendência central nos dão uma ideia de onde os dados estão concentrados.

Medidas de tendência central

- `mean`: Média de um vetor
- `median`: Mediana de um vetor
- `table`: Tabela de frequências/contingência
- `mlv`: Moda de um vetor
- `colSums`: Soma por colunas de base de dados
- `colMeans`: Média por coluna de uma base de dados

Medidas de variabilidade

As medidas de variabilidade nos dão uma ideia da dispersão dos dados.

Medidas de variabilidade

- `min`: Menor valor de um vetor
- `max`: Maior valor de um vetor
- `which.min`: Posição em que o mínimo ocorre
- `which.max`: Posição em que o máximo ocorre
- `quantile`: Quantis de um vetor
- `var`: Variância de uma matriz/vetor
- `sd`: Desvio padrão de um vetor
- `cov`: Covariância entre duas variáveis
- `cor`: Correlação entre duas ou mais variáveis

Resumo dos dados

Uma maneira simples de resumir os dados é utilizando a função `summary`

```
> summary(miniBase)
    PRS_FORNO          PRS_GAS
Min.   :-0.037000   Min.   :  0.00
1st Qu.: 0.007000   1st Qu.: 91.40
Median  : 0.010000   Median : 95.30
Mean    : 0.009762   Mean   : 95.63
3rd Qu.: 0.012000   3rd Qu.:100.00
Max.   : 0.042000   Max.   :125.10
```

Agora é a sua vez

Análise descritiva

Utilizando a base de dados **dadosForno** encontre (quando possível):

- A média, mediana e variância
- O maior e o menor valor juntamente com a data em que entraram no forno
- Existem valores atípicos? O que ocorre com a média retirando 1
- Divida o desvio padrão pela média e multiplique por 100. O que podemos concluir?

Utilize as variáveis:

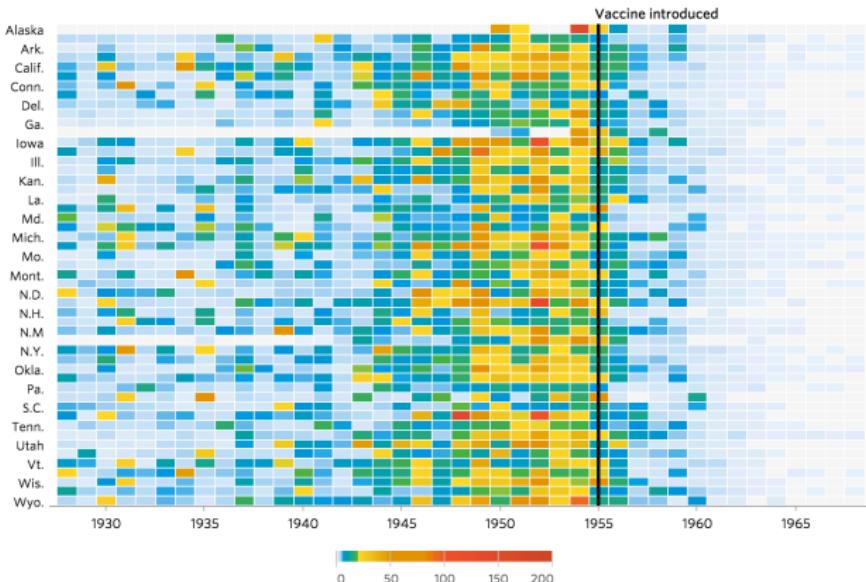
- **DIS_ULT_BLOCO_ENFORNADO**,
- **CMP_BLOCO_ENFORNAMENTO** e
- **PER_02_CANAL_FUMOS**

Gráficos estatísticos

Gráficos estatísticos

Visualizar os dados ajuda no entendimento do problema e também na apresentação dos resultados. Sendo assim é parte **essencial** de uma análise.

Polio



Gráficos estatísticos

Gráficos mostram de uma forma clara e objetiva o comportamento dos dados.

Gráficos estatísticos

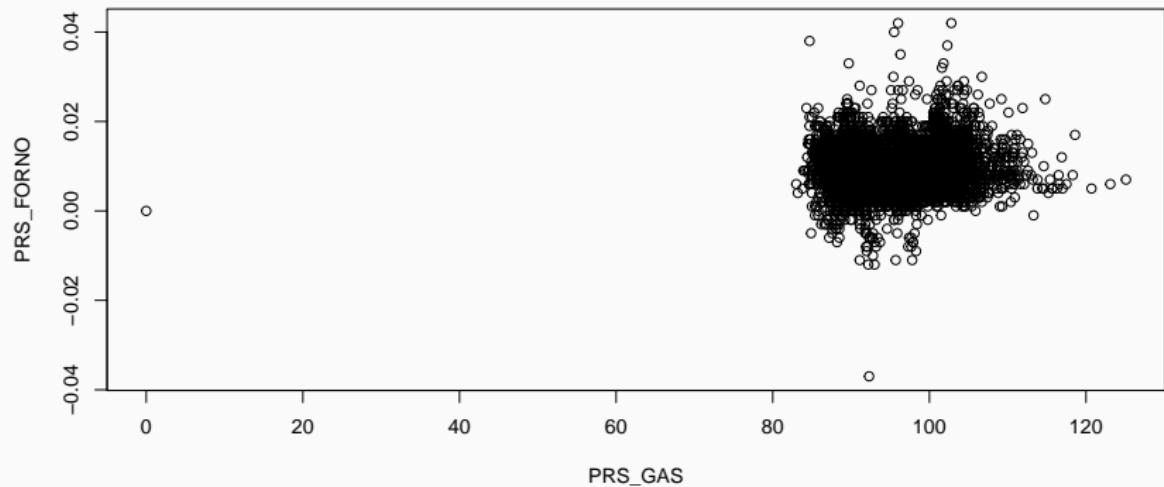
- Dispersão
- Linhas
- Barras
- Setores/Pizza
- Histograma
- Boxplot

Veremos no **próximo módulo** como **customizar** estes e outros gráficos.

Gráficos estatísticos - Dispersão

Gráfico de dispersão: Gráfico contrapondo duas variáveis que pode evidenciar correlação entre as variáveis.

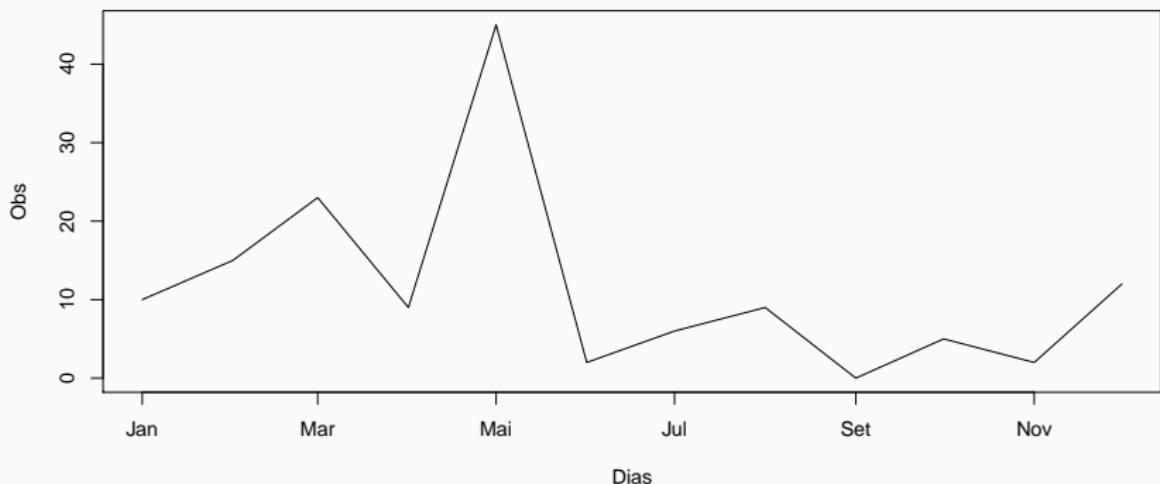
```
> plot(x = PRS_GAS, y = PRS_FORNO)
```



Gráficos estatísticos - Linhas

Gráfico de linhas: Gráfico contrapondo duas variáveis em que normalmente o eixo x apresenta uma variável temporal.

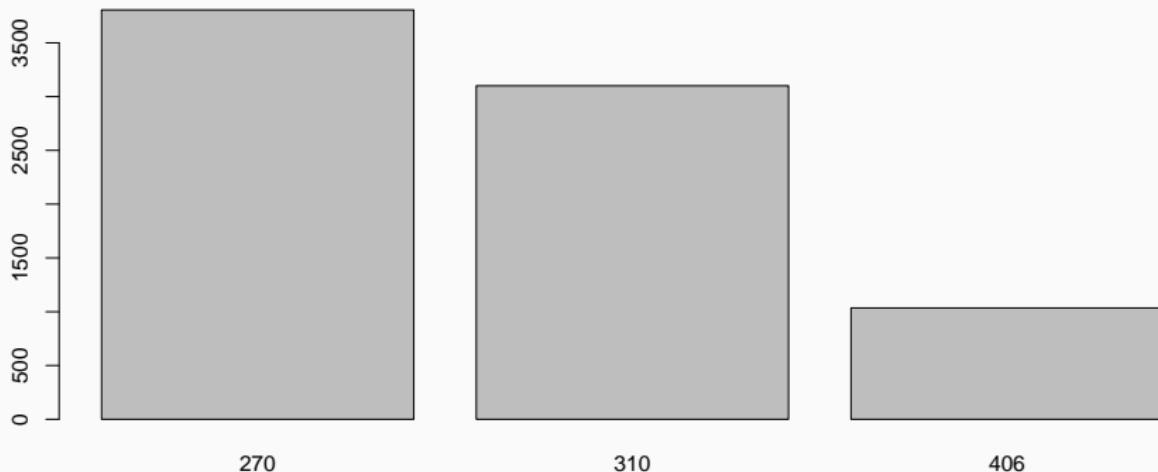
```
> Dias <- seq(as.Date("2016/01/01"), by = "month", length.out = 12)
> Obs <- c(10, 15, 23, 9, 45, 2, 6, 9, 0, 5, 2, 12)
> plot(x = Dias, y = Obs, type = "l")
```



Gráficos estatísticos - Barras

Gráfico de barras: É uma forma de resumir um conjunto de dados categóricos. Cada barra representa uma categoria particular e a altura de cada barra é proporcional à frequência de cada categoria.

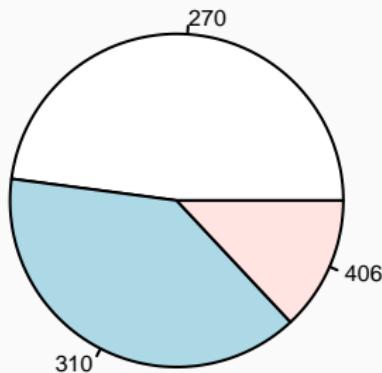
```
> DIM_BLOCO_ENFORNAMENTO <- dadosForno$DIM_BLOCO_ENFORNAMENTO  
> cont <- table(DIM_BLOCO_ENFORNAMENTO)  
> barplot(height = cont)
```



Gráficos estatísticos - Setores/Pizza

Gráfico de setores: Útil para visualizar a participação de cada categoria no total geral.

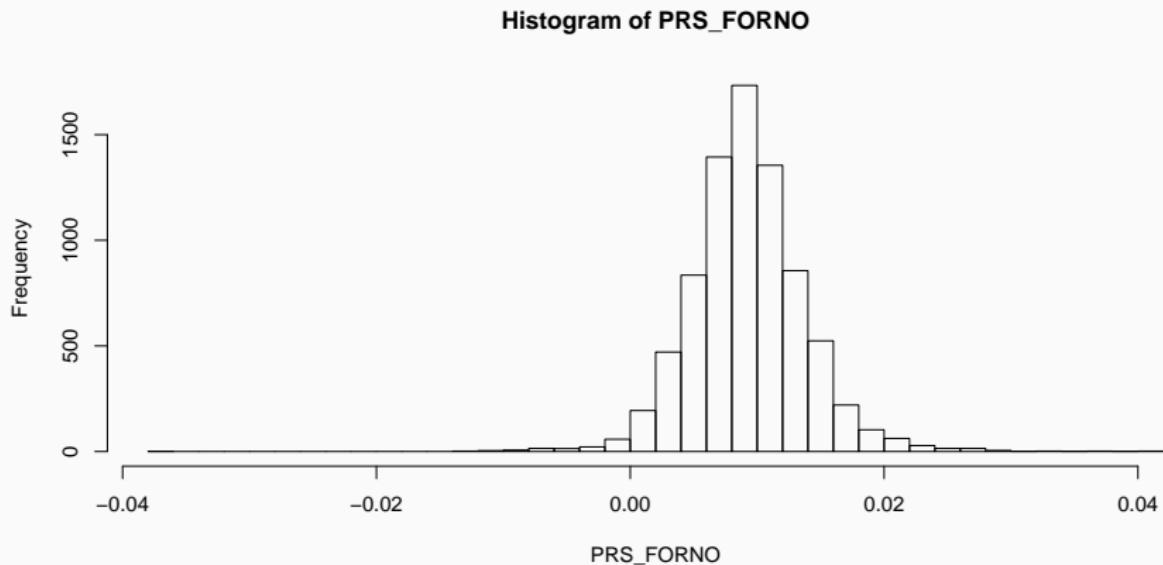
```
> pie(x = cont, cex = 0.5)
```



Gráficos estatísticos - Histograma

Histograma: Útil para caracterizar a distribuição dos dados.

```
> hist(PRS_FORNO, breaks = 50)
```

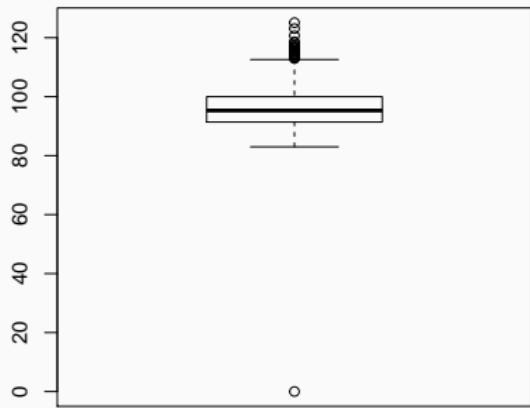


Gráficos estatísticos - Boxplot

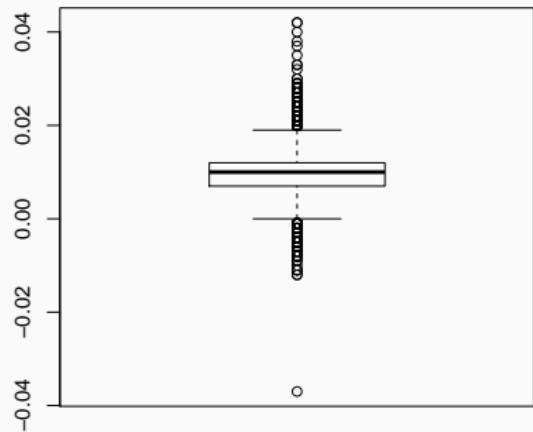
Boxplot: Útil para caracterizar a distribuição dos dados. Cada elemento do gráfico representa uma estatística e os pontos os *outliers*.

```
> par(mfrow=c(1,2))
> boxplot(PRS_GAS, main = "PRS_GAS")
> boxplot(PRS_FORNO, main = "PRS_FORNO")
```

PRS_GAS



PRS_FORNO



Agora é a sua vez

Gráficos descritivos

Utilizando a base de dados `dadosForno` e as variáveis `DIS_ULT_BLOCO_ENFORNADO`, `CMP_BLOCO_ENFORNAMENTO` e `PER_02_CANAL_FUMOS`, faça:

- O gráfico de dispersão entre DIS e CMP
- O histograma e o boxplot das variáveis DIS e CMP
- O gráfico de barras e de setores da variável PER

Distribuições de probabilidade

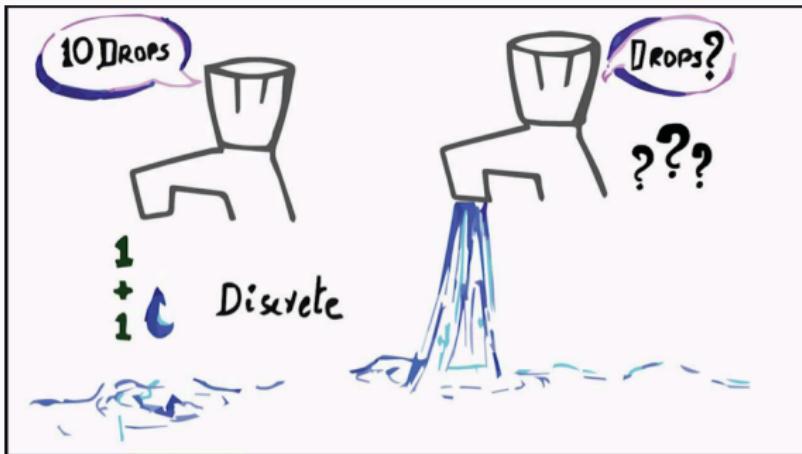
Distribuições de probabilidade

Distribuições contínuas

Assumem valores em uma escala contínua. Exemplo: Altura

Distribuições discretas

Assumem um número finito de valores. Exemplo: Número de filhos



Distribuições discretas

- `sample`: Uniforme
- `binom`: Binomial
- `multinom`: Multinomial
- `pois`: Poisson
- `geom`: Geométrica
- `nbinom`: Binomial negativa
- ...

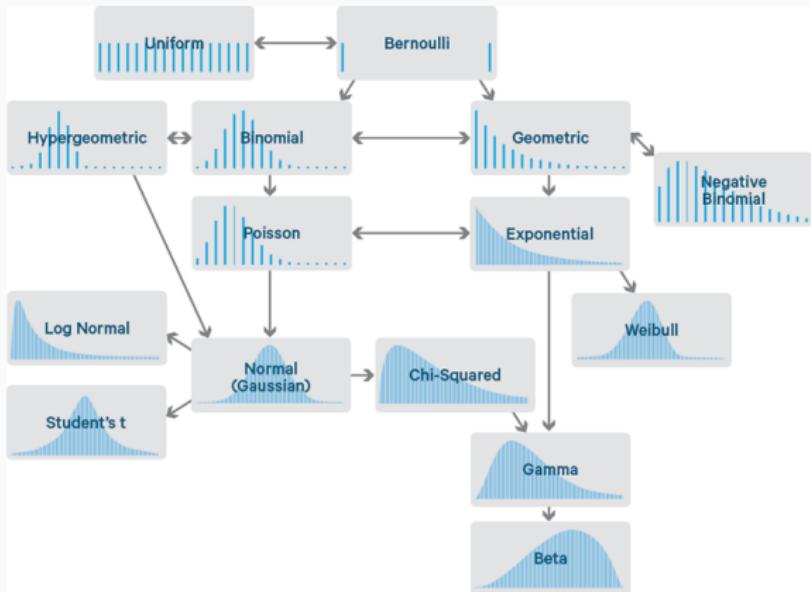
Distribuições contínuas

- `unif`: Uniforme
- `norm`: Normal
- `exp`: Exponencial
- `chisq`: Qui-quadrado
- `t`: t-student
- `f`: F de snedecor
- ...

O uso de uma distribuição de probabilidade depende da natureza da variável em análise. Para isso é interessante conhecer o suporte das distribuições de probabilidade.

Distribuições de probabilidade

Existem muitas (**Muitas mesmo**) distribuições de probabilidade. Na prática algumas delas são mais utilizadas (como as citadas no *slide* anterior).



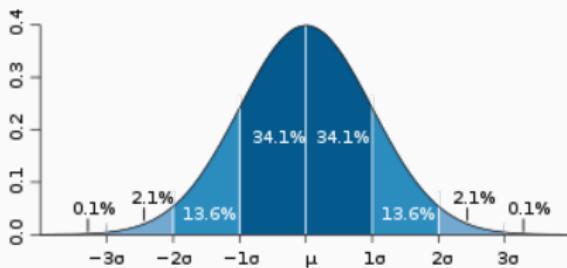
Distribuições de probabilidade

No R as distribuições são utilizadas com um dos prefixos abaixo.

Distribuições de probabilidade no R

- Prefixo **d**: Densidade
- Prefixo **q**: Quantil
- Prefixo **p**: Probabilidade
- Prefixo **r**: Números aleatórios

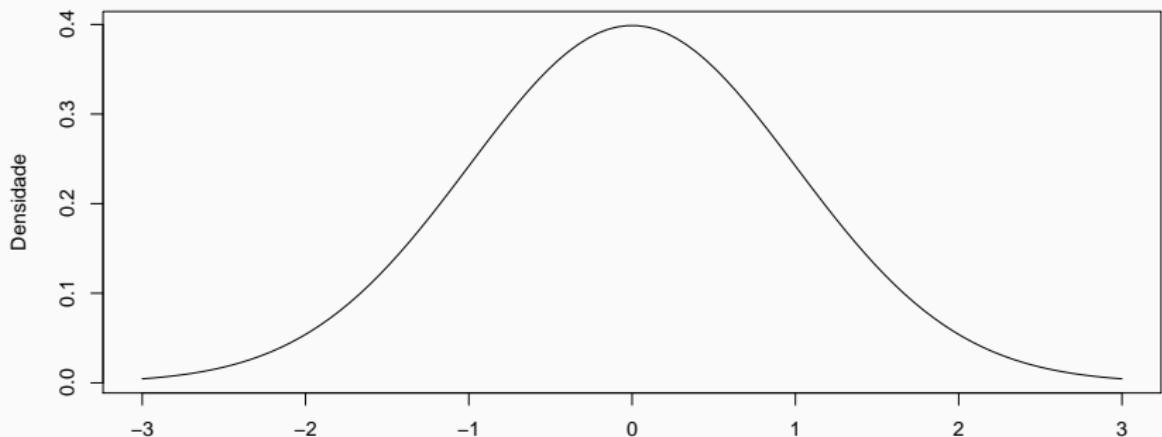
8 2 0 3 1 4 5 8 2 1 7 2 7 3 8 5 5 2 9 0 0 6 3 1 6 4
0 2 3 3 8 6 2 1 4 2 4 0 2 6 6 9 8 0 3 6 2 5 2 7 5 2
4 7 3 6 6 3 0 7 7 1 9 1 6 1 7 4 7 1 3 7 9 3 0 0
1 9 3 9 5 3 4 9 5 5 2 7 5 8 0 3 4 8 8 1 2 7 5 3 4
2 8 7 8 1 4 1 4 9 4 2 4 1 5 2 9 4 6 2 1 5 2 8 1 9
8 4 6 5 3 1 3 9 6 6 0 7 2 1 9 0 2 0 6 7 0 6 0 7 3 0
0 3 5 2 1 9 3 7 4 9 5 0 2 0 5 6 2 5 4 6 6 6 6 7 4
3 4 5 2 1 9 3 7 4 9 5 0 2 0 5 6 2 5 4 6 6 6 7 4
3 4 5 2 1 9 3 7 4 9 5 0 2 0 5 6 2 5 4 6 6 6 7 4
4 1 5 3 0 9 1 3 7 2 5 8 7 7 1 3 6 3 9 7 8 7 9 1 7
7 2 9 5 6 7 8 5 4 5 3 3 4 5 4 1 9 8 6 6 7 5 7 9 3 1 8
5 9 2 8 9 8 6 4 4 4 1 5 3 5 0 0 8 0 2 5 6 0 6 1 2 0
1 2 4 5 6 2 5 2 1 0 0 5 0 3 5 4 9 0 3 4 5 5 7 0 8
8 0 0 1 0 8 6 2 5 2 1 0 0 5 0 3 5 4 9 0 3 4 5 5 7 0 1
7 7 0 6 6 3 2 8 8 5 8 9 5 6 4 0 3 9 1 8 0 3 4 9 4
3 3 8 5 7 5 7 4 3 4 5 7 9 6 9 5 0 7 7 6 6 8 8 5 9
9 1 7 2 3 6 9 2 9 9 4 2 3 3 3 0 7 7 7 6 4 7 2
6 2 2 8 0 9 4 5 3 7 2 5 4 6 6 5 6 6 5 0 4 6 5 6 6 8
1 7 3 9 0 0 2 0 5 6 5 8 5 1 9 5 3 3 7 4 0 5 8 2 4
0 3 9 6 9 4 7 3 5 7 0 6 5 4 7 1 1 8 5 3 2 8 0 9 8



Distribuições de probabilidade - O prefixo d

O prefixo 'd' seguido do nome de uma distribuição corresponde à **densidade** da variável.

```
> x <- seq(-3, 3, length.out = 100)
> densidade <- dnorm(x = x, mean = 0, sd = 1)
> plot(x = x, y = densidade, type = "l",
+       xlab = "", ylab = "Densidade")
```



Distribuições de probabilidade - O prefixo q

O prefixo ‘q’ seguido do nome de uma distribuição corresponde ao **quantil** da variável.

```
> quantil1 <- qnorm(p = 0.025, mean = 0, sd = 1)
> quantil1
[1] -1.959964
```

```
> quantil2 <- qnorm(p = 0.975, mean = 0, sd = 1)
> quantil2
[1] 1.959964
```

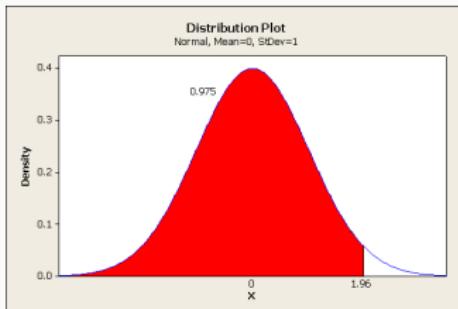
Ou seja, em uma distribuição Normal padrão, a probabilidade acumulada em $(-\infty, -1.96] \cup [1.96, \infty)$ é de 0.05.

Distribuições de probabilidade - O prefixo p

O prefixo 'p' seguido do nome de uma distribuição corresponde à **probabilidade acumulada** até determinado quantil.

```
> prob1 <- pnorm(q = quantil1, mean = 0, sd = 1)
> prob1
[1] 0.025
```

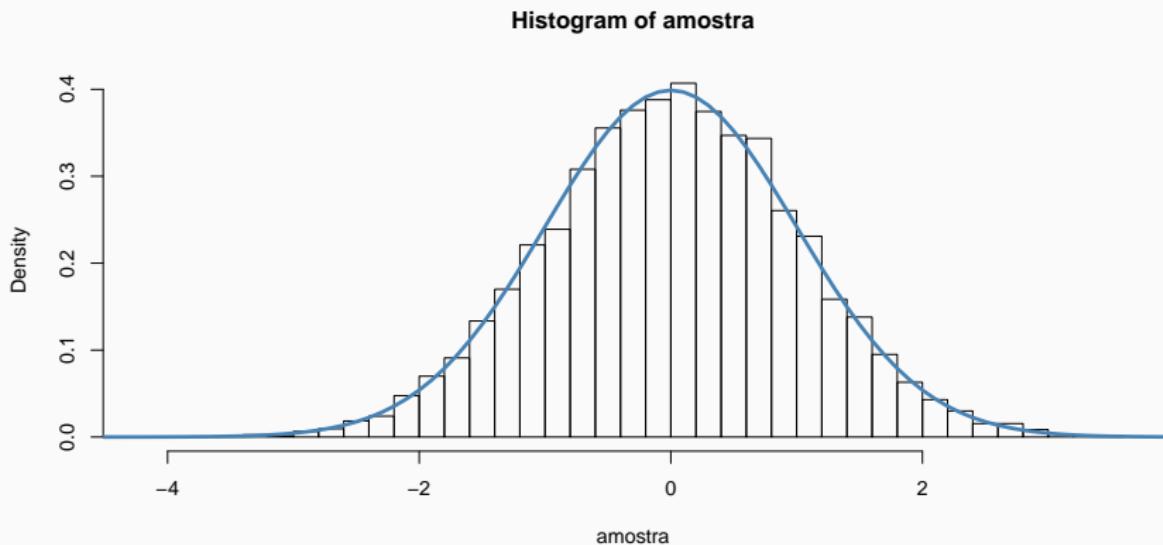
```
> prob2 <- pnorm(q = quantil2, mean = 0, sd = 1)
> prob2
[1] 0.975
```



Distribuições de probabilidade - O prefixo r

O prefixo 'r' seguido do nome de uma distribuição permite a geração de números aleatórios.

```
> amostra <- rnorm(n = 10000, mean = 0, sd = 1)
> hist(amostra, probability = T, breaks = 30)
> curve(dnorm, from = -5, to = 5, add = T,
+         col = "steelblue", lwd = 3)
```



Agora é a sua vez

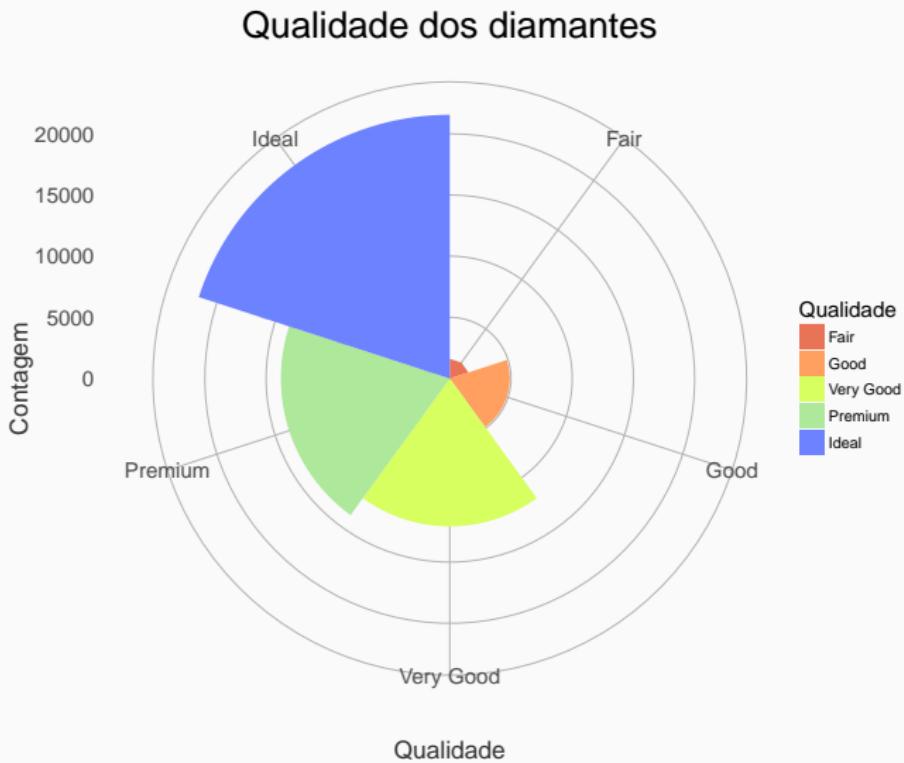
Gráficos de uma distribuição

Utilizando os prefixos 'd', 'q', 'p' e 'r' e se valendo dos nomes das distribuições vistas no slide Distribuições de probabilidade:

- Gere 1.000 observações de duas distribuições à sua escolha
- Faça o histograma e a curva da distribuição (ao mesmo tempo)
- Encontre o quantil 0.025 e o quantil 0.975 destas distribuições
- Qual a proporção de observações maiores e menores que estes quantis (2.5)
- Qual a probabilidade de observarmos valores entre -1 e 1?
- Qual a proporção encontrada na nossa amostra?

Módulo VIII: Visualização de dados

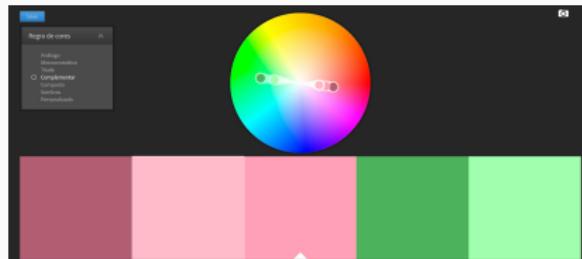
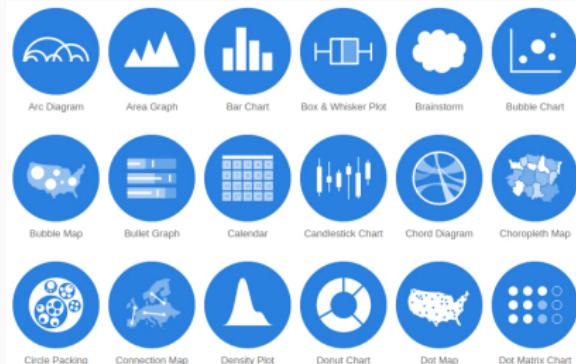
Motivação



Visualização de dados

Duas boas referências

- **Gráficos:** <http://www.datavizcatalogue.com/>
- **Cores:** <https://color.adobe.com/pt/create/color-wheel/>



Algumas pacotes de visualização

- **graphics**: Pacote já na base do R
- **ggplot2**: <http://r4stats.com/examples/graphics-ggplot2/>
- **ggvis**: <http://ggvis.rstudio.com/0.1/quick-examples.html>
- **googleVis**: https://cran.r-project.org/web/packages/googleVis/vignettes/googleVis_examples.html
- **leaflet**: <https://rstudio.github.io/leaflet/>
- **plotly**: <https://plot.ly/r/>
- ...

Parâmetros globais

Parâmetros globais

Para evitar a configuração isolada de cada um dos gráficos, podemos criar parâmetros globais através da função `par`. No decorrer desse módulo vamos conhecer ferramentas e parâmetros interessantes e com isso vamos compor nossos parâmetros globais. O objetivo é terminarmos este módulo com um tema gráfico. Alguns parâmetros interessantes:

Função `par`

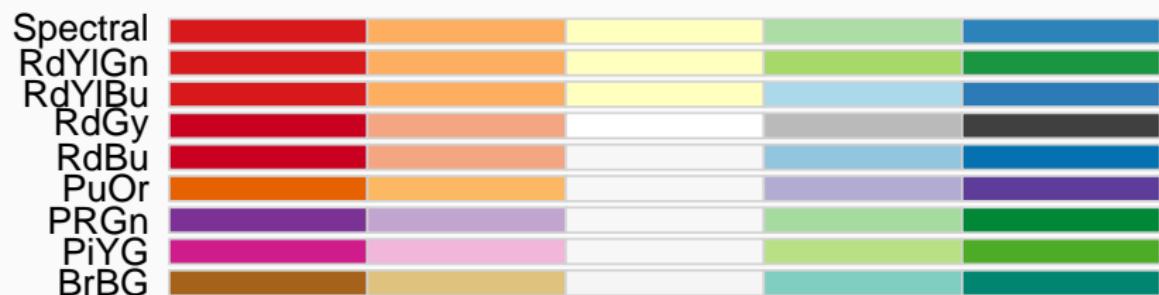
- `bg`: Cor de fundo dos gráficos
- `bty`: Tipo de fundo nos gráficos
- `cex`: Tamanho dos elementos (linhas, pontos)
- `cex.axis`: Tamanho dos elementos do eixo x e y
- `col`: Cor *default* dos elementos gráficos
- `fg`: Cor da linha dos eixos
- `Veja mais em`: <https://stat.ethz.ch/R-manual/R-devel/library/graphics/html/par.html>

Customização de cores

Customização de cores

Existem muitas cores pré-definidas no R (veja em [Cores R](#)). Demais cores são passadas para as funções em formato hexadecimal, ou seja, assumem os seguintes valores: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

- Uma cor é composta de 6 símbolos
- Dois símbolos extras se referem à [transparência](#)
- Existem algumas paletas pré-definidas. (<https://cran.r-project.org/web/packages/RColorBrewer/RColorBrewer.pdf>)



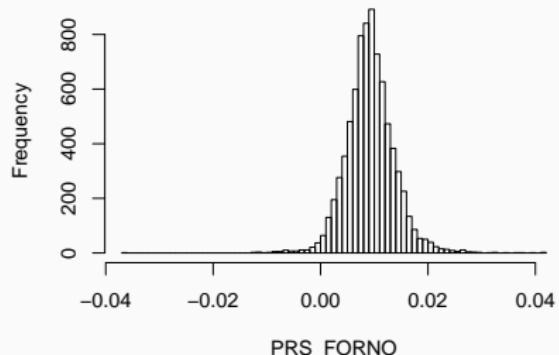
Configuração de cores

- `col`: Cor dos elementos gráficos
- `col.axis`: Cor das anotações dos eixos
- `col.lab`: Cor dos *labels* dos eixos
- `col.main`: Cor do título
- `bg`: Cor de fundo do gráfico
- `fg`: Cor da linha dos eixos
- ...

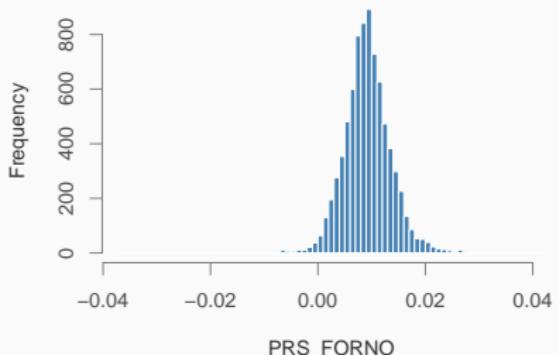
Customização de cores

```
> par(mfrow=c(1, 2))
> hist(PRS_FORNO, breaks = 100)
> par(bg = "grey95", fg = "grey50",
+     col.axis = "grey30", col.lab = "grey20")
> hist(PRS_FORNO, breaks = 100, col = "#4682b4", border = "white")
```

Histogram of PRS_FORNO



Histogram of PRS_FORNO



Customização de títulos, *labels* e fontes

Customização de títulos, *labels* e fontes

O R conta com vários tipos de fontes para serem utilizados em seus gráficos.

- **Fontes padrão:** 'serif', 'sans' e 'mono'
- **Hershey:** <https://stat.ethz.ch/R-manual/R-devel/library/grDevices/html/Hershey.html>
- **extrafont:** Pacote que permite utilizar suas fontes preferidas

```
> library(extrafont)
> loadfonts() #--- font_import()
> head(fonts(), n = 10)
[1] "Padauk"                  "Padauk Book"           "Abyssinica SIL"
[4] "Andale Mono"              "Arial Black"            "Arial"
[7] "Bitstream Vera Sans"     "Bitstream Vera Sans Mono" "Bitstream Vera Serif"
[10] "cmex10"
```

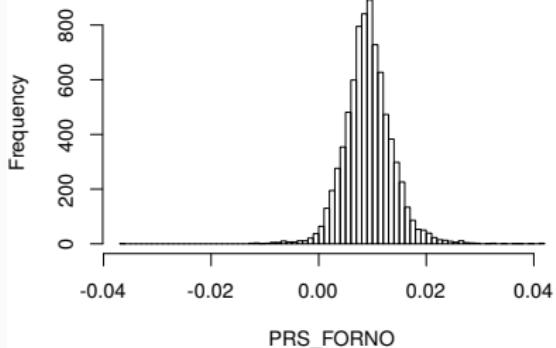
Configuração de títulos, *labels* e fontes

- `font`: 1=Normal, 2=Negrito, 3=Itálico, 4=Negrito e Itálico
- `font.axis`: Fonte das anotações dos eixos
- `xlab`, `ylab`: *Label* dos eixos
- `font.lab`: Fonte dos *labels* dos eixos
- `main`: Título
- `font.main`: Fonte do título
- `family`: Família da fonte
- ...

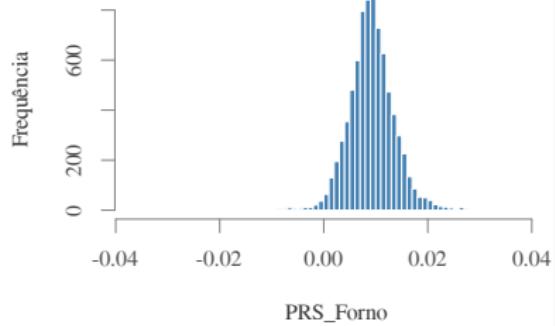
Customização de títulos, *labels* e fontes

```
> par(mfrow=c(1, 2))
> hist(PRS_FORNO, breaks = 100)
> par(bg = "grey95", fg = "grey50",
+      col.axis = "grey30", col.lab = "grey20")
> par(family = "Norasi", cex = 1.1)
> hist(PRS_FORNO, breaks = 100, col = "#4682b4", border = "white",
+       main = "PRS_Forno",
+       xlab = "PRS_Forno", ylab = "Frequênci")
```

Histogram of PRS_FORNO



PRS_Forno



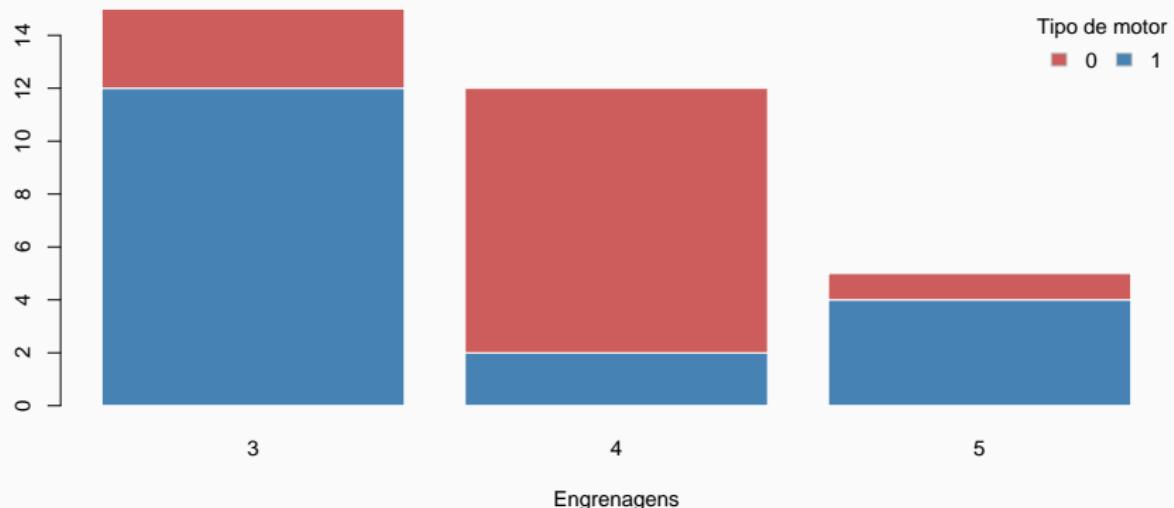
Customização da legenda

O R conta com alguns parâmetros para configuração de legenda na função `legend`

- `x`, `y`: Posição da legenda
- `legend`: Vetor com os textos na legenda
- `col`: Cor das linhas ou pontos na legenda
- `box.col`: Cor da caixa da legenda
- `border`: Cor da borda dos elementos da legenda
- `cex`: Tamanho da fonte
- `horiz`: Legenda na horizontal ou vertical

Customização da legenda

```
> cont <- table(mtcars$vs, mtcars$gear)
> barplot(cont, col=c("steelblue","Indianred"), legend = rownames(cont),
+           border = "white", xlab = "Engrenagens",
+           args.legend = list(x='topright', title='Tipo de motor',
+                               legend = rownames(cont),
+                               border = "grey", box.col = "transparent",
+                               horiz = T))
```



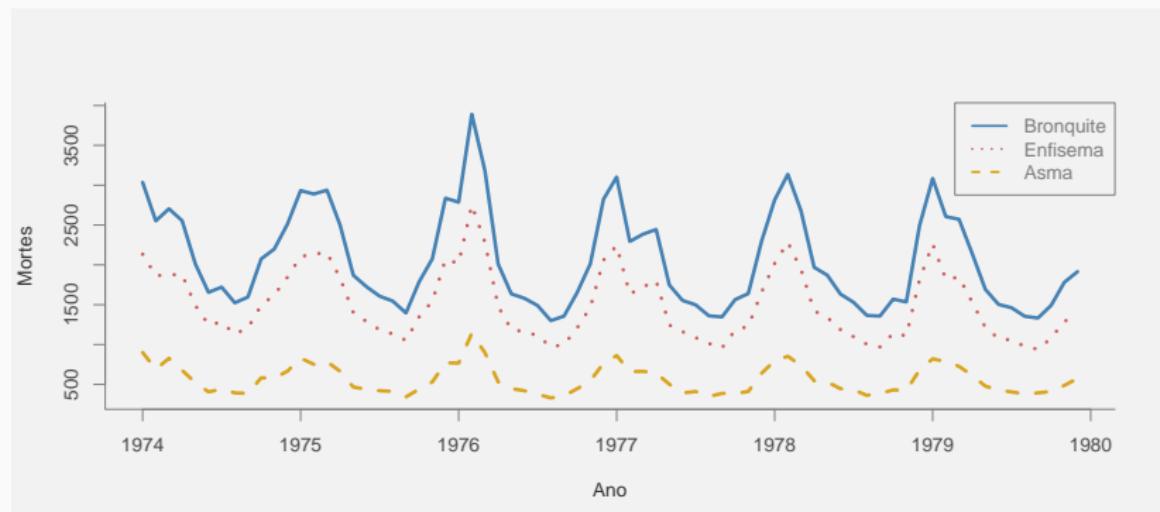
Customização de formas

Customização de formas

- `lty`: Tipo de linha (1 a 6)
- `lwd`: Tamanho da linha (*default = 1*)
- `pch`: Símbolo em gráficos de pontos (1 a 25)
- Veja mais em:
http://www.cookbook-r.com/Graphs/Shapes_and_line_types/

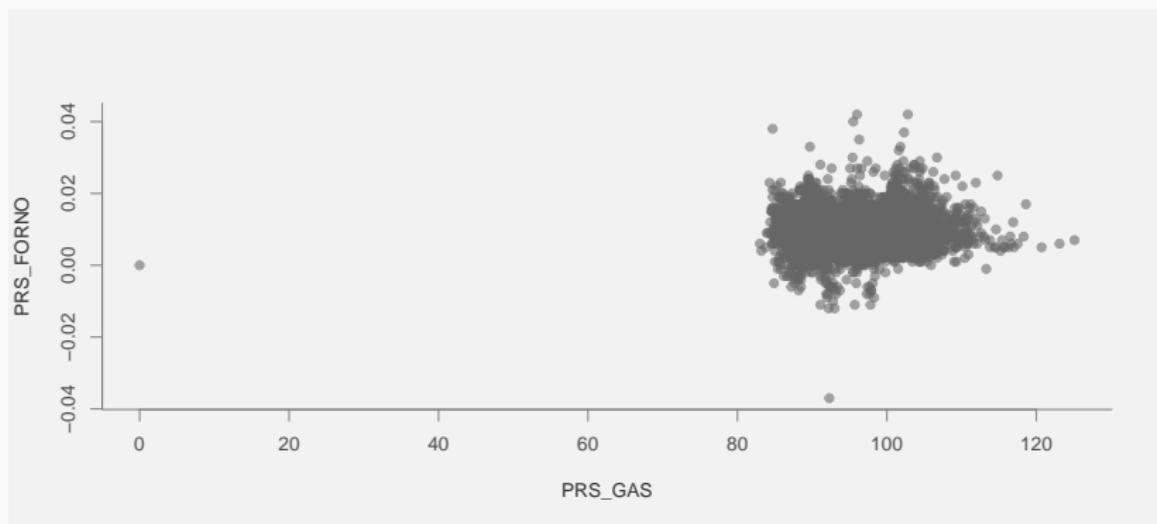
Customização de formas

```
> par(bg = "grey95", fg = "grey50", bty = "l",
+     col.axis = "grey30", col.lab = "grey20")
> ts.plot(ldeaths, mdeaths, fdeaths,
+         gpars=list(xlab="Ano", ylab="Mortes", lwd = 3,
+                     lty = c('solid', 'dotted', 'dashed'),
+                     col = c("steelblue", "indianred", "goldenrod")))
> legend(x = "topright", legend = c("Bronquite", "Enfisema", "Asma"),
+         col = c("steelblue", "indianred", "goldenrod"),
+         lty = c('solid', 'dotted', 'dashed'),
+         lwd = c(2,2,2))
```



Customização de formas

```
> par(bg = "grey95", fg = "grey50", bty = "1",
+     col.axis = "grey30", col.lab = "grey20")
> plot(x = PRS_GAS, y = PRS_FORNO, pch = 20, cex = 1.5,
+       col = paste0(grey(.4), 95))
```



Agora é a sua vez

Customizando gráficos

Com base nos exemplos e nas referências:

- Defina alguns parâmetros gráficos
- Crie um gráfico de dispersão
- Modifique a fonte e a cor do título
- Modifique a forma do elemento
- Crie uma variável de grupo (ou utilize alguma da base)
- Crie uma legenda no canto inferior direito

Composição de gráficos

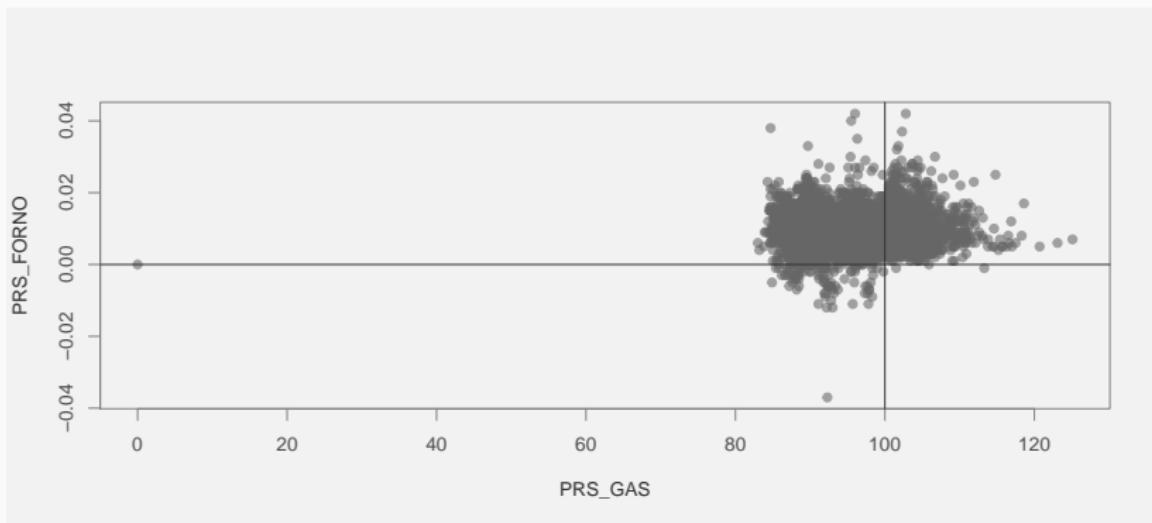
Muitas vezes queremos colocar mais de uma informação em um gráfico.

Exemplos

- Histograma e uma densidade
- Polígonos em um gráfico
- Linhas e pontos
- Marcar quadrantes em um gráfico de dispersão
- Pontos em um mapa
- ...

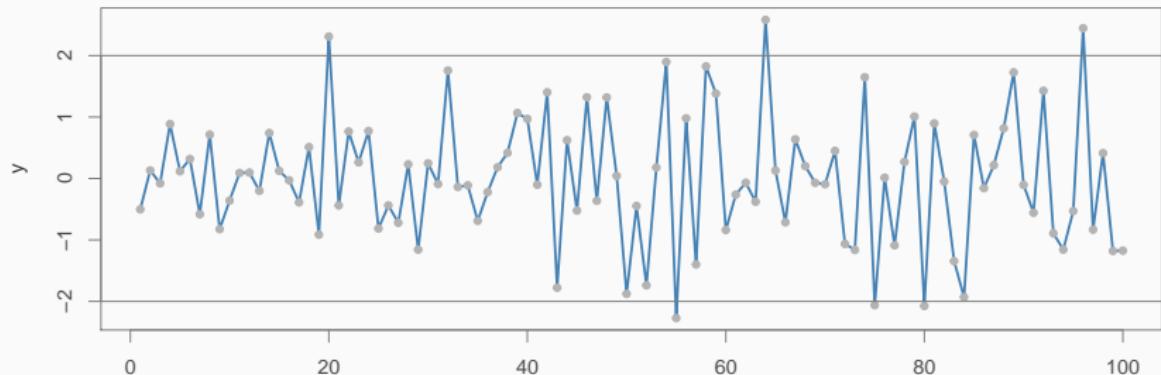
Composição de gráficos

```
> par(bg = "grey95", fg = "grey50",
+     col.axis = "grey30", col.lab = "grey20")
> plot(x = PRS_GAS, y = PRS_FORNO, pch = 20, cex = 1.5,
+       col = paste0(grey(.4), 95))
> abline(h = 0, v = 100, col = grey(.2))
```



Composição de gráficos

```
> par(fg = "grey50", col.axis = "grey30", col.lab = "grey20")
> set.seed(100)
> y <- rnorm(100)
> plot(x = 1:100, y = y, type = "l",
+       lwd = 2, col = "Steelblue", xlab="")
> points(x = 1:100, y = y, pch = 20, col = grey(.7), cex = 1.2)
> abline(h = c(-2, 2))
```

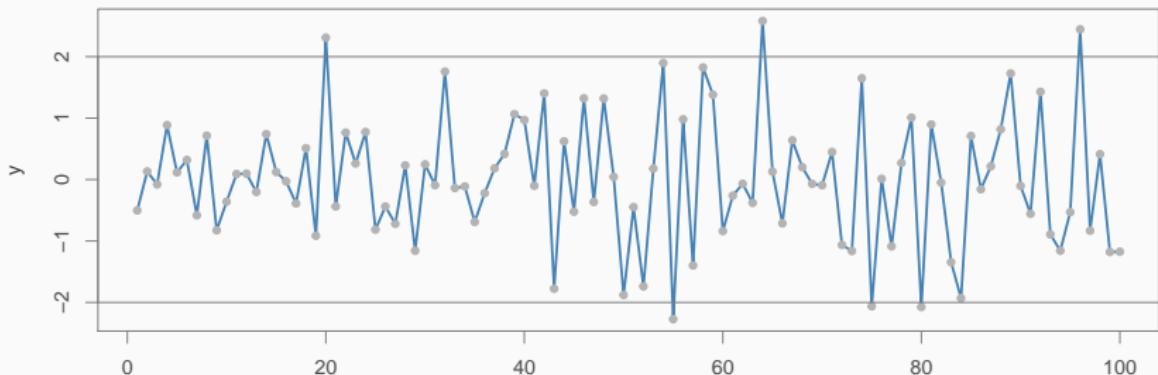


Identificando pontos

Identificando pontos

Só funciona dentro do R!

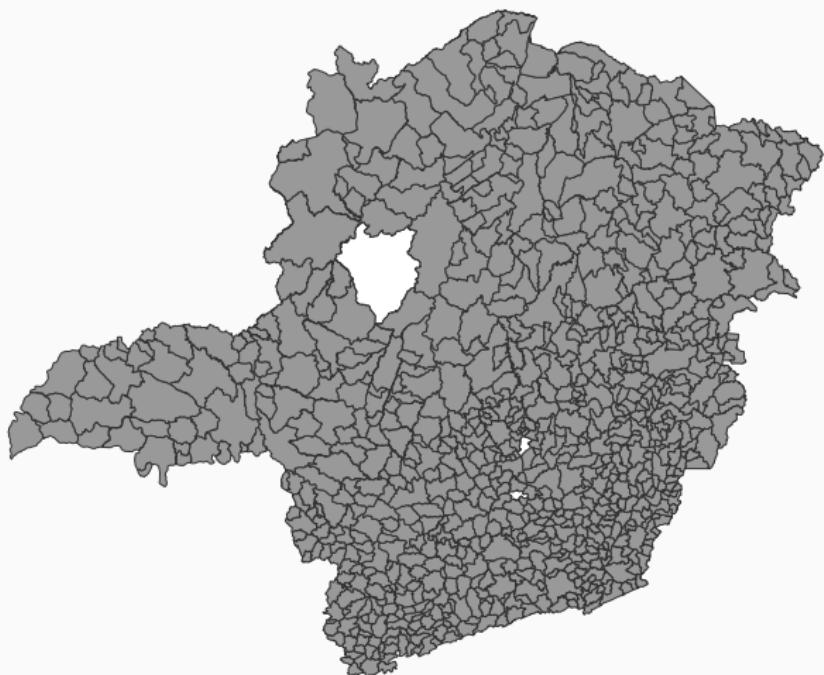
```
> par(fg = "grey50", col.axis = "grey30", col.lab = "grey20")
> plot(x = 1:100, y = y, type = "l",
+       lwd = 2, col = "Steelblue", xlab="")
> points(x = 1:100, y = y, pch = 20, col = grey(.7), cex = 1.2)
> abline(h = c(-2, 2))
```



```
> #identify(x = 1:100, y = y, labels = paste0("ID: ",1:100))
```

Mapas simples

Mapas simples no R



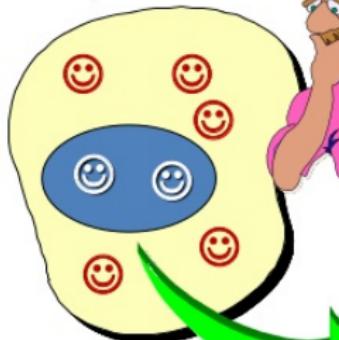
Módulo IX: Testes de hipóteses

Motivação



HYPOTHESIS TESTING

Population



I believe the population mean age is 50 (hypothesis).

Reject hypothesis! Not close.

Random sample

Mean $\bar{X} = 20$ ☺



Temos um pedido de **50.000 tubos** com um diametro de $\mu = 10mm$.

Coletamos uma amostra de tamanho **1000** e observamos **11** tubos com mais de **12mm** de diametro (o que é inaceitável para o cliente). O cliente deixa claro que aceita até **1%** de tubos fora da especificação. A proporção encontrada na amostra tem uma probabilidade alta de ser devolvida por excesso de itens não conformes?



Tipos de testes

Testes

Paramétricos



Não paramétricos



Testes não paramétricos fazem menos suposições a respeito dos dados. Porém, testes paramétricos são mais poderosos quando as suposições dos testes são atendidas. Assim sendo testes/modelos paramétricos são amplamente mais utilizados.

Nível de significância e p-valor

Nível de significância e p-valor

Um teste de hipótese possui uma **hipótese nula** (aquele a ser testada) e a **hipótese alternativa** (o oposto da hipótese nula).

p-valor

Dado que a hipótese nula é verdadeira, o p-valor representa a probabilidade de obtermos um valor igual ou mais extremo do que o observado.

Nível de significância

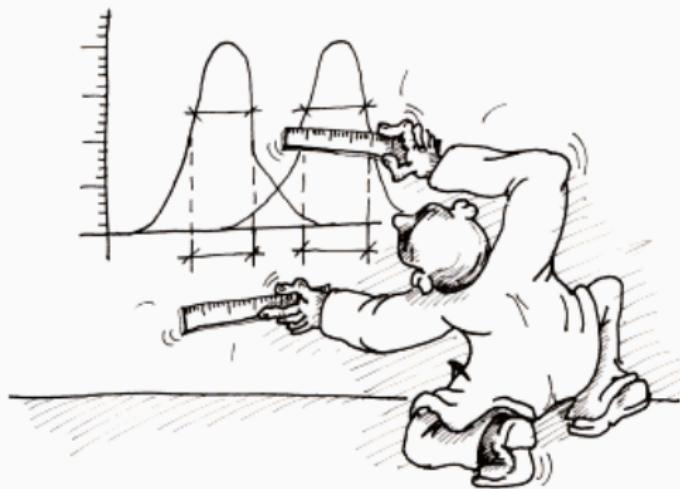
O nível de significância limita a probabilidade de obtermos um valor igual ou mais extremo do que o observado. Ou seja, limita a probabilidade de estarmos rejeitando a hipótese nula quando esta é verdadeira.



Testes paramétricos

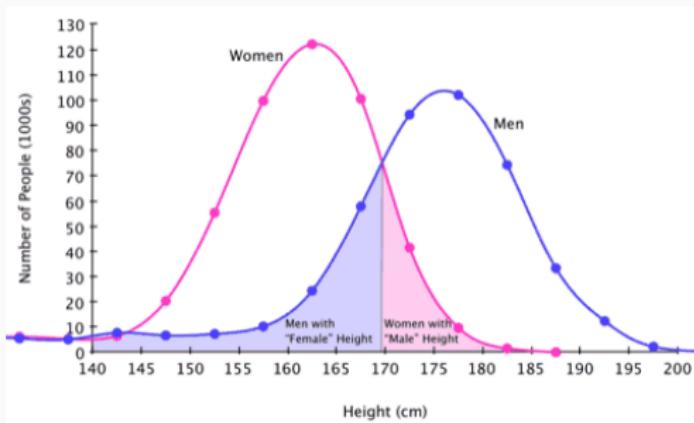
Testes paramétricos:

- **Teste t:** Utilizado para testar diferenças entre médias
- **Teste F:** Utilizado para testar diferenças entre variâncias



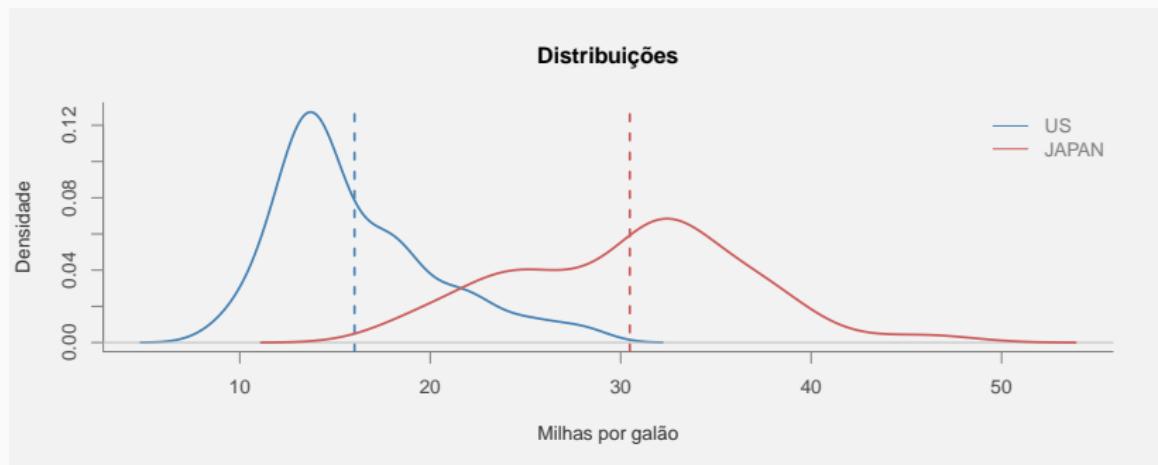
Teste t

- Utilizado para testar a diferença entre uma média e um valor
- **Hipótese nula:** A média é **igual** ao valor
- Utilizado para testar diferença entre médias de grupos (pareados e independentes)
- **Hipótese nula:** As médias dos grupos são **iguais**
- **Pressupõe normalidade dos dados** (veremos como testar normalidade na seção testes não paramétricos)



Teste t

Exemplo: Milhas por galão de carros Americanos e Japoneses



Teste t

```
> t.test(x = cars$US, y = cars$JAPAN,
+           conf.level = 0.95, paired = F)

Welch Two Sample t-test

data: cars$US and cars$JAPAN
t = -17.338, df = 138.23, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-16.10429 -12.80710
sample estimates:
mean of x mean of y
16.02532 30.48101
```

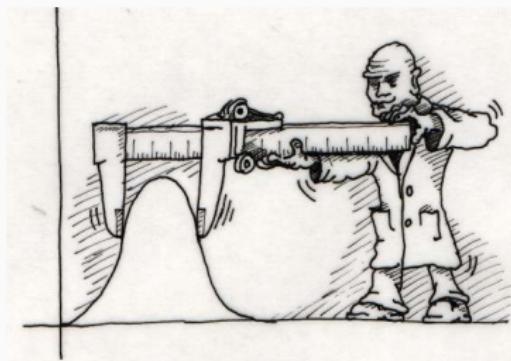
Teste t

```
> media_teste <- 17
>
> t.test(x = cars$US, mu = media_teste,
+         conf.level = 0.95)

One Sample t-test

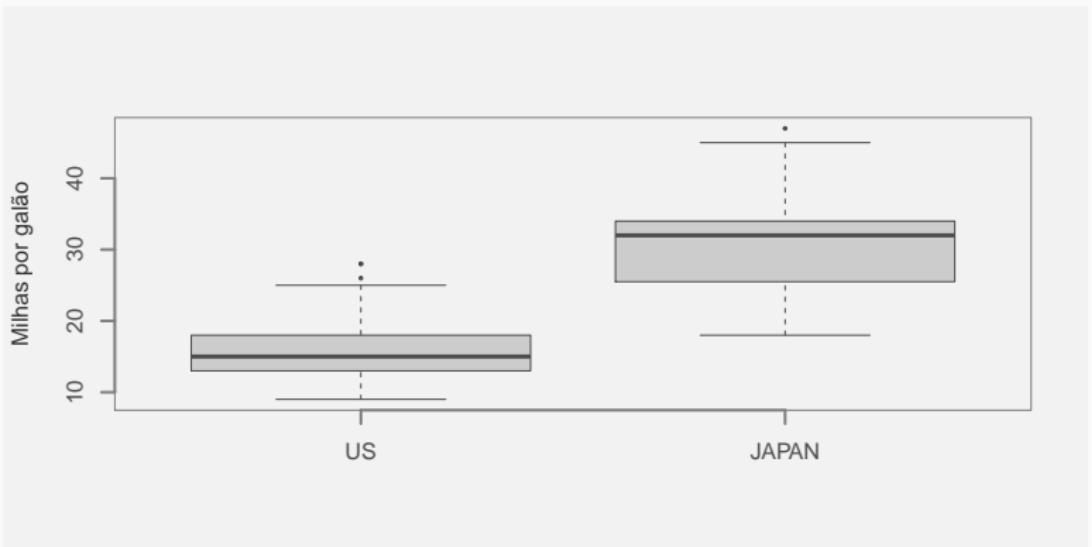
data: cars$US
t = -2.0641, df = 78, p-value = 0.04233
alternative hypothesis: true mean is not equal to 17
95 percent confidence interval:
15.08524 16.96539
sample estimates:
mean of x
16.02532
```

- Utilizado para testar diferença entre a variância e um valor
- **Hipótese nula:** A variância é **igual** ao valor
- Utilizado para testar diferença entre variâncias de grupos
- **Hipótese nula:** As variâncias dos grupos são **iguais**
- **Pressupõe normalidade dos dados**



Teste F

```
> par(bg = "grey95", fg = "grey50", cex = 0.5, lwd = 0.5,  
+     col.axis = "grey30", col.lab = "grey20")  
>  
> boxplot(cars, ylab = "Milhas por galão",  
+           col = grey(0.8), border = grey(0.3), pch = 20)
```



Teste F

```
> var.test(x = cars$US, y = cars$JAPAN)

F test to compare two variances

data: cars$US and cars$JAPAN
F = 0.47219, num df = 78, denom df = 78, p-value = 0.001093
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
0.3019564 0.7384046
sample estimates:
ratio of variances
0.4721927
```

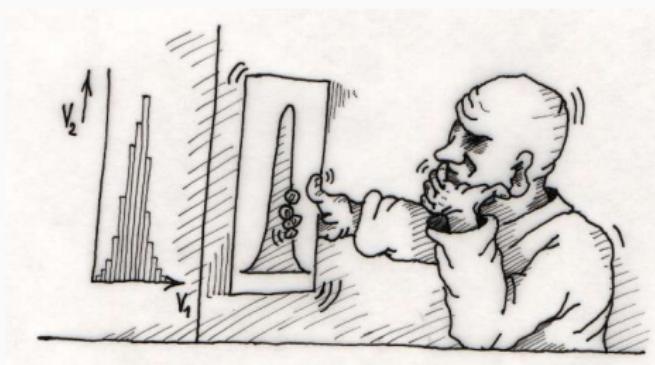
Testes não paramétricos

Testes não paramétricos:

- **Kolmogorov-Smirnov**: Normalidade
- **Shapiro-Wilks**: Normalidade
- **Anderson-Darling**: Normalidade
- **Levene**: Igualdade de variância
- **Bartlett**: Igualdade de variância
- **Wilcoxon**: Igualdade de médias (amostras pareadas)
- **Kruskal-Wallis**: Igualdade de médias (amostras independentes)
- **Qui-quadrado**: Independência

Testes de normalidade

- Utilizados para testar a distribuição dos dados
- Na prática estamos interessados em descobrir se podemos assumir normalidade
- **Hipótese nula:** Os dados **seguem** uma distribuição normal



Testes de normalidade

Kolmogorov-Smirnov:

```
> ks.test(x = cars$JAPAN, y = 'pnorm')
Warning in ks.test(x = cars$JAPAN, y = "pnorm"): ties should not be present for the
Kolmogorov-Smirnov test

One-sample Kolmogorov-Smirnov test

data: cars$JAPAN
D = 1, p-value < 2.2e-16
alternative hypothesis: two-sided
```

Testes de normalidade

Shapiro-Wilk:

```
> shapiro.test(x = cars$JAPAN)

Shapiro-Wilk normality test

data: cars$JAPAN
W = 0.97676, p-value = 0.1586
```

Anderson-Darling:

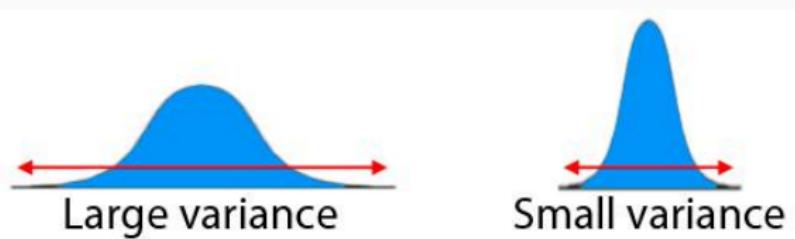
```
> library(nortest)
> ad.test(x = cars$JAPAN)

Anderson-Darling normality test

data: cars$JAPAN
A = 0.73389, p-value = 0.05355
```

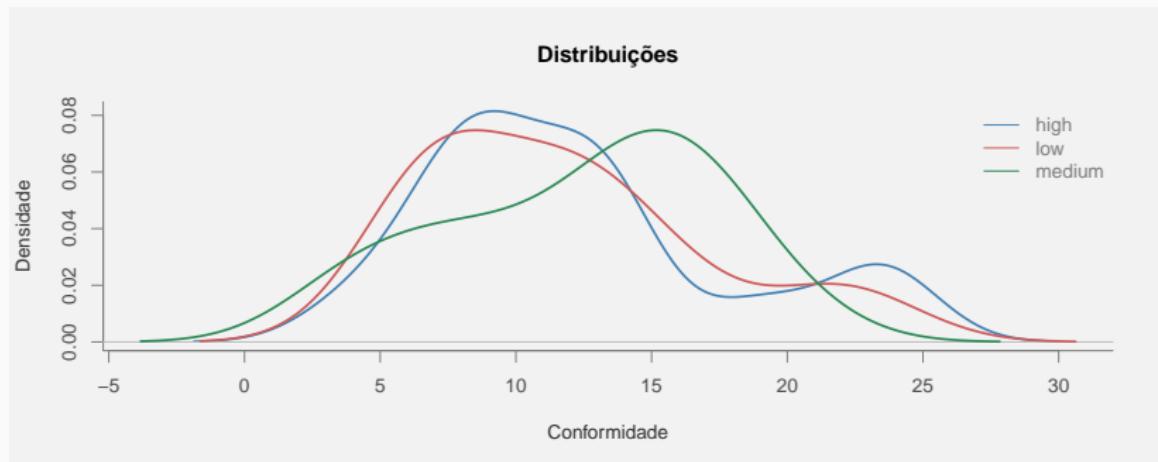
Testes de igualdade de variâncias

- Utilizados para testar a igualdade de variâncias de dois ou mais grupos
- **Hipótese nula:** Os grupos possuem variâncias **iguais**



Testes de igualdade de variâncias

Exemplo: Experimento psicológico realizado em duplas de diferentes classes sociais. Neste experimento confrontavam-se opiniões e observava-se a concordância com a opinião alheia.



Testes de igualdade de variâncias

```
> bartlett.test(conformity ~ fcatecategory, data=Moore)

Bartlett test of homogeneity of variances

data: conformity by fcatecategory
Bartlett's K-squared = 0.33858, df = 2, p-value = 0.8443
```

```
> leveneTest(conformity ~ fcatecategory, data=Moore)
Levene's Test for Homogeneity of Variance (center = median)
  Df F value Pr(>F)
group  2   0.046 0.9551
      42
```

- Utilizado para testar a igualdade de médias em grupos **pareados**
- **Hipótese nula:** As médias dos grupos são **iguais**

Serve para testar, por exemplo, se a introdução de um medicamento no tratamento traz ou não alguma mudança significativa nas horas de sono de uma pessoa que sofre de insônia.

Testes de igualdade de médias - Wilcoxon

Exemplo: Escala de depressão de nove pacientes antes e depois do inicio de terapia.

```
> x <- c(1.83, 0.50, 1.62, 2.48, 1.68, 1.88, 1.55, 3.06, 1.30)
> y <- c(0.878, 0.647, 0.598, 2.05, 1.06, 1.29, 1.06, 3.14, 1.29)
> wilcox.test(x, y, paired = TRUE, alternative = "greater")

Wilcoxon signed rank test

data: x and y
V = 40, p-value = 0.01953
alternative hypothesis: true location shift is greater than 0
```

A diferença entre a escala inicial e a final é significativamente maior que 0.

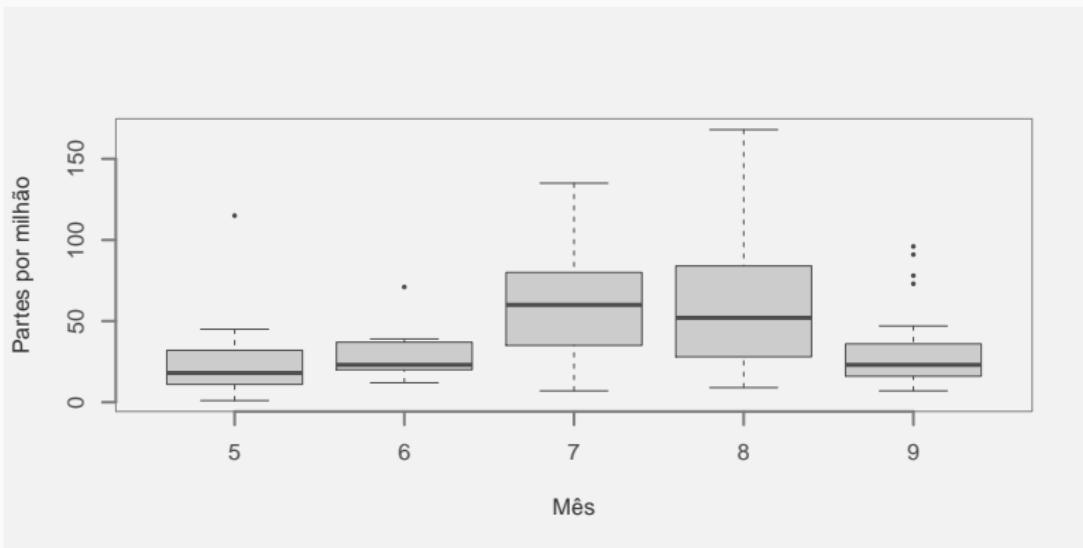
Testes de igualdade de médias - Kruskal-Wallis

- Utilizado para testar a igualdade de médias em grupos **independentes**
- **Hipótese nula:** As médias dos grupos são **iguais**

Serve para testar, por exemplo, se as médias dos diâmetros dos tubos em diferentes lotes são iguais.

Testes de igualdade de médias - Kruskal-Wallis

Exemplo: Avaliação da qualidade do ar (ozônio) em NY.



Testes de igualdade de médias - Kruskal-Wallis

```
> kruskal.test(Ozone ~ Month, data = airquality)

Kruskal-Wallis rank sum test

data: Ozone by Month
Kruskal-Wallis chi-squared = 29.267, df = 4, p-value = 6.901e-06
```

- Utilizado para testar associação entre duas variáveis **categóricas**
- **Hipótese nula:** As variáveis são independentes

Serve para testar, por exemplo, se tubos de boa qualidade estão associados à alguma faixa de temperatura específica.

Teste de associação

Exemplo Testando se os sabores de sorvete preferidos de homens e mulheres são diferentes.

```
> men = c(100, 120, 60)
> women = c(350, 200, 90)
>
> ice.cream.survey = as.data.frame(rbind(men, women))
> names(ice.cream.survey) = c('chocolate', 'vanilla', 'strawberry')
>
> ice.cream.survey
      chocolate vanilla strawberry
men          100     120        60
women        350     200        90
```

Teste de associação

```
> chisq.test(ice.cream.survey)

Pearson's Chi-squared test

data: ice.cream.survey
X-squared = 28.362, df = 2, p-value = 6.938e-07
```

Agora é a sua vez

Testes estatísticos

Escolha uma variável categórica e uma variável contínua de interesse e com base nela:

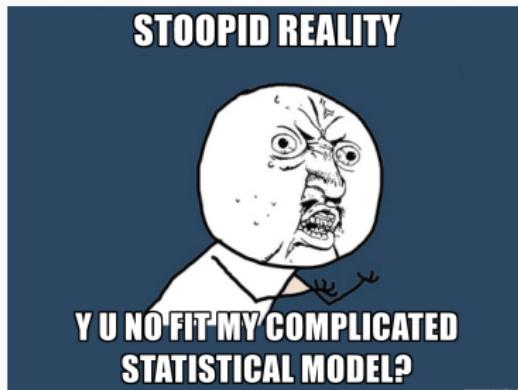
- Faça gráficos caracterizando a distribuição
- A variável contínua pode ser assumida normal?
- Faça um teste de hipótese cujo interesse é testar se a média é igual a mediana (calcule a mediana com base nos dados)
- Divilde as variáveis em três grupos (com base nos índices).
- A variável categórica pode ser considerada independente do grupo?
- Faça um teste de médias (paramétrico e um não-paramétrico) para testar se as médias são iguais dentro de cada grupo.

Módulo X: Modelos estatísticos

Introdução

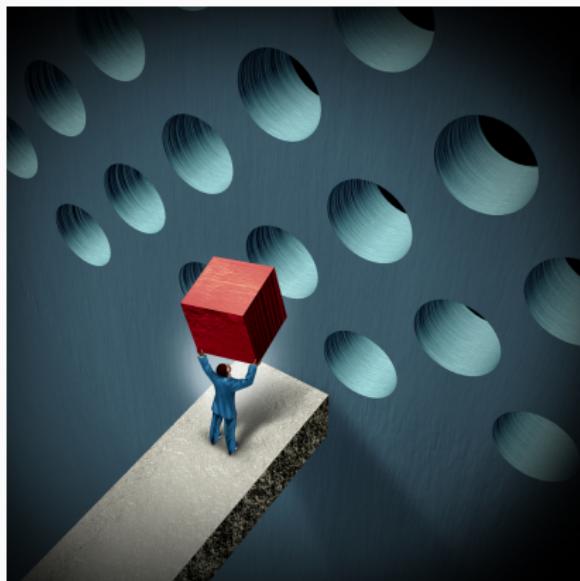
Modelo estatístico

- Visa explicar uma variável de interesse com base em características da mesma
- Queremos entender a influência das características na variável resposta
- Com isso visamos identificar o padrão gerador dos dados observados para podermos pensar em previsões
- Para tanto algumas suposições devem ser respeitadas (depende de cada modelo)



Suposições

- Cada modelo tem as suas suposições que devem ser verificadas
- O não cumprimento de alguma suposição pode invalidar a análise (e as conclusões)

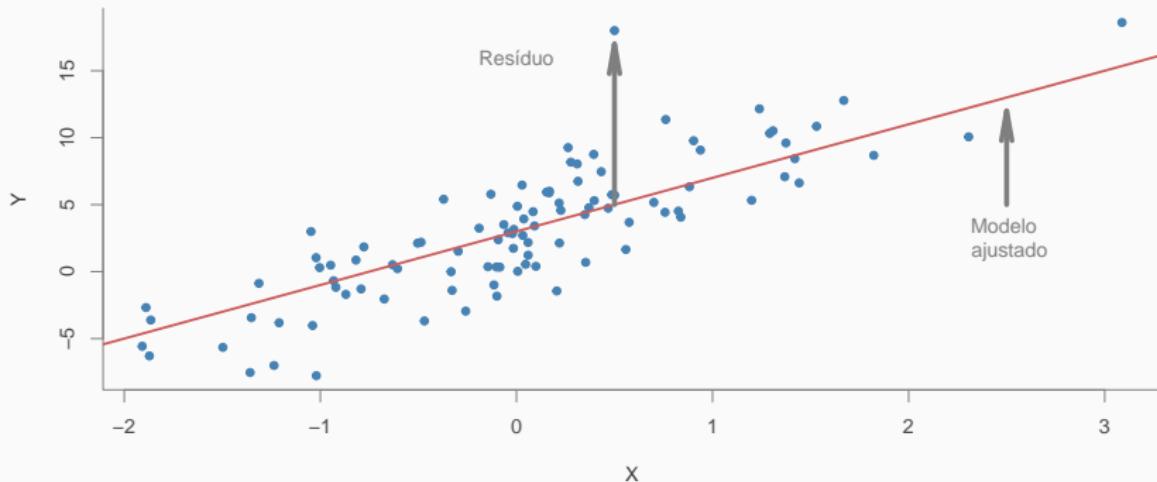


"Quando a única ferramenta que se tem é um **martelo**, então todo problema fica parecido com um **prego**." - Abraham Harold Maslow (1908 - 1970)



De forma geral os modelos que vamos trabalhar são da seguinte forma:

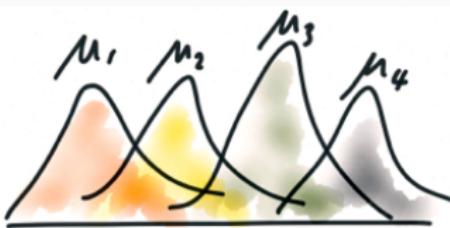
$$Y_i = \beta_0 + \beta_1 X_{1i} + \dots + \beta_p X_{pi} + \epsilon_i$$



Análise de variância

Características

- Modelo paramétrico (parecido com o Kruskal-Wallis)
- Visa comparar a média de três ou mais grupos
- Variável resposta é contínua
- Podem ser inseridos fatores explicativos (covariáveis de grupos)
- Nos traz o efeito de cada fator na variável resposta



ANOVA
 $\mu_1 = \mu_2 = \mu_3 = \mu_4 ?$

Suposições

- Normalidade dos resíduos
- Erros são homocedásticos (tem a mesma variância)
- Após realizar a análise é necessário investigar se as suposições foram satisfeitas



Comandos básicos

- **aov**: Ajusta vários tipos de modelos de ANOVA
- **TukeyHSD**: Teste de comparação múltipla

Exemplo: Rendimento de ervilhas dada a aplicação de Nitrogênio, Fosfato e Potássio em 6 blocos.

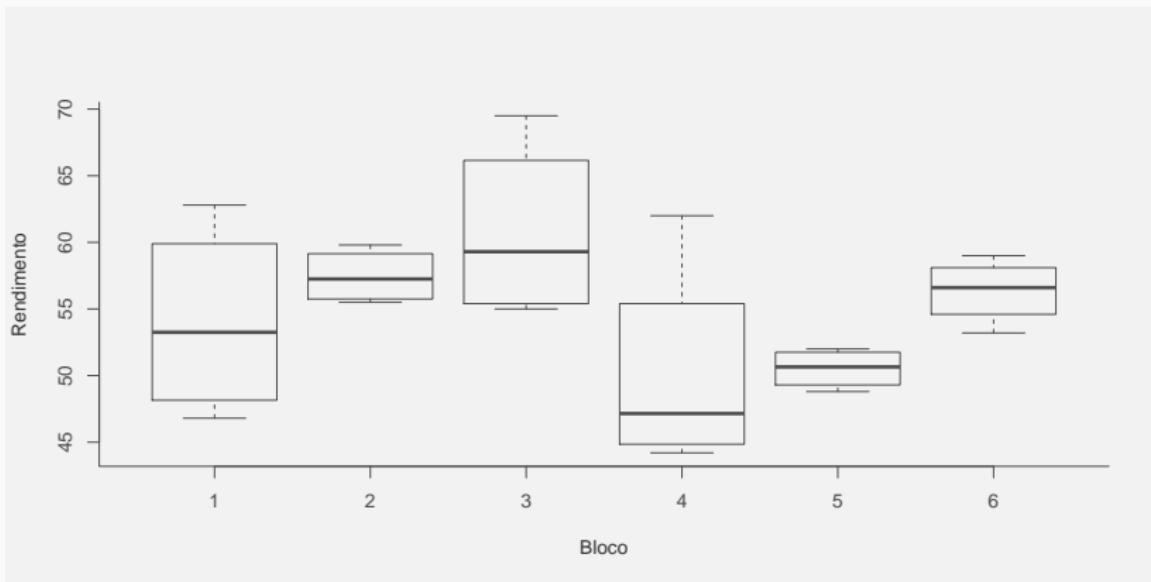
```
> head(npk)
  block N P K yield
1     1 0 1 1  49.5
2     1 1 1 0  62.8
3     1 0 0 0  46.8
4     1 1 0 1  57.0
5     2 1 0 0  59.8
6     2 1 1 1  58.5
```

```
> modelo_anova <- aov(formula = yield ~ block + N*P*K, data = npk)
```

Análise de variância

Visualizando os dados por **bloco**:

```
> par(bg = "grey95", fg = "grey30", bty = "l",
+     col.axis = "grey30", col.lab = "grey20")
> boxplot(yield ~ block, data = npk, xlab = "Bloco", ylab = "Rendimento")
```



Análise de variância

```
> summary(modelo_anova)
    Df Sum Sq Mean Sq F value Pr(>F)
block      5 343.3  68.66   4.447 0.01594 *
N          1 189.3 189.28  12.259 0.00437 **
P          1    8.4    8.40   0.544 0.47490
K          1   95.2   95.20   6.166 0.02880 *
N:P        1   21.3   21.28   1.378 0.26317
N:K        1   33.1   33.14   2.146 0.16865
P:K        1    0.5    0.48   0.031 0.86275
Residuals 12 185.3   15.44
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Análise de variância

```
> comp_medias <- TukeyHSD(modelo_anova, conf.level = 0.95,
+                               which = c("block", "N", "K"))
> comp_medias$block
    diff      lwr      upr      p adj
2-1  3.425 -5.907892 12.7578919 0.81341001
3-1  6.750 -2.582892 16.0828919 0.22035618
4-1 -3.900 -13.232892  5.4328919 0.72474514
5-1 -3.500 -12.832892  5.8328919 0.80023617
6-1  2.325 -7.007892 11.6578919 0.95448819
3-2  3.325 -6.007892 12.6578919 0.83038945
4-2 -7.325 -16.657892  2.0078919 0.16138970
5-2 -6.925 -16.257892  2.4078919 0.20076270
6-2 -1.100 -10.432892  8.2328919 0.99840052
4-3 -10.650 -19.982892 -1.3171081 0.02248374
5-3 -10.250 -19.582892 -0.9171081 0.02867873
6-3 -4.425 -13.757892  4.9078919 0.61771600
5-4  0.400 -8.932892  9.7328919 0.99998870
6-4  6.225 -3.107892 15.5578919 0.28840930
6-5  5.825 -3.507892 15.1578919 0.34984275
```

Nossos resultados são válidos?

A análise de resíduos nos responde...

Testes de normalidade

```
> library(nortest)
> residuos_anova <- modelo_anova$residuals
> shapiro.test(x = residuos_anova)
```

Shapiro-Wilk normality test

```
data: residuos_anova
W = 0.96875, p-value = 0.6363
```

```
> ad.test(x = residuos_anova)
```

Anderson-Darling normality test

```
data: residuos_anova
A = 0.18956, p-value = 0.8897
```

Testes de homocedasticidade

```
> library(car)
> leveneTest(yield ~ N*P*K, data = npk)
Levene's Test for Homogeneity of Variance (center = median)
      Df F value Pr(>F)
group    7  0.3301 0.9287
          16
```

```
> bartlett.test(yield ~ interaction(N, P, K), data = npk)

Bartlett test of homogeneity of variances

data: yield by interaction(N, P, K)
Bartlett's K-squared = 3.2535, df = 7, p-value = 0.8606
```

Agora é a sua vez

Análise de variância

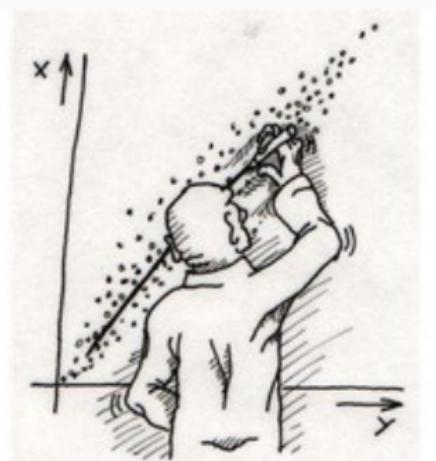
Utilizando a base `?InsectSprays`, faça:

- Visualização de dados adequada
- Análise de variância e se necessário a comparação de médias
- Análise de resíduos
- O que podemos fazer para contornar o problema?

Análise de regressão linear

Características

- Modelo paramétrico
- Visa explicar uma variável resposta através de covariáveis
- Variável resposta é contínua
- Nos traz o efeito de cada covariável na variável resposta



Suposições

- Normalidade dos resíduos
- Erros são homocedásticos (tem a mesma variância)
- Não há multicolinearidade
- Não há autocorrelação nos erros



Comandos básicos

- `lm`: Ajusta vários tipos de modelos lineares
- `confint`: Retorna os intervalos de confiança dos coeficientes
- `predict`: Faz previsões com base em um modelo

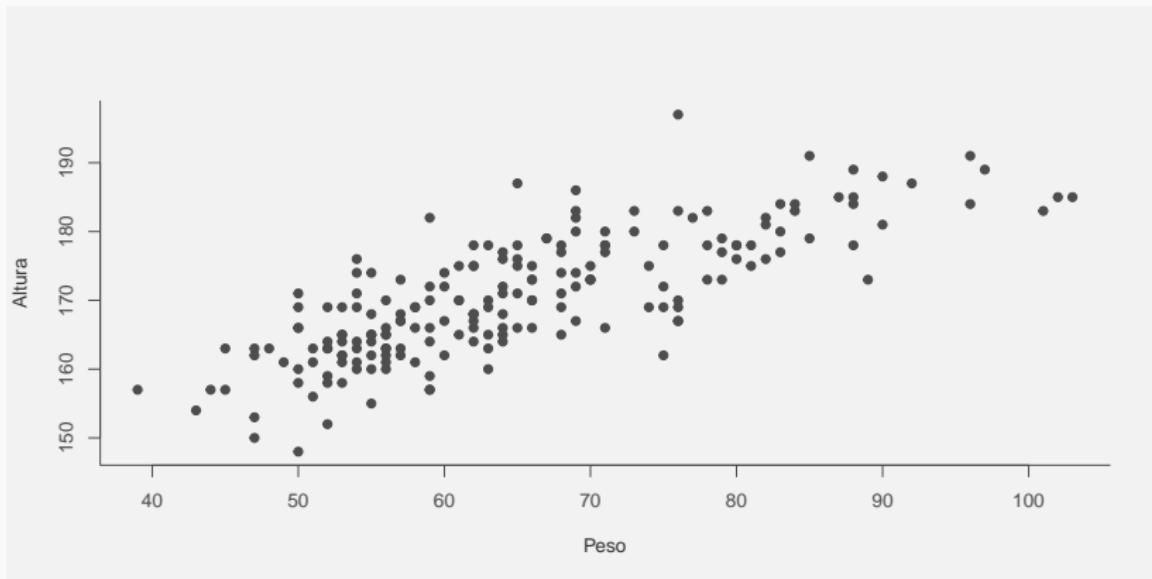
Exemplo: Peso de adultos dado suas respectivas alturas.

```
> head(Davis)
   sex weight height repwt reptht
1   M     77    182     77    180
2   F     58    161     51    159
3   F     53    161     54    158
4   M     68    177     70    175
5   F     59    157     59    155
6   M     76    170     76    165
```

```
> Davis <- subset(x = Davis, weight <= 110) # Tirando dois outliers
> modelo_reg <- lm(formula = weight ~ height, data = Davis)
```

Análise de regressão linear

```
> par(bg = "grey95", fg = "grey30", bty = "l",
+     col.axis = "grey30", col.lab = "grey20")
> plot(x = Davis$weight, y = Davis$height,
+       xlab = "Peso", ylab = "Altura", pch = 20, cex = 1.5)
```



Análise de regressão linear

A diferença de 1 kg no peso entre dois adultos implica em média em uma diferença de 1.12 centímetros de altura.

```
> summary(modelo_reg)

Call:
lm(formula = weight ~ height, data = Davis)

Residuals:
    Min      1Q  Median      3Q     Max 
-18.9000 -5.2946 -0.5476  5.0698 21.9765 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -126.57554   10.84059 -11.68   <2e-16 ***
height       1.12349    0.06348  17.70   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.971 on 196 degrees of freedom
Multiple R-squared:  0.6151,    Adjusted R-squared:  0.6131 
F-statistic: 313.2 on 1 and 196 DF,  p-value: < 2.2e-16
```

Análise de regressão linear

```
> confint(modelo_reg)
              2.5 %      97.5 %
(Intercept) -147.954718 -105.196359
height        0.998302   1.248682
```

```
> predict(object = modelo_reg, newdata = data.frame(height = c(175, 180, 185, 190)))
     1      2      3      4
70.03557 75.65303 81.27049 86.88795
```

Testes de normalidade

```
> residuos_reg <- modelo_reg$residuals  
> shapiro.test(x = residuos_reg)
```

Shapiro-Wilk normality test

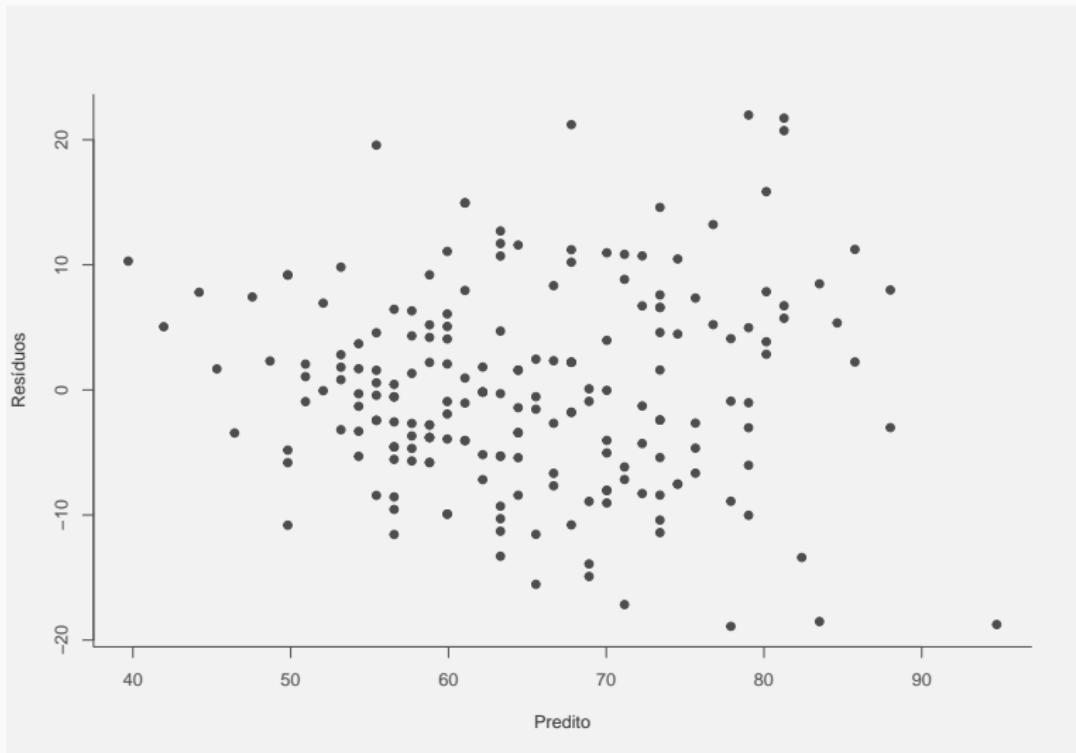
```
data: residuos_reg  
W = 0.99064, p-value = 0.2272
```

```
> ad.test(x = residuos_reg)
```

Anderson-Darling normality test

```
data: residuos_reg  
A = 0.39956, p-value = 0.3604
```

Homocedasticidade



Agora é a sua vez

Regressão Linear

Utilizando a base `?airquality` e a variável resposta `Temp`, faça:

- Visualização de dados adequada
- Análise de regressão linear
- Análise de resíduos
- Intervalos de confiança
- Cria uma base para previsões

Outros modelos interessantes

Outros modelos

- Modelos lineares generalizados
 - Regressão Poisson
 - Regressão Binária
 - Análise de sobrevivência/confiabilidade
 - Agrupamentos hierárquicos
 - ...



Regressão logística

Utilizado para modelar dados de binários.

Exemplo: Entender a razão de um item ser ou não ser defeituoso dado características de como o mesmo foi construído.



Regressão logística

Exemplo: Modelar o que foi determinante para a morte dos tripulantes do Titanic.

	1st	2nd	3rd	Crew
No	122	167	528	673
Yes	203	118	178	212

(Intercept)	Class2nd	Class3rd	ClassCrew	SexMale	AgeChild
-2.0438374	1.0180950	1.7777622	0.8576762	2.4200603	-1.0615424

Regressão Poisson

Utilizado para modelar dados de contagem.

Exemplo: Explicar o número de itens defeituosos por lote dado características especiais do lote de onde vieram.



Análise de sobrevivência

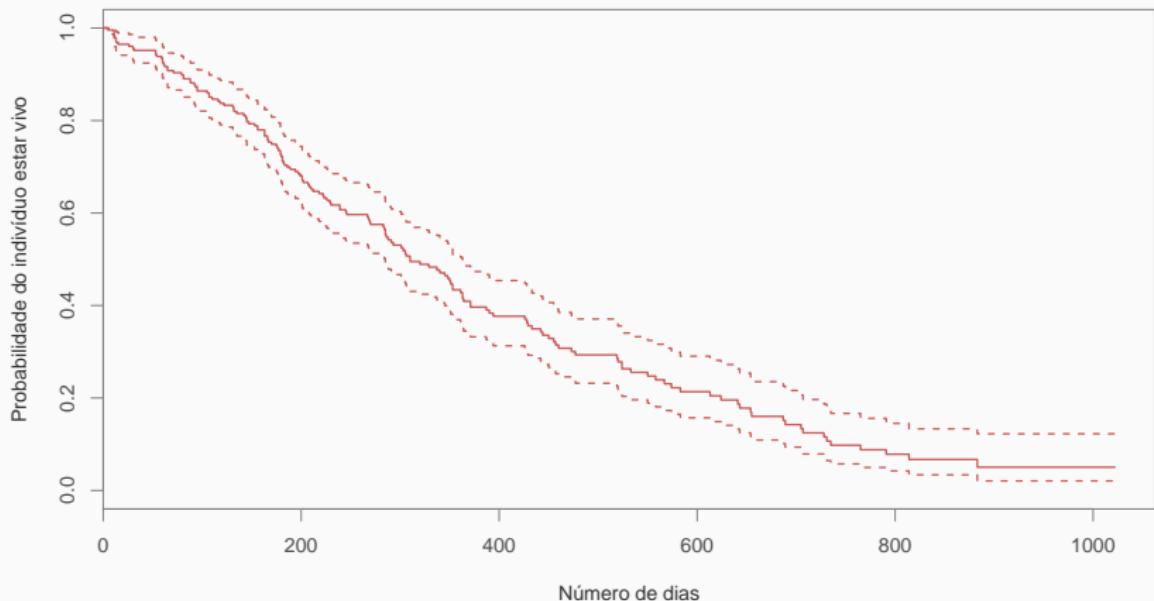
Utilizada para modelar o tempo até determinado evento.

Exemplo: Entender quais fatores aumentam ou diminuem o tempo até a falha de um determinado equipamento



Análise de sobrevivência

Exemplo: Sobrevivência em pacientes com câncer após entrar no *North Central Cancer Treatment Group*.



Agrupamentos hierárquicos

Utilizado para encontrar agrupar observações semelhantes.

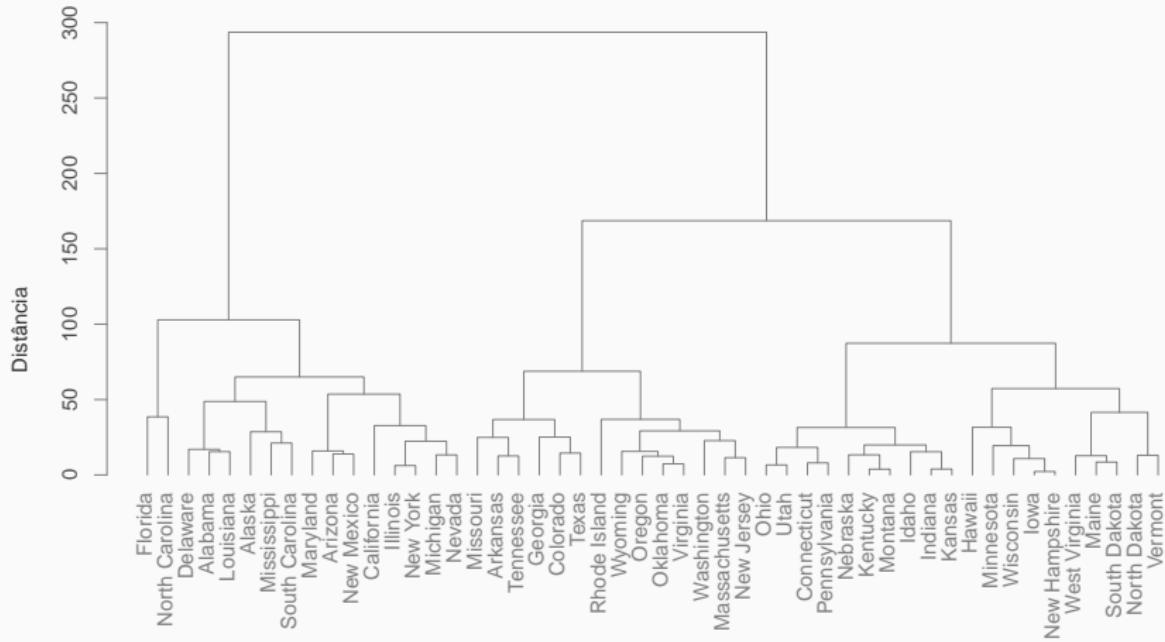
Exemplo: Dado um universo de 100 máquinas podemos agrupá-las em grupos dado o número de itens defeituosos fabricados.



Grouping

Agrupamentos hierárquicos

Exemplo: Cluster de condados americanos de acordo com dados de prisões e de população urbana.



Módulo XI: Tratamento e Manipulação de Dados

Tratamento de Dados

Criando novas variáveis

- Para criar uma nova variável em `data.frame()`, basta usar os operadores `$` e `<-`.
- Vamos usar a função `ifelse()` para criar uma nova variável.

```
> df <- data.frame(nomes = c('JoAo30', 'lara43', 'Manoel21', 'Pedro34', 'DeNIse25'),
+                     altura = c(145, 184, 191, 160, 158),
+                     peso   = c(45,   64,  85, 65, 58),
+                     sexo   = c('M', 'F', 'M', 'F', 'M'),
+                     stringsAsFactors = F)
> df$sexo2 <- ifelse(df$sexo == 'M', 'Masculino', 'Feminino')
> df
  nomes altura peso sexo    sexo2
1 JoAo30     145   45    M Masculino
2 lara43     184   64    F Feminino
3 Manoel21    191   85    M Masculino
4 Pedro34     160   65    F Feminino
5 DeNIse25    158   58    M Masculino
```

- Podemos sobrescrever a variável `sexo` diretamente.

```
> df$sexo  <- ifelse(df$sexo == 'M', 'Masculino', 'Feminino') #ou
> df$sexo2 <- NULL #deletando a varivavel sexo2 do data.frame
```

Agora é a sua vez

IMC - Índice de Massa Corporal

1. Crie uma nova variável dentro do data frame `df` que armazene o IMC de cada indivíduo.
2. Crie uma outra variável que indica se o indivíduo estiver acima do peso ou não: valores entre [25, 30).

$$IMC = \frac{Peso\ (kg)}{Altura\ (m)^2}$$

Categorizando variáveis

- Categorizar varáveis contínuas é muito comum em análise dados.
- Vamos criar uma nova variável categórica a partir do `imc`.

Categoria	Descrição
< 17	Muito abaixo do peso
17 - 18.49	Abaixo do peso
18.5 - 24.99	Peso normal
25 - 29.99	Acima do peso
30 - 34.99	Obesidade I
35 - 39.99	Obesidade II
≥ 40	Obesidade III

Table 2: Tabela de IMC.

Categorizando variáveis

- Para categorizar variáveis contínuas no R podemos usar a função `cut()`.

Função `cut()`

- `cut`: divide um vetor em intervalos e categorias rotuladas.
 - `breaks`: pontos para realizar a quebra do vetor
 - `labels`: rótulos para cada uma das quebras

```
> brks <- c(0, 17, 18.5, 25, 30, 35, 40, 50)
> rotulos <- c("Muito abaixo do peso", "Abaixo do peso", "Peso normal", "Acima do peso", "Muito acima do peso")
> df$imc <- df$peso/(df$altura/100)^2
> df$imc.cat <- cut(df$imc, breaks = brks, labels = rotulos, right = FALSE)
> df
  nomes altura peso      sexo      imc      imc.cat
1 JoAo30     145   45 Masculino 21.40309 Peso normal
2 lara43     184   64 Feminino 18.90359 Peso normal
3 Manoel21    191   85 Masculino 23.29980 Peso normal
4 Pedro34     160   65 Feminino 25.39062 Acima do peso
5 DeNIse25    158   58 Masculino 23.23346 Peso normal
```

```
> class(df$imc.cat)
[1] "factor"
```

- Variáveis categóricas no R devem ser tratadas pela classe `factor()`.

Classe: `factor()`

- **Fatores**: uma classe para representar variáveis que possuem um número limitado de valores (categorias).

```
> data <- factor(c("alpha", "beta", "gamma", "alpha", "beta"))
> (fdata <- factor(data))
[1] alpha beta gamma alpha beta
Levels: alpha beta gamma
```

- **levels**: são os valores que a variável pode assumir.
- Podemos recategorizar um `factor()` facilmente.

```
> (rdata <- factor(data, labels = c("I", "II", "III")))
[1] I   II  III I   II
Levels: I II III
```

```
> levels(data) <- list(A="alpha", B="beta", C="gamma")
> data
[1] A B C A B
Levels: A B C
```

Recodificando variáveis categóricas

- Independente se estamos trabalhando com fatores ou não, podemos recodificar facilmente variáveis no R.

```
> dados <- read.table(header = T, text = '  
+ individuo sexo controle cond1 cond2  
+     1   M      7.9  12.3  10.7  
+     2   F      6.3  10.6  11.1  
+     3   F      9.5  13.1  13.8  
+     4   M     11.5  13.4  12.9  
+ ')  
> valores_rec <- c("M" = 0, "F" = 1)  
> valores_rec["M"]  
M  
0
```

```
> dados$scode <- valores_rec[dados$sexo]  
> dados  
individuo sexo controle cond1 cond2 scode  
1       1   M      7.9  12.3  10.7      1  
2       2   F      6.3  10.6  11.1      0  
3       3   F      9.5  13.1  13.8      0  
4       4   M     11.5  13.4  12.9      1
```

Recodificando variáveis categóricas

- Podemos usar o pacote **plyr** e a função **mapvalues**:

```
> library(plyr)
> dados$scode <- mapvalues(x = dados$sexo, from = c("M", "F"), to = c(0, 1))
> dados
  individuo sexo controle cond1 cond2 scode
1           1     M      7.9  12.3  10.7      0
2           2     F      6.3  10.6  11.1      1
3           3     F      9.5  13.1  13.8      1
4           4     M     11.5  13.4  12.9      0
```

- Ou a função **revalue**.

```
> dados$scode <- revalue(x = dados$sexo, replace = c("M"="1", "F"="2"))
> dados
  individuo sexo controle cond1 cond2 scode
1           1     M      7.9  12.3  10.7      1
2           2     F      6.3  10.6  11.1      2
3           3     F      9.5  13.1  13.8      2
4           4     M     11.5  13.4  12.9      1
```

Agora é a sua vez

Recodificando variáveis categóricas

- Carregue os dados `faithful`, usando a função `data()`.
- Entendendo as variáveis `?faithful`.
- Crie uma nova variável `eruptions.cat`, onde: `eruptions > 3` são `longas` e caso contrário são `curtas`.
- Recodifica a variável `eruptions.cat` para `Grupo 1 = longas` e `Grupo 2 = curtas`.

Limpando e Organizando os dados

- 80% do tempo de uma análise de dados é gasto limpando e organizando os dados.

Happy families are all alike; every unhappy family is unhappy in its own way - Leo Tolstoy

- Dados bagunçados tem os seus defeitos próprios.
- Precisando então de um conjunto de ferramentas que seja capaz de organizar e limpar a maioria dos problemas encontrados em base de dados.



Dados organizados:

- Cada variável forma uma coluna.
- Cada observação forma uma linha.
- Nome das variáveis não deve conter dado.

Limpando e Organizando os dados

- Vamos retomar o objeto `df`.

```
> df
  nomes altura peso      sexo      imc      imc.cat
1 JoAo30    145   45 Masculino 21.40309  Peso normal
2 lara43    184   64 Feminino 18.90359  Peso normal
3 Manoel21   191   85 Masculino 23.29980  Peso normal
4 Pedro34    160   65 Feminino 25.39062 Acima do peso
5 DeNIse25   158   58 Masculino 23.23346  Peso normal
```

1. Note que a variável `nomes` contém alguns números no final de cada nome. Estes valores representam a idade de cada indivíduo.
2. Note também que alguns nomes possuem problemas com as letras maiúsculas e minúsculas.

Limpando e Organizando os dados

- Separando o `nomes` da `idade` e criando uma nova variável:
- Usando as funções para manipular strings: pacote `stringr`.
- `str_extract()`: função para extrair exatamente o padrão desejado.
- `str_to_title()`: função para deixar a primeira letra maiúscula e de cada palavra.

```
> library(stringr)
> df$idade <- str_extract(string = df$nomes, pattern = "\\d+") #extraindo os numeros da string
> df$nomes <- str_replace(string = df$nomes, pattern = "\\d+", replacement = "")
> df$nomes <- str_to_title(df$nomes)
> df
    nomes altura peso      sexo      imc      imc.cat idade
1   Joao     145   45 Masculino 21.40309 Peso normal    30
2   Lara     184   64 Feminino 18.90359 Peso normal    43
3 Manoel    191   85 Masculino 23.29980 Peso normal    21
4 Pedro     160   65 Feminino 25.39062 Acima do peso    34
5 Denise    158   58 Masculino 23.23346 Peso normal    25
```

Pacote: `tidyverse`

- O pacote `tidyverse` foi desenvolvido para facilitar algumas operações de organização e limpeza dos dados.

`tidyverse`

- `gather()`: empilha a tabela de dados.
- `spread()`: desempilha a tabela de dados.
- `separate()`: separa uma colunas em duas ou mais colunas.
- `unite()`: une duas ou mais colunas.

tidyverse::gather()

- **gather()**: empilha a tabela de dados
- **data**: objeto com os dados que serão empilhados
- **key**: nome para armazenar a variável que será criada.
- **value**: nome para armazenar a variável de valores.
- **...**: indices das colunas para serem empilhadas.

country	year	cases	country	1999	2000
Afghanistan	1999	745	Afghanistan	745	2666
Afghanistan	2000	2666	Brazil	37737	80488
Brazil	1999	37737	China	212258	213766
Brazil	2000	80488			
China	1999	212258			
China	2000	213766			

The diagram illustrates the transformation of the 'cases' column from the first table into the '1999' and '2000' columns of the second table. Six arrows originate from the 'cases' values in the first table and point to the corresponding '1999' and '2000' values in the second table. The arrows are grouped by country: one group for Afghanistan (745 to 745, 2666), one for Brazil (37737 to 37737, 80488), and one for China (212258 to 212258, 213766).

tidyverse::gather()

- Exemplificando a função `gather`

```
> library(tidyverse)
> table4a
# A tibble: 3 × 3
      country `1999` `2000`
      <chr>    <int>   <int>
1 Afghanistan     745    2666
2 Brazil         37737   80488
3 China          212258  213766
```

```
> gather(data = table4a, key = "ano", value = "casos", 2:3)
# A tibble: 6 × 3
      country     ano   casos
      <chr>    <chr>   <int>
1 Afghanistan 1999     745
2 Brazil       1999   37737
3 China        1999  212258
4 Afghanistan 2000    2666
5 Brazil       2000   80488
6 China        2000  213766
```

tidyverse::spread()

- **spread()**: desempilha a tabela de dados
- **data**: objeto com os dados que serão empilhados
- **key**: nome da variável “categorias”.
- **value**: nome da variável de valores.

The diagram illustrates the process of spreading data from two separate data frames into a single, wider data frame. On the left, there are two data frames. The first data frame has columns: country, year, key, and value. It contains four rows for Afghanistan (1999 cases: 745, population: 19987071) and three rows for Brazil (1999 cases: 37737, population: 172006362) and China (1999 cases: 212258, population: 1272915272). The second data frame also has columns: country, year, cases, and population. It contains four rows for Afghanistan (2000 cases: 2666, population: 20595360), two rows for Brazil (2000 cases: 80488, population: 174504898), and two rows for China (2000 cases: 213766, population: 1280428583). Curved arrows point from each row in the first data frame to the corresponding row in the second data frame, indicating the mapping of the 'key' and 'value' columns to the 'cases' and 'population' columns of the second data frame.

country	year	key	value	country	year	cases	population
Afghanistan	1999	cases	745	Afghanistan	1999	745	19987071
Afghanistan	1999	population	19987071	Afghanistan	2000	2666	20595360
Afghanistan	2000	cases	2666	Brazil	1999	37737	172006362
Afghanistan	2000	population	20595360	Brazil	2000	80488	174504898
Brazil	1999	cases	37737	China	1999	212258	1272915272
Brazil	1999	population	172006362	China	2000	213766	1280428583
Brazil	2000	cases	80488				
Brazil	2000	population	174504898				
China	1999	cases	212258				
China	1999	population	1272915272				
China	2000	cases	213766				
China	2000	population	1280428583				

tidyverse::spread()

- Exemplificando a função `spread()`

```
> head(table2)
# A tibble: 6 × 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
```

```
> spread(data = table2, key = "type", value = "count")
# A tibble: 6 × 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
```

Agora é a sua vez

`gather()`

1. Carregue os dados `data/dadosForno.RData` com a função `load()`.
2. Selecione as variáveis: `IDC_REGISTRO`, `PRS_FORNO`, `PRS_GAS`, `PRS_AR`.
3. Use a função `gather()` para empilhar os dados.
4. Remova a string `PRS_`.

spread()

1. Use a função `spread()` no tabela gerada no exercício anterior para retornar para a tabela original.
2. Note no nome das variáveis.

tidyverse::separate()

- **separate()**: separa uma colunas em duas ou mais colunas.
- **data**: objeto com os dados que serão empilhados
- **col**: nome da variável que armazena os valores que serão separados.
- **into**: vetor de caracteres informando o nome das novas variáveis.
- **sep**: caractere que separa as colunas

```
> table3
# A tibble: 6 × 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
```

```
tidyverse::separate()
```

```
> separate(data = table3, col = rate, into = c("casos", "populacao"), sep = ",")  
# A tibble: 6 × 4  
# Groups:   country [2]  
  country     year    casos populacao  
  <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
```

tidyverse::unite()

- **unite()**: é o oposto **separate**.
- **data**: objeto com os dados que serão empilhados.
- **col**: nome da variável que será criada.
- **...**: especifica as colunas que serão unificadas.
- **sep**: caractere que separa os valores ou a posição

```
> table5
# A tibble: 6 × 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
```

```
tidyverse::unite()
```

```
> unite(data = table5, col = "ano", century, year, sep = "")  
# A tibble: 6 × 3  
    country     ano           rate  
* <chr>      <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
```

Agora é a sua vez

separate()

1. Carregue os dados `data/dadosFicEnf.RData`.
2. Crie um novo `data.frame` com todas as variáveis com que contém a palavra `ZONA` e `IDC_REGISTRO`.
3. Use a função `gather()` para empilhando as colunas com a palavra `ZONA`.
4. Use a função `separate()` para criar duas colunas, sendo uma delas contenha apenas a `ZONA`.

Módulo XI: Tratamento e Manipulação de Dados

Família apply()

Família `apply()`

- Anteriormente estudamos algumas estruturas de *loops*, entretanto sabemos que é sempre melhor usarmos a característica vetorial do R.
- As funções vetorizadas mais importantes no R para análise de dados são as funções `apply()`.
- Estas funções tem como argumento uma função que será aplicada em forma de loop em alguma estrutura de dados.

Manipulação de dados

- `apply()`: aplica uma função em linhas ou colunas de uma matriz/data.frame.
- `lapply()`: aplica uma função em cada posição da lista e retorna uma lista.
- `sapply()`: aplica uma função em cada posição da lista e retorna um vetor.
- `mapply()`: versão multivariada da `sapply()`.
- `tapply()`: aplica uma função por grupo/categoria.

Função `apply()`

Os argumentos da função `apply()` são os seguintes:

- **X**: matriz ou data.frame.
- **MARGIN**: 1 aplica nas linhas; 2 aplica nas colunas.
- **FUN**: função a ser aplicada.
- ...: argumentos extras da função a ser aplicada

Função `apply()`: exemplos

- Exemplificando a função `apply()`

```
> mat <- matrix(rnorm(30), ncol = 6, nrow = 5) #gerando uma matriz aleatoria
> apply(X = mat, MARGIN = 1, FUN = sd) #desvio-padrao das linhas
> apply(X = mat, MARGIN = 2, FUN = sd) #desvio-padrao das colunas
[1] 1.2061661 0.4833341 0.9753170 0.8181863 1.3802540
[1] 0.6780942 1.3151118 0.5977216 1.0246014 1.0748801 0.7338465
```

Função `apply()`: exemplos

- Exemplificando a função `apply()`

```
> mat <- matrix(rnorm(30), ncol = 6, nrow = 5) #gerando uma matriz aleatoria
> apply(X = mat, MARGIN = 1, FUN = sd) #desvio-padrão das linhas
> apply(X = mat, MARGIN = 2, FUN = sd) #desvio-padrão das colunas
[1] 1.2061661 0.4833341 0.9753170 0.8181863 1.3802540
[1] 0.6780942 1.3151118 0.5977216 1.0246014 1.0748801 0.7338465
```

- Podemos definir nossas próprias funções e então usar `apply()`:

```
> soma_quadrado <- function(x) {
+   out <- sum(x^2)
+   return(out)
+ }
> apply(X = mat, MARGIN = 1, FUN = soma_quadrado)
[1] 9.214498 1.201749 6.204277 3.656018 9.867534
```

```
> apply(X = mat, MARGIN = 2, FUN = soma_quadrado)
[1] 2.318818 7.704480 2.870958 7.742678 5.121916 4.385226
```

Função `apply()`: exemplos

- Exemplificando a função `apply()`

```
> mat <- matrix(rnorm(30), ncol = 6, nrow = 5) #gerando uma matriz aleatoria
> apply(X = mat, MARGIN = 1, FUN = sd) #desvio-padrão das linhas
> apply(X = mat, MARGIN = 2, FUN = sd) #desvio-padrão das colunas
[1] 1.2061661 0.4833341 0.9753170 0.8181863 1.3802540
[1] 0.6780942 1.3151118 0.5977216 1.0246014 1.0748801 0.7338465
```

- Podemos definir nossas próprias funções e então usar `apply()`:

```
> soma_quadrado <- function(x) {
+   out <- sum(x^2)
+   return(out)
+ }
> apply(X = mat, MARGIN = 1, FUN = soma_quadrado)
[1] 9.214498 1.201749 6.204277 3.656018 9.867534
```

```
> apply(X = mat, MARGIN = 2, FUN = soma_quadrado)
[1] 2.318818 7.704480 2.870958 7.742678 5.121916 4.385226
```

- Qual a outra forma de fazermos a operação anterior?

Função `apply()`: exemplos

- Exemplificando a função `apply()`

```
> mat <- matrix(rnorm(30), ncol = 6, nrow = 5) #gerando uma matriz aleatoria
> apply(X = mat, MARGIN = 1, FUN = sd) #desvio-padrão das linhas
> apply(X = mat, MARGIN = 2, FUN = sd) #desvio-padrão das colunas
[1] 1.2061661 0.4833341 0.9753170 0.8181863 1.3802540
[1] 0.6780942 1.3151118 0.5977216 1.0246014 1.0748801 0.7338465
```

- Podemos definir nossas próprias funções e então usar `apply()`:

```
> soma_quadrado <- function(x) {
+   out <- sum(x^2)
+   return(out)
+ }
> apply(X = mat, MARGIN = 1, FUN = soma_quadrado)
[1] 9.214498 1.201749 6.204277 3.656018 9.867534
```

```
> apply(X = mat, MARGIN = 2, FUN = soma_quadrado)
[1] 2.318818 7.704480 2.870958 7.742678 5.121916 4.385226
```

- Qual a outra forma de fazermos a operação anterior?
- `?rowSums` e `?colSums`

Função `apply()`: exemplos

- Passando argumentos para a função que será aplicada.

```
> soma_potencia <- function(x, potencia, ...) {  
+   out <- sum(x^potencia, ...)   
+   return(out)  
+ }  
> apply(X = mat, MARGIN = 1, FUN = soma_potencia, potencia = 3)  
[1] -9.9074053 -0.7051484 -12.5543557  1.3767931 -9.3717230
```

Função `apply()`: exemplos

- Passando argumentos para a função que será aplicada.

```
> soma_potencia <- function(x, potencia, ...) {  
+   out <- sum(x^potencia, ...)   
+   return(out)  
+ }  
> apply(X = mat, MARGIN = 1, FUN = soma_potencia, potencia = 3)  
[1] -9.9074053 -0.7051484 -12.5543557  1.3767931 -9.3717230
```

- Usando o argumento `na.rm` através do argumento `....`

```
> mat[1, 1] <- NA #inserindo NA na primeira linha e primeira coluna  
> apply(X = mat, MARGIN = 2, FUN = soma_potencia, potencia = 3)  
[1] NA -11.090702  2.923621 -14.819562 -3.848098 -5.836177
```

```
> apply(X = mat, MARGIN = 2, FUN = soma_potencia, potencia = 3, na.rm = T)  
[1]  0.3848404 -11.0907019  2.9236215 -14.8195623 -3.8480985 -5.8361769
```

Agora é a sua vez

Função apply()

1. Faça uma matrix com 10 linhas e 15 colunas com valores aleatórios de uma distribuição normal/Gaussiana com média 50 e desvio-padrão 5.
2. Faça uma função que calcule a razão entre o mínimo e o máximo.
3. Aplique esta função nas linhas e colunas da matriz criada no primeiro item.

Funções `lapply()` e `sapply()`: argumentos

- As funções `lapply()` e `sapply()` percorrem todas as posições de uma lista e em cada uma delas aplica a função definida no argumento `FUN`.

Função `lapply()`

Os argumentos da função `lapply()` são os seguintes:

- **X**: lista
- **FUN**: função a ser aplicada
- ...: argumentos extras da função a ser aplicada

Funções `lapply()` e `sapply()`: exemplos

- Exemplificando a função `lapply()`

```
> lista <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE, FALSE, FALSE, TRUE))
> lapply(X = lista, FUN = mean)
$a
[1] 5.5

$beta
[1] 4.535125

$logic
[1] 0.5
```

Funções `lapply()` e `sapply()`: exemplos

- Passando argumentos para a função `?quantile()` que será aplicada.

```
> lapply(lista, quantile, probs = 1:3/4) #probs é argumento da função quantile()
$a
 25%   50%   75%
3.25  5.50  7.75

$beta
      25%      50%      75%
0.2516074 1.0000000 5.0536690

$logic
 25% 50% 75%
0.0 0.5 1.0
```

Funções `lapply()` e `sapply()`: exemplos

- Usando a função `sapply()` a única diferença é a estrutura da saída.

```
> sapply(lista, mean)
      a     beta    logic
5.500000 4.535125 0.500000
```

```
> sapply(lista, quantile, probs = 1:3/4)
      a     beta logic
25% 3.25 0.2516074 0.0
50% 5.50 1.0000000 0.5
75% 7.75 5.0536690 1.0
```

Agora é a sua vez

Funções `lapply()` e `sapply()`

1. Carregue o conjunto de dados `?cars` com a função `data()`.
2. Faça uma função que calcule a diferença entre o máximo e o mínimo.
3. Aplique esta função no conjunto de dados `?cars` usando as funções: `lapply` e `sapply`. Lembre que um `data.frame` é um caso especial de lista.
4. Faça um gráfico de dispersão `speed × dist`.
5. Faça uma regressão linear `dist ~ speed` e apresente a linha de regressão no gráfico.

Função `mapply()`: argumentos

- A função `mapply()` é usada quando cada argumento da função está contido, por exemplo, em cada coluna de uma matriz ou data.frame.

Função `mapply()`

Os argumentos da função `mapply()` são os seguintes:

- **FUN**: função a ser aplicada.
- **...:** argumentos que serão vetorizados.
- **MoreArgs**: uma lista com outros argumentos extras.

Função `mapply()`: exemplos

- Exemplificando a função `mapply()`:

```
> mapply(rep, times = 1:4, x = 4:1)
[[1]]
[1] 4

[[2]]
[1] 3 3

[[3]]
[1] 2 2 2

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

- Criando a nossa própria função:

```
> imc_fun <- function(peso, altura) {
+   imc <- peso/(altura/100)^2
+   return(imc)
+ }
> library(car)
> mapply(FUN = imc_fun, peso = Davis$weight, altura = Davis$height)
```

Função `tapply()`: argumentos

- A função `tapply()` aplica uma determinada função para cada categoria/fator. Exemplo, média de temperatura por forno.

Função `tapply()`

Os argumentos da função `tapply()` são os seguintes:

- **X**: um vetor de valores.
- **FUN**: função a ser aplicada.
- **INDEX**: lista dos fatores/grupos para valor do vetor **X**.
- ...: argumentos opcionais para a função **FUN**.

Função `tapply()`: exemplos

- Exemplificando a função `tapply()`: vamos calcular a média da variável `Sepal.Length` pela variável `Species` da base de dados `?iris`.

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

```
> tapply(X = iris$Sepal.Length, INDEX = iris$Species, FUN = mean)
    setosa versicolor  virginica
      5.006      5.936      6.588
```

Função `tapply()`: exemplos

- Podemos usar mais de uma variável de grupo: vamos usar a base de dados `mtcars`.

```
> head(mtcars)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

```
> tapply(X = mtcars$mpg, INDEX = list(mtcars$cyl, mtcars$am), FUN = mean)
```

0	1
4 22.900	28.07500
6 19.125	20.56667
8 15.050	15.40000

Agora é a sua vez

Função tapply()

1. Carregue a base de dados `Davis` do pacote `car` usando a função `data`.
2. Calcule o peso médio por sexo.
3. Carregue a base de dados `Orange` do pacote `base` usando a função `data`.
4. Calcule a circunferência e idade média para os diferentes tipos de árvores.
5. Faça um boxplot da variável `circumference` para cada tipo de `Tree`.

Análise/Manipulação de Dados

Manipulação de dados

1. Filtro (seleção de observações): `subset()`, `dplyr::filter()`;
2. Ordenar base de dados: `dplyr::arrange()`;
3. Casos duplicados: `duplicated()`;
4. Reunir duas bases de dados: `merge()`;
5. Agregar base de dados: `aggregate()`, `dplyr::summarise()`.
6. Seleção de variáveis: `dplyr::select()`;
7. Criação de variáveis: `dplyr::mutate()`;

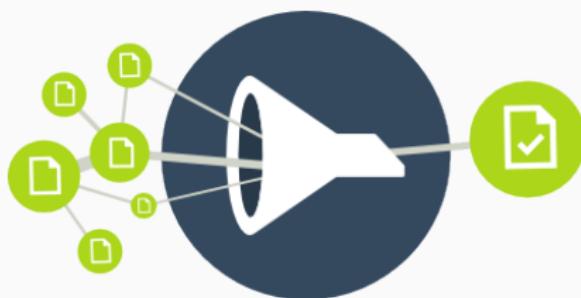


Filtrando observações

- Usando o **R base**, nós temos duas opções para selecionar observações/linhas:

Funções para filtrar

- [TESTE LOGICO, COLUNA]:
- `subset()`
 - `x`: `data.frame`
 - `subset`: teste lógico
 - `select`: nome da coluna (opcional)



Filtrando observações: exemplos

- Exemplificando a função `subset()`.

```
> subset(airquality, Temp > 92, select = c(Ozone, Temp))
```

- Usando o índice negativo `-`.

```
> subset(airquality, Day == 1, select = -Temp)
```

- Selecionando mais de uma coluna com a função `:`.

```
> subset(airquality, select = Ozone:Wind)
```

- O resultado da função `subset()` pode ser armazenado em uma outra variável:

```
> dia_um <- subset(airquality, Day == 1, select = -Temp)
```

Filtrando observações: exemplos

- Filtrando valores NA.

```
> dados_na <- subset(airquality, Ozone == NA, select = -Temp)
> dim(dados_na)
[1] 0 5
```

Filtrando observações: exemplos

- Filtrando valores NA.

```
> dados_na <- subset(airquality, Ozone == NA, select = -Temp)
> dim(dados_na)
[1] 0 5
```

- Para filtrar os valores NA existe uma função especial: `is.na()`

```
> dados_na <- subset(airquality, is.na(Ozone), select = -Temp)
> dim(dados_na)
[1] 37  5
```

- Usando a função `%in%` para filtrar mais de uma categoria

```
> seleciona_dias <- subset(airquality, Day %in% c(1, 10, 25))
```

Encontrando valores duplicados: `duplicated()`

- Para encontrar valores duplicados em uma variável/matriz/data.frame devemos usar a função: `duplicated()`.
 - `x`: o vetor/matriz/data.frame para procurar os valores duplicados
 - `fromLast`: indicador lógico indicando se o segundo valor encontrado é o duplicado ou o primeiro.

```
> x <- c(9:20, 1:5, 3:7, 0:8)
> duplicated(x)
[1] FALSE FALSE
[14] FALSE FALSE FALSE FALSE TRUE  TRUE  TRUE FALSE FALSE TRUE  TRUE  TRUE
[27] TRUE  TRUE  TRUE  TRUE FALSE
```

Encontrando valores duplicados: `duplicated()`

- Para encontrar valores duplicados em uma variável/matriz/data.frame devemos usar a função: `duplicated()`.
 - `x`: o vetor/matriz/data.frame para procurar os valores duplicados
 - `fromLast`: indicador lógico indicando se o segundo valor encontrado é o duplicado ou o primeiro.

```
> x <- c(9:20, 1:5, 3:7, 0:8)
> duplicated(x)
[1] FALSE FALSE
[14] FALSE FALSE FALSE FALSE TRUE  TRUE  TRUE FALSE FALSE TRUE  TRUE  TRUE
[27] TRUE  TRUE  TRUE  TRUE FALSE
```

- Uma vez que a função `duplicated()` retorna um vetor lógico, podemos usar o resultado dela para selecionar os valores únicos.

```
> (xu <- x[!duplicated(x)])
[1]  9 10 11 12 13 14 15 16 17 18 19 20  1  2  3  4  5  6  7  0  8
```

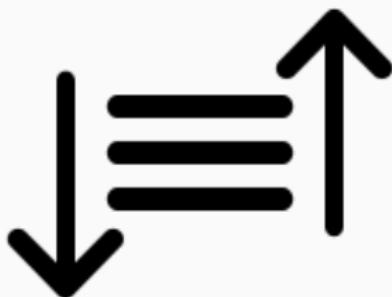
Agora é a sua vez

Função `duplicated()` e `subset()`

1. Carregue a base de dados `data/alunos.csv`, separada por ; e decimal é ..
2. Use as funções `duplicated()` e `subset()` para encontrar os alunos duplicados.
3. Crie outro objeto sem as linhas duplicadas.

Ordenação: `dplyr::arrange()`

- Ordenar uma coluna por ordem crescente ou decrescente é uma tarefa comum em análise de dados. O R possui as funções `order()` e `sort()`. Porém a função `arrange()` do pacote `dplyr` é mais intuitiva.
- `arrange()`: ordena uma base de dados de acordo com as colunas que são passadas.
 - `.data`: `data.frame` que será ordenado
 - `...`: variáveis separadas por vírgula que serão usadas para ordenar a base.
 - `desc()`: função que indica se é para order na ordem decrescente.



dplyr::arrange(): exemplo

- Exemplificando a função `arrange()`.

```
> library(dplyr)
> alunos <- read.table("../data/alunos.csv", sep = ";", dec = ".", header = T)
> alunos <- arrange(alunos, VL_PRF_ALN_11)
> head(alunos)
  CD_ALUNO CD_ESCOLA VL_PRF_ALN_11
1    56033  31217824    286.1902
2    13626  31001422    286.1902
3   143874  31213705    286.1902
4    22768  31011771    289.1880
5    19321  31011347    290.0728
6   105032  31272311    290.4429
```

- Exemplificando a função `order()`.

```
> alunos <- alunos[order(alunos$VL_PRF_ALN_11), ]
> head(alunos)
  CD_ALUNO CD_ESCOLA VL_PRF_ALN_11
1    56033  31217824    286.1902
2    13626  31001422    286.1902
3   143874  31213705    286.1902
4    22768  31011771    289.1880
5    19321  31011347    290.0728
6   105032  31272311    290.4429
```

dplyr::arrange(): exemplo

- Ordem decrescente com a função `desc()`

```
> alunos <- arrange(alunos, desc(VL_PRF_ALN_11))
> head(alunos)
  CD_ALUNO CD_ESCOLA VL_PRF_ALN_11
1    175335   31248304     752.415
2    180166   31081604     752.415
3    272858   31166782     752.415
4    243876   31310662     752.415
5    11665    31001163     752.415
6    236058   31134643     752.415
```

- Ordem decrescente com a função `order()`

```
> alunos <- alunos[order(alunos$VL_PRF_ALN_11, decreasing = TRUE), ]
> head(alunos)
  CD_ALUNO CD_ESCOLA VL_PRF_ALN_11
1    175335   31248304     752.415
2    180166   31081604     752.415
3    272858   31166782     752.415
4    243876   31310662     752.415
5    11665    31001163     752.415
6    236058   31134643     752.415
```

dplyr::arrange(): exemplo

- Ordenando com mais de uma coluna.

```
> head(arrange(mtcars, cyl, disp))  
   mpg cyl  disp  hp drat    wt  qsec vs am gear carb  
1 33.9   4  71.1 65 4.22 1.835 19.90  1  1     4    1  
2 30.4   4  75.7 52 4.93 1.615 18.52  1  1     4    2  
3 32.4   4  78.7 66 4.08 2.200 19.47  1  1     4    1  
4 27.3   4  79.0 66 4.08 1.935 18.90  1  1     4    1  
5 30.4   4  95.1 113 3.77 1.513 16.90  1  1     5    2  
6 22.8   4 108.0 93 3.85 2.320 18.61  1  1     4    1
```

```
> head(arrange(mtcars, desc(disp)))  
   mpg cyl  disp  hp drat    wt  qsec vs am gear carb  
1 10.4   8  472 205 2.93 5.250 17.98  0  0     3    4  
2 10.4   8  460 215 3.00 5.424 17.82  0  0     3    4  
3 14.7   8  440 230 3.23 5.345 17.42  0  0     3    4  
4 19.2   8  400 175 3.08 3.845 17.05  0  0     3    2  
5 18.7   8  360 175 3.15 3.440 17.02  0  0     3    2  
6 14.3   8  360 245 3.21 3.570 15.84  0  0     3    4
```

Agora é a sua vez

Função `arrange()`

1. Carregue a base de dados `data/alunos.csv`, separada por ; e decimal é ..
2. Ordene a base de dados de acordo com as variáveis `CD_ESCOLA` e `CD_ALUNO`.
3. Faça um histograma da varável `VL_PRF_ALN_11` e desenhe a curva da normal com a cor vermelha. Dica: `curve`.

Combinando dois data.frames: `merge()`

- Combinar dois data.frames de acordo uma ou mais variáveis de identificação é um tarefa muito simples no R. Basta usarmos a função `merge()`

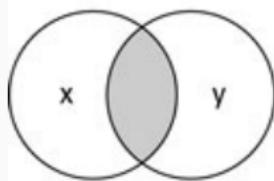
Função `merge()`

- `x`, `y`: data.frames que serão combinado em um único data.frame
- `by`, `by.x`, `by.y`: nomes das variáveis de chaveamento.
- `all`, `all.x`, `all.y`: indicadora se deve manter as linhas que não tiveram par com o outro data.frame.
- `suffixes`: quando existem as mesmas variáveis em ambos data.frames, então é necessário colocar os sufixos de cada data.frame. Padrão: `.x` e `.y`.

Combinando dois data.frames: `merge()`

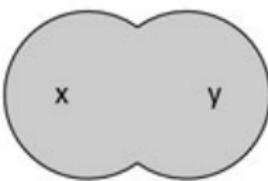
- Explicando o argumento: `all`, `all.x`, `all.y`

`all = FALSE`



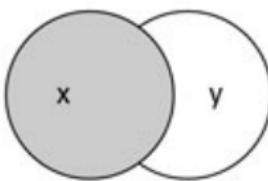
natural join

`all = TRUE`



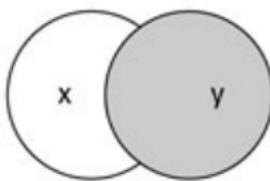
full outer join

`all.x = TRUE`



left outer join

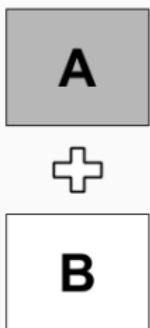
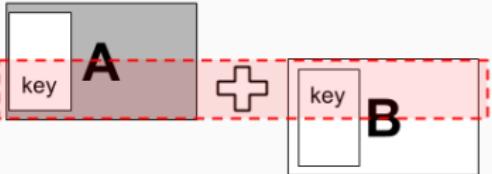
`all.y = TRUE`



right outer join

Combinando dois data.frames: `merge()`

- Diferença entre `rbind()`, `cbind()` e `merge()`

<code>rbind(A, B)</code>	<code>cbind(A, B)</code>	<code>merge(A, B, by = "key")</code>
		

merge(): exemplos

- Exemplificando função `merge`

```
> authors <- data.frame(  
+   surname = I(c("Tukey", "Venables", "Tierney", "Ripley", "McNeil")),  
+   nationality = c("US", "Australia", "US", "UK", "Australia"),  
+   deceased = c("yes", rep("no", 4)))  
> books <- data.frame(  
+   name = I(c("Tukey", "Venables", "Tierney",  
+             "Ripley", "Ripley", "McNeil", "R Core")),  
+   title = c("Exploratory Data Analysis",  
+             "Modern Applied Statistics ...",  
+             "LISP-STAT",  
+             "Spatial Statistics", "Stochastic Simulation",  
+             "Interactive Data Analysis",  
+             "An Introduction to R"),  
+   other.author = c(NA, "Ripley", NA, NA, NA, NA,  
+                 "Venables & Smith"))
```

merge(): exemplos

- Exemplificando função `merge`

```
> (m1 <- merge(authors, books, by.x = "surname", by.y = "name"))
   surname nationality deceased           title other.author
1   McNeil     Australia      no Interactive Data Analysis      <NA>
2   Ripley        UK          no       Spatial Statistics      <NA>
3   Ripley        UK          no Stochastic Simulation      <NA>
4  Tierney        US          no            LISP-STAT      <NA>
5   Tukey         US         yes Exploratory Data Analysis      <NA>
6 Venables     Australia      no Modern Applied Statistics ...    Ripley
```

```
> (m2 <- merge(books, authors, by.x = "name", by.y = "surname"))
   name           title other.author nationality deceased
1 McNeil Interactive Data Analysis      <NA>     Australia      no
2   Ripley       Spatial Statistics      <NA>          UK      no
3   Ripley       Stochastic Simulation      <NA>          UK      no
4  Tierney        LISP-STAT      <NA>          US      no
5   Tukey Exploratory Data Analysis      <NA>          US     yes
6 Venables Modern Applied Statistics ...    Ripley     Australia      no
```

merge(): exemplos

- R-core apenas aparece se usarmos o argumento `all = TRUE`.

```
> merge(authors, books, by.x = "surname", by.y = "name", all = TRUE)
   surname nationality deceased           title      other.author
1   McNeil     Australia       no  Interactive Data Analysis      <NA>
2   R Core          <NA>      <NA>  An Introduction to R Venables & Smith
3   Ripley        UK         no    Spatial Statistics      <NA>
4   Ripley        UK         no  Stochastic Simulation      <NA>
5 Tierney        US         no        LISP-STAT      <NA>
6   Tukey        US        yes Exploratory Data Analysis      <NA>
7 Venables  Australia       no Modern Applied Statistics ...      Ripley
```

Agora é a sua vez

Função `merge()`

1. Carregue a base de dados `data/alunos.csv`, separada por ; e decimal é ..
2. Carregue a base de dados `data/escola.csv`, separada por ; e decimal é ..
3. Use a função `merge()` para combinar as duas bases de dados.
4. Verifique a dimensões das bases.
5. Ordene a base em ordem decrescente pela variável `VL_PRF_ALN_11`.
6. Calcule a média da variável `VL_PRF_ALN_11` para cada escola.
7. Construa um novo data.frame com o `CD_ESCOLA` e a média de cada escola.
Dica 1: argumento `simplify` da função `tapply()`. Dica 2: a função `unlist()`.

Pacote dplyr

Pacote `dplyr`

- O pacote `dplyr` é amplamente utilizado para manipulação de dados, principalmente pela facilidade.
- A facilidade do pacote é pelo fato de usar o operador `%>%`, mais adiante ficará claro o uso dele.



dplyr::filter()

- A função `filter()` funciona da mesma forma que função `subset()` do pacote base do R.

```
> library(dplyr)
> load('../data/flights.RData')
> dim(flights)
[1] 336776     19
```

```
> head(flights)
# A tibble: 6 × 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>      <dbl>    <int>          <int>
1  2013     1     1      517            515        2     830          819
2  2013     1     1      533            529        4     850          830
3  2013     1     1      542            540        2     923          850
4  2013     1     1      544            545       -1    1004         1022
5  2013     1     1      554            600       -6     812          837
6  2013     1     1      554            558       -4     740          728
# ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

dplyr::filter()

- Filtrando a base de dados com a função `filter()`. Note a diferença com a função `subset()`

```
> filter(flights, month == 1, day == 1)
# A tibble: 842 × 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     1     1      517            515        2     830          819
2  2013     1     1      533            529        4     850          830
3  2013     1     1      542            540        2     923          850
4  2013     1     1      544            545       -1    1004         1022
5  2013     1     1      554            600       -6     812          837
6  2013     1     1      554            558       -4     740          728
7  2013     1     1      555            600       -5     913          854
8  2013     1     1      557            600       -3     709          723
9  2013     1     1      557            600       -3     838          846
10 2013     1     1      558            600       -2     753          745
# ... with 832 more rows, and 11 more variables: arr_delay <dbl>, carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

- Agora podemos passar as condições separadas por vírgula.

dplyr::select()

- **select** é uma função para selecionar variáveis de forma bastante simples:
- Selecionando pelos nomes das variáveis:

```
> select(flights, year, month, day)
# A tibble: 336,776 × 3
  year month   day
  <int> <int> <int>
1 2013     1     1
2 2013     1     1
3 2013     1     1
4 2013     1     1
5 2013     1     1
6 2013     1     1
7 2013     1     1
8 2013     1     1
9 2013     1     1
10 2013    1     1
# ... with 336,766 more rows
```

dplyr::select()

- Selecionando todas as colunas que estão entre duas colunas: `year:day`.

```
> select(flights, year:day)
# A tibble: 336,776 × 3
  year month   day
  <int> <int> <int>
1 2013     1     1
2 2013     1     1
3 2013     1     1
4 2013     1     1
5 2013     1     1
6 2013     1     1
7 2013     1     1
8 2013     1     1
9 2013     1     1
10 2013    1     1
# ... with 336,766 more rows
```

dplyr::select()

- Seleciona todas as colunas, exceto as colunas que estão entre
-(year:day)

```
> select(flights, -(year:day))
# A tibble: 336,776 × 16
  dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier
    <int>        <int>     <dbl>    <int>        <int>     <dbl>    <chr>
1      517          515       2     830         819      11     UA
2      533          529       4     850         830      20     UA
3      542          540       2     923         850      33     AA
4      544          545      -1    1004        1022     -18     B6
5      554          600      -6     812         837     -25     DL
6      554          558      -4     740         728      12     UA
7      555          600      -5     913         854      19     B6
8      557          600      -3     709         723     -14     EV
9      557          600      -3     838         846      -8     B6
10     558          600      -2     753         745       8     AA
# ... with 336,766 more rows, and 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

dplyr::select()

- Outras formas de selecionar variáveis:

```
> iris <-tbl_df(iris) # so it prints a little nicer
> select(iris, starts_with("Petal")) #inicia com a string Petal
> select(iris, ends_with("Width"))   #finaliza com a string Width
> select(iris, contains("etal"))    #contem a string etal
```

dplyr::mutate()

- `mutate()` é uma função que facilita a criação de novas variáveis dentro do nosso `data.frame`.

```
> mutate(flights,
+   gain = arr_delay - dep_delay,
+   speed = distance / air_time * 60)
# A tibble: 336,776 × 21
  year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>     <int>          <int>     <dbl>    <int>          <int>
1  2013     1     1      517            515        2     830            819
2  2013     1     1      533            529        4     850            830
3  2013     1     1      542            540        2     923            850
4  2013     1     1      544            545       -1    1004           1022
5  2013     1     1      554            600       -6     812            837
6  2013     1     1      554            558       -4     740            728
7  2013     1     1      555            600       -5     913            854
8  2013     1     1      557            600       -3     709            723
9  2013     1     1      557            600       -3     838            846
10 2013     1     1      558            600       -2     753            745
# ... with 336,766 more rows, and 13 more variables: arr_delay <dbl>, carrier <chr>,
# flight <int>, tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
# distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>, gain <dbl>,
# speed <dbl>
```

dplyr::mutate()

- Com a função `mutate()` podemos criar novas variáveis a partir de variáveis que ainda não foram definidas.

```
> mutate(flights,
+   gain = arr_delay - dep_delay,
+   gain_per_hour = gain / (air_time / 60)
+ )
# A tibble: 336,776 × 21
  year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     1     1      517            515        2     830          819
2  2013     1     1      533            529        4     850          830
3  2013     1     1      542            540        2     923          850
4  2013     1     1      544            545       -1    1004         1022
5  2013     1     1      554            600       -6     812          837
6  2013     1     1      554            558       -4     740          728
7  2013     1     1      555            600       -5     913          854
8  2013     1     1      557            600       -3     709          723
9  2013     1     1      557            600       -3     838          846
10 2013     1     1      558            600       -2     753          745
# ... with 336,766 more rows, and 13 more variables: arr_delay <dbl>, carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>, gain <dbl>,
#   gain_per_hour <dbl>
```

dplyr::mutate()

- Se desejar criar um novo data.frame somente com as variáveis novas, use: **transmute()**.

```
> transmute(flights,
+   gain = arr_delay - dep_delay,
+   gain_per_hour = gain / (air_time / 60)
+ )
# A tibble: 336,776 × 2
  gain gain_per_hour
  <dbl>      <dbl>
1     9      2.378855
2    16      4.229075
3    31     11.625000
4   -17     -5.573770
5   -19     -9.827586
6    16      6.400000
7    24      9.113924
8   -11     -12.452830
9    -5     -2.142857
10   10      4.347826
# ... with 336,766 more rows
```

dplyr::summarise()

- A função `summarise()` é utilizada para obtermos um único valor do nosso `data.frame`. Por exemplo, a média de uma variável.

```
> summarise(flights,
+   delay = mean(dep_delay, na.rm = TRUE), sd = sd(dep_delay, na.rm = TRUE))
# A tibble: 1 × 2
  delay      sd
  <dbl>    <dbl>
1 12.63907 40.21006
```

dplyr::group_by()

- A função **group_by** extrapola a função **tapply**. Agora podemos calcular qualquer função por grupo e para múltiplas variáveis.

```
> by_tailnum <- group_by(flights, tailnum) #criando uma classe de grupos
> delay <- summarise(by_tailnum,
+   count = n(),
+   dist = mean(distance, na.rm = TRUE),
+   delay = mean(arr_delay, na.rm = TRUE))
> delay <- filter(delay, count > 20, dist < 2000)
> delay
# A tibble: 2,962 × 4
  tailnum count     dist    delay
  <chr>   <int>   <dbl>    <dbl>
1 NOEGMQ     371 676.1887  9.9829545
2 N10156      153 757.9477 12.7172414
3 N102UW       48 535.8750  2.9375000
4 N103US      46 535.1957 -6.9347826
5 N104UW      47 535.2553  1.8043478
6 N10575     289 519.7024 20.6914498
7 N105UW      45 524.8444 -0.2666667
8 N107US      41 528.7073 -5.7317073
9 N108UW      60 534.5000 -1.2500000
10 N109UW     48 535.8750 -2.5208333
# ... with 2,952 more rows
```

dplyr::group_by()

- Podemos agrupar com mais de uma variável.

```
> daily <- group_by(flights, year, month, day)
> (per_day   <- summarise(daily, flights = n()))
Source: local data frame [365 x 4]
Groups: year, month [?]

  year month  day flights
<int> <int> <int>    <int>
1  2013     1     1      842
2  2013     1     2      943
3  2013     1     3      914
4  2013     1     4      915
5  2013     1     5      720
6  2013     1     6      832
7  2013     1     7      933
8  2013     1     8      899
9  2013     1     9      902
10 2013     1    10      932
# ... with 355 more rows
```

Combinando todos os comandos:

```
> filter(  
+   summarise(  
+     select(  
+       group_by(flights, year, month, day),  
+       arr_delay, dep_delay  
+     ),  
+     arr = mean(arr_delay, na.rm = TRUE),  
+     dep = mean(dep_delay, na.rm = TRUE)  
+   ),  
+   arr > 30 | dep > 30  
+ )
```

Operador %>%

- Com o operador `%>%` podemos usar múltiplos comandos, sendo que o output de um comando é o input do próximo comando.

```
> flights %>%
+   group_by(year, month, day) %>%
+   select(arr_delay, dep_delay) %>%
+   summarise(
+     arr = mean(arr_delay, na.rm = TRUE),
+     dep = mean(dep_delay, na.rm = TRUE)
+   ) %>%
+   filter(arr > 30 | dep > 30)
```

Agora é a NOSSA vez

Pacote dplyr

1. Usando os dados `iris`.
2. Empilhe os dados usando a função `tidyverse::gather()`.
3. Calcule a media de cada atributo por espécie.
4. Ordene de acordo com a média de cada atributo.

Agora é a sua vez

Pacote dplyr

1. Use os dados de alunos já combinados com os dados das escolas:
2. Calcule a proficiência média de cada escola.
3. Filtre as linhas das escolas acima do quantil .90.
4. Quantas escola estão acima do quantil .90?
5. Ordene as escolas pela proficiência média.

Módulo XII: Integração R, Python, Oracle e D3

Integração R x Bancos de Dados

Pacotes R para acesso a Banco de Dados

1. RJDBC : Provê acesso a banco de dados através da interface JDBC;
2. RODBC : Interface para acesso a banco de dados via ODBC;
3. ROracle : Interface de acesso ao Oracle baseado na OCI(*Oracle Call Interface*);
4. RMySQL: Acesso ao MySQL e MariaDB;
5. RPostgreSQL : Acesso ao BD PostgreSQL;
6. RSQLite : Embute a *engine* do banco 'SQLite' no R (BD utilizados em *smartphones*).

Todos esses pacotes estão disponíveis no CRAN.

O que é YAML?

- **YAML** é uma linguagem para serialização de dados legível para o ser humano.

O que é YAML?

- **YAML** é uma linguagem para serialização de dados legível para o ser humano.
- **Serialização** é um processo de converter o estado de um objeto ou estrutura de dados para um formato que possa ser armazenado e posteriormente reconstruído no mesmo computador ou em outro.

O que é YAML?

- **YAML** é uma linguagem para serialização de dados legível para o ser humano.
- **Serialização** é um processo de converter o estado de um objeto ou estrutura de dados para um formato que possa ser armazenado e posteriormente reconstruído no mesmo computador ou em outro.
- **YAML** é comumente utilizada para **arquivos de configuração**.

O que é YAML?

- **YAML** é uma linguagem para serialização de dados legível para o ser humano.
- **Serialização** é um processo de converter o estado de um objeto ou estrutura de dados para um formato que possa ser armazenado e posteriormente reconstruído no mesmo computador ou em outro.
- **YAML** é comumente utilizada para **arquivos de configuração**.
- Também é utilizada em muitas aplicações onde dados são armazenados (ex.: arquivos de *debug*). **YAML** 1.2 é um superset de **JSON**.

O que é YAML?

- **YAML** é uma linguagem para serialização de dados legível para o ser humano.
- **Serialização** é um processo de converter o estado de um objeto ou estrutura de dados para um formato que possa ser armazenado e posteriormente reconstruído no mesmo computador ou em outro.
- **YAML** é comumente utilizada para **arquivos de configuração**.
- Também é utilizada em muitas aplicações onde dados são armazenados (ex.: arquivos de *debug*). **YAML** 1.2 é um superset de **JSON**.

Vamos utilizar a **YAML** para armazenar as configurações necessárias para acesso a banco de dados.

Arquivo de Configurações para acesso a Banco de Dados

```
> rmstr:  
+  dirScripts : '/home/rtrainning/RTrainning/R/'  
>  logFile : "/home/rtrainning/RTrainning/R/Log.log"  
> oracle:  
+  stringConexao:"jdbc:oracle:thin:usuario/senha@localhost:1522/XE"  
>  arqJar : "/u01/app/oracle/product/11.2.0/xe/jdbc/lib/ojdbc6.jar"  
> pacotes:  
+  dirPkg : "/home/rtrainning/Documents/Vallourec/RPackages/"  
>  dirLib : "/home/rtrainning/Documents/Vallourec/lib/"
```

Conectando ao Banco de Dados

- Carrega pacotes;
- Define diretórios de configuração e de dados;
- Acessa arquivo de configuração;
- Carrega arquivo de funções de acesso ao Banco de Dados;
- Conecta ao Banco de Dados.

```
> library("yaml")
> library("RJDBC")
>
> dirScripts <- "/home/rtrainning/RTrainning/R/"
> dirDados <- "/home/rtrainning/RTrainning/"
> options(force.init = TRUE, java.parameters = "-Xmx500m")
> configFile <- paste(dirScripts, "config.yml", sep="/");
> config <- yaml.load_file(configFile)
>
> source(paste0(dirScripts, "FuncoesAcessaDadosBD.R"))
> conexao <- RetornaConexaoBDOracle( config )
```

Pacote DBI

- `dbConnect`: Cria uma conexão com o DBMS.
- `dbGetQuery`: Envia SQL, recebe os resultados e limpa o *result set*.
- `dbWriteTable`: Cria *data frames* em tabelas do BD.
- `dbDisconnect`: Finaliza uma conexão.

Conectando e Desconectando do BD

- **dbConnect**

```
> RetornaConexaoBDOracle <- function (pConfig) {  
+   stringConexao <- pConfig$oracle$stringConexao  
+   arqJar       <- pConfig$oracle$arqJar  
+   drv <- JDBC("oracle.jdbc.driver.OracleDriver", arqJar, "")  
+   conn <- dbConnect(drv, stringConexao)  
+ }
```

- **dbDisconnect**

```
> FechaConexaoBDOracle <- function (pConexao) {  
+   dbDisconnect(pConexao)  
+ }
```

Executando consultas SQL

- Uma vez que a conexão com o Banco de Dados foi estabelecida, podemos executar comandos SQL para acesso aos dados.

```
> RetornaDadosZFORNO <- function (pConexao) {  
+   dadosForno <- dbGetQuery(pConexao,  
+                           paste("SELECT IDC_REGISTRO,  
+                                 IDC_BLOCO  
+                               FROM ZFORNO"))  
+   return(dadosForno)  
+ }
```

Agora é a sua vez

Tempo de parada

A partir da tabela ZPARADASPQF, calcular tempo de parada em minutos :

- Por turno.
- Por tipo de parada, ordenado por ordem descendente do tempo de parada.
- Por tipo de parada para determinado turno especificado.

Solução por Turno

```
> RetornaTempoParadaTurno <- function (pConexao) {  
+   consultaSQL <- paste("SELECT TURNO_OCORR_PARADA,  
+                         SUM(DURACAO_MINUTOS) TOTAL_MINUTOS  
+                         FROM ZPARADASPQF  
+                         GROUP BY TURNO_OCORR_PARADA")  
+   dadosTempoParadaTurno <- dbGetQuery(pConexao, consultaSQL )  
+   return(dadosTempoParadaTurno)  
+ }
```

Solução por Tipo de parada

```
> RetornaTempoTipoParada <- function (pConexao) {  
+   consultaSQL <- paste("SELECT TIPO_PARADA,  
+                         SUM(DURACAO_MINUTOS) TOTAL_MINUTOS  
+                     FROM ZPARADASPQF  
+                   GROUP BY TIPO_PARADA  
+                  ORDER BY TOTAL_MINUTOS DESC")  
+   dadosTempoTipoParada <- dbGetQuery(pConexao, consultaSQL)  
+   return(dadosTempoTipoParada)  
+ }
```

Solução tipo de parada para determinado turno.

```
> RetornaTempoTipoParadaTurno <- function (pConexao, pTurno) {  
+   if (!is.numeric(pTurno)) {  
+     stop("Turno tem que ser numérico!")  
+   }  
+   consultaSQL <- paste("SELECT TIPO_PARADA,  
+                         SUM(DURACAO_MINUTOS) TOTAL_MINUTOS  
+                         FROM ZPARADASPQF  
+                         WHERE TURNO_OCORR_PARADA = ", pTurno,  
+                               " GROUP BY TIPO_PARADA  
+                               ORDER BY TOTAL_MINUTOS DESC")  
+   dadosTempoTipoParadaTurno <- dbGetQuery(pConexao, consultaSQL)  
+   return(dadosTempoTipoParadaTurno)  
+ }
```

Acessando dados de ZFORNO

```
> dadosForno <- RetornaDadosZFORNO(conexao)
> dadosForno$COD_ORDEM_PRODUCAO
> df <- as.data.frame(cbind(dadosForno$PRS_FORNO,dadosForno$PRS_AR))
> colnames(df) <- c("PRS_FORNO","PRS_AR")
> write.table(df,file = paste0(dirDados,'dadosForno.csv'),
+               row.names = FALSE,
+               dec = ".",
+               sep = ";")
```

Manipulando data frames com SQL

Pacote sqldf

```
> ZPARADASPQF <- RetornaDadosZPARADASPQF(conexao)
> consultaSQL <- paste("SELECT TURNO_OCORR_PARADA,
+                         SUM(DURACAO_MINUTOS) TOTAL_MINUTOS
+                         FROM ZPARADASPQF
+                         GROUP BY TURNO_OCORR_PARADA")
> sqldf(consultaSQL)
> consultaSQL <- paste("SELECT TIPO_PARADA,
+                         SUM(DURACAO_MINUTOS) TOTAL_MINUTOS
+                         FROM ZPARADASPQF
+                         GROUP BY TIPO_PARADA
+                         ORDER BY TOTAL_MINUTOS DESC")
> sqldf(consultaSQL)
```

Pacote sqldf

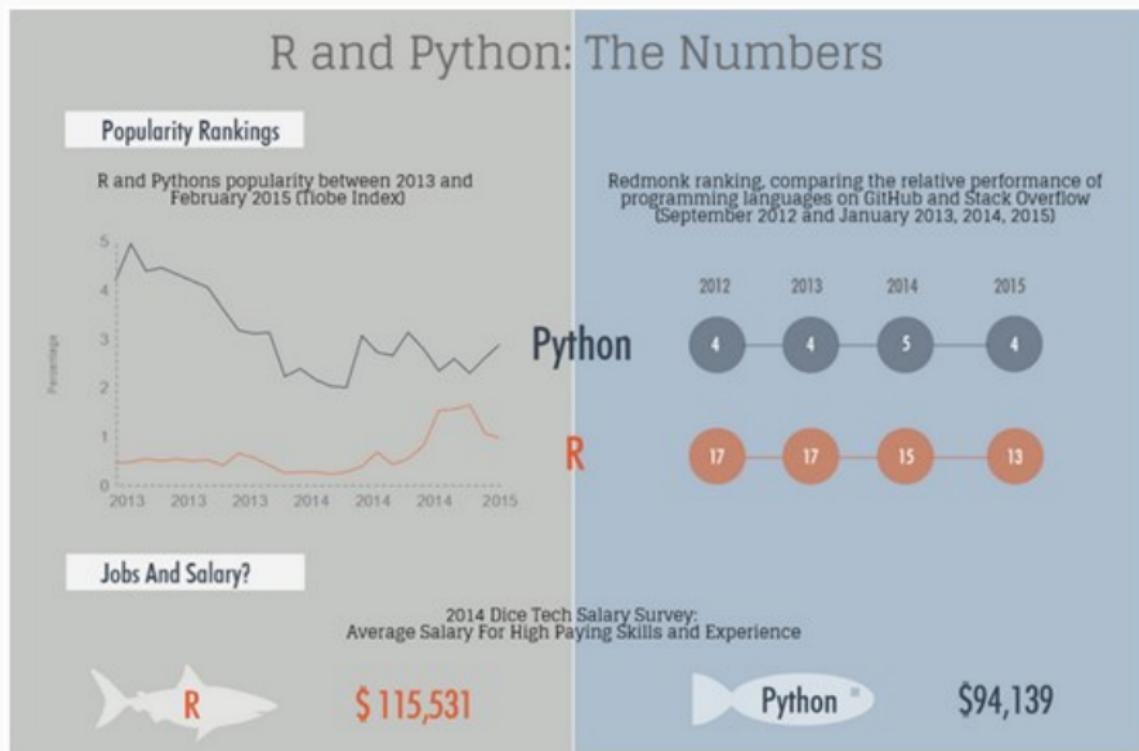
```
> ZPARADASPF <- RetornaDadosZPARADASPF(conexao)
> turno <- 1
> consultaSQL <- paste("SELECT TIPO_PARADA,
+                         SUM(DURACAO_MINUTOS) TOTAL_MINUTOS
+                         FROM ZPARADASPF
+                         WHERE TURNO_OCORR_PARADA = ", turno,
+                         " GROUP BY TIPO_PARADA
+                         ORDER BY TOTAL_MINUTOS DESC")
> sqldf(consultaSQL)
```

Agora é a sua vez

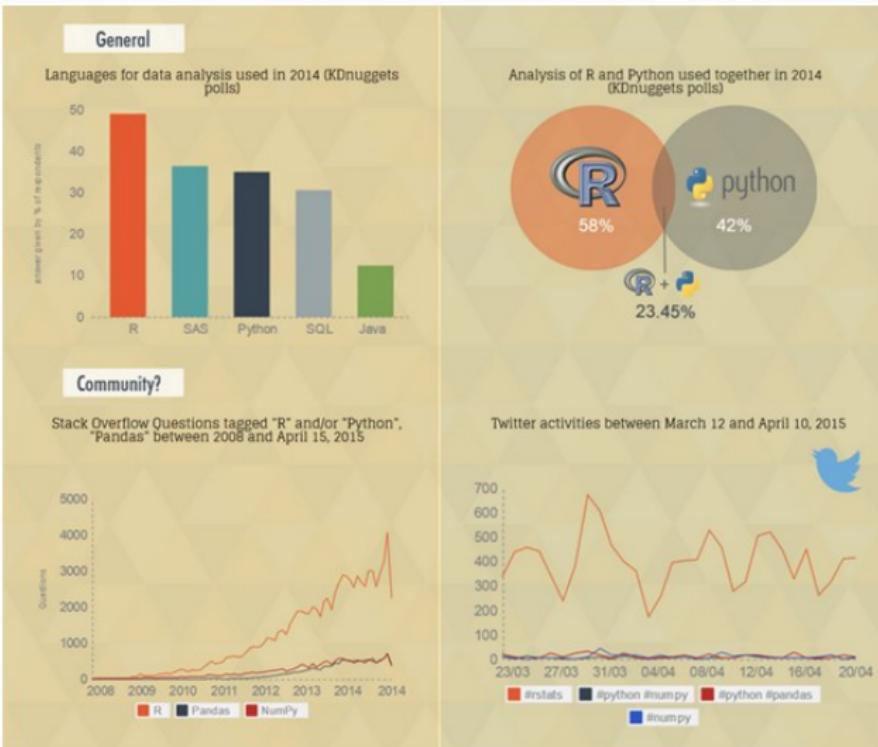
Integração R, Python e D3

R vs Python for data science : the winner is ...

- www.kdnuggets.com/2015/05/r-vs-python-data-science.html



R vs Python for data sciente : the winner is ...



R : Pros e Contras

R : Pros

- Uma figura diz mais que 1000 palavras
- Ecosistema
- Língua franca da ciência dos dados
- Muito mais pacotes para análises estatísticas
- Gráficos e *dashboards* iterativos

R : Contras

- R é lento! Mas existe muitos pacotes para melhorar a performance do R : **pqR**, **renjin**, **FastR**, **Riposte** e muitos outros, além de ser possível gerar *byte-code*.
- R tem uma curva de aprendizado íngreme!

Python : Pros

- IPython Notebook
- Linguagem de uso geral
- Melhor que o R nas tarefas de **Web scraping and crawling**
- Transações com Banco de Dados

Python : Contras

- Visualizações : Apesar de ter algumas bibliotecas interessantes, elas são normalmente mais complicadas de usar do que as bibliotecas do R.
- Python não oferece uma alternativa aos milhares de pacotes do R. Apesar do surgimento de novas bibliotecas, é pouco provável que os usuários de R o abandonem, até porque o R está em constante atualização por empresas e pela academia.

R vs Python

[www.mango-solutions.com/wp/2015/10/
integrating-python-and-r-into-a-data-analysis-pipeline-part-1](http://www.mango-solutions.com/wp/2015/10/integrating-python-and-r-into-a-data-analysis-pipeline-part-1)

Python tends to outperform R in such areas as:

- **Web scraping and crawling:** though *rvest* has simplified web scraping and crawling within R, Python's *beautifulsoup* and *Scrapy* are more mature and deliver more functionality.
- **Database connections:** though R has a large number of options for connecting to databases, Python's *sqlalchemy* offers this in a single package and is widely used in production environments.

Whereas R outperforms Python in such areas as:

- **Statistical analysis options:** though Python's combination of *Scipy*, *Pandas* and *statsmodels* offer a great set of statistical analysis tools, R is built specifically around statistical analysis applications and so provides a much larger collection of such tools.
- **Interactive graphics/dashboards:** *bokeh*, *plotly* and *intuitics* have all recently extended the use of Python graphics onto web browsers, but getting an example up and running using *shiny* and *shiny dashboard* in R is faster, and often requires less code.

Integração R / Python

Tipos de Integração R / Python

- Processos independentes

Tipos de Integração R / Python

- Processos independentes
- Chamando o Python através do R com o pacote rPython

Tipos de Integração R / Python

- Processos independentes
- Chamando o Python através do R com o pacote rPython
- Criação de subprocessos a partir da execução de scripts

Tipos de Integração R / Python

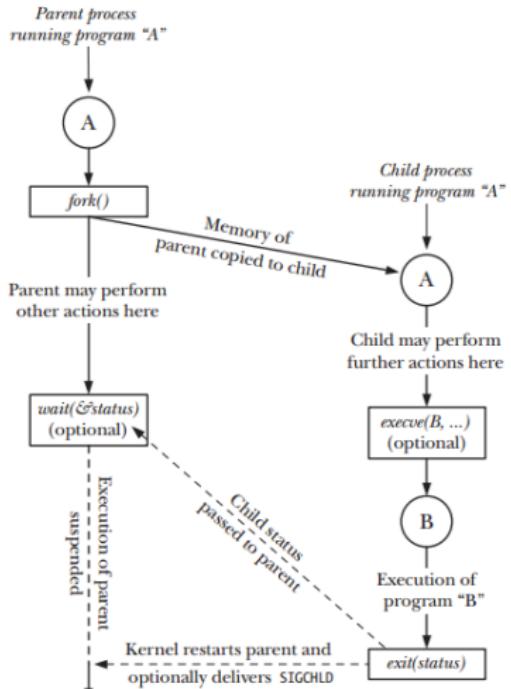
- Processos independentes
- Chamando o **Python** através do **R** com o pacote **rPython**
- Criação de subprocessos a partir da execução de **scripts**
- Chamando o **R** através do **Python** utilizando a biblioteca **pandas**

Processos independentes

- Através de arquivos
 - Python → R
 - R → Python
- Através de tabelas de Banco de Dados
 - Python → R

Integração R x Python : criando subprocessos

Criando subprocessos



Overview of the use of `fork()`, `exit()`, `wait()`, and `execv()`

IDE e Bibliotecas

- IDE Python - [Spyder](#).
- Bibliotecas : [subprocess](#) e [Pandas.rpy2](#)

Web Framework : Flask

- Micro web framework desenvolvida em [Python](#).
- Baseado nas bibliotecas :
 - [Werkzeug \(WSGI : Web Server Gateway Interface\)](#) e
 - [Jinja2](#) - biblioteca que segue o modelo de arquitetura [MVT](#) (Model-View-Template).

Scripts Python

- run.py
- init.py
- views.py
- models.py
- PythonR.py

Scripts HTML

- index.html
- base.html

Script principal : views.py

- Carrega pacote `subprocess`;
- Monta variável com o comando a ser executado;
- Executa `script` como subprocesso;
- Prepara os dados para exibição na página Web.

```
>     'import subprocess'  
>  
>     path2script = '~/RTrainning/R/RegressaoSimples.R'  
>     cmd = '[command, path2script]'  
>     resultRReg = subprocess.check_output(cmd,  
+                                         universal_newlines=True)  
>     resultRRegDetalhe = resultRReg.split(os.linesep)
```

- `universal newlines` = Maneira de interpretar textos de forma a reconhecer como fim de linha os caracteres: “barra n” ([Unix](#)), “barra r barra n” ([Windows](#)), e “barra r” ([Macintosh](#)).
- `universal newlines=False` : Saída é aceita como `byte` e não como `string Unicode`.

Executando o script... juntos

Agora é a sua vez

Criar scripts para exibir resultado na Web

1. **Regressão Linear:** Ajustar modelo e retornar resultado para exibição na Web
2. **Acessar Dados de Parada:** Executar consulta na tabela ZPARADASPQF retornando 100 registros e as colunas TIPO PARADA e DURACAO MINUTOS. Listar os tipos de paradas e exibir histograma dos minutos.

Chamando o R através do Python utilizando a biblioteca pandas

Conclusão : R ou Python para ciência dos dados ?

Tratamento de Erro

Tipos de tratamento de erros no R

1. `try()`: permite continuar a execução mesmo quando ocorre erro.
2. `tryCatch()`: permite especificar uma função que controla o que deve acontecer quando uma condição é sinalizada.
3. `withCallingHandlers()`: é uma variante do `tryCatch()` que executa a função em um contexto diferente. Raramente utilizado, mas importante saber da sua existência.

Exemplo de trycatch()

```
> RetornaDadosZForno <- function (pConexao) {  
+   dadosForno = tryCatch({  
+     dadosForno <- dbGetQuery(pConexao, paste("SELECT IDC_REGISTRO  
+                               FROM ZFORNO  
+                               WHERE ROWNUM <= 100"))  
+  
+     J("java.lang.System")$gc()  
+  
+     return(dadosForno)  
+   }, warning = function(w) {  
+     print(w); gWMsg <- paste("Warning Acesso Dados Forno : ",w); stop(gWMsg)  
+   }, error = function(e) {  
+     print(e); gEMsg <- paste("Erro Acesso Dados Forno : ",e); stop(gEMsg)  
+   }, finally = {  
+   })  
+   return(dadosForno)  
+ }
```

* Argumento **finally** do **tryCatch()** : Especifica um bloco de código (não é uma função) para ser executado independentemente de ter ocorrido alguma condição ou não. Útil para liberação de objetos, ex.: deleção de arquivos, fechamento de conexões, etc).

Debug
