

Aula 1: Introdução ao R

Ivette Luna

19 de março de 2019

Contents

Sobre o R e o RStudio	3
Fontes de consulta e aprendizado	3
E como funciona?	3
Operações matemáticas (R como calculadora)	4
Criando variáveis no nosso primeiro script	5
Tipos básicos de dados	5
Objetos	6
Funções	6
Vetores	7
Subsetting	8
Operações com vetores	8
Outras funções que lidam com vetores	9
Matrizes	11
Algumas funções interessantes a usar sobre matrizes:	11
Criando matrizes especiais	12
Operações com matrizes	12
Combinação de matrizes	13
Geração de números aleatórios	14
Listas	14
Comparações lógicas e condições	15
Data frames	16
Criando do zero	16
Subsetting dataframes	17
Exportar dados	18
Importar dados	18
Análise exploratória de dados	19
Criando variáveis	20
Duas variáveis categóricas - tabulação cruzada	21
Duas variáveis contínuas - correlação	21
Análise gráfica	21
Para salvar a figura	25
Histogramas, distribuições e box-plots	25

Sobre o R e o RStudio

O R é ambiente para Data Science: análise estatística e gráfica. É uma plataforma que nos permite laborar as nossas próprias **rotinas** e utilizar rotinas de terceiros (**pacotes**) disponíveis para uso, e que nos permite realizar uma série de atividades de análise empírica:

1. Analisar bases de dados
2. Realizar análises estatísticas
3. Ajustar modelos dos mais diversos tipos (matemáticos, estatísticos, econométricos, redes neurais etc)
4. Realizar simulações
5. Elaborar documentos (apresentações, páginas, relatórios etc)

Obviamente, não é a única alternativa principalmente quando os nossos objetivos são outros: Python, Java, C++ etc

E o RStudio?

- É uma interface de visualização e uso do R a partir de uma tela *mais bonita*
- Facilita muito o uso do R
- É uma das interfaces mais usadas (e a mais eficiente ao meu parecer): Tinn-R

Fontes de consulta e aprendizado

As que destaco:

- R-bloggers: <https://www.r-bloggers.com/how-to-learn-r-2/>
- Stackoverflow: <https://stackoverflow.com/documentation/r/topics>
- Cookbook-R: <http://www.cookbook-r.com/>
- Para cursos: Datacamp

E como funciona?

Para descobrir, veja a composição da tela no RStudio. Temos 4 seções. Este layout pode ser alterado de acordo ao seu gosto. Para isso, ir a **Tools>Global Options>Pane Layout**.

Podemos ver:

1. A tela de **Console**: ou tela de comando, onde podemos executar diretamente um comando curto (para outra coisa não é muito prático);
2. Temos a tela de **Source**: ou tela de programação, é um editor onde abriremos um arquivo especial chamado *script*, para nele, guardar as nossas implementações ou rotinas. Salvando as nossas rotinas gora, podemos reutilizá-las e editá-las posteriormente;
3. Ainda, temos as telas de **Help, Environment, History, Files, Plots** etc.

Digite no Console a soma de dois números:

```
2 + 5
```

```
## [1] 7
```

e aperte o botão **Enter**.

Operações matemáticas (R como calculadora)

- Soma: +
- Subtração: -
- Multiplicação: *
- Divisão: /
- Potência de dois números: ^

Podemos digitar as seguintes linhas na tela de edição do R:

```
2 + 5  
## [1] 7  
  
2 - 5  
## [1] -3  
  
2*5  
## [1] 10  
  
2/5  
## [1] 0.4  
  
2^5  
## [1] 32
```

- Selecione as linhas que deseja executar e aperte **Ctrl + Enter**.
- Se quiser executar apenas uma linha, basta posicionar o cursor na linha e apertar **Ctrl + Enter**.

Observações:

- Se o R fosse apenas uma calculadora, não precisaríamos de aulas para o seu uso e qualquer um poderia pesquisas e estudos empíricos em geral, com a qualidade que o R permite;
- Logo, o exemplo anterior não justifica o uso do R;
- **Rotina (Script):** um conjunto de comandos e sequências de código que salvamos em um arquivo de extensão .R via a tela de edição;
- **Pacote:** conjuntos de rotinas desenvolvidos por outros usuários e disponível para a comunidade (**base**, **ggplot**, **dplyr**)
- Uma lista de pacotes: <https://support.rstudio.com/hc/en-us/articles/201057987-Quick-list-of-useful-R-packages>
- O R é *Case sensitive!!!!*
- Se quer limpar a sua tela de comando, posicione o cursor no Coonsole e aperte **Ctrl + L**.
- Nós não criaremos pacotes. Criaremos scripts.
- Para acessar a documentação de uma função, usamos o operador ?. Por exemplo, o comando **?mean**, nosm ostrará na tela do **help** no RStudio, a descrição da função com nome **mean**, começando pelo pacote que o contém, o seu cabeçalho e a descrição dos argumentos (inputs) e saída (o que nos dá como resultado).

```
?sqrt  
help(median)
```

Criando variáveis no nosso primeiro script

Assim como na matemática criamos variáveis que para nós representam uma série, uma incógnita, uma variável econômica, um modelo, etc. Podemos criar variáveis e trabalhar com elas:

```
# O hashtag logo no começo de uma linha indica ao R que esta linha é um comentário  
# Logo, podemos usar o hashtag para documentar e descrever os nossos comandos
```

```
# criando variáveis
```

```
a <- 2  
b <- 5
```

```
# para somar  
a + b
```

```
## [1] 7
```

```
# ou ainda podemos criar uma terceira variável que armazene o resultado
```

```
soma <- a + b
```

```
subtracao = a - b
```

```
# para eliminar uma variável  
rm( soma )
```

```
# para eliminar todas as variáveis da memória  
rm( list = ls(all.names = TRUE) )
```

- Veja que o símbolo `<-` ou `=` no nosso caso são equivalentes e significam **atribuição** do resultado que será processado à sua direita e será armazenado na variável cujo nome aparece à esquerda, exista ou não.

```
a <- 2  
b <- 5
```

```
soma <- a + b
```

```
soma
```

```
## [1] 7
```

```
soma <- a + 10
```

```
soma
```

```
## [1] 12
```

Tipos básicos de dados

- **Numerics**: números reais, inclui os decimais como 3.7
 - Note que o separador de decimais no R é o ponto (.)
- **Integers**: números naturais como 0, 9. Indicadores de posição são sempre do tipo **Integer**. Integers também são numerics
- **Logical**: variáveis binárias ou booleanas (TRUE or FALSE)

- **Characters**: quando os valores não são números, são textos (ou strings)
- **Factors**: valores associados a variáveis categóricas (estratos, códigos CPF, CNPJ etc).

Por exemplo:

```
# as aspas indicam que o que está dentro delas é um string
nome_cidade <- "Piracicaba"

nome_cidade

## [1] "Piracicaba"
com_emploi <- TRUE

# para ver o tipo de dado, usamos o comando 'class'
class( nome_cidade )

## [1] "character"
class( com_emploi )

## [1] "logical"
```

Objetos

Um objeto é uma entidade construída com os seu respectivo nome, elementos e atributos. Os tipos mais gerais:

- **Vetores**: sequência de valores numéricos ou de caracteres;
- **Matrizes**: coleção de vetores em linhas e colunas, todos os vetores devem ser do mesmo tipo;
- **Dataframe**: coleção de objetos, todos do mesmo tipo e estrutura (microdados);
- **Listas**: conjunto de vetores, data.frames ou de matrizes;
- **Funções**: rotinas criadas para fazer algum cálculo ou processamento.

Funções

Uma função no R segue a mesma lógica que as funções no Cálculo, porém são mais flexíveis já que dependendo a sua natureza, estas podem lidar com os diversos tipos de dados e objetos possíveis na plataforma. As funções no R são comandos que possuem argumentos de entrada (um ou mais) e produzem uma saída de acordo com o que se programou. Há funções já prontas agrupadas em *pacotes*, porém na maioria das vezes há a necessidade de elaborar as nossas próprias funções.

A seguir alguns exemplos de funções base no R:

```
x1 = -3

abs( x1 )

log( -x1 )

exp( x1 )

print( x1 )
```

Vetores

Um vetor é um conjunto de elementos **do mesmo tipo** agrupados em um único objeto. No R, criamos um vetor com a função (ou comando) `c()` de *Combinar*.

```
x2 = c(3, 2, 1, -6, -2, 0)

classes <- c("A", "B", "C")

# algumas funções aceitam diversos tipos de dados e podem ter mais de um argumento
exp(x2)

## [1] 20.085536923 7.389056099 2.718281828 0.002478752 0.135335283
## [6] 1.000000000
round(x2, 2)

## [1] 3 2 1 -6 -2 0
print(classes)

## [1] "A" "B" "C"
# criando um dataframe

id = c(1, 2, 3, 4)

municipio = c(2345, 4333, 9543, 2001)

nome = c("Maria", "Joao", "Pedro", "Mateo")

sexo = c("F", "M", "M", "M")

renda_med = c(1.5, 2.3, 4, 2.9)

base_dados = data.frame( id, municipio, nome, sexo, renda_med )

base_dados

##   id municipio nome sexo renda_med
## 1  1      2345 Maria     F     1.5
## 2  2      4333 Joao      M     2.3
## 3  3      9543 Pedro     M     4.0
## 4  4      2001 Mateo     M     2.9

names(base_dados)

## [1] "id"          "municipio"    "nome"        "sexo"        "renda_med"

str(base_dados)

## 'data.frame': 4 obs. of 5 variables:
## $ id       : num  1 2 3 4
## $ municipio: num  2345 4333 9543 2001
## $ nome     : Factor w/ 4 levels "Joao","Maria",...: 2 1 4 3
## $ sexo     : Factor w/ 2 levels "F","M": 1 2 2 2
## $ renda_med: num  1.5 2.3 4 2.9
```

```
dim(base_dados)
```

```
## [1] 4 5
```

Subsetting

Na maioria das aplicações, precisaremos obter subconjuntos de valores. Para isso, usamos o operador [] e um índice numérico que nos permite informar ao R as posições dos valores que queremos extrair do vetor.

```
nome[ 3 ]
```

```
nome[ c(2,4) ]
```

```
nome[ -c(1,2) ]
```

- No caso, o sinal negativo antes do índice indica ao R que nós **não** queremos os valores nas posições indicadas.

Operações com vetores

O vetor no sentido matemático o conhecemos. No R o que sabemos de vetores se aplica para realizar operações matemáticas. Obvio que para isso, tais vetores devem ser numéricos e atender aos requisitos de tamanho.

Operações vetoriais

```
# Operações básicas (matemática)
```

```
v1 <- c(5, 8, 9, 6.25, 7, 7)
```

```
v2 <- c(7, 5, 10, 3, 3, 4)
```

```
soma_vetores <- v1 + v2
```

```
soma_vetores
```

```
## [1] 12.00 13.00 19.00 9.25 10.00 11.00
```

```
# para ver o tamanho do vetor  
length(v1)
```

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

```
# produto por um escalar - calculando a média  
vetor <- soma_vetores*0.5
```

```
vetor
```

```
## [1] 6.000 6.500 9.500 4.625 5.000 5.500
```

```
# produto escalar
```

```
prod_escalar = sum(v1*v2)
```

```
prod_escalar
```

```
## [1] 232.75
```

```

# a norma de um vetor

norma_v1 = sqrt( sum(v1*v1) )

norma_v1

## [1] 17.5232

# total de alunos da turma
n <- length(soma_vetores)

media_soma <- sum( soma_vetores )/n

media_soma

## [1] 12.375

# ou
mean(soma_vetores)

## [1] 12.375

```

Operações não vetoriais

```

medias <- round( media_soma, 1 )

medias

## [1] 12.4

logaritmo <- log( media_soma )

raiz <- sqrt( media_soma )

# potencia
potencia <- media_soma^2

# criar um vetor com n elementos repetidos
v <- rep(2, n)

# e dividir dois vetores, elemento a elemento
divisao <- soma_vetores/v

produto <- soma_vetores*potencia

```

Outras funções que lidam com vetores

Além da função `rep()`, temos também a função `seq()` e o operador `:`.

```

pares <- seq(from=2, to=20, by=2) # ou seq(2,20,2)

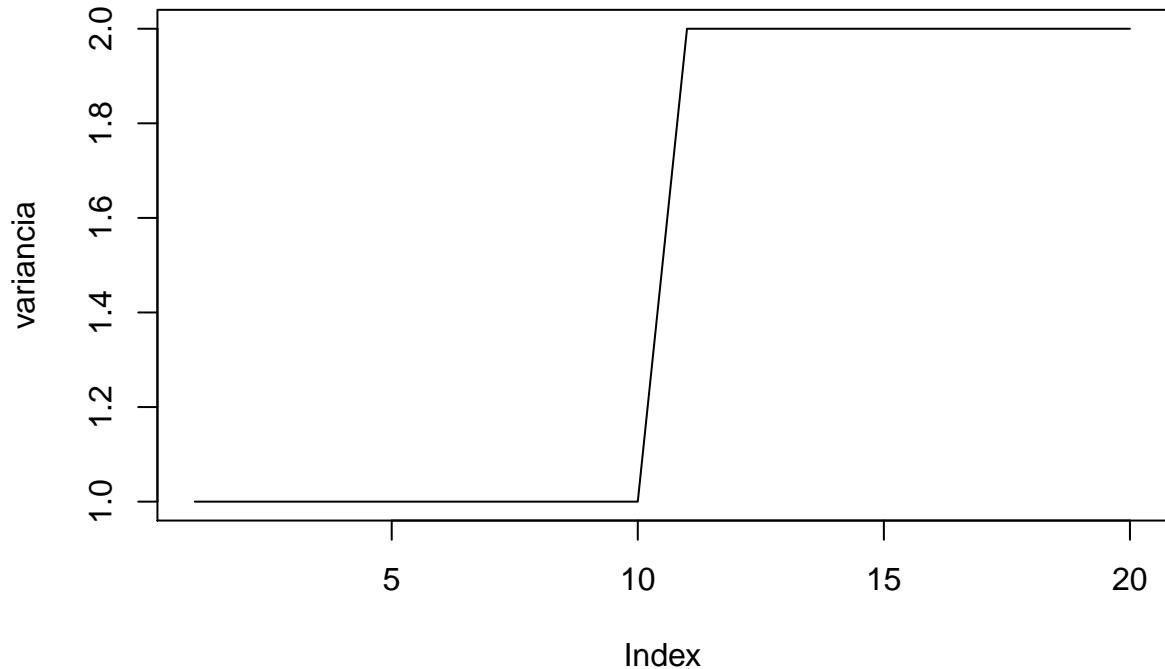
anos <- 1978:2018

# podemos ainda, concatenar ou combinar dois vetores
variancia <- c( rep(1,10), rep(2,10) )

variancia

```

```
## [1] 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2  
plot(variancia, type="l")
```



```
# ou criar um vetor vazio  
vazio <- c()  
  
# ou ordenar os valores de forma decrescente  
vetor_ordenado <- sort(soma_vetores, decreasing = FALSE)
```

Matrizes

Sabemos também que uma matriz é um objeto *bidimensional* e há a necessidade de respeitar as *regras* que estes objetos exigem quanto à sua dimensão e operações matemáticas possíveis.

```
# Para criar uma matriz, usamos o comando matrix() sobre algum vetor contendo os seus
# elementos. Suponha que queremos construir a matriz a seguir
# 2 3 -1
# 0 4 1
# 1 1 4

# criamos o vetor
elementos_matriz <- c(2, 0, 1, 3, 4, 1, -1, 1, 4)

matriz <- matrix( elementos_matriz, nrow=3, ncol=3 )
print(matriz)

##      [,1] [,2] [,3]
## [1,]    2    3   -1
## [2,]    0    4    1
## [3,]    1    1    4

# se precisamos de acessar a um dos elementos da matriz precisamos informar os valores
# associados aos índices e e j vistos na aula teórica

matriz[ 1, 3]

## [1] -1

# se queremos extrair todos os elementos da segunda coluna:
matriz[ , 2]

## [1] 3 4 1
# ou da segunda linha
matriz[ 2, ]

## [1] 0 4 1

submatriz <- matriz[ 1:2, c(1,3)  ]

submatriz

##      [,1] [,2]
## [1,]    2   -1
## [2,]    0    1
```

Note como o R preenche as matrizes (por colunas).

Algumas funções interessantes a usar sobre matrizes:

```
# para obter a dimensão da matriz
dim( matriz )

## [1] 3 3

nrow( matriz )

## [1] 3
```

```

ncol( matriz )

## [1] 3

diag( matriz )

## [1] 2 4 4

det( matriz ) # determinante

## [1] 37

solve( matriz ) # inversa

## [,1]      [,2]      [,3]
## [1,]  0.40540541 -0.35135135  0.18918919
## [2,]  0.02702703  0.24324324 -0.05405405
## [3,] -0.10810811  0.02702703  0.21621622

```

Criando matrizes especiais

```

matriz_unitaria <- matrix( 1, nrow=3, ncol=3 )
print(matriz_unitaria)

## [,1] [,2] [,3]
## [1,] 1 1 1
## [2,] 1 1 1
## [3,] 1 1 1

matriz_diagonal <- diag( c(1, 2, 3) )
print(matriz_diagonal)

## [,1] [,2] [,3]
## [1,] 1 0 0
## [2,] 0 2 0
## [3,] 0 0 3

matriz_identidade <- diag( 1 , 3 )
print(matriz_identidade)

## [,1] [,2] [,3]
## [1,] 1 0 0
## [2,] 0 1 0
## [3,] 0 0 1

```

Operações com matrizes

A adição (+), subtração (-), produto matricial (%*%), transposta (t) e produto elemento a elemento (*) são possíveis desde que se atenda aos requisitos nas dimensões das matrizes envolvidas. Temos também a função `crossprod()` que nos dá o produto matricial $A^T \cdot B$.

```

soma_matrizes <- matriz_unitaria + matriz_diagonal
print(soma_matrizes)

## [,1] [,2] [,3]
## [1,] 2 1 1
## [2,] 1 3 1
## [3,] 1 1 4

```

```

subtracao_matrizes <- matriz_unitaria - matriz_diagonal
print(subtracao_matrizes)

##      [,1] [,2] [,3]
## [1,]    0    1    1
## [2,]    1   -1    1
## [3,]    1    1   -2

prod_por_escalar <- soma_matrizes*4
print(prod_por_escalar)

##      [,1] [,2] [,3]
## [1,]    8    4    4
## [2,]    4   12    4
## [3,]    4    4   16

transposta <- t( soma_matrizes )
print(transposta)

##      [,1] [,2] [,3]
## [1,]    2    1    1
## [2,]    1    3    1
## [3,]    1    1    4

# O produto matricial tem um operador especial
produto_matrizes <- matriz %*% solve(matriz)
print(produto_matrizes)

##      [,1]      [,2]      [,3]
## [1,] 1.000000e+00 -5.551115e-17    0
## [2,] 0.000000e+00  1.000000e+00    0
## [3,] -5.551115e-17 -5.551115e-17    1

crossprod(matriz, solve(matriz))

##      [,1]      [,2]      [,3]
## [1,]  0.7027027 -0.67567568  0.5945946
## [2,]  1.2162162 -0.05405405  0.5675676
## [3,] -0.8108108  0.70270270  0.6216216

apply( matriz, 2, mean) # o que vc acha que o apply faz?

## [1] 1.000000 2.666667 1.333333

```

Combinação de matrizes

Para combinar matrizes, desde que respeitando as dimensões, podemos usar as funções `cbind()` e `rbind()`.

```
por_colunas <- cbind( matriz, matriz_diagonal )
```

```
por_colunas
```

```

##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    2    3   -1    1    0    0
## [2,]    0    4    1    0    2    0
## [3,]    1    1    4    0    0    3

```

```

por_linhas <- rbind( matriz, matriz_diagonal )

por_linhas

##      [,1] [,2] [,3]
## [1,]    2    3   -1
## [2,]    0    4    1
## [3,]    1    1    4
## [4,]    1    0    0
## [5,]    0    2    0
## [6,]    0    0    3

```

Geração de números aleatórios

A geração apropriada de números aleatórios é fundamental em ambientes de programação e em estatística e econometria (ex.: simulações de Monte Carlo). Veja por exemplo, as opções possíveis no R usando o comando `?RNG`.

Para garantir a replicabilidade, precisamos especificar a *semente* de geração. Para isso, usamos a função `set.seed()`. Logo, podemos por exemplo usar o comando `rnorm` para gerar números aleatórios com distribuição normal.

```

set.seed(1020)

rnorm(2)

## [1] -2.038686 -2.028341

rnorm(2)

## [1] -2.535613 -0.641793

set.seed(1020)

rnorm(2)

## [1] -2.038686 -2.028341

rnorm(2, mean=4, sd=2)

## [1] -1.071226  2.716414

sample(1:5)

## [1] 2 1 3 5 4

sample(1:5, replace=TRUE)

## [1] 2 1 5 2 4

```

Listas

São vetores genéricos onde cada elemento pode ser qualquer tipo de objeto: um vetor, um dataframe, um escalar, uma matriz etc (muito útil para análises de regressão). Para criar uma lista, usamos a função `list()`.

```

dados amostra <- list( amostra = rnorm(5),
                      distribuicao = "normal",
                      parametros = list(media = 0, dp = 1))

```

```

dados_amostra

## $amostra
## [1] -0.2332297 -0.3498551 -0.2915161  0.9582354 -0.5568181
##
## $distribuicao
## [1] "normal"
##
## $parametros
## $parametros$media
## [1] 0
##
## $parametros$dp
## [1] 1

```

Para acessar um dos elementos da lista, usaos o operador `[[]]` ou o `$`. Note-se que em qualquer dos casos, podemos acessar simultâneamente apenas um objeto da lista por vez.

```

dados_amostra$amostra

## [1] -0.2332297 -0.3498551 -0.2915161  0.9582354 -0.5568181
dados_amostra[[ 1 ]]

## [1] -0.2332297 -0.3498551 -0.2915161  0.9582354 -0.5568181
dados_amostra[[ "distribuicao" ]]

## [1] "normal"
dados_amostra[["parametros"]]$media

## [1] 0

```

Comparações lógicas e condições

Em toda linguagem de programação, utilizamos operadores lógicos e de comparação:

- Igualdade: `==`
- Não igual: `!=`
- Menor: `<`
- Maior: `>`
- Menor ou igual: `<=`
- Maior ou igual: `>=`

- E: `&`
- Ou: `|`

que podemos utilizar em estruturas de controle (`if`, `for`, `while`, etc) ou para selecionar subconjuntos a partir de uma determinada condição.

Por exemplo:

```

tamanho_firmas <- sample(30:600, 10, replace=TRUE)

tamanho_firmas < 99

## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

```

```

firmas_medias <- tamanho_firmas > 99 & tamanho_firmas < 499

firmas_medias

## [1] FALSE TRUE TRUE FALSE TRUE FALSE FALSE TRUE TRUE TRUE
tamanho_firmas[ firmas_medias ]

## [1] 472 435 458 212 270 126

```

Data frames

Um dataframe é uma matriz de dados. É um arranjo composto por uma lista (pilha) de vetores, todos com o mesmo tamanho.

Assim, cada linha representa a um elemento do conjunto (uma observação) e cada coluna representa uma variável (uma característica, um atributo).

Criando do zero

Sejam as variáveis, `cpf`, `idade`, `salario`.

```

cpf <- c( "0123", "8324", "2233", "9100" )

idade <- c( 23, 18, 47, 51 )

salario <- c( 2.5, 1, 4.5, 6 )

# para criar o dataframe
rais <- data.frame(cpf, idade, salario)

# para ver a estrutura da base
str(rais)

## 'data.frame':    4 obs. of  3 variables:
## $ cpf      : Factor w/ 4 levels "0123","2233",..: 1 3 2 4
## $ idade    : num  23 18 47 51
## $ salario: num  2.5 1 4.5 6
# para fazer um levantamento estatistico preliminar
summary(rais)

##      cpf          idade         salario
## 0123:1   Min.   :18.00   Min.   :1.000
## 2233:1   1st Qu.:21.75   1st Qu.:2.125
## 8324:1   Median :35.00   Median :3.500
## 9100:1   Mean    :34.75   Mean    :3.500
##             3rd Qu.:48.00   3rd Qu.:4.875
##                 Max.   :51.00   Max.   :6.000

```

Para extrair uma variável, podemos:

1. Usar o operador `$` seguido pelo nome da variável (ou da coluna)
2. Usar a notação matricial e indicar o nome da coluna ou
3. Usa a notação matricial e usar o número associado à coluna/variável que nos interessa.

```

# para extrair uma variável
idade_empregados <- rais$idade
print(idade_empregados)

## [1] 23 18 47 51

salario_empregados <- rais[ , "salario" ]
print(salario_empregados)

## [1] 2.5 1.0 4.5 6.0

cpf_empregados <- rais[ , 1]
print(cpf_empregados)

## [1] 0123 8324 2233 9100
## Levels: 0123 2233 8324 9100

```

Observe que o último vetor é do tipo **factor**:

```

class(cpf_empregados)

## [1] "factor"

```

Isto porque quando se cria um dataframe, se nada é dito a respeito do processamento de **strings** (textos), estes serão processados como **factors**, um tipo de dado no R, usado principalmente para variáveis categóricas (sexo, estrato de renda, tipo de contrato laboral, setor de atuação etc) ou para informações associadas a algum tipo de identificador (CPF, CNPJ, PIS etc).

Deve-se tomar cuidado com este tipo de dado pois no geral estes ocupam mais memória. Com bases pequenas isso não é um problema, porém, com bases grandes (*big data*), problemas de processamento da base podem ser intensificados.

Subsetting dataframes

Para manipular bases de dados no R, o pacote mais usado atualmente é o **dplyr**.

O que você acha que a próxima linha de comando faz?

```

library(dplyr)

medias <- rais %>% select( idade, salario) %>%
  summarize(media_idade = mean(idade),
            media_salario = mean(salario),
            n_obs = n()
            )

class(medias)

print(medias)

```

E esta linha?

```

grupo_rais <- rais %>% filter( idade > 20 & idade < 50 )

print(grupo_rais)

```

Exportar dados

No geral, quando analisamos uma base de dados ou ajustamos um modelo, queremos salvar os resultados fora do R, de maneira que estes possam ser utilizados em algum relatório, projeto ou artigo. Estes resultados costumam ser armazenados em tabelas que devem ser exportadas em um arquivo que popularmente é do tipo `csv` ou `txt`, separado por tabulação (seja com `,`, `;` ou algum outro separador).

Para isso, fazemos uso do pacote `foreign`.

```
library(foreign)

# para salvar uma tabela em formato csv
write.csv( rais, "rais.csv" )

# em txt
write.table(rais, "rais.txt")
```

Onde estão os arquivos gerados?

Querendo descobrir onde os nossos arquivos foram salvos, usamos a função `getwd()`.

Para definir a pasta de trabalho, usamos a função `setwd()`, tendo como argumento o `string` contendo o endereço do diretório ou pasta. Por exemplo:

```
getwd() # identificamos o diretório atual de trabalho

pasta_resultados = "D:/iluna 2018/aulas 2018/graduacao/Metodos IV"

setwd( pasta_resultados )

getwd() # verificamos se houve alteração
```

- Note que as barras `\` que vem quando você copia o endereço do *Explorer* devem ser trocadas pela barra invertida `/`.

Importar dados

De forma geral as bases já existem fora do R: dados de comércio internacional, a RAIS, o Censo, a PNAD, a PIA, PINTEC etc. Logo, para poder fazer uso dessas informações precisamos carregar tais bases na memória do R.

Por exemplo, podemos analisar os microdados contidos no arquivo `italian_dfs.txt`. Carregue o arquivo e analise a sua estrutura e conteúdo. Para isso, podemos usar a função `read.table()` do pacote `foreign`:

```
industry_df <- read.table("industry_df.txt", sep = " ", header = TRUE, stringsAsFactors = FALSE)

head(industry_df)

##   SETOR    PO      VA VENDAS
## 1     17 6256 326795 815997
## 2     17  749 32456 117211
## 3     17 1486 57863 172387
## 4     17 1083 58655 202492
## 5     17  892 44033 112413
## 6     17 1004 54624 141330

names(industry_df)

## [1] "SETOR"    "PO"        "VA"        "VENDAS"
```

Análise exploratória de dados

Antes de realizar qualquer ajuste, precisamos sempre conhecer a nossa base, explorar os dados, realizar o processo de limpeza e deixar a base pronta para o seu uso.

Podemos começar avaliando o tamanho da base e com uma análise estatística geral:

```
dim(industry_df) # num de observações e num de variáveis

## [1] 44380      4

str(industry_df) # analisa a estrutura

## 'data.frame': 44380 obs. of 4 variables:
##   $ SETOR : int 17 17 17 17 17 17 17 17 17 17 ...
##   $ PO    : int 6256 749 1486 1083 892 1004 861 580 NA 500 ...
##   $ VA    : int 326795 32456 57863 58655 44033 54624 37248 39885 NA 20081 ...
##   $ VENDAS: int 815997 117211 172387 202492 112413 141330 112320 99332 NA 48492 ...

head(industry_df) # verifica se a base foi importada corretamente

##   SETOR   PO     VA VENDAS
## 1    17 6256 326795 815997
## 2    17  749  32456 117211
## 3    17 1486  57863 172387
## 4    17 1083  58655 202492
## 5    17  892  44033 112413
## 6    17 1004  54624 141330

summary(industry_df) # estatística descritiva geral

##      SETOR          PO            VA        VENDAS      
## Min.   :17.0   Min.   : 14.0   Min.   :    0   Min.   :    0  
## 1st Qu.:17.0   1st Qu.: 29.0   1st Qu.: 1615  1st Qu.: 1763 
## Median :17.0   Median : 46.0   Median : 3205  Median : 7982 
## Mean   :19.4   Mean   :122.6   Mean   :11095  Mean   :31624 
## 3rd Qu.:24.0   3rd Qu.: 94.0   3rd Qu.: 7431  3rd Qu.:22283 
## Max.   :24.0   Max.   :15553.0  Max.   :2900936 Max.   :7698624 
## NA's    :21683  NA's   :21683  NA's   :21683  NA's   :21683
```

Ao notar a existência de *missings*, devemos tomar algumas decisões. Podemos começar este processo de *cleaning*, eliminando os microdados com NA's.

```
indices_dados_completos <- complete.cases(industry_df)

complete_data <- industry_df[ indices_dados_completos , ]

summary( complete_data )

##      SETOR          PO            VA        VENDAS      
## Min.   :17.00   Min.   : 14.0   Min.   :    0   Min.   :    0  
## 1st Qu.:17.00   1st Qu.: 29.0   1st Qu.: 1615  1st Qu.: 1763 
## Median :17.00   Median : 46.0   Median : 3205  Median : 7982 
## Mean   :19.23   Mean   :122.6   Mean   :11095  Mean   :31624 
## 3rd Qu.:24.00   3rd Qu.: 94.0   3rd Qu.: 7431  3rd Qu.:22283 
## Max.   :24.00   Max.   :15553.0  Max.   :2900936 Max.   :7698624 

nrow(complete_data)
```

```
## [1] 22697
```

Vemos que o tamanho da amostra foi reduzido.

Agora, dado que a variável SETOR é um tipo de ID, podemos transformá-la em uma variável do tipo **factor**:

```
complete_data$SETOR <- as.factor( complete_data$SETOR )
```

```
str(complete_data)
```

```
## 'data.frame': 22697 obs. of 4 variables:  
## $ SETOR : Factor w/ 2 levels "17","24": 1 1 1 1 1 1 1 1 1 ...  
## $ PO    : int 6256 749 1486 1083 892 1004 861 580 500 352 ...  
## $ VA    : int 326795 32456 57863 58655 44033 54624 37248 39885 20081 12714 ...  
## $ VENDAS: int 815997 117211 172387 202492 112413 141330 112320 99332 48492 64766 ...
```

Quando temos uma variável categórica (discreta, fator), não faz sentido calcular médias ou variâncias. Mas podemos analisar a frequência:

```
summary(complete_data$SETOR )
```

```
##      17     24  
## 15478   7219
```

Também podemos analisar as estatísticas gerais por setor:

```
media_bysector <- complete_data %>% group_by( SETOR ) %>%  
  
  summarise_at( vars( PO, VA, VENDAS ), funs(mean(., na.rm=TRUE))  
  ) %>%  
  ungroup()
```

Outra forma de calcular as estatísticas por grupo e para uma variável por vez usando o R básico é a função **tapply**:

```
media_po <- tapply( complete_data$PO, complete_data$SETOR, mean )
```

Criando variáveis

Podemos querer categorizar as variáveis pelo seu tamanho.

```
max_po = max(complete_data$PO)  
breaks_tam = c(0, 29, 99, 499, max_po )  
classes_tam = c( "ME", "P", "M", "G")  
  
complete_data$TAM <- cut(complete_data$PO, breaks= breaks_tam, labels=classes_tam )  
  
head(complete_data, 10) # outra config doa função head
```

```
##   SETOR   PO     VA VENDAS TAM  
## 1    17 6256 326795 815997  G  
## 2    17  749  32456 117211  G  
## 3    17 1486  57863 172387  G  
## 4    17 1083  58655 202492  G  
## 5    17  892  44033 112413  G  
## 6    17 1004  54624 141330  G  
## 7    17  861  37248 112320  G
```

```

## 8      17  580 39885 99332   G
## 10     17  500 20081 48492   G
## 11     17  352 12714 64766   M
summary(complete_data$TAM)

##      ME      P      M      G
## 5816 11521 4468   892

```

Duas variáveis categóricas - tabulação cruzada

```

xtabs( data = complete_data, ~SETOR + TAM )

##      TAM
## SETOR   ME      P      M      G
##    17 4464 8230 2559 225
##    24 1352 3291 1909 667

```

Duas variáveis contínuas - correlação

```

cor( complete_data$PO, complete_data$VA ) # pearson
## [1] 0.8846779
cor( complete_data$PO, complete_data$VA , method = "spearman" )
## [1] 0.8551828

```

Análise gráfica

Atualmente há diversos pacotes que permitem analisar graficamente dados no R. Um dos mais populares atualmente é o pacote `ggplot2`. Para isso, primeiro carregamos o pacote usando a função `library()`:

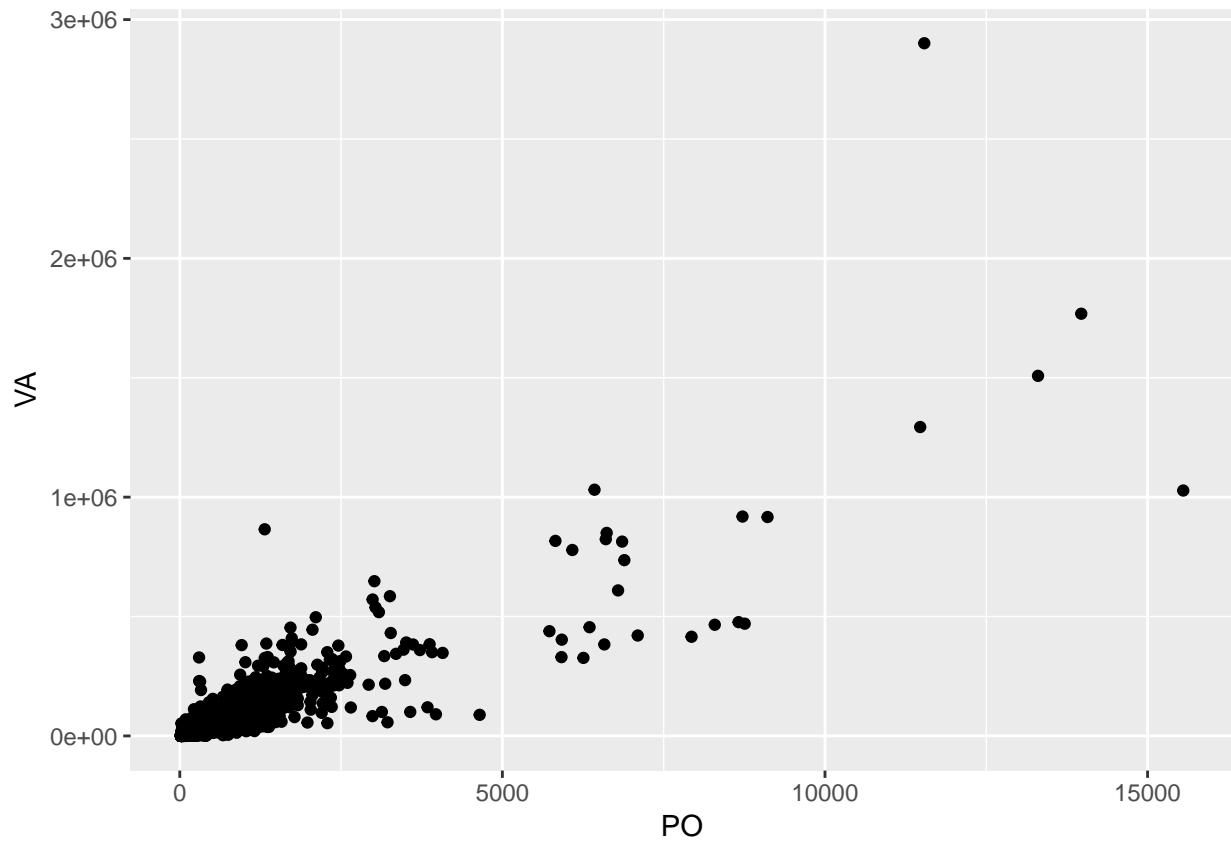
```

library(ggplot2)

## Warning: package 'ggplot2' was built under R version 3.4.4
scatter_plot1 <- ggplot(data = complete_data, aes(x= PO, y = VA )) +
  geom_point()

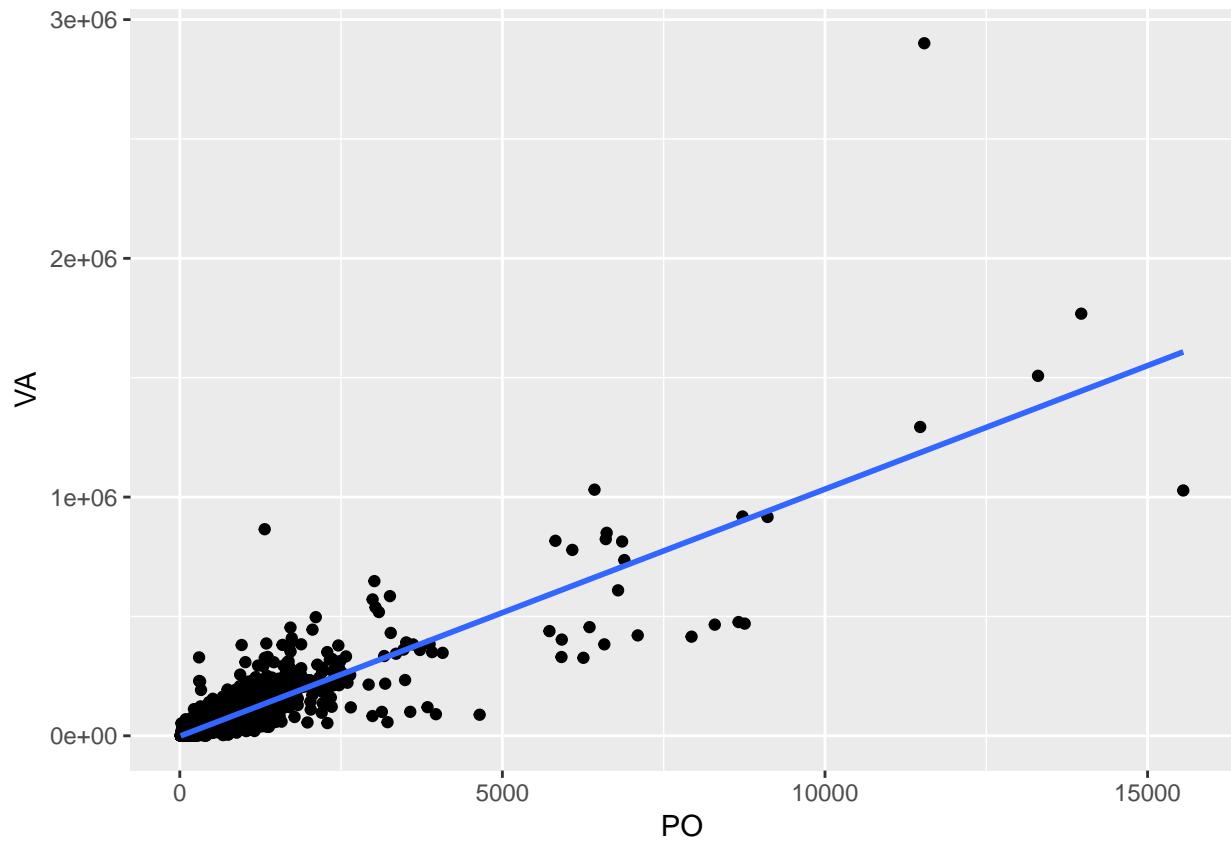
scatter_plot1

```



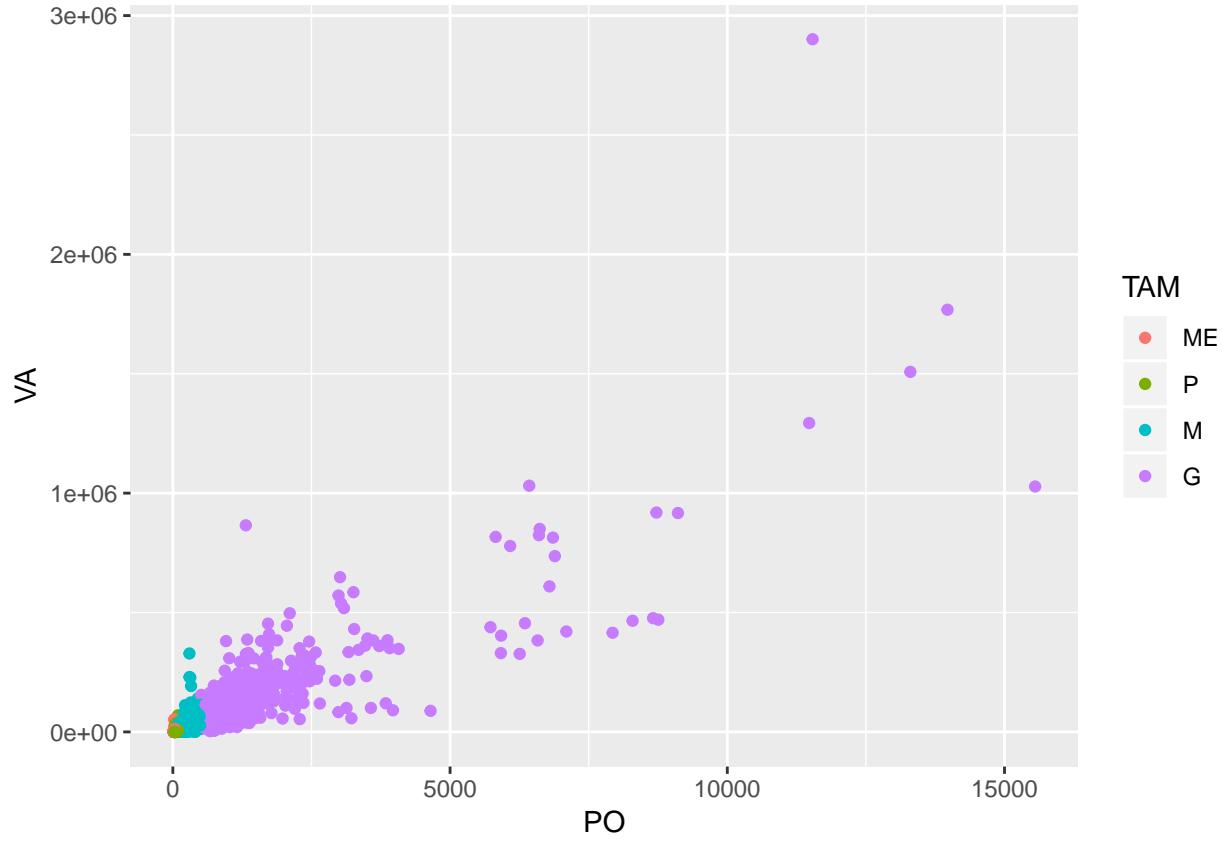
Podemos adicionar uma tendência:

```
scatter_plot2 <- ggplot(data = complete_data, aes(x= PO, y = VA )) +  
  geom_point() +  
  geom_smooth(method="lm", se = TRUE)  
scatter_plot2
```



Podemos também identificar os pontos de acordo com a classe de tamanho da empresa:

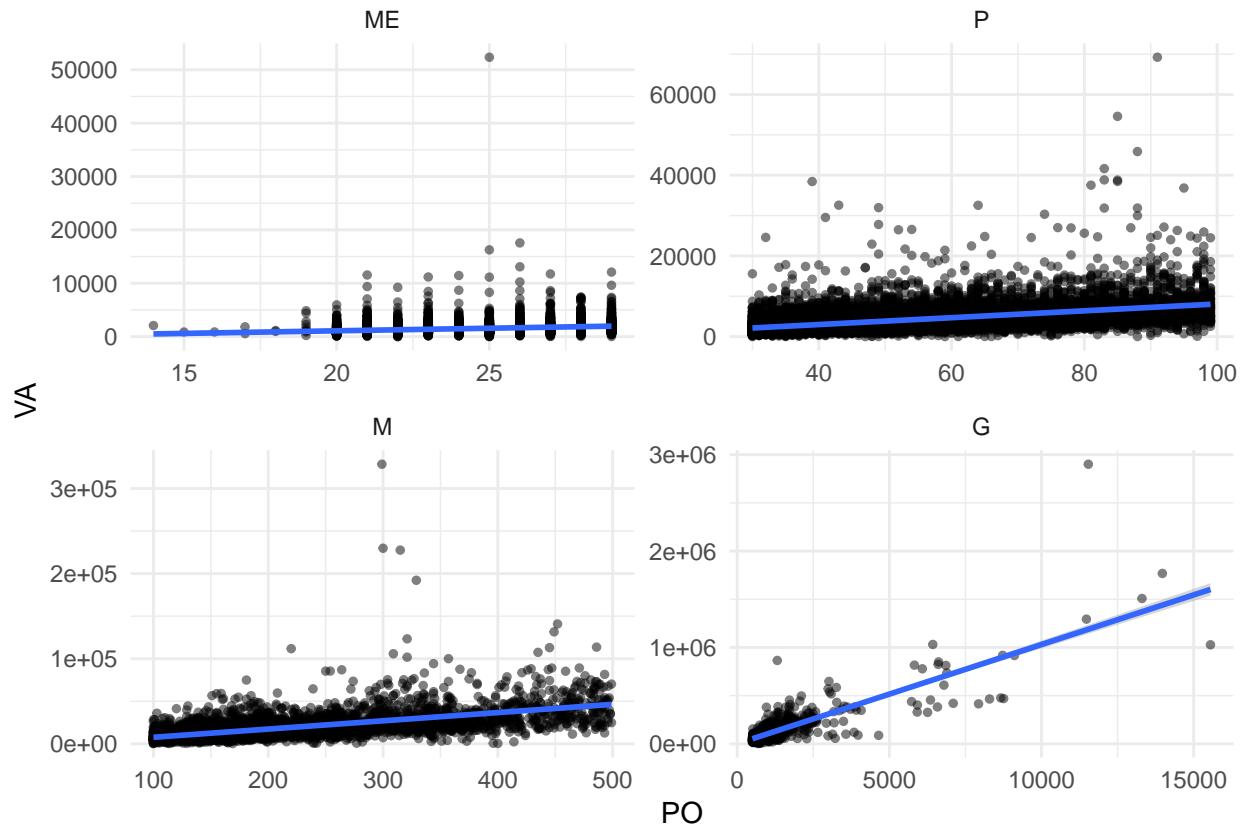
```
scatter_plot3 <- ggplot(data = complete_data, aes(x= PO, y = VA, col = TAM )) +  
  geom_point()  
  
scatter_plot3
```



Podemos também ainda, preferir um gráfico de dispersão por categoria de tamanho e mudar o *estilo* do gráfico:

```
scatter_plot4 <- ggplot(data = complete_data, aes(x= PO, y = VA )) +
  geom_point( size=1, alpha = 0.5 ) +
  geom_smooth(method="lm")+
  facet_wrap(~TAM, scales="free") +
  theme_minimal()

scatter_plot4
```



Para salvar a figura

```
png("scatter_plot.png", height = 768, width = 1024 )
scatter_plot4
dev.off()

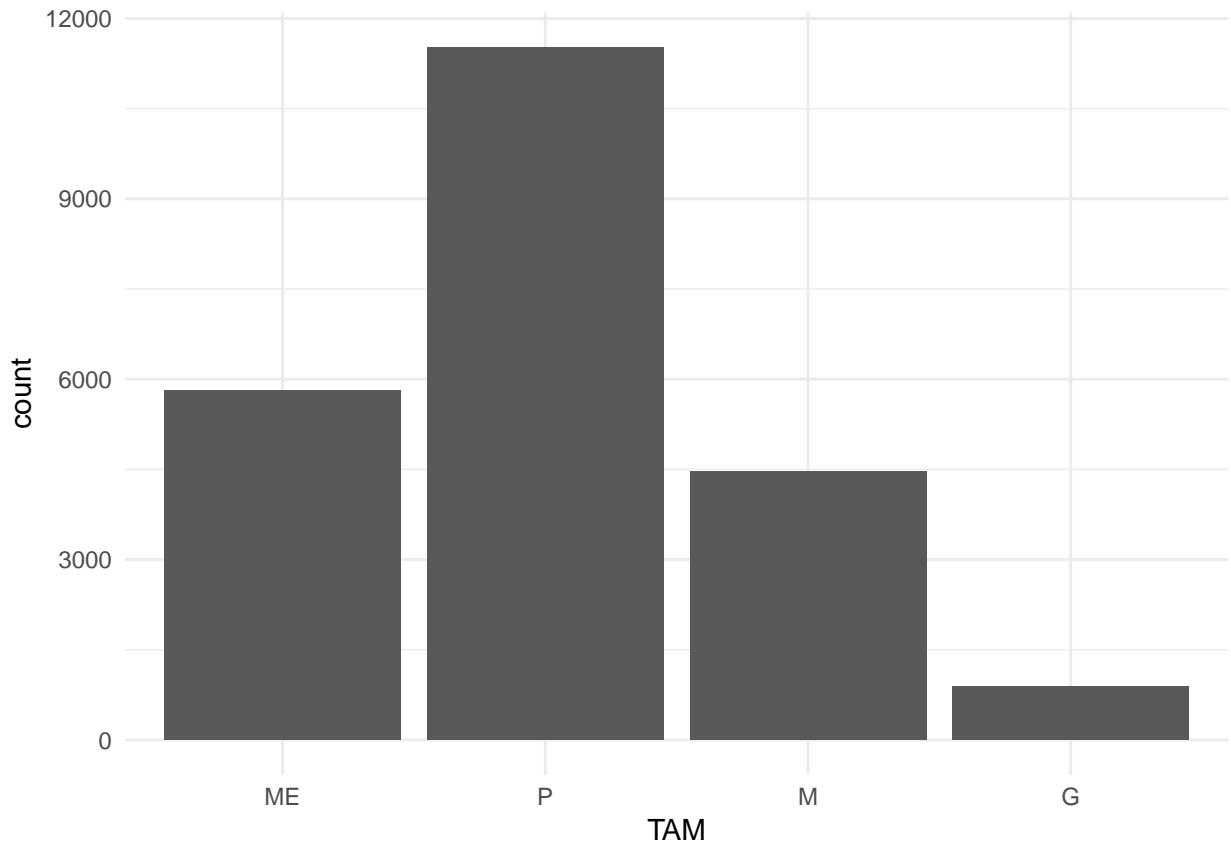
## pdf
## 2
```

Histogramas, distribuições e box-plots

Se queremos um histograma de frequência da composição dessa indústria por tamanho, podemos elaborar um histograma de frequência. Por exemplo, para a variável categórica TAM:

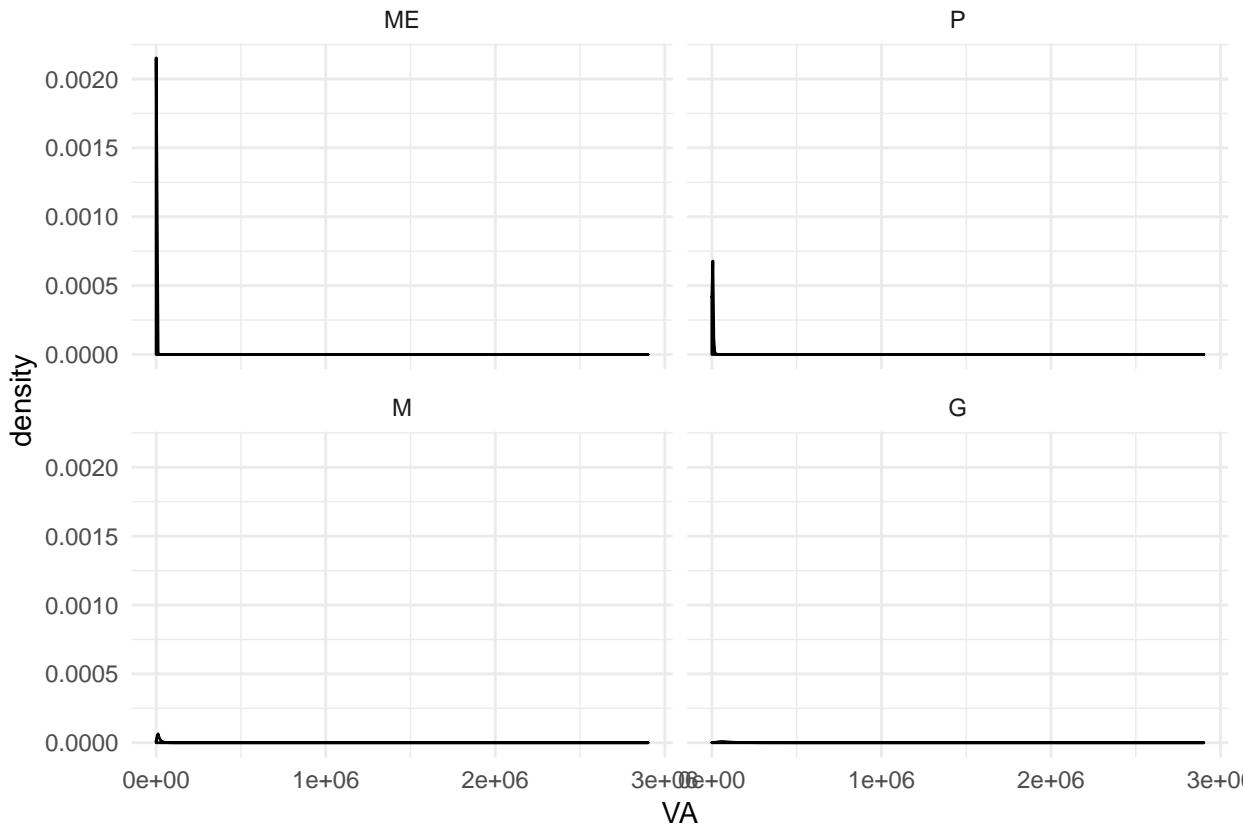
```
bar_plot5 <- ggplot(data = complete_data, aes(x = TAM )) +
  geom_bar() +
  theme_minimal()

bar_plot5
```



Para o caso de variáveis contínuas, podemos estimar a função de densidade. Usando a variável VA:

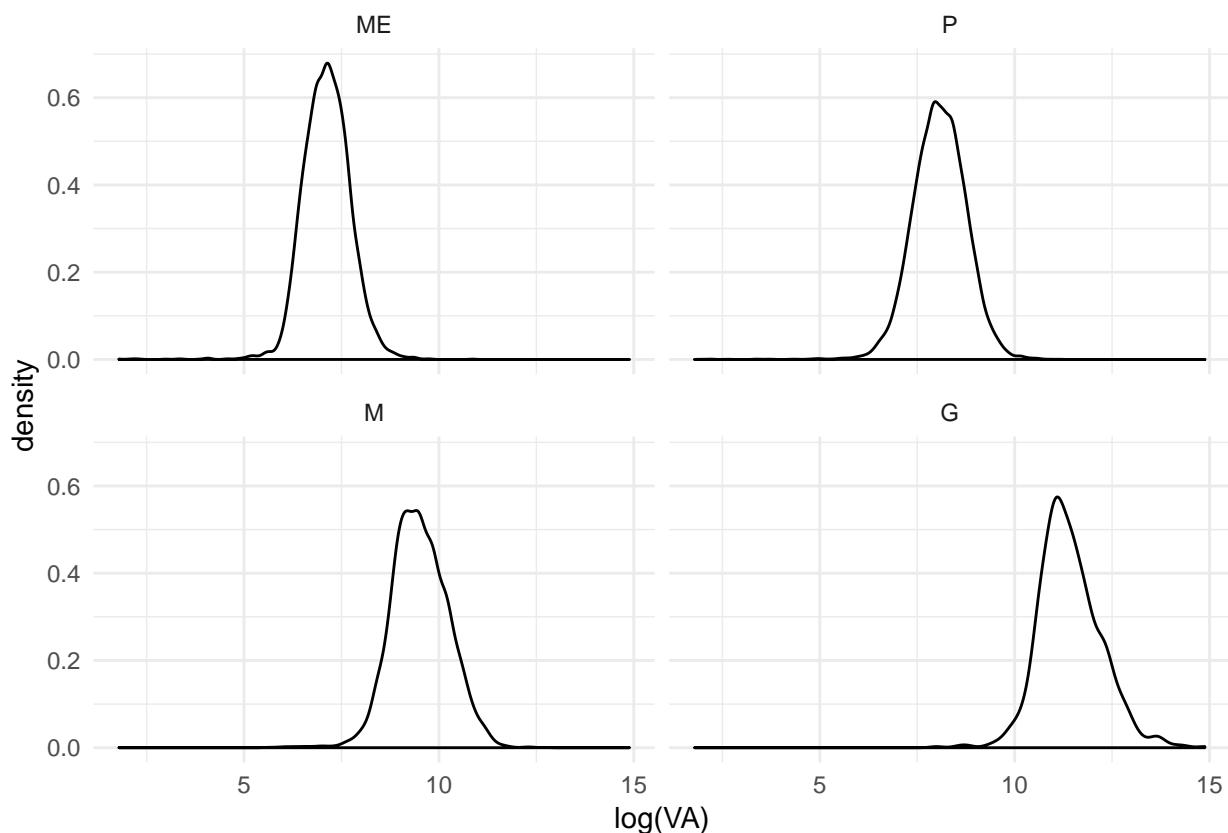
```
density_plot <- ggplot(data = complete_data, aes( x = VA )) +  
  geom_density() +  
  facet_wrap(~TAM) +  
  theme_minimal()  
  
density_plot
```



Como se nota, esta ilustração é um tanto inútil. Mas podemos tentar uma transformação da variável usando agora o `log(VA)`:

```
density_plot2 <- ggplot(data = complete_data, aes( x = log(VA) )) +
  geom_density() +
  facet_wrap(~TAM) +
  theme_minimal()

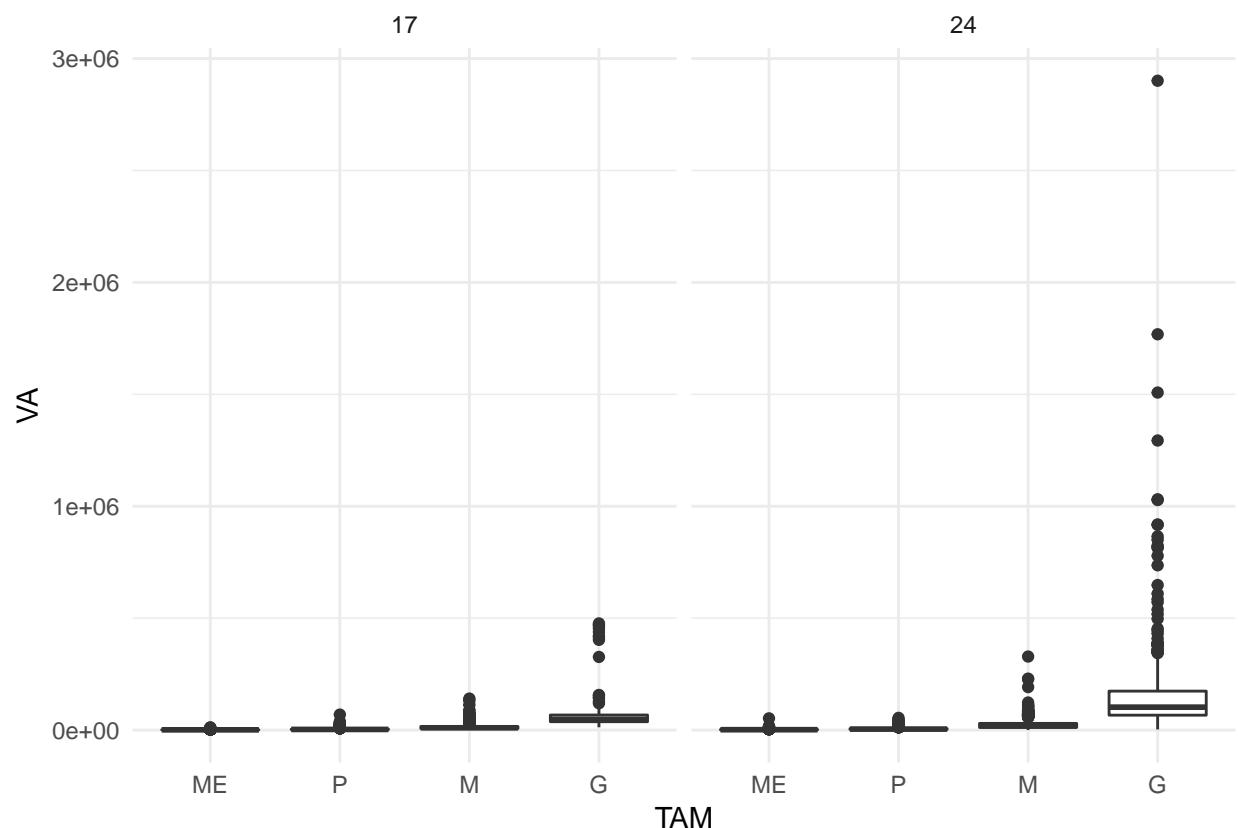
density_plot2
```



Finalmente, para produzir um box-plot, seguimos a mesma lógica:

```
box_plot <- ggplot(data = complete_data, aes( x = TAM, y = VA )) +
  geom_boxplot() +
  facet_wrap(~SETOR) +
  theme_minimal()

box_plot
```



Links intéressantes

- 10 R packages I wish I knew about earlier
- Awesome R
- Graphics with ggplot2
- Multiple regression
- Descriptive statistics