Introdução à linguagem R

Jean S. S. Resende, Jéssica M. Magno, João C. D. Muzzi, Mauro A. A. Castro

2023-06-04

Contents

Chapter 1

Prefácio

A linguagem R foi criada por professores do departamento de estatística da univer sidade de Auckland no ano 2000. A intenção destes professores era disponibilizar uma linguagem *open-source* para computação estatística. Com o avanço e popularização da linguagem, profissionais de diversas áreas passaram a utilizá-la em suas análises de dados.

Esta apostila contém uma introdução à linguagem de programação R. Aborda desde conteúdos teóricos quanto práticos, com exemplos didáticos a fim de facilitar o aprendizado. Esta apostila foi produzida principalmente pelos autores: Jean Silva de Souza Resende, Jéssica Maria Magno, João Carlos Degram Muzzi sob orientação de Mauro Antônio Alves Castro. Em versões anteriores, tivemos a colaboração dos autores: Sheyla Trefflich, Danrley R. Fernandes e Giuseppe Pasqualato Neto.

Chapter 2

Introdução

2.1 Contextualizando a linguagem de programação R

A linguagem R é uma linguagem de programação com o foco em computação estatística e manipulação de gráficos. Criada no início dos anos 90 por Geroge Ross Ihaka e Robert Clifford Gentleman, o R é usado mais utilizado por estatísticos, bioinformatas, analistas de dados e desenvolvedor de *software* estatístico. No entanto ele tem se destacado na comunidade científica. Em maio de 2023, o R ocupava a 16ª posição no índice TIOBE, uma medida de popularidade da linguagem de programação, sendo que em agosto de 2020 o R atingiu seu pico em ficando 8º lugar.

O R é um ambiente de *software* livre de código aberto, disponível sob a *GNU General Public License*. Seus executáveis pré-compilados são fornecidos para vários sistemas operacionais. Ele tem uma interface de linha de comando, mas também possui interfaces gráficas de usuário (GUI) de terceiros como o Rstudio - que será a IDE (*Integrated Development Envirenment*) que iremos utilizar na apostila.

2.2 Instalação do R

1. Acesse o repositório do R (clique aqui).
2. Acesse o link referente ao seu sitema operacional: Linux, macOS ou Windows. 2.1. Linux: escolha a distribuição linux (debian, fedora, redhat, suse ou ubuntu) e então prossiga com os comandos no terminal. 2.2. macOS: escolha o instalador conforme o modelo da sua máquina e execute-o. 2.3. Windows: acesse o link Base e então baixe o instalador e execute-o.

2.3 Instalação do RStudio

1. Clique aqui para acessar o repositório do RStudio.
2. Baixe o instalador conforme o sistema operacional da sua máquina (Linux/macOS/Windows).

2.4 Interface do RStudio

Por padrão o RStudio abre quatro janelas (pode ocorrer de uma estar oculta, mas observe o botão de minimizar/maximizar no canto superior direito de cada janela).

- **Editor de código** (*janela do canto superior esquerdo*): Nela você digita os comando a serem executados no RStudio. Para executá-los aperte as teclas 'CTRL' e 'ENTER' simultaneamente na linha ou bloco de código selecionado.
- **Console** (*janela do canto inferior esquerdo*): É visto as saídas dos comandos que são rodados. Também é possível digitar e rodar códigos diretamente nesta janela.
- **Histórico** (*janela do canto superior direito*): Nesta janela ficam salvos os objetos, históricos de comandos e conexões com outros aplicativos.
- **Visualização** (*janela do canto inferior direito*): Aqui você pode visualizar os gráficos no RStudio, navegar entre os arquivos do seu computador, visualizar os pacotes instalados e ver a ajuda de comandos e descrições de tabelas de dados e por fim navegar entre os arquivos html.

2.5 Pacotes

Por ser *Open Source*, o R permite que qualquer usuário disponibilize funções e bancos de dados a comunidade. As funções/bancos de dados são disponibilizados através de pacotes. A instalação de um pacote depende do repositório que ele está armazenado: máquina local, CRAN, GitHub, Bioconductor, entre outros. O repostitório CRAN contêm muitos pacotes e não é direcionado à uma área específica (como é o caso do Bioconductor que se destina a pacotes voltados para área de biotecnologia). A instalação de um pacote do repositório CRAN é feita pelo menu "Tools > Install Packages" ou simplesmente utilizando o seguinte comando:

```
installed.packages("nomeDoPacote")
```

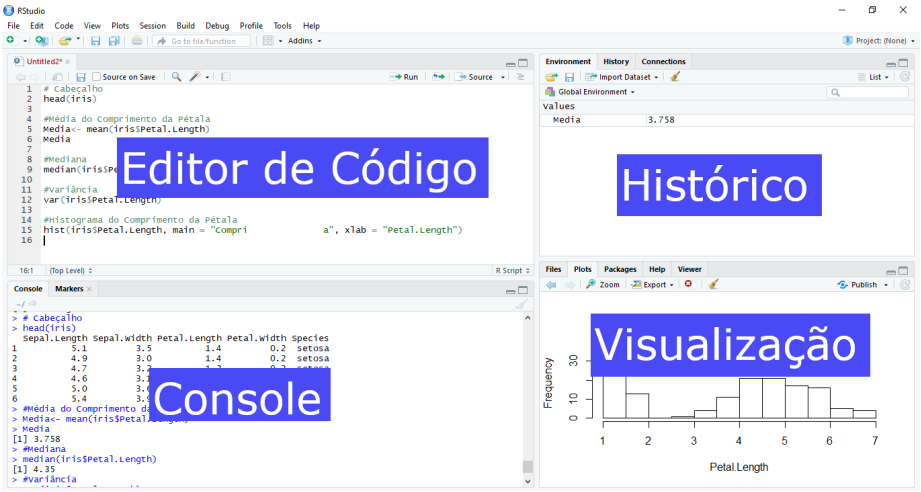


Figure 2.1: Janelas do RStudio. Ref: https://www.est.ufmg.br/~cristianocs/Pacotes2021/Intro.html#6

Para que você possa utilizar as funções do pacote que instalou, você deve usar um dos dois comandos a seguir, para de fato carregar as funções do pacote para o ambiente R:

```
library(nomeDoPacote)
require(nomeDoPacote)
```

A função `library()` é utilizada normalmente no corpo do script, enquanto que a função `require()` é utilizada dentro de outras funções.

2.6 Começando de fato a programar em R

No console (janela do canto inferior esquerdo) digite o comando a seguir e tecle ENTER:

```
print("Hello World")
```

Agora digite o mesmo comando no editor de código (janela do canto superior esquerdo) e com o cursor na mesma linha do comando, tecle CTRL e ENTER simultaneamente:

```
print("Hello World")
```

A diferença é que quando executamos os comandos no editor de código, o comando continua no editor para ser executado, ou seja svocê está construindo um script. Mas você executa comando diretamente no console, eles não ficam gravados em um editor.

O dado de saída da função, foi um print do que estava dentro da função. Mas como você saber o que usar dentro de uma determinada função, como `print()`? Você precisa acessar o manual desta função.

2.7 Acessando o manual da função

Esta é uma etapa muito importante que antecede a sua caminhada no aprendizado do R. Você pode visualizar o manual da função executando um comando onde um ponto de interregoção (?) precisa anteceder a função:

```
?print
```

Mas se você deseja encontrar funções que realizam uma determinada ação, basta inserir dois pontos de interrogação antecedendo a ação desejada:

```
??priting
```

O comando acima realizará uma busca por tópicos que contenham a palavra *plotting*. Outra opção alternativa ao **?** é o uso da função `help()` e `help.search()` para **??**

```
help("print")
help.search("priting")
```

Algumas funções possuem exemplos de sua execução. Se você quer saber como utilizar uma determinada função através de exemplos, execute a função `example()`.

```
example("print")
```

2.8 Comentando códigos no R

A maioria das linguagens de programação e até linguagem de marcação, possuem uma forma de inserção de textos que não serão executados pela linguagem. Esse procedimento é denominado de comentário. Você pode comentar os seus códigos. Isso é algo essencial para todos os programadores, indiferente da linguagem. Pois, códigos comentados facilitam a interpretação do mesmo por

outros programadores e até mesmo pelo autor, devido a um período de tempo que se passou desde a criação daquele código.

Para comentar linhas no R você precisa inserir o # antes do que seria o comentário:

```
# isto é um comentário
```

Exemplo aplicado:

```
print("Hello World") # imprimindo na tela Hello World
```

Perceba que o conteúdo após o # não é interpretado no R, ou seja, este conteúdo é um comentário.

Chapter 3

Fundamentos básicos da programação

3.1 Variáveis

Utilizamos a variável para armazenar um valor qualquer em um local da memória RAM do computador. Deste modo, é possível reutilizar esse valor, usando o nome da sua variável.

3.1.1 Declaração e atribuição de variáveis

Em R declaramos uma variável atribuindo a ela um valor em três formas diferente: **símbolo de atribuição <-**, **símbolo de atribuição =** e **função assign()**.

```
nome.var <- valor # atribuicao: menor e traco
nome.var = valor # atribuicao: igual
assign("nome_var",valor) # funcao: assign
```

3.1.2 Dicas para nomear variáveis

As variáveis podem ser nomeadas com o uso letras, números, ponto (.) e underline (_), no entanto é necessário se atentar para algumas dicas de como nomear as variáveis:

1. O nome da variável deve sempre começar com uma letra ou um ponto, ou seja, não pode iniciar com números ou símbolos. Se iniciar com ponto o próximo caracter não pode ser um número.

2. O nome da variável que contêm mais de uma palavra é recomendado o uso do underline (_) para separa-la.
3. O nome da variável não pode ser palavras reservadas da linguagem como TRUE, if, while, entre outras.
4. O nome da variável não pode conter espaços.
5. O nome da variável deve ser condizente com o seu valor.

3.1.3 Tipos de dados das variáveis

Em R o tipo de dado da variável é obtido a partir do valor atribuído à ela. Isto faz da linguagem R: **Linguagem dinamicamente tipada**. Pois, o tipo de dado de uma variável pode ser alterado dinamicamente enquanto o programa/script é executado.

As variáveis em R podem ser do tipo: inteiro (interger), ponto flutuante (double), complexo (complex), caracteres (character/string) e lógico (logical).

```
var_int <- 2L      # var integer
var_db1 <- 1.5     # var double
var_db2 <- 2       # var double
var_comp <- 2 + 3i # var complex
var_str <- "a_01"  # var string/character
var_log <- TRUE    # var logical
```

Podemos verificar o tipo das variáveis criadas no *chunk* anterior através da função `typeof()`.

```
typeof(var_int)
```

```
## [1] "integer"
```

```
typeof(var_db1)
```

```
## [1] "double"
```

```
typeof(var_db2)
```

```
## [1] "double"
```

```
typeof(var_comp)
```

```
## [1] "complex"
```

```
typeof(var_str)
```

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

```
typeof(var_log)
```

```
## [1] "logical"
```

Para verificar quais variáveis o R está usando *workspace* usando a função `ls()`.

```
ls()
```

```
## [1] "var_comp" "var_db1"  "var_db2"  "var_int"  "var_log"  "var_str"
```

Para excluir variáveis, ou seja, desalocar determinada variável da memória RAM, basta usar a função `rm()`.

```
rm(var_str)     # desaloca a variavel var_str
rm(list = ls()) # desaloca todas as variaveis
```

3.2 Operações em R

Podemos executar operações matemáticas, lógicas e comparações em R. Para isso o R faz uso de **operadores**. Os operadores são divididos em: aritmético, relacional e lógico.

Os operadores aritméticos como o nome já diz são usados em operações aritméticas e são eles:

- Adição: +
- Subtração: -
- Multiplicação: *
- Divisão: /
- Resto de divisão: %%
- Divisão inteira: %/%
- Potenciação: ^

```
2+2   # soma
5-2   # subtracao
2*5   # multiplicacao
5/2   # divisao
5%%2  # resto de divisao
5%/%2 # divisao inteira
2^5   # potenciacao
```

Já os operadores relacionais, tratam da relação de um valor com o outro e são eles:

- Menor: <
- Maior: >
- Menor ou igual: <=
- Maior ou igual: >=
- Igual: ==
- Diferente: !=

```
2<5 # menor
2>5 # maior
2<=2 # menor ou igual
2>=5 # maior ou igual
5==5 # igual
2!=2 # diferente
```

Por fim, os operadores lógicos são:

- *logical NOT*: !
- *logical AND*: &
- *logical OR*: |

```
!TRUE # NOT = qual e o contrario de TRUE?
TRUE | FALSE # OR = um dos dois ou os dois é ou são verdadeiros?
TRUE & FALSE # AND = os dois são verdadeiros?
```

3.3 Condições e loops

Existem dois passos que são trilhados por toda linguagem de programação, e alguns progrmadores dizem que se uma linguagem de programação não permite a execução destes dois passos, ela não é bem considerada uma linguagem de programação. Um exemplo é a linguagem HTML, essa linguagem é dita como

linguagem de marcação sua finalidade é trabalhar com estruturação de textos. Não iremos utilizá-la para cálculos ou procedimentos que demandam de uma rotina computacional com base em cálculos e nos dois passos. Mas quais são estes dois passos? R: condições e *loops*.

3.3.1 Condições

Se alguma coisa for verdadeira (TRUE) o R vai agir de uma maneira, caso seja mentira (FALSE) ele vai agir de outra maneira. Você pode estabelecer algumas condições para que seja feita uma função.

3.3.1.1 Condição: if `if()`

Determinado código será executado somente se a condição for verdadeira, abaixo é apresentado a estrura do `if`.

```
# -- estrutura

# if(condicao){
#   comandos a serem executados
# }
```

Vamos agora fazer uma aplicação: se o número dois for maior que o número um, então imprima na tela a frase: dois é maior que um. Caso contrário não faça nada.

```
# -- aplicacao
## -- verdadeiro
if(2>1){
  print("dois é maior que um")
}
```

```
## [1] "dois é maior que um"
```

No exemplo abaixo a condição é falsa, logo o comando dentro de if não é executado.

```
## -- falso
if(2<1){
  print("dois é menor que um")
}
```

3.3.1.2 Condição: if else `if() else()`

Podemos querer que um comando seja executado se condição for verdadeira e outro comando seja executado se a condição for falsa. Faremos da seguinte forma:

```
if(TRUE){
  print("comando dentro do if")
}else{
  print("comando dentro do else")
}
```

```
## [1] "comando dentro do if"
```

Por ser verdadeira a condição dentro do `if`, foi executado o primeiro comando.

```
if(FALSE){
  print("comando dentro do if")
}else{
  print("comando dentro do else")
}
```

```
## [1] "comando dentro do else"
```

A condição dentro do `if` é falsa então foi executado o comando dentro do `else`

Outra forma de aplicar a condição if else é usando a função `ifelse()`.

```
ifelse(2 > 1, 2*1, 1/2) # condicao verdadeira
```

```
## [1] 2
```

```
ifelse(2 < 1, 2*1, 1/2) # condicao falsa
```

```
## [1] 0.5
```

3.3.2 *Loops*

É muito trabalhoso reescrever código a fim de obeter repetições, sem mencionar o tempo gasto nesta reescrita. Sendo assi, o R possui algumas funções de repetições são elas: `for()`, `while()` e `repeat()`.

A função `for()` repete o código para o comprimento da sequência indicada à ela.

```
for(variavel in sequencia){
  comandos a serem repetidos
}
```

No exemplo abaixo a variável i vai assumir um valor da sequência numérica 1, 2, 3, 4 e 5, e então executar a função `print()` em i para cada valor da sequência adotada por i.

```
# : cria uma sequencia Ex.: sequencia do 1 ao 5 = 1:5
for(i in 1:5){
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Outro exemplo do uso do for: Vamos printar na tela as cindo primeiras letras do alfabeto.

```
for(letra in letters[1:5]){
  print(letra)
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

Já a função `while()` executa os comandos enquanto a condição informada a ela for verdadeira.

```
while(condição){
  comandos a serem repetidos
}
```

Por exemplo: vamos construir um temporizador que determina um espaço de tempo de cinco segundos.

```
contador <- 1
while(contador <= 5){
  print(contador)
  contador = contador + 1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Perceba que não foi exatamente um espaço de tempo de cinco segundos, foi mais rápido. Vamos inserir um comando ao R dizendo a ele para aguardar um segundo após a execução anterior.

```
contador <- 1
while(contador <= 5){
  print(contador)
  contador = contador + 1
  Sys.sleep(1)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

A função `repeat()` é usada quando queremos repetir um código sem a avaliação de uma condição. Atenção: vamos precisar utilizazr a função `break()` para dizer ao programa o momento em que deve parar a execução, ou seja a repetição. Também utilizaremos a função `if()` para avaliar a condição e então chamar o **break**.

```
contador <- 10
repeat{
  print(contador)
  contador <- contador + 10
  if(contador > 100) break()
}
```

```
## [1] 10
```

```
## [1] 20
## [1] 30
## [1] 40
## [1] 50
## [1] 60
## [1] 70
## [1] 80
## [1] 90
## [1] 100
```

3.4 Lista de exercícios

1. Declare três variáveis atribuindo valores numéricos e apresente o resultado da multiplicação das suas combinações dois a dois destas três variáveis (cada variável com um número). Ex.: variáveis A, B, e C mostre AxB, AxC e BxC com atribuição dos valores as variáveis.

2. Converta (no R) a temperatura Fahrenheit 78 °F para Centígrados. Fórmula: C = (F-32) x (5/9).

3. Calcule (no R):

- o resto da divisão de 7 por 9
- 2 elevado ao cubo
- raiz quadrada de 64

4. Elabore um algoritmo que:

- crie um vetor com uma sequência numérica de 5 números
- faça um loop para calcular a soma destes números

5. Elabore um algoritmo que:

- crie um vetor com uma sequência numérica de 7 números
- faça um loop para calcular a média destes números

6. Elabore um algoritmo que:

- crie um vetor com uma sequência numérica de 12 números
- faça a soma dos números pares

Chapter 4

Fundamentos do R

Em linguagens de programação um objeto é qualquer coisa que pode ser armazenado em uma variável. Em R, os principais tipos de objetos são:

- **Vetores**: objeto unidimensional contendo TODOS os valores de um tipo de dado.
- **Matrizes**: vetor (es) em duas dimensões.
- **Arrays**: matrizes em mais dimensões.
- **Fatores**: semelhante a vetores, mas com níveis/agrupamentos.
- **Dataframes**: parecido com matriz mas permite vetores de vários tipos
- **Listas**: parecido com vetor mas os elementos podem ser de tipos diferentes.
- **Funções**: cálculo

Vamos comentar sobre alguns objetos. Não iremos falar sobre todos, pois depende do porquê você quer usar o R, qual tipo de dado e análises você vai utilizar.

4.1 Vetores

Haverá momentos que precisaremos armazenar uma sequência de dados. Por exemplo: queremos armazenar as idades de pacientes de uma clínica, não seria viável declarar uma variável para cada paciente. Sendo assim, o R possui alguns objetos que são capazes de trabalhar com dados de acordo com o seu tipo.

O primeiro objeto que iremos trabalhar é o **vetor**. Em R um vetor é uma sequência de dados de um mesmo tipo. A função que iremos utilizar para criarmos um vetor é a `c()` de concatenar.

```
idades <- c(45,67,78,49)
idades
```

```
## [1] 45 67 78 49
```

Podemos armazenar dados de textos em um vetor.

```
pacientes <- c("id_003","id084","id009","id102")
pacientes
```

```
## [1] "id_003" "id084"  "id009"  "id102"
```

Mas não podemos criar um vetor mesclando os tipos de dados, ao fazer isso o R converte todos os valores para `character`. Isso é um procedimento denominado de coerção, veremos mais adiante

```
exemplo <- c(5,"Maria",TRUE)
```

Os vetores são classificados em classes, são elas:

- **Vetor numérico**: constituído por valores numéricos.
- **Vetor de *strings***: constituído por caracteres.
- **Vetor lógico**: constituído por valores lógicos

```
v_num <- c(1,2,3,4,5)   # vetor numerico
v_str <- c("a","b","c") # vetor de strings/character
v_log <- c(T,F,T)       # vetor logico
```

Para verificar a classe de um objeto, iremos utilizar a função `class()`.

```
class(v_num)
```

```
## [1] "numeric"
```

```
class(v_str)
```

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

```
class(v_log)
```

```
## [1] "logical"
```

4.1.1 Indexação de vetores

Podemos recuperar todos os valores de uma só vez de um vetor ou apenas alguns elementos. A posição inicial de um vetor possui valor 1 e segue de forma crescente da esquerda para a direita. Se quisermos acessar um elemento de determinada posição no vetor, precisaremos utilizar `[posição desejada]`

```
genes <- c("TP53","COI","RHO")
genes[2]
```

```
## [1] "COI"
```

4.1.2 Gerando sequências e número aleatórios

Uma forma de criar sequências é utilizar : (início:fim).

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Outra forma é utilizar a função `seq()`. Com ela podemos informar o início, fim e intervalo.

```
seq(1,10,2)
```

```
## [1] 1 3 5 7 9
```

As funções que geram números aleatórios são importantes nos processos de amostragem. as funções mais utilizada para esse tipo de terafa são: `sample()` e `runif()`.

```
sample(0:50, 10) # gerar 10 num inteiros entre 0 e 50 sem reposicao
```

```
##  [1] 33  5 29 24 40  2 31  1 21 49
```

```
sample(0:10, 5, replace = TRUE) # gerar 5 valores entre 0 e 10 com reposicao
```

```
## [1] 2 1 0 6 2
```

```
runif(5, 0, 10) # gerar 5 valores decimais entre 0 e 10
```

```
## [1] 4.202064 9.915460 8.838954 2.080994 4.290114
```

Também podemos trabalhar com do tipo `character`.

```
sample(c("feminino", "masculino"), 10, replace = TRUE)
```

```
##  [1] "masculino" "feminino"  "masculino" "feminino"  "feminino"  "masculino"
##  [7] "feminino"  "masculino" "masculino" "feminino"
```

4.1.3 Operações com vetores

Depois de criar os vetores podemos fazer diversas operações com eles. Abaixo seguem alguns exemplos básicos de algumas operações com vetores.

```
# vetores
a <- c(1,3,5,7)
b <- c(2,4,6,8)

# soma de vetores
soma_V <- a+b
soma_V
```

```
## [1]  3  7 11 15
```

```
# subtracao_V de vetores
subtacao_V <- b-a
subtacao_V
```

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

```
# produtos escalares
produto_V <- a*b
produto_V
```

```
## [1]  2 12 30 56
```

```
# divisao
divisao_V <- b/a
divisao_V
```

```
## [1] 2.000000 1.333333 1.200000 1.142857
```

```
# potenciacao
potenciacao <- a^b
potenciacao
```

```
## [1]       1      81   15625 5764801
```

Algumas operações básicas voltadas para a parte estatística envolvendo os vetores:

```
prostate <- c(65,58,59,63,78,67)

max(prostate)      # valor maximo
```

```
## [1] 78
```

```
min(prostate)      # valor minimo
```

```
## [1] 58
```

```
range(prostate)    # minimo e maximo
```

```
## [1] 58 78
```

```
sum(prostate)      # soma dos valores
```

```
## [1] 390
```

```
length(prostate)   # numero de observacoes
```

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

```
mean(prostate)     # media
```

```
## [1] 65
```

```
median(prostate)   # mediana
```

```
## [1] 64
```

```
quantile(prostate) # quartis
```

```
##   0%  25%  50%  75% 100%
## 58.0 60.0 64.0 66.5 78.0
```

```
summary(prostate)  # resumo
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    58.0    60.0    64.0    65.0    66.5    78.0
```

4.2 Matrizes

Matriz é um objeto que distribui os elementos em coordenadas (linhas e colunas). Matriz é mais utilizada em análises multivariadas de fenômenos biológicos, como por exemplo, na organização, manejo e cálculos realizados com dados de mudanças conformacionais de proteínas ao longo do tempo. Há diversas formas de construir uma matriz, veremos algumas delas.

Função `matrix()`: Esta função é própria para a criação de matrizes em R. Para saber mais a fundo sosbre os argumentos da função acesse o manual da função (`?matrix`). Basicamente precisamos informar um conjunto de dados para a função criar a matriz.

```
matrix(1:10)
```

```
##       [,1]
##  [1,]    1
##  [2,]    2
##  [3,]    3
##  [4,]    4
##  [5,]    5
##  [6,]    6
##  [7,]    7
##  [8,]    8
##  [9,]    9
## [10,]   10
```

No entanto, podemos usar mais argumentos da função.

```
# setando numero de linhas e colunas
matrix(1:10, ncol = 5, nrow = 2)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

```
# setando a ordem prenchimento pelas linhas
matrix(1:10, ncol = 5, nrow = 2, byrow = TRUE)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
```

Funções `cbind()` e `rbind()`: Outr forma de criar matrizes é juntando vetores por meio da junção de colunas (`cbind()`) ou linhas (`rbind()`).

```
vet_1 <- 1:5
vet_2 <- seq(from=12, to=20, by=2)

# unindo vetores atraves das colunas
cbind(vet_1, vet_2)
```

```
##      vet_1 vet_2
## [1,]     1    12
## [2,]     2    14
## [3,]     3    16
## [4,]     4    18
## [5,]     5    20
```

```
# unindo vetores atraves das linhas
rbind(vet_1, vet_2)
```

```
##       [,1] [,2] [,3] [,4] [,5]
## vet_1    1    2    3    4    5
## vet_2   12   14   16   18   20
```

4.2.1 Acessando elementos da matriz

Os comandos para acessar elementos de uma matriz são parecidos com os comandos utilizados para acessar os elementos de vetores, a diferença é que na matriz temos linhas e colunas, sendo assim precisamos informar a linha e a coluna do elemento que desejamos visualizar.

```
A <- matrix(runif(15,0,10), ncol = 3)

A[3,2] # elemento com coordenadas: terceira linha  e segunda coluna
```

```
## [1] 0.1061726
```

```
A[,1] # elemento com coordenadas: todas as linhas e somente a primeira coluna
```

```
## [1] 7.008574 2.771562 9.219508 3.999402 6.972004
```

```
A[1:3,] # elemento com coordenadas: linhas 1, 2 e 3 e todas as colunas
```

```
##          [,1]      [,2]      [,3]
## [1,] 7.008574 3.6827593 9.2561433
## [2,] 2.771562 6.1772811 6.8985037
## [3,] 9.219508 0.1061726 0.7745459
```

4.3 *Data frames*

Esse objeto é como um banco de dados, as colunas são as variáveis e as linhas referem-se aos registros. A diferença entre matrizes e data frames é que matrizes são vetores com duas dimensões de modo que, possuem uma única classe, já os data frames possuem colunas que podem ter classes diferentes.

4.3.1 *Data frames* disponibilizazdos pelo R

O R disponibiliza vários bancos de dados como exemplo. Vamos trabalhar com um banco de dados disponibilizado pelo R. Para isso vamos utilizar a função `data()`.

```
# consultando os bancos de dados disponiveis
data()

# acessando um banco de dados
data("iris")

# documentacao do banco de dados
?iris
```

4.3.2 Criando um *data frame*

Para criar um *data frame* no R vamos utilizar a função `data.frame()`.

```
# construindo uma tabela de metadados de pacientes
metadados_pac <- data.frame(
  paciente = c("A01", "A02", "B03", "B05", "C01"),
  idade = c(45, 48, 65, 54, 72),
  sexo = c("masculino", "masculino", "feminino", "masculino", "feminino"),
  fumante = c(TRUE, FALSE, FALSE, FALSE, TRUE))

# visualizando o data frame
metadados_pac
```

```
##   paciente idade      sexo fumante
## 1      A01    45 masculino    TRUE
## 2      A02    48 masculino   FALSE
## 3      B03    65  feminino   FALSE
## 4      B05    54 masculino   FALSE
## 5      C01    72  feminino    TRUE
```

```
# visualizando a idade e sexo
metadados_pac[,c("idade","sexo")]
```

```
##   idade      sexo
## 1    45 masculino
## 2    48 masculino
## 3    65  feminino
## 4    54 masculino
## 5    72  feminino
```

4.3.3 Executando algumas funções em *data frames*

```
# -- banco de dados: iris --
## -- primeiro contato com o dado
?iris # documentacao

dim(iris)         # dimensoes: linhas e colunas
```

```
## [1] 150   5
```

```
head(iris)        # cabecalho
```

```
##   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
```

```
tail(iris)        # calda
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 145          6.7         3.3          5.7         2.5 virginica
## 146          6.7         3.0          5.2         2.3 virginica
## 147          6.3         2.5          5.0         1.9 virginica
## 148          6.5         3.0          5.2         2.0 virginica
## 149          6.2         3.4          5.4         2.3 virginica
## 150          5.9         3.0          5.1         1.8 virginica
```

```
summary(iris)     # resumo estatistico
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

```
str(iris)         # estrutura
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 .
```

```
iris$Sepal.Length # selecionando uma variavel
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
##  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
##  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
##  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
##  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
##  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

4.4 Listas

Uma lista no R é como um vetor capaz de armazenar elementos com diferentes tipos de dados (armazenando em uma dimensão). Uma mesma lista podemos armazenar vetor, matriz, dataframe, e outros. Sendo assim, as lista podem ser complexas .

Para criar uma lista utilize a função `list()`

```
lista <- list("Hello world",vet_1,A, metadados_pac)
lista
```

```
## [[1]]
## [1] "Hello world"
##
## [[2]]
## [1] 1 2 3 4 5
##
## [[3]]
##          [,1]      [,2]      [,3]
## [1,] 7.008574 3.6827593 9.2561433
## [2,] 2.771562 6.1772811 6.8985037
## [3,] 9.219508 0.1061726 0.7745459
## [4,] 3.999402 5.0926704 6.9443905
## [5,] 6.972004 0.5400815 5.5505336
##
## [[4]]
##   paciente idade      sexo fumante
## 1      A01    45 masculino    TRUE
## 2      A02    48 masculino   FALSE
## 3      B03    65  feminino   FALSE
```

```
## 4      B05    54 masculino   FALSE
## 5      C01    72  feminino    TRUE
```

Os colchetes duplos `[[]]` indicam qual elemento da lista está sendo apresentado, enquanto que o simples `[]` mostra o subelemento da lista.

4.5 Funções

Um dos grandes poderes do R é a criação de funções com o intuito de realizar trabalhos específicos. A sintaxe básica é: function(lista de argumentos) corpo da função. function indica para o R que estamos criando uma função. Lista de argumentos são argumentos que serão avaliados pela função, tais argumentos são separados por vírgulas. Já o corpo da função é onde iremos escrever o nosso algoritmo a ser utilizado para realizar os cálculos. Utilizamos o assignment para nomear a função.

```
# criando a funcao
mult_lad <- function(x,y){
  x * y
}

# executando a funcao
mult_lad(2,5)
```

```
## [1] 10
```

```
mult_lad(10,17)
```

```
## [1] 170
```

Podemos criar uma funcao que calcule o pH de écidos e bases fortes

```
# criando a funcao
ph <- function(h){
  -log10(h)
}

# utilizando a funcao
ph(0.01)
```

```
## [1] 2
```

```
ph(10^-5)
```

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

4.6 Lista de exercícios

1. Crie uma matriz com a sequência de 1 a 20, contendo 4 linhas
2. Calcule a média de todas as colunas da matriz do exercício anterior.
3. Calcule a média dos números pares de todas as colunas da matriz do exercício 1.
4. Crie um objeto com 100 valores aleatórios de uma distribuição uniforme. Quantos elementos são maiores ou iguais a 0,5?

Chapter 5

Processamento de dados

O R possui diversas funções para várias operações com manejo de dados e neste tópico iremos ensinar o básico, ou seja, como importá-los, exportá-los e editá-los no R. Na maioria das vezes os dados não estão em um formato ideal e precisamos manejá-los. A manipulação de dados envolve a transformação de um tipo de objeto em outro, limpeza e filtragem, agregações, reestruturação e junção dos dados.

Para obter os arquivos de input utilizados neste tópico acesse:

5.1 Primeiros passos

5.1.1 Tipos de variáveis

Os objetos no R, podem armazenar diferentes tipos de dados, no decorrer deste subtópico iremos trabalhar com alguns desses tipos.

```
# -- TIPOS DE VARIAVEIS --

## tipo caracter
caracter <- "string"

## tipo numerico inteiro
inteiro <- 7L

## tipo numerico decimal
decimal <- 7.5

## tipo logico
logico <- TRUE
```

Para verificar o tipo de cada objeto criado vamos utilizar a função `class()`.

```
# -- VERIFICACAO DE VARIAVEIS --
## funcao class()
class(caracter)
```

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

```
class(inteiro)
```

```
## [1] "integer"
```

```
class(decimal)
```

```
## [1] "numeric"
```

```
class(logico)
```

```
## [1] "logical"
```

```
## funcao is.
is.double(caracter)
```

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

```
is.numeric(caracter)
```

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

```
is.character(caracter)
```

```
## [1] TRUE
```

```
is.double(decimal)
```

```
## [1] TRUE
```

```
is.numeric(decimal)
```

```
## [1] TRUE
```

```
is.integer(decimal)
```

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

Existem também tipos mais complexos como: vector; array; matrix; list; data.frame; factor.

5.1.2 Conversão de tipos de variáveis

Muitas vezes quando importamos os dados para o R, ele os trata como sendo de um tipo diferente do qual desejamos, assim, é preciso convertê-los.

```
# -- CONVERSAO DE VARIAVEIS --
## alguns exemplos de transformacoes
conv_decimal <- as.double(inteiro) # convertendo para decimal
conv_inteiro <- as.integer(decimal) # convertendo para inteiro
conv_caracter <- as.character(inteiro) # convertendo para caracter
```

5.1.3 Valores faltantes

Valores faltantes, assim como o próprio nome já diz, são valores ausentes, ou seja, quando você estiver trabalhando com seus dados, talvez você não tenha todos os valores esperados, então você precisa preencher esses "valores faltantes"com a sigla NA, pois o R ao encontrar o NA ele entende que é um valor que está faltando. Abaixo seguem algumas funções para trabalhar com NA.

```
# -- TRABALHANDO COM VALORES FALTANTES --
## Trabalhando com banco de dados do R
### verificar os bancos de dados data () disponiveis no R

### escolher o banco de dados iris para tralhar
data("iris")

### fazer uma copia do banco de dados
data_iris <- iris

### remover o banco de dados do ambiente de trabalho
```

```
rm(iris)

### estudar o banco de dados
?iris

### verificar o tipo do objeto
class(data_iris)

### "sentindo os dados" - resumo estatistico
summary(data_iris)

### verificar a ocorrencia de NA
#### is.na retorna TRUE caso haja NA (valores não anotados), do contrário retorna FALSE
is.na(data_iris$Sepal.Length)
is.na(data_iris$Sepal.Width)

### verificar se todo o objeto esta preenchido
#### complete.cases retorna TRUE se todos os dados estão preenchidos e FALSE para os fa
complete.cases(data_iris$Sepal.Length)
complete.cases(data_iris$Sepal.Width)

## Trabalhando com seus dados (exemplo)
### criando banco de dados
coorte <- data.frame(
barcode = c("A01B1", "A01B2", "A02B3"),
sexo = c("Feminino", "Feminino", NA),
fumante = c(T, F, T),
idade = c(65, 68, 79))

### visualizar os dados
View(coorte)

### "sentindo os dados" - resumo estatistico
summary(coorte)

### vericar o conteudo de NA
is.na(coorte$sexo)
complete.cases(coorte$sexo)

# Perceba que complete.cases e is.na retornam respostas contrários, pois o primeiro ve

### preencher NA com "Não informado"
coorte$sexo[is.na(coorte$sexo)] <- "Não informado"
### verificar se o objeto esta preenchido
complete.cases(coorte$fumante)
```

5.2 Importação de arquivos

Haverá momentos que precisaremos importar arquivos para o ambiente R e transformá-los em objetos. Antes de importar nossos dados, precisamos executar alguns comandos para averiguar se estamos no diretório correto (onde o arquivo está armazenado), ou seja, confirmar se o arquivo realmente está no diretório.

```
# -- DIRETORO DE TRABALHO --
## Verificar o diretorio de trabalho
getwd()

## Definir diretorio de trabalho
setwd("caminho/onde/se/encontra/o/arquivo")

## Verificar quais arquivos estao no mesmo diretorio
dir()
```

Agora iremos utilizar alguns comandos para importar para o ambiente R arquivos com diferentes extensões.

```
# -- IMPORTACAO DE ARQUIVOS PARA O AMBIENTE R --
input_1 <- read.table(file = "ChickWeight.csv", header = T, sep = ",")
input_2 <- read.csv(file = "arquivo(,).csv", header = T)
input_3 <- read.csv2("arquivo(;).csv", header = T)
```

Agora vamos utilizar a função read_delim() do pacote readr para ler arquivos com diversos tipos de separadores. Por exemplo, vamos importar um arquivo compactado:

```
# -- IMPORTACAO DE ARQUIVOS COMPACTADOS PARA O AMBIENTE R --
library(readr)
input_zip <- read_delim("arquivo.zip", delim = ";")
```

Quando não quisermos importar um arquivo inteiro, mas apenas algumas observações (linhas) e variáveis (colunas), faremos da seguinte maneira:

```
# -- IMPORTACAO DE PARTES DE UM ARQUIVO PARA O AMBIENTE R --
## Vamos importar as quatro primeiras linhas de um arquivo
input_txt_4l <- read_delim("arquivo(;).csv", delim = ";", n_max = 4)
```

Para importar arquivos do Excel (extensão “xlsx”) utilizaremos o pacote readxl.

```
# -- IMPORTACAO DE ARQUIVOS DO EXCEL --
input_xlsx <- readxl::read_xlsx("arquivo.xlsx")
```

5.3 Exportação de arquivos

Para exportar arquivos do ambiente R, para um diretório do seu computador, utilizaremos a função `write.table()`. Esta função está contida no pacote base do R, ou seja, esse pacote é instalado no momento em que instalamos o R.

```
#  -- EXPORTACAO DE ARQUIVOS DO AMBIENTE R PARA O UM DIRETORIO NO COMPUTADOR
## Exportando arquivo no diretorio de trabalho
write.csv(input_csv_4l, file = "teste.csv", row.names = F, quote = F)
write.csv2(input_csv_4l, file = "teste2.csv", row.names = F, quote = F)

## Exportando arquivo em um dirtorio especifico
write.csv2(input_csv_4l, file ="~/caminho/do/diretorio/desejado/teste.csv",row.names =
```

5.4 Manipulação de dados com pacotes básicos

A manipulação de dados consiste em modificar ou selecionar variáveis de interesse dentro de um conjunto de dados. Nesse exercício, vamos baixar dados de biópsias de nódulos de mama de Wiscosin (W.N. Street, W.H. Wolberg and O.L. Mangasarian, 1993).

```
# Vamos baixar os dados de características de nódulos de mama de Wiscosin
# Mais informações sobre esse dataset em https://archive.ics.uci.edu/ml/datasets/Breas

download.file(
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/breast-
cancer-wisconsin/wdbc.data',
destfile = "wbdc.data")

# Vamos importar os dados para o ambiente do R.
# Os dados estão separados por vírgula, por isso o argumento sep=','
# outras opções e argumentos podem ser vistos em help(read.table)
brca <- read.table(file = "./wbdc.data", sep = ',')

# Ver nomes das colunas
colnames(brca)
View(brca)
# Mudar nome da segunda coluna
```

```
colnames(brca)[2] <- "Coluna2"
colnames(brca)
# Mudar todos os nomes de colunas conforme descritos no dataset
colnames(brca) <- c('id_number', 'diagnosis', 'radius_mean',
                    'texture_mean', 'perimeter_mean', 'area_mean',
                    'smoothness_mean', 'compactness_mean',
                    'concavity_mean','concave_points_mean',
                    'symmetry_mean', 'fractal_dimension_mean',
                    'radius_se', 'texture_se', 'perimeter_se',
                    'area_se', 'smoothness_se', 'compactness_se',
                    'concavity_se', 'concave_points_se',
                    'symmetry_se', 'fractal_dimension_se',
                    'radius_worst', 'texture_worst',
                    'perimeter_worst', 'area_worst',
                    'smoothness_worst', 'compactness_worst',
                    'concavity_worst', 'concave_points_worst',
                    'symmetry_worst', 'fractal_dimension_worst')
head(brca)
View(brca)
class(brca)
summary(brca)
```

Ao observarmos o summary do data frame que criamos, percebemos que a maioria das colunas foram identificadas com valores numéricos, com exceção da coluna 'diagnosis'. Essa coluna foi identificada contendo caracteres e no summary indica 369 caracteres (1 para cada linha). Entretanto, ao observarmos melhor, essa coluna apresenta somente valores de 'B' para amostras benignas e 'M' para amostras malignas. Nesse caso, podemos alterá-los para fatores para facilitar o manejo dos dados.

```
# Acessando o resumo da coluna 'diagnosis' do data frame 'brca'
summary(brca$diagnosis)

# Transformando os valores da coluna em fatores
brca$diagnosis <- as.factor(brca$diagnosis)

# Agora o summary indica a quantidade de linhas em cada fator. Um resumo que faz mais sentido pa
summary(brca$diagnosis)
```

5.4.1 Acessando e modificando dados

Aqui vamos ver maneiras de acessar, visualizar ou modificar elementos dentro de um data frame. Existe mais de uma forma de realizar as mesmas ações como podemos ver nos exemplos a seguir.

```
# Acessando itens do data frame
# Visualizando item na 2 linha e 3 coluna
brca[2,3]

# Mudando valor da 2 linha e 3 coluna
brca[2,3] <- 22
brca[2,3]

# Visualizando valores da 5 linha
brca[5,]

# Visualizando valores da 1 â5 linha
brca[1:5,]

# Visualizando valores da 2 coluna
brca[,2]

# Visualizando coluna "diagnosis"
brca$diagnosis

# Visualizando as 2, 3 e 4 colunas
brca[,c(2,3,4)]

# Visualizando a 2, 3 e 4 colunas e as linhas 2,4,6
brca[c(2,4,6), c(2,3,4)]

# Visualizando as colunas "diagnosis", "radius_mean" e
# "texture_mean" para as linhas 2,4,6
brca[c(2,4,6), c("diagnosis", "radius_mean", "texture_mean")]

# Mudando ordem na visualização das colunas
brca[c(2,4,6), c("radius_mean", "diagnosis", "texture_mean")]

# Excluir 1 coluna
brca <- brca[,-1]

# Criando novo data frame somente com as colunas de interesse:
brca <- brca[,c("diagnosis", "radius_mean", "texture_mean",
"symmetry_mean", "area_mean", "perimeter_mean")]

# Adicionar coluna com a razão do perímetro pelo raio dos nódulos
brca$perimeter.radius <- brca$perimeter_mean/brca$radius_mean summary(brca$perimeter.ra

# Perceba como os valores da razão do perímetro pelo raio dos nódulos se
aproxima do valor de 2*pi (aprox. 6.28)
```

5.4.2 Filtrar observações

Os data frames ou matrizes no R são entendidos como objetos com 2 dimensões, a primeira sendo as observações (linhas) e a segunda as variáveis (colunas). Aqui começaremos tratando de operações com as linhas (observações).

As observações (linhas) do data frame podem ser selecionados de acordo com seus valores ou características. Para isso podemos usar valores lógicos (TRUE or FALSE) ou índices (1,2,3...). Operadores lógicos são úteis para adicionarmos condições a serem cumpridas na seleção dos dados. Na linguagem R, o operador de igual '=' se refere a atribuição de valores, assim como '<-'. Portanto, o operador lógico para igualdade é '=='. Ele irá testar se o valor à esquerda é igual ao valor da direita, caso positivo retorna TRUE, do contrário FALSE. O operador para diferença é '!=', tendo uso análogo. Além disso é possível comparar valores maiores ('>'), menores ('<'), maiores e iguais ('>=') ou menores e iguais ('<=').

```
# Filtrar só amostras benignas
benign <- brca[brca$diagnosis == "B",]
summary(benign)
dim(benign)

# Filtrar só amostras não benignas
malign <- brca[brca$diagnosis != "B",]
summary(malign)
dim(malign)

# Filtrar só tumores com raio maior do 16
radius16 <- brca[brca$radius_mean > 16,]
summary(radius16)

# Filtrar tumores com raio menor ou igual a 16
radius.menor16 <- brca[brca$radius_mean <= 16,]
summary(radius16.menor16)
```

Além disso, é possível adicionar condições. Por exemplo o operador ' ' ('E') indica a intersecção de condições, ou seja, a condição1 E a condição2 devem ser atendidas para que o valor retornado seja TRUE. Caso qualquer uma não seja atendida, o teste lógico retorna FALSE.

```
# Filtrar só tumores com raio maior do que 16 E (&) benignas
radius16ebenign <- brca[brca$radius_mean > 16 & brca$diagnosis == "B",]
summary(radius16ebenign)
```

á o operador '|' ('OU') indica a união de condições, ou seja, a condição1 OU a condição2, ou ambas podem ser atendidas para que o valor retornado seja

TRUE. Nesse caso, o teste lógico só retorna FALSE caso as duas condições NÃO sejam atendidas.

```
# Filtrar só tumores com raio maior do que 16 ou (|) benignas
radius16oubenign <- brca[brca$radius_mean > 16 | brca$diagnosis == "B",]
summary(radius16oubenign)
```

5.4.3 Adicionar e ordenar linhas

Por fim, podemos adicionar linhas ao data frame ou mesmo unir dois data frames verticalmente utilizando a função `rbind()` (row bind).

```
# Adicionar linhas, importante ter o mesmo número de variáveis do que o dataframe
novalinha <- c("M", 20.5, 21.5, 0.2, 1203, 130.5, 6.6)
dim(brca)
brca2 <- rbind(brca, novalinha)
dim(brca2)

# Unindo dados de benignos e malignos
brca2 <- rbind(benign, malign)
summary(brca2)
```

A função `order()` retorna o ordenamento dos índices das linhas. Essa função serve justamente para ordenar um objeto de acordo com seu valor. Caso tenha valor numérico, a função ordena do menor para o maior, caso seja uma letra ou palavra, a função ordena em ordem alfabética. O argumento `decreasing = T`, oferece a opção de ordenamento decrescente tanto numérico quanto alfabético.

```
# Ordenar tabela
brca <- brca[order(brca$radius_mean),]
brca <- brca[order(brca$radius_mean, decreasing = T),]
brca <- brca[order(brca$diagnosis, decreasing = F),]
```

5.4.4 Selecionar colunas, unir data frames horizontalmente e adicionar colunas

De maneira semelhante ao que foi feito com as observações (linhas), aqui podemos selecionar e unir variáveis (colunas). Alguns exemplos já foram mostrados nas seções anteriores. Aqui focaremos em como criar novos data frames com algumas colunas e uni-los com a função `cbind()` (column bind).

```
# Ver dimensões do data frame
dim(brca)

## Cria novo data frame com as 3 primeiras colunas
MeiaTabela <- brca[,1:3]
dim(MeiaTabela)

## Cria novo data frame com as 3 colunas seguintes
MeiaTabela2 <- brca[,4:6]
dim(MeiaTabela2)

## Une as duas tabelas
NovaTabela <- cbind(MeiaTabela, MeiaTabela2)
dim(NovaTabela)

## Adiciona coluna perimeter.radius ao data frame
NovaTabela2 <- cbind(NovaTabela, brca$perimeter.radius)
NovaTabela3 <- cbind(NovaTabela, perimeter.radius = brca$perimeter.radius)
```

5.4.5 Removendo objetos

Os objetos no R são salvos na memória RAM. Ao final da seção, podemos salvar os objetos ou liberar a memória alocada. Entretanto, durante o trabalho podemos limpar o ambiente de trabalho removendo objetos que não serão mais utilizados. Em especial, isso se torna útil quando carregamos data frames muito grandes, os quais podem consumir muito espaço na memória do computador.

```
# Remover objetos do ambiente de trabalho
rm(NovaTabela, NovaTabela2, NovaTabela3, novalinha,
MeiaTabela, MeiaTabela2, brca2, radius16.benign,
radius16ebenign, radius16oubenign)
```

5.5 Manipulação de dados com tidyverse

Neste tópico vamos trabalhar com funções do pacote `dplyr` e `tidyr`. A comunidade que trabalha com R conhece bem esses pacotes devido às funções importantes para manipulação de dados que eles contêm. Essencialmente os pacotes `dplyr` e `tidyr` desempenham as mesmas funções das apresentadas pelas funções básicas. As vantagens se dão em tornar o código mais limpo e legível.

5.5.1 Filtrar observações

A primeira função que utilizaremos é a função `filter()` do pacote `dplyr`, ela filtra observações.

```
# -- MANIPULACAO DE DADOS --
## - FILTRAR DADOS -
### Criando banco de dados
alunos <- data.frame(
nome = c("Jean", "Dan", "Pepe", "Bruna", "Rafaela"),
altura = c(1.74, 1.70, 1.73, 1.65, 1.68),
massa = c(69, 66, NA, 55, 53),
sexo = c("masculino", "masculino", "masculino", "feminino", "feminino")
)

### Filtrar dados
library(dplyr)

#### filtrar observacoes com altura maior que 1.7
filter(alunos, altura >= 1.7)

#### filtrar observacoes com altura maior ou igual a 1.68 sendo do sexo
feminino
filter(alunos, altura >= 1.68 & sexo == "feminino")

#### filtrar observacoes com altura maior ou igual a 1.68, sendo do sexo
feminino e com massa maior que 50
filter(alunos, altura >= 1.68 & sexo == "feminino" & massa > 50)
```

A função `filter()` também está disponível no pacote `stats` (um dos pacotes básicos do R), assim, certifique-se que esteja carregando o pacote correto ou utilize a sintaxe `dplyr::filter()` para ter certeza de que está usando a função certa.

```
# -- MANIPULACAO DE DADOS --
## - FILTRAR DADOS -
dplyr::filter(alunos, altura > 1.68 & sexo == "feminino")
```

5.5.2 Selecionar variáveis

O pacote dplyr disponibiliza a função select(), ela seleciona as variáveis do nosso objeto.

```
# -- MANIPULACAO DE DADOS --
## - SELECIONAR VARIAVEIS -
### selecionar as colunas: nomes, sexo, massa
select(alunos, nomes, sexo, massa)

### selecionar as colunas que terminam com o caracter: "a"
select(alunos, ends_with("a"))

### selecionar as colunas que contêm a sequência de caracteres: "no"
select(alunos, contains("no"))

### selecionar as colunas: 2, 3 e 4
select(alunos, 2:4)

### visualizar todas as linhas e as colunas 1, 2 e 3
alunos[,1:3]

### visuzalizar todas as linhas e as colunas 1, 4 e 3
alunos[, c(1,4,3)]
```

5.5.3 Modificação de dados

Utilizando o mesmo exemplo, vamos criar uma coluna com o nome de imc e colocarmos o resultado do cálculo nessa coluna.

```
# -- MANIPULACAO DE DADOS --
## - INSERIR UMA COLUNA NO CONJUNTO DE DADOS -
### inserir uma coluna nos conjunto de dados com o resultado do imc de cada
aluno
alunos_2 <- mutate(alunos, imc = (peso/(altura)^2))
```

5.5.4 Renomeando variáveis

Que tal renomear o nome de alguma variável? Podemos renomear nossas variáveis utilizando a função `rename()` que também é do pacote `dplyr`.

```
# -- MANIPULACAO DE DADOS --
## - RENOMEAR UMA COLUNA DO CONJUNTO DE DADOS-
### renomear as colunas: altura para altura.m e massa para massa.kg
alunos <- rename(alunos,
                 altura.m = altura,
                 massa.kg = massa)
```

5.5.5 Uso de pipes

Quando queremos fazer operações múltiplas em sequência, podemos utilizar o pipe (%>%), ele pertence ao pacote magrittr. O uso de pipes serve para simplificar o código. Supondo que precisamos selecionar, sumarizar e plotar alguns dados, podemos fazer comandos para cada execução, porém fica confuso porque temos que executar um de cada vez. Com o uso de pipes colocamos um comando em sequência do outro e o resultado de um comando serve como entrada para o outro comando.

```
# -- MANIPULACAO DE DADOS --
## - USO DE PIPES -
### Sem pipes
select(filter(alunos_2, imc >= 23), nome, massa)

### Com pipes
alunos_2 %>%
filter(imc >= 23) %>%
select(nome, massa, imc)

#### a media das massas e o somatorio das massas separando por sexo
alunos_2 %>%
group_by(sexo) %>%
summarise(media = mean(massa),
somatorio = sum(massa))

#### a media, desvio padrao e quantidade das massas separando por sexo
alunos_2 %>%
group_by(sexo) %>%
summarise(media = mean(massa),
sd.massa = sd(massa),
total.massa = n())

#### a media, desvio padrao e quantidade das massas separando por sexo
alunos %>%
group_by(sexo) %>%
summarise(media = mean(massa, na.rm = TRUE),
sd.massa = sd(massa, na.rm = T),
total.massa = n())
```

5.5.6 Ordenar dados

Há momentos em que precisaremos ordenar nossos dados, para termos uma melhor visualização facilitando nossas análises. Para ordenar os dados utilizaremos a função `arrange()` do pacote `dplyr`.

```
# -- MANIPULACAO DE DADOS --
## - ORDENAR DADOS -
### ordenando de forma decrescente o objeto "alunos" atraves da variavel massa
alunos %>%
arrange(desc(massa))

### agrupando por sexo e ordenando pela massa
alunos %>%
group_by(sexo) %>%
arrange(desc(massa), .by_group = TRUE)

#### ---- note que informei ".by_grupo = TRUE" para poder ordenar primeiro o "
feminino" e depois o "masculino" ----
```

5.5.7 Combinar variáveis

Quando quisermos combinar variáveis podemos utilizar a `função bind_cols()`.

```
# -- MANIPULACAO DE DADOS --
## - COMBINAR VARIAVEIS -
### criar conjunto de dados para combinar
complemento <- data.frame(
  nome = c("Jean", "Dan", "Pepe", "Bruna", "Rafaela"),
  tipo.sang = c("A+", "A+", "B-", "B+","B-"))

### combinando objetos
bind_cols(alunos, complemento)
```

Unir variáveis à esquerda (left jooin) Neste caso não é necessário ter o mesmo número de observações, uma vez que serão unidos somente os objetos que contém a mesma chave.

```
# -- MANIPULACAO DE DADOS --
## - UNIR VARIAVEIS A ESQUERDA -
### criar conjunto de dados para combinar
complemento_1 <- data.frame(
  nome = c("Dan", "Pepe"),
  estado = c("PR", "SC"))

### combinando objetos
left_join(alunos, complemento_1)
```

Unir variáveis à direita (right join) Vamos unir observações que correspondem à objetos a direita. Para unir objetos cuja variável de um seja diferente da variável do outro basta informar ao comando.

```
# -- MANIPULACAO DE DADOS --
## - UNIR VARIAVEIS A DIREITA -
### criar conjunto de dados para combinar
alunos_4 <- data.frame(
  aluno = c("Jean", "Tamara", "Rafaela"),
  cid.natal = c("Umuarama-PR", "Curitiba-PR", "Maringa-PR"))

### unindo o objeto da direita ao da esquerda considerando que as variaveis
nome e aluno sao as mesmas
right_join(alunos_4, alunos, by = c("aluno" = "nome"))
```

Unir variáveis internamente (inner join) Diversas vezes queremos unir bancos de dados incompletos, assim, podemos utilizar a função `inner_join()`, ela irá desconsiderar as observações que não possuem todas as variáveis completas.

```
# -- MANIPULACAO DE DADOS --
## - UNIR VARIAVEIS INTERNAMENTE -
### unindo objetos considerando que as variaveis nome e aluno sao as mesmas
inner_join(alunos_4, alunos, by = c("aluno" = "nome"))
```

Unir todas as observações dos objetos (full join) Se desejamos unir todas as observações de dois objetos podemos utilizar a função `full_join()`.

```
# -- MANIPULACAO DE DADOS --
## - UNIR TODAS AS VARIAVEIS -
### unindo objetos considerando que as variaveis nome e aluno sao as mesmas
full_join(alunos_4, alunos, by = c("aluno" = "nome"))
```

Semi join e anti join Se desejamos saber quais observações de um objeto possuem correspondências em outro objeto podemos fazer uso da função `semi_join()`.

```
# -- MANIPULACAO DE DADOS --
## - MESMAS CORRESPONDENCIAS -
### criar um novo conjunto de dados.
alunos_5 <- left_join(complemento_1, alunos)

### verificar observacoes de "alunos" que possuem correspondencias com
observacoes com "alunos.5".
semi_join(alunos, alunos_5)
```

Porém, se desejamos saber quais observações não possuem correspondências em outro objeto podemos fazer uso da função `anti_join()`.

```
# -- MANIPULACAO DE DADOS --
## - CORRESPONDENCIAS DIFERENTES -
### verificar observacoes de "alunos" que nao possuem correspondencias com observacoes de "alunos
anti_join(alunos, alunos_5)
```

Se quisermos distinguir os dados, ou seja, quais observações são de um grupo A e quais são de um grupo B, podemos criar uma lista com os objetos, nomear cada um dos elementos da lista e em seguida unir com a função `bind_rows()` definindo o argumento ".id".

```
# -- MANIPULACAO DE DADOS --
## - SEPARAR EM GRUPOS -
### exemplo 01
lista.1 <- list(alunos_5, alunos)
names(lista.1) <- c("A", "B")
lista.1

### exemplo 02
bind_rows(lista.1, .id = "turma")
```

Pivot wider e pivot longer Duas funções que podem parecer um pouco difíceis de utilizar em um primeiro momento, mas são bastante úteis em determinadas situações são `pivot wider()` e `pivot longer()`. Como o nome sugere, a função `pivot wider()` "alonga"seus dados no sentido horizontal. Ou seja, ela vai distribuir os valores alocados nas linhas dentro de várias variáveis. Na prática, ela diminui o número de linhas do seu data frame distribuindo em mais colunas. A função `pivot longer()` faz o oposto, ela une variáveis (colunas) dentro de uma ou mais colunas pivôs, repetindo os demais valores. Na prática, ela aumenta o número de linhas e diminui o de colunas. Essa função é especialmente útil no manejo de dados para a construção de boxplots (veremos adiante).

Utilizaremos as 10 primeiras linhas do data set com informações da quantidade de cartões que cada jogador levou na Copa do Mundo de 2018.

```
# pivot_wider e longer
# 10 primeiras linhas da tabela de cartões por jogador da Copa do Mundo de 2018
# https://pt.wikipedia.org/wiki/Dados_disciplinares_da_Copa_do_Mundo_FIFA_de_2018
df <- data.frame(
  stringsAsFactors = FALSE,
  Jogador = c("Carlos Sánchez","Michael Lang","Igor Smolnikov","Jérôme Boateng",
              "Sebastian Larsson","Aleksandar Mitrovi","Ante Rebi",
              "Armando Cooper","Blaise Matuidi","Casemiro"),
Selecao = c("Colômbia","Suíça","Rússia","Alemanha","Suécia","Sérvia","Croácia",
            "Panamá","França","Brasil"),
Cartao.Amarelo = c(1L, 0L, 0L, 0L, 3L, 2L, 2L, 2L, 2L, 2L),
```

```
Cartao.Amarelo.Vermelho = c(0L, 0L, 1L, 1L, 0L, 0L, 0L, 0L, 0L, 0L),
Cartao.Vermelho = c(1L, 1L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L))

# Vamos escolher as colunas pivôs, que nesse caso serão as 3 colunas com as quantidade
# A opção names_to se refere a coluna que será gerada com os nomes das colunas pivos e
dim(df)
df_longer <-
tidyr::pivot_longer(
  data = df,
  names_to = "Cartao",
  values_to = "Qtde",
  cols = c("Cartao.Amarelo","Cartao.Amarelo.Vermelho","Cartao.Vermelho"))
dim(df_longer)

# O pivot_wider faz a operação contrária, escolhemos de onde virão os nomes das novas
df_wider <-
  tidyr::pivot_wider(
    data = df_longer,
    names_from = c("Cartao"),
    values_from = c("Qtde"))
dim(df_wider)
```

5.6 Lista de exercícios

1. Utilizando o data frame “trees” responda as seguintes questões:

- Qual a dimensão do objeto “trees”?
- Alguma variável possui valores ausentes?
- Existe alguma variável do tipo “caracter”?
- Qual a média de cada variável?
- Qual o mínimo e máximo de cada variável?
- Na descrição desse data set qual a unidade de cada coluna? Metros, pés, polegadas?

2. Para esse exercício considere 1 polegada = 0.0254 metros e 1 pé = 0.3048 metros e 1 pé^3 = 0.028317 metros cúbicos.Faça as seguintes alterações:

- Passe os valores da variável “Girth” para metros.
- Passe os valores da variável “Height” para metros.
- Passe os valores da variável “Volume” para m^3.

3. Filtre a tabela para somente observações com altura acima de 23m. Quantas observações sobraram?
4. Exporte esse conjunto de dados para o seu diretório de trabalho

Chapter 6

Gráficos

6.1 Gráficos no R

Um dos maiores pontos fortes do R é sua capacidade de visualização dos dados. Existem pacotes especializados que oferecem excelentes recursos de visualização mesmo de elementos complexos. Além disso, existe a possibilidade de geração de gráficos em 3D, gráficos interativos e por fim a aplicação desses gráficos em relatórios (como no Rmarkdown) ou em dashboards ou web apps (como no Shiny).

Nesse capítulo, iremos ensinar os princípios básicos na criação de gráficos, inicialmente utilizando os pacotes básicos do R para em seguida avançarmos no pacote ggplot2. Esse pacote faz parte do conjunto de bibliotecas do tidyverse e apresenta uma sintaxe dos gráficos que facilita a manipulação ou sobreposição de elementos. Além disso, o ggplot2 é compatível com uma série de outros pacotes os quais permitem integrá-lo para as mais diversas finalidades.

Para algumas aplicações específicas outros pacotes gráficos também podem ser utilizados como por exemplo o pheatmap ou ComplexHeatmap para a geração de mapas de calor mais complexos, os quais não iremos abordar por enquanto.

6.2 Gráficos base em R

6.2.1 Visualizando dados

Abra o RStudio e na janela de comando instale o pacote `ggplot2`. Após a instalação, carregue o pacote com a função library.

```
install.packages("ggplot2")
```

```
## Installing package into '/home/jeanresende/R/x86_64-pc-linux-gnu-library/4.3'
## (as 'lib' is unspecified)
```

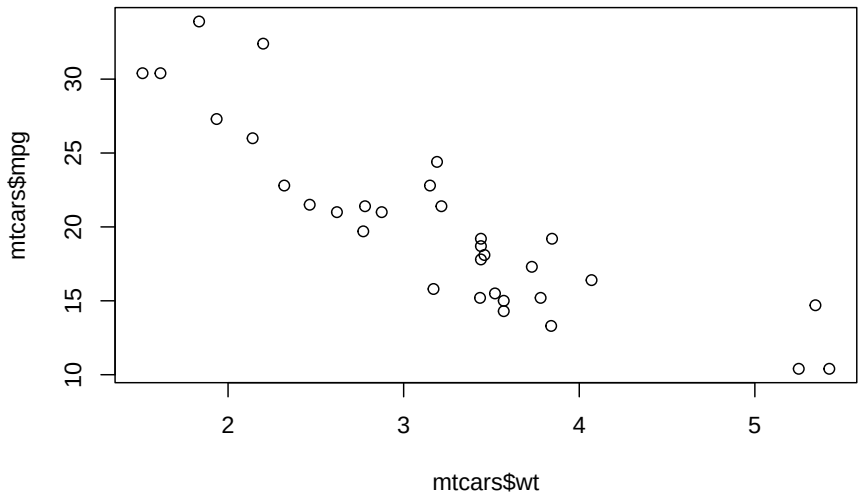
```
library(ggplot2)
```

Antes de conhecer o pacote ggplot2, para uma exploração rápida de dados em R as vezes é mais útil utilizar as funções básicas de plots no R. Essas funções básicas já vem previamente instaladas e não requerem a instalação de pacotes para serem utilizadas. No entanto, para fazer gráficos mais elaborados se torna mais fácil de usar o pacote ggplot2. Primeiramente vamos aprender a utilizar as funções básicas para a criação de plots em R para então, na segunda parte da aula passar a utilizar as funções do pacote ggplot2.

6.2.2 Criando gráficos de pontos

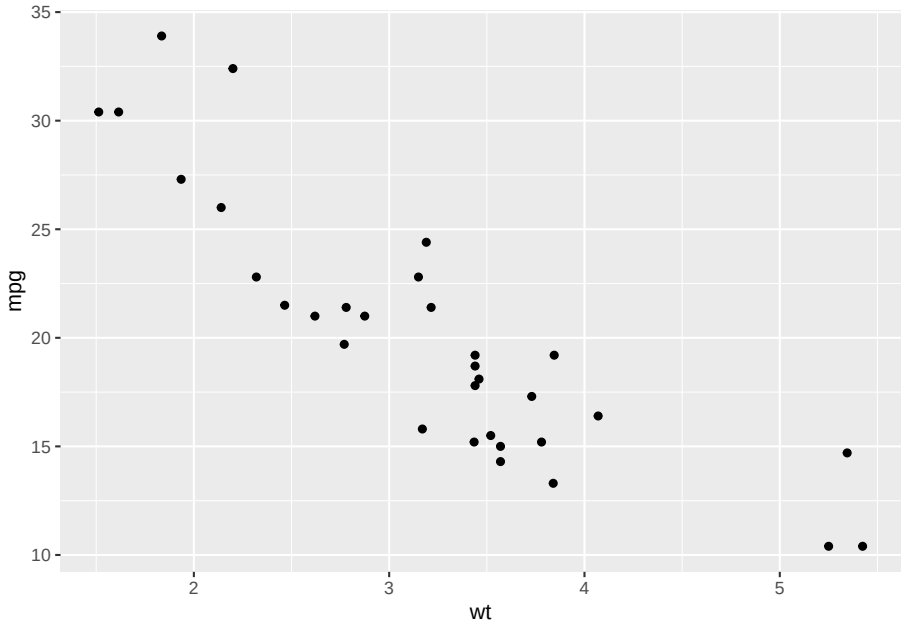
Basicamente para montar um gráfico de pontos basta usar a função plot() e passar como argumento os valores de x e y. Na janela de comando no RStudio digite o seguinte código:

```
plot(mtcars$wt, mtcars$mpg)
```



```
# com o ggplot2 faríamos
qplot(wt, mpg, data = mtcars)
```

```
## Warning: `qplot()` was deprecated in ggplot2 3.4.0.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

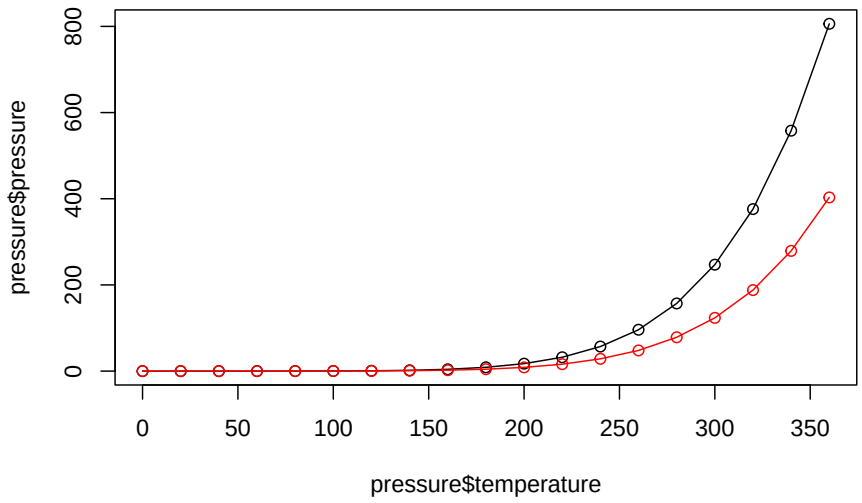


6.2.3 Criando gráficos de linha

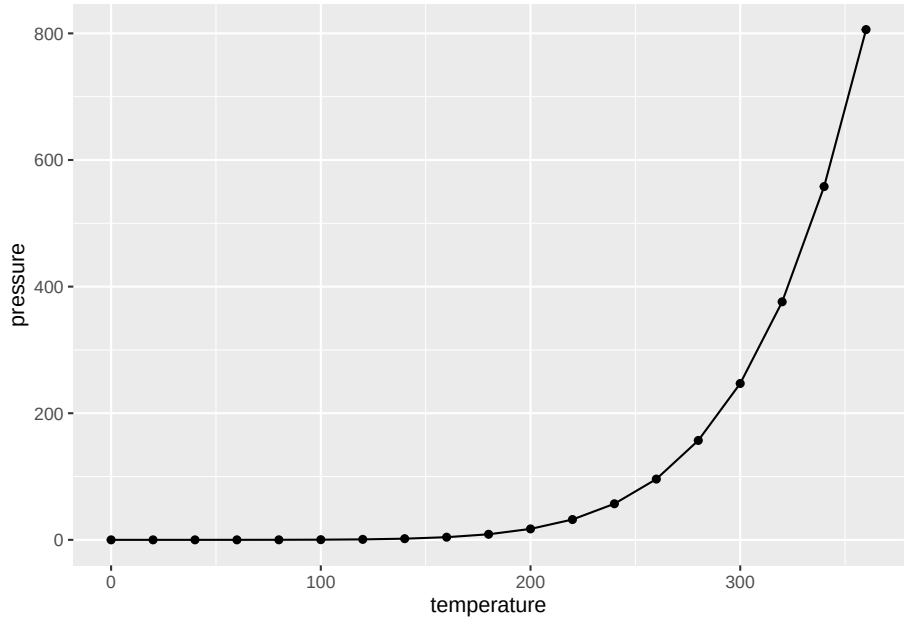
Para fazer um gráfico em linha com a função plot, basta adicionar um outro argumento a função. Na janela de comando (Console) digite o código abaixo e atente para o argumento type = "l".

```
plot(pressure$temperature, pressure$pressure, type = "l")

# para adicionar pontos e/ou múltiplas linhas, primeiramente
# faz se o uso de plot() para a primeira linha e então usa-se
# a função points() para pontos adicionais e lines() para
# linhas adicionais

plot(pressure$temperature, pressure$pressure, type = "l")
```

```
points(pressure$temperature, pressure$pressure)
lines(pressure$temperature, pressure$pressure/2, col = "red")
points(pressure$temperature, pressure$pressure/2, col = "red")
```




```
# com o pacote ggplot2 fazemos uso da
# função qplot(), concatenando linhas e pontos no mesmo vetor
# da seguinte forma

qplot(temperature, pressure, data = pressure, geom = c("line",
"point"))
```



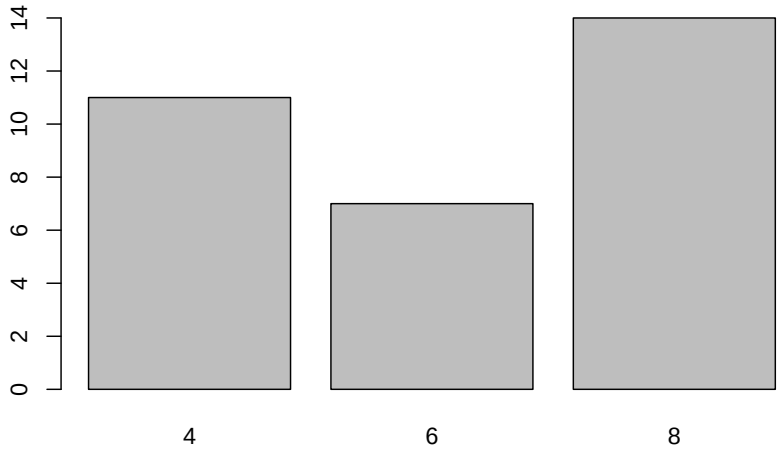
6.2.4 Criando gráficos de barra

A função utilizada para criar gráficos em barra é a `barplot()` e é utilizada passando-se o vetor de valores para a altura de cada barra como argumento da função e de modo opcional, outro argumento que também é utilizado é o vetor de labels para cada barra. As vezes, gráficos em barra se referem a gráficos onde as barras representam a quantidade de casos em cada categoria. Para gerar a quantia de cada valor único em um vetor, usa-se a função `table()`.

```
table(mtcars$cyl)
```

```
##
##  4  6  8
## 11  7 14
```

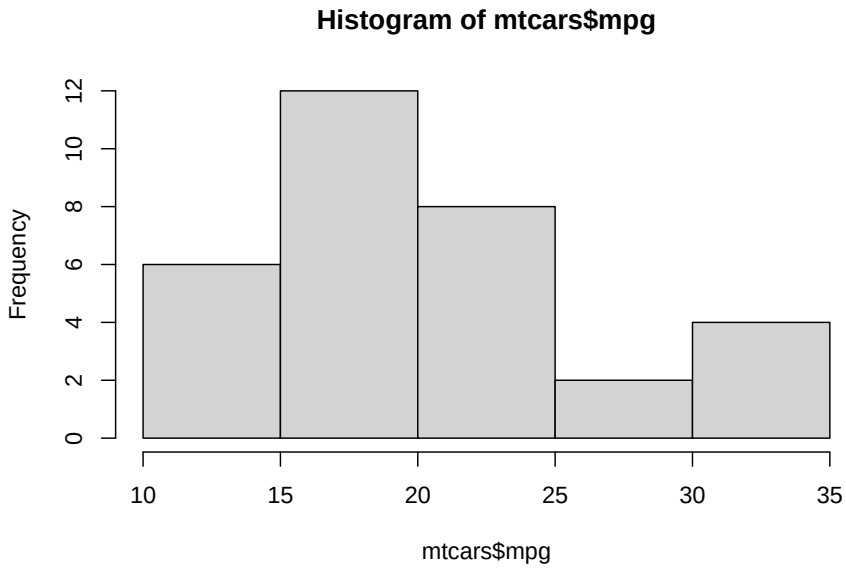
```
##
## 4 6 8
## 11 7 14
# o que significa que há 11 casos do valor 4, 7 casos do
# valor 6 e 14 casos do valor 8

# para gerar um gráfico em barra dessas quantias basta passar
# como argumento para a função barplot()
barplot(table(mtcars$cyl))
```



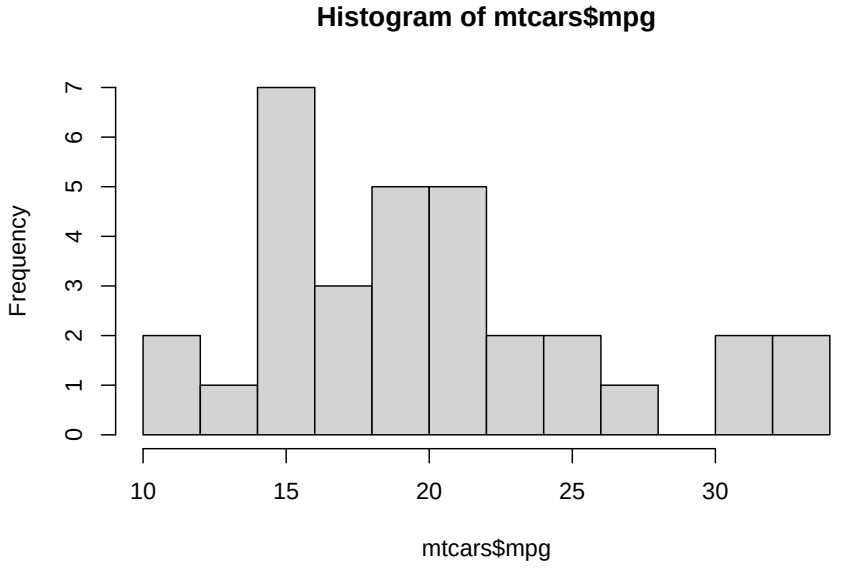
6.2.5 Criando histogramas

Para histogramas faz-se uso da função `hist()` passando como argumento o vetor de valores.

```
hist(mtcars$mpg)
```



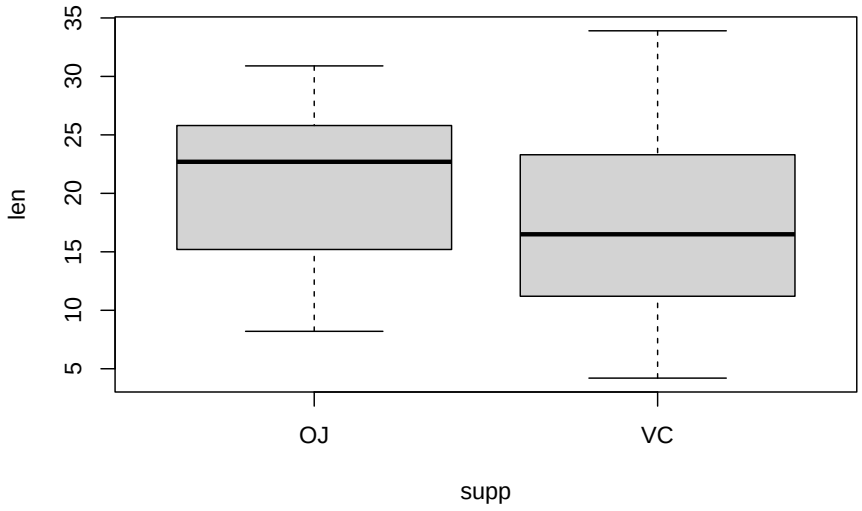

```
# para especificar o número de quebras usa-se o argumento
# breaks
hist(mtcars$mpg, breaks = 10)
```



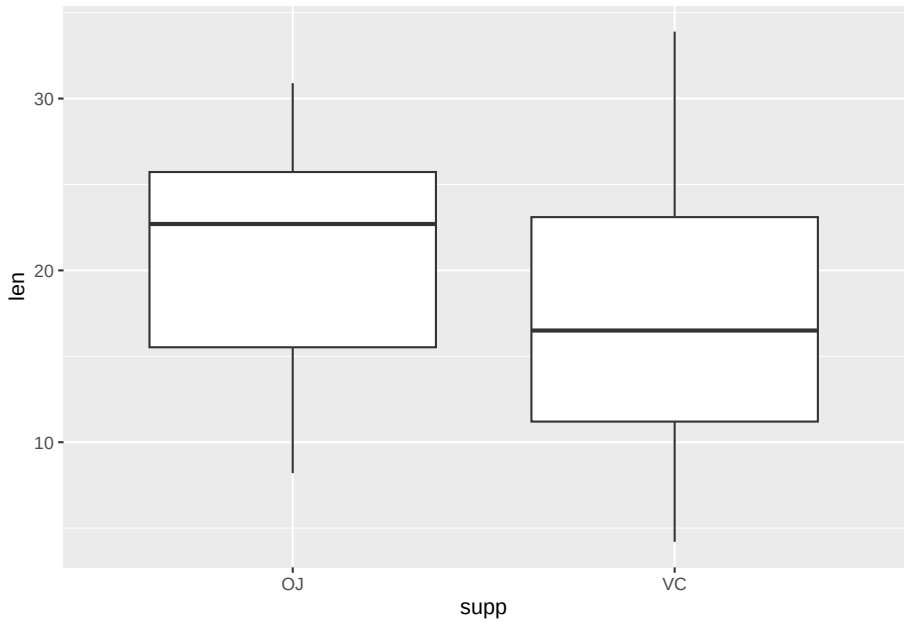
6.2.6 Criando boxplot

O gráfico do tipo boxplot é utilizado para comparar distribuições. Utiliza-se uma fórmula de sintaxe para combinar as variáveis a serem analisadas.

```
boxplot(len ~ supp, data = ToothGrowth)
```




```
# outra forma de obter o mesmo gráfico com o pacote ggplot2 é
qplot(supp, len, data = ToothGrowth, geom = "boxplot")
```

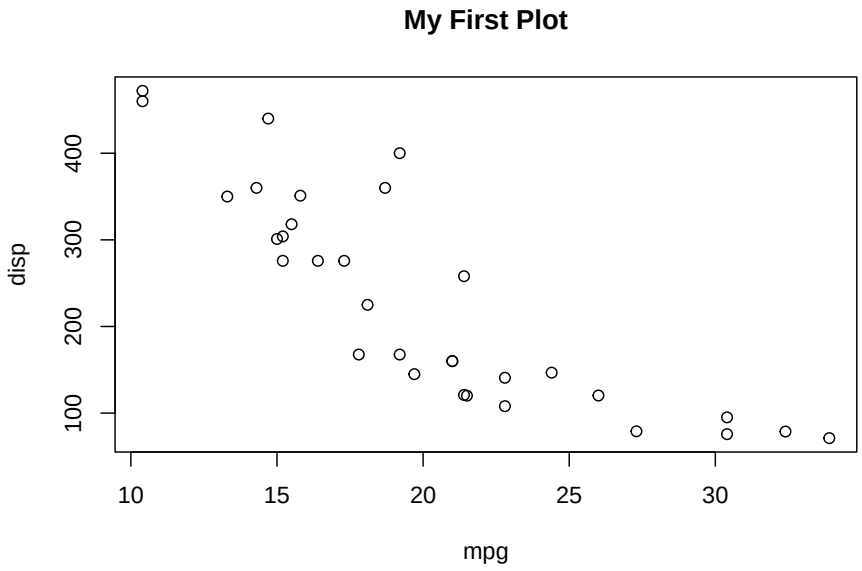


Ao longo desta primeira parte da aula pudemos observar que os gráficos em R são montados passo a passo. E que, caso se queira incrementar títulos, e especificar outros detalhes, isso pode ser feito com a adição de outros argumentos nas funções utilizadas para criar os gráficos. Para saber mais a respeito dos argumentos de cada função, basta fazer `?` seguido do nome da função ou através de `help("nomedafuncao")`.

```
# `?`(plot)
# help("plot")

data(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
```

```
plot(disp ~ mpg, data = mtcars, main = "My First Plot")
```



6.3 Lista de exercícios 01

1. Utilizando o dataset cars, construa um gráfico com a função `plot()`. Mostre a relação entre as variáveis distância (dist) e velocidade (speed). Como título do gráfico coloque "Relação entre distância e velocidade" e, para as labels do eixo x e y, "Velocidade (milhas por hora)" e "Distância percorrida (milhas)", respectivamente. Os pontos do gráfico devem ser vermelhos.

2. A função `rnorm()` gera desvios aleatórios. Usando a ajuda, veja a documentação para essa função e exemplos de utilização. Comece com `rnorm(10)` e veja o que é retornado. Em seguida, usando a função `hist()` plote um histograma para visualizar a distribuição destes desvios aleatórios, agora com `rnorm(1000)`.

3. Utilizando o dataset airquality, use a função head() para ver quantas colunas contém o data set. Para selecionar uma coluna específica do dataset, usamos a notação "$". Ou seja, se desejamos utilizar os dados contidos na camada "Ozone" devemos usar o seguinte comando airquality$Ozone. Plote um histograma da coluna Ozone contida neste dataset, fazendo uso da notação $.

4. Ainda utilizando o dataset airquality, agora vamos construir um boxplot. Plote um boxplot de Ozônio em função dos Meses, para o dataset airquality. Lembre-se de utilizar a sintaxe vista em aula `Ozone ~ Month`, ou seja,

uma em função da outra, da mesma maneira que fazemos, y dependente de x `(y ~ x)`, neste caso, fazendo Ozone dependente de Month.

5. No boxplot do exercício anterior, os nomes dos eixos x e y não foram especificados e logo, não foram mostrados no gráfico. Como vimos, os gráficos em R são montados em etapas, com a adição de argumentos nas funções. Utilizando a seta para cima do teclado, recupere o comando digitado anteriormente e adicione novos argumentos. Defina o argumento `xlab` igual à "Month" e o argumento `ylab` igual à "Ozone(ppb)". Além destes, coloque também os argumentos `col.axis` igual à "blue" e `col.lab` igual à "red" e veja o que acontece com o seu boxplot.

6.4 GGPLOT2

GGPlot2 é um pacote R disponível no repositório CRAN e pode ser instalado através da função install.packages(). Trata-se de uma implementação do conceito "A Gramática dosGráficos" criado por Leland Wilkinson e implementada por Hadley Wickham enquanto ele eraestudante de graduação na Universidade Estatal de Iowa. Uma gramática de gráficos representauma abstração de gráficos, ou seja, a teoria dos gráficos na qual conceitualiza peças básicas a partirdas quais você pode construir novos gráficos e objetos gráficos.

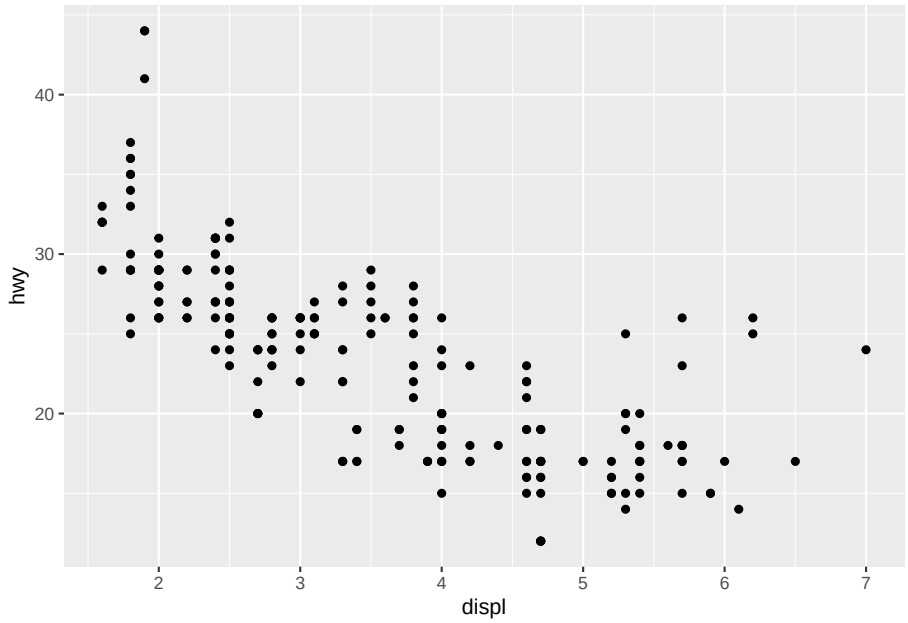
O pacote ggplot2 é composto por um conjunto de componentes independentes que podem ser compostos de muitas diferentes formas. Utilizando este pacote você pode criar gráficos que sejam precisamente adaptados para o seu problema. Os componentes utilizados para construir um gráfico incluem estéticas (aesthetics) as quais são atributos como cor; forma; tamanho e objetos geométricos (geoms) tais como pontos, linhas e barras.

O pacote GGPlot2 possui duas funções principais, sendo elas `qplot()` e `ggplot()`. A função `qplot()` funciona como a função `plot()` base do R. Com a função `qplot()` é possível criar muitos tipos de plots (como gráficos de pontos, histogramas, boxplot). Já a função `ggplot()` é mais avançada, a qual é mais flexível e pode ser customizada para fazer coisas que a `qplot()` não faz. Vamos começar vendo a utilização da função `qplot()`.

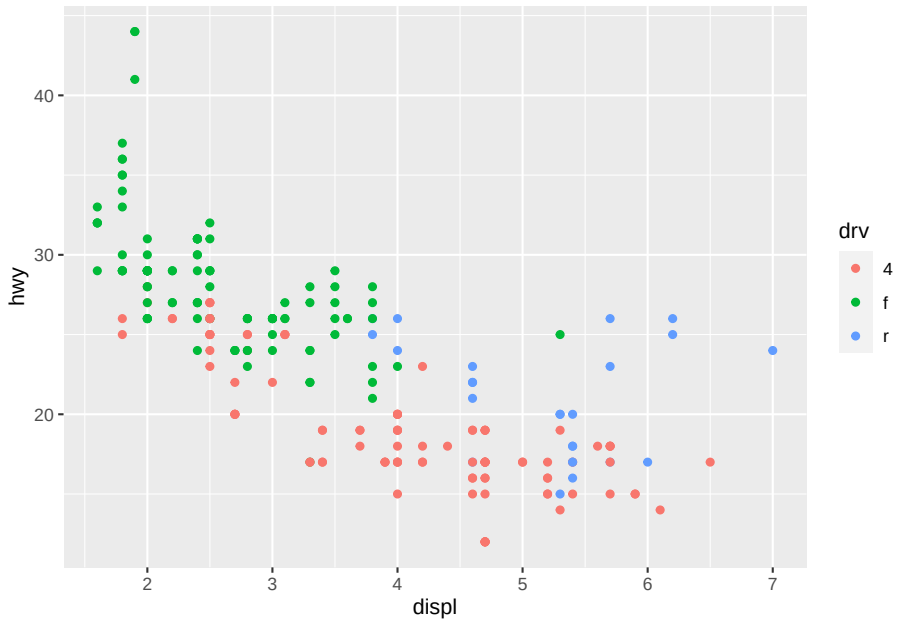
Para isso vamos fazer uso de um dataset que vem com o pacote ggplot2, o dataset mpg. Este contém dados econômicos de combustível para 38 modelos de carros manufaturados em 1999 e 2008.

```
library(ggplot2)
# A utilização da função ésimilar as funções base para a
# construção de gráficos Para montar um scatterplot, e
# analisar a relação entre variáveis do data set como por
# exemplo deslocamento do motor (displ) e milhas de rodovia
# por galão (hwy) precisamos passar essas duas variáveis como
```

```
# parâmetro na função e especificar o data set que contém
# estas variáveis
qplot(displ, hwy, data = mpg)
```

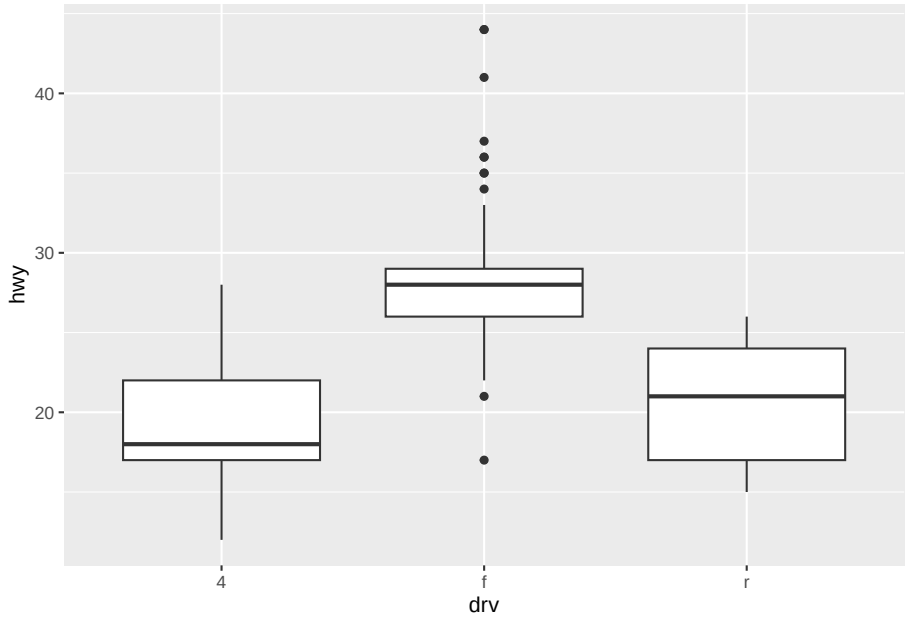



```
# Supondo que gostaríamos de utilizar diferentes cores para
# distinguir entre os 3 subconjuntos (fatores) tipos de
# movimentação (drv) bastaria colocar um novo argumento,
# conhecido como estética. Neste caso, cor, e definir igual a
# drv
qplot(displ, hwy, data = mpg, color = drv)
```



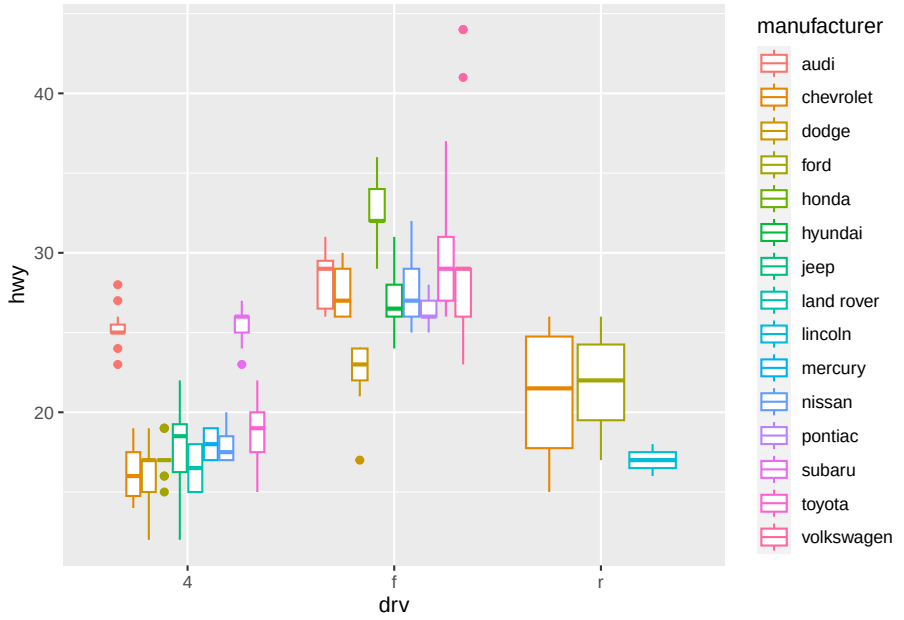
Utilizando o argumento geom (que se refere aos objetos geométricos) podemos especificar pontos, formas e até mesmo o tipo de gráfico que será plotado. Agora vamos construir um boxplot utilizando a função `qplot()`.

```
# Primeiro especificamos a variável na qual queremos dividir
# os dados (drv), para então em seguida, especificar a
# variável na qual queremos examinar, neste caso hwy. O
# terceiro argumento se refere ao dado (=mpg) e o quarto,
# geom definido como a string 'boxplot'
qplot(drv, hwy, data = mpg, geom = "boxplot")
```



Com o código acima pudemos visualizar 3 boxes, um para cada tipo de movimentação. No entanto, podemos definir com cores, para cada marca, basta colocar um outro argumento ao código anterior, color igual a manufacturer.

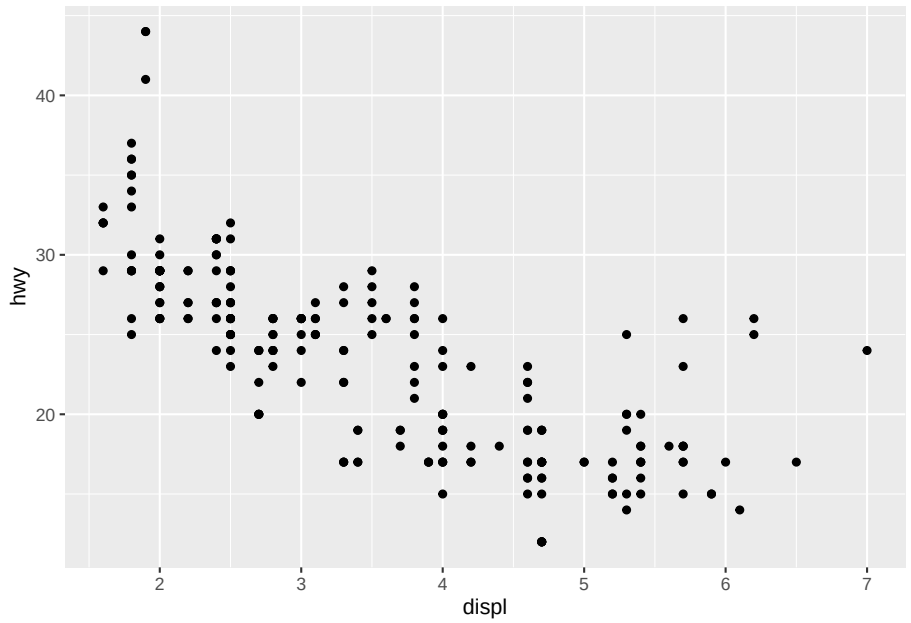
```
qplot(drv, hwy, data = mpg, geom = "boxplot", color = manufacturer)
```




```
# note que ainda temos 3 regiões nos gráficos,
# correspondentes aos 3 tipos de movimentação, e em cada
# temos subdivisões de acordo com as marcas
```

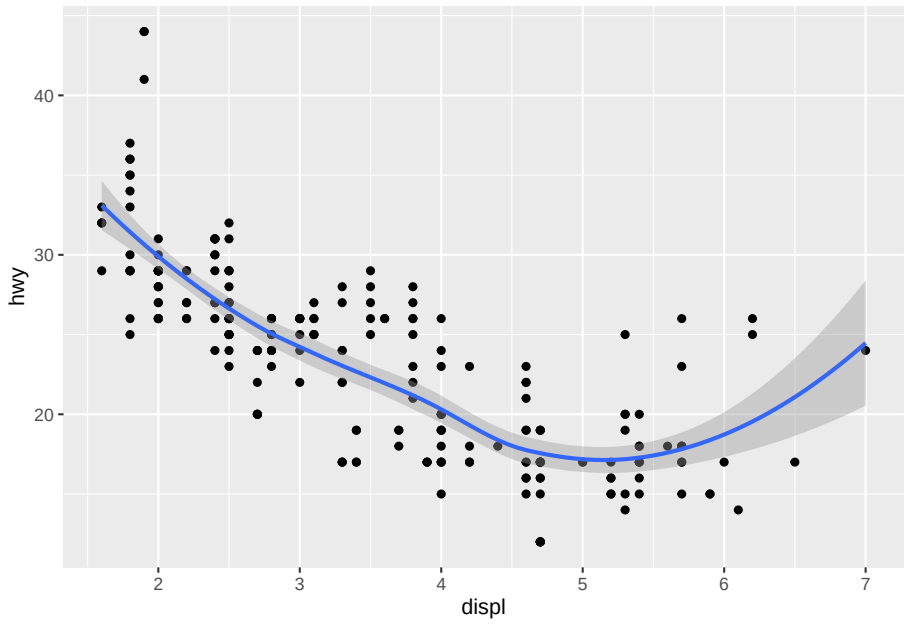
Agora depois de ver um pouco sobre a função `qplot()` iremos focar em alguns componentes fundamentais do pacote ggplot2. Para montar gráficos utilizando a função `ggplot()` vamos fazer em uma série de etapas. Isto com o intuito de analisar o que está acontecendo em cada passo até a obtenção do gráfico final. Primeiro iremos criar uma variável "g" e designar à esta uma chamada a função ggplot com 2 argumentos. O primeiro se refere a mpg (nosso data set em questão) e o segundo dirá a ggplot o que queremos plotar, que neste caso são as variáveis displ e hwy. Como estas se tratam das estéticas que queremos representar, passamos as mesmas como argumentos na função `aes`.

```
g <- ggplot(mpg, aes(displ, hwy))

# o que ggplot fez foi criar um objeto gráfico o qual
# designamos àvariável g O pacote ggplot2 precisa saber como
# os dados serão visualizados, logo épreciso especificar.
# Para criar um gráfico de pontos, podemos fazer por exemplo

g + geom_point()
```




```
# ao fazer chamada para a função geom_point() foi adicionada
# uma camada ao objeto gráfico. Perceba que você nao precisou
# passar nenhum argumento para a função geom_point(), isto
# porque o objeto g já possui todo o dado armazenado nele.
# Vamos agora adicionar outra camada, fazendo uma chamada
# para a função geom_smooth

g + geom_point() + geom_smooth()
```

```
## `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```

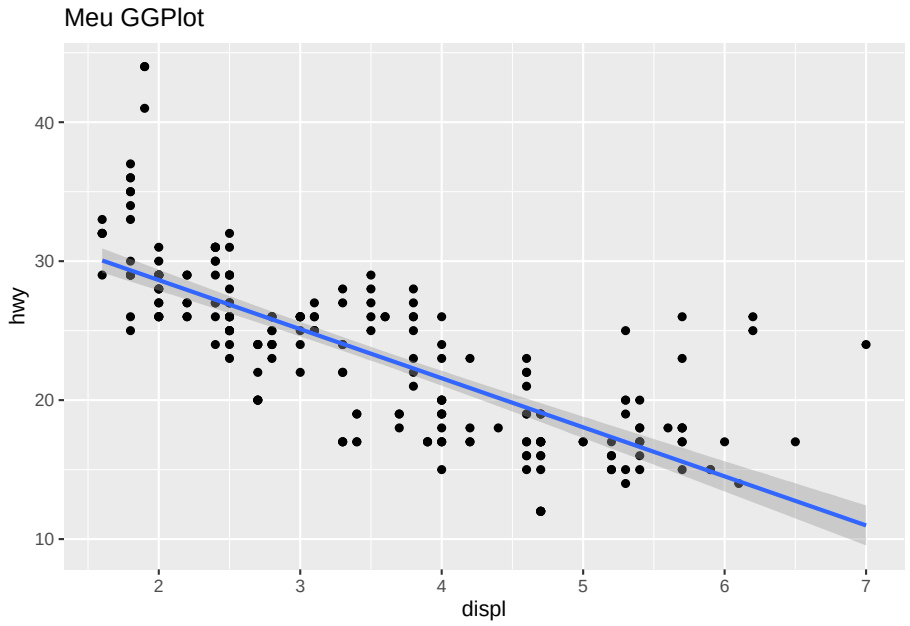



```
# A sombra cinza ao redor da linha azul éo intervalo de
# confiança. Épossível utilizar uma função diferente de
# suavização, definindo método igual àlm (method = 'lm')
```

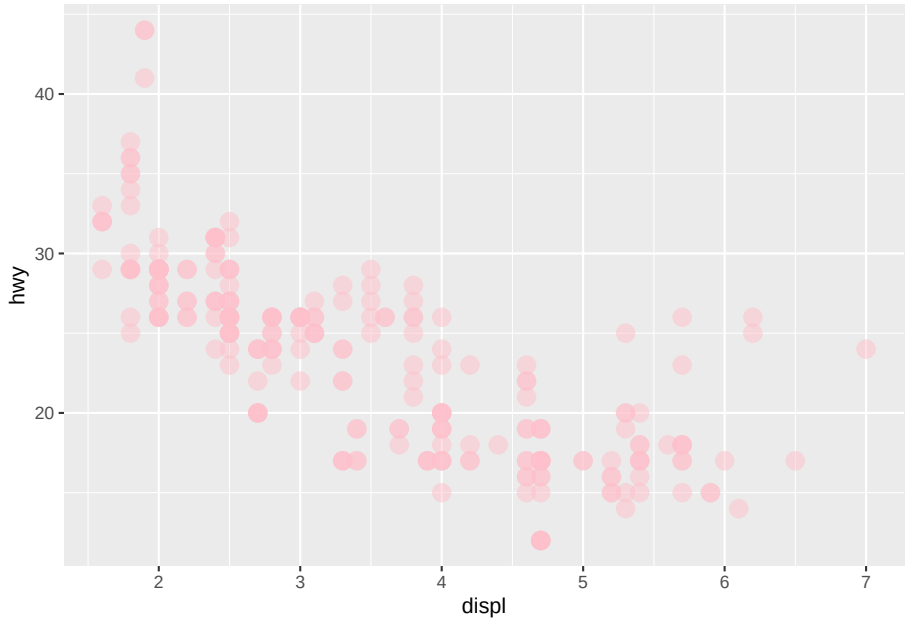
Além das definições padrões, podemos adicionar outras anotações utilizando funções como `xlab()`, `ylab()` e `ggtitle()`. Vamos fazer um exemplo adicionando um título ao gráfico. Como por exemplo "Meu GGPlot".

```
g + geom_point() + geom_smooth(method = "lm") + ggtitle("Meu GGPlot")
```

```
## `geom_smooth()` using formula = 'y ~ x'
```

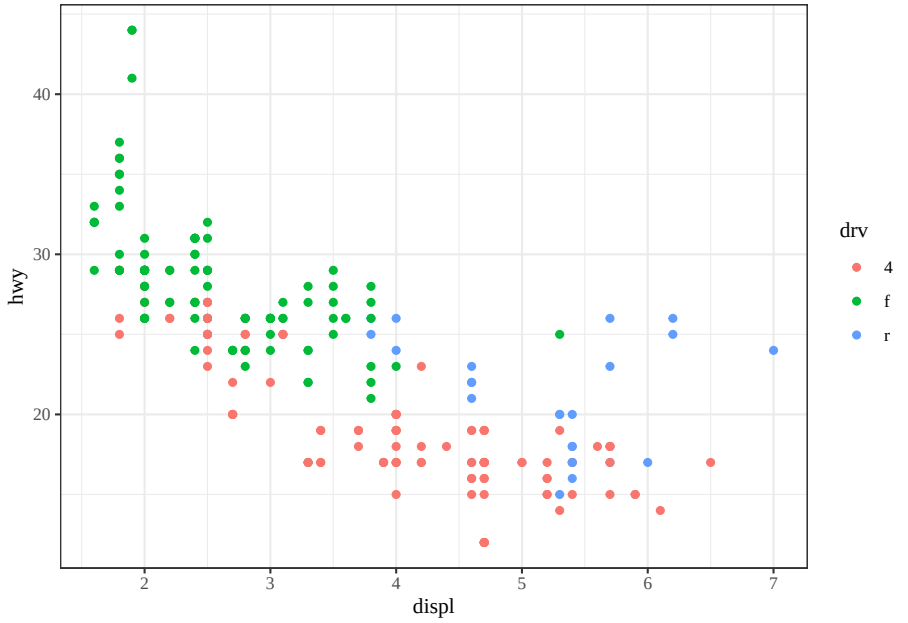



```
# Podemos mudar outras estéticas do gráfico, ao definir cores
# e tamanhos, por ex:
g + geom_point(color = "pink", size = 4, alpha = 1/2)
```



```
# os diferentes tons de rosa são resultado do argumento
# alpha, que diz o quão transparentes os pontos precisam ser.
# Os círculos mais escuros indicam que há múltiplos pontos
```

```
g + geom_point(aes(color = drv)) + theme_bw(base_family = "Times")
```



6.4.1 Exemplos com os dados de tumor de mama de Wiscosin

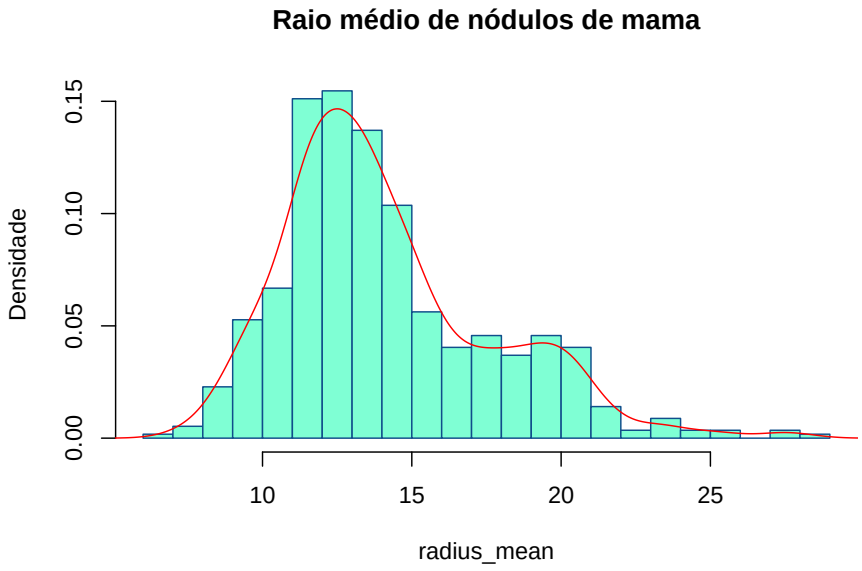
Nos exemplos abaixo, iremos utilizar novamente o data set com informações sobre nódulos de mama de Wiscosin. Vamos gerar gráficos utilizando as funções básicas do R e do GGplot. Alteramos algumas variáveis dos gráficos para apresentar algumas possibilidade de customização.

```
# Vamos baixar os dados de características de nódulos de mama de Wiscosin
# Mais informações sobre esse dataset em https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wi
download.file(url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisc

# Vamos importar os dados para o ambiente do R.
# Os dados estão separados por vírgula, por isso o argumento sep=','
# outras opções e argumentos podem ser vistos em help(read.table)
brca <- read.table(file = "./wbdc.data", sep = ',')
```

```
colnames(brca) <- c('id_number', 'diagnosis', 'radius_mean',
                    'texture_mean', 'perimeter_mean', 'area_mean',
                    'smoothness_mean', 'compactness_mean',
                    'concavity_mean','concave_points_mean',
                    'symmetry_mean', 'fractal_dimension_mean',
                    'radius_se', 'texture_se', 'perimeter_se',
                    'area_se', 'smoothness_se', 'compactness_se',
                    'concavity_se', 'concave_points_se',
                    'symmetry_se', 'fractal_dimension_se',
                    'radius_worst', 'texture_worst',
                    'perimeter_worst', 'area_worst',
                    'smoothness_worst', 'compactness_worst',
                    'concavity_worst', 'concave_points_worst',
                    'symmetry_worst', 'fractal_dimension_worst')
brca$diagnosis <- as.factor(brca$diagnosis)
# Adicionar coluna com a razão do perímetro pelo raio dos nódulos
brca$perimeter.radius <- brca$perimeter_mean/brca$radius_mean
```

Histogramas

```
# Pacote básico
# Em hist() por padrão usa-se contagem.
# Para densidade, mudar argumento freq = F
hist(brca$radius_mean, breaks = 20, col = "aquamarine",
     border = "dodgerblue4",
     xlab = "radius_mean", ylab = "Densidade",
     main = "Raio médio de nódulos de mama",
     freq = F)
# Adicionar linha com a densidade
lines(density(brca$radius_mean), col = "red")
```



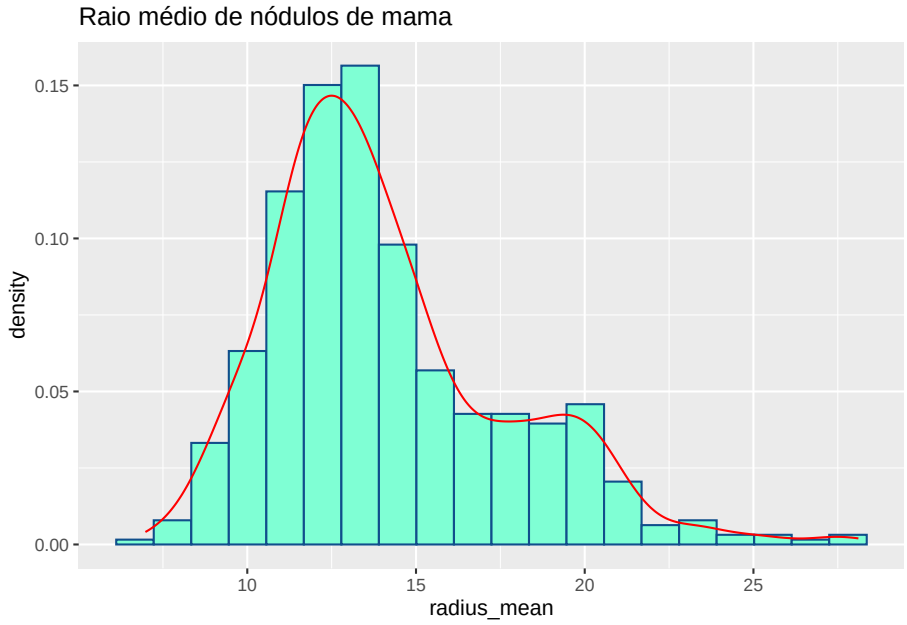

```
# Salvar gráfico em PDF
# https://r-coder.com/save-plot-r/
# Nesse link tem outras opções (ex.; svg, pdf, jpeg, bmp, tiff)
pdf(file = "./Meuplot.pdf")
hist(brca$radius_mean, breaks = 20, col = "aquamarine",
     border = "dodgerblue4",
     xlab = "Raio médio", ylab = "Densidade",
     main = "Raio médio de nódulos de mama", freq = F)
lines(density(brca$radius_mean), col = "red")
dev.off()
```

```
## pdf
##   2
```

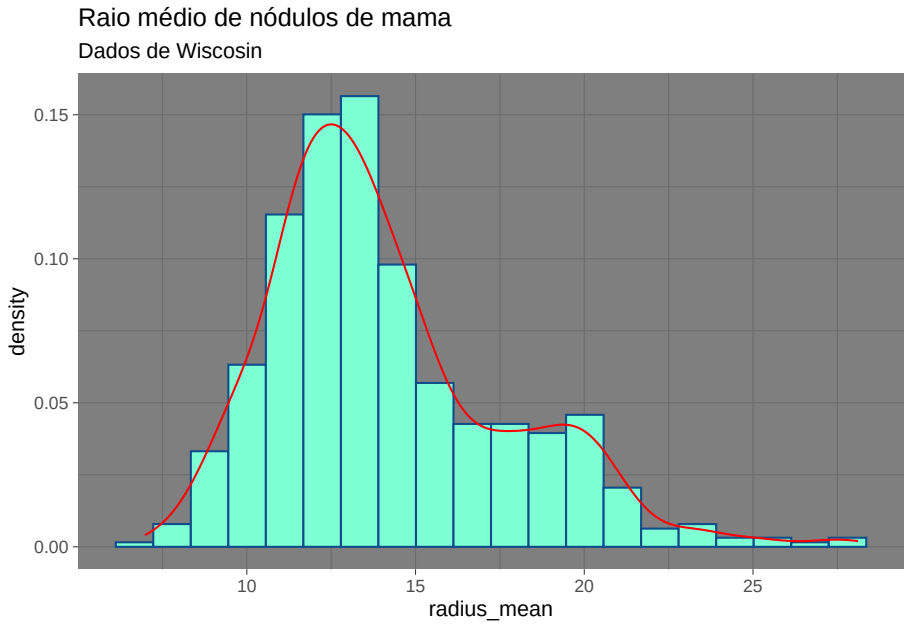
```
# GGplot2
g <- ggplot(data = brca, aes(radius_mean)) +
  geom_histogram(bins = 20, color = "dodgerblue4", fill = "aquamarine",
                 aes(y = ..density..)) +
  geom_density(color = "red") +
  labs(title = "Raio médio de nódulos de mama")
g
```

```
## Warning: The dot-dot notation (`..density..`) was deprecated in ggplot2 3.4.0.
## i Please use `after_stat(density)` instead.
```

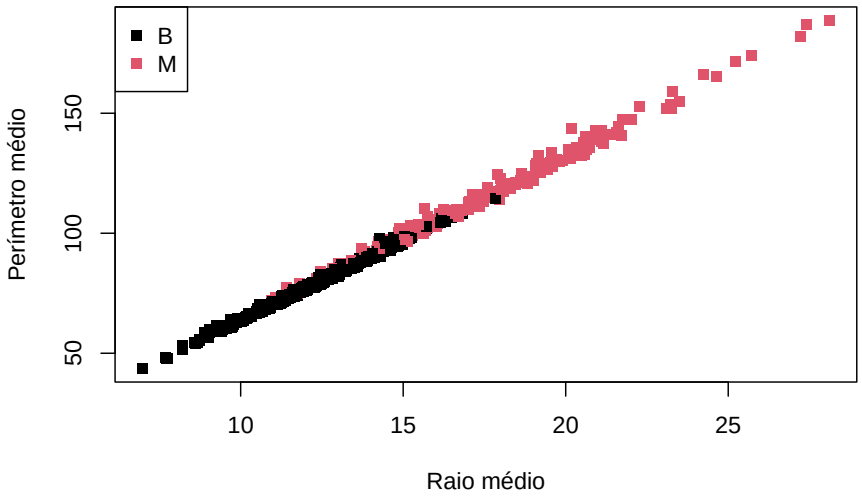
```
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```




```
# Mudando o tema e adicionando subtítulo
g + theme_dark() +
  labs(subtitle = "Dados de Wiscosin")
```




```
# R base
plot(brca$radius_mean, brca$perimeter_mean, pch = 15,
     xlab = "Raio médio", ylab = "Perímetro médio",
     col = brca$diagnosis)
# Adicionar legenda
# https://statisticsglobe.com/add-legend-to-plot-in-base-r
legend("topleft", legend = c("B","M"),
       col = 1:2, pch = 15)
```




```
# GGplot2
# Criando variável categórica para area
# Vamos utilizá-la para alterar a forma dos pontos no gráfico
summary(brca$area_mean)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   143.5   420.3   551.1   654.9   782.7  2501.0
```

```
brca$Area <-
  ifelse(brca$area_mean < 420,
         "Baixo",
         ifelse(brca$area_mean < 750, "Intermediário", "Alto"))
brca$Area <- factor(brca$Area,
                    levels = c("Baixo", "Intermediário", "Alto"))

# Agora os comandos para o gráfico em si
ggplot(data = brca,
       aes(x= radius_mean, y = perimeter_mean,
           color = diagnosis, shape = Area))+
  geom_point()+
  labs(x = "Raio médio",
       y = "Perímetro médio",
       color = "Diagnóstico",
```

```
      shape = "Área")+
  scale_fill_discrete(labels = c("Benigno", "Maligno"))
```

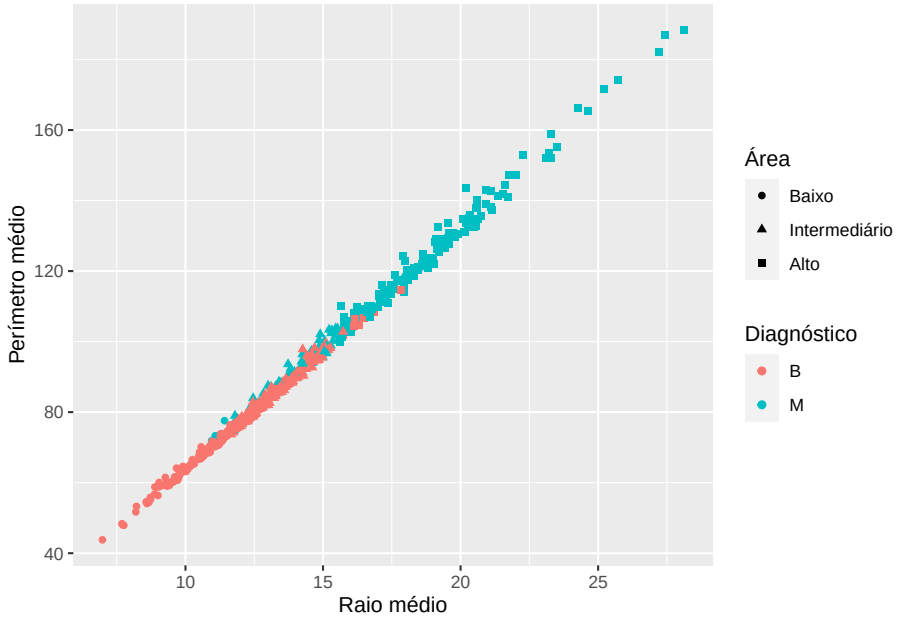
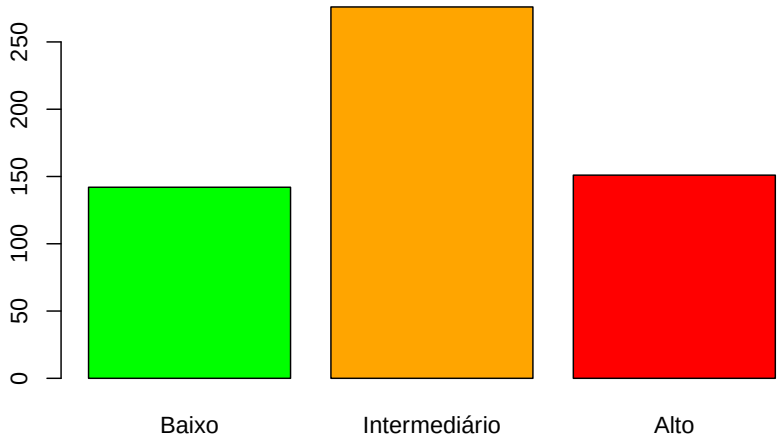


Gráfico de barras

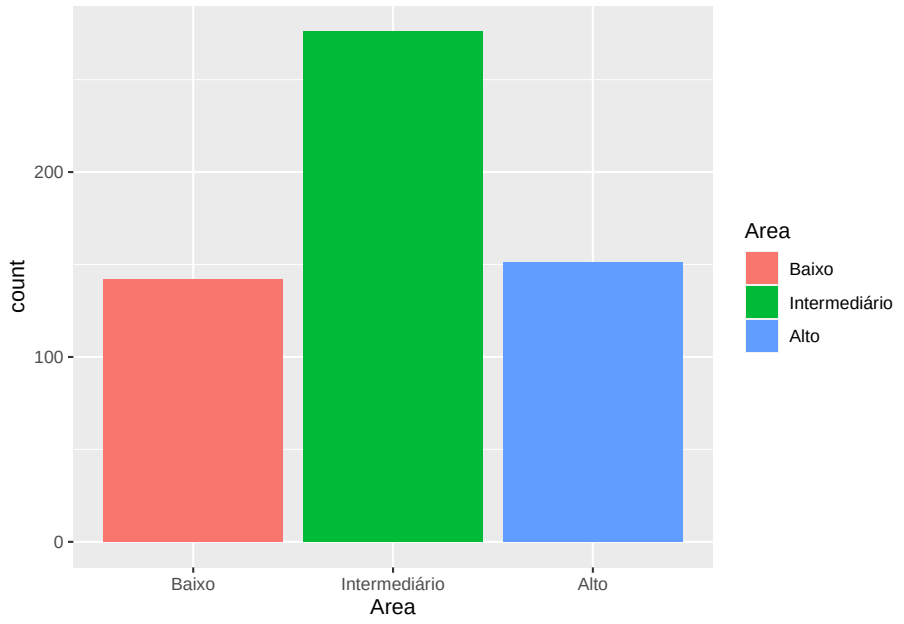
```
# Gráfico de barras
# http://www.sthda.com/english/wiki/bar-plots-r-base-graphs

# R base
table(brca$Area)
```

```
##
##         Baixo Intermediário          Alto
##           142           276           151
```

```
counts <- table(brca$Area)
barplot(counts, col = c("green", "orange", "red"))
```

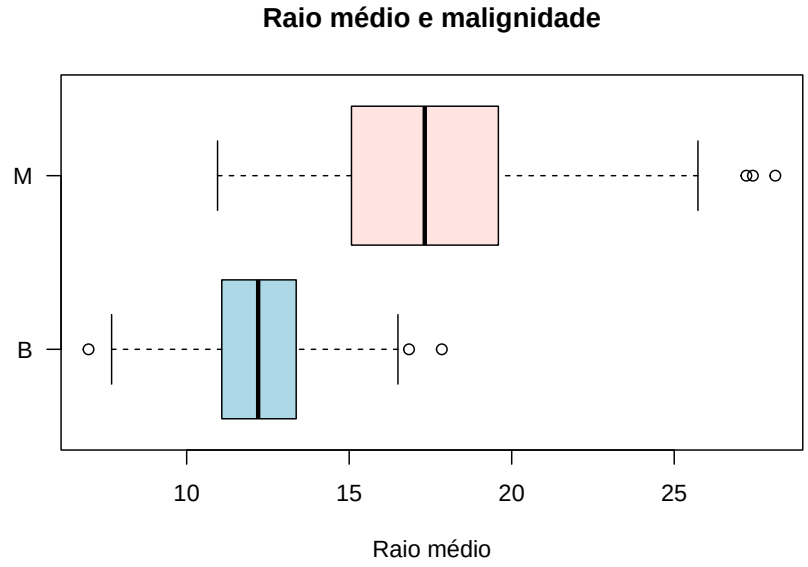



```
# Ggplot2
# http://www.sthda.com/english/wiki/ggplot2-barplots-quick-start-guide-r-software-and-
ggplot(brca,
       aes(Area)) +
  geom_bar(stat="count", aes(fill=Area))
```

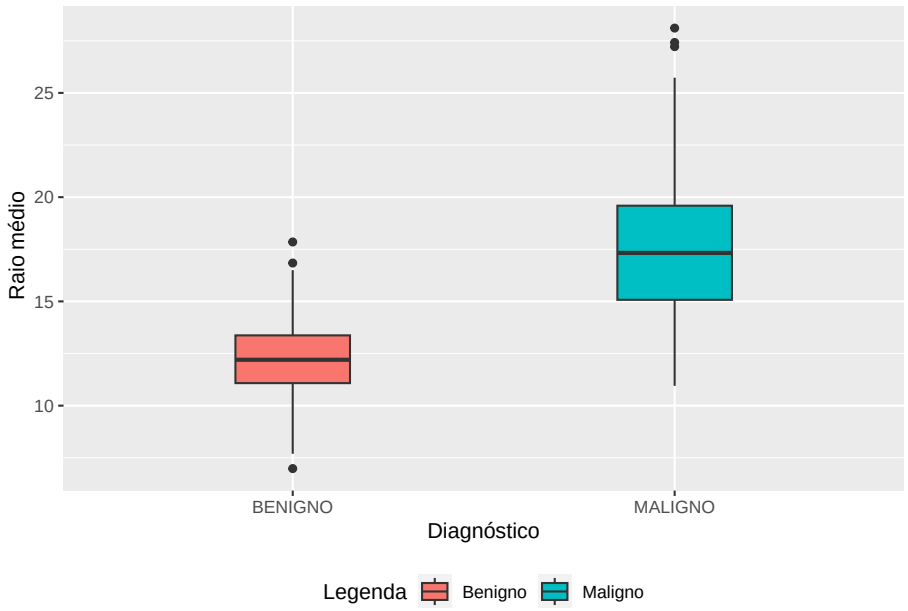


Boxplot

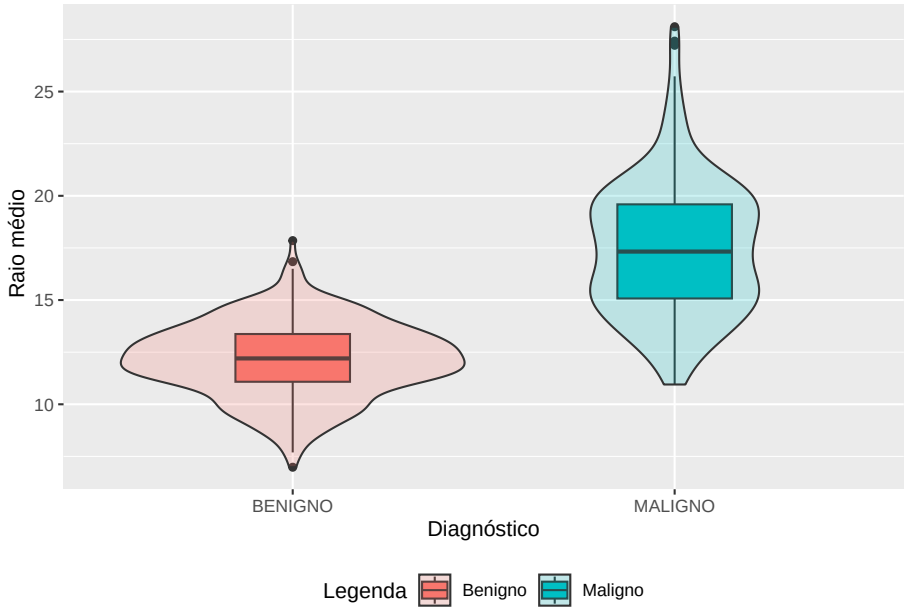
```
# R base
# https://statisticsglobe.com/boxplot-in-r
# '~' éum operador para fórmular no R
# Significa "depende de"
# A aplicação évar_dep ~ var_ind
boxplot(brca$radius_mean ~ brca$diagnosis,
        horizontal = T, las = 1,
        main = "Raio médio e malignidade",
        col = c("lightblue", "mistyrose"),
        xlab = "Raio médio", ylab = "")
```




```
# GGplot2
# Criar boxplot e ajustar legenda
# https://r-charts.com/ggplot2/legend/
g <- ggplot(data = brca,
            aes(x= diagnosis, y = radius_mean, fill = diagnosis))+
  geom_boxplot(width = 0.3) +
  labs(x = "Diagnóstico",
       y = " Raio médio",
       fill = "Legenda") +
  scale_x_discrete(labels = c("BENIGNO", "MALIGNO")) +
  scale_fill_discrete(labels = c("Benigno", "Maligno"))+
  theme(legend.position = "bottom")
g
```



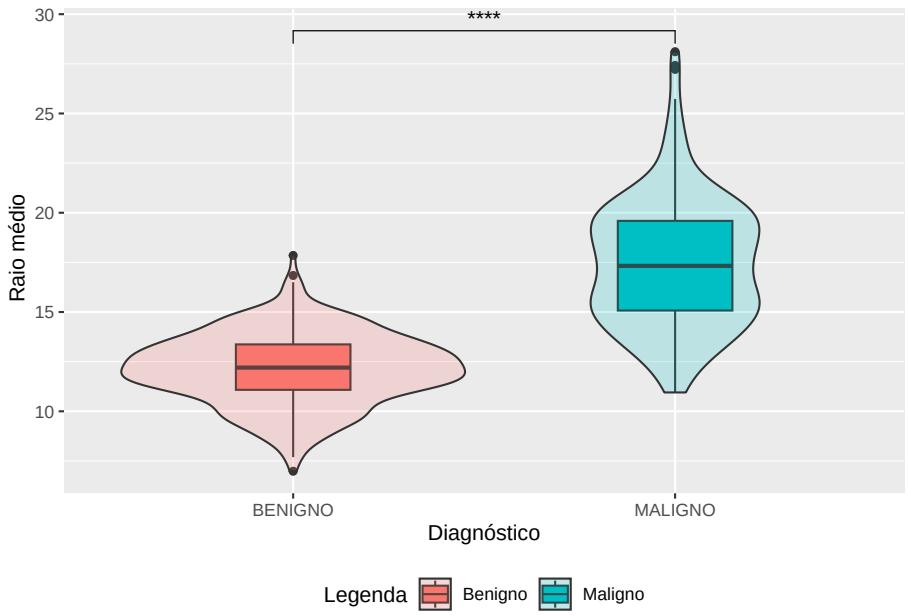

```
# Adicionar violion plot
g <- g + geom_violin(alpha = 0.2)
g
```



```
# Para adicionar teste estatístico
# http://sthda.com/english/articles/24-ggpubr-publication-ready-plots/76-add-p-values-
# Instalar pacote ggpubr caso não esteja instalado
install.packages("ggpubr")
```

```
## Installing package into '/home/jeanresende/R/x86_64-pc-linux-gnu-library/4.3'
## (as 'lib' is unspecified)
```

```
# Carregar pacote
library("ggpubr")
# Adicionar teste estatístico ao gráfico
g +
stat_compare_means(method = "t.test", label = "p.signif",
comparisons = list(c("B", "M")))
```



6.5 Bonus track - Gráficos interativos com plotly

O pacote plotly é uma forma bastante simples de se produzir gráficos interativos. Ele apresenta uma sintaxe própria que pode ser aprendida nas descrições das funções, no site oficial do pacote (https://plotly.com/r/) ou em vários tutoriais online. Aqui vamos apenas apresentar uma facilidade que o pacote traz a qual é sua compatibilidade com gráficos gerados pelo ggplot2. A função ggplotly()

recebe um gráfico do ggplot e torna-o interativo. Essa função é compatível com a maior parte dos recursos do ggplot, apesar de existirem algumas poucas exceções. Além disso, os gráficos interativos gerados pelo plotly são compatíveis com o pacote shiny, podendo ser usado em dashboards e web apps. Abaixo vamos apresentar 2 exemplos simples transformando gráficos gerados com o ggplot em interativos.

```
#install.packages("plotly")
library(plotly)
```

```
##
## Attaching package: 'plotly'
```

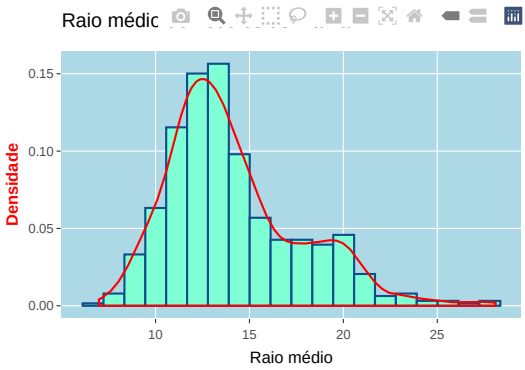
```
## The following object is masked from 'package:ggplot2':
##
##     last_plot
```

```
## The following object is masked from 'package:stats':
##
##     filter
```

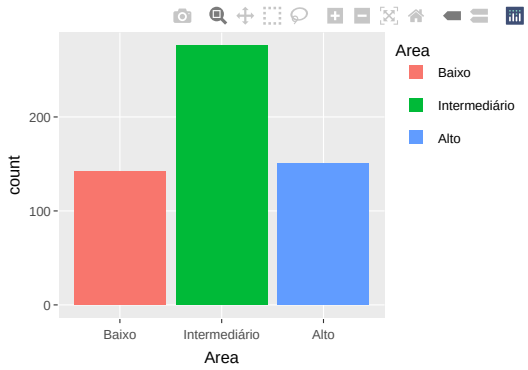
```
## The following object is masked from 'package:graphics':
##
##     layout
```

```
# Criando um histograma com ggplot
g<-ggplot(data = brca, aes(radius_mean)) +
  geom_histogram(bins = 20, color = "dodgerblue4", fill = "aquamarine",
                 aes(y = ..density..)) +
  geom_density(color = "red") +
  labs(title = "Raio médio de nódulos de mama",
       x = "Raio médio",
       y = "Densidade",
       subtitle = "Dados de Wiscosin")+
  theme(axis.title.y = element_text(face = "bold", color= "red"),
        panel.background = element_rect(fill = "lightblue"))

# Tornando-o interativo:
ggplotly(g)
```

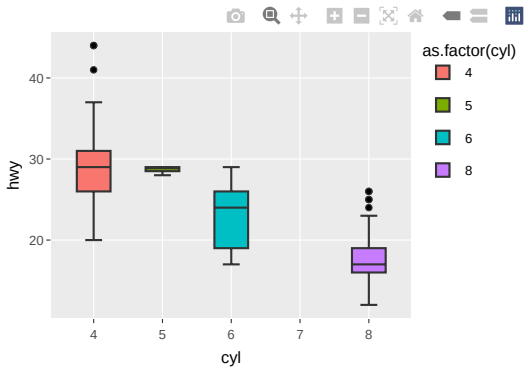
```
## PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed,
```




```
# Barplot
ggplotly(ggplot(brca,
                aes(Area)) +
           geom_bar(stat="count", aes(fill=Area)))
```




```
# Você pode testar com os demais exemplos que criamos na seção anterior
# Crie seu gráfico no ggplot e insira dentro da função ggplotly.
# Perceba que as funções do ggpubr (de comparação estatística)
# ainda não são suportados pelo plotly

ggplotly(ggplot(mpg, aes(x=cyl, y=hwy, fill=as.factor(cyl), group=cyl))+
           geom_boxplot())
```




```
summary(as.factor(mpg$cyl))
```

```
##  4  5  6  8
## 81  4 79 70
```

6.6 Lista de exercícios 02

1. Utilizando o data set mpg: como você plotaria a relação entre as variáveis cty (a milhagem média da cidade) e hwy (a quilometragem média da rodovia)? Como você descreveria essa relação?

2. Para adicionar variáveis a um gráfico precisamos mapeá-las em estéticas. Em duas dimensões, podemos usar os eixos x e y. Para adicionar uma terceira (ou quarta, etc) variável precisamos utilizar estéticas tais como forma, cor e tamanho. Utilizando o data set mpg, plote um gráfico de pontos usando a função ggplot, mostrando a relação entre as variáveis displ, hwy e drv. Dica: como se tratam de mais de duas variáveis, use a estética shape para representar drv.

3. Continuando a usar o data set mpg, vamos primeiro ver como fica a distribuição dos dados de milhas rodadas na cidade por galões de combustível (cty). Para isso, crie um histograma básico com a variável cty, use o argumento breaks = 10. Os valores se concentram ao redor de qual número?

4. Agora vamos investigar como o consumo urbano se comporta dependendo da classe do carro. Crie um boxplot onde o eixo x serão as classes dos carros (class) e no eixo y a quantidade de milhas que o carro faz por galão de combustível (cty). Valores mais altos em cty indicam carros que gastam menos combustível. Quais as 2 classes mais econômicas na cidade (maiores valores em cty)? Quais as 2 classes que mais consomem combustível na cidade (menores valores em cty)?