

Hands-on Linguagem R básica para Análise de Dados

Parte 02-1

1. **R-GUI e R-Studio (1)**
2. **Biblioteca Cran-R (2)**
3. **Estrutura da Linguagem (3)**
 - 3.1 Objetos R (3.1)
 - 3.2 Operações simples com objetos (3.2)
 - 3.3 Criando funções (3.3)
 - 3.4 Concatenação e Arrays (3.4)
 - 3.5 Listando e removendo objetos (3.5)
 - 3.6 Geração de Sequências (3.6)
 - 3.7 Indexação de Objetos (3.7)
 - 3.8 Data frames (3.8)
 - 3.9 Data Querying e estatística base (3.9)
 - 3.10 Lidando com dados não disponíveis (3.10)
 - 3.11 Lendo dados de um dataset em arquivo (3.11)
 - 3.12 Salvando um dataset em arquivo (3.12)
 - 3.13 Funções estatísticas básicas (3.13)
 - 3.14 Estruturas de programação (3.14)
4. **Avaliando uma análise de dados em R com um modelo de árvore de decisão (4)**

1. R-GUI e R-Studio

R pode ser baixado do site: <http://www.R-project.org>, usando o link do CRAN (Comprehensive R Archive Network) abaixo do cabeçalho. Selecione a localização mais próxima de você e então escolha a versão/tipo de sistema operacional que você utiliza: Windows, Mac ou Linux. Selecione o pacote base de acordo com o seu sistema e após baixa-lo faça a instalação seguindo os passos apresentados nas caixas de diálogo d o instalador. Quando a instalação terminar, dê um clique duplo no ícone 'R' que aparecerá no desktop – você ativará o R-GUI, que é o ambiente padrão para desenvolvimento em R.

Um outro ambiente para desenvolvimento que é muito utilizado é o R-Studio. Atualmente ele não faz parte do pacote de instalação da linguagem e deve ser baixado separadamente e possui licenças comerciais para uso.

2. Biblioteca Cran-R

A CRAN-R (Comprehensive R Archive Network) é um repositório de pacotes/bibliotecas para a linguagem R.

A instalação de pacotes é feita através da execução da função `install.package()`. Por exemplo:

```
> install.packages("dplyr")  
>install.packages("rpart")  
>install.packages("rpart.plot")
```

Ao executar a função acima, será feito o download do pacote diretamente de um site espelho da CRAN-R que deve ser selecionado anteriormente no ambiente de trabalho. Caso haja a necessidade de instalar outros pacotes da lista de dependência de uma dada biblioteca, isto será feito automaticamente pelo `install.package()`.

Se você quiser ver alguns datasets que foram disponibilizados no pacote recém instalado, basta utilizar a função `data()`.

```
> data(package="dplyr")
```

Para utilizar um pacote específico, deixando-o disponível na sua seção de trabalho, basta utilizar a função `library()`. Observe que não se deve utilizar as aspas (""):

```
> library(dplyr)
```

As principais fontes de informação sobre a linguagem R estão nos seguintes endereços:

<https://www.rdocumentation.org>

<https://www.r-project.org/other-docs.html>

3. Estrutura da Linguagem

3.1 Objetos R

Podemos dizer que quase tudo em R é um objeto (variáveis, instruções, funções, etc). Entretanto, os objetos desta linguagem possuem uma característica própria: eles são não-transientes na seção. Antes de explicar o que isso significa, vamos criar um objeto:

```
> d <- 30
```

No exemplo acima, criamos o objeto "d" e atribuímos a ele o valor 30. O sinal de maior (>) no início da linha representa o prompt de comando da linguagem e o sinal de menor com um traço (<-) representa o sinal de atribuição de valor.

Esta é uma linguagem para análise estatística de dados e possui muitos facilitadores. Por exemplo, caso você queira ver o valor de um determinado objeto, digite no prompt o nome do objeto e tecle ENTER.

```
> d
```

```
[1] 30
```

A saída acima significa que o objeto d é igual a 30 e possui apenas uma única parte de dados armazenada nele. Este conceito é importante porque um objeto pode ter uma quantidade variada de dados. Em R, você pode carregar um dataset inteiro em um objeto e depois realizar as devidas análises sobre ele. Agora vamos ver outro exemplo:

```
> b <- "Bom dia"
```

Acabamos de criar o objeto `b` com o valor “Bom dia”, uma string. Observe que utilizamos as aspas (”) para limitar o conteúdo da string. Se digitarmos no prompt “`b`”, teremos o seguinte resultado:

```
> b
```

```
[1] "Bom dia"
```

Agora, se fizermos o seguinte:

```
> m <- b
```

```
> m
```

```
[1] "Bom dia"
```

Teremos o objeto “`m`” com o valor “Bom dia” que é o mesmo valor armazenado no objeto “`b`”.

Em R o nome dos objetos devem ser iniciados por uma letra, podem conter números, ponto parágrafo ou sublinhado. Os nomes são *case sensitive*. Exemplo de nomes válidos de objetos:

```
> a..b <- 4
```

```
> nom_cliente <- "Fulano"
```

```
Y1 <- a..b + 3
```

```
y1 <- "mínimo"
```

3.2 Operações simples com objetos

A linguagem R é dotada de todas as funções matemáticas desde as mais básicas até as mais complexas. Para ilustrar, podemos começar com os seguintes objetos:

```
> dist1 <- 15
```

```
> dist2 <- 20
```

```
> dist1 + dist2
```

```
[1] 35
```

Ao realizarmos no prompt a operação `dist1 + dist2` pudemos observar o resultado igual a 35. As operações aritméticas básicas podem ser realizadas com os símbolos já conhecidos na programação: adição (+), subtração (-), multiplicação (*) e divisão (/). Então, se fizermos:

```
> dist2-dist1
```

```
> [1] 5
```

A seguir temos uma lista básica das principais funções para a realização de outras operações matemáticas na linguagem R:

Função Matemática	Função R
Raiz quadrada de x	<code>sqrt(x)</code>
X elevado a y	<code>X^y</code>
Exponencial de x	<code>exp(x)</code>
Logaritmo natural de x na base y	<code>log(x,y)</code>
Fatorial de x	<code>factorial(x)</code>
Seno de x em radianos	<code>sin(x)</code>
Cosseno de x em radianos	<code>cos(x)</code>

Tangente de x em radianos	<code>tan(x)</code>
Valor absoluto/módulo de x	<code>abs(x)</code>
Média das linhas do data frame x	<code>rowMeans(x)</code>
Soma das colunas do data frame x	<code>colMeans(x)</code>
Soma das linhas do data frame x	<code>rowSums(x)</code>

Na tabela acima temos um novo conceito que será explicado mais adiante: data frame!

Se quisermos testar um exemplo de uso de outras operações matemáticas, podemos fazer o seguinte:

```
> sqrt(256)
[1] 16
> factorial(4)
[1] 24
> log(256,2)
[1] 8
```

3.3 Criando funções

Anteriormente utilizamos algumas funções matemáticas para fazermos alguns cálculos. Em R, funções são objetos criados com pedaços de códigos organizados para realizar uma operação desejada com base em parâmetros e que retornam valores resultantes para quem acionou a função. Tomemos os seguintes exemplos a seguir:

```
> somador <- function(a,b,c) {
+ result <- a+b+c
+ print(result)
+ }
> diminuidor <- function(a,b,c) {
+ a-b-c
+ }
```

Foram criadas no prompt de comando duas novas funções: somador e diminuidor. Ambas esperam receber 3 parâmetros (a,b,c) para realizarem respectivamente a soma de 3 números e a subtração de 3 números. Olhando de perto temos o seguinte: as funções possuem em princípio um nome (somador e diminuidor) um símbolo de atribuição de objeto e o marcador function seguido pela lista de parâmetros entre parêntesis. O código da função deve estar delimitado por chaves que marcam o início e o fim do corpo do código. Nos dois exemplos podemos observar que as funções realizam o retorno do valor calculado de forma diferente: a primeira de forma declarativa (result) e a segunda de forma implícita. Isto é comum na linguagem R. Estas facilidades buscam agradar quem tem origem na programação e quem tem origem nas planilhas e processadores matemáticos/estatísticos. Vamos para o teste das funções. Experimente digitar as funções e depois testá-las como no exemplo a seguir:

```
> somador(1,2,3)
```

```
[1] 6
> soma <- somador(2,3,6)
> soma
[1] 11
> diminuidor(3,2,1)
[1] 0
> x <- diminuidor(4,2,1)
> x
[1] 1
```

Os cálculos dinâmicos também são permitidos em R. Experimente fazer:

```
> somador( x+4, 3, diminuidor(5,3,1) )
```

3.4 Concatenação e Arrays

Em R a forma que utilizamos para criar arrays (estruturas compostas de dados) é através da concatenação. Nesta linguagem, concatenação é o processo de combinar, ligar ou juntar valores numéricos ou caracteres. Esta é uma das operações mais utilizadas para a realização de análise de dados e isto é alcançado através da função concatenar, que é conhecida pela forma: `c()`.

Para começar, vamos iniciar com um pequeno array numérico criado a partir de concatenação:

```
> postos <- c(1,2,3,4,5,6)
> postos
[1] 1 2 3 4 5 6
```

Assim como fizemos com outros objetos, podemos realizar operações matemáticas com arrays, por exemplo:

```
> postos + 3
[1] 4 5 6 7 8 9
```

Podemos também criar outro objeto com base em operações aritméticas com arrays, por exemplo:

```
> salas <- postos * 2
> salas
[1] 2 4 6 8 10 12
```

Outra coisa possível, é a realização de operações entre arrays, como esta:

```
> salas + postos
[1] 3 6 9 12 15 18
```

Uma observação importante: tenha sempre cuidado ao realizar operações entre arrays, pois resultados inesperados poderão ocorrer caso os mesmos possuam tamanhos diferentes. A

linguagem R consegue realizar a operação sem problemas, mas tenha cautela quando utilizar este recurso. Experimente criar um array numérico com 7 valores e outro com 3 valores e em seguida realize a soma entre eles e veja o resultado.

3.5 Listando e removendo objetos

Lembra quando foi dito que os objetos em R são não-transientes na seção? Pronto, agora vamos ver o que isto quer dizer. Experimente executar a função a seguinte:

```
> ls()
```

Você deverá estar vendo uma lista contendo todos os objetos que você criou na atual seção de R que você está trabalhando. Observe que na lista estão aparecendo também as funções, isto acontece porque elas também são objetos. Mas você falou sobre serem não transientes na seção, o que isto quer dizer? Bem, isto quer dizer que enquanto você estiver utilizando esta seção eles estarão na memória da sua máquina aguardando serem utilizados. Se eles fossem transientes, eles seriam automaticamente removidos após um determinado tempo de expiração. Este é um recurso interessante da linguagem e deve ser gerenciado pelo programador.

Para remover um determinado objeto da memória, devemos utilizar a função `rm()` da seguinte forma:

```
> rm(a..b)
```

Após teclamos ENTER o objeto `a..b` será removido da memória e isto poderá ser verificado executando novamente a função `ls()`.

Caso seja, por algum motivo, necessário remover todos os objetos da memória de trabalho da sua seção, utilize a função `rm()` da seguinte forma:

```
> rm(list=ls())
```

Após realizar esta operação, nenhum objeto poderá ser recuperado da memória. Lembre-se disto.

3.6 Geração de Sequências

Na linguagem R é possível gerar sequências numéricas de várias formas e a mais comum é através do símbolo ":" (dois pontos). Através dele é possível gerar sequencias lineares regulares, como por exemplo:

```
> 1:15
```

```
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

```
> 21:28
```

```
[1] 21 22 23 24 25 26 27 28
```

Podemos gerar também as sequências inversas e também negativas:

```
> 3:-5
```

```
[1] 3 2 1 0 -1 -2 -3 -4 -5
```

Vale lembrar que os dois pontos possuem a maior prioridade numa expressão, então se fizermos `3*2:4` a prioridade será a mesma que `3*(2*4)`

```
> 3*2:4  
[1] 6 9 12  
  
> 3*(2:4)  
[1] 6 9 12
```

Caso haja a necessidade de construir uma sequência mais elaborada, podemos utilizar a função `seq()`. Através dela podemos utilizar uma razão para estabelecer o crescimento da sequência. No exemplo a seguir temos uma sequência numérica de 1 a 10 com a razão 2:

```
> seq(from=1, to=10, by=2)  
[1] 1 3 5 7 9
```

Mantendo sua característica estatística e matemática, R também consegue calcular a razão da sequência entre dois valores com base no seu comprimento. Se pegarmos o exemplo anterior e fizermos o seguinte:

```
> seq(from=1, to=10, length=5)  
[1] 1.00 3.25 5.50 7.75 10.00
```

Comparando os dois resultados, temos 5 passos de comprimento, entretanto a primeira sequência não chega até 10 devido à razão=2, porém no segundo a função calculou a razão=2.25 automaticamente com base na quantidade de passos definidos em `length=5`.

Vamos tentar agora o seguinte exemplo:

```
> seq(from=-10, to=10, length=8)  
[1] -10.000000 -7.142857 -4.285714 -1.428571 1.428571 4.285714  
7.142857  
[8] 10.000000
```

Observe que agora temos uma linha iniciada com [8] isto quer dizer que ao lado dele começa o oitavo elemento da sequência, isto devido à quebra de linha no prompt de comando.

No próximo exemplo, temos a versatilidade da função `seq()`, desta vez sem utilizar o parâmetro “to”.

```
> seq(from=10, length=5, by=1.5)  
[1] 10.0 11.5 13.0 14.5 16.0
```

E agora utilizando o parâmetro “along”:

```
> salas=1:5  
> salas  
[1] 1 2 3 4 5  
  
> seq(from=10, to=12, along=salas)  
[1] 10.0 10.5 11.0 11.5 12.0
```

O parâmetro `along` informa à função `seq()` que é para ela utilizar como referência de comprimento da sequência o tamanho do objeto informado, no caso da sequência “salas” o

tamanho é 5 elementos e como resultado a nova sequência foi gerada da mesma forma, com 5 elementos.

3.7 Indexação de Objetos

Indexação de objetos e subscripting representam o ato de acessar ou extrair valores de um objeto multivalorado. Acessar dados através de indexação é a forma básica de realizar consultas e realizar análises em R.

Para esta atividade utilizamos o colchetes “[]” quando formos acessar um elemento específico de um array. Vamos criar o seguinte array:

```
> litros<-seq(from=2, to=10, length=6)
> litros
[1] 2.0 3.6 5.2 6.8 8.4 10.0
```

Agora vamos consultar o terceiro elemento deste array:

```
> litros[3]
[1] 5.2
```

Experimente realizar o índice 0 e o índice 7 para ver o que acontece.

Caso você queira extrair todos os valores de litros com exceção do segundo valor, utilize:

```
> litros[-2]
```

Caso você queira extrair dois ou mais valores, você pode utilizar a função de concatenação `c()`. Por exemplo, para extrair o segundo e o quinto, basta fazer o seguinte:

```
> litros[c(2,5)]
```

Caso você não queira nem o segundo nem o quinto, use:

```
> litros[c(-2,-5)]
```

Outra forma de consulta é utilizando sequências:

```
> litros[2:4]
[1] 3.6 5.2 6.8
```

Subscripting é uma técnica de indexação baseada em valores lógicos. Em R podemos fazer uma consulta para listar os valores acima de 6 litros da seguinte forma:

```
> litros[litros>6]
[1] 6.8 8.4 10.0
```

3.8 Data Frames

Data frames são as estruturas de dados mais comuns utilizadas na linguagem R. Outras estruturas como vetores, arrays, listas, matrizes e fatores também estão disponíveis, entretanto, devido a característica de ser um objeto composto por múltiplos valores estruturados como uma tabela (com linhas e colunas), os Data Frames circundam o mundo das análises dados.

Data frames podem ser criados a partir de arquivos ou através de outros objetos, vamos ver um exemplo:

```
> df_reservatorio <- data.frame(Tanque=11:20, Litros=seq(from=20, by=1.25,
length=10))
> df_reservatorio
  Tanque Litros
1     11  20.00
2     12  21.25
3     13  22.50
4     14  23.75
5     15  25.00
6     16  26.25
7     17  27.50
8     18  28.75
9     19  30.00
10    20  31.25
```

No exemplo acima, criamos um data frame com duas colunas: Tanque e Litros. Utilizamos objetos do tipo sequência para criar os valores que foram armazenados nele. Observe que o número das linhas são gerados automaticamente pela linguagem. Os data frames são organizados em linhas e colunas e podem ser consultados de forma parecida com os arrays utilizando os colchetes da seguinte forma:

```
> df_reservatorio[3,2]

[1] 22.5
```

Neste caso, obtivemos o valor 22.5 que corresponde ao item da linha 3 e coluna 2 (representando os litros). Se fizermos `df_reservatorio[6,1]` obteremos o valor 16 que corresponde ao número do tanque.

Podemos também visualizar todos os valores de uma coluna como um array. Por exemplo:

```
> df_reservatorio[,2]

[1] 20.00 21.25 22.50 23.75 25.00 26.25 27.50 28.75 30.00 31.25
```

Podemos forçar o array da coluna se comportar como um data frame, usando um cast. Por exemplo:

```
> as.data.frame(df_reservatorio[,2])
  df_reservatorio[, 2]
1                20.00
2                21.25
3                22.50
4                23.75
5                25.00
6                26.25
7                27.50
8                28.75
9                30.00
```

10

31.25

Podemos também mostrar os valores de uma linha específica:

```
> df_reservatorio[2,]  
  Tanque Litros  
2      12  21.25
```

Assim como fizemos com o array, podemos indexar os valores utilizando sequências:

```
> df_reservatorio[2:5,]  
  Tanque Litros  
2      12  21.25  
3      13  22.50  
4      14  23.75  
5      15  25.00
```

Os data frames permitem a extração de valores via subscripting pelo nome da coluna, utilizando o cifrão (\$).

```
> df_reservatorio$Litros  
[1] 20.00 21.25 22.50 23.75 25.00 26.25 27.50 28.75 30.00 31.25
```

É possível também utilizar o subscripting via nome e especificar a posição do elemento:

```
> df_reservatorio$Litros[5]  
[1] 25
```

Assim como podemos utilizar sequência e concatenação:

```
> df_reservatorio$Litros[1:3]  
[1] 20.00 21.25 22.50  
  
> df_reservatorio$Litros[c(2,6)]  
[1] 21.25 26.25
```

Data frames são úteis para manipular Data Sets e por isso muitas vezes é importante conhecermos o número de dimensões que eles possuem. Por este motivo, existem funções específicas para obtermos informações essenciais sobre como estes objetos estão dimensionados.

Se quisermos verificar a quantidade de colunas ou de linhas de um Data Frame, podemos fazer o seguinte:

```
> ncol(df_reservatorio)  
[1] 2  
  
> nrow(df_reservatorio)  
[1] 10
```

Outra forma de fazer isso é através da função `dim()`, que retorna o número de linhas e colunas:

```
> dim(df_reservatorio)
```

```
[1] 10  2
```

No caso de arrays, existe a função `length()` que retorna a quantidade de elementos existentes no mesmo:

```
> length(df_reservatorio[,2])
```

```
[1] 10
```

3.9 Data Querying e estatística base

Quando começamos a trabalhar com um dataset desconhecido devemos analisá-lo de forma a termos ideia de suas dimensões e das características dos dados contidos nele. Por isso, é comum realizarmos questionamentos que nos levarão a obtermos as devidas respostas sobre o perfil do dataset e seus dados.

Para fins de estudo, R disponibiliza por padrão os chamados in-built datasets. Na listagem a seguir temos 104 deles listados via comando `ls("package:datasets")`:

```
> ls("package:datasets")
[1] "ability.cov"      "airquality"      "airPassengers"
[4] "attitude"        "austres"          "attenu"
[7] "beaver2"          "BJsales"          "beaver1"
[10] "BOD"              "cars"             "BJsales.lead"
[13] "chickwts"         "co2"              "ChickWeight"
[16] "crimtab"          "discoveries"      "CO2"
[19] "esoph"            "euro"             "DNase"
[22] "eurodist"         "EuStockMarkets"  "euro.cross"
[25] "fdeaths"          "Formaldehyde"    "faithful"
[28] "freemx"           "freemx.y"         "freeny"
[31] "freemx.x"         "freemx.y"         "HairEyeColor"
[34] "Harman23.cor"     "Harman74.cor"    "Indometh"
[37] "infert"           "InsectSprays"    "iris"
[40] "iris3"            "islands"          "JohnsonJohnson"
[43] "LakeHuron"        "ldeaths"          "lh"
[46] "LifeCycleSavings" "Loblolly"         "longley"
[49] "lynx"             "mdeaths"          "morley"
[52] "mtcars"           "nhtemp"           "Nile"
[55] "nottem"           "npk"              "occupationalStatus"
[58] "Orange"           "OrchardSprays"    "PlantGrowth"
[61] "precip"           "presidents"       "pressure"
[64] "Puromycin"        "quakes"           "randu"
[67] "rivers"           "rock"             "Seatbelts"
[70] "sleep"            "stack.loss"       "stack.x"
[73] "stackloss"        "state.abb"        "state.area"
[76] "state.center"     "state.division"   "state.name"
[79] "state.region"     "state.x77"        "sunspot.month"
[82] "sunspot.year"     "sunspots"         "swiss"
[85] "Theoph"           "Titanic"          "ToothGrowth"
[88] "treering"         "trees"            "UCBAdmissions"
[91] "UKDriverDeaths"   "UKgas"            "USAccDeaths"
[94] "USArrests"        "UScitiesD"        "USJudgeRatings"
[97] "USPersonalExpenditure" "uspop"          "VADeaths"
[100] "volcano"          "warppbreaks"      "women"
[103] "WorldPhones"     "WWWusage"
```

Vamos escolher o dataset `sunspot.year` que contém o número de manchas solares observadas e registradas entre os anos de 1700 e 1988.

```
> sunspot.year
Time Series:
Start = 1700
End = 1988
Frequency = 1
 [1]  5.0  11.0  16.0  23.0  36.0  58.0  29.0  20.0  10.0  8.0  3.0  0.0
[13]  0.0  2.0  11.0  27.0  47.0  63.0  60.0  39.0  28.0  26.0  22.0  11.0
[25] 21.0 40.0 78.0 122.0 103.0 73.0 47.0 35.0 11.0 5.0 16.0 34.0
[37] 70.0 81.0 111.0 101.0 73.0 40.0 20.0 16.0 5.0 11.0 22.0 40.0
[49] 60.0 80.9 83.4 47.7 47.8 30.7 12.2 9.6 10.2 32.4 47.6 54.0
```

Acima temos um parte deste dataset. Como foi dito anteriormente, a melhor forma de trabalharmos com um dataset e colocando ele num data frame. Faremos isto da seguinte forma:

```
> manchas_solares <- data.frame(Ano=1700:1988, NumManchas=sunspot.year)
```

Agora que temos nosso data frame criado, podemos utilizar algumas funções de Data Querying para explorá-lo. Muitas vezes é útil visualizarmos o início de um dataset para sabermos como ele está organizado:

```
> head(manchas_solares)
  Ano NumManchas
1 1700         5
2 1701        11
3 1702        16
4 1703        23
5 1704        36
6 1705        58
```

A função `head()` lista as 6 primeiras linhas do data frame. Outra função que trabalha de maneira semelhante é `tail()`, que mostra as últimas 6 linhas do data frame.

```
> tail(manchas_solares)
  Ano NumManchas
284 1983       66.6
285 1984       45.9
286 1985       17.9
287 1986       13.4
288 1987       29.2
289 1988      100.2
```

Quando estamos estudando o perfil dos dados, muitas vezes precisamos saber se eles se enquadram em algum critério de qualidade. Por exemplo, a existência de valores negativos é indesejável num dataset de manchas solares. Podemos testar isso da seguinte forma:

```
> any(manchas_solares[,2]<0)
[1] FALSE
```

A função `any()` verificou se existia algum valor negativo na coluna `NumManchas` e retornou `FALSE`, indicando que não há. Da mesma forma, poderíamos verificar se existe algum ano que esteja fora do período anunciado dos dados que vai de 1700 a 1988.

```
> any(manchas_solares[,1]<1700 | manchas_solares[,1] > 1988)
[1] FALSE
```

Após verificarmos que o dataset está em condições de uso, podemos iniciar um estudo dos dados com alguma estatística descritiva sobre ele. Por exemplo, podemos verificar a média do manchas solares do período:

```
> mean(manchas_solares[,2])  
[1] 48.61349  
  
> round(mean(manchas_solares[,2]),2)  
[1] 48.61
```

No segundo exemplo acima, utilizamos a função `round()` para que a média fosse mostrada com duas casas decimais.

Podemos também verificar a quantidade máxima e mínima de manchas solares:

```
> max(manchas_solares[,2])  
[1] 190.2  
  
> min(manchas_solares[,2])  
[1] 0
```

Agora quero saber quais os anos onde não foi registrado nenhuma mancha solar.

```
> which(manchas_solares[,2]==0)  
[1] 12 13 111  
  
> manchas_solares[which(manchas_solares[,2]==0),]  
   Ano NumManchas  
12 1711          0  
13 1712          0  
111 1810          0
```

A função `which()` permite realizar Data Querying em datasets grandes com excelente performance. No segundo exemplo anterior, listamos o conteúdo das linhas onde a mancha solar é zero, enquanto no primeiro apenas obtivemos as linhas referentes a zero na contagem de manchas.

Se você quisesse saber a quantidade de anos onde a leitura foi zero, bastaria fazer o seguinte:

```
> length(which(manchas_solares[,2]==0))  
[1] 3
```

A variedade de funções de Data Querying em R é bastante variada. Podemos de uma única vez realizar um sumário estatístico dos dados fazendo:

```
> summary(manchas_solares[,2])  
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.   
   0.00   15.60   39.00   48.61   68.90   190.20
```

Veja que temos o mínimo, o primeiro quartio, a mediana o terceiro quartio e o máximo valores.

Podemos fazer o total de manchas solares entre 1700 e 1988:

```
> colSums(manchas_solares[2])
NumManchas
14049.3
```

3.10 Lidando com dados não disponíveis

Uma coisa muito comum para quem lida com Data Sets é a situação onde em determinadas colunas existem lacunas de dados ou dados ausentes. Isto faz parte da nossa rotina.

Entretanto, não é por este motivo que não poderemos realizar nossas análises, mas temos que saber lidar com esta situação.

Vamos utilizar outro dataset in-built do R, desta vez sacaremos o `airquality` que possui informações sobre a qualidade do ar medida na cidade de Nova York entre os meses de maio e setembro de 1973.

```
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190   7.4   67     5    1
2    36     118   8.0   72     5    2
3    12     149  12.6   74     5    3
4    18     313  11.5   62     5    4
5    NA      NA  14.3   56     5    5
6    28      NA  14.9   66     5    6
```

Ao visualizarmos as primeiras 6 linhas do dataset, podemos perceber a presença de valores “NA” que significam não disponível (Not Available).

A presença de valores NA nos impedem de realizar alguns cálculos como por exemplo:

```
> mean(airquality[,2])
[1] NA
```

Este resultado foi exibido porque a função `mean()` não sabia o que fazer ao encontrar NA. Entretanto, é possível orientar a função a desconsiderar o valor NA através do parâmetro `na.rm` (NA remove):

```
> mean(airquality[,2], na.rm=TRUE)
[1] 185.9315
```

Você pode se antecipar ao utilizar um dataset e verificar se o mesmo contém valores NA através da função `any()` em conjunção com uma função de teste lógico: `is.na()`:

```
> any(is.na(airquality))
[1] TRUE
```

Caso haja uma decisão de projeto onde as linhas que contém NA deverão ser desconsideradas, podemos utilizar a função `na.omit()` para fazer isso gerando um novo dataset.

```
> airquality_ok <- na.omit(airquality)
> head(airquality_ok)
  Ozone Solar.R Wind Temp Month Day
1    41     190   7.4   67     5    1
```

2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
7	23	299	8.6	65	5	7
8	19	99	13.8	59	5	8

3.11 Lendo dados de um dataset em arquivo

Como em qualquer outra linguagem, R também possui recursos para trabalhar com fontes de dados externas. Uma das formas mais comuns é a leitura de datasets contidos em arquivos texto. Para isso a linguagem fornece a função `read.table()` que cria automaticamente um data frame a partir dos dados lidos.

Vamos começar com um dataset in-built que contém informações mensais sobre o quantitativo de passageiros em voos internacionais entre 1949 e 1960, chamado `AirPassengers`. Vamos pegar seu conteúdo, colar num bloco de notas e salvá-lo no Desktop com o nome "Passageiros.txt".

Agora vamos definir o Desktop como nossa pasta de trabalho:

```
> setwd("c:/Documents and Settings/seu_usuario/Desktop/")
> getwd() # use para verificar se funcionou
```

Agora vamos criar nosso data frame lendo o arquivo que acabamos de criar:

```
> passageiros <- read.table("Passageiros.txt", header=TRUE, sep=" ")
> head(passageiros)
      Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1949 112 118 132 129 121 135 148 148 136 119 104 118
1950 115 126 141 135 125 149 170 170 158 133 114 140
1951 145 150 178 163 172 178 199 199 184 162 146 166
1952 171 180 193 181 183 218 230 242 209 191 172 194
1953 196 196 236 235 229 243 264 272 237 211 180 201
1954 204 188 235 227 234 264 302 293 259 229 203 229
```

Na função `read.table()` informamos o nome do arquivo, sinalizamos que o mesmo possui um cabeçalho e que o separador de valores corresponde a um espaço.

Esperimente executar a função abaixo e depois avalie o que aconteceu:

```
> str(passageiros)
```

3.12 Salvando um dataset em arquivo

Seguindo uma forma similar à leitura, temos a operação de salvamento de um dataset em arquivo e para isto, temos a função `write.table()`. Antes de utilizá-la, vamos incrementar um pouco o dataset do tópico anterior.

Vamos adicionar a ele uma coluna com o total anual de passageiros.

```
> passageiros$TotAnual <- cbind(rowSums(passageiros))
> head(passageiros)
      Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec TotAnual
1949 112 118 132 129 121 135 148 148 136 119 104 118      1520
```

```
1950 115 126 141 135 125 149 170 170 158 133 114 140 1676
1951 145 150 178 163 172 178 199 199 184 162 146 166 2042
1952 171 180 193 181 183 218 230 242 209 191 172 194 2364
1953 196 196 236 235 229 243 264 272 237 211 180 201 2700
1954 204 188 235 227 234 264 302 293 259 229 203 229 2867
```

Utilizamos a função `cbind()` para gerar uma coluna de dados anônima e em seguida criamos uma nova coluna (`TotAnual`) em nosso data frame com a utilização do cifrão (\$).

Feito isso, podemos gravar nosso novo dataset:

```
> write.table(passageiros, "tot_passageiros.txt", col.names=NA, row.names=TRUE,
quote=FALSE, sep="\t")
```

3.13 Funções estatísticas básicas

A seguir temos um pequeno resumo de algumas funções estatísticas básicas disponíveis em R.

Função em R	Valor retornado
<code>mean(x)</code>	Média aritmética dos valores do objeto x
<code>median(x)</code>	Mediana dos valores do objeto x
<code>max(x)</code>	Máximo valor no objeto x
<code>min(x)</code>	Mínimo valor no objeto x
<code>range(x)</code>	Vetor do <code>min(x)</code> e <code>max(x)</code>
<code>sum(x)</code>	Total de todos os valores em x
<code>var(x)</code>	Variância de x
<code>quantile(x)</code>	Vetor contendo o mínimo, o menor quantil, mediana, quantil superior e o máximo de x
<code>cumsum(x)</code>	Soma acumulativa de x
<code>cumprod(x)</code>	Produto acumulativo de x
<code>colMeans(x)</code>	Média das colunas do data frame x
<code>rowMeans(x)</code>	Média das linhas do data frame x
<code>colSums(x)</code>	Soma das colunas do data frame x
<code>rowSums(x)</code>	Soma das linhas do data frame x

3.14 Estruturas de programação

Escrever scripts ou programas em R é similar a outras linguagens, por exemplo:

```
# Exemplo de script em R
fatura1 <- 110
fatura2 <- 350

fat_medio <- (fatura1+fatura2)/2

if (fat_medio < 50) {
  print("Faturamento abaixo da meta")
} else {
  print("Meta cumprida!")
}
fat_medio
```


O script acima é um exemplo de programa simples em R onde foi utilizada uma estrutura condicional.

Podemos também utilizar um conceito de iteração restrita para a criação de laços:

```
# Exemplo de script em R

for( i in 1:5 ) {
  if ( i < 3 ) {
    cat("Setor Sul",i,"\n")
  } else {
    cat("Setor Norte",i,"\n")
  }
}
```

Outro conceito importante é o de iteração irrestrita para a criação de loops:

```
# Exemplo de script em R

i <- 1
while( i <= 5 ) {
  if ( i < 3 ) {
    cat("Setor Sul",i,"\n")
  } else {
    cat("Setor Norte",i,"\n")
  }
  i <- i + 1
}
```

Iterações irrestrita é algo que deve ser evita ao se trabalhar com analytics devido à possibilidade de estabelecer um loop infinito. Quando pensamos em grande volume de dados, uma farm de servidores pode ser sacrificada por um deslize de um programador ao utilizar este recurso, portanto tenha cautela ao utilizá-las.

4. Avaliando uma análise de dados em R com um modelo de árvore de decisão

Vamos tomar por base o seguinte script/programa em R:

```
####
# Exemplo de Arvore de Decisao em R
# com base em regressao linear
# Utilizando uma base de teste de transito em dias de chuva
# Arquivo utilizado: chuva_transito.csv
#

# importa as bibliotecas
library(readr) # lib especializada em data frames
library(rpart) # lib especializada em arvores de regressao e particionamento
de ramos
library(rpart.plot) # lib especializada em plotar arvores de regressao e
particionamento de ramos

# ler o arquivo com as informacoes sobre chuva e o transito
chuvaTransito = read.csv("chuva_transito.csv",header=TRUE, sep=";")

# calcula a regressao linear para a construcao da arvore
arv_transito = rpart(velocidade ~ mm_chuva + bairro, data=chuvaTransito,
method="poisson")

# realiza a plotagem da arvore de decisao
```

```
prp(arv_transito)
```

Este script constrói uma análise de dados a partir de um dataset contendo informações sobre a velocidade do trânsito em dois bairros de Salvador com base no volume de chuva apurada nos horários de pico.

Vamos analisar o código acima:

Utilizamos as bibliotecas `readr` e `rpart` juntamente com o aporte gráfico `rpart.lib`.

A `readr` amplia a capacidade de R para operar com data frames de tamanhos maiores com um subset maior de funções de Data Querying.

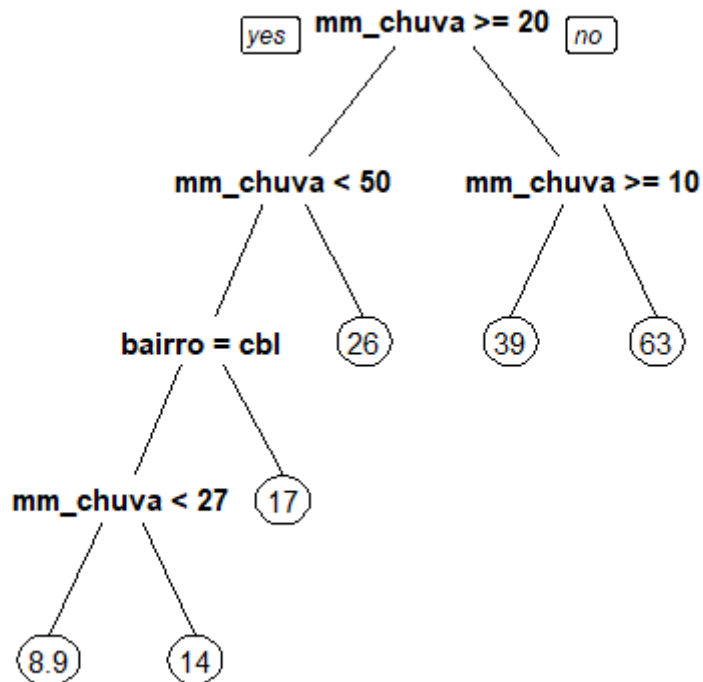
A `rpart` é uma biblioteca especializada na construção de árvores de decisão simples possibilitando a geração de gráficos a partir de um subset de pacotes específicos para isso.

Criamos um dataframe `chuvaTransito` para armazenar os dados do arquivo CSV. Aqui utilizamos uma variação da função `read.table()` que também pode ser escrita como `read.csv()` (isso é outra coisa comum em R).

```
arv_transito = rpart(velocidade ~ mm_chuva + bairro, data=chuvaTransito,
method="poisson")
```

Na linha acima, estabelecemos a fórmula do modelo analítico que queremos explorar. A função `rpart` funciona com base em uma fórmula matemática para estabelecer uma relação causal to tipo `y` em função de `x` (ou `y ~ x`), sendo que `x` é uma operação direta entre valores. Desta forma, podemos traduzir a fórmula da seguinte maneira: `velocidade ~ mm_chuva + bairro` significa que queremos a correlação linear da velocidade do trânsito em função da chuva levando em consideração o bairro. Para isso, informamos à função que ela deverá utilizar como fonte de dados o data frame `chuvaTransito` e que a árvore deve utilizar o método de Poisson. Os métodos aceitos pela `rpart` são: `class`, `Poisson`, `anova` e `exp` (somente para objetos do tipo `survive`, que são orientados a eventos ao longo do tempo).

Em seguida plotamos a árvore com a função `prp()`. Esta função abre uma janela gráfica com a representação da análise.



Com a árvore disponível busque fazer a interpretação dos valores da mesma, comparando-os com os dados contidos no arquivo csv.

Agora realize algumas modificações na fórmula e refaça o gráfico. Busque também realizar a construção da árvore com outros datasets.