# 

# Capítulo 0

# Ferramentas

## *Programação com estatística básica*

***Master Foo and the Shell Tools[[1]](#footnote-0)***

*Um aprendiz do caminho Unix veio ao Mestre Foo e disse: “Estou confuso. Não é o caminho Unix que cada programa deve se concentrar em uma coisa e fazê-la bem?*

*Mestre Foo assentiu.*

*O aprendiz continuou: “Também não é do caminho Unix que a roda não deve ser reinventada?*

*Mestre Foo assentiu novamente.*

*“Então, por que existem diversas ferramentas com capacidades similares em processamento de texto: sed, awk e Perl? Com qual delas posso praticar melhor o caminho Unix?”*

*Mestre Foo perguntou ao aprendiz: “Se você tem um arquivo de texto, qual ferramenta usaria para produzir uma cópia com algumas palavras trocadas por uma string de sua escolha?”*

*O aprendiz torceu o nariz e disse: “As expressões regulares de Perl seriam um excesso para tarefa tão simples. Eu não conheço awk, e venho escrevendo scripts sed nas últimas semanas. Como tenho experiência com sed, eu preferiria ele no momento. Mas se o trabalho precisa ser feito apenas uma vez, um editor de textos funcionaria.”*

*Mestre Foo assentiu e respondeu: “Quando você estiver com fome, coma; quando estiver com sede, beba; quando estiver cansado, durma.”*

*E, ao ouvir isso, o aprendiz foi iluminado.*

# Computadores

Ao longo do texto, usaremos exemplos com software. Computadores são úteis para acelerar os cálculos necessárias para nossos objetivos.

Há milênios, o homem usa instrumentos, como ábacos e tabelas, para fazer operações extensas e precisas envolvendo grandes números. Dado um problema ou dado a ser computado, esses instrumentos mecanismos automatizam partes do processo devido à maneira como foram construídos. A principal diferença destas ferramentas para os computadores de hoje é que nossas máquinas podem ser programadas para fazer computações arbitrárias.

Ada Lovelace (10 December 1815 – 27 November 1852) foi a primeira a descobrir essa possibilidade. Estudando a Máquina Analítica de Charles Babbage, Ada concebeu uma maneira de realizar computações para as quais a máquina não havia sido desenhada originalmente. O programa concebido calculava os Números de Bernoulli.

Máquinas desse tempo pesavam toneladas e eram muito mais lentas. O avançar dos anos tornou a tecnologia mais acessível, ao ponto de possibilitar computadores pessoais de alta potência e baixo-custo.

Os programas aqui apresentados são escritos em *R, Stan* e Python. As três têm código aberto, podendo ser obtidas, instaladas e usadas sem custos. Sendo um texto didático, o estilo empregado nos exemplos busca balancear legibilidade, performance e robustez. Os três frameworks usam libs em C/C++/Fortran para otimizar computações e interface com GPU (graphic processing unit)

R será mais usada. É uma linguagem interpretada voltada à computação estatística, possuindo ferramentas úteis em sua biblioteca de base. Entre estas, funções para gerar e manipular distribuições probabilísticas. Sendo uma linguagem de ‘alto nível’, não temos overhead cognitivo com manejo de memória e hardware no código, o que é feito automaticamente pelo interpretador. O ecossistema para visualização de dados possui poder e flexibilidade. A comunidade R cresce rápido e fluência nessa linguagem dá acesso a ferramentas muito diversas com bases grandes de suporte. Há suporte para estilo funcional e orientado a objetos.

*Stan* é uma linguagem/plataforma de domínio específico bastante popular entre estatísticos bayesianos. Possui ferramentas poderosas (e.g: *Variational inference, MCMC* com NUTS e HMC) para lidar com distribuições probabilísticas e inferência nesse contexto.

Python é uma linguagem de propósito geral. Bastante popular e dotada de uma base de usuários imensa. Linguagem de primeira escolha como interface de alto nível (*wrapper*) para a maioria das tecnologias de aplicação industrial (*e.g: PyTorch, Pyro, Tensorflow*). Com dois “*dialetos*” incompatíveis (2.*X* e 3.*X*) diferindo em mínimos detalhes, possui também uma variedade de opções que pode confundir iniciantes (*e.g:* pip vs. conda). Vamos precisar de Python (PyMC/Pyro) para combinar inferência Bayesiana e redes neurais.

## R

## *Curso rápido*

Programas de computador são importantes ao longo dos próximos capítulos para realizar cálculos, gerar dados e visualizações.

Felizmente, os programas que escreveremos são simples, de forma que não precisamos conhecer todos os recursos e características da linguagem R. Neste capítulo, entenderemos os instrumentos básicos para caminharmos.

Veremos diversas maneiras de escrever um programa para calcular a variância σ2 de um conjunto de medidas.

## Instalação

### R

Instruções para download e instalação podem ser encontradas em:  
<https://cloud.r-project.org/>

Em Windows, o processo costuma consistir em clicar no executável de instalação e concordar com os prompts.

Para Linux, envolve adiciona o CRAN à lista de repositórios e baixar o *r-base* ou baixar o código/tarball diretamente do website. Há inúmeros tutoriais explicando a instalação.

### Rstudio

Com o R instalado, recomendo o uso do ambiente de desenvolvimento RStudio ( <https://www.rstudio.com/> ) para obter algumas facilidades. Entre elas: atalhos vim, editor com highlight de sintaxe, autocompletar, *go to function definition*, renderização em tempo real de animações e plots, visualização de datasets, ambiente de desenvolvimento, logs, suporte a markup languages, como Markdown, RMarkdown e Latex.

## Tipos

Entre os tipos básicos do R, lidamos rotineiramente com vetores. Estes são células contíguas contendo dados. Os dados podem ser de muitos tipos:

"logical": a vector containing logical values (TRUE/FALSE)  
"integer": a vector containing integer values (1,2,3,4…,23,26...)  
"double": a vector containing real values (3.14…)  
"complex"a vector containing complex values (2 +2i)  
"character": a vector containing character values (“string”)

Para saber o tipo de um objeto em R, use *typeof(objeto)*. Podemos acessar elementos de um vetor pelo seu índice, independente do tipo. Declaramos dois vetores, *character* e *double*

>a <- c(“banana”, “terracota”, “pie”)  
>b <- c(2.2, 4.4, 5.5)  
> typeof(a)  
[1] "character"  
> typeof(b)  
[1] "double"

Para nossas aplicações, vamos usar números reais (*double*) na maioria dos casos.

## Operadores

Além dos operadores clássicos (+,-,/,-, ...), usamos constantemente dois operadores pouco comuns: O operador “<-” atribui o valor da expressão a sua direita ao objeto à sua esquerda. É preferível ao operador “=” para evitar confusão ao passar argumentos de funções e fazer comparações lógicas.

> a <- 3  
> a  
[1] 3

O operador “%>%” da biblioteca magrittr fornece o resultado da expressão à sua esquerda como argumento para a expressão à sua direita. Evita aninhamento de expressões, tornando fluxos de computações mais legíveis.

As expressões a seguir são equivalentes.

>library(magrittr)  
>result <- 3 %>% exp %>% exp  
>result  
[1] 528491311  
>result == exp(exp(3))  
[1] TRUE

Onde “3 %>% exp %>% exp” equivale a “exp(exp(3))”. exp(a) = *e*a, e ~ 2.72… Usando parênteses, partimos da última computação. Usando o pipe (%>%), começamos com a primeira operação.

## Funções

Uma das formas de escrever programas é através de funções.

Podemos declarar funções que (1) aceitam argumentos de entrada, (2) executam um algoritmo e (3) retornam outputs na saída.

Assim, podemos criar a função *soma2*, que recebe dois argumentos numéricos e retorna a soma de ambos.

>soma2 <- function(argumento1,argumento2){  
 return(argumento1+argumento2)  
}  
Ao invocarmos *soma2* com os argumentos 2 e 5, recebemos *soma2(argumento1=2, argumento2=5)* = 2+5 = 7.

>soma2(2,5)  
[1] 7

Podemos omitir o nome dos argumentos. Assim os objetos são passados na ordem de entrada.  
>soma2(2,3)  
[1] 5

Por padrão, o valor retornado é mostrado no console.

R aceita em sua sintaxe que uma função seja argumento de outra numa mesma instrução:  
> soma2( 2, soma2(3,2) )  
[1] 7

A expressão acima é equivalente a (2 + (3+2)) = 7.

Podemos definir a função de média para um vetor de números, dado pela soma dividida pelo tamanho do vetor:

>mean\_vec <- function(x){  
 sum(x)/length(x)  
}  
>mean\_vec(b) # Definido por b <- c(2.2, 4.4, 5.5)   
[1] 4.033333  
sum(x) retorna a soma de todos os elementos do vetor x. *length(x*) retorna o tamanho do vetor *x.*

A média é uma medida de tendência central para um conjunto de observações. É o ponto mais perto de todos os outros.

#### Muitas formas de calcular a variância

Também podemos calcular uma medida relacionada ao quanto nossos valores se afastam do centro.

Primeiro, calculamos uma distância entre cada elemento **x** e a média das observações **μ**. A distância deve ser um valor positivo. Supondo que **x** e **μ** são medidas num espaço ordenado, podemos usa o módulo da diferença entre os valores: |**x** - **μ**|. Ainda, podemos usar o quadrado da diferença: ***d****i* = (**xi** - **μ**)².

A variância **σ²** das observações é uma medida da dispersão de toda a amostra.

Para calcular **σ²,** somamos todas as distâncias **d*i*** e dividimos o resultado por *n-1[[2]](#footnote-1).*

>var\_2 <- function(x) sum((x - mean(x))^2) / (length(x) - 1)  
>var\_2 (b)  
[1] 2.823333

A variância **σ²** tende a ser maior quando os valores são muito distintos entre si:

>c <- c(100,200,1,45,-24)  
>var\_2(c)  
[1] 7966.3

Outra medida de dispersão, dada nas unidades originais da medida observada, é o desvio-padrão **σ**, dado pela raiz da variância **σ²**.

> var\_2(b) %>% sqrt  
[1] 1.680278

O R possui funções embutidas para muitas aplicações estatísticas: sd (desvio-padrão), var (variância), mean (média)... Em especial, temos funções prontas para trabalhar com diversas distribuições probabilísticas de variáveis aleatórias. Para sortear 10 números de uma distribuição normal:

> rnorm(n=10, mean=0, sd=1)  
 [1] 0.2874490 0.2931469 3.1897423 1.7445002 3.3998010 -0.1482911  
 [7] 2.0257046 -0.6002109 -0.2840376 -0.7715565

Distribuição gamma.   
> rgamma(n=10, shape=1)  
 [1] 1.1183441 1.2770135 1.0972053 1.4820536 2.3542620 0.8231831 0.5535210  
 [8] 5.0481559 0.2853060 0.1623315

Exponencial:

> rexp(n=10, rate = 1)  
 [1] 0.31657586 0.26676766 0.02288276 0.92801416 0.44006133 0.05238540  
 [7] 1.10213153 0.91931786 2.58807134 0.41825081

## Vetores, loops e recursões

Anteriormente, definimos a função para calcular variância como:

>var\_2 <- function(x) sum((x - mean(x))^2) / (length(x) - 1)

Isso só é possível porque o R aplicar funções a vetores de maneira automática. Assim, a expressão (*x - mean(x))^2* subtrai a média de cada elemento do vetor x.

Normalmente, é necessário usar estruturas recursivas para isso. O laço for (*for loop*) define uma sequência de tamanho *n* definido e repete um bloco de comandos n vezes. Se queremos imprimir números entre 1 e 10:

> for (i in 1:10) print(i)

A instrução avalia print(i) para valores i=1,2,3..,10 de forma repetida.

Vamos reescrever nossa função para calcular variância **σ²**. Podemos definir um loop com o tamanho do vetor x e calcular o quadrado da diferença em cada elemento.

Assim,:

var\_3 <- function(x){  
 accumulator <- numeric() #armazena distâncias  
 for (i in 1:length(x))  
 accumulator[i] <- (x[i] - mean(x))^2 # calcula e armazena distâncias.  
 return (sum (accumulator) / (length(x) - 1) ) #calcula media  
}

Ambas definições apresentam o mesmo resultado que a implementação nativa do R:

> var(b)  
[1] 2.823333  
> var\_2(b)  
[1] 2.823333  
> var\_3(b)  
[1] 2.823333

Ainda, uma maneira de manipular muitos elementos é através de funções de alta ordem. Estas funções recebem outras funções como argumentos. Um exemplo é a função **map** da lib **purrr**. Definimos uma função para a distância, f(y) = (y - μ)², e aplicamos ela a todos os elementos. Só então, somamos os resultados e dividimos por *n-1*.

Tudo pode ser feito em apenas um pipe:

>map(.f = function(y) (y - mean(arg))^2, .x = arg) %>%   
 unlist %>% sum(.)/(length(arg) - 1)

Nossa função pode ser escrita:

var\_4 <- function(arg){  
 purrr::map(.f = function(y) (y - mean(arg))^2, .x = arg) %>% # Define e aplica função  
 unlist %>% sum(.)/(length(arg) - 1) # Soma as distancias e divide por n-1  
}  
> var\_4(b)  
[1] 2.823333

## Matrizes e data frames

O R possui estruturas que ajudam a manipulação de dados estruturados como os que vemos comumente em ciências.

A mais simples é a lista. Uma lista é um conjunto de objeto de quaisquer tipos. Assim, podemos ter uma lista contendo vetores, *doubles*, matrizes e gráficos! Tudo em uma estrutura.

> mlist <- list(a = c(1,5,6,7), b = c("a","b","c",”d”))

> mlist  
$a  
[1] 1 5 6 7  
$b  
[1] "a" "b" "c" “d”

> class(mlist)  
[1] "list"

Podemos acessar estruturas internas pelo nome usando o operador *$:*  
> typeof(mlist$a)  
[1] "double"  
> typeof(mlist$b)  
[1] "character"

Outro tipo útil é composto pelas matrizes, que correspondem às matrizes da matemática, podendo também conter caracteres em suas células.

>matrix(data=c(mlist$a,mlist$b),ncol=2)  
 [,1] [,2]  
[1,] "1" "a"  
[2,] "5" "b"  
[3,] "6" "c"  
[4,] "7" "d"

Podemos conduzir multiplicação de matrizes facilmente.

>mat\_example <- matrix(c(.5,.25,.25,.5,0,.5,.25,.25,.5), nrow=3, byrow=TRUE)  
> mat\_example  
 [,1] [,2] [,3]  
[1,] 0.50 0.25 0.25  
[2,] 0.50 0.00 0.50  
[3,] 0.25 0.25 0.50  
> mat\_example %\*% c(1,0,1)

Por fim, *data.frames* são extensões das matrizes:

> mat\_example %>% data.frame  
 X1 X2 X3  
1 0.50 0.25 0.25  
2 0.50 0.00 0.50  
3 0.25 0.25 0.50

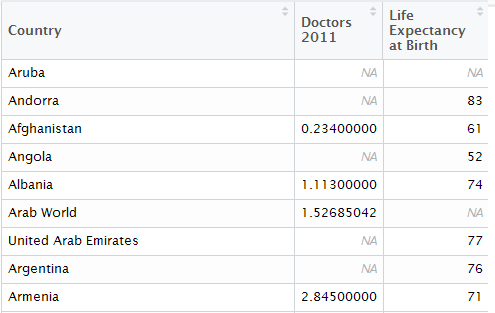
Data frames são os objetos mais comumente tratados em R e seguem o formato tidy.

1 — Cada variável corresponde a uma coluna.

2 — Cada observação corresponde a uma linha.

3 — Cada tipo de unidade observacional forma uma tabela.

Um exemplo visual torna as coisas mais fáceis. A seguir, um recorte do banco de dados usado no post sobre a quantidade de médicos no Brasil. Temos uma variável categórica (País) e duas numéricas (Número de médicos por 1.000 habitantes em 2011 e Expectativa de vida ao nascer):



País, Número de médicos a cada 1000 habitantes em 2011 e Expectativa de vida ao nascer. “NA” corresponde a dados faltantes no R. Layout do RStudio Fonte: WHO

Note que cada linha corresponde a apenas um país (observação) e cada coluna representa uma variável. Se queremos ver a observação 9, vamos à linha correspondente e podemos encontrar os valores: “Armenia” (País), “2,845” (Médicos/1.000 hab. em 2011) e “71” (Expectativa de vida ao nascer).

Para acessar o valor correspondente,usamos índices separados por vírgula. O primeiro espaço é reservado às linhas selecionadas e deve ser um vetor de números (linhas selecionadas) ou vetor com valores lógico do tamanho do dataset (valores com índices TRUE serão incluídos). O segundo espaço corresponde às colunas e deve conter índices numéricos ou nomes das variáveis.

>iris[1:5,c("Species",'Sepal.Length')] *# primeiras 5 linhas com variaveis especies e sepal.length*’  
 Species Sepal.Length  
1 setosa 5.1  
2 setosa 4.9  
3 setosa 4.7  
4 setosa 4.6  
5 setosa 5.0

## Gramática dos gráficos e ggplot

Uma das ferramentas de destaque no ecossistema R é a *ggplot*. Ela provê uma sintaxe bastante poderosa e flexível para plotar visualizações. O segredo está em seu design, que utiliza gramática de gráficos (**G**rammarOf**G**raphics**Plot).** Bertin[[3]](#footnote-2) delineou essa abordagem, que consiste em mapear características dos dados a elementos visuais seguindo uma sintaxe consistente. A lib *ggplot* implementa uma gramática em camadas[[4]](#footnote-3), possibilitando superposições para gráficos complexos.

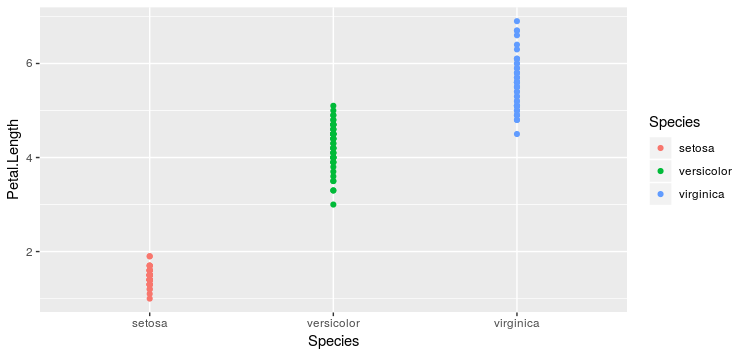
>head(sleep)  
 extra group ID  
1 0.7 1 1  
2 -1.6 1 2  
3 -0.2 1 3

Para usarmos o *ggplot*, podemos declarar (1) o dataframe usado, (2) a relação entre medidas e parâmetros estéticos e (3) objetos geométricos. Parâmetros opcionais podem ser usados, aumentando o número de camadas ou criando transformações.

Assim, podemos plotar um histograma das medidas dos dois grupos com (1) dataset *iris;* (2) dimensão y: tamanho da pétala, cores:espécie, dimensão x: espécie; e (3) objeto geométrico: ponto.

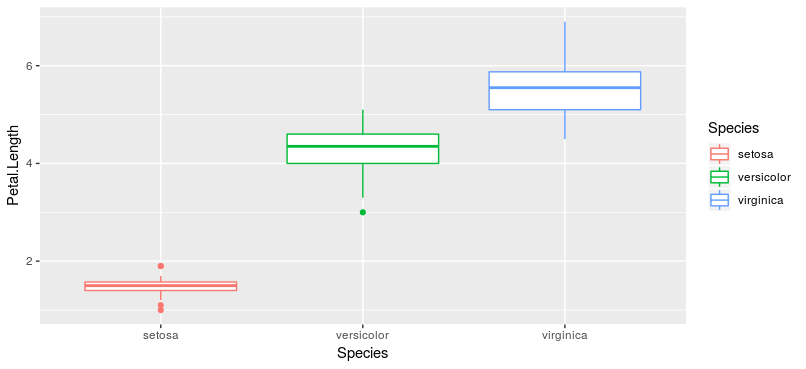
Assim, teremos pontos com a altura (dimensão y) correspondente à medida da pétala e separados ao longo do eixo x por espécies. O ggplot automaticamente discretiza o eixo x.

>library(ggplot2)  
>ggplot(data=iris,aes(y=Petal.Length,x=Species,color=Species))+  
 geom\_point()



Para ilustrar a flexibilidade da biblioteca, note que mudando apenas o objeto geométrico (geom), obtemos um gráfico diferente, mantendo dados e relações (mappings) iguais :

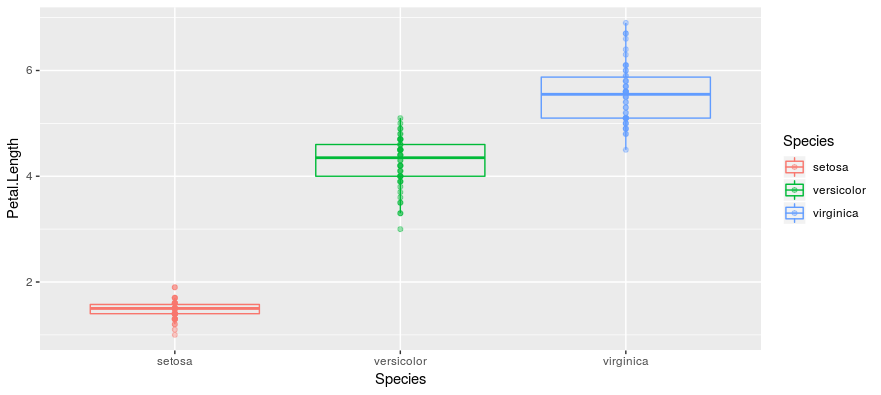
>ggplot(data=iris,aes(y=Petal.Length,x=Species,color=Species))+ **geom\_boxplot()**



As figuras acimas são conhecidas como *boxplots*. O centro correspondente à mediana (percentil 50), as bordas correspondem aos percentis 25 (inferior) e 75 (superior). Os fios, conhecidos como “bigodes”, estendem-se até 1,5\* IQR (onde IQR = Percentil 75 - Percentil 25).

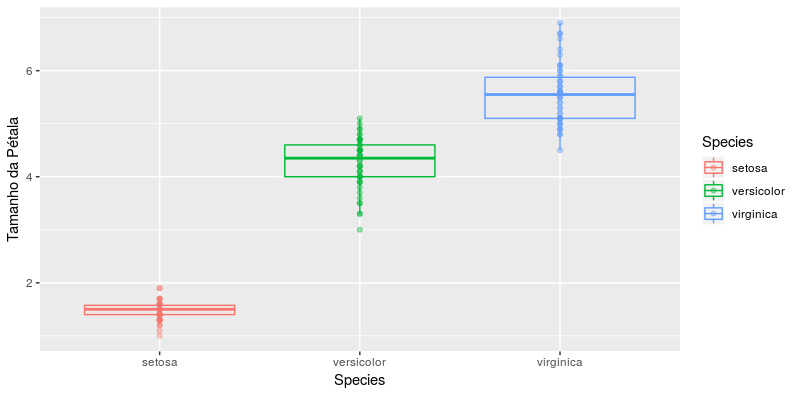
É possível adicionar camadas e estas podem sobrescrever informação de camadas anteriores. Isso torna a sintaxe do ggplot altamente modular. A seguir, superpomos pontos e boxplot:

>ggplot(data=iris,aes(y=Petal.Length,x=Species,color=Species))+  
 geom\_point(alpha=0.4)+ **# camada 1**  
 geom\_boxplot(alpha=0) **# camada 2**



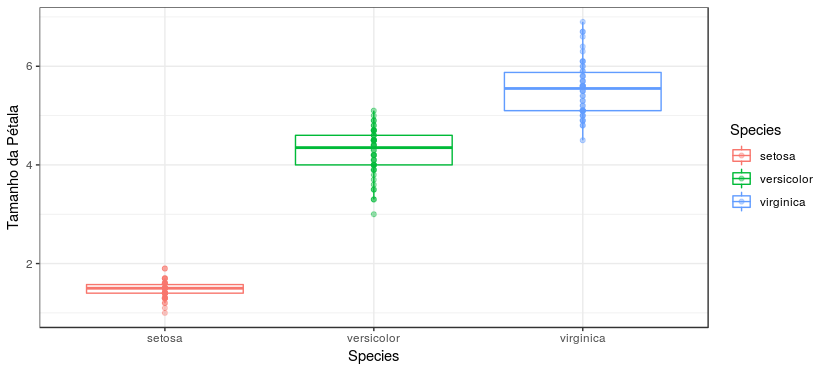
O parâmetro *alpha* regula a transparência dos objetos. Colocamos os boxplot com transparência total (alpha=0), dando visibilidade aos pontos (alpha=0.4). Adicionamos algum grau de transparência para que pontos superpostos sejam mais escuros que pontos individuais. Adicionaremos uma terceira camada, que substitui o rótulo do eixo y para uma legenda em português:

>ggplot(data=iris,aes(y=Petal.Length,x=Species,color=Species))+  
 geom\_point(alpha=0.4)+ **# camada 1**  
 geom\_boxplot(alpha=0)+ **# camada 2**  
 ylab("Tamanho da Pétala") **# camada 3**

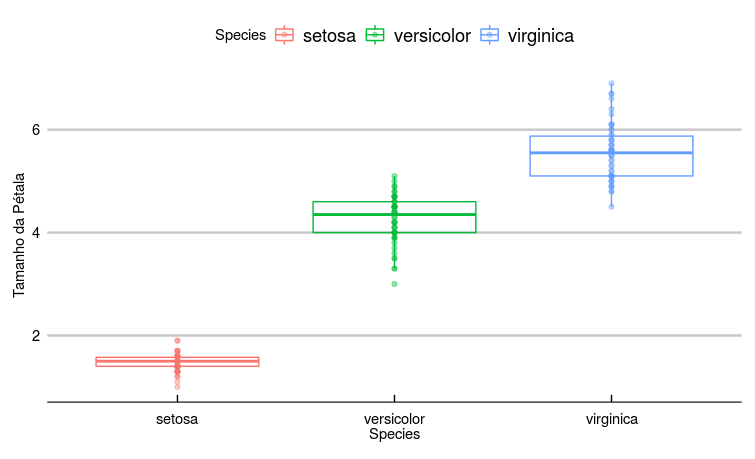


Ainda, existem temas prontos para mudar o estilo geral da imagem:

>ggplot(data=iris,aes(y=Petal.Length,x=Species,color=Species))+  
 geom\_point(alpha=0.4)+ **# camada 1**  
 geom\_boxplot(alpha=0)+ **# camada 2**  
 ylab("Tamanho da Pétala") **# camada 3**   
 theme\_bw() **# camada 4: tema**



ggplot(data=iris,aes(y=Petal.Length,x=Species,color=Species))+  
 geom\_point(alpha=0.4)+ **# camada 1**  
 geom\_boxplot(alpha=0)+ **# camada 2**  
 ylab("Tamanho da Pétala") **# camada 3**   
 theme\_economist\_white(gray\_bg = F) **# camada 4: tema**

****

### Exercícios

1. Qual a diferença entre linguagens compiladas e interpretadas?
2. *Um programa escrito em R pode ser escrito em qualquer outra linguagem.* Está afirmação é verdadeira? Por quê?
3. Cite 3 recursos que uma IDE fornece ao programador.
4. Modifique o tema de fundo do RStudio para um de cor escura (*menos luz para os olhos :)* ).
5. Usando o operador *<- ,* produza:
   1. Um vetor com componentes do tipo logical
   2. Dois vetores de 5 elementos do tipo double
   3. A soma dos elementos nos vetores do item b.
   4. A divisão entre elementos dos vetores do item b.
6. Aplique as funções sd, mean e bar em amostras normais aleatórias de n = 10, 30, 100 e 300. A função *rnorm (n,mean,sd)* pode ajudar. Compare os valores da distribuição de origem com os obtidos.
7. *UnLISP it!* Transforme as seguintes expressões, substituindo parênteses aninhados pelo operador pipe (%>%) quando julgar conveniente:
   1. round ( mean ( c(10 , 2, 3 ) )
   2. round (mean ( rnorm (n = ceiling (runif (1,0,10)))))
   3. paste("a",seq(1:rnorm(n=mean(c(3,2,1,16)))))
   4. round(nrow(iris) + exp(1), digits = ceiling(runif(1,0,10)))
8. Usando o código das funções var\_2 (vetorizado), var\_3 (*for loop*) e var\_4 (função de alta ordem *map*)
   1. Escreva as funções correspondentes para desvio-padrão e compare com a função padrão do R (*sd*). Basta aplicar raiz quadrada ao valor final!
   2. Reescreva as funções usando %>%
9. Usando o dataset *iris*
   1. Selecione apenas os exemplos com tamanho de pétala maior que 4.
   2. Selecione os 10 maiores exemplares. O tamanho aqui é dado pela média das 4 medidas fornecidas.
   3. Calcule a média e o desvio-padrão para duas medidas em cada espécie.
   4. Faça um scatterplot entre duas medidas
      1. Adicione cores de acordo com a espécie
      2. Adicione o rótulo de texto a um dos pontos
      3. Mude títulos (principal, eixos x e y, legenda)
      4. Mude o tema de fundo. Dica: experimente os temas da lib *ggthemes*

1. http://catb.org/esr/writings/unix-koans/shell-tools.html [↑](#footnote-ref-0)
2. Usamos (n-1) no denomidador no lugar de n, como esperado. Essa é a chamada Correção de Bessel, servindo para corrigir a estimativa da variância, que é mais baixa na amostra.. [↑](#footnote-ref-1)
3. Bertin, J. (1983),Semiology of Graphics [↑](#footnote-ref-2)
4. Hadley Wickham.A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, vol. 19, no. 1, pp. 3–28, 2010. [↑](#footnote-ref-3)