

# **Exemplos Integradores**

Prof. Wagner Hugo Bonat

# Regressão logística

## Classificador binário

- ▶ Ferramenta popular em modelagem estatística e aprendizado de máquina.
- ▶ Objetivo: classificar um indivíduo ou observação em uma entre duas categorias.
- ▶ Exemplos:
  - ▶ Classificar um paciente como sadio ou doente.
  - ▶ Classificar um cliente como bom ou mal pagador.
  - ▶ Classificar uma operação como legítima ou fraudulenta.
- ▶ Diversos algoritmos estão disponíveis na literatura:
  - ▶ Árvores de classificação.
  - ▶ Máquina de vetores de suporte.
  - ▶ Redes neurais.
  - ▶ *Gradient boost* ....
- ▶ Regressão logística é muito popular.

## Descrição matemática

- ▶ Suponha que temos um conjunto de observações  $y_i$  para  $i = 1, \dots, n$ .
- ▶ Cada  $y_i \in [0,1]$ .
- ▶ Potenciais objetivos:
  - ▶ Descrever o relacionamento de  $y_i$  com um conjunto de variáveis explanatórias  $x_{ij}$  com  $j = 1, \dots, p$ .
  - ▶ Classificar uma nova observação como 0 ou 1.

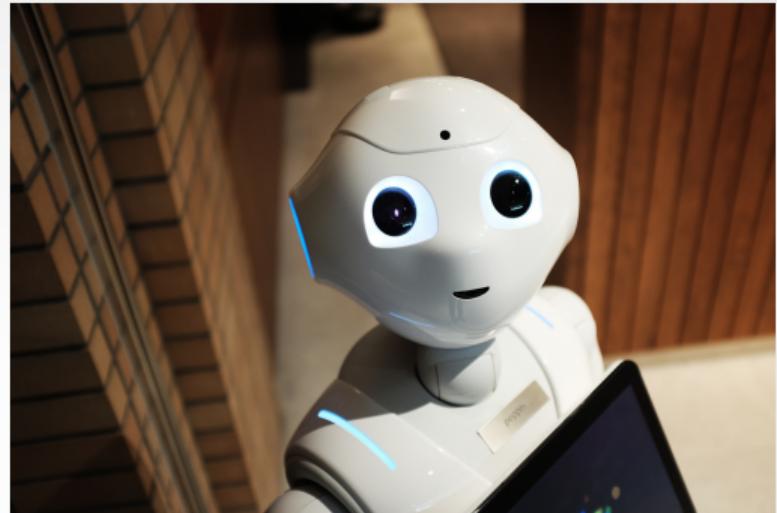


Figura 1. Foto de Alex Knight no Pexels.

## Exemplo de aplicação

- ▶ Suponha um conjunto de dados com três colunas:
  - ▶ Renda anual do usuário (por mil).
  - ▶ Anos de experiência do usuário.
  - ▶ Se o usuário é *premium* ou não.
- ▶ Objetivos:
  - ▶ Identificar como as covariáveis **renda** e **anos** influenciam a compra *premium*.
  - ▶ Predizer se um novo usuário será ou não assinante *premium*.
  - ▶ Identificar na base potenciais usuários para o serviço *premium*.
  - ▶ Orientar campanhas de *marketing*.

## Exemplo de aplicação

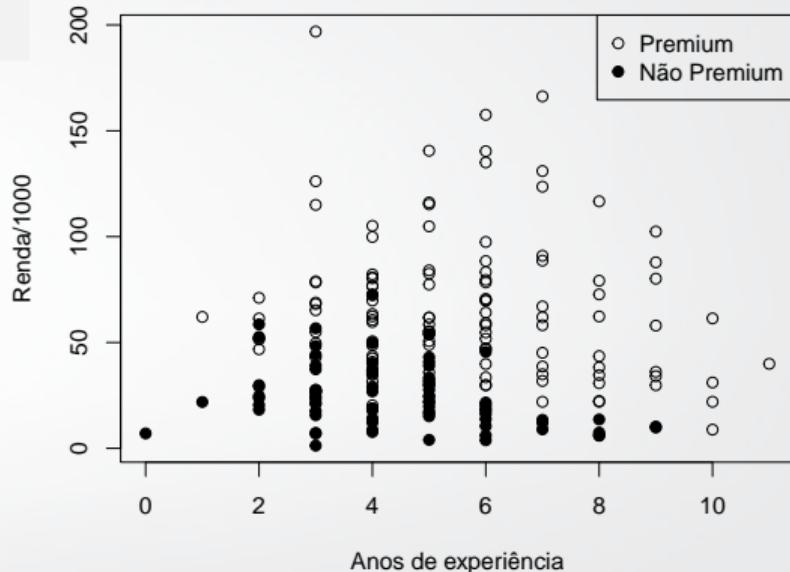
- ▶ Carregando a base e mostrando as primeiras linhas.

```
dados <- read.table("./data/reg_log.txt",
                     header = TRUE)
```

```
head(dados)
```

```
##   Premium    Renda Anos
## 1      0 18.90256     4
## 2      1 38.66267     7
## 3      1 82.16108     4
## 4      1 22.34817     8
## 5      1 36.13398     9
## 6      0 52.61761     2
```

- ▶ Graficamente



# Etapas para construir o classificador

- ▶ Especificar o modelo que descreve a relação entre  $y_i$  e  $x_{ij}$ .
- ▶ Especificar a função perda.
- ▶ Otimizar a função perda.
  - ▶ Qual algoritmo escolher?
  - ▶ Como implementá-lo?
- ▶ Analisar o modelo ajustado.



# Construindo um classificador binário

## Construindo um classificador binário

- Modelo linear

$$y_i \approx \beta_0 + \beta_1 \text{renda}_i + \beta_2 \text{anos}_i$$

- Função perda quadrática

$$SQ(\beta) = \sum_{i=1}^n (y_i - (\beta_0 + \beta_1 \text{renda}_i + \beta_2 \text{anos}_i))^2.$$

- Implementação computacional

```
f_ols <- function(par, y, renda, anos) {  
  mu <- par[1] + par[2]*renda + par[3]*anos  
  SQ <- sum( (y - mu)^2 )  
  return(SQ)  
}
```

# Construindo um classificador binário

- ▶ Otimizando os parâmetros (treinamento).

```
fit_ols <- optim(par = c(0,0,0),
                  fn = f_ols,
                  y = dados$Premium,
                  renda = dados$Renda,
                  anos = dados$Anos)
list(fit_ols$par, fit_ols$value)

## [[1]]
## [1] -0.001263985  0.007946350  0.048480489
##
## [[2]]
## [1] 29.83881
```



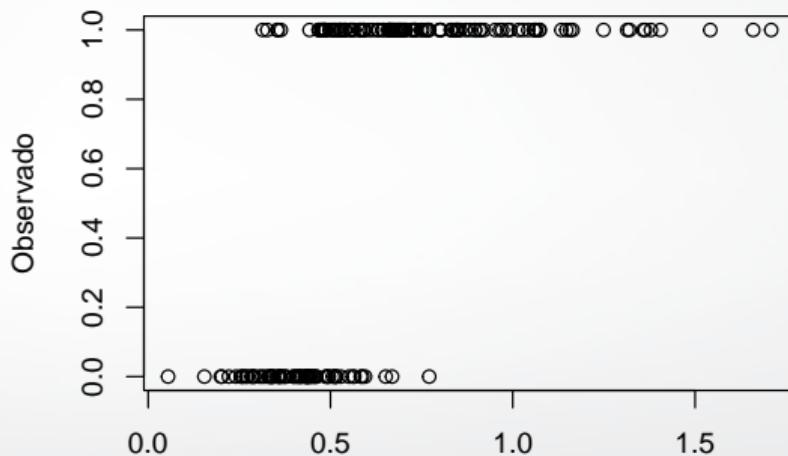
Figura 2. Foto de Mike no Pexels.

## Verificando o ajuste

- ▶ Obtendo os valores preditos.

```
preditos <- fit_ols$par[1] + fit_ols$par[2]*dados$Renda + fit_ols$par[3]*dados$Anos
```

- ▶ Gráfico dos preditos contra observados.



## Comentários

- ▶ Possível mas tem algumas inconveniências:
  - ▶ Predição resulta em valores menores que zero e maiores do que um.
  - ▶ Interpretação não é clara e não reflete a realidade de forma consistente.
  - ▶ Modelo está muito simples!



Figura 3. Foto de Kaboompics .com no Pexels.

# Melhorando o modelo

## Melhorando o modelo

- Podemos trocar o modelo linear

$$y_i \approx \beta_0 + \beta_1 \text{renda}_i + \beta_2 \text{anos}_i.$$

por um modelo mais realista.

- Modelo logístico

$$y_i \approx \frac{1}{(1 + e^{-(\beta_0 + \beta_1 \text{renda}_i + \beta_2 \text{anos}_i)})}.$$

- Graficamente

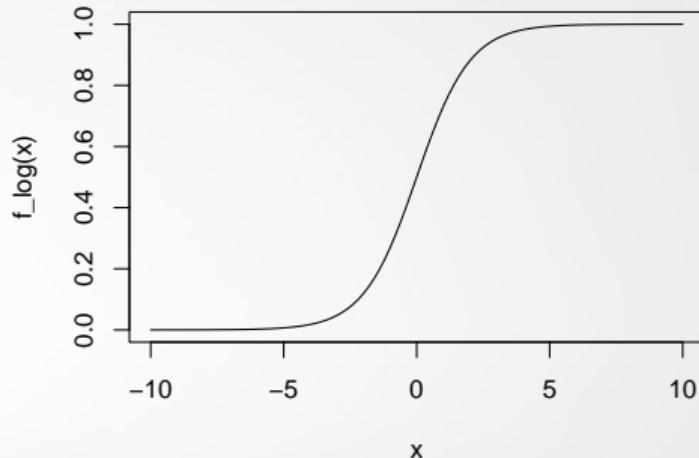


Figura 4. Função logística.

## Função perda

- Combinando o modelo logístico com a função perda quadrática, tem-se

$$SQ_{logit}(\beta) = \sum_{i=1}^n \left( y_i - \frac{1}{(1 + e^{-(\beta_0 + \beta_1 renda_i + \beta_2 anos_i)})} \right)^2.$$

- Implementação computacional.

```
f_logit <- function(par, y, renda, anos) {  
  mu <- 1/(1+ exp(-(par[1] + par[2]*renda + par[3]*anos)))  
  SQ_logit <- sum( (y - mu)^2 )  
  return(SQ_logit)  
}
```

# Otimizando os parâmetros (treinamento)

- Otimizador numérico (Nelder-Mead).

```
fit_logit_ols <- optim(par = c(0,0,0),
                        fn = f_logit,
                        y = dados$Premium,
                        renda = dados$Renda,
                        anos = dados$Anos)
list(fit_logit_ols$par, fit_logit_ols$value)

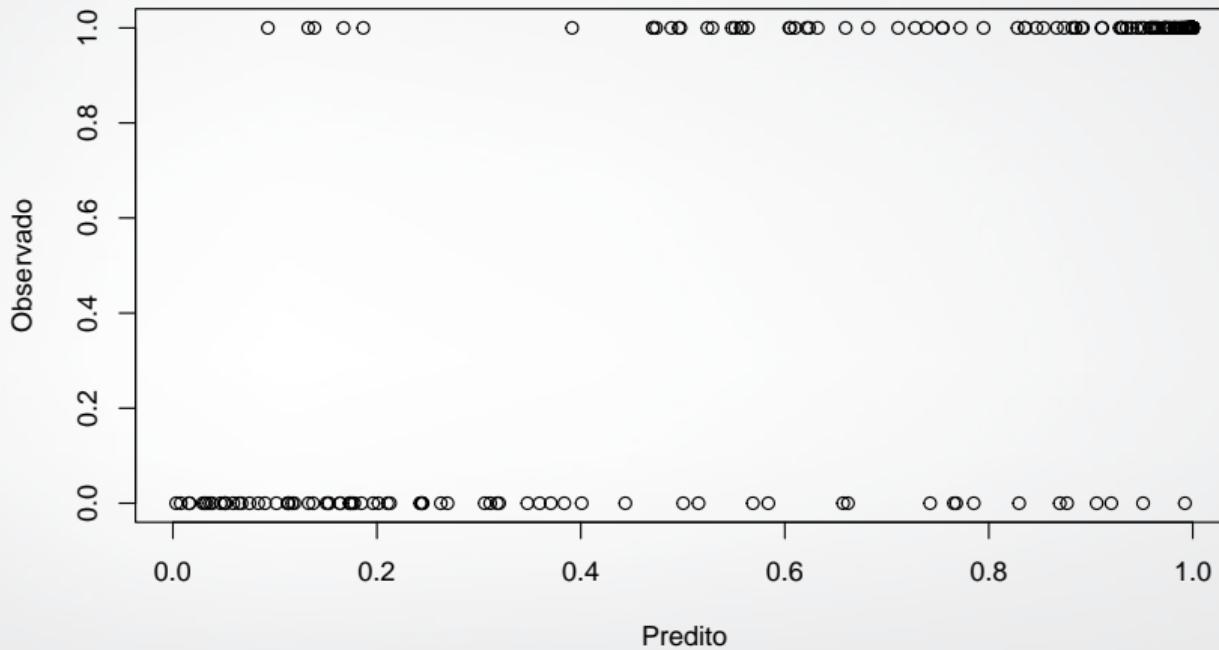
## [[1]]
## [1] -6.6424785  0.1285065  0.5390356
##
## [[2]]
## [1] 21.35787
```



Figura 5. Foto de Mike no Pexels.

## Analisando os resultados

- ▶ Plotando os preditos contra os observados.



## Comentários

- ▶ Modelos só prediz valores no intervalo  $(0,1)$ .
- ▶ Função perda quadrática trata todas as perdas de forma simétrica.
- ▶ Será que isso é razoável?
- ▶ Como lidar com essa situação? → trocar a função perda.



Figura 6. Foto de Kaboompics.com no Pexels.

# Melhorando a função objetivo

## Melhorando a função objetivo

- ▶ Distribuição Bernoulli

$$P(Y = y_i) = \mu^{y_i} (1 - \mu)^{1-y_i}.$$

- ▶ Modelando o parâmetro da distribuição

$$\mu_i = \frac{1}{1 + e^{-(\beta_0 + \beta_1 \text{renda}_i + \beta_2 \text{anos}_i)}}.$$

- ▶ Nova função objetivo (verossimilhança)

$$L(\beta) = \prod_{i=1}^n \mu_i^{y_i} (1 - \mu_i)^{1-y_i}.$$

- ▶ Mais conveniente computacionalmente (log-verossimilhança)

$$l(\beta) = \sum_{i=1}^n y_i \log(\mu_i) + (1 - y_i) \log(1 - \mu_i).$$

# Implementação computacional

- ▶ Função objetivo (maximizar).

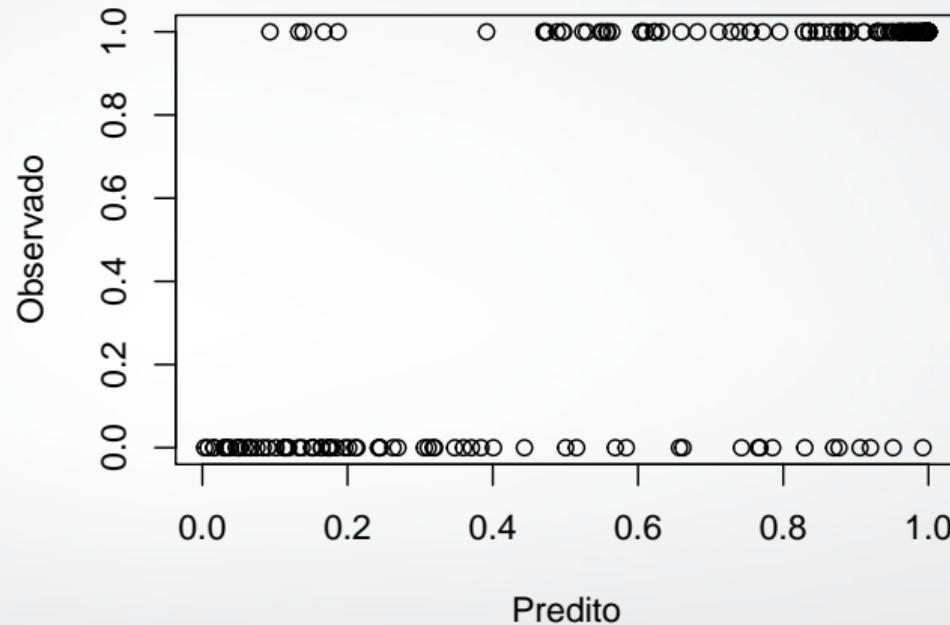
```
f_ber <- function(par, y, renda, anos) {  
  eta = par[1] + par[2]*renda + par[3]*anos  
  mu <- 1/(1+exp(-eta))  
  out <- sum(y*log(mu) + (1-y)*log(1-mu))  
  return(out)  
}
```

- ▶ Otimizando numéricamente.

```
fit_ber <- optim(par = c(0,0,0),  
                  fn = f_ber,  
                  y = dados$Premium,  
                  renda = dados$Renda,  
                  anos = dados$Anos,  
                  control = list(fnscale = -1))  
list(fit_ber$par, fit_ber$value)  
  
## [[1]]  
## [1] -6.1193677  0.1124951  0.5039379  
##  
## [[2]]  
## [1] -68.07937
```

## Analisando os resultados

- Gráfico dos preditos contra observados.



# Melhorando o algoritmo de ajuste

## Melhorando o algoritmo de ajuste

- ▶ Função objetivo

$$l(\beta) = \sum_{i=1}^n y_i \log(\mu_i) + (1 - y_i) \log(1 - \mu_i).$$

- ▶ Algoritmo de Newton

$$\beta^{(i+1)} = \beta^{(i)} - J(\beta^{(i)})^{-1} l'(\beta^{(i)}),$$

onde  $J(\beta^{(i)})$  é a matriz hessiana e  $l'(\beta^{(i)})$  é o vetor gradiente.

## Obtendo o vetor gradiente

- O vetor gradiente é dado por

$$l'(\beta^{(i)}) = \left( \frac{\partial l(\beta)}{\partial \beta_0}, \frac{\partial l(\beta)}{\partial \beta_1}, \frac{\partial l(\beta)}{\partial \beta_2} \right)^\top.$$

- Para simplificar a notação, seja  $\eta_i = \beta_0 + \beta_1 \text{renda}_i + \beta_2 \text{Anos}_i$ .
- Assim, tem-se

$$\frac{\partial l(\beta)}{\partial \beta_j} = \frac{\partial l(\beta)}{\partial \mu} \frac{\partial \mu}{\partial \eta} \frac{\partial \eta}{\partial \beta_j}, \quad \text{para } j = 1, 2, 3.$$

- Obtendo cada uma das entradas, tem-se

$$\frac{\partial l(\beta)}{\partial \mu} = \frac{\partial}{\partial \mu} y_i \log(\mu_i) + (1 - y_i) \log(1 - \mu_i) = \frac{y_i}{\mu_i} - \frac{(1 - y_i)}{(1 - \mu_i)} = \frac{y_i - \mu_i}{\mu_i(1 - \mu_i)}.$$

## Obtendo o vetor gradiente

- Derivando  $\mu$  em relação ao  $\eta$ , tem-se

$$\frac{\partial \mu}{\partial \eta} = \frac{\partial}{\partial \eta} \frac{1}{1 + e^{-\eta}} = \frac{e^{-\eta}}{(1 + e^{-\eta})^2} = \mu_i(1 - \mu_i).$$

- Derivando  $\eta$  em relação aos  $\beta$ 's, tem-se

$$\frac{\partial \eta}{\partial \beta_0} = 1, \quad \frac{\partial \eta}{\partial \beta_1} = renda_i \quad \text{e} \quad \frac{\partial \eta}{\partial \beta_2} = anos_i.$$

- Finalmente, temos o vetor gradiente

$$l'(\beta^{(i)}) = \left( \sum_{i=1}^n (y_i - \mu_i) 1, \sum_{i=1}^n (y_i - \mu_i) renda_i, \sum_{i=1}^n (y_i - \mu_i) anos_i \right)^\top.$$

- Cálculos similares levam a

$$J(\beta) = - \begin{bmatrix} \mu_i(1 - \mu_i)1 & \mu_i(1 - \mu_i)renda_i & \mu_i(1 - \mu_i)anos_i \\ \mu_i(1 - \mu_i)renda_i & \mu_i(1 - \mu_i)renda_i^2 & \mu_i(1 - \mu_i)renda_i anos_i \\ \mu_i(1 - \mu_i)anos_i & \mu_i(1 - \mu_i)renda_i anos_i & \mu_i(1 - \mu_i)anos_i^2 \end{bmatrix}.$$

# Implementação computacional

```
# Gradiente
gradiente <- function(par, y, renda, anos) {
  eta = par[1] + par[2]*renda + par[3]*anos
  mu <- 1/(1+exp(-eta))
  db0 <- sum((y - mu))
  db1 <- sum((y-mu)*renda)
  db2 <- sum((y-mu)*anos)
  out <- c(db0, db1, db2)
  return(out)
}
```

```
# Hessiana
hessiano <- function(par, y, renda, anos) {
  J <- matrix(NA, ncol = 3, nrow = 3)
  eta = par[1] + par[2]*renda + par[3]*anos
  mu <- 1/(1+exp(-eta))
  J[1,1] <- sum(mu*(1-mu))
  J[1,2] <- sum(mu*(1-mu)*renda)
  J[1,3] <- sum(mu*(1-mu)*anos)
  J[2,1] <- sum(mu*(1-mu)*renda)
  J[2,2] <- sum(mu*(1-mu)*(renda^2) )
  J[2,3] <- sum(mu*(1-mu)*renda*anos)
  J[3,1] <- sum(mu*(1-mu)*anos)
  J[3,2] <- sum(mu*(1-mu)*(renda*anos))
  J[3,3] <- sum(mu*(1-mu)*(anos^2))
  return(-J)
}
```

## Relembrando: Método de Newton

- Método de Newton.

```
newton <- function(fx, jacobian, x1, tol = 1e-04, max_iter = 10, ...) {  
  solucao <- matrix(NA, ncol = length(x1), nrow = max_iter)  
  solucao[1,] <- x1  
  for(i in 1:max_iter) {  
    J <- jacobian(solucao[i,], ...)  
    grad <- fx(solucao[i,], ...)  
    solucao[i+1,] = solucao[i,] - solve(J, grad)  
    if( sum(abs(solucao[i+1,] - solucao[i,])) < tol) break  
  }  
  return(solucao)  
}
```

## Ajustando os parâmetros (treinamento)

### ► Método de Newton.

```
newton(fx = gradiente, jacobian = hessiano,  
       x1 = c(0,0,0), y = dados$Premium,  
       renda = dados$Renda, anos = dados$Anos)
```

```
##          [,1]      [,2]      [,3]  
## [1,]  0.000000  0.0000000  0.0000000  
## [2,] -2.004560  0.03178176 0.1938505  
## [3,] -3.622509  0.06384417 0.3023678  
## [4,] -5.112017  0.09318818 0.4179397  
## [5,] -5.942349  0.10917130 0.4881873  
## [6,] -6.116275  0.11242702 0.5036863  
## [7,] -6.122200  0.11253511 0.5042327  
## [8,] -6.122206  0.11253523 0.5042333  
## [9,]        NA        NA        NA  
## [10,]       NA       NA       NA
```

# Clusterização usando *kmeans*

## Clusterização usando kmeans

- ▶ Tarefa de agrupar indivíduos semelhantes é muito comum.
    - ▶ Segmentação de clientes.
    - ▶ Separar músicas por gênero.
    - ▶ Separar vídeos por faixa etária.
  - ▶ Métodos de aprendizagem não-supervisionada.
  - ▶ Método *kmeans* é muito popular.
- ▶ Intuição do método.
    - ▶ Dado um vetor de características criar grupos.
    - ▶ Indivíduos dentro do mesmo grupo são mais parecidos do que indivíduos em grupos diferentes.
    - ▶ Medida resumo de cada grupo a média amostral.
    - ▶ Objetivo é fazer com que cada indivíduo pertença ao grupo com a média mais próxima à ele.

## Aplicação

- ▶ Duas variáveis contínuas  $x_1$  e  $x_2$ .
- ▶ Observadas em  $n = 300$  indivíduos.
- ▶ Objetivo: agrupar indivíduos em  $k$  grupos de forma que a distância de cada indivíduo ao centróide de seu grupo seja a menor possível.

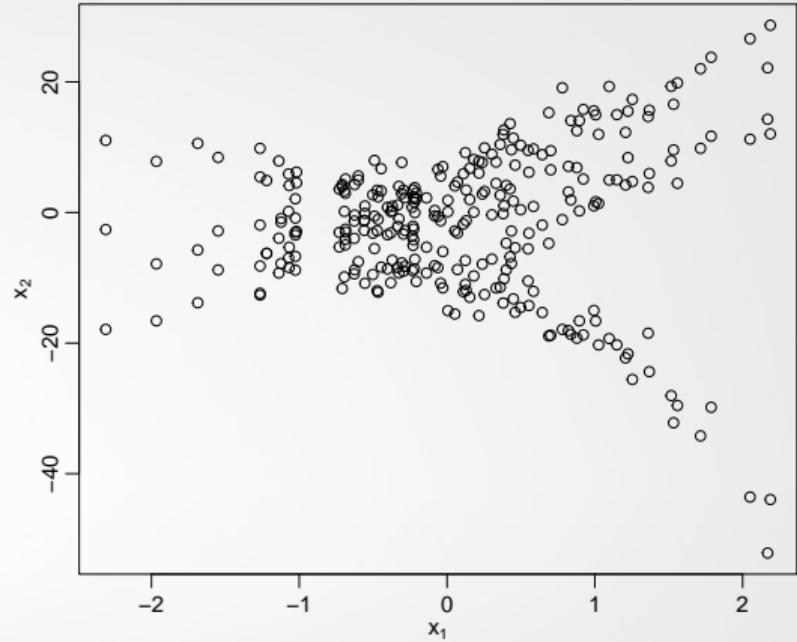


Figura 7. Ilustração de conjunto de dados para uso do método kmeans.

## Descrição matemática

- ▶ Denote  $(x_1, \dots, x_n)$  um vetor  $p$ -dimensional de valores reais.
- ▶ Objetivo do método *kmeans* é particionar as  $n$  observações em  $k < n$  grupos, digamos  $G = \{G_1, \dots, G_k\}$ .
- ▶ Minimizando a soma de quadrados dentro de cada grupo.
- ▶ Em termos de notação matemática, temos

$$\arg \min_G \sum_{i=1}^k \sum_{x \in G_i} (x - \mu_i)^2.$$

- ▶ Não é um problema de otimização usual!

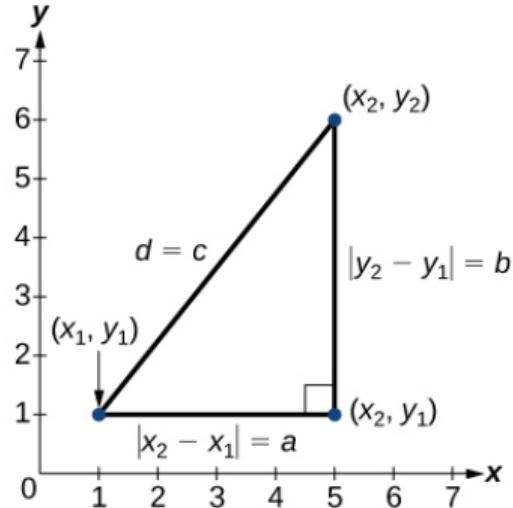
## Algoritmo de refinamento

- ▶ Dado um conjunto inicial de  $k$  médias, digamos,  $\mu_1^{(1)}, \mu_2^{(1)}, \dots, \mu_k^{(1)}$  o algoritmo alterna entre dois passos:
  1. Passo de agrupamento: agrupa cada observação ao grupo que tem a média mais próxima baseado em alguma medida de distância, sendo a distância Euclidiana a mais comum.
  2. Passo de atualização: recalcula as médias (centróides) dado o agrupamento do Passo 1.
- ▶ Até que algum critério de parada seja atingido.

# Implementação computacional

- ▶ Começamos com a distância Euclidiana.

```
my_dist <- function(dados, media) {  
  out <- apply(dados, 1, function(x, media) {  
    sqrt(sum((x - media)^2))  
  }, media = media)  
  return(out)  
}
```



## Implementação didática: Método kmeans

```
my_kmeans <- function(data, k, max_iter = 20, set.seed = 123) {  
  # Inicializando  
  n <- dim(data)[1]  
  data_temp <- data  
  ## Inicializando os grupos de forma aleatórias  
  set.seed(set.seed)  
  data_temp$grupo <- sample(1:k, size = n, replace = TRUE)  
  # Cria matriz para guardar os grupos intermediários  
  grupos <- matrix(NA, ncol = max_iter, nrow = n)  
  # Primeiro grupo  
  grupos[,1] <- data_temp$grupo  
  n_col <- dim(data)[2] # Número de variáveis para fazer o agrupamento  
  criterio <- c()  
  criterio[1] <- n # inicia o critério de parada com n trocas  
  SOMA_QQ <- matrix(NA, nrow = max_iter, ncol = k)
```

## Implementação didática: Método kmeans (cont.)

```
for(i in 2:max_iter) {  
  media_grupo <- aggregate(data, list(grupos[,c(i-1)]), mean)  
  temp <- list()  
  for(j in 1:k) {  
    temp[[j]] <- my_dist(dados = data, media = media_grupo[j, 2:c(n_col+1)])  
  }  
  temp <- do.call(cbind, temp)  
  SOMA_QQ[i,] <- colSums(temp)  
  grupos[,i] <- as.factor(apply(temp, 1, which.min)) # Escolhe a qual grupo a observação pertence  
  data_temp$grupo <- grupos[,i]  
  criterio[i] <- sum(abs(grupos[,i] - grupos[,c(i-1)]))  
  if(abs(criterio[i] - criterio[c(i-1)]) == 0) break  
  print(criterio[i])  
}  
return(list("grupos" = grupos, "data" = data_temp, "criterio" = criterio,  
          "centers" = media_grupo, "GOF" = SOMA_QQ))  
}
```

## Aplicando o método

- ▶ Usando a função criada para construir 3 grupos.

```
resultado <- my_kmeans(data = dados, k = 3)
```

```
## [1] 281  
## [1] 88  
## [1] 31  
## [1] 14  
## [1] 12  
## [1] 6  
## [1] 3  
## [1] 4  
## [1] 3
```

- ▶ Resultado.

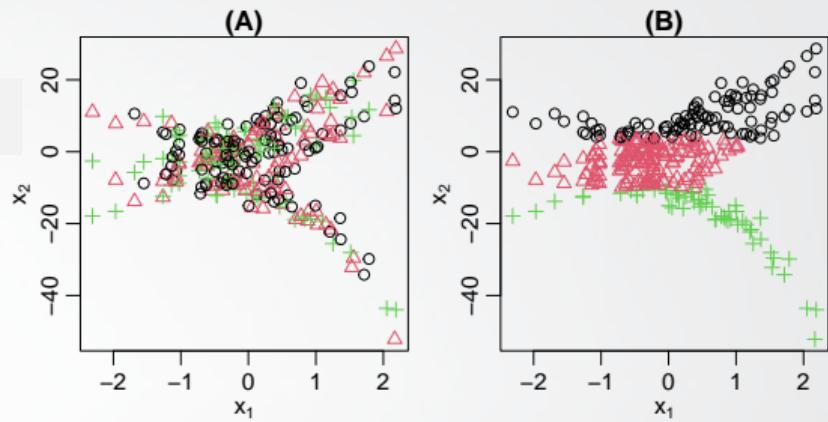


Figura 8. Ilustração do método kmeans, solução inicial (A) e solução final (B).

# Redes neurais artificiais

## Redes neurais artificiais

- ▶ Umas das principais técnicas em uso por cientistas de dados.
- ▶ Principal força de trabalho dentro da grande área de Inteligência Artificial.
- ▶ Aplicações de destaque:
  - ▶ Reconhecimento de imagens.
  - ▶ *Chatbots*.
  - ▶ Carros autônomos.
  - ▶ Assistentes virtuais.
  - ▶ Reconhecimento facial.
  - ▶ Processamento da linguagem natural.
- ▶ *Multilayer perceptron*.

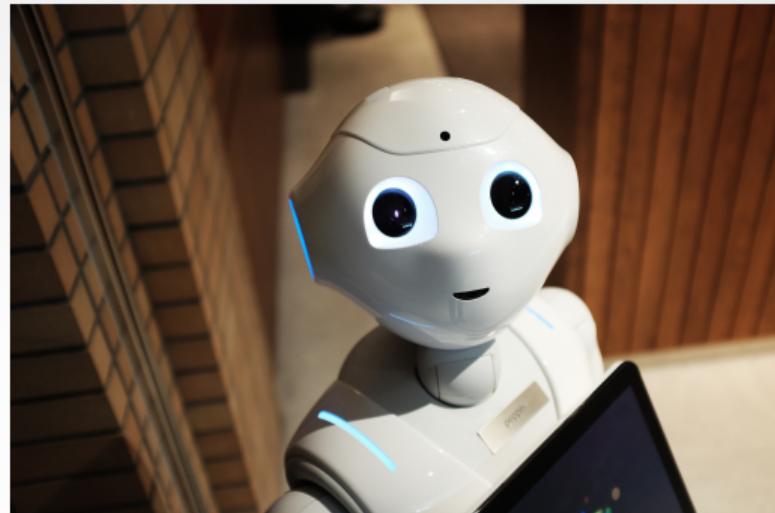


Figura 9. Foto de Alex Knight no Pexels.

## Ingredientes de uma rede neural

1. Dados de entrada ou *input*.
2. Exemplos da saída esperada ou *output*.
3. Uma forma de medir se o algoritmo é capaz de reproduzir os exemplos de saída.
4. Alguma forma de treinamento: gradiente descendente e variações.



## Estrutura de uma rede neural

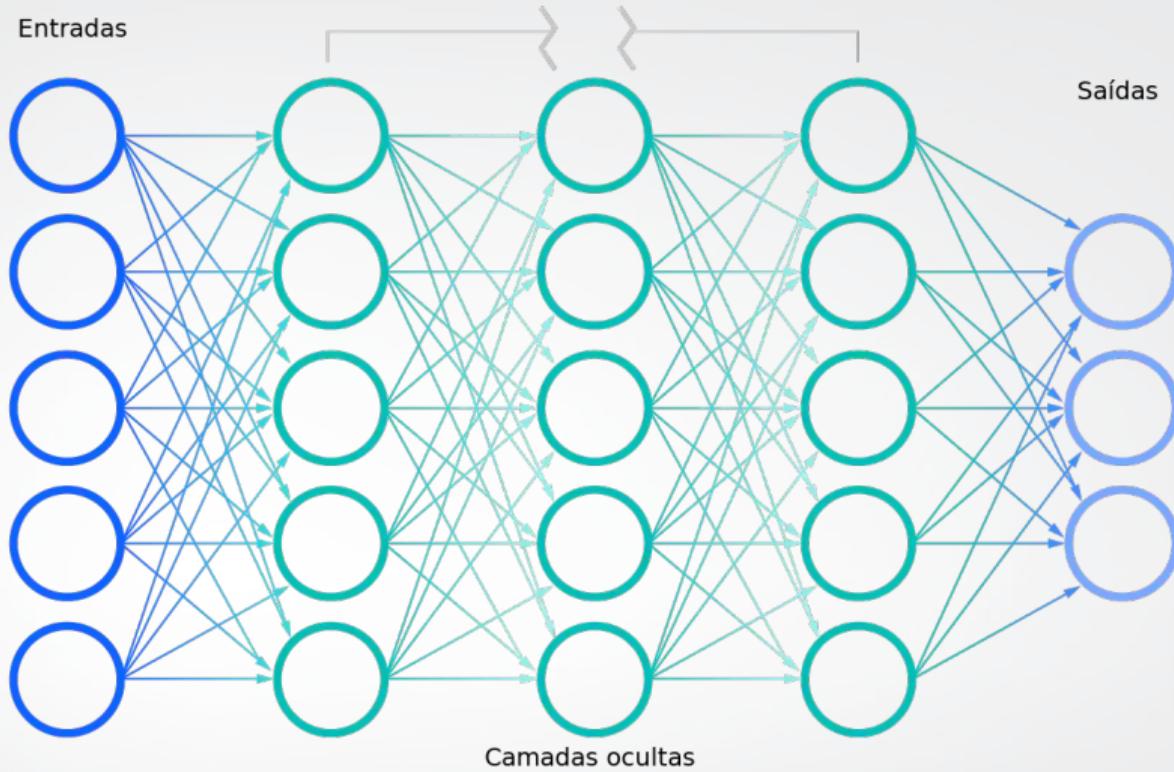
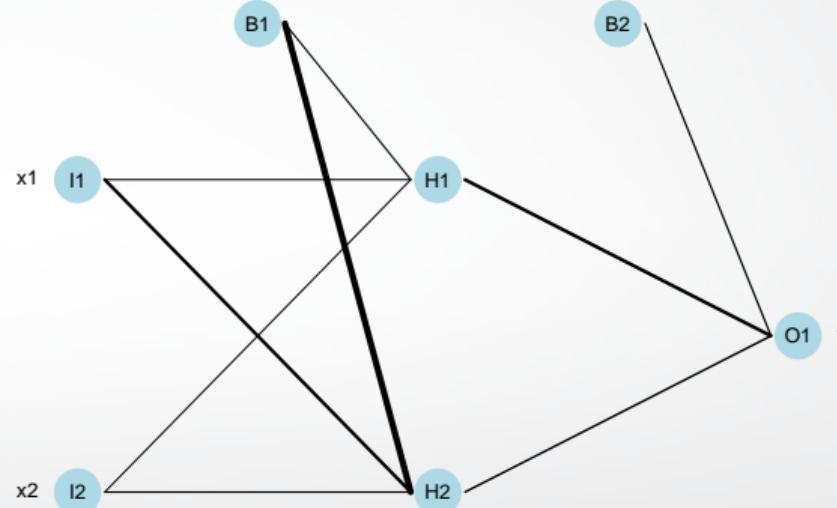


Figura 11. Desenho esquemático da estrutura de uma rede neural.

## Estrutura de uma rede neural

- ▶ Considere que temos duas variáveis de entrada  $x_1$  e  $x_2$ .
- ▶ Uma variável de saída contínua.
- ▶ Uma camada oculta e dois neurônios.



## Descrição matemática

- Passo 1 - *Inputs*

$$\eta_1 = w_{01} + w_{11}x_1 + w_{12}x_2 \quad \text{e} \quad \eta_2 = w_{02} + w_{21}x_1 + w_{22}x_2.$$

- Passo 2 - Função de ativação

$$h_1 = \frac{1}{1 + e^{-\eta_1}} \quad h_2 = \frac{1}{1 + e^{-\eta_2}}.$$

- Passo 3 - Adiciona o *bias* e tem a saída

$$\hat{y} = w_{0O} + w_{1O}h_1 + w_{2O}h_2.$$

# Implementação computacional

- ▶ Carregando um conjunto de dados simulado.

```
dados <- read.table("data/NN.txt", header = TRUE, sep = ";")  
head(dados)
```

```
##           x1          x2          y  
## 1 -0.56047565 -0.71040656 21.83917  
## 2 -0.23017749  0.25688371 29.90718  
## 3  1.55870831 -0.24669188 31.13087  
## 4  0.07050839 -0.34754260 33.93173  
## 5  0.12928774 -0.95161857 25.43137  
## 6  1.71506499 -0.04502772 26.23336
```

# Processo feedforward

```
# Variáveis de entrada
x <- dados[1, 1:2]
# Variável de saída
y <- dados[1, 3]

# Primeiro neurônio
w01 <- 7.20; w11 <- 6.31; w12 <- 4.73
eta1 <- w01 + w11*x[1] + w12*x[2]

# Segundo neurônio
w02 <- 2.54; w21 <- 1.59; w22 <- 1.52
eta2 <- w02 + w21*x[1] + w22*x[2]

## Passando pela camada oculta
h1 <- 1/(1+exp(-eta1))
h2 <- 1/(1+exp(-eta2))

# Gerando a saída
w00 <- 9.21; w10 <- 9.41; w20 <- 11.65
y_hat <- w00 + w10*h1 + w20*h2

# Comparando com o dado observado
c(y, y_hat)
## [[1]]
## [1] 21.83917
##
## $x1
## [1] 22.06178
```

## Ajustando os parâmetros (treinamento)

- ▶ Parâmetros (pesos) da rede neural:  $\mathbf{w} = (w_{01}, w_{11}, w_{12}, w_{02}, w_{21}, w_{22}, w_{0O}, w_{1O}, w_{2O})^\top$ .
- ▶ Função perda quadrática fica dada por

$$\text{SQ}(\mathbf{w}) = \sum_{i=1}^n (y_i - \hat{y}_i)^2,$$

- ▶ Queremos minizar a soma de quadrados dos erros.

# Implementação mais genérica

- ▶ Função de ativação e suas derivadas.

```
act_fc <- function(pesos, X, derivative = FALSE) {  
  eta <- X%*%pesos  
  mu <- 1/(1+exp(-eta))  
  output <- list("mu" = mu)  
  if(derivative == TRUE) {  
    derivada <- mu*(1-mu)  
    D <- list()  
    for(i in 1:dim(derivada)[2]) {  
      D[[i]] <- derivada[,i]*X  
    }  
    output$D_mu <- D  
  }  
  return(output)  
}
```

# Implementação mais genérica

- Passo *feedforward*.

```
neural_net <- function(w, X, n_neuronio) {  
  n_weight <- dim(X)[2] ## Qts pesos em cada neurônio  
  n_param <- length(w) ## Qtd total de parâmetros  
  w_E <- w[1:c(n_neuronio*n_weight)]  
  W_E <- matrix(w_E, ncol = n_neuronio) # Forma matricial  
  w_h <- w[c(n_neuronio*n_weight+1):n_param]  
  mu <- act_fc(pesos = W_E, X = X)  
  H <- model.matrix(~ mu$mu)  
  y_hat <- H%*%w_h  
  return(y_hat)  
}
```

## Ilustração do passo feedforward

- Passo *feedforward*.

```
X <- model.matrix(~ x1 + x2, data = dados) # Cria a matriz X
head(X)
```

```
##   (Intercept)      x1      x2
## 1           1 -0.56047565 -0.71040656
## 2           1 -0.23017749  0.25688371
## 3           1  1.55870831 -0.24669188
## 4           1  0.07050839 -0.34754260
## 5           1  0.12928774 -0.95161857
## 6           1  1.71506499 -0.04502772
```

```
y_hat <- neural_net(w = w, X = X, n_neuronio = 2)
head(y_hat, 4)
```

```
##      [,1]
## 1 22.05528
## 2 29.43020
## 3 30.15612
## 4 29.00182
```

# Implementação da função perda quadrática

- ▶ Função perda quadrática.

```
SQ <- function(w, X, Y, n_neuronio) {  
  y_hat <- neural_net(w = w, X = X, n_neuronio = n_neuronio)  
  out <- 0.5*sum((Y - y_hat)^2)  
  return(out)  
}
```

## Ajustando os parâmetros (treinamento)

### ► Algoritmos convencionais

```
fit_NM <- optim(par = w, fn = SQ, method = "Nelder-Mead",
                  X = X, Y = dados$y, n_neuronio = 2)
fit(CG <- optim(par = w, fn = SQ, method = "CG",
                  X = X, Y = dados$y, n_neuronio = 2)
fit_BFGS <- optim(par = w, fn = SQ, method = "BFGS",
                   X = X, Y = dados$y, n_neuronio = 2)
fit_SANN <- optim(par = w, fn = SQ, method = "SANN",
                   X = X, Y = dados$y, n_neuronio = 2)
```

## Comparando os resultados

- ▶ Resultados usando diferentes algoritmos de minimização.

```
resultado <- data.frame("Nelder-Mead" = fit_NM$value,
                        "Gradiente Conjugado" = fit(CG$value,
                        "BFGS" = fit_BFGS$value,
                        "SANN" = fit_SANN$value)
resultado
##   Nelder.Mead Gradiente.Conjugado      BFGS      SANN
## 1     663.9797          658.6414 651.481 658.3407
```

# Gradiente

- ▶ Implementação do gradiente da função perda no caso da rede neural.

```
gradiente <- function(w, X, Y, n_neuronio = 2) {  
  n_weight <- dim(X)[2] ## Qts pesos em cada neurônio  
  n_param <- length(w) ## Qtd total de parâmetros  
  w_E <- w[1:c(n_neuronio*n_weight)];  
  W_E <- matrix(w_E, ncol = n_neuronio) # Forma matricial  
  w_h <- w[c(n_neuronio*n_weight+1):n_param];  
  mu <- act_fc(pesos = W_E, X = X, derivative = TRUE)  
  H <- model.matrix(~ mu$mu);  y_hat <- H%*%w_h  
  res <- as.numeric(-(Y - y_hat));  grad <- list()  
  for(i in 1:c(length(w_h)-1)) {  
    grad[[i]] <- res*mu$D_mu[[i]]*w_h[c(i+1)]  
  }  
  output <- colSums(cbind(do.call(cbind, grad), res*H))  
  return(output)  
}
```

# Relembrando: gradiente descendente

## ► Gradiente descendente

```
grad_des <- function(fx, x1, alpha, max_iter = 1e+05, tol = 1e-02, ...) {  
  solucao <- matrix(NA, ncol = length(x1), nrow = max_iter)  
  solucao[1,] <- x1  
  obj <- c()  
  for(i in 1:c(max_iter-1)) {  
    solucao[i+1,] <- solucao[i,] - alpha*fx(solucao[i,], ...)  
    obj[c(i+1)] <- SQ(w = solucao[i+1,], ...)  
    if( sum(abs(solucao[i+1,] - solucao[i, ])) < tol) break  
  }  
  return(list("solucao" = solucao, "Objetivo" = obj,  
            "SolucaoOptima" = solucao[i+1,], "Optimo" = obj[c(i+1)]))  
}
```

## Ajustando os parâmetros (treinamento)

- ▶ Treinando o modelo.

```
fit_grad_des <- grad_des(fx = gradiente, x1 = w, alpha = 0.005, tol = 1e-04,  
                         max_iter = 1e+05, X = X, Y = dados$y, n_neuronio = 2)
```

- ▶ Visualizando o processo de ajuste.

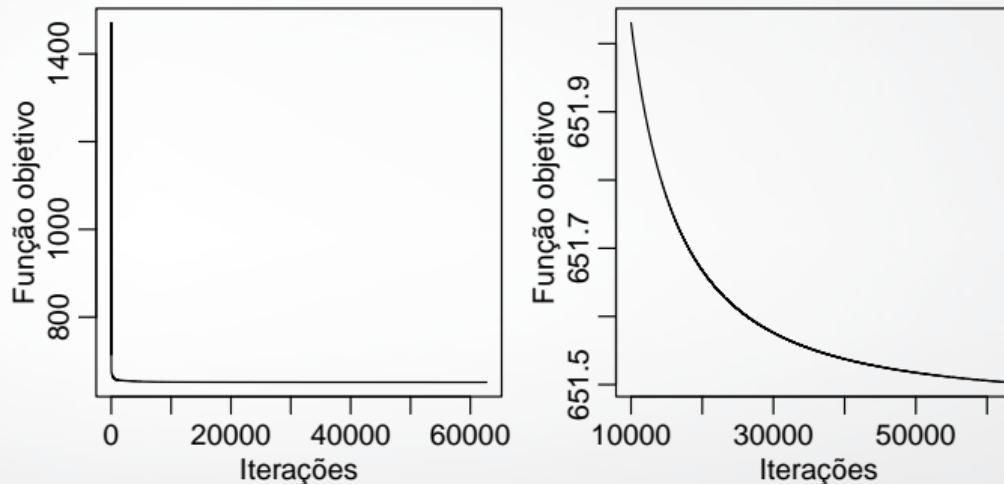


Figura 13. Progresso do algoritmo gradiente descendente.

## Ajuste via neuralnet

### ► Pacote padrão R

```
require(neuralnet)
fit_nn <- neuralnet(formula = y ~ x1 + x2, hidden = 2,
                      startweights = w,
                      algorithm = "backprop", learningrate = 0.005,
                      linear.output = TRUE, data = dados)
nn_w <- c(fit_nn$weights[[1]][[1]][,1],
          fit_nn$weights[[1]][[1]][,2],
          fit_nn$weights[[1]][[2]])
```

## Comparando os resultados

- ▶ Resultados usando diferentes esquemas de treinamento.

```
resultado <- data.frame("Nelder-Mead" = fit_NM$value,
                        "Gradiente Conjugado" = fit(CG$value,
                        "BFGS" = fit_BFGS$value,
                        "SANN" = fit_SANN$value,
                        "GradDes" = fit_grad_des$Otimo,
                        "Neuralnet" = SQ(w = nn_w, X = X, Y = dados$y, n_neuronio = 2))
resultado

##   Nelder.Mead Gradiente.Conjugado      BFGS       SANN  GradDes
## 1     663.9797          658.6414 651.481 658.3407 651.5035
##   Neuralnet
## 1     651.5001
```

# Discussão

## Discussão

- ▶ Técnicas com diferentes objetivos, mas todas seguem as mesmas premissas:
  - ▶ Especificar um modelo.
  - ▶ Escolher uma função objetivo.
  - ▶ Usar algum esquema numérico para ajustar os parâmetros do modelo.
  - ▶ Analisar os resultados.
- ▶ Uso intenso de **funções**.
- ▶ Analisar o comportamento de funções → Cálculo Diferencial e Integral.
- ▶ Muitas observações → Álgebra Matricial.
- ▶ Processo de treinamento → Métodos Numéricos.

# Projeto integrador

## Ajustando modelos não-lineares

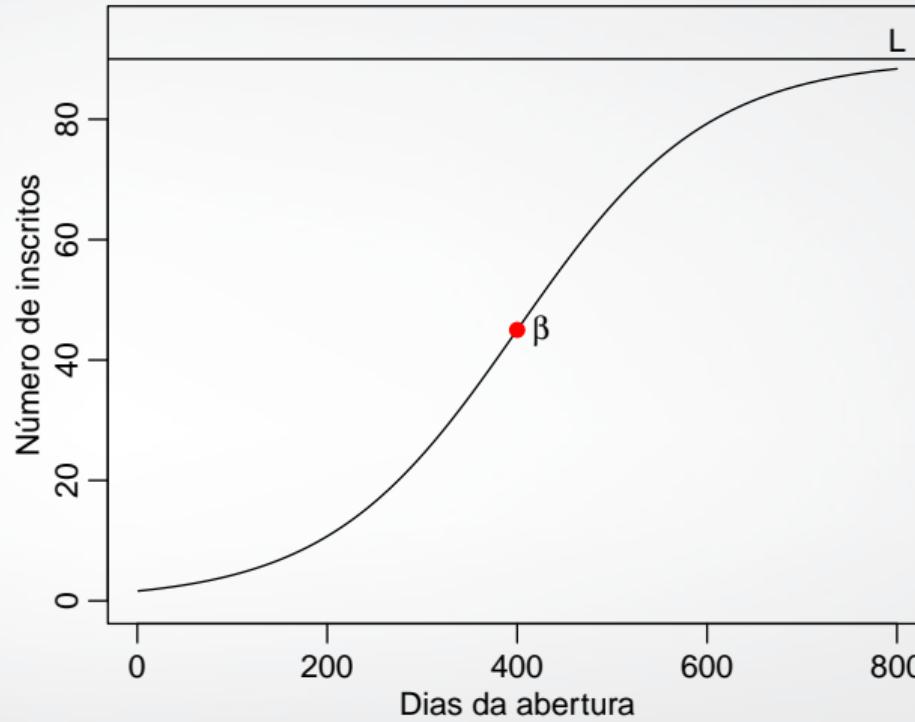
- ▶ O conjunto de dados `youtube.csv` apresenta o número de *views* e inscritos desde o dia de sua abertura para dois canais de sucesso do *youtube*.
- ▶ Objetivo: predizer o número acumulado de inscritos em cada um destes canais para o próximo ano (365 dias).
- ▶ Para isto você decidiu emprestar um modelo biológico do crescimento de bactérias chamado modelo logístico, dado pela seguinte equação:

$$y = \frac{L}{1 + \exp(\beta(x - \beta_0))}$$

onde  $L$  é o valor máximo da curva,  $\beta_0$  o valor de  $x$  no ponto médio da curva e  $\beta$  é a declividade da curva.

## Ajustando modelos não-lineares

- ▶ Curva logística



## Ajustando modelos não-lineares

- ▶ Proponha e descreva um algoritmo para ajustar este modelo aos dados disponíveis.
- ▶ Ajuste o modelo aos dados dos canais e reporte a sua predição de forma gráfica.

