Capítulo 4
Exemplos integradores

Neste Capítulo vamos apresentar a construção detalhada de três técnicas populares em ciência de dados: regressão logística, clusterização usando o algoritmo *kmeans* e redes neurais artificiais. O objetivo é mostrar que apesar das técnicas serem diferentes todas seguem as mesmas premissas e são apenas representações matemáticas para diferentes tipos de problemas. De forma geral, todas são uma certa combinação de funções que devem ser otimizadas em algum sentido. Importante ressaltar que o objetivo é apenas ilustrar como as diversas ferramentas matemáticas apresentadas no decorrer do livro são aplicadas simultaneamente e não no uso prático das respectivas técnicas.

4.1
Construindo um classificador binário

Classificadores binários são ferramentas populares em estatística e aprendizagem de máquina. O objetivo de um classificador é, como o nome já diz, classificar um indivíduo ou observação como pertencente a um entre dois grupos. Por exemplo, dado um certo conjunto de exames clínicos classificar um paciente como sadio ou doente. Dentre os vários classificadores, a regressão logística é um dos mais simples e populares e se aplica a uma variável de interesse (resposta) do tipo binária. As aplicações são as mais diversas possíveis cobrindo desde aplicações na indústria onde itens são classificados como conforme ou não-conforme até a medicina onde se estuda a presença ou ausência de uma certa doença e como ela se relaciona com hábitos de vida dos pacientes, tais como sexo, idade, tabagismo entre outros.

O objetivo desta seção é discutir apenas os aspectos matemáticos relacionados à construção de um classificador binário. Mais especificamente, vamos discutir como, a partir da ideia simples do modelo de regressão linear múltipla, podemos chegar ao modelo de regressão logística. Os aspectos práticos desta técnica não serão discutidos neste livro.

A especificação e implementação do modelo de regressão logística envolve uma série de conhecimentos em métodos matemáticos que começam em funções, passando por cálculo diferencial onde a necessidade de resolver um sistema não-linear fica evidente. No processo de solução de sistemas não-lineares aparecem de forma proeminente operações com vetores, matrizes, obtenção de inversas e de forma mais geral a solução de sistemas lineares. Entretanto, todo o processo de ajuste ou treinamento do modelo pode ser descrito como um simples problema de programação não-linear.

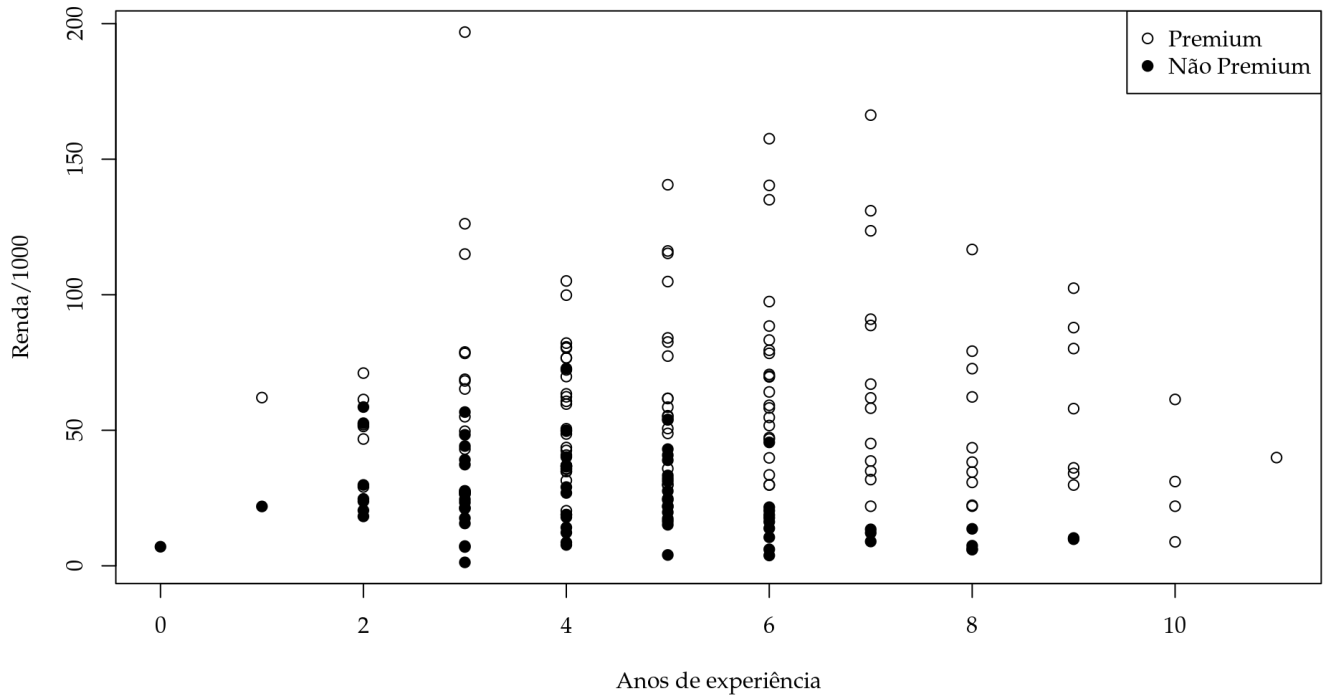
4.1.1
Descrição do problema

Suponha que temos um conjunto de observações y_i com $i = 1, \ldots, n$ de uma variável de interesse do tipo binária, ou seja, $y_i \in [0, 1]$. O objetivo pode ser descrever o relacionamento de y_i com um conjunto de variáveis explanatórias, digamos $\boldsymbol{x}_{ij}$, ou mesmo apenas classificar uma nova observação em um dos grupos 0 ou 1 de acordo com os valores das variáveis explanatórias.

Como um exemplo de aplicação, suponha que temos um conjunto de dados com três colunas: renda do usuário (por mil), anos de experiência do usuário e se o usuário é assinante *premium* ou não. Nosso objetivo é descrever como as variáveis explanatórias (renda e anos de experiência do usuário) estão relacionados com o fato dele ser ou não um assinante *premium*. Uma vez identificado tal relacionamento, podemos usá-lo para predizer se um novo usuário será ou não assinante *premium*, bem como, identificar dentro da base, usuários que podem se interessar pela assinatura *premium*. O resultado também pode ser usado para orientar campanhas de marketing, entre outros. A base de dados é fornecida no arquivo *reg_log.txt* e as primeiras linhas são mostradas abaixo.

```
##   Premium Renda Anos
## 1       0  18.9    4
## 2       1  38.7    7
## 3       1  82.2    4
## 4       1  22.3    8
## 5       1  36.1    9
## 6       0  52.6    2
```

Podemos visualizar os dados plotando as variáveis explanatórias e marcar com diferentes símbolos usuários *premium* e não *premium*.



4.1.2

Função perda quadrática para variáveis binárias

A ideia de construir um classificador é similar ao modelo de regressão linear múltipla, ou seja, queremos minimizar uma certa distância entre os dados y_i e nosso modelo linear $\beta_0 + \beta_1 renda_i + \beta_2 anos_i$. Sendo assim, uma primeira tentativa óbvia é minimizar

$$SQ(\boldsymbol{\beta}) = \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 renda_i + \beta_2 anos_i))^2.$$

Isso pode ser feito facilmente em R uma vez que temos vários otimizadores numéricos disponíveis. O primeiro passo é escrever a função objetivo:

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

O segundo passo consiste em otimizar a função objetivo. Por simplicidade vamos usar um algoritmo numérico, porém lembre-se que neste caso temos expressões fechadas disponíveis.

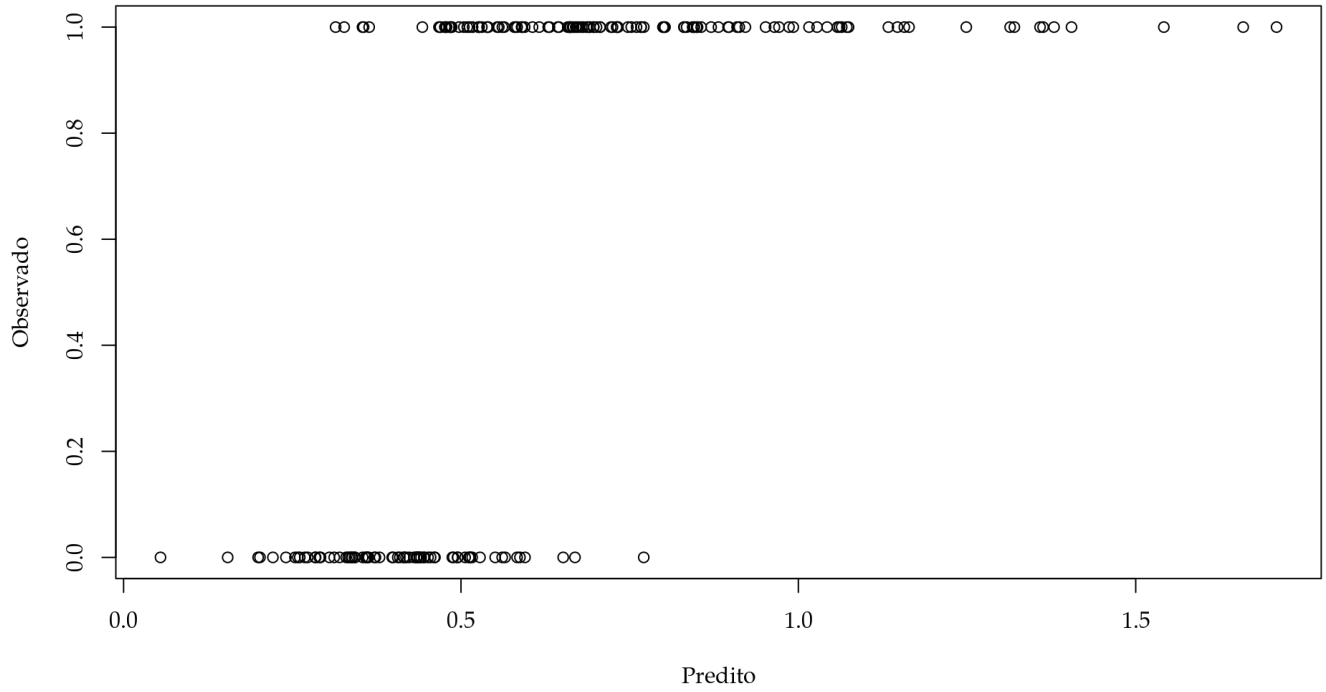
```
fit_ols <- optim(par = c(0,0,0), fn = f_ols, y = dados$Premium,
                 renda = dados$Renda, anos = dados$Anos)
fit_ols
```

```
## $par
## [1] -0.00126  0.00795  0.04848
##
## $value
## [1] 29.8
##
## $counts
## function gradient
##      124       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

O algoritmo convergiu e forneceu o vetor de β's que minimiza a soma de quadrados. Podemos agora usar estes valores para predizer se um usuário é ou não *premium* dado sua renda e anos de experiência. Neste caso, apenas como exemplo vamos predizer os mesmos usuários que foram usados para ajustar/treinar o modelo. Note que isso também indica se o modelo tem um bom poder de predição. Entretanto, na prática, é mais usual separar a base em treino e teste e predizer os usuários na base teste. A predição é bastante simples: entramos com a renda e anos de estudo e saímos com uma predição para y_i.

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

Vamos plotar os preditos contra os observados para verificar a performance do nosso modelo.



Apesar de ser possível, essa abordagem leva a algumas inconveniências. Por exemplo, o ideal seria que as previsões fossem 0 ou 1 refletindo se o novo usuário será ou não um assinante *premium*. No entanto, vimos que o modelo fornece preditos maiores que 1 e pode potencialmente fornecer preditos negativos para usuários com pouca experiência e renda menor. Neste sentido, a interpretação do modelo não é clara e não reflete de forma fidedigna a realidade. Lembrando que um modelo nada mais é do que uma representação simplificada da realidade. Neste caso, parece que o nosso modelo está muito simplista e não reflete os aspectos da realidade de forma adequada. Porém, dadas as funções que estudamos no Capítulo 1 podemos construir um modelo mais realista.

4.1.3
Melhorando o modelo

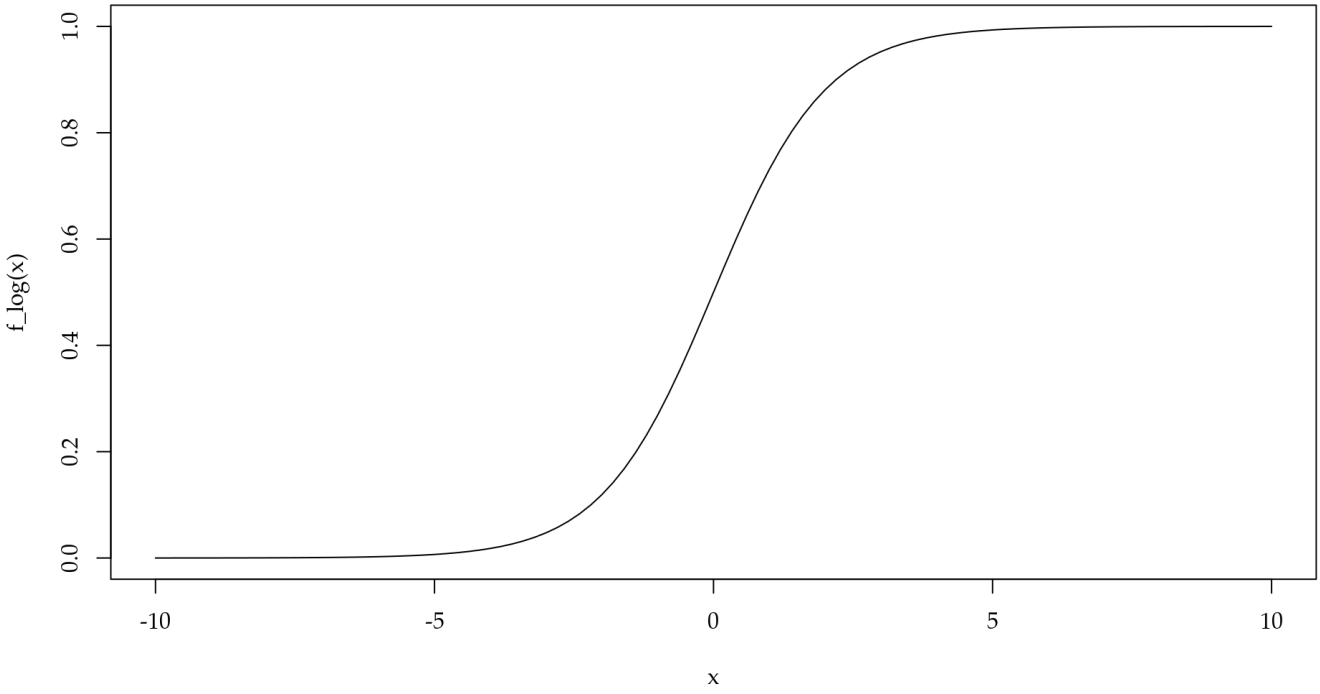
Uma vez que nosso modelo não parece uma aproximação razoável para a realidade, podemos tentar melhorá-lo fazendo com que aspectos práticos sejam levados em consideração. Note que se a previsão fosse restrita ao intervalo unitário poderíamos interpretar a previsão como uma medida de pertinência aos grupos, o que seria mais intuitivo. O que gostaríamos é que grandes valores de $\hat{\beta}_0 + \hat{\beta}_1 renda_i + \hat{\beta}_2 anos_i$ resultassem em valores próximos a 1. Da mesma forma, pequenos valores de $\hat{\beta}_0 + \hat{\beta}_1 renda_i + \hat{\beta}_2 anos_i$ devem resultar em valores próximos a 0. Para incluir em nosso modelo essa intuição podemos usar uma função matemática que tenha essa característica. O que precisamos é

uma função que receba um número real qualquer e resulte em um número no intervalo unitário. Em outros termos, precisamos de uma função que tenha domínio sendo os números reais e tenha como imagem o intervalo unitário, ou seja, $f(\cdot) : \Re \to (0, 1)$.

Existem na literatura diversas funções que tem tal propriedade. Uma escolha popular é a função logística dada por:

$$f(x) = \frac{1}{1 + e^{-x}},$$

cujo gráfico é apresentado abaixo.



Com a função logística em mãos podemos reescrever nossa função objetivo como

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

Novamente, podemos facilmente otimizar essa função objetivo, ou seja, encontrar β_0, β_1 e β_2 que tornam SQ_{logit} o menor possível. Novamente escrevemos a função objetivo

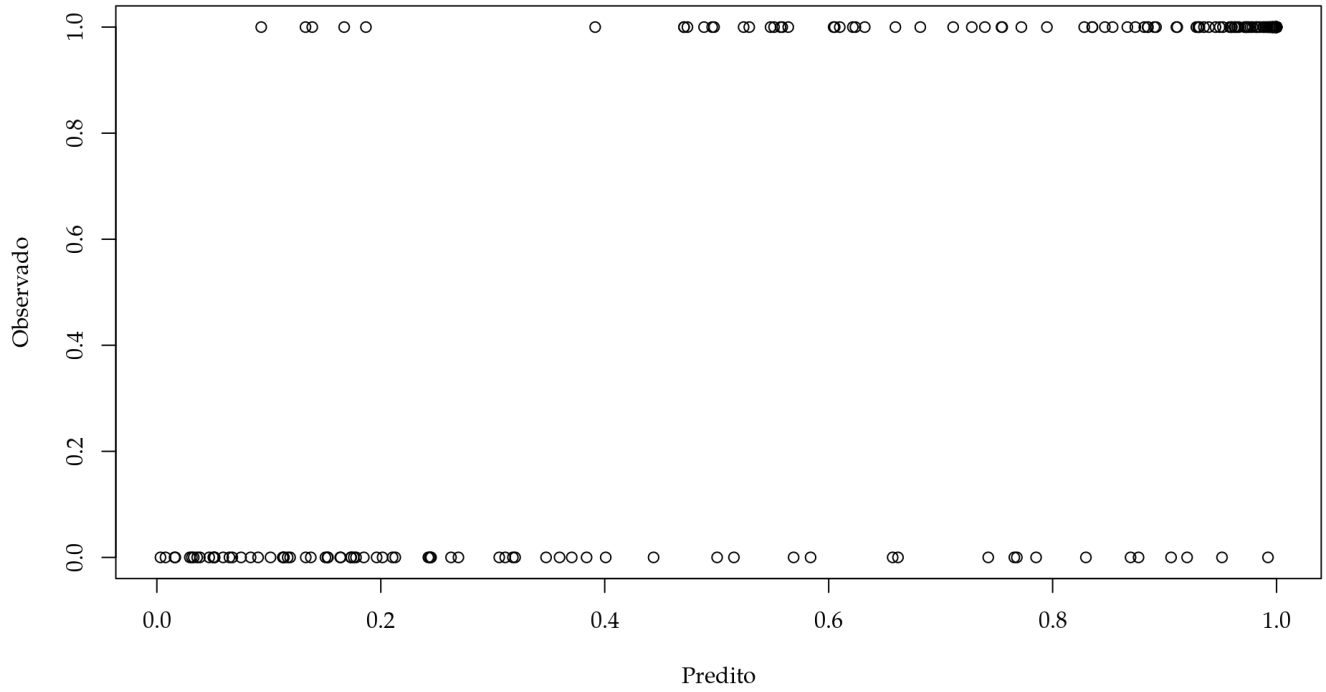
```
f_logit <- function(par, y, renda, anos) {
  mu <- 1/(1+ exp(- (par[1] + par[2]*renda + par[3]*anos)))
  SQ_logit <- sum( (y - mu)^2 )
  return(SQ_logit)
}
```

Otimizamos numericamente.

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

```
## $par
## [1] -6.642  0.129  0.539
##
## $value
## [1] 21.4
##
## $counts
## function gradient
##      220       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Plotando os preditos contra os observados.



Neste caso fica claro que nosso modelo só prediz valores no intervalo unitário como gostaríamos. Claramente quanto maior é o preditor maior é a chance dos usuários serem do grupo 1 como queríamos.

4.1.4

Melhorando a função objetivo

Apesar de melhor que a estratégia anterior, essa abordagem ainda faz uso da perda quadrática. Isso significa que assumimos que nossos erros são simétricos ao redor da curva ajustada, neste caso a logística. Entretanto, tal suposição não reflete o fato de que nas caudas da curva os erros não podem ser simétricos. Por exemplo, no extremo da cauda esquerda (baixa renda e pouca experiência) só podemos errar para mais, enquanto que no extremo da cauda direita só podemos errar para menos. Assim, seria interessante termos uma função objetivo que melhor represente nossa intuição.

Para construir uma função objetivo que atenda a nossa intuição, podemos recorrer a uma distribuição de probabilidade adequada para dados binários. A distribuição Bernoulli modela variáveis aleatórias que podem assumir apenas dois valores: 0 ou 1, sendo 1 chamado de sucesso e 0 de fracasso. A distribuição Bernoulli, tem a seguinte função de probabilidade

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

onde μ é a probabilidade de sucesso, ou seja, $\mathrm{P}(Y = 1) = \mu$.

Para um conjunto de n observações, cada uma sendo binária, podemos usar o produto das funções de probabilidade como uma função objetivo a ser maximizada neste caso. Note que a função de probabilidade fornece a probabilidade da variável aleatória assumir um certo valor dado um valor para μ. Uma vez que temos os dados observados y_i, a única quantidade desconhecida é o valor de μ. E a nossa função objetivo fornece a probabilidade de observar os dados realmente observados y_i caso o valor do μ seja um valor fixado. O objetivo é encontrar $\hat{\mu}$ que maximize essa probabilidade, ou seja, de observar o que você realmente observou. Essa é a ideia por trás de um dos métodos mais populares e poderosos de estimação chamado método de máxima verossimilhança.

Como μ é a probabilidade de sucesso, ou seja, o usuário ser assinante *premium*, podemos descrever tal probabilidade de acordo com as características dos usuários, neste caso renda e anos de experiência. Uma vez que μ é uma probabilidade só pode assumir valores no intervalo unitário, assim usando a função logística podemos descrever/modelar μ pela seguinte equação:

$$\mu_i = \frac{1}{1 + e^{-(\beta_0 + \beta_1 renda_i + \beta_2 anos_i)}}.$$

Substituindo na equação acima chegamos a seguinte função objetivo

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

Entretanto, em termos computacionais usar o produto de números entre 0 e 1 é inconveniente, porque conforme o número de termos sendo multiplicado cresce a função objetivo vai rapidamente para zero. Assim, é mais conveniente computacionalmente usar o logaritmo do produto das funções de probabilidade o que resulta na seguinte função objetivo:

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

Essa função objetivo reflete o fato dos erros não serem simétricos. Para finalizar, vamos implementar essa nova função objetivo e otimizá-la numéricamente. A função em R fica dada por

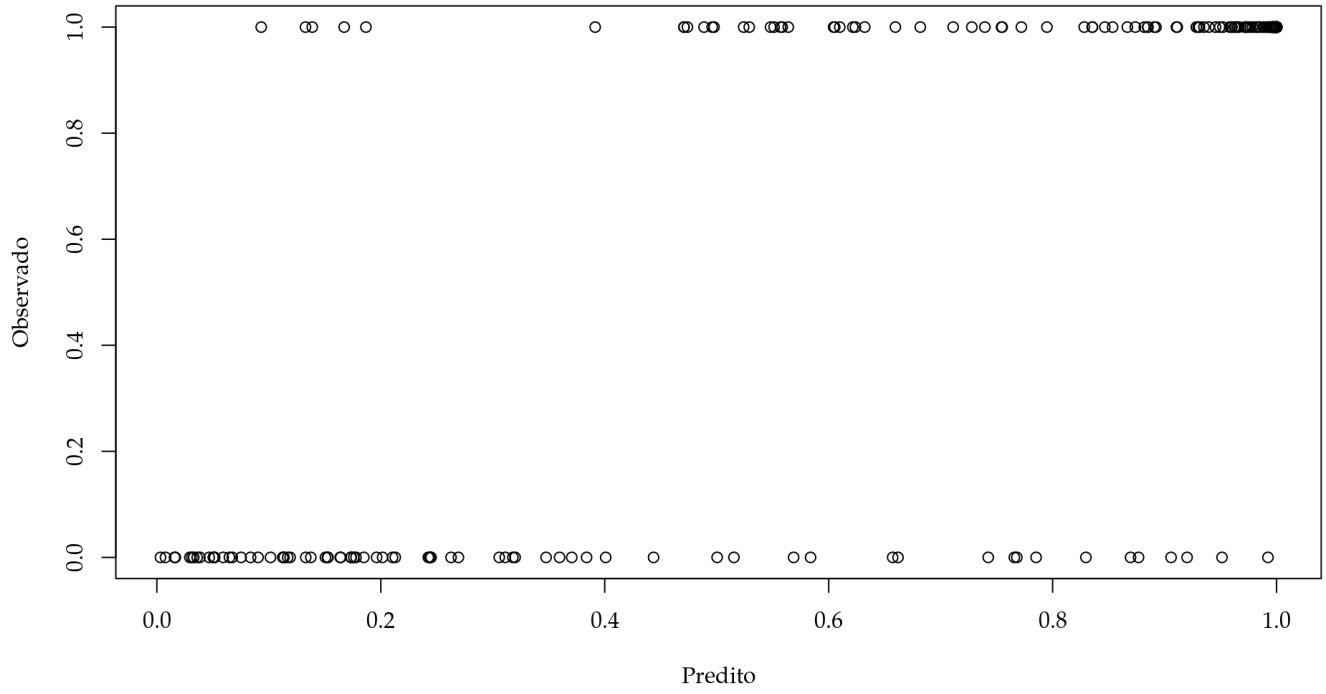
```
f_ber <- function(par, y, renda, anos) {
  eta = par[1] + par[2]*renda + par[3]*anos
  mu <- 1/(1+exp(-eta))
  out <- sum(y*log(mu) + (1-y)*log(1-mu))
  return(out)
}
```

Fazendo a otimização numericamente, temos

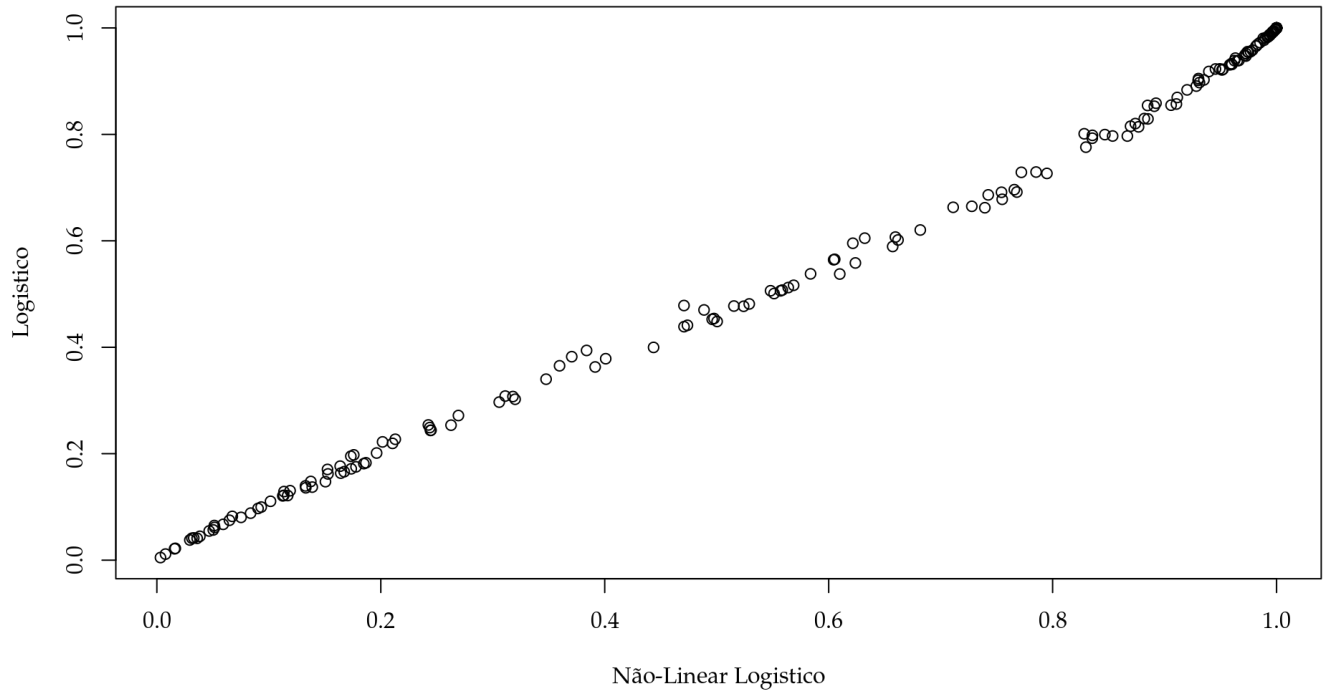
```
fit_ber <- optim(par = c(0,0,0), fn = f_ber, y = dados$Premium,
                 renda = dados$Renda, anos = dados$Anos,
                 control = list(fnscale = -1))
fit_ber
```

```
## $par
## [1] -6.119  0.112  0.504
##
## $value
## [1] -68.1
##
## $counts
## function gradient
##      166       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Note que neste caso temos que maximizar a função objetivo, por isso usamos o argumento adicional da função `optim()` `fnscale = -1` para tornar o problema de minimização, uma vez, que por *default* a função `optim()` minimiza a função objetivo. Podemos novamente calcular os preditos usando esse novo modelo.



Caso fosse de interesse, podemos até comparar os preditos obtidos por cada uma das abordagens. Por exemplo,



Neste caso, vemos uma grande similaridade entre os dois classificadores. Para classificar, podemos definir um limiar por exemplo, se $\hat{\mu} < l1$ dizemos que o usuário não é *premium* e caso contrário, ou seja, se $\hat{\mu} > l1$ o usuário é *premium*.

É importante notar que nós fomos passo-a-passo tornando nosso modelo mais condizente com a realidade e consequentemente atendendo às nossas intuições de como a realidade deve ser. O uso da distribuição Bernoulli para construir a função objetivo combinada com a função logística é o que se denomina de modelo de regressão logística na literatura estatística.

4.1.5
Melhorando o algoritmo de ajuste

Uma vez que o modelo foi definido e a função objetivo escolhida, o processo de treinamento ou ajuste do modelo consiste em um problema de otimização numérica. Até o momento, nós simplesmente usamos otimizadores genéricos já implementados na função `optim()` do software `R`.

Como já discutimos, a otimização de uma função pode ser feita de diversas formas. Nos exemplos anteriores nós usamos o algoritmo de Nelder-Mead que não depende de derivadas. O algoritmo de Newton é o mais eficiente em termos de número de iterações para atingir a convergência. Sendo assim, vamos agora discutir como implementar o algoritmo de Newton para o modelo de regressão logística. Lembre-se que a função a ser otimizada é

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

O algoritmo de Newton trabalha com a seguinte equação de iteração

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

onde $\mathbf{J}(\boldsymbol{\beta}^{(i)})$ é uma matriz hessiana de dimensão 3×3 e $l'(\boldsymbol{\beta}^{(i)})$ é o vetor gradiente de dimensão 3×1. Precisamos de um vetor inicial $\boldsymbol{\beta}^{(1)}$ para o vetor de parâmetros 3×1 no caso do nosso exemplo. O vetor gradiente é dado por

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

Para obter cada uma das entradas do vetor gradiente usamos as regras básicas de derivação associadas a regra da cadeia. Para simplificar a notação, seja $\eta_i = \beta_0 + \beta_1 renda_i + \beta_2 Anos_i$. Assim, tem-se

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

Obtendo cada uma das entradas, tem-se

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

Derivando o μ em relação ao η, 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 η em relação aos β's temos

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

Finalmente, temos o vetor gradiente

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

Usando cálculos similares, podemos obter a matriz hessiana. Lembre-se, não deixe as derivadas te assustar, use um software de matemática simbólica caso não se sinta confortável com os resultados.

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

Uma vez obtidos o gradiente e o hessiano, podemos implementar ambas funções em `R`.

```r
# 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)
}
```

O método de Newton foi discutido e implementado, assim podemos usar a nossa função.

```
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)
}
```

Otimizando a função, tem-se

```
##        [,1]   [,2]  [,3]
##  [1,]  0.00 0.0000 0.000
##  [2,] -2.00 0.0318 0.194
##  [3,] -3.62 0.0638 0.302
##  [4,] -5.11 0.0932 0.418
##  [5,] -5.94 0.1092 0.488
##  [6,] -6.12 0.1124 0.504
##  [7,] -6.12 0.1125 0.504
##  [8,] -6.12 0.1125 0.504
##  [9,]    NA     NA    NA
## [10,]    NA     NA    NA
```

Pelo traço do algoritmo é fácil verificar a convergência. Como exercício, tente implementar o mesmo modelo usando o algoritmo gradiente descendente e o método quasi-Newton BFGS.

Importante notar que o mesmo modelo pode ser ajustado ou treinado de diversas formas. Neste exemplo, usamos o método de Nelder-Mead via a função `optim()` o que foi simples e rápido. No caso do método de Newton foi necessário maior entendimento de cálculo para obter o vetor gradiente e a matriz Hessiana. De forma geral, para um exemplo simples com apenas duas variáveis explanatórias, a diferença em termos de tempo computacional é muito pequena. Porém, para situações mais complexas pode ser a diferença entre obter convergência ou não. Em `R` a função `glm()` implementa o modelo de regressão logística

usando um outro algoritmo similar ao método de Newton chamado de *Fisher scoring*. A principal diferença é que a matriz hessiana é substituída pela sua esperança. Esse é um algoritmo com um componente estatístico (a esperança) e está fora do escopo deste livro.

4.2

Clusterização usando *kmeans*

A tarefa de agrupar indivíduos/unidades semelhantes é muito comum e importante em ciência de dados. Exemplos comuns aparecem na segmentação de clientes para aplicar diferentes estratégias de *marketing*. Classificar gêneros musicais para melhorar a experiência dos usuários em plataformas especializadas, entre diversas outras. Técnicas de agrupamento fazem parte dos chamados métodos de aprendizagem não-supervisionada na literatura de aprendizagem de máquina.

Dentre as diversas estratégias para agrupamento, o método conhecido como *kmeans* é muito popular. A ideia do método é dado um vetor de características de um indivíduo criar grupos onde indivíduos dentro do mesmo grupo são mais parecidos do que indivíduos em grupos diferentes. Para fazer esse agrupamento o método *kmeans* usa como medida resumo de cada grupo a média amostral e o objetivo é fazer com que cada indivíduo pertença ao grupo com a média mais próxima à ele. A média de cada grupo corresponde ao centróide de cada grupo e ilustra o comportamento usual de determinado grupo.

Para ilustrar a ideia considere os dados da Figura 4.1 onde temos duas variáveis contínuas x_1 e x_2 observadas em um conjunto de 300 indivíduos. O objetivo do método *kmeans* é agrupar os 300 indivíduos, em digamos k grupos de forma que a distância de cada indivíduo ao centróide de seu grupo seja a menor possível.

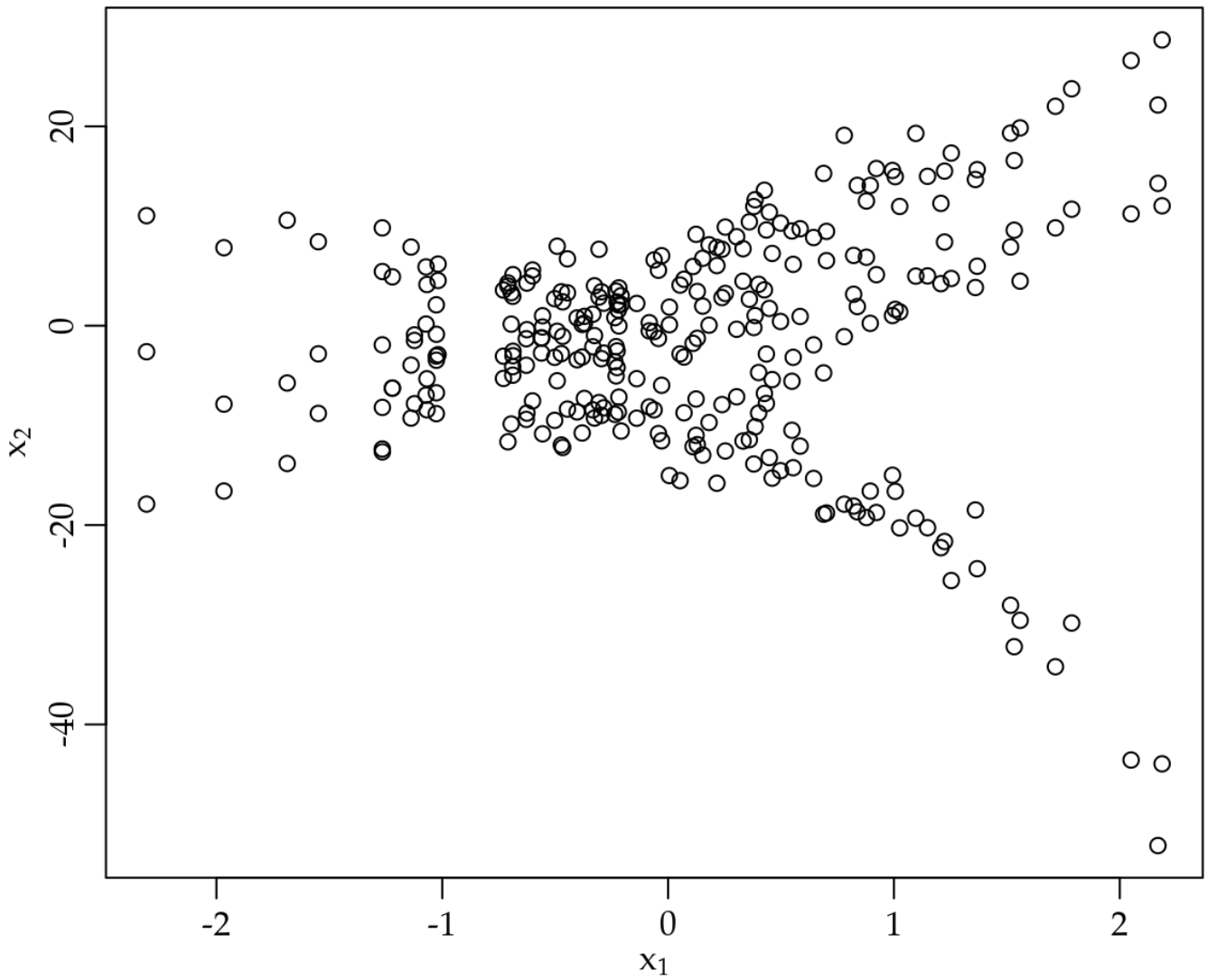


Figura 4.1: Ilustração de conjunto de dados para uso do método kmeans.

De forma mais geral, considere um conjunto de observações $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$, sendo cada observação um vetor p-dimensional de valores reais e n o tamanho do conjunto de dados. O objetivo do método de agrupamento *kmeans* é particionar as n observações em $k < n$ grupos, digamos $G = \{G1, \ldots, G_k\}$ de tal forma a minimizar a soma de quadrados dentro de cada grupo, ou de forma equivalente, maximizar a heterogeneidade entre grupos. Em termos de notação matemática podemos escrever esse objetivo da seguinte forma:

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

Note que esse não é um problema de otimização usual como aqueles apresentados no Capítulo 3. A principal diferença é que nós desejamos otimizar uma função que depende dos elementos dentro de cada grupo. Isso gera uma função que não é contínua e portanto não podemos usar os algoritmos de otimização usuais baseados em derivadas e solução de sistemas não-lineares.

Para resolver este problema o algoritmo mais comum usa a chamada técnica de refinamento. A ideia é muito simples, dado um conjunto inicial de k médias, digamos, $\mu_1^{(1)}, \mu_2^{(1)}, \ldots, \mu_k^{(1)}$ 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.

O algoritmo repete até que algum critério de parada seja atingido. Um critério comum é que as observações parem de mudar de grupo, ou equivalentemente, a composição dos grupos não mude. Em geral este algoritmo não tem convergência garantida, mas funciona bem em termos práticos. Note que a medida de distância usada é uma parte importante deste algoritmo.

O Código 4.1 apresenta uma função genérica para calcular a distância Euclidiana entre uma matriz de observações (argumento `dados`) e um ponto (argumento `media`). Note que o argumento `media` será o centróide do grupo, na implementação do método *kmeans*.

Código 4.1 *Distância Euclidiana entre uma matriz de observações e um ponto.*

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

No Código 4.1 foi utilizada a função `apply()` para percorrer todas as linhas da matriz `dados` e calcular a sua distância Euclidiana até o vetor `media`. De forma equivalente poderia ter sido utilizada uma instrução do tipo `for()` para esta tarefa.

Com a função que calcula a distância pronta, podemos facilmente implementar o método *kmeans* como ilustrado pelo Código 4.2.

Código 4.2 *Implementação didática do método kmeans.*

```r
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)
  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)
    # Escolhe a qual grupo a observação pertence
    grupos[,i] <- as.factor(apply(temp, 1, which.min))
    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))
}
```

Devemos nos atentar a alguns detalhes sobre a função no Código 4.2. A primeira decisão para a implementação é como inicializar o algoritmo. Neste exemplo, optamos por gerar n números aleatórios entre 1 e k, sendo k o número de grupos criados. Note que k é um

hiperparâmetro do modelo, uma vez que precisa ser especificado previamente para que o algoritmo possa ser executado. Para guardar todo o histórico do que acontece durante as iterações do algoritmo, foram criados três objetos `grupos`, `criterio` e `SOMA_QQ`. Estes objetos vão guardar a composição de cada grupo, o criterio vai receber o número absoluto de trocas de grupos, medida como a diferença entre a composição dos grupos no passo i menos $i-1$. Como critério para parada fixamos que a composição dos grupos pare de mudar, ou seja, o `criterio` no passo i é igual ao critério no passo $i-1$. Por fim, calculamos a soma de quadrados que é uma medida de heterogeneidade entre os grupos. Assim, esperamos que essas somas cresçam conforme o algoritmo itera, melhorando assim a homogeneidade interna dos grupos. Para calcular a media dentro de cada grupo usamos a função `aggregate()`. Para ilustrar o progresso do algoritmo imprimimos a cada iteração o valor do objeto `criterio`. O que esperamos é que esse número vá diminuindo até estabilizar, indicando a convergência do algoritmo para uma solução.

Com a nossa função implementada podemos usar o método para criar os grupos baseados nos dados apresentados na Figura 4.1. Neste caso vamos fixar o número de grupos em 3, por simplicidade. Na prática é comum usar diferentes números de grupos e comparar os resultados em termos da composição dos grupos para então decidir qual o número ideal de 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
```

Importante ressaltar que a convergência para um ótimo global não é garantida. Além disso, dada a inicialização aleatória é perfeitamente possível que o método resulte em agrupamentos diferentes para diferentes chamadas da função `my_kmeans`. Para tornar reproduzível para o leitor foi fixada a semente (argumento `set.seed`). A Figura 4.2 apresenta os grupos no passo inicial (aleatório) e no passo final do algoritmo *kmeans*.

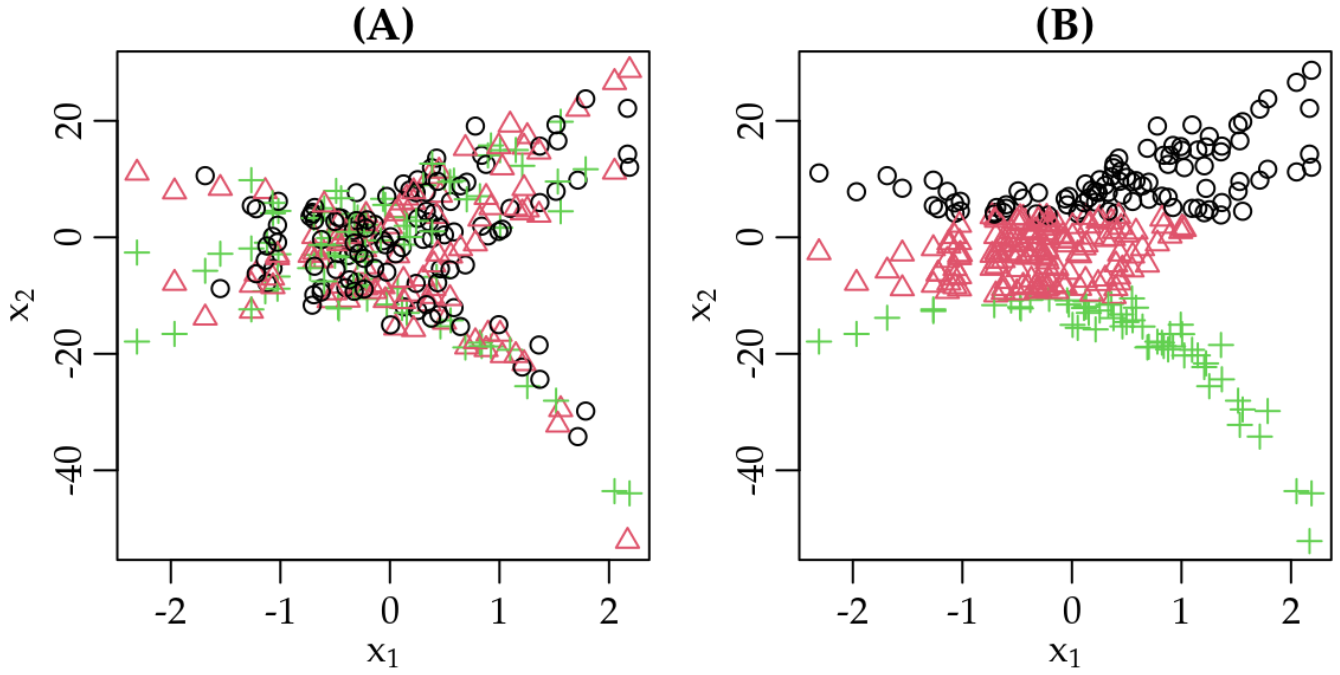


Figura 4.2: Ilustração do método kmeans, solução inicial (A) e solução final (B).

Não é o objetivo deste livro discutir os aspectos práticos da aplicação do método *kmeans*. Porém, é perceptível por meio dos gráficos da Figura 4.2 que o algoritmo foi capaz de encontrar três grupos com comportamento distintos. Em `R` a função `kmeans()` do pacote `stats` implementa diferentes algoritmos para realizar agrupamentos usando o método *kmeans*.

Outro aspecto importante é que agora que você entende a lógica matemática por trás de um algoritmo de agrupamento, você pode facilmente propor novas técnicas trocando alguns de seus componentes. Por exemplo, trocar a medida de distância ou trocar a medida resumo de cada grupo. Alguns métodos derivados com essas ideias são o $kmedians$ e o $kmenoids$. O primeiro troca a medida resumo de cada grupo da média para a mediana. O segundo estabelece que o centróide do grupo deve ser um indivíduo observada (*menoid*). O importante é que conhecendo um algoritmo em detalhes você está mais perto de entender diversos outros que são apenas modificações da ideia inicial.

4.3

Redes neurais artificiais

Redes neurais artificiais configuram uma das principais técnicas em uso por cientistas de dados. Além disso, são a principal força de trabalho dentro de uma grande área de pesquisa chamada genericamente de **Inteligência Artificial**. Dado o enorme desenvolvimento de tais técnicas nos últimos anos, hoje temos uma infinidade de construções de redes neurais para

desempenhar os mais diversos tipos de tarefas como, por exemplo, a construção de *chatbots*, carros autônomos, assistentes virtuais, detecção de objetos em imagens, reconhecimento facial e análise da linguagem natural.

Como a maioria da técnicas criadas para extrair informações de dados as redes neurais precisam de três ingredientes:

1. Dados de entrada ou *input*: o que será o *input* para uma rede neural vai depender da tarefa a ser realizada. Por exemplo, para reconhecimento de discurso os dados de entrada podem ser arquivos de som de uma pessoa falando. Se a tarefa é reconhecer objetos em uma imagem, a entrada pode ser um conjunto de fotos ou mesmo frames de vídeo. Para tarefas mais convencionais como, por exemplo, predizer o preço de um certo imóvel, os dados de entrada podem ser características do imóvel, de forma análoga ao modelo de regressão linear múltipla discutido na seção (RLM).

2. Exemplos da saída esperada ou *output*: no caso de reconhecimento de discurso a saída pode ser um conjunto de textos traduzidos por humanos a partir de arquivos de som. Imagens classificadas por humanos como tendo ou não um certo objeto. E novamente, no caso da precificação de imóveis um conjunto de preços arbitrado por alguém *expert* da área.

3. Uma forma de medir se o algoritmo é capaz de reproduzir os exemplos de saída: a ideia é de alguma forma medir a distância entre o *output* e o que o algoritmo prediz para o *output*. Isso nada mais é do que definir uma função perda que mede o quanto estamos perdendo ao usar o modelo ao invés dos dados. Nós já discutimos essa ideia, em especial na seção (RD) quando deduzimos a média amostral como um resumo de um único número de um conjunto de dados.

Existem diversos tipos de redes neurais, porém neste livro vamos discutir apenas uma tipo de redes neurais chamada de *Multilayer Perceptron*. Uma forma de pensar neste tipo de rede neural é como um modelo de regressão, porém a forma da relação entre as covariáveis (*inputs*) e a variável resposta (*output*) é dada por um conjunto de função não-lineares encadeadas em um formato de rede, conforme ilustrado na Figura 4.3.

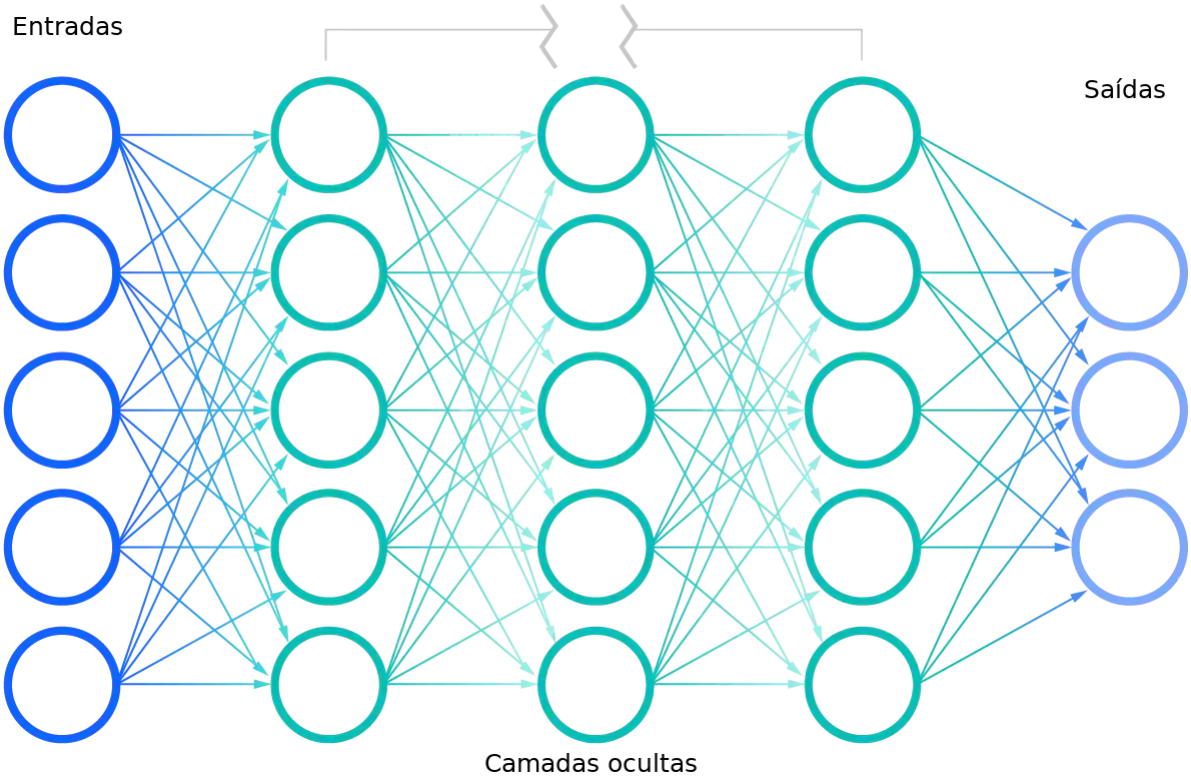


Figura 4.3: Desenho esquemático da estrutura de uma rede neural.

Na Figura 4.3 a primeira coluna indica as variáveis de entrada. As camadas em verde são chamadas de camadas ocultas, no sentido que serão geradas com base na camada de entrada por meio de combinações lineares das variáveis de entrada. Cada círculo das camadas ocultas são chamados de neurônios. Dentro de cada neurônio uma função não-linear (função de ativação) é aplicada a uma combinação linear das entradas de modo a resultar em um número. Na sequência esse número vira uma entrada para o próximo neurônio e assim por diante até chegar a camada de saída. A camada de saída corresponde ao valor que o nosso modelo prediz para a variável resposta. Este processo de percorrer as variáveis de entrada pela rede é chamado de *feedforward*.

Para materializar essa ideia considere um conjunto de dados com duas variáveis de entrada x_1 e x_2 e uma variável de saída (contínua). Por simplicidade, vamos considerar apenas uma camada oculta e dois neurônios. Neste caso a rede neural tem a estrutura apresentada na Figura 4.4.

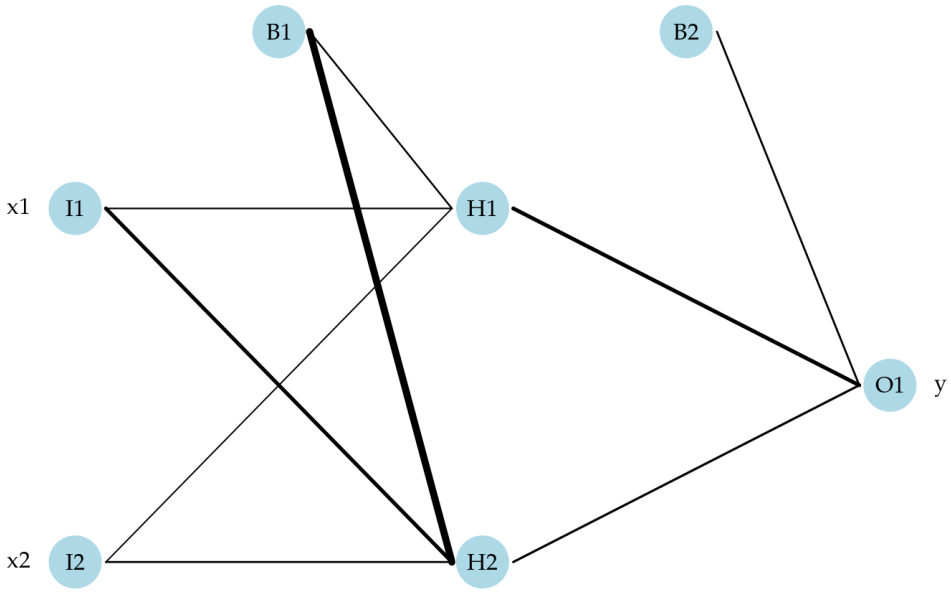


Figura 4.4: Desenho esquemático da estrutura de uma rede neural com duas variáveis de entrada e uma camada oculta.

Fica claro que as variáveis de entrada x_1 e x_2 ativam os dois neurônios da camada oculta gerando novos valores h_1 e h_2. Adicionalmente, em cada camada é adicionado um *bias* denotado por b_1 e b_2. Esse termo é análogo ao intercepto do modelo de regressão linear múltipla. Como mencionado dentro de cada neurônio as entradas passam por uma função de ativação que precisa ser escolhida. Para este exemplo, vamos usar a função logística já apresentada na seção 1.1.2.

Vamos agora ver em detalhes os passos da etapa *feedforward*. Para simplificar a discussão, considere que a base de dados tem apenas um indivíduo. Assim, o vetor de entrada é dado por $\mathbf{x} = (x_1, x_2)^\top$. O primeiro passo é gerar uma combinação linear entre essas duas variáveis de entrada. Para isso selecionamos dois pesos w_{11} e w_{21} para representar o efeito das entradas x_1 e x_2 no neurônio 1 e denotamos por w_{01} o *bias* associado ao primeiro neurônio. De forma, similar denotamos por w_{02} o *bias* associado ao segundo neurônio e w_{12} e w_{22} pesos representando os efeitos das entradas x_1 e x_2 no neurônio 2. Assim, temos

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

O próximo passo é passar os valores η_1 e η_2 pela função de ativação logística para gerar os valores h_1 e h_2. Neste caso, temos

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

Por fim, na última camada também conhecida como camada de saída, combinamos h_1 e h_2 com o termo de *bias* para gerar o valor predito pela rede. Para este passo também podemos usar alguma função não-linear, porém isso vai depender da natureza da variável de saída. Neste exemplo, como temos uma variável contínua o usual é usar como função de saída a função identidade. Portanto, temos

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

em que w_{0O} denota o *bias* da camada de saída. De forma similar w_{1O} e w_{2O} os pesos associados aos efeitos ocultos h_1 e h_2. Vamos ilustrar esse passo-a-passo em `R`. Para isso considere o conjunto de dados `NN.txt` disponível na página do livro. Esse conjunto de dados foi simulado apenas para ilustrar a implementação computacional de uma rede neural.

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

```
##        x1     x2    y
## 1 -0.5605 -0.710 21.8
## 2 -0.2302  0.257 29.9
## 3  1.5587 -0.247 31.1
## 4  0.0705 -0.348 33.9
## 5  0.1293 -0.952 25.4
## 6  1.7151 -0.045 26.2
```

Vamos selecionar apenas o primeiro indivíduo e proceder o passo-a-passo da etapa de *feedforward*. Para poder executar os cálculos vamos especificar valores numéricos para os pesos em todas as camadas do modelo. Para este exemplo vamos usar como pesos alguns valores pré-selecionados para resultar em um ajuste razoável para os dados.

```r
# 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
w0O <- 9.21; w1O <- 9.41; w2O <- 11.65
y_hat <- w0O + w1O*h1 + w2O*h2

# Comparando com o dado observado
c(y, y_hat)
```

```
## [[1]]
## [1] 21.8
##
## $x1
## [1] 22.1
```

Note que o valor de saída da rede neural foi o valor 22.1, enquanto que o valor observado para este indivíduo foi de 21.8. Essa diferença é resultado dos valores pré-especificados para os pesos da rede.

O próximo passo é criar uma forma de medir o quanto o modelo está diferente das observações. Isso é atingido por especificar uma função perda, de forma análoga com o que fizemos para obter a média amostral e os coeficientes de regressão na seção 2.2.7.

Novamente, a função perda mais popular é a perda quadrática. Denote por $\hat{y}_i(\mathbf{w})$ a saída da rede neural (passo *feedforward*) para o indivíduo i e note que a notação $\hat{y}_i(\mathbf{w})$ enfatiza que os valores de saída da rede neural são funções dos pesos $\mathbf{w} = (w_{01}, w_{11}, w_{12}, w_{02}, w_{21}, w_{22}, w_{0O}, w_{1O}, w_{2O})^\top$. Denote por y_i o valor observado para o indivíduo i. Nestas condições a função perda quadrática fica dada por

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

sendo obviamente uma função dos pesos $\mathbf{w}$. Note que os pesos foram omitidos da notação no termo $\hat{y}_i$ para facilitar a discussão.

Nossa tarefa agora consiste em treinar o nosso modelo, no sentido de ajustar os pesos para que a rede neural forneça valores próximos aos valores observados. Para esta tarefa podemos usar os métodos de otimização apresentados na seção 3.4.3. Porém, a forma mais popular de treinar redes neurais é por meio do algoritmo gradiente descendente. Lembre-se que neste caso precisamos obter o gradiente da função $\text{SQ}(\mathbf{w})$ que consiste em obter as derivadas parciais com relação a cada um dos pesos da rede. Neste caso, temos nove pesos para serem treinados.

Para implementar este processo de treinamento precisamos implementar a função de ativação e suas derivadas. Tal função é apresentada no Código 4.3.

Código 4.3 *Função de ativação logística e suas derivadas parciais.*

```
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)
}
```

Note que temos como argumentos os pesos e uma matriz `X` com uma coluna de 1's concatenada com as variáveis de entrada. Usando a função `act_fc()` podemos agora implementar o passo *feedforward* de forma genérica para um dado conjunto de variáveis de entrada e pesos, conforme apresentado na função 4.4.

Código 4.4 *Implementação computacional do passo feedforward para uma rede neural com uma camada oculta.*

```
neural_net <- function(w, X, n_neuronio) {
  n_weight <- dim(X)[2] ## Qts pesos em cada neurônio
  n_param <- length(w) ## Qtd total de parametros
  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)
}
```

Vamos ilustrar o uso da função `neural_net()` para o conjunto de dados simulados.

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

```
##   (Intercept)      x1     x2
## 1           1 -0.5605 -0.710
## 2           1 -0.2302  0.257
## 3           1  1.5587 -0.247
## 4           1  0.0705 -0.348
## 5           1  0.1293 -0.952
## 6           1  1.7151 -0.045
```

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

```
##   [,1]
## 1 22.1
## 2 29.4
## 3 30.2
## 4 29.0
## 5 27.5
## 6 30.2
```

```
dim(y_hat)
```

```
## [1] 100   1
```

A saída da função `neural_net()` é o valor predito pela rede para cada uma das 100 observações do nosso conjunto de dados. Com isso podemos agora implementar a função perda quadrática, conforme apresentado no Código 4.5.

Código 4.5 *Função perda quadrática no contexto de uma rede neural.*

```
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)
}
```

Importante ressaltar que a multiplicação por 0.5 é apenas para facilitar a posterior obtenção do gradiente e manter compatibilidade com a função `neuralnet()` do pacote `neuralnet`. Este pacote implementa tais modelos em `R` e será usado posteriormente para comparação com a nossa implementação didática. Vamos avaliar a função perda para dois conjuntos de pesos gerados aleatoriamente.

```
SQ(w = rnorm(9), X = X, Y = dados$y, n_neuronio = 2) # Pesos gerados al
```

```
## [1] 36992
```

```
SQ(w = rnorm(9), X = X, Y = dados$y, n_neuronio = 2)
```

```
## [1] 38082
```

A função `SQ()` retorna a soma de quadrados dos resíduos dividido por dois, para cada conjunto de pesos. Assim, o próximo passo é otimizar ou treinar os pesos de modo que tal soma de quadrados seja a menor possível. Para começar vamos usar os métodos de otimização não-linear vistos na seção 3.4.3 e implementados na função `optim()` .

```
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)
```

Vamos comparar o valor da soma de quadrados ótima obtido por cada um destes métodos.

```
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         664                 659  651  654
```

Note que cada um dos métodos retornou um valor diferente para a função objetivo otimizada. Isso indica que cada método encontrou um ponto de mínimo local diferente e evidencia um fato conhecido que em geral em modelos de redes neurais a função objetivo apresenta múltiplos mínimos locais tornando o uso deste tipo de método difícil para redes com múltiplas camadas e muitos pesos.

É neste ponto que o uso do método gradiente descendente aparece como uma forma de apenas andar na função por um certo número de iterações para encontrar um mínimo local. Lembre-se que o gradiente aponta na direção de maior crescimento da função. Assim, andar na direção contrária do gradiente com um certo tamanho de passo α garante que estamos descendo na superfície multidimensional gerada pela função objetivo. Para usar este método precisamos obter o gradiente o que pode ser feito analiticamente, conforme implementado no Código 4.6 ou mesmo numericamente. É claro que para modelos pequenos como o do nosso exemplo obter o gradiente numericamente é mais fácil e factível, porém em modelos reais o indicado é via *automatic differentiation* o que está fora do escopo deste livro.

Código 4.6 *Gradiente da função perda quadrática para uma rede neural.*

```
gradiente <- function(w, X, Y, n_neuronio = 2) {
  n_weight <- dim(X)[2] ## Qts pesos em cada neuronio
  n_param <- length(w) ## Qtd total de parametros
  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))
  #output <- colSums(cbind(res*mu$D_mu[,1:3]*w_h[2], res*mu$D_mu[,4:6]*w_h[3], res*H))
  return(output)
}
```

O Código @ref(lem:grad_des2) apresenta uma versão expandida do método do gradiente descendente para podermos passar argumentos adicionais a função objetivo.

(#lem:grad_des2) *Método do gradiente descendente para uma função qualquer com argumentos extras.*

```
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,
              "SolucaoOtima" = solucao[i+1,], "Otimo" = obj[c(i+1)]))
}
```

Finalmente, podemos proceder com o treinamento. O próximo código deve demorar alguns minutos para finalizar. Note que o número máximo de iterações foi fixado em $1e+05$ e estamos usando um *tuning* $\alpha = 0.005$. Esse valor foi obtido via diversas tentativas até obter convergência.

```
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)
```

Uma forma de visualizar o processo de treinamento é fazer um gráfico dos valores da função objetivo contra o indice da iteração para ver o progresso do algoritmo. Na Figura 4.5(A) apresentamos todas as iterações enquanto que na Figura 4.5(B) descartamos as 10000 primeiras iterações.

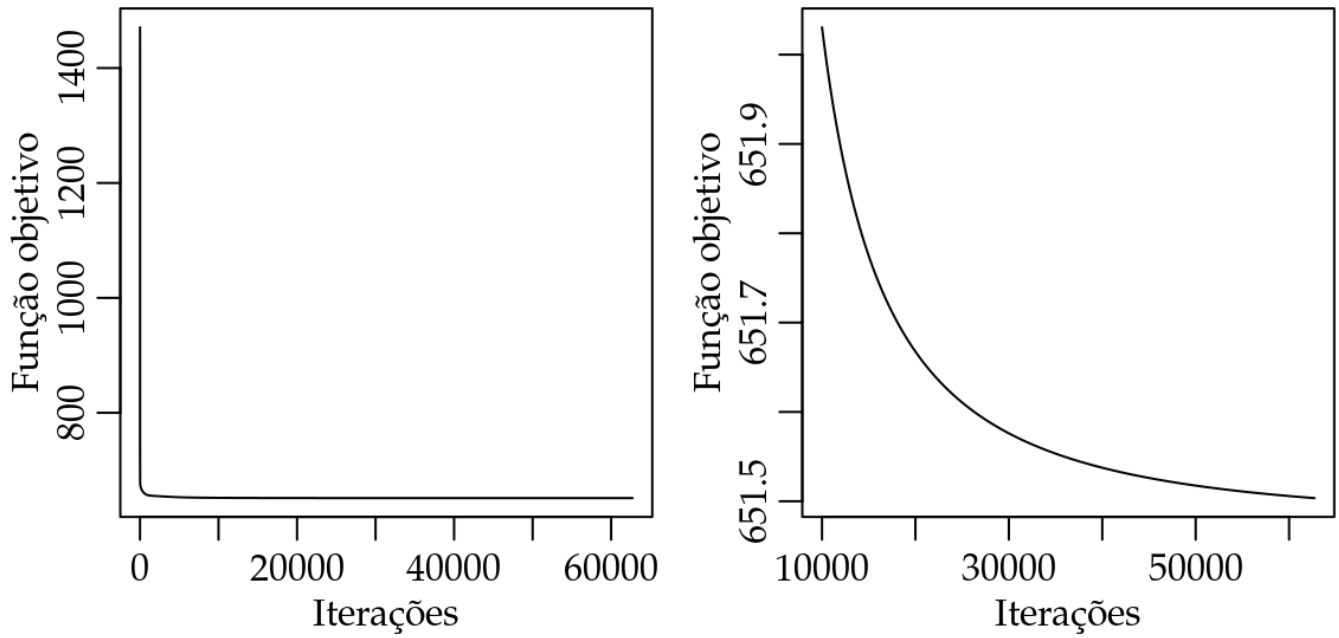


Figura 4.5: Progresso do algoritmo gradiente descendente.

A Figura 4.5 mostra claramente que a função objetivo decresce rapidamente nas primeiras iterações, porém após esse rápido decréscimo o algoritmo faz apenas pequenas modificações na função objetivo. Isso ilustra o porquê é necessário tantas iterações e a complexidade do processo de treinamento de tais modelos.

Por fim, vamos treinar a nossa rede usando a função `neuralnet()` do pacote `neuralnet` e comparar todos os resultados.

```
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]]
```

Podemos comparar o valor da função objetivo obtida por cada esquema 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 Neuralnet
## 1         664                 659  651  654     652       652
```

Neste exemplo vemos que o método BFGS apresentou a menor soma de quadrados, porém a diferença entre a nossa implementação e a da `neuralnet` é de apenas uma unidade. O algoritmo utilizado pela `neuralnet` foi o *back propagation* ou apenas *backprop* que é exatamente o que implementamos. Para finalizar, podemos comparar os valores obtidos para os pesos da rede.

```
pesos <- data.frame("Nelder-Mead" = fit_NM$par,
                    "CG" = fit_CG$par,
                    "BFGS" = fit_BFGS$par,
                    "SANN" = fit_SANN$par,
                    "GradDes" = fit_grad_des$SolucaoOtima,
                    "neuralnet" = nn_w)
pesos
```

```
##   Nelder.Mead    CG  BFGS  SANN GradDes neuralnet
## 1        8.20  6.58  6.23  6.72    6.23      6.23
## 2        6.88  6.01  5.42  5.94    5.43      5.43
## 3        6.46  5.04  4.57  5.10    4.58      4.58
## 4        4.19  7.81 30.99 13.68   27.43     27.67
## 5        2.36  3.85 15.85  7.04   14.07     14.19
## 6        1.11  1.96  7.68  3.49    6.82      6.88
## 7        3.81  3.47  7.08  6.06    7.01      7.02
## 8       10.53 12.88 14.65 13.73   14.64     14.64
## 9       15.76 13.58  8.17 10.01    8.26      8.25
```

A similaridade entre a nossa implementação e a da `neuralnet` é bastante evidente. As pequenas diferenças se devem provavelmente a diferentes critérios de parada. Para os outros algoritmos o BFGS forneceu uma solução ligeiramente melhor que o gradiente descendente e os demais Nelder-Mead, gradiente conjugado e *simulating anealling* soluções sub-ótimas.

4.4

Projeto integrador

Para terminar este livro sugerimos ao leitor o desafio de treinar um modelo ligeiramente diferente dos que foram apresentados durante o curso.

Para isto considere, o conjunto de dados `youtube.csv` que apresenta o número de *views* e inscritos de dois canais de sucesso do *youtube* desde o dia de sua abertura. O objetivo é 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 que modela o crescimento de bactérias chamado de modelo logístico, dado pela seguinte equação:

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

onde L é o valor máximo da curva (platô), β_0 é o valor de x no ponto médio da curva (tempo de meia-vida) e β é a declividade da curva.

A Figura 4.6 apresenta um gráfico do modelo logístico.

```r
par(mfrow = c(1,1), mar=c(2.6, 3, 1.2, 0.5), mgp = c(1.6, 0.6, 0))
f_log <- function(DIAS, L, beta, beta0) {
  out <- L/(1+ exp(-beta*(DIAS - beta0)))
  return(out)
}
DIAS <- 1:800
plot(f_log(DIAS = DIAS, L = 90, beta = 0.01, beta0 = 400) ~ DIAS,
     ylab = "Número de inscritos", xlab = "Dias da abertura",
     type = "l", ylim = c(0,95))
abline(h = 90)
text(x = 800, y = 93, label = "L")
text(x = 425, y = f_log(DIAS = 400, L = 90, beta = 0.01, beta0 = 400),
     label = expression(beta))
points(x = 400, pch = 19, col = "red",
       y = f_log(DIAS = 400, L = 90, beta = 0.01, beta0 = 400))
```

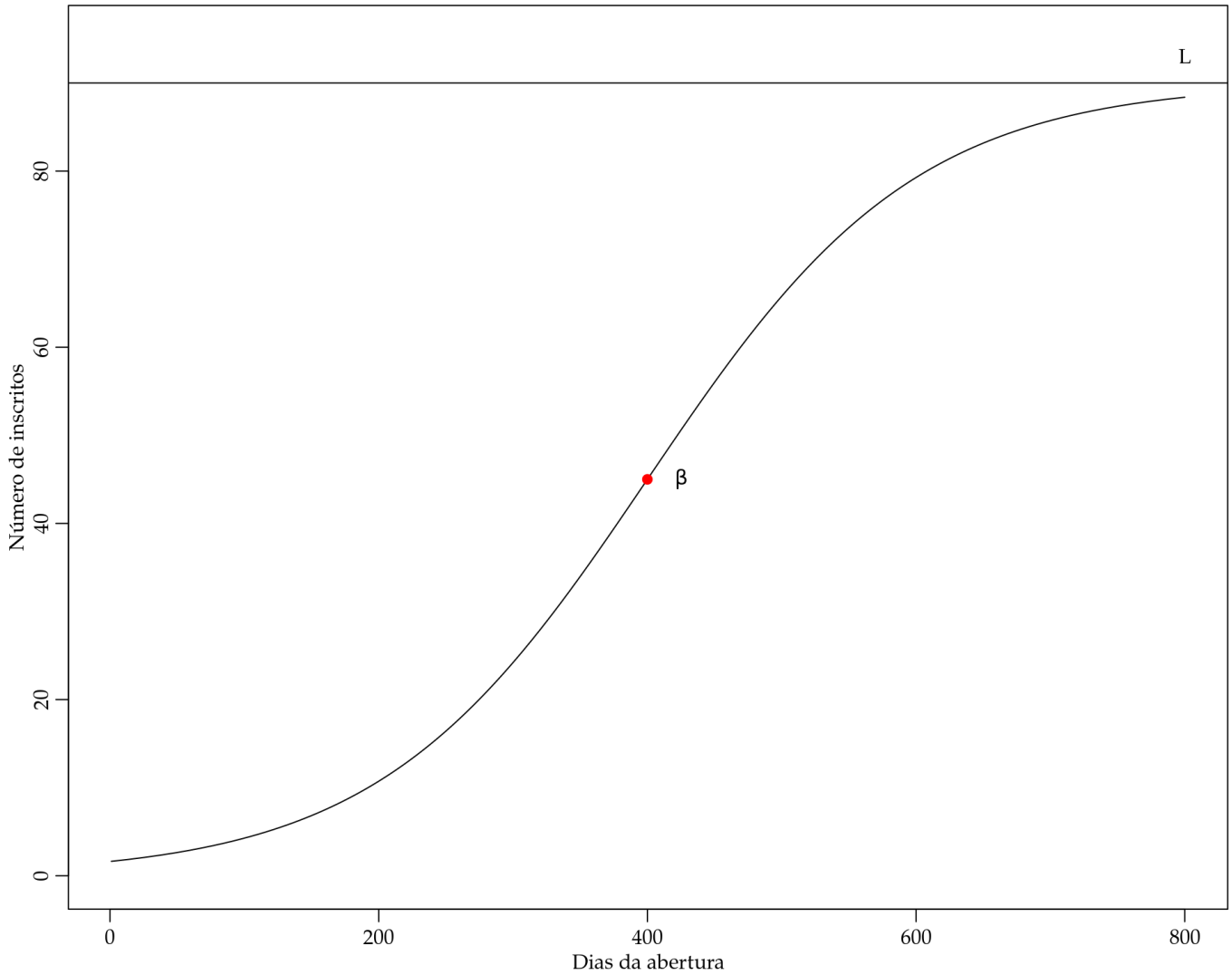


Figura 4.6: Modelo logístico.

Note que o modelo representa a intuição de como o número acumulado de inscritos em um canal deve se comportar. No Código abaixo a base de dados é carregada e organizada por canal.

```
dados <- read.table("data/youtube.txt", header = TRUE)
dados_canal <- split(dados, dados$CANAL)
dados1 <- dados_canal[[1]]
dados2 <- dados_canal[[2]]
dados1$INSCRITOS <- dados1$INSCRITOS/100000
dados1$Y <- cumsum(dados1$INSCRITOS)
dados2$INSCRITOS <- dados2$INSCRITOS/100000
dados2$Y <- cumsum(dados2$INSCRITOS)
```

Podemos fazer o gráfico dos dados observados para explicitar o objetivo de predizer o número de inscritos acumulado para os próximos 365 dias.

```
par(mfrow = c(1,2), mar=c(2.6, 3, 1.2, 0.5), mgp = c(1.6, 0.6, 0))
plot(dados1$Y ~ dados1$DIAS, xlim = c(0, 1215), ylim = c(0, 25),
     ylab = "Número de inscritos*100000", main = "Canal 1",
     xlab = "Dias", type = "o", cex = 0.1)
abline(v = 850)

plot(dados2$Y ~ dados2$DIAS, ylab = "Número de inscritos*100000", main = "Canal 2",
     xlab = "Dias", ylim = c(0, 50), xlim = c(0, 980), type = "p", cex = 0.1)
abline(v = 607)
```

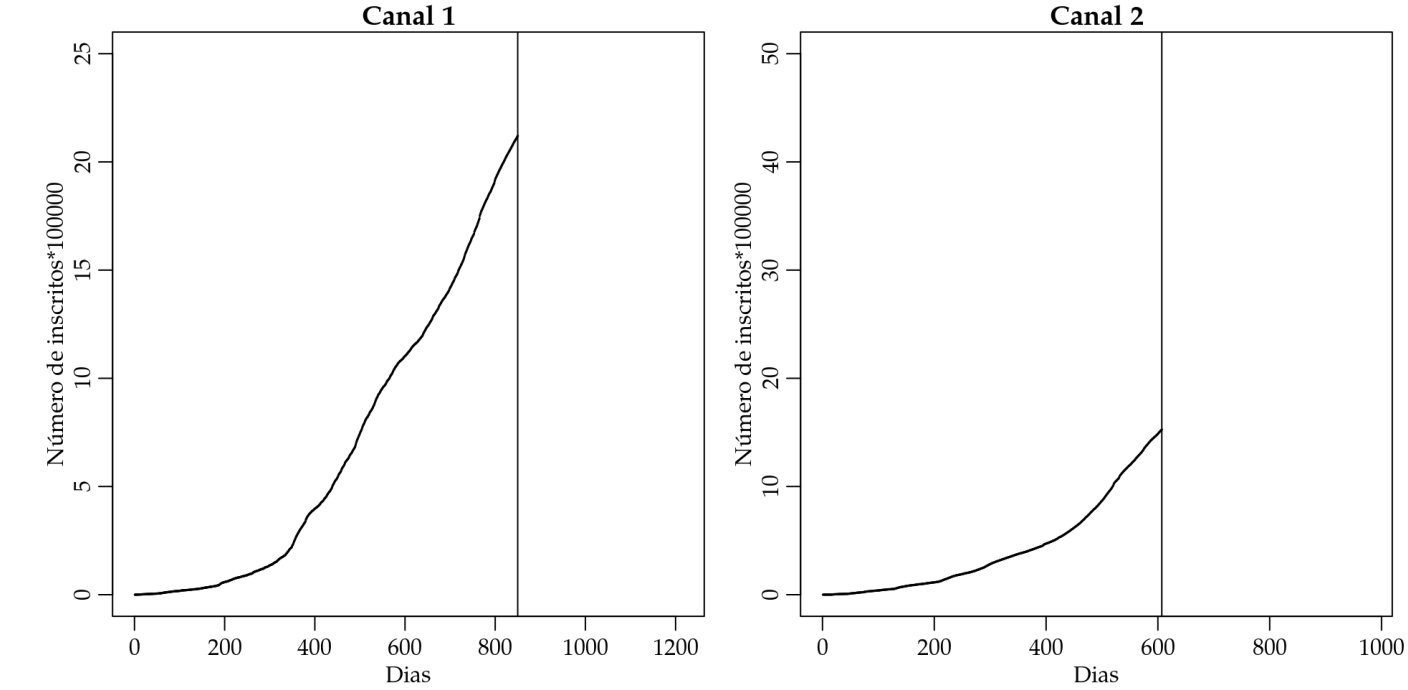


Figura 4.7: Curva do número de inscritos para dois canais do youtube.

O objetivo deste projeto é ajustar o modelo logístico para os dados de cada um destes canais e predizer qual será o número acumulado de inscritos para os próximos 365 dias. Ao apresentar sua solução computacional faça o máximo para explicar as suas decisões e estratégias de implementação. Tome o máximo de cuidado para que a sua análise seja reproduzível.

Becker, R. A., J. M. Chambers, and A. R. Wilks. 1988. *The New s Language: A Programming Environment for Data Analysis and Graphics*. Computer Science Series. Wadsworth & Brooks/Cole Advanced Books & Software.

Berkelaar, Michel, and others. 2020. https://CRAN.R-project.org/package=lpSolve.

Borchers, Hans W. 2021. *Pracma: Practical Numerical Math Functions*. https://CRAN.R-project.org/package=pracma.

Deisenroth, M. P., A. A. Faisal, and C. S. Ong. 2020. *Mathematics for Machine Learning*. Cambridge University Press.

Dowling, E. T. 1984. *Elementos de Matemática Aplicada à Economia e Administração*. Coleção Schaum. McGraw-Hill do Brasil.

Gilat, A., and V. Subramaniam. 2009. *Métodos Numéricos Para Engenheiros e Cientistas: Uma Introdução Com Aplicações Usando o MATLAB*. Grupo A - Bookman.

Goodfellow, I., Y. Bengio, and A. Courville. 2016. *Deep Learning*. The MIT Press.

Guidorizzi, H. L. 2013. *Um Curso de cálculo*. LTC.

Leithold, L. 1988. *Matemática Aplicada a Economia e Administração*. Harbra.

Molenberghs, G., and G. Verbeke. 2006. *Models for Discrete Longitudinal Data*. Springer Series in Statistics. Springer New York.

Rencher, A. C., and G. B. Schaalje. 2008. *Linear Models in Statistics*. Wiley.