Universidade de Brasília

DEPARTAMENTO DE ESTATÍSTICA

13 maio 2024

Entrega 3

Prof. Dr. George von Borries

Aluno: Bruno Gondim Toledo

Matrícula: 15/0167636

Aluno: Stefan Zurman Gonçalves

Matrícula: 19/0116994

Tópicos 2

1º/2024

6

Gradiente descendente

Kneusel traz uma visão prática do gradiente descendente. Este, seria a implementação computacional para encontrar mínimos de funções, de difícil cálculo analítico.

Trazendo um exemplo unidimensional, traz o cálculo analítico da derivada que minimiza a função. Após, traz uma implementação computacional em Python que converge para o resultado analítico. Logo após, traz a implementação para o caso bidimensional, em que o cálculo das derivadas parciais analiticamente ainda é possível, porém consideravelmente mais complicado. Já computacionalmente, o procedimento não difere muito do caso unidimensional.

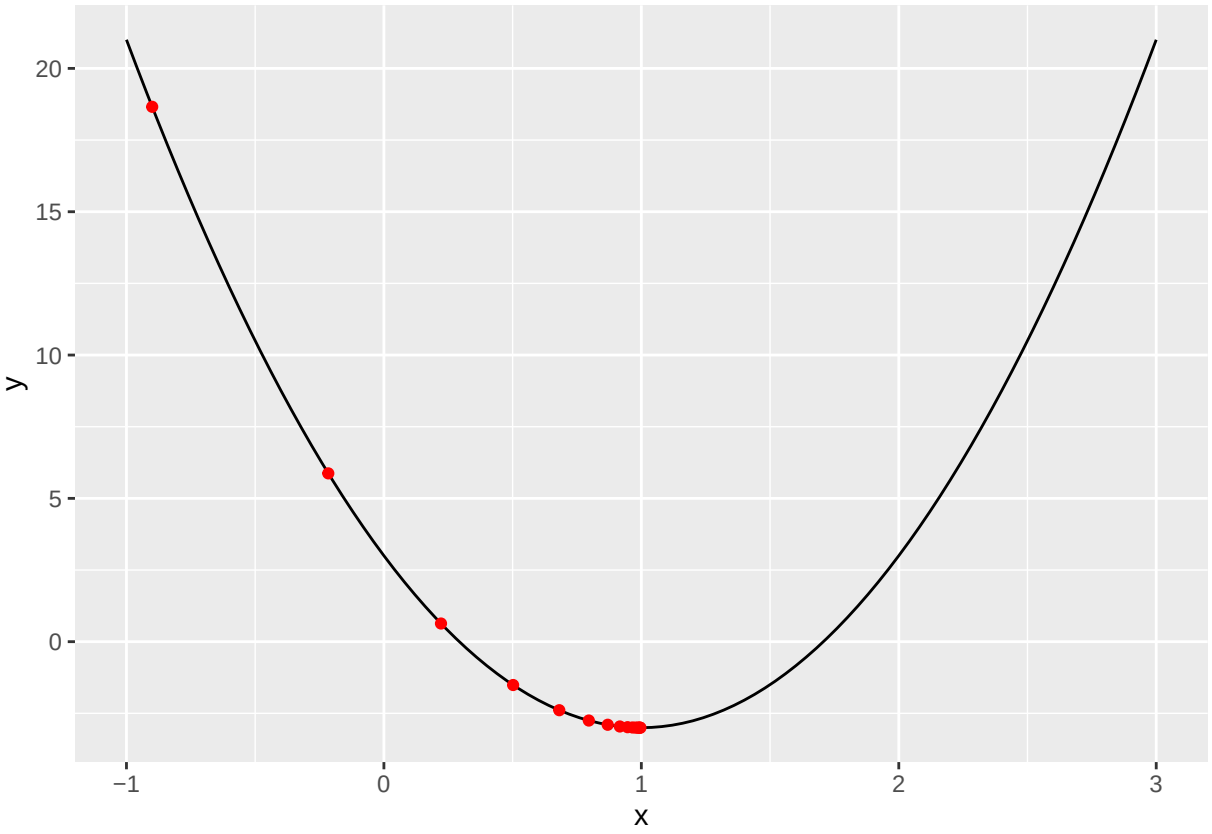
Destes, é possível expandir para qualquer situação p-dimensional, onde o cálculo analítico das derivadas é impossível, mas o computacional é trivial.

O autor argumenta que sobre a importância de definir bem o tamanho do passo η, também o chamando de 'learning rate',assim como elucida que este não necessariamente precisa ser uma constante. Argumenta ainda que, por mais que este seja importante, o formato da função será ainda mais determinante para o bem funcionamento do algoritmo de gradiente descendente.

Argumenta ainda sobre casos em que existem mínimos locais. Neste caso, o sucesso do algoritmo dependerá do ponto inicial, visto que pode ficar 'preso' em um mínimo local caso passe por um. Entretando, para casos práticos de aprendizado de máquina, existem diversos mínimos locais e parecidos entre sí, portanto, para muitos casos, encontrar um mínimo local será suficiente para endereçar o problema, não necessitando necessariamente atingir o mínimo global.

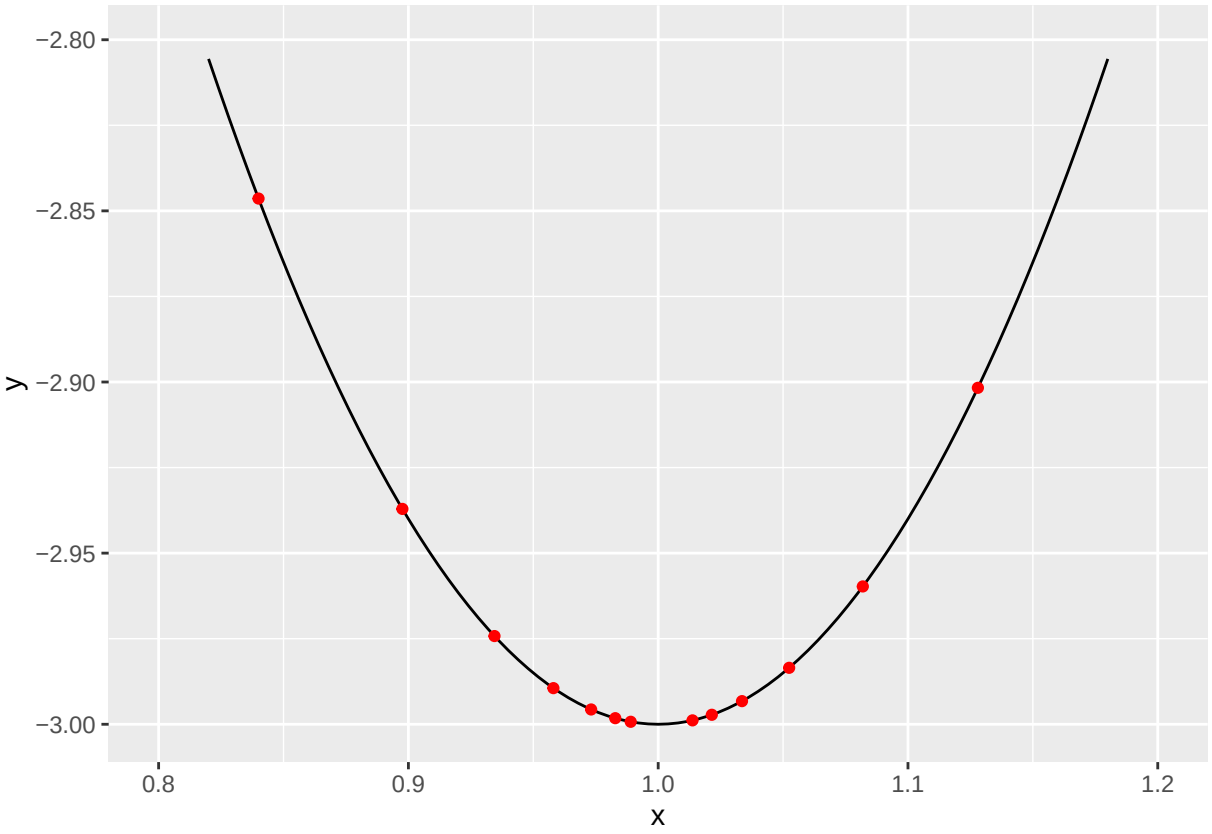
O gradiente descendente unidimensional

```r
# Implementação do gradiente descendente em Kneusel (2022)
library(tidyverse)
f = function(x){
  return(6*x^2 - 12*x + 3)
}

d = function(x){
  return(12*x - 12)
}

x <- seq(-1, 3, length.out = 1000)

p <- ggplot(data.frame(x), aes(x)) +
  stat_function(fun = f, geom = "line")

x <- -0.9 # Ponto inicial de avaliação da função
eta <- 0.03 # tamanho de cada passo

pontos <- data.frame(x = numeric(), y = numeric())

for(i in 1:15) {
  pontos <- rbind(pontos, data.frame(x = x, y = f(x)))
  x <- x - eta * d(x) # Fórmula de atualização do gradiente descendente
}

p <- p + geom_point(data = pontos, aes(x, y), color = "red")
p
```



Problemas possíveis no gradiente descendente: Passos longos; fuga do mínimo.

```r
# Mudando ponto inicial; e aumentando tamanho do passo

x <- seq(-1, 3, length.out = 1000)

p <- ggplot(data.frame(x), aes(x)) +
  stat_function(fun = f, geom = "line") +
  ylim(-3.001, -2.8) +
  xlim(.8, 1.2)

x <- .75 # Ponto inicial de avaliação da função
eta <- 0.15 # tamanho de cada passo

pontos <- data.frame(x = numeric(), y = numeric())

for(i in 1:15) {
  pontos <- rbind(pontos, data.frame(x = x, y = f(x)))
  x <- x - eta * d(x) # Fórmula de atualização do gradiente descendente
}

p <- p + geom_point(data = pontos, aes(x, y), color = "red")
p
```



Gradiente descendente estocástico

O gradiente descendente estocástico é uma variação do gradiente descendente que, ao invés de calcular o gradiente da função em relação a todos os exemplos de treinamento, calcula o gradiente em relação a um 'minibatch' de treinamento por vez. Isso torna o processo de treinamento muito mais rápido, especialmente para conjuntos de dados muito grandes, além de fornecer por vezes estimativas melhores, evitando o algoritmo de cair em mínimos locais

O tamanho deste 'minibatch' é um hiperparâmetro do algoritmo, e deve ser ajustado de acordo com o problema em questão. O autor define epoch, ou época, para definir uma passagem completa pelo conjunto de dados de treinamento. Quando separamos o conjunto de dados originais em 'minibatches', o número de amostras dividido pelo tamanho dos minibatches irá determinar o número de minibatches por época. Ou seja, será reamostrado diversas vezes o conjunto de dados original, e cada vez que o conjunto dessas reamostras for formado, contendo todos os dados originais, teremos completado uma época; e isto se repetirá por k épocas.

Momentos

A ideia dos momentos é de que o gradiente descendente estocástico pode ser melhorado ao considerar a direção e a magnitude dos passos. O autor argumenta que, ao invés de considerar apenas o gradiente da função, podemos considerar um peso relacionado ao gradiente anterior. Este peso é um hiperparâmetro do algoritmo, e deve ser ajustado de acordo com o problema em questão.

Na aplicação, é mais fácil entender a ideia de adicionar este peso. A descida do gradiante é feita em direção ao mínimo local, porém, ao adicionar o peso, o algoritmo considera a direção e a magnitude do passo anterior. Isto é, o algoritmo considera a direção e a magnitude do passo anterior para determinar o passo atual, possibilitando assim 'fugir' de um mínimo local.

Traz também o conceito de 'momento de Nesterov', que é uma variação do momento tradicional, que considera o gradiente da função em um ponto adiantado, e não no ponto atual. Isto é, o algoritmo considera a direção e a magnitude do passo anterior, porém, considera o gradiente da função em um

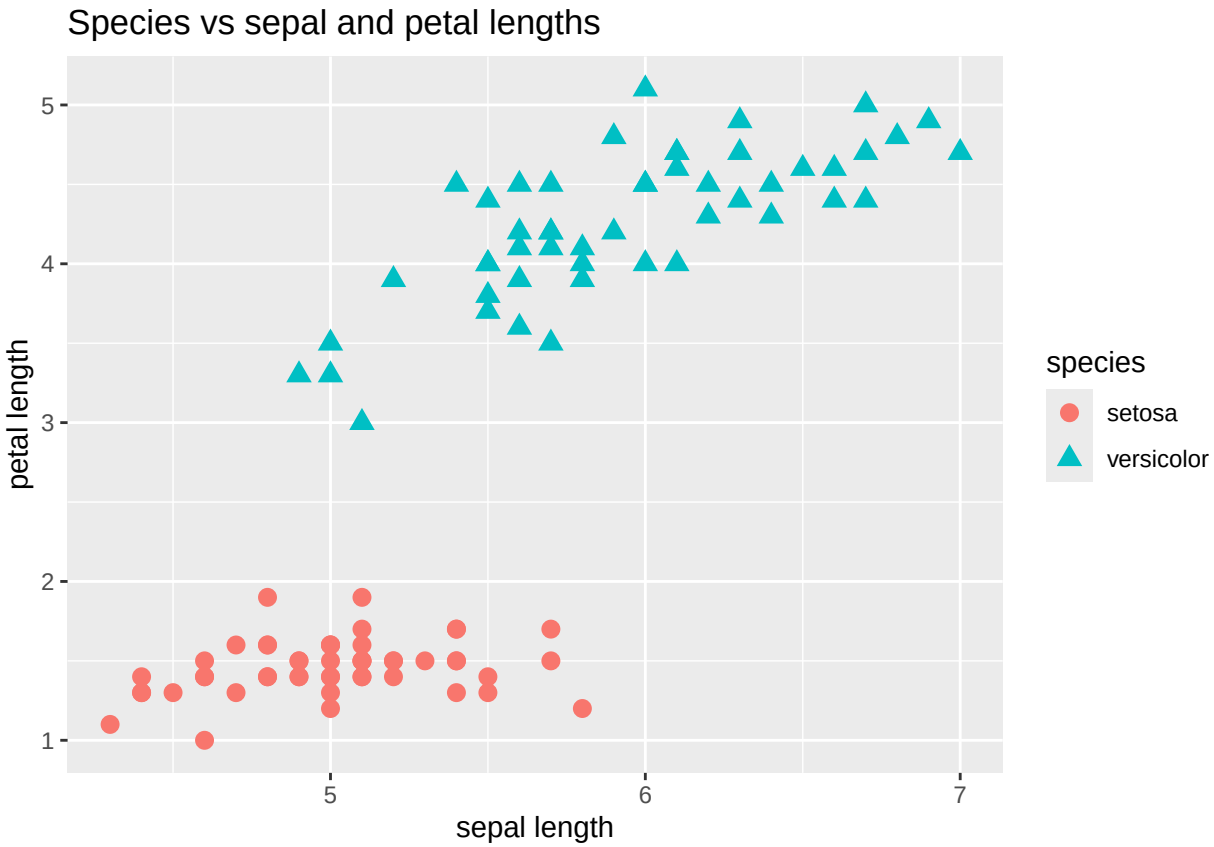
ponto adiantado, possibilitando assim 'fugir' de um mínimo local em menos passos, bem como acertar o mínimo mais rapidamente.

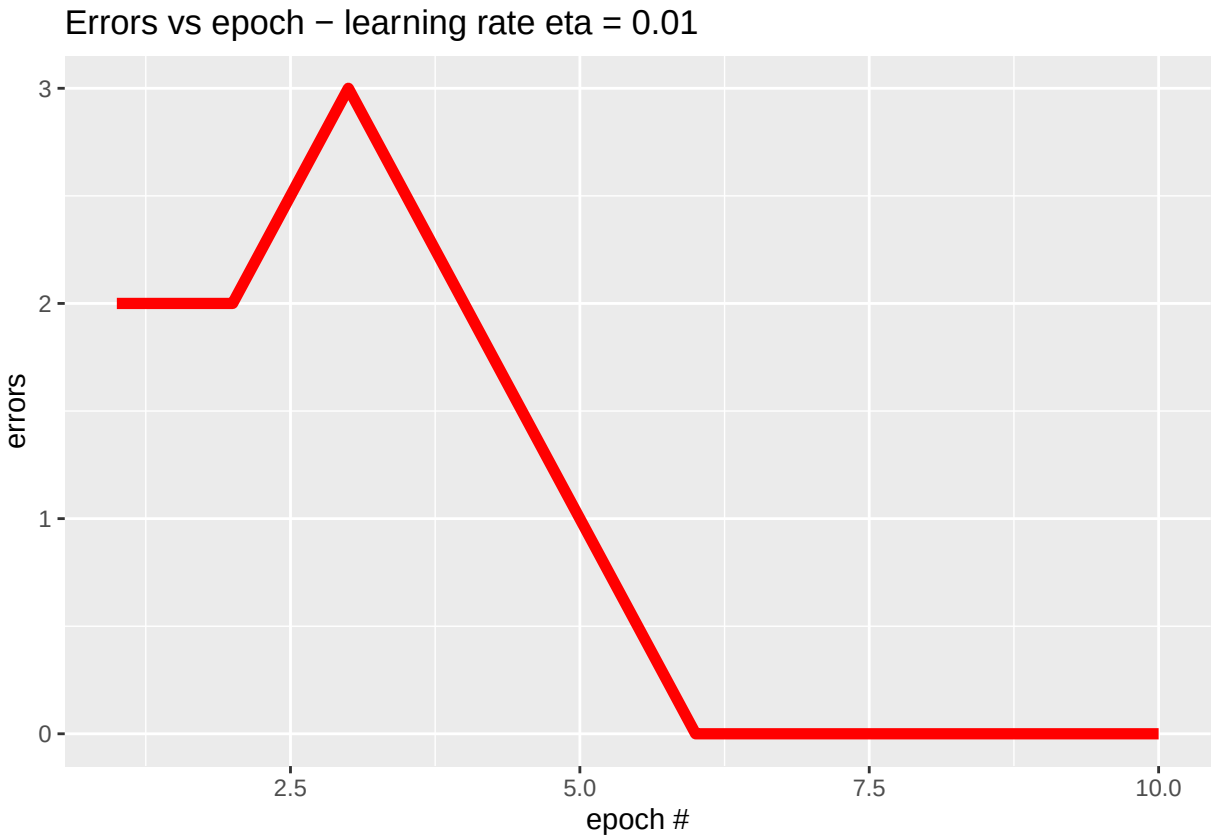
O artigo traz ainda propostas de otimização.

7

HASAN traz uma implementação "crua" do gradiente descendente, de forma a possibilitar analisar o funcionamento da função. Traz como exemplo o clássico banco de dados iris.

Utilizaremos aqui da implementação do autor em outro banco de dados, para observar o funcionamento do algoritmo.

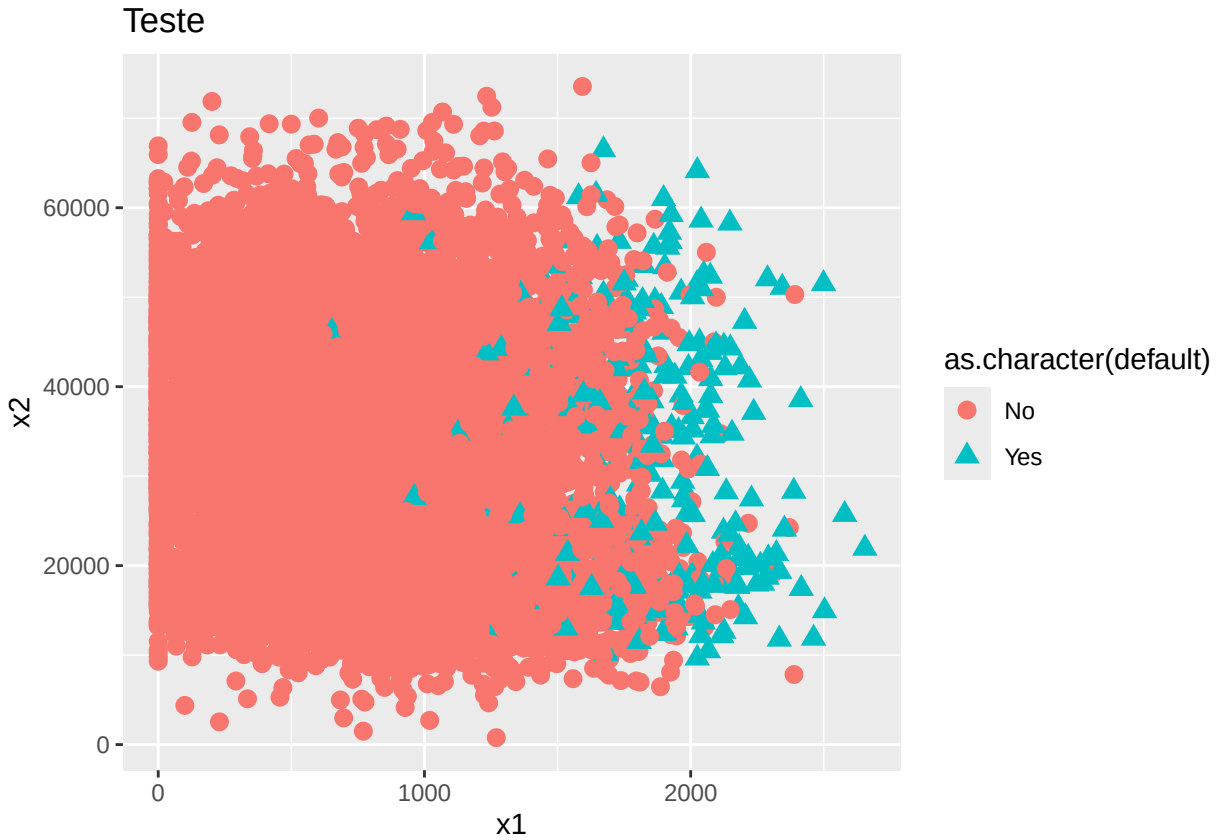


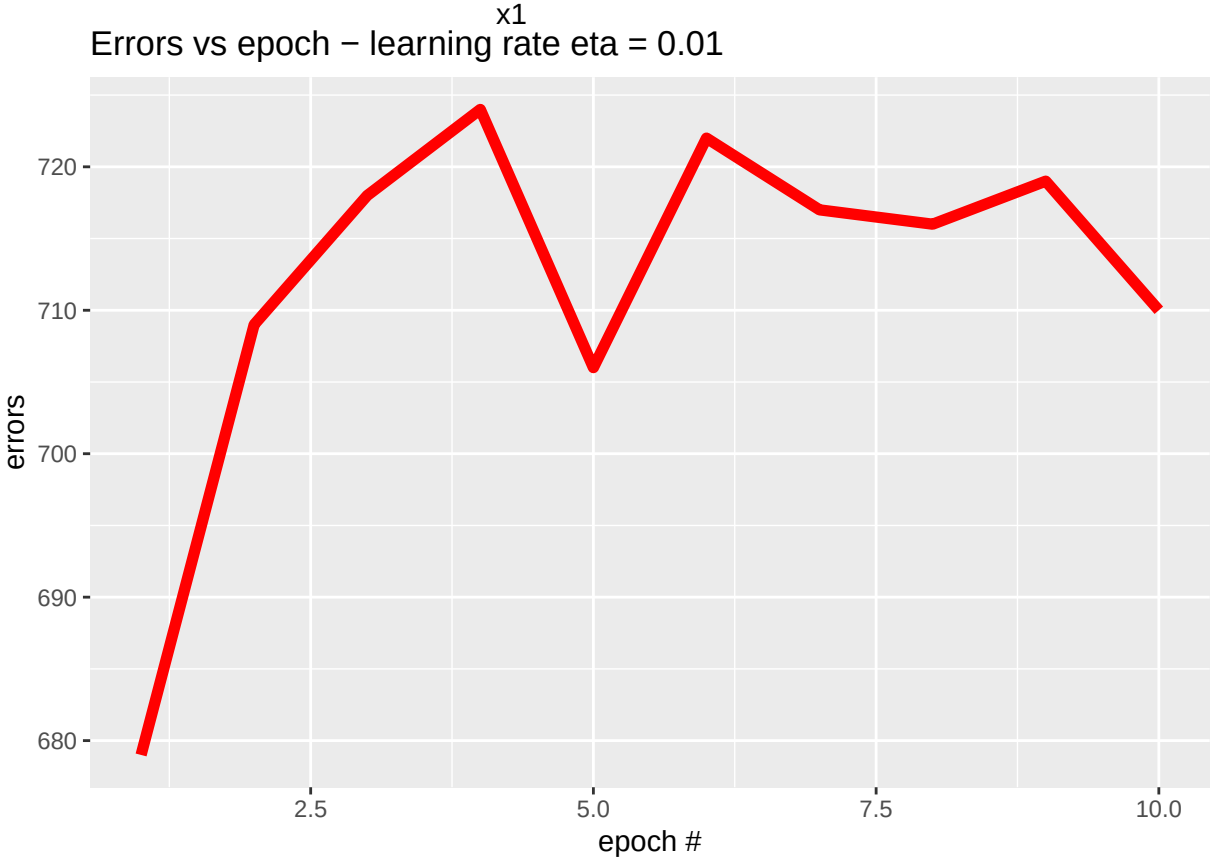


Iremos testar a implementação com outros conjuntos de dados

Default (pacote *ISLR*)

Como veremos no gráfico de dispersão, este é um conjunto de dados de difícil separação linear direta, com muitas intersecções nos pontos. Vejamos como o Perceptron opera neste conjunto, sem nenhum tipo de engenharia de **feature** nos dados.



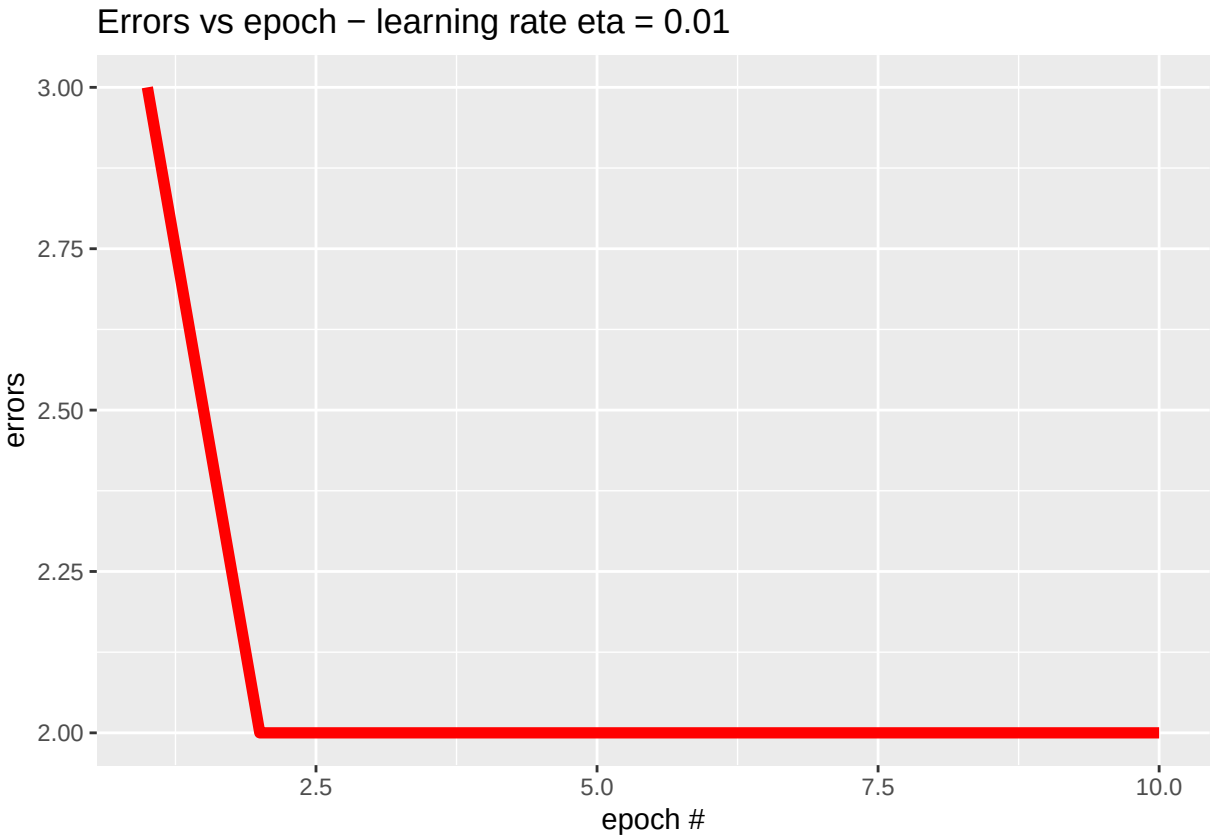


Vemos que a taxa de erro ficou em torno de 700 unidades. Considerando que este era um conjunto com 10000 observações, sendo 333 do tipo 1 (Default) e o restante do tipo -1 (No Default), a taxa de erro de 700 é aceitável, visto o problema de separação linear. Vemos que o perceptron traçou uma threshould possivelmente próximo de onde deveria.

Este exemplo mostra a utilidade primordial do Perceptron, mas demonstra também a necessidade de algoritmos e/ou técnicas mais sofisticadas para lidar com problemas mais complexos como este.

Bank (pacote *gclus*)

Vejamos agora como o Perceptron opera neste outro conjunto de dados. Este é um conjunto já explorado em Multi1; em que não foi difícil separar linearmente as notas falsas das genuínas. O trago aqui, para testar o Perceptron considerando uma dimensionalidade de dados maior. Até agora, testamos apenas para p = 2. Neste caso, irei utilizar p = 6 para testar o Perceptron. Note que não é mais possível trazer o gráfico de dispersão para observar os dados.



Notamos que o algoritmo performou muito bem em separar as notas em genuínas e falsas. A taxa de erro foi de 2, o que é um resultado excelente. Isto mostra que o Perceptron é uma ferramenta útil para problemas simples, mas que pode ser facilmente superado por algoritmos mais sofisticados para problemas mais complexos.

Considerações sobre o Perceptron:

O perceptron é um algoritmo relativamente simples ao que é executado e aplicado atualmente na área de machine learning. Um sabido problema do algoritmo é que ele admite infinitas soluções em problemas linearmente separáveis. Ainda assim, é uma ferramenta útil para tratar de problemas simples, ainda sendo capaz de atingir alguma precisão aceitável para conjuntos de dados simples.

8

a)

Utilizando a função *perceptron* do pacote *mlpack*

Iremos refazer o exemplo do conjunto de dados *bank*, mas dessa vez utilizando a implementação do *perceptron* do pacote *mlpack*.

Notas: - Esta é uma implementação direta da documentação do pacote *mlpack* para o conjunto de dados *bank*. A documentação do pacote e o código originalmente utilizado para a adaptação pode ser encontrado executando o comando *?perceptron* no console do *R*; após ter carregado o pacote *mlpack*. - O pacote tem algumas peculiaridades de *input*. Os labels devem ser >1, ou seja, não é possível utilizar o clássico binário 0,1. Neste caso, utilizaremos 1 e 2. - Além disso, o input dos labels deve ser uma matriz, ainda que na prática sejam um vetor. É necessário fazer a conversão, senão o pacote retorna erro.

```
data("bank")
training_data <- bank[sample(1:nrow(bank), 0.7*nrow(bank)),]
test_data <- bank[-sample(1:nrow(bank), 0.7*nrow(bank)),]
training_labels = ifelse(training_data[,1]==0,1,2) # Para o pacote mlpack, os labels devem ser 1 e 2
test_labels = ifelse(test_data[,1]==0,1,2)
training_data = training_data[,2:7]
test_data = test_data[,2:7]

training_labels <- as.matrix(training_labels) # O input dos labels deve ser uma matriz para o pacote
test_labels <- as.matrix(test_labels)

output <- mlpack::perceptron(training=training_data, labels=training_labels)
perceptron_model <- output$output_model

output <- mlpack::perceptron(input_model=perceptron_model, test=test_data)
predictions <- output$predictions
setdiff(test_labels,predictions)
```

```
## numeric(0)
```

A implementação é incrivelmente rápida, pelo fato de ser escrito em *C++*. O número de iterações foi deixado *default*, que no caso são 1000. Não consegui passar de 100 iterações na implementação em *R* anterior sem o computador travar. O pacote *mlpack* é uma excelente alternativa para implementações mais complexas, visto que é muito mais rápido e eficiente.

Quanto ao resultado do modelo, separei o conjunto em treino (70%) e teste (30%) para validação do modelo. O resultado da validação é o *setdiff* retornado acima; ou seja: O modelo acertou 100% do conjunto de testes!

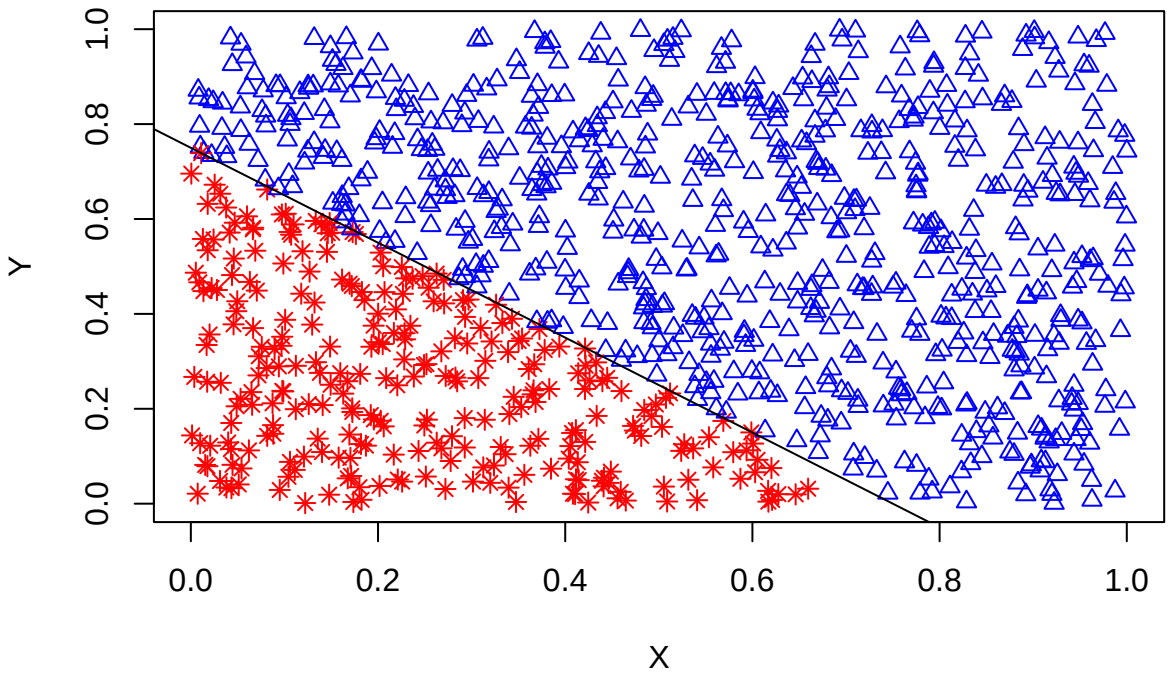
b)

Não encontrei outras implementações envelopadas em pacotes do perceptron além do *mlpack*. Entretando, encontrei esta implementação manual do perceptron, com possibilidade de visualização em gráfico 2D e 3D! A implementação original pode ser consultada aqui

Output da implementação em 2D:

```
Random.Unit <-function(n, dim, threshold) {
  points <- runif(n * dim)
  points <- matrix(points, ncol = dim)
  label <- ifelse(apply(points, 1, sum) < threshold, -1, 1)
  return(cbind(label, x0 = rep(1, n), points))
```

```
}

Classify <- function(x, weights) {
  return(sign(x %*% weights))
}

Perceptron <- function(data, threshold) {
  w <- c(-threshold, runif(ncol(data) - 2))
  n <- nrow(data)
  label <- data[ , 1]
  obs <- data[ , 2:ncol(data)]
  misclassfied <- TRUE
  while (misclassfied) {
    misclassfied <- FALSE
    for (i in 1:n) {
      if (label[i] * Classify(obs[i , ], w) <= 0) {
        w <- w + label[i] * obs[i , ]
        misclassfied <- TRUE
      }
    }
  }
  return(w)
}

Plot2D <- function(points, a, b) {
  plot(points[, 3:4], xlab = "X", ylab = "Y",
       pch = ifelse(points[, 1] == 1, 2, 8),
       col = ifelse(points[, 1] == 1, "blue", "red"))
  abline(a, b)
}

THRESHOLD <- 0.75
pts <- Random.Unit(1000, 2, THRESHOLD)
Plot2D(pts, THRESHOLD, -1)
```



c)

Implementando o perceptron em *Julia*

NOTA: Esta é uma adaptação simples e direta da documentação do pacote *Julia Perceptron*, para o conjunto de dados *iris*. A documentação do pacote e o código originalmente utilizado para a adaptação pode ser encontrado aqui

```
using Perceptrons, RDatasets, DataFrames, Plots, Random, StatsBase, Random, MLUtils

iris = dataset("datasets", "iris")

iris = iris[iris.Species .!= "virginica", :]

iris.Species = iris.Species .== "setosa"

iris = iris[shuffle(1:end), :]

train_proportion = 0.7
train_size = Int(floor(train_proportion * nrow(iris)))
train = iris[1:train_size, :]
test = iris[train_size+1:end, :]

X_train = Matrix(train[:, 1:4])
Y_train = Vector(train[:, 5])

X_test = Matrix(test[:, 1:4])
Y_test = Vector(test[:, 5])
Y_train = convert(Array{Float64}, Y_train)

model = Perceptrons.fit(X_train,Y_train,centralize=true,mode="voted")
Y_pred = Perceptrons.predict(model,X_test)
```

Deste código *Julia*, notamos que a rotina de implementação não difere muito do aplicado em R, apesar da sintaxe de assemelhar mais a *Python*. Começamos carregando os pacotes, que devem ser previamente instalados. Em seguida, carregamos o conjunto de dados *iris*, contido no pacote *Julia RDatasets*. Excluímos "virginica" dos dados para binarizar o problema, exatamente como executado no exercício 7. Depois, binarizamos a espécie (setosa = 0, versicolor = 1). Embaralhamos então o conjunto de dados para então separar o conjunto em treino (70%) e teste (30%). Guardamos cada conjunto em um objeto, fazemos as conversões de tipo de variável necessário para a implementação (matriz de elementos Float64; vetor de elementos Float64 para labels.) Após isso, utilizamos o método (função) Perceptron do pacote Perceptrons para treinar o modelo. Numa sintaxe muito similar ao python (fit-predict), o modelo é ajustado e após utilizado para predições, comparado ao vetor de teste para verificar a quantidade de acertos do modelo, que para esta rotina foi:

```
println("[Perceptron] accuracy : $(acc(Y_test,Y_pred))")
```

```
## [Perceptron] accuracy : 1.0
```

9 (Stefan)

10 (Stefan)