Contenido

Estimación de los parámetros de un modelo de regresión lineal mediante descenso de gradiente.

Machine Learning by Andrew Ng, Stanford University

Considérese el modelo de regresión lineal simple:

$$Y = \hat{\beta}_0 + \hat{\beta}_1 X_1 + e$$

Dada una muestra de entrenamiento, el ajuste de un modelo lineal consiste en encontrar los parámetros estimados ($\hat{\beta}_0$, $\hat{\beta}_1$) que definen la recta que pasa más cerca de todos los puntos. Para ello se tiene que minimizar la suma de cuadrados residuales:

$$RSS(\beta_0, \beta_1) = \sum_{I=1}^{n} \left(y_i - \left(\hat{\beta}_0 + \hat{\beta}_1 x_1\right)\right)^2$$

Para este problema, existe una solución explícita con la se puede obtener el mínimo:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$$\hat{\beta}_0 + \bar{y} - \hat{\beta}_1 \bar{x}$$

En muchos otros casos no existe tal posibilidad, haciendo necesaria la utilización de algoritmos de optimización. La sencillez del problema de regresión lineal simple hace que sea un buen ejemplo para ilustrar cómo se solucionan problemas de optimización. Además, esta función tiene una propiedad muy interesante que facilita mucho su optimización, se trata de una función convexa, por lo que tiene un único mínimo que coincide con el mínimo global.

Los algoritmos de optimización siguen un proceso iterativo en el que, partiendo de una posición inicial dentro del dominio de la función, realizan desplazamientos en la dirección correcta hasta alcanzar el mínimo. Para poder llevar a cabo el proceso se necesita conocer:

- La función objetivo a minimizar.
- La posición desde la que iniciar la búsqueda.
- La distancia (step) que se va a desplazar el algoritmo en cada iteración de búsqueda. Es importante escoger un "step" adecuado en cada escenario. Si es muy pequeño, se tardará demasiado en llegar al mínimo y, si es demasiado grande, el algoritmo saltará de una región a otra pasando por encima del mínimo sin alcanzarlo.
- La dirección de búsqueda en la que se produce el desplazamiento.
- La tolerancia: Al tratarse de un algoritmo iterativo, hay que indicar una regla de parada. Lo idóneo es que el algoritmo se detenga al encontrar el mínimo, pero como normalmente se desconoce cuál es, hay que conseguir que se pare lo más cerca posible. Una forma de cuantificar la proximidad al mínimo es controlar cuanto desciende el valor de la función entre iteraciones consecutivas. Si el descenso es muy pequeño, significa que se encuentra muy

cerca del valor. La tolerancia se define como un valor tal que, si la diferencia en el descenso es menor o igual, se considera que se ha alcanzado el mínimo (el algoritmo converge). Una alternativa a calcular la distancia entre las dos iteraciones es medir la longitud del vector gradiente en cada iteración. Si su longitud es muy pequeña, también lo es la distancia al mínimo.

- Iteraciones máximas: Si la posición de inicio en la búsqueda está muy alejada del mínimo de la función y el step es muy pequeño, se pueden necesitar muchas iteraciones para alcanzarlo. Para evitar un proceso iterativo excesivamente largo, se suele establecer un número máximo de iteraciones permitidas.

Optimización por el método de descenso de gradiente

El algoritmo de descenso de gradiente se caracteriza porque emplea como dirección de búsqueda aquella en la que la función desciende en mayor medida. Esta dirección se corresponde con (gradiente) de la función. El gradiente de la función es el vector formado por las derivadas parciales de la función respecto a cada variable.

La función objetivo a optimizar en regresión lineal es la suma de los residuos cuadrados:

$$J(\beta_0, \beta_1) = \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_i))^2 = \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2$$

Para facilitar posteriores simplificaciones, se multiplica la función por el factor 1/2.

$$J(\beta_0, \beta_1) = \frac{1}{2} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2$$

Las derivadas parciales de la función respecto a cada una de las variables, en este caso, son:

$$\frac{dJ}{d(\beta_0)} = 2 * \frac{1}{2} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i) * -1 = -\sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)$$

$$\frac{dJ}{d(\beta_1)} = 2 * \frac{1}{2} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i) * -x_i = -\sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i) x_i$$

El vector gradiente para un modelo lineal simple es, por lo tanto:

$$\begin{bmatrix} -\sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i) \\ -\sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i) x_i \end{bmatrix}$$

Algoritmo: Método de descenso de gradiente para $J(\beta 0,\beta 1)$

1. Seleccionar valores iniciales

$$\hat{\beta}_0, \hat{\beta}_1, t > 0$$

2. Iniciar un proceso iterativo en el que:

$$\hat{\beta}_0 = \hat{\beta}_0 - t * -\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i)$$

$$\hat{\beta}_1 = \hat{\beta}_1 - t * -\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i)x_i$$

hasta alcanzar una condición de parada.

Véase ahora un ejemplo práctico. En primer lugar, se simula una muestra a partir de una función conocida en la que el usuario determina los parámetros poblacionales. Conocer los parámetros poblacionales permitirá evaluar cómo de buenas son las estimaciones obtenidas.

```
# Estimación de los parámetros de un modelo de regresión lineal
# mediante optimización convexa
# Optimización por el método de descenso de gradiente
n <- 100
# muestra
set.seed(123) # Para permitir reproducibilidad
x <- runif(n, min = 0, max = 5) # n puntos aleatorios en el intervalo [mi
n,max]  Generamos valores de x (aleatorios)
str(x)
```

puntos aleatorios entre 0 y 5

```
##  num [1:100] 1.44 3.94 2.04 4.42 4.7 ...
```

Asignando coeficientes del modelo a optimizar

```
beta0 <- 2    # Parametro beta0 del modelo
beta1 <- 5    # Parametro beta1 del modelo
set.seed(123) # Para permitir reproducibilidad
epsilon <- rnorm(n, sd = 1) # error (con desviacion tipica sd=1)
str(épsilon)
```

```
##  num [1:100] -0.5605 -0.2302 1.5587 0.0705 0.1293 ...
```

Generando un nuevo y con el valor epsilon (predecimos)

```
y <- beta0 + beta1 * x + epsilon # calculo de cada y_i
datos <- data.frame(x, y)
head(datos, 3)
```

```
##          x        y
## 1 1.437888  8.628962
## 2 3.941526 21.477451
## 3 2.044885 13.783131
```

```
str(datos)
```

```
## 'data.frame':    100 obs. of  2 variables:
##  $ x: num  1.44 3.94 2.04 4.42 4.7 ...
##  $ y: num  8.63 21.48 13.78 24.15 25.64 ...
```

Generamos el modelo con la función lm() que aplica la solución explicita. Con valores estimados

```
modelo_lm <- lm(y ~ x , data = datos)
coefficients(modelo_lm)
```

```
## (Intercept)           x
##    2.117657    4.989068
```

Como era de esperar, las estimaciones de los coeficientes de regresión obtenidos mediante lm() se aproximan mucho a los parámetros poblacionales.

Ajuste del modelo por optimización de la suma de residuos cuadrados

```
# Ajuste del modelo por optimizacion de la suma de residuos cuadrados
```

```r
# FUNCION OBJETIVO A MINIMIZAR
sum_residuos <- function(x, y, beta_0, beta_1){
  return(sum((y - (beta_0 + beta_1 * x))^2))
}
# CALCULO DE GRADIENTE PARA MODELO LINEAL SIMPLE
calc_gradiente <- function(beta_0, beta_1, x, y){
  # beta_0: valor del interseccion
  # beta_1: coeficiente de regresion del predictor
  # x: valores del predictor observados en la muestra
  # y: valores de la variable respuesta observados en la muestra
  grad_1 <- sum(y - beta_0 - beta_1 * x)
  grad_2 <- sum((y - beta_0 - beta_1 * x) * x)
  return(c(grad_1, grad_2)) }

# OPTIMIZACION
optimizacion_grad <- function(beta_0 = 2, beta_1 = 5, x, y, t = 0.001,
                              max_iter = 10000, tolerancia = 1e-10){
  # Matriz para almacenar el valor de las estimaciones en cada iteracion
  estimaciones <- matrix(NA, nrow = max_iter, ncol = 4)
  colnames(estimaciones) <- c("iteracion", "beta0", "beta1", "residuos")

  for(i in 1:max_iter){
    gradiente <- calc_gradiente(beta_0 = beta_0, beta_1 = beta_1, x = x, y
= y)

    if(sqrt(sum(gradiente^2)) < tolerancia){
      # Si el tamano del vector es menor que la tolerancia,
      # se considera que el proceso a llegado a convergencia.
      message("El algoritmo ha alcanzado convergencia")
      break
    }else{
      estimaciones[i, 1] <- i
      estimaciones[i, 2] <- beta_0
      estimaciones[i, 3] <- beta_1
      estimaciones[i, 4] <- sum_residuos(x, y, beta_0, beta_1)
      beta_0 <- beta_0 + t * gradiente[1]
```

```
      beta_1 <- beta_1 + t * gradiente[2]
    }
  }

  print(paste("Estimacion de beta_0:", beta_0))
  print(paste("Estimacion de beta_1;", beta_1))
  print(paste("Numero de iteraciones:", i))
  print(paste("Suma de residuos cuadrados:", estimaciones[i-1, 4]))
  return(list(beta_0 = beta_0,
              beta_1 = beta_1,
              iteraciones = i,
              estimaciones = as.data.frame(na.omit(estimaciones))
  )
  )
}

resultados <- optimizacion_grad(beta_0 = 2, beta_1 = 5, x = datos$x,
                                y = datos$y, t = 0.001, max_iter = 10000,
                                tolerancia = 1e-6)
```

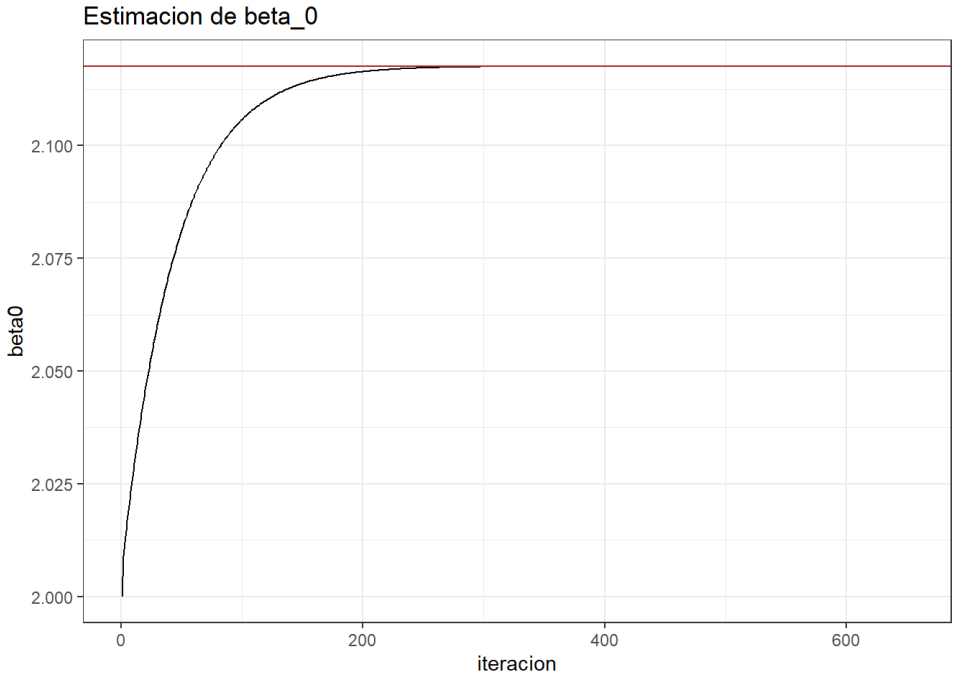
```
## El algoritmo ha alcanzado convergencia
## [1] "Estimacion de beta_0: 2.11765679672856"
## [1] "Estimacion de beta_1; 4.98906813521441"
## [1] "Numero de iteraciones: 655"
## [1] "Suma de residuos cuadrados: 82.4660264805964"
```

Empleando un valor $t = 0.001$, los valores estimados mediante optimización por descenso de gradiente son casi idénticos a los obtenidos mediante la función lm().

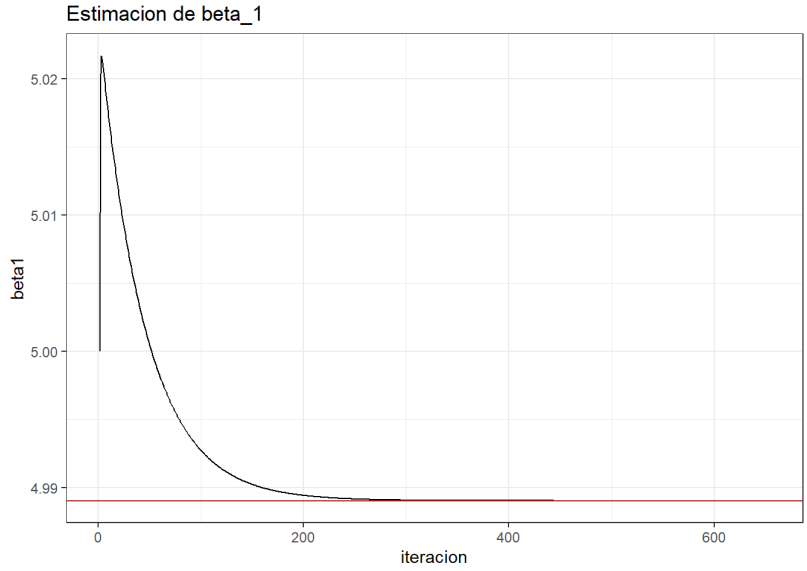
Los siguientes gráficos muestran la evolución de los parámetros estimados y el error tras cada iteración.

```
library(ggplot2)
ggplot(data = resultados$estimaciones,
       aes(x = iteracion, y = beta0)) +
  geom_path() +
  geom_hline(yintercept = modelo_lm$coefficients[1], color = "firebrick") +
```

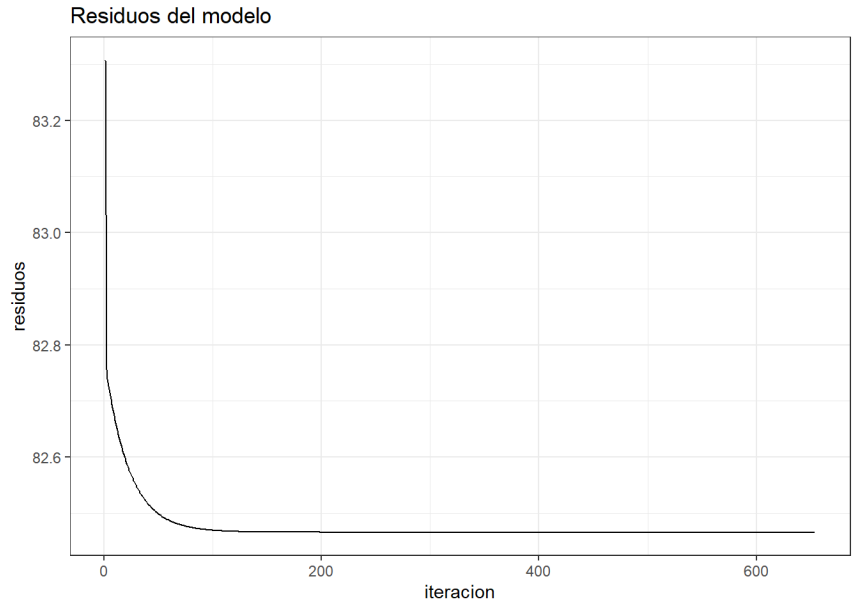
```
  theme_bw() +
  ggtitle("Estimacion de beta_0")
```




```
ggplot(data = resultados$estimaciones,
       aes(x = iteracion, y = beta1)) +
  geom_path() +
  geom_hline(yintercept = modelo_lm$coefficients[2], color = "firebrick")
+
  theme_bw() +
  ggtitle("Estimacion de beta_1")
```

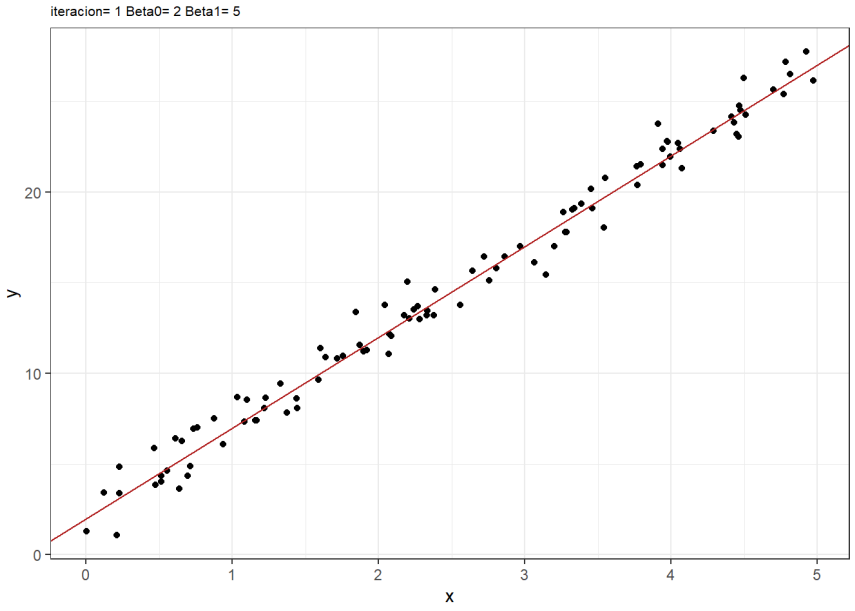


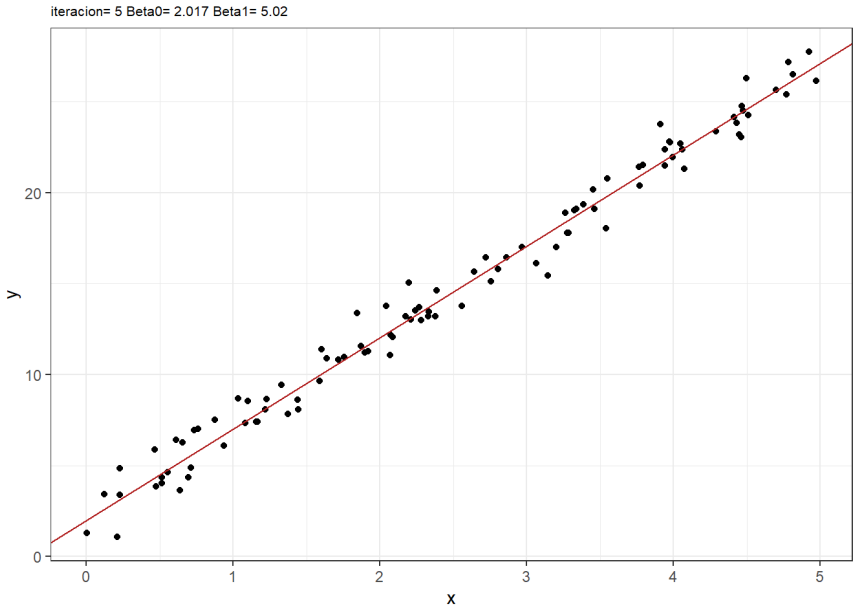

```
ggplot(data = resultados$estimaciones,
       aes(x = iteracion, y = residuos)) +
  geom_path() +
  theme_bw() +
  ggtitle("Residuos del modelo")
```

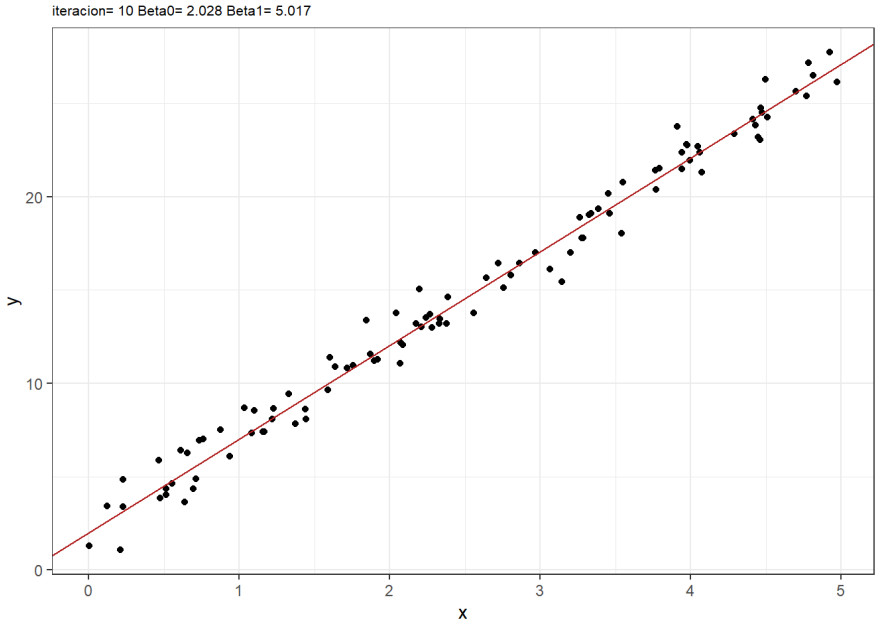


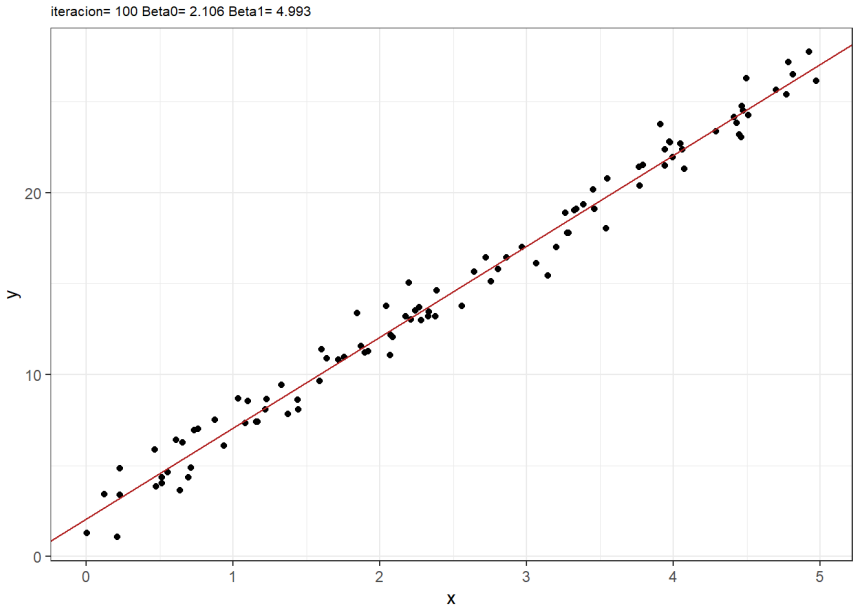
Es importante evaluar gráficamente la evolución de los parámetros estimados y de la función objetivo, sobre todo esta última, ya que aporta mucha información sobre si el algoritmo está aproximándose al mínimo (descenso en la curva) y si está llegando a convergencia (estabilización de la curva).

```r
for(i in c(1,5,10,100)) {
  beta_0 <- round(resultados$estimaciones$beta0[i],3)
  beta_1 <- round(resultados$estimaciones$beta1[i],3)
  p  <- ggplot(data = datos, aes(x = x, y = y)) +
    geom_point() +
    geom_abline(intercept = beta_0, slope = beta_1,
                color = "firebrick") +
    ggtitle(paste("iteracion=",i,"Beta0=",beta_0, "Beta1=",beta_1)) +
    theme_bw() +
    theme(plot.title = element_text(size = 8))
  print(p)
}
```




iteracion= 5 Beta0= 2.017 Beta1= 5.02
y
x
0
10
20
0
1
2
3
4
5


iteracion= 10 Beta0= 2.028 Beta1= 5.017
y
x
0
10
20
0
1
2
3
4
5



Este método está muy influenciado por el tamaño de *t*. Para valores altos, la suma de cuadrados residuales puede ser un valor demasiado grande para el software empleado lo almacene. Por ejemplo, este mismo código ejecutado para una muestra de tamaño *n=1000* devuelve un error. Las simulaciones que he realizado parecen indicar que empleando un valor de *t* aproximadamente de *1/(10n)* se adapta bastante bien.

```
n <- 10000
set.seed(123)
x <- runif(n, min = 0, max = 5)
beta0 <- 2
beta1 <- 5
set.seed(123)
epsilon <- rnorm(n, sd = 1)
y <- beta0 + beta1 * x + epsilon
datos2 <- data.frame(x, y)
resultados <- optimizacion_grad(beta_0 = 2, beta_1 = 5, x = datos2$x,
                                y = datos2$y, t = 1/(10*length(x)),
                                max_iter = 10000, tolerancia = 1e-6)
```

```
## El algoritmo ha alcanzado convergencia
## [1] "Estimacion de beta_0: 1.99327105963717"
## [1] "Estimacion de beta_1; 5.0017514798942"
## [1] "Numero de iteraciones: 721"
```

```
## [1] "Suma de residuos cuadrados: 9971.6909756411"
```

Ajuste del modelo por optimización de la media de residuos cuadrados

En lugar de minimizar la suma de cuadrados residuales, también se puede minimizar la media de los residuos cuadrados.

```
# Ajuste del modelo por optimizacion de la media de residuos cuadrados

# FUNCION OBJETIVO A MINIMIZAR
# ============================
mean_residuos <- function(x, y, beta_0, beta_1){
  return((1/length(x)) * sum((y - (beta_0 + beta_1 * x))^2))
}

# CALCULO DE GRADIENTE PARA MODELO LINEAL SIMPLE
# ====================
calc_gradiente <- function(beta_0, beta_1, x, y){
  # beta_0: valor de interseccion
  # beta_1: coeficiente de regresion del predictor
  # x: valores del predictor observados en la muestra
  # y: valores de la variable respuesta observados e n la muestra
  grad_1 <- (1/length(x)) * sum(y - beta_0 - beta_1 * x)
  grad_2 <- (1/length(x)) * sum((y - beta_0 - beta_1 * x) * x)
  return(c(grad_1, grad_2))
}



# OPTIMIZACION
# ========================
optimizacion_grad <- function(beta_0 = 2, beta_1 = 5, x, y, t = 0.001,
                              max_iter = 10000, tolerancia = 1e-10){
  # Matriz para almacenar el valor de las estimaciones en cada iteracion
  estimaciones <- matrix(NA, nrow = max_iter, ncol = 4)
  colnames(estimaciones) <- c("iteracion", "beta0", "beta1", "residuos")
  for(i in 1:max_iter){
    gradiente <- calc_gradiente(beta_0 = beta_0, beta_1 = beta_1, x = x,
                                y = y)
```

```
    if(sqrt(sum(gradiente^2)) < tolerancia){
      # Si el tamano del vector es menor que la tolerancia,
      # se considera que el proceso a llegado a convergencia.
      message("El algoritmo ha alcanzado convergencia")
      break
    }else{
      estimaciones[i, 1] <- i
      estimaciones[i, 2] <- beta_0
      estimaciones[i, 3] <- beta_1
      estimaciones[i, 4] <- mean_residuos(x, y, beta_0, beta_1)
      beta_0 <- beta_0 + t * gradiente[1]
      beta_1 <- beta_1 + t * gradiente[2]
    }
  }
  print(paste("Estimacion de beta_0:", beta_0))
  print(paste("Estimacion de beta_1;", beta_1))
  print(paste("Numero de iteraciones:", i))
  print(paste("Media residuos cuadrados:", estimaciones[i-1, 4]))
  return(list(beta_0 = beta_0,
              beta_1 = beta_1,
              iteraciones = i,
              estimaciones = as.data.frame(na.omit(estimaciones))
  )
  )
}
resultados <- optimizacion_grad(beta_0 = 10, beta_1 = 10, x = datos$x,
                                y = datos$y, t = 0.1, max_iter = 10000,
                                tolerancia = 1e-6)
```

```
## El algoritmo ha alcanzado convergencia
## [1] "Estimacion de beta_0: 2.11765258759229"
## [1] "Estimacion de beta_1; 4.9890694466244"
## [1] "Numero de iteraciones: 451"
## [1] "Media residuos cuadrados: 0.824660264810596"
```

Empleando como función objetivo la media de los residuos cuadrados, el valor de *t* puede ser mayor que cuando se utiliza la suma de los residuos cuadrados (partiendo del mismo punto inicial).

Método de descenso de gradiente estocástico (Stochastic Gradient Descent)

Machine Learning by Andrew Ng, Stanford University

En el método de descenso de gradiente visto en el apartado anterior, se utiliza la muestra de entrenamiento al completo para cada actualización del valor de los parámetros (cada iteración). Esto supone un alto coste computacional cuando la muestra contiene muchas observaciones ya que, con frecuencia, se necesitan cientos o miles de actualizaciones hasta llegar a una estimación aceptable. Existe una alternativa al método de descenso de gradiente llamada gradiente estocástico que proporciona buenos resultados. La idea es que, en lugar de esperar a evaluar todas las observaciones para actualizar los parámetros, se puedan ir actualizando con cada observación por separado. El algoritmo para el caso particular del modelo de regresión lineal simple es el siguiente:

Algoritmo: Método de descenso de gradiente estocástico para $J(\beta 0,\beta 1)$

1. Ordenación aleatoria de las observaciones
2. Seleccionar valores iniciales

$$\hat{\beta}_0, \hat{\beta}_1, t > 0$$

3. Repetir

   Iniciar un proceso iterativo en el que para i=1,..,n

$$\hat{\beta}_0 = \hat{\beta}_0 - t * -(y_i - \beta_0 - \beta_1 x_i)$$

$$\hat{\beta}_1 = \hat{\beta}_1 - t * -(y_i - \beta_0 - \beta_1 x_i) x_i$$

hasta alcanzar una condición de parada.

Con este método se actualiza el valor de los parámetros con cada observación de la muestra de entrenamiento, evitando utilizar todos los datos en cada iteración. Puede observarse que el algoritmo tiene un bucle externo que engloba a un bucle interno de actualización de parámetros. Este blucle externo determina el número de veces que se recorre el set de datos completo en la optimización. En el ámbito de *machine learning* se le conoce como *epochs*. Su valor óptimo depende del tamaño del set de datos de entrenamiento, valores entre 1 y 10 suelen generar buenos resultados.

```
# Metodo de descenso de gradiente estocastico
# (Stochastic Gradient Descent)

opt_grad_estoc <- function(beta_0 = 2, beta_1 =5, x, y, t = 0.01,
```

```
                              epochs = 10){
  # Matriz para almacenar el valor de las estimaciones en cada epoch
  estimaciones <- matrix(NA, nrow = epochs, ncol = 3)
  colnames(estimaciones) <- c("epoch", "beta0", "beta1")
  for(j in 1:epochs){
    estimaciones[j, 1] <- j
    estimaciones[j, 2] <- beta_0
    estimaciones[j, 3] <- beta_1
    for(i in 1:length(x)){
      beta_0 <- beta_0 + t * (y[i] - beta_0 - beta_1 * x[i])
      beta_1 <- beta_1 + t * x[i] * (y[i] - beta_0 - beta_1 * x[i])
    }
  }
  print(paste("Estimacion de beta_0:", beta_0))
  print(paste("Estimacion de beta_1;", beta_1))
  print(paste("Numero de epochs:", j))
  return(list(beta_0 = beta_0,
              beta_1 = beta_1,
              epochs= j ,
              estimaciones = estimaciones
  ))
}
resultados <- opt_grad_estoc(beta_0 = 10, beta_1 = 10, x = datos2$x,
                             y = datos2$y, t =0.01, epochs = 10)
```

```
## [1] "Estimacion de beta_0: 1.8814987610849"
## [1] "Estimacion de beta_1; 4.99802652516896"
## [1] "Numero de epochs: 10"
```

Bibliografía

OpenIntro Statistics: Third Edition, David M Diez, Christopher D Barr, Mine Çetinkaya-Rundel

An Introduction to Statistical Learning: with Applications in R (Springer Texts in Statistics)

Linear Models with R, Julian J.Faraway

An introduction to Logistic Regresion Analysis and Reporting. Chao-Ying Joanne Peng

http://www.ats.ucla.edu/stat/r/dae/logit.htm

http://ww2.coastal.edu/kingw/statistics/R-tutorials/index.html

R Tutorials by William B. King, Ph.D http://ww2.coastal.edu/kingw/statistics/R-tutorials/

Points of Significance: Association, correlation and causation. Naomi Altman & Martin Krzywinski Nature Methods

Points of Significance: Simple linear regression Naomi Altman & Martin Krzywinski. Nature Methods

Resampling Data: Using a Statistical Jackknife S. Sawyer | Washington University | March 11, 2005

http://www.biostat.jhsph.edu/~bcaffo/651/files/lecture12.pdf

https://en.wikipedia.org/wiki/Resampling_(statistics)#Jackknife

The Trusty Jackknife Method identifies outliers and bias in statistical estimates by I. Elaine Allen and Christopher A. Seaman