Trabajo 2

Francisco Javier Caracuel Beltrán

26 de Abril de 2017

A continuación, se detalla el resultado del trabajo 2 de Aprendizaje Automático. Este documento incluye todo el código utilizado en el archivo *trabajo2.R*, así como los comentarios añadidos durante su creación.

Como se entrega el fichero *trabajo2.Rmd*, no se incluyen puntos de parada en el código fuente que se encuentra en el fichero *trabajo2.R*.

Se establece el directorio de trabajo

```
setwd("/mnt/A0DADA47DADA18FC/Fran/Universidad/3º/2 Cuatrimestre/AA/Mis prácticas/Práctica 2/datos")
```

Se establece la semilla para la generación de datos aleatorios

```
set.seed(1822)
```

Ejercicio 1. Gradiente Descendente. Implementar el algoritmo de gradiente descendente.

Apartado a

```
####################
# Función 1a
#
f1a = function(u,v){
  ((u^2)*exp(v)-(2*(v^2)*exp(-u)))^2
}
```

Se debe calcular la derivada parcial de la expresión $E(u,v) = ((u^2 * e^v) - (2v^2 * e^{-u}))^2$

Punto 1

El resultado de calcular las derivadas es:

$E'(u) = 2((u^2 * e^v) - (2v^2 * (e^{-u}))) * (2ue^v + 2v^2 * e^{-u})$

$E'(v) = 2((u^2 * e^v) - (2v^2 * e^- u)) * (u^2 * e^v - 2e^- u * 2v)$

```
####################
# Se realiza la derivada parcial de la expresión:
# E(u,v) = ((u^2*e^v) - (2v^2*e^-u))^2
#
# E'(u) = 2((u^2*e^v) - (2v^2*e^-u)) * (2ue^v + 2v^2*e^-u)
# E'(v) = 2((u^2*e^v) - (2v^2*e^-u)) * (u^2*e^v - 2e^-u*2v)
#
# Implementación de las funciones de las derivadas
#
fduv1a = function(u,v){

  # - Con respecto a u:
  du = function(u,v){
```

```
    2*((u^2)*exp(v)-2*(v^2)*(exp(-u)))*(2*u*exp(v)+2*(v^2)*exp(-u))
  }

  # - Con respecto a v:
  dv = function(u,v){
    2*((u^2)*exp(v)-2*(v^2)*(exp(-u)))*((u^2)*exp(v)-2*exp(-u)*2*v)
  }

  c(du(u,v), dv(u,v))

}

####################
# Función Gradiente Descendente
# f ->       función original
# fduv ->    derivada de f
# w ->       valores iniciales de x,y o de u,v. Por defecto, (1,1)
# mu ->      tasa de aprendizaje
# tol ->     valor en el que debe parar el GD cuando f sea menor que él
# maxIter -> máximo número de iteraciones que el algoritmo tiene permitido
#            realizar. Por defecto, es el mayor entero que acepta la máquina.
# Se debe guardar todos los puntos que va generando para utilizarlo en el
# apartado 1b
#
GD = function(f, fduv, w=c(1,1), mu, tol, maxIter=.Machine$integer.max){

  # Se guarda el vector para hacer los cálculos intermedios. Se le suma
  # el máximo para que la primera vez entre en el ciclo
  wIni = w+.Machine$integer.max

  # Contador de iteraciones
  i = 0

  # Booleano para saber si debe seguir ejecutando el bucle while. Será FALSE
  # cuando el nuevo vector de pesos no mejore o no empeore más de un umbral
  # dado
  continue = TRUE

  # Como se deben devolver los pesos generados, se crea un vector donde estarán
  # todos inicializado vacío
  wGlobal = matrix(nrow = 0, ncol = 2)

  # Mientras que pueda seguir realizando iteraciones y no haya llegado al
  # umbral
  while(i<maxIter && continue){

    # Se calcula el valor de la función E con ambos pesos
    EIni = f(wIni[1], wIni[2])
    E = f(w[1], w[2])

    # Si los pesos cumplen la condición de que f(u,v) es menor que tol,
    # termina de calcular
    if(abs(EIni-E) < tol){
```

```
      continue = FALSE

    } else{

      wIni = w

      # Se guardan los pesos actuales
      wGlobal = rbind(wGlobal, c(w[1], w[2]))

      # Se calcula el gradiente
      g = fduv(w[1], w[2])

      # Se calculan los nuevos pesos.
      # Se resta por el paso 4 de la página 28 de la sesión 5.
      w[1] = w[1] - mu*g[1]
      w[2] = w[2] - mu*g[2]

      # Se aumenta el contador de iteraciones
      i = i+1

    }

  }

  # Se devuelven aquí para no condicionar al resto con el formato de uno
  list(data = wGlobal, ini1 = wGlobal[1,1], ini2 = wGlobal[1,2], f = E, u = w[1], v= w[2], i = i)

}
```

Punto 2

Se utiliza 10^{-4} ya que el tiempo empleado en obtener un valor menor que 10^{-14} es muy elevado.

```
# Ejecución del ejercicio 1a
res = GD(f1a, fduv1a, c(1,1), 0.1, 10^-4)

print(paste("El número de iteraciones que tarda en obtener un valor inferior a 10^{-4} es: ", res$i))
```

```
## [1] "El número de iteraciones que tarda en obtener un valor inferior a 10^{-4} es:  4"
```

Punto 3

```
print(paste("Los valores obtenidos son: u=",
            round(res$u, digits = 5), ", v=", round(res$v, digits = 5),
            " con f = ", f1a(res$u, res$v)))
```

```
## [1] "Los valores obtenidos son: u= 9.86457 , v= -24.43828  con f =  0.00385551786688815"
```

```
# Con tolerancia 10^-10
#res = GD(f1a, fduv1a, c(1,1), 0.1, 10^-10)

#print(paste("El número de iteraciones que tarda en obtener un valor inferior a 10^-10 es: ", res$i))

#print(paste("Los valores obtenidos son: u=",
```

```
            #round(res$u, digits = 5), ", v=", round(res$v, digits = 5),
            #" con f = ", f1a(res$u, res$v)))

# Con tolerancia 10^-14
#res = GD(f1a, fduv1a, c(1,1), 0.1, 10^-14)

#print(paste("El número de iteraciones que tarda en obtener un valor inferior a 10^-14 es: ", res$i))

#print(paste("Los valores obtenidos son: u=",
            #round(res$u, digits = 5), ", v=", round(res$v, digits = 5),
            #" con f = ", f1a(res$u, res$v)))
```

Conclusión:

Tarda mucho en realizar la operación con una tolerancia de 10^{-14}, por lo que se puede comprobar que con 10^{-4} solo tarda 4 iteraciones. El hecho de utilizar tan pocas iteraciones afecta directamente al resultado final, ya que no llega a ni el propio mínimo local y no se tiene el resultado real, con $u = 9.86457$ y $v = -24.43828$, siendo el resultado de evaluar la función $f(u, v) = 0.00385551786688815$

Ajustando una tolerancia de 10^{-10} tarda bastante más en hacer la operación, pero se puede apreciar como si ajusta mejor los valores finales con $u = 12.59608$ y $v = -24.2137$, siendo el resultado de evaluar la función $f(u, v) = 0.00005757596460248$, aunque necesita 156939 iteraciones.

Se puede apreciar como el mínimo encontrado con 10^{-10} es bastante inferior al encontrado con 10^{-4}.

Como no se sabe realmente cuando se está en el mínimo local y la recta se puede hacer infinita, hay que encontrar un equilibrio entre el número de iteraciones que se realiza y la diferencia de los valores en cada iteración.

Apartado b

```
####################
# Función 1b
#
f1b = function(x,y){
  (x-2)^2 + 2*(y-2)^2 + 2*sin(2*pi*x)*sin(2*pi*y)
}
```

Punto 1

Al igual que se ha hecho en el *apartado a*, se debe realizar la derivada parcial de

$f(x, y) = (x - 2)^2 + 2(y - 2)^2 + 2sin(2pix)sin(2piy)$

El resultado es:

$d'(x) = 2(x - 2) + 4picos(2pix)sin(2piy)$

$d'(y) = 4(y - 2) + 4picos(2piy)sin(2pix)$

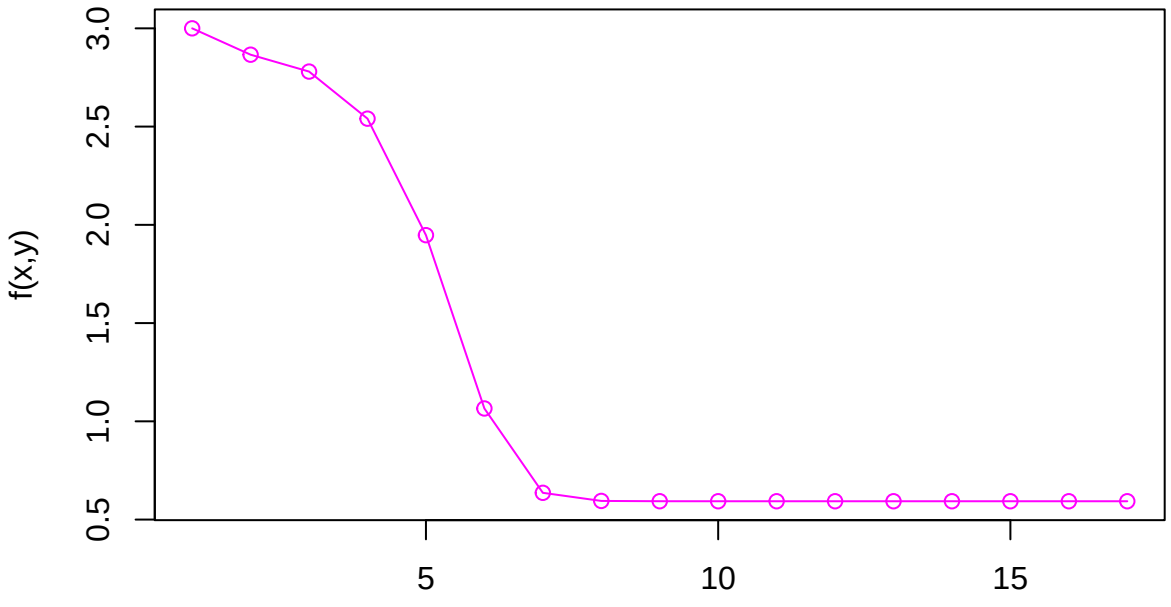
```
####################
# Se realiza la derivada parcial de la expresión:
# f(x,y) = (x-2)^2 + 2*(y-2)^2 + 2*sin(2*pi*x)*sin(2*pi*y)
#
# d'(x) = 2(x-2) + 4*pi*cos(2*pi*x)*sin(2*pi*y)
# d'(y) = 4(y-2) + 4*pi*cos(2*pi*y)*sin(2*pi*x)
#
```

```r
# Implementación de las funciones de las derivadas
#
fdxy1b = function(x,y){

  # - Con respecto a x:
  dx = function(x,y){
    2*(x-2)+ 4*pi*cos(2*pi*x)*sin(2*pi*y)
  }

  # - Con respecto a y:
  dy = function(x,y){
    4*(y-2) + 4*pi*cos(2*pi*y)*sin(2*pi*x)
  }

  c(dx(x,y), dy(x,y))

}

# Apartado 1.b.1
```

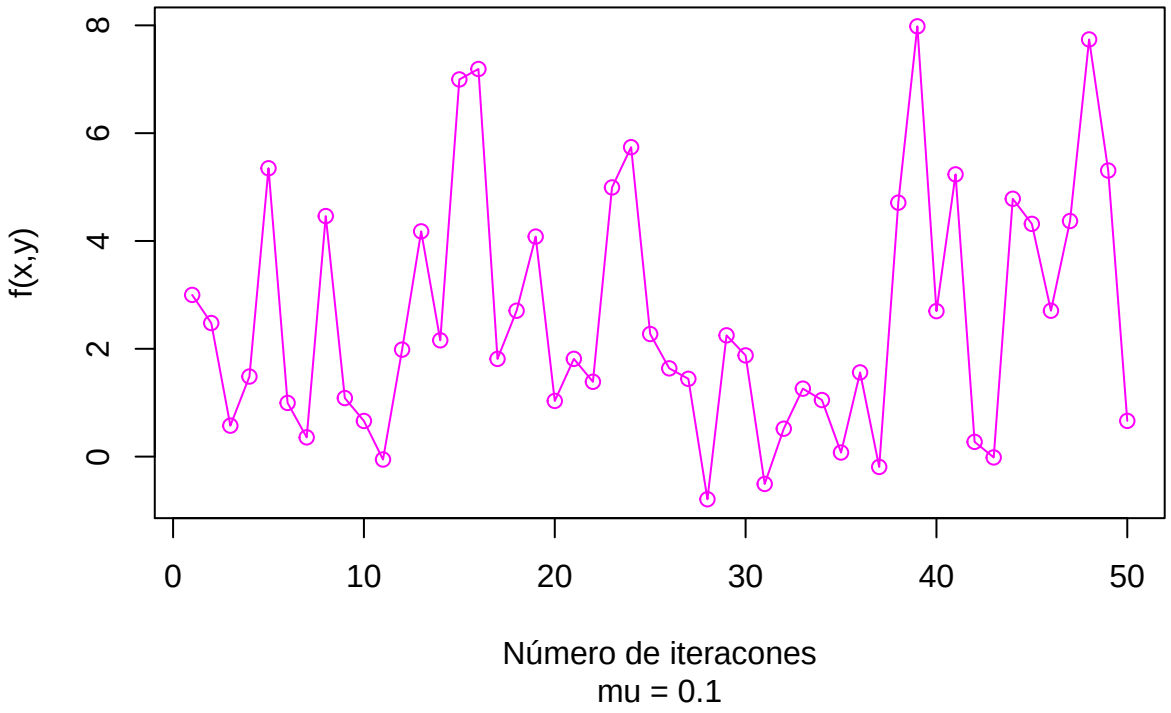
Las gráficas que muestran el descenso del valor de la función son:

```r
# Ejecución del ejercicio 1b1 con mu=0.01
res = GD(f1b, fdxy1b, c(1,1), 0.01, 10^-14, 50)

# Se guardan los datos por los que ha ido pasando
data = res$data

plot(f1b(data[,1], data[,2]),
     type="o",
     main="Ejercicio 1.b.1",
     sub="mu = 0.01",
     col=6,
     xlab="Número de iteracones",
     ylab="f(x,y)")
```

Ejercicio 1.b.1




```
# Ejecución del ejercicio 1b1 con mu=0.1
res = GD(f1b, fdxy1b, c(1,1), 0.1, 10^-14, 50)

# Se guardan los datos por los que ha ido pasando
data = res$data

plot(f1b(data[,1], data[,2]),
     type="o",
     main="Ejercicio 1.b.1",
     sub="mu = 0.1",
     col=6,
     xlab="Número de iteracones",
     ylab="f(x,y)")
```



Conclusiones:

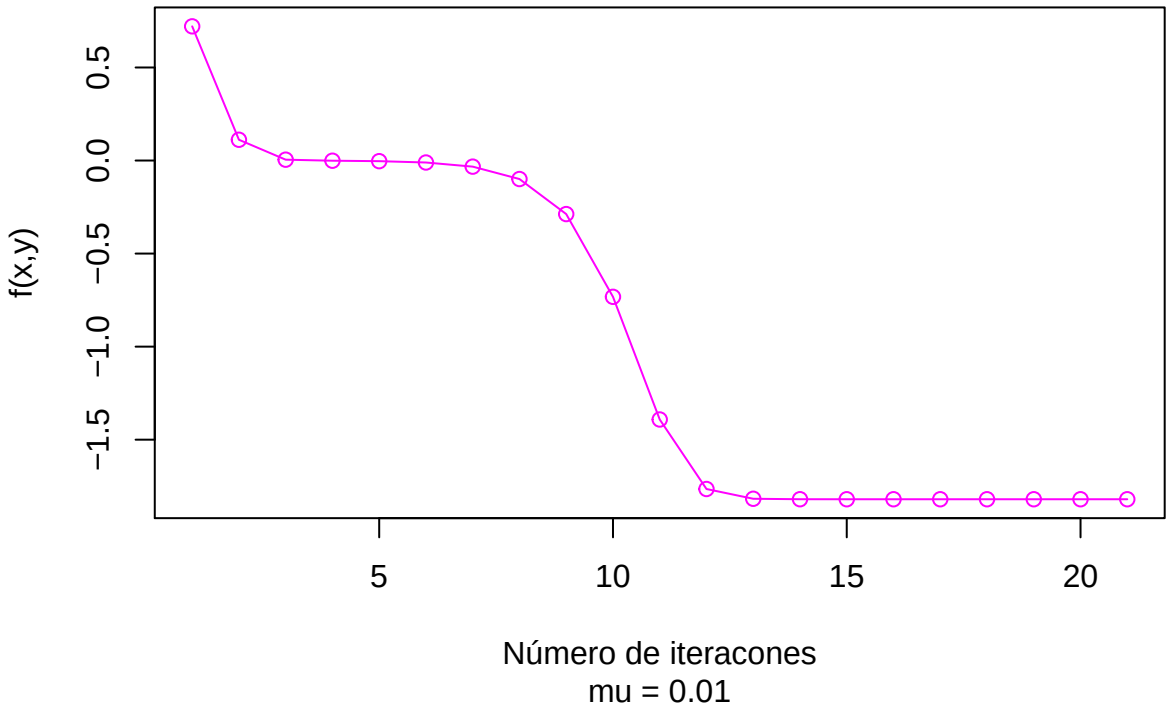
Con la tasa de aprendizaje igual a 0.01, se llega a un mínimo local muy rápido y ya no es capaz de salir de él, debido a que esta tasa de aprendizaje es pequeña.

Con la tasa de aprendizaje igual a 0.1, se obtienen muchos saltos porque es demasiado grande y sale de esos mínimos locales

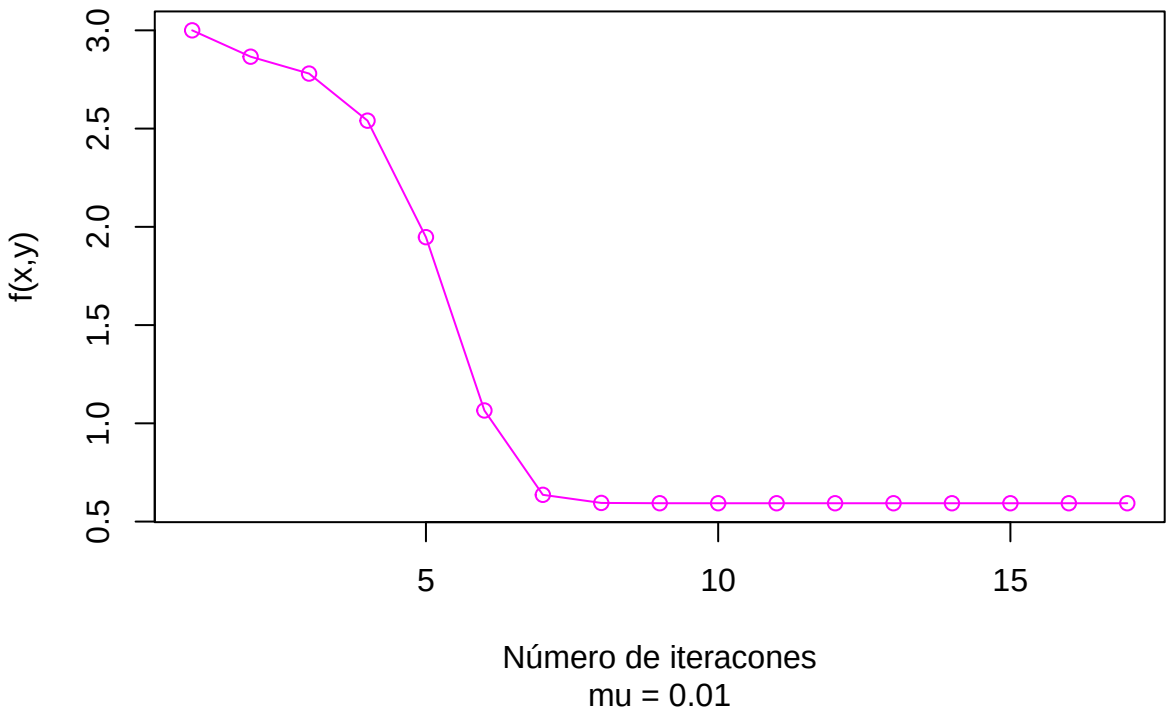
Punto 2

Se muestran las gráficas de cómo actúa el *Gradiente Descendente* con los cuatro puntos de inicio indicados:

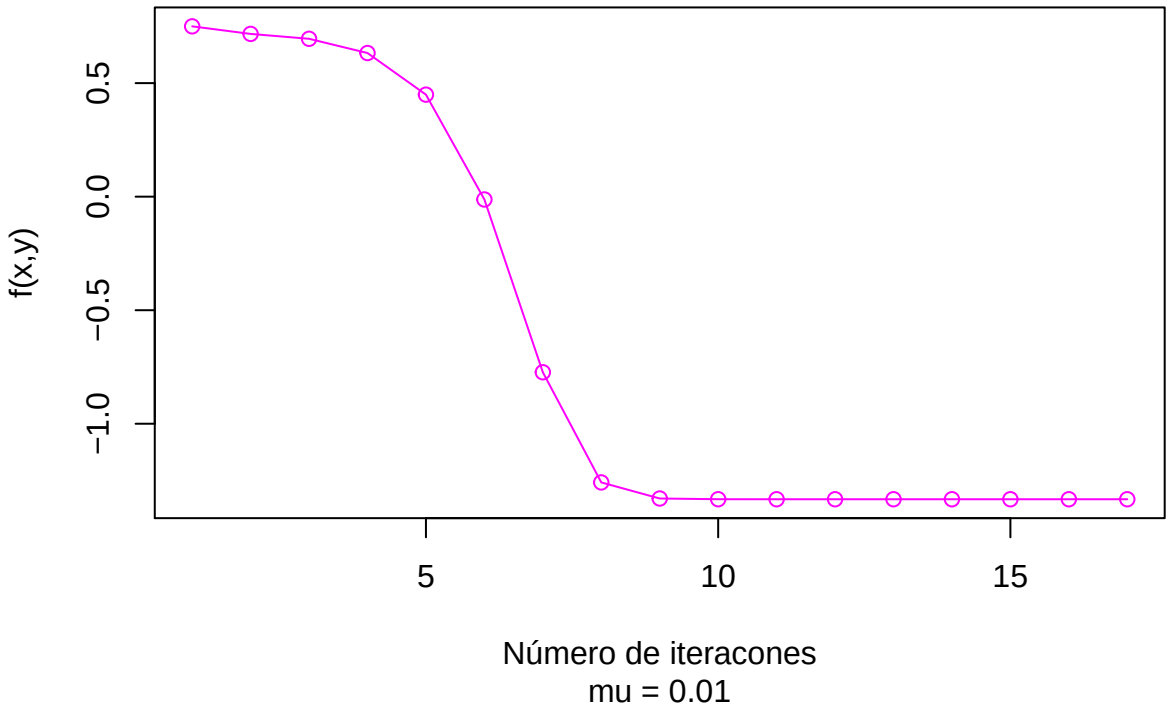
```
# Se irá guardando en res1b2
res1b2 = matrix(ncol = 7, nrow = 0)

# Punto de inicio (2.1, 2.1)
gd1b2 = GD(f1b, fdxy1b, c(2.1,2.1), 0.01, 10^-14, 50)

plot(f1b(gd1b2$data[,1], gd1b2$data[,2]),
     type="o",
     main="Ejercicio 1.b.2 - Punto de inicio (2.1, 2.1)",
     sub="mu = 0.01",
     col=6,
     xlab="Número de iteracones",
     ylab="f(x,y)")
```



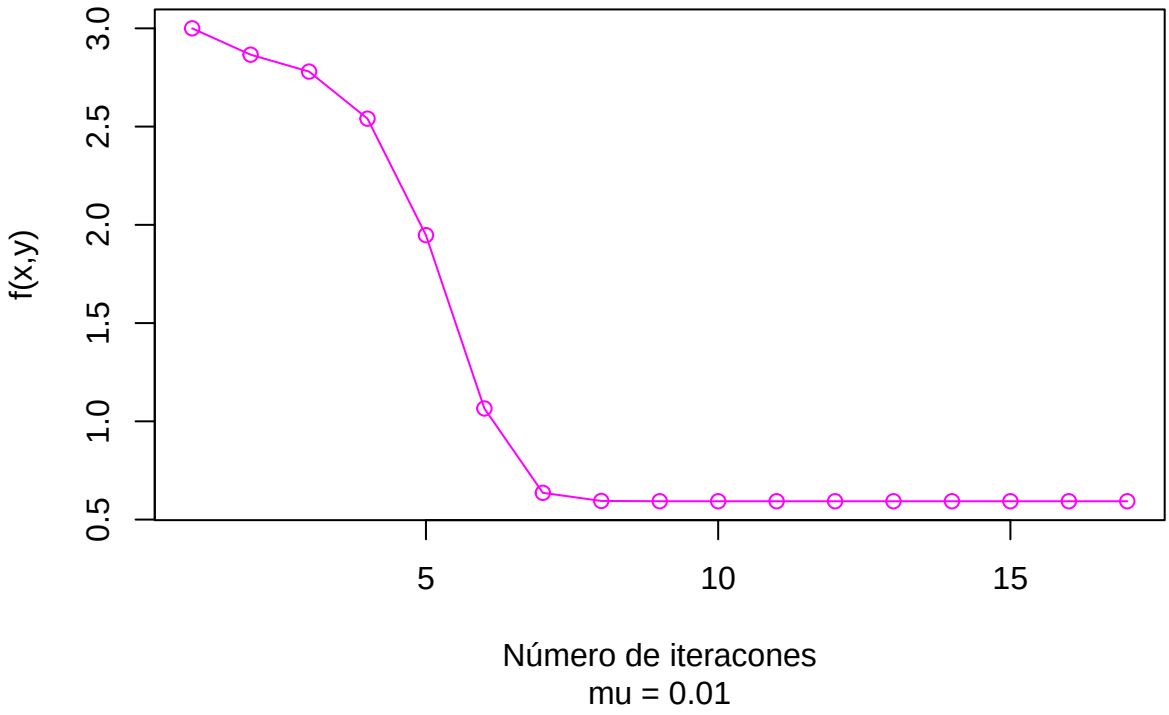

```r
# Punto de inicio (2.1, 2.1)
res1b2 = rbind(res1b2, gd1b2)

# Punto de inicio (3, 3)
gd1b2 = GD(f1b, fdxy1b, c(3,3), 0.01, 10^-14, 50)

plot(f1b(gd1b2$data[,1], gd1b2$data[,2]),
     type="o",
     main="Ejercicio 1.b.2 - Punto de inicio (3, 3)",
     sub="mu = 0.01",
     col=6,
     xlab="Número de iteracones",
     ylab="f(x,y)")
```




```r
# Punto de inicio (3, 3)
res1b2 = rbind(res1b2, gd1b2)

# Punto de inicio (1.5, 1.5)
gd1b2 = GD(f1b, fdxy1b, c(1.5, 1.5), 0.01, 10^-14, 50)

plot(f1b(gd1b2$data[,1], gd1b2$data[,2]),
     type="o",
     main="Ejercicio 1.b.2 - Punto de inicio (1.5, 1.5)",
     sub="mu = 0.01",
     col=6,
     xlab="Número de iteracones",
     ylab="f(x,y)")
```




```
# Punto de inicio (1.5, 1.5)
res1b2 = rbind(res1b2, gd1b2)

# Punto de inicio (1, 1)
gd1b2 = GD(f1b, fdxy1b, c(1,1), 0.01, 10^-14, 50)

plot(f1b(gd1b2$data[,1], gd1b2$data[,2]),
     type="o",
     main="Ejercicio 1.b.2 - Punto de inicio (1, 1)",
     sub="mu = 0.01",
     col=6,
     xlab="Número de iteracones",
     ylab="f(x,y)")
```




```r
# Punto de inicio (1, 1)
res1b2 = rbind(res1b2, gd1b2)

# Se elimina la primera columna, que es la que contiene los pesos que ha
# ido generando en todo el proceso
res1b2 = res1b2[,-1]

dimnames(res1b2) = list(c(), c("Inicio 1", "Inicio 2", "f(x,y)", "x", "y", "Iteraciones"))

# Se muestra la tabla
print(res1b2)
```

```
##      Inicio 1 Inicio 2 f(x,y)    x         y        Iteraciones
## [1,] 2.1      2.1      -1.820079 2.243805  1.762074 21
## [2,] 3        3        0.5932694 3.21807   2.712812 17
## [3,] 1.5      1.5      -1.332481 1.268623  1.762145 17
## [4,] 1        1        0.5932694 0.7819297 1.287188 17
```

Conclusión:

Encontrar un mínimo global depende del tipo de función que se esté utilizando y depende también del punto de partida elegido.

Realmente no se sabe si se encuentra el mínimo global, incluso ni el local (como ha ocurrido cuando el punto de inicio es (2.1, 2.1).

El algoritmo se queda con facilidad en un punto en el que la función varía muy poco, por lo que es complicado encontrar el local o global, ya que no se puede saber si tras x iteraciones, la función va a encontrar nuevos mínimos.

Cambiando la tasa de aprendizaje se podría salir de esta zona para explorar otras y encontrar el local o

global, pero esto puede provocar que también se salga del global en el caso hipotético de haberlo encontrado.

Todo esto hace difícil ajustar bien los resultados que se quieren obtener, por lo que esta técnica, pienso, no sería suficiente para conseguir los datos que se desean aunque si es una buena aproximación en un tiempo aceptable.

Ejercicio 2. Regresión Logística.

Para crear los datos que se van a utilizar en este ejercicio se usa la función *simula_unif()* de la práctica 1. Tendrá dimensión 2, la cantidad de puntos aleatorios será 100 y el rango será (0,2):

```
####################
# Función simula_unif()
# Genera una matriz de "N" puntos y dimensión "dim" con valores comprendidos en
# "rango"
# Por defecto, N = 100, dim = 2, rango = [0,1]
#
simula_unif = function(N=100, dim=2, rango=c(0,1)){

  m = matrix(runif(N*dim, min=rango[1], max=rango[2]),
             nrow=N, ncol=dim, byrow=T)

  m

}
```

También es necesario crear una recta que pase por los puntos (-1,1) y se usa la función *simula_recta()*, también de la práctica 1:

```
####################
# Función simula_recta()
# Devuelve la pendiente y el punto de corte de una recta dando un intervalo "c"
#
simula_recta = function(intervalo=c(-1,1), visible=F){

  # Se generan 2 puntos
  puntos = simula_unif(2, 2, intervalo)

  # Se calcula la pendiente
  a = (puntos[1,2] - puntos[2,2]) / (puntos[1,1] - puntos[2,1])

  # Se calcula el punto de corte
  b = puntos[1,2] - a*puntos[1,1]

  # Si se ha recibido que sea visible, pinta la recta y los 2 puntos
  if (visible) {

    # Si no está abierta la gráfica, se dibuja con plot
    if (dev.cur()==1)
      plot(1, type="n", xlim=intervalo, ylim=intervalo)

    # Pinta de color verde los puntos y la recta
    points(puntos,col=3)
    abline(b,a,col=3)
```

```
  }

  # Se devuelve el par "pendiente" y el "punto de corte"
  c(a,b)

}
```

Apartado a

Para hacer la Regresión Logística más sencilla, se van a definir las funciones intermedias que son necesarias para realizarla.

```
####################
# Calcula la Norma de la diferencia entre dos vectores
#
norm = function(w1, w2){
  sqrt(sum((w1-w2)^2))
}
```

También se hace uso de la función de la práctica 1 que clasifica los puntos en una recta

```
####################
# getValueF2a() -> "get" el "value" de la "F" definida en el apartado 2a
# Función que devuelve el resultado de f(x,y), dependiendo de la entrada
# "punto" con respecto a la recta "recta".
# f(x,y) = y - ax - b
#
getValueF = function(punto, recta){

  # Si falta algún parámetro se termina la función
  if(missing(punto) | missing(recta))
    stop("Faltan algunos parámetros")

  # f(x,y) = y - ax - b
  punto[2] - recta[1]*punto[1] - recta[2]

}
```

Se debe implementar la Regresión Logística (LR) con Gradiente Descendente Estocástico (SGD) con las siguientes condiciones:

- Vector de pesos inicializado a 0.
- $||wOld - w|| < 0.01$ entre el inicio y fin de una época.
- Aplicar una permutación aleatoria de los datos antes de cada época.
- Tasa de aprendizaje = 0.01.

```
####################
# Función Regresión Logística con Gradiente Descendente Estocástico
#
# data ->    muestra de datos
# label ->   etiquetas de los datos (-1, +1)
# w ->       vector de pesos
# maxIter -> épocas máximas que puede crear
# mu ->      tasa de aprendizaje
```

```
# tol ->     tolerancia al comprobar la normal de la diferencia de pesos
#
# Referencias: Página 28, Sesión 5
#
LR_SGD = function(data, label, w, maxIter, mu, tol){

  # Se inicializa el contador de iteraciones
  i = 0

  # En cada iteración hay que comparar la norma de la diferencia de los
  # vectores nuevos y antiguos con la tolerancia recibida.
  # Si la primera vez los dos vectores son iguales, la norma será 0, por lo que
  # no entrará nunca en el while. Para corregir este problema se suma 999
  # al vector de pesos "antiguo".
  wOld = w+999

  # El algoritmo se estará ejecutando siempre que no se supere un número de
  # iteraciones y la diferencia entre el vector al inicio de la época y al
  # final de la época sea menor que 0.01.
  # i es el contador de las iteraciones.
  while(i<maxIter && tol<norm(wOld, w)){

    # Se va a calcular el nuevo vector de pesos, por lo que se guarda el
    # que se tiene al inicio de la época
    wOld = w

    # Al inicio de cada época se debe realizar una permutación de los datos,
    # por lo que se obtienen las posiciones aleatorias de éstos
    randomPos = sample(nrow(data), nrow(data))

    # Para evitar tener que pasar una época completa para actualizar una vez
    # los pesos, se va calculando con cada dato aleatorio de la muestra y
    # actualizando al momento
    for(i in randomPos){

      g = -(label[i]*data[i,])/(1+exp(label[i]*t(w)%*%data[i,]))
      w = w - mu*as.vector(g)

    }

    # Se aumenta el contador de las iteraciones
    i = i+1

  }

  return(list(w=w, i=i))

}

####################
# Calcula el error en una muestra, teniendo en cuenta dicha muestra, las
# etiquetas y un vector de pesos
#
```

```
# Referencias: Página 28, Sesión 5
#
eLR = function(data, label, w){

  total = 0

  for(i in 1:nrow(data)){
    total = total + log(1+exp(sign(-label[i]*t(w)%*%data[i,])))
  }

  total / nrow(data)

}

# Se generan los datos aleatorios como especifica el ejercicio.
# Al igual que ocurría en la Regresión Lineal de la práctica 1, la matriz
# debe coincidir con el vector de pesos, por lo que se le concatena
# una columna con 1
data = cbind(simula_unif(100, 2, c(0, 2)), 1)

# Se crea la recta. Por defecto, los parámetros generan una recta de -1,1,
# por lo que no se le pasa ningún vector
line = simula_recta()

# Se crean las etiquetas de la muestra teniendo en cuenta la recta
valuesF = apply(data, 1, getValueF, line)

# Lo que interesa es tener etiquetado cada punto, para eso se crea
# un vector con -1, 1, teniendo en cuenta el signo de cada punto.
label = sign(valuesF)

# El vector de pesos debe estar inicializado a 0
w = c(0,0,0)

# Se hace la Regresión Logística con 1000 iteraciones y la tolerancia
# especificada de 0.01
res = LR_SGD(data, label, w, 1000, 0.01, 0.01)

w = res$w
```

Apartado b

```
####################
# Calcula un error eligiendo una muestra aleatoria
#
getEOutData = function(N, dim, rango, w){

  dataTest = cbind(simula_unif(N, dim, rango), 1)

  # Se crea la recta. Por defecto, los parámetros generan una recta de -1,1,
  # por lo que no se le pasa ningún vector
  #lineTest = simula_recta()
```

```
  # Se crean las etiquetas de la muestra teniendo en cuenta la recta
  valuesFTest = apply(dataTest, 1, getValueF, line)

  # Lo que interesa es tener etiquetado cada punto, para eso se crea
  # un vector con -1, 1, teniendo en cuenta el signo de cada punto.
  labelTest = sign(valuesFTest)

  eLR(dataTest, labelTest, w)

}

x <- replicate(1000, {
  getEOutData(100, 2, c(0,2), w)
})

print(paste("La media de error de las 1.000 muestras es: ",
            round(mean(x)*100, digits = 3), "%"))
```

```
## [1] "La media de error de las 1.000 muestras es:  34.415 %"
```

Conclusión:

Se ha utilizado Regresión Logística con Gradiente Descendente Estocástico para aproximar una solución. Esta aproximación, como es lógico ha sido mejor para los datos de entrenamiento, pero para los datos de test se ha obtenido un error elevado.

A medida que se consiga hacer más iteraciones, el error debería mejorar ya que los pesos se ajustarían más a los datos.

Ejercicio 3

Para leer los dígitos 4 y 8 se hace uso del código implementado en la práctica 1

```
########################################
# Función getMeanSym()
# Función que dada una matriz devuelve su valor medio y el
# grado de simetría vertical que tiene
#
# Referencias:
# http://stackoverflow.com/questions/9135799/how-to-reverse-a-matrix-in-r
#
getMeanSym = function(matrix){

  # Se calcula la media de la matriz
  originalMatrixMean = mean(matrix)

  ## Cálculo de la simetría vertical

  # a: se le da la vuelta a las columnas
  invertedMatrix = matrix[, ncol(matrix):1]

  # b: se calcula la diferencia entre la matriz original y la invertida
  matrix = matrix - invertedMatrix
```

```
  # c: se calcula la media global de los valores absolutos de la matriz
  invertedMatrixMean = mean(abs(matrix))

  c(originalMatrixMean, invertedMatrixMean)

}

#######################################
# Función errorData()
# Calcula el error de una muestra de datos
errorData = function(data, label, w){

  # Se comienza sin ningún error
  error = 0

  # Se recorre todo el vector de datos
  for(i in 1:nrow(data)){

    # Si el signo que obtiene de multiplicar matricialmente el punto actual
    # con el vector de pesos, no coincide con la etiqueta que le corresponde,
    # se aumenta en 1 el error
    if(sign(data[i,]%*%w) != sign(label[i])){
      error = error + 1
    }

  }

  # Se calcula el porcentaje [0,1]
  error/nrow(data)

}

#######################################
# Función Regress_Lin(datos, label)
#
# Referencias:
# - página 14-16, sesión 3.pdf
# - http://stackoverflow.com/questions/36234136/apply-svd-linear-regression-in-r
Regress_Lin = function(datos, label){

  # Primero se consigue SVD
  udv = svd(datos)

  # Se calcula su inversa. (xT*x)^-1 = v*d*d*vT
  # Como svd() devuelve $d, $u, $v, si se quiere acceder a "v", se hace con udv$v
  # v -> v
  # d -> diagonal d
  # uT -> traspuesta u
  datos.inverse = udv$v %*% (diag(1/udv$d))^2 %*% t(udv$v)

  # La pseudoinversa se calcula multiplicando la inversa calculada con la
  # traspuesta de la matriz original: (xT*X)^-1 * xT
  datos.pseudoinverse = datos.inverse %*% t(datos)
```

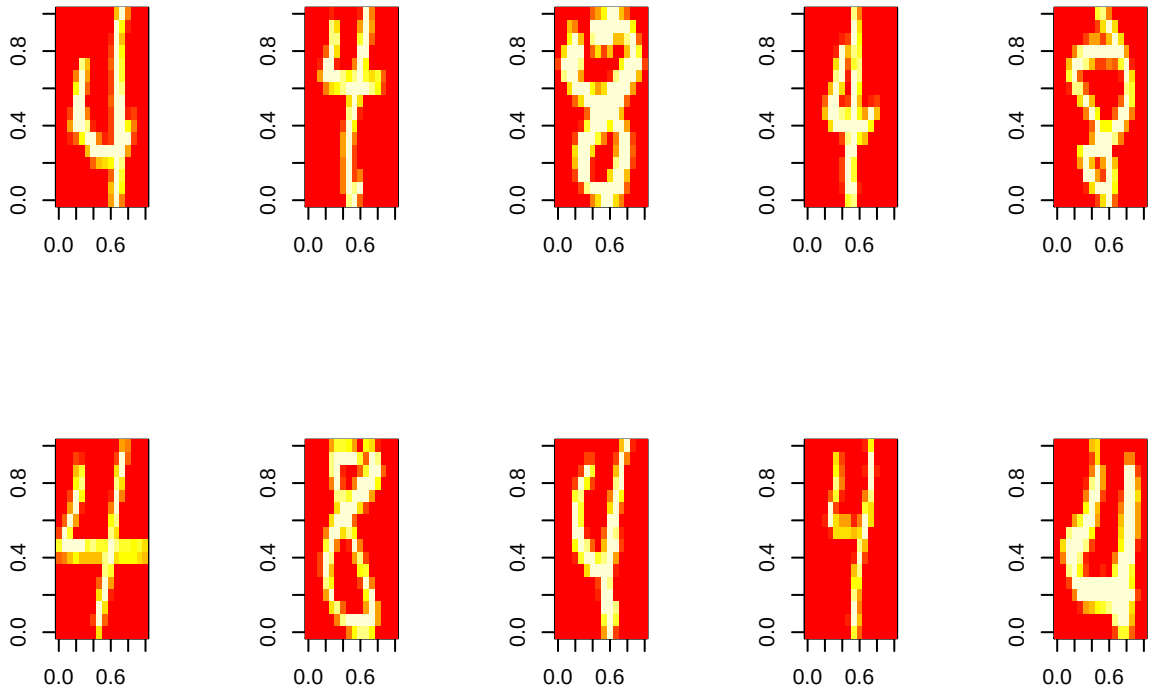
```
  # Se termina obteniendo los pesos, multiplicando la pseudoinversa con las
  # etiquetas que se recibe por parámetro
  w = datos.pseudoinverse %*% label

  w

  # Se ajusta la pendiente y el punto de corte con el umbral
  #c(-w[1]/w[2], -w[3]/w[2])

}

########################################
# Función ajusta_PLA_POCKET
# Calcula el hiperplano solución a un problema de clasificación binaria
# usando el algoritmo PLA mejorado haciéndolo pocket.
# La diferencia ahora es que se guarda el mejor resultado obtenido.
# Devuelve los pesos y el número de iteracionesque ha necesitado para
# conseguirlo
# Referencias:
# - Página 33. Sesión 1. AA.
# - http://www.cleveralgorithms.com/nature-inspired/neural/perceptron.html
# - http://r-econ.blogspot.com.es/2011/04/trabajando-con-matrices-en-r.html
ajusta_PLA_POCKET = function(datos, label, max_iter, vini = c(0, 0, 0)){

  # w es el vector de pesos. Comenzará con el valor establecido en vini
  w = vini

  # Bool que indica si ha encontrado el valor óptimo de w en todas las
  # iteraciones
  finded = F

  # Contador del bucle
  i = 1

  # Se debe guardar siempre el mejor vector de pesos para devolverlo
  bestW = w

  # A la vez que se guarda el mejor vector de pesos, se debe guardar cuantos
  # errores ha producido ese vector y así compararlo con los generados.
  # Se calcula el error generado con el vector de pesos inicial.
  bestError = errorData(cbind(datos, 1), label, w)

  # repeat
  # Se itera un número máximo de iteraciones o hasta que ya haya encontrado
  # el ajuste óptimo sin cambiar los pesos
  while(!finded & i<=max_iter){

    # Suponemos que va a encontrar el óptimo en esta iteración
    finded = T

    # for each xi € D do
    # Se recorren todas las filas de la matriz. Se puede coger el número de
    # filas de label o el número de filas de datos
```

```
  # No se sale del bucle cuando encuentre una etiqueta que no coincide, ya
  # que no se obtienen los resultados deseados y no mejora el error.
  for(j in 1:length(label)){

    # Se guarda en "punto" los valores x, y de la fila que se está tratando.
    # Como el vector de pesos tiene longitud 3, para hacer la multiplicación
    # del punto con el vector de pesos, "punto" debe tener longitud 3.
    # Se añade a "punto", 1 como tercer valor del vector.
    punto = append(datos[j,], 1)

    # Se multiplica de manera matricial (%*%) el punto con w
    res = punto%*%w

    # Interesa el signo. Si vini se ha recibido con c(0, 0, 0), el resultado
    # de la multiplicación es 0. En este caso, interesa que
    # la etiqueta sea 1.
    sign = ifelse(sign(res) >= 0, 1, -1)

    # if: sign(wT*xi) != yi
    # Si la etiqueta obtenida no coincide con la real, se debe ajustar el
    # perceptron
    if(sign != label[j]){

      # w_new = w_old + yi*xi
      # Se actualiza el vector w con los pesos para ajustarlos
      w = w + punto*label[j]

      # Si no ha encontrado un óptimo, se indica que debe seguir
      finded = F

      # Cada vez que cambia el vector de pesos se comprueba si esta
      # versión es mejor que la mejor actual
      auxError = errorData(cbind(datos, 1), label, w)

      # Se compara si es mejor y en el caso de serlo, se actualizan
      # los datos
      if(bestError > auxError){

        bestError = auxError
        bestW = w

      }

    }

  }

  # Se aumenta el contador
  i = i+1

}

# Hay que convertir el vector de pesos w en la pendiente y punto de corte
```

```
  # para dibujar la recta
  #c(-w[1]/w[2], -w[3]/w[2], i-1)

  # Para calcular el error se deben devolver los pesos calculados. En este
  # caso se devuelve el mejor vector de pesos encontrado.
  c(bestW, i-1)

}
```

Apartados a y b

```
# Se guarda en digit.train los datos que se leen del fichero de entrenamiento
digit.train <- read.table("zip.train",
                          quote="\"", comment.char="", stringsAsFactors=FALSE)

# En digitos48 se guardan los datos de entrenamiento de los números 4 y 8
digitos48.train = digit.train[digit.train$V1==4 | digit.train$V1==8,]

# En la primera columna de la matriz que tiene todos los datos de entrenamiento
# se indica el dígito al que corresponde. Se consigue un vector que indica en
# cada posición, el número correspondiente.
digitos = digitos48.train[,1]

# Se guarda el total de números 4 y 8 que hay
ndigitos = nrow(digitos48.train)

# Se retira la clase y se monta una matriz 3D: 599*16*16
grises = array(unlist(subset(digitos48.train, select=-V1)), c(ndigitos, 16, 16))

# Ya no es necesario tener las matrices leídas con los 4 y 8 y se eliminan
rm(digit.train)
rm(digitos48.train)

# Se guarda cuantas imágenes se quieren mostrar
numImageToShow = 10

# Se ajusta el plot para que muestre varios números a la vez
par(mfrow=c(2, 5))

# Se van a mostrar las "numImageToShow" primeras imágenes
for(i in 1:numImageToShow){

  # image necesita una matriz para mostrar la imagen
  # Se rota además, para mostrarla correctamente
  imagen = grises[i,,16:1]
  image(z=imagen)

}
```




```
# Etiquetas correspondientes a las "numImageToShow" primeras imágenes
digitos[1:numImageToShow]
```

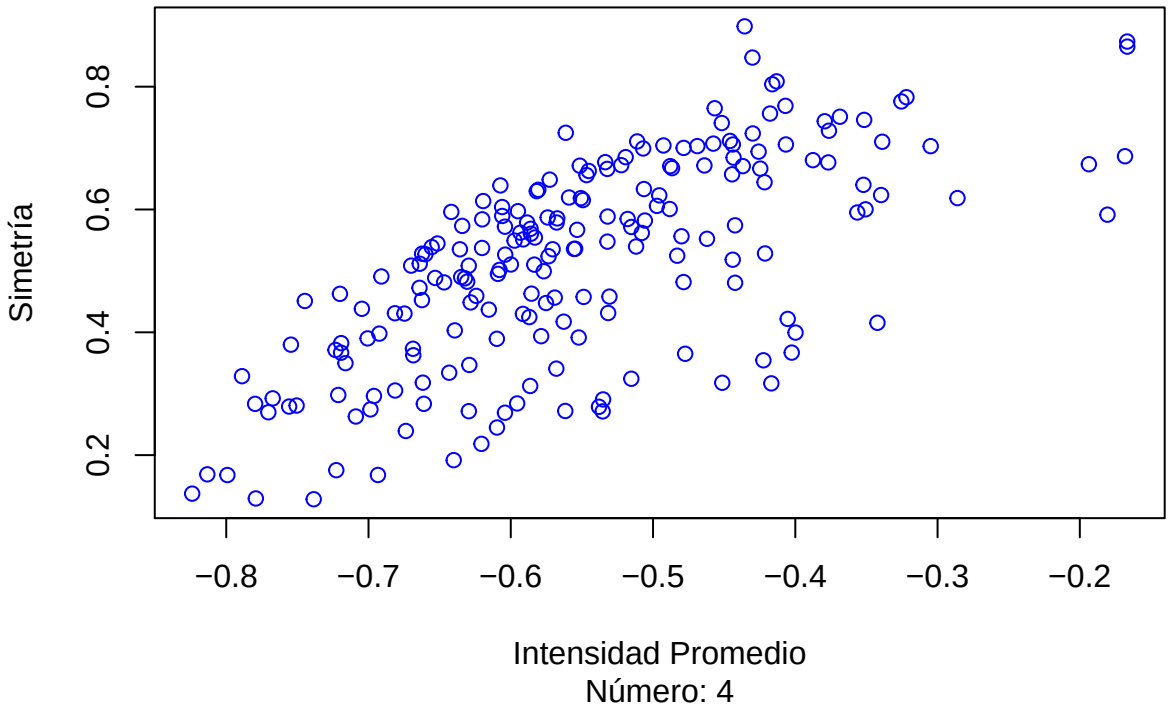
```
##  [1] 4 4 8 4 8 4 8 4 4 4
```

```
# Se mostrará una sola gráfica a la vez
par(mfrow=c(1, 1))

# Se crea una matriz con 2 columnas para la media y la simetría y tantas
# filas como números 4 y 8 haya
matrixMeanSym = matrix(ncol = 2, nrow = ndigitos)

# Se le da nombre a las columnas
dimnames(matrixMeanSym) = list(c(), c("mean", "symmetry"))

# Se recorre la matriz 3d donde están todos los datos y se va creando
# la matriz con los resultados de los cálculos.
# He intentado hacerlo con apply por filas, obteniendo la matriz de cada
# número pero no conseguía reflejar los datos como debía
for(i in 1:ndigitos){

  matrixMeanSym[i, ] = getMeanSym(grises[i, , ])

}
```

Secciones 1 y 2

Gráfica con los dígitos *4* de los datos de entrenamiento:

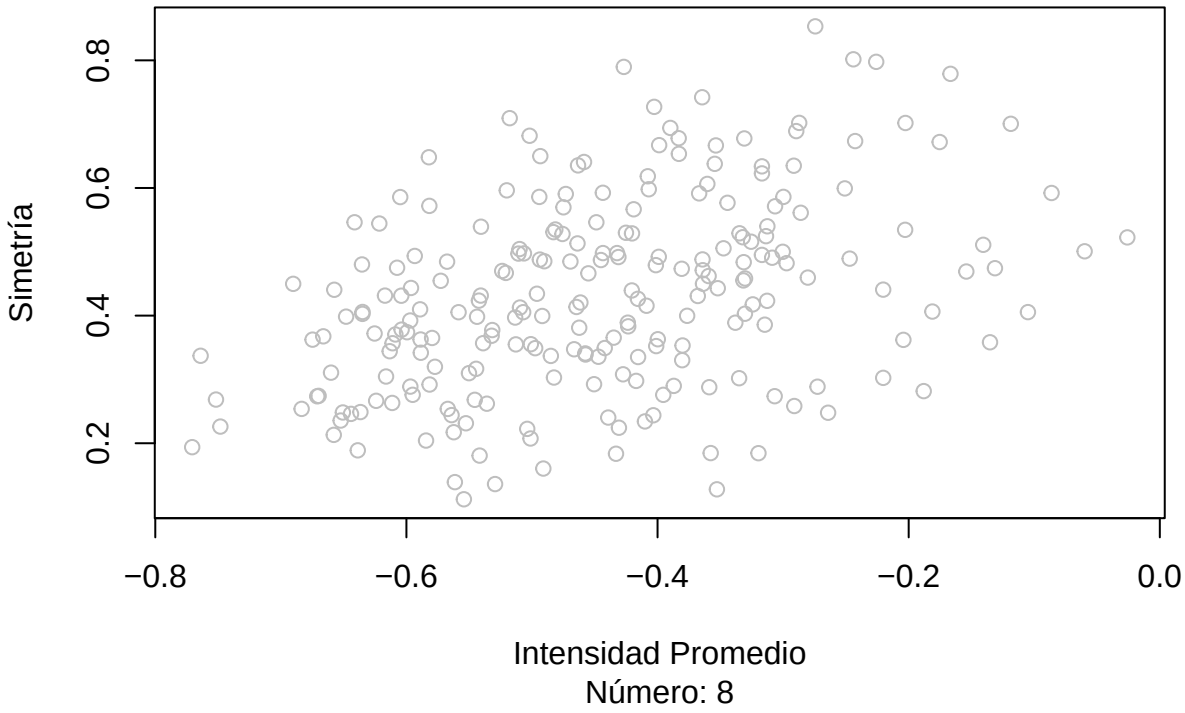
```
# Gráfica con los 4
plot(matrixMeanSym[which(digitos==4), ],
     xlab="Intensidad Promedio", ylab="Simetría", col=4,
     main="Ejercicio 3", sub="Número: 4")
```



Gráfica con los dígitos *8* de los datos de entrenamiento:

```
# Gráfica con los 8
plot(matrixMeanSym[which(digitos==8), ],
     xlab="Intensidad Promedio", ylab="Simetría", col=8,
     main="Ejercicio 3.", sub="Número: 8")
```

Ejercicio 3.



Se genera el gráfico con los datos de entrenamiento.

La línea rosa ha sido generada con Regresión Lineal.

La línea verde ha sido generada con el algoritmo PLA Pocket.

```
# Gráfica con los dos números a la vez
plot(matrixMeanSym, col=digitos, xlab="Intensidad Promedio",
     ylab="Simetría", main="Ejercicio 3",
     sub="Datos de entrenamiento")

# Se escribe en la gráfica a qué corresponde cada color
legend(x="topleft", legend=c("4","8"), col=c(4, 8), lwd=3)

# Se copian las etiquetas de los números para cambiar los 4 por -1 y los
# 8 por 1 y ajustarlas a Regress_lim con -1, 1
lD3 = digitos
lD3[which(lD3==4)] = -1
lD3[which(lD3==8)] = 1

# Se calcula la regresión lineal con la matriz de medias y simetrias
# verticales y con las etiquetas de -1, 1 que representa los digitos
wLR3 = Regress_Lin(cbind(matrixMeanSym, 1), lD3)

# Se dibuja la línea que ajustaría el error mínimo
abline(-wLR3[3]/wLR3[2], -wLR3[1]/wLR3[2], col=6)

# Número máximo de iteraciones que dará el algoritmo PLA_POCKET
max_iter = 20
```

```r
# Se guarda el error antes de calcular los pesos con PLA_POCKET
EinPLA_POCKET = errorData(cbind(matrixMeanSym, 1), lD3, wLR3)

print(paste("Ein calculado con Regresión Lineal: ",
            round(EinPLA_POCKET*100, digits = 3), "%"))
```

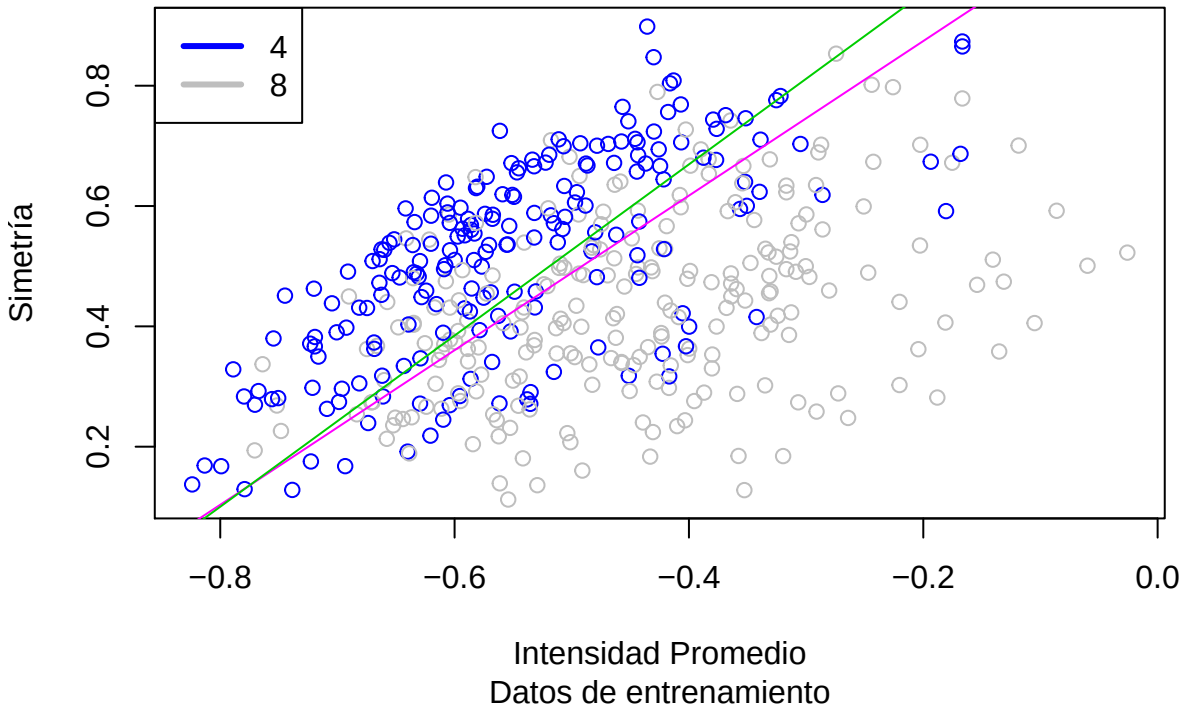
```
## [1] "Ein calculado con Regresión Lineal:  24.769 %"
```

```r
pla = ajusta_PLA_POCKET(matrixMeanSym, lD3, max_iter, wLR3)

# Se guarda el error después de calcular los pesos con PLA_POCKET
EinPLA_POCKET = errorData(cbind(matrixMeanSym, 1), lD3, pla[1:3])

print(paste("Ein calculado con PLA_POCKET: ",
            round(EinPLA_POCKET*100, digits = 3), "%"))
```

```
## [1] "Ein calculado con PLA_POCKET:  22.222 %"
```

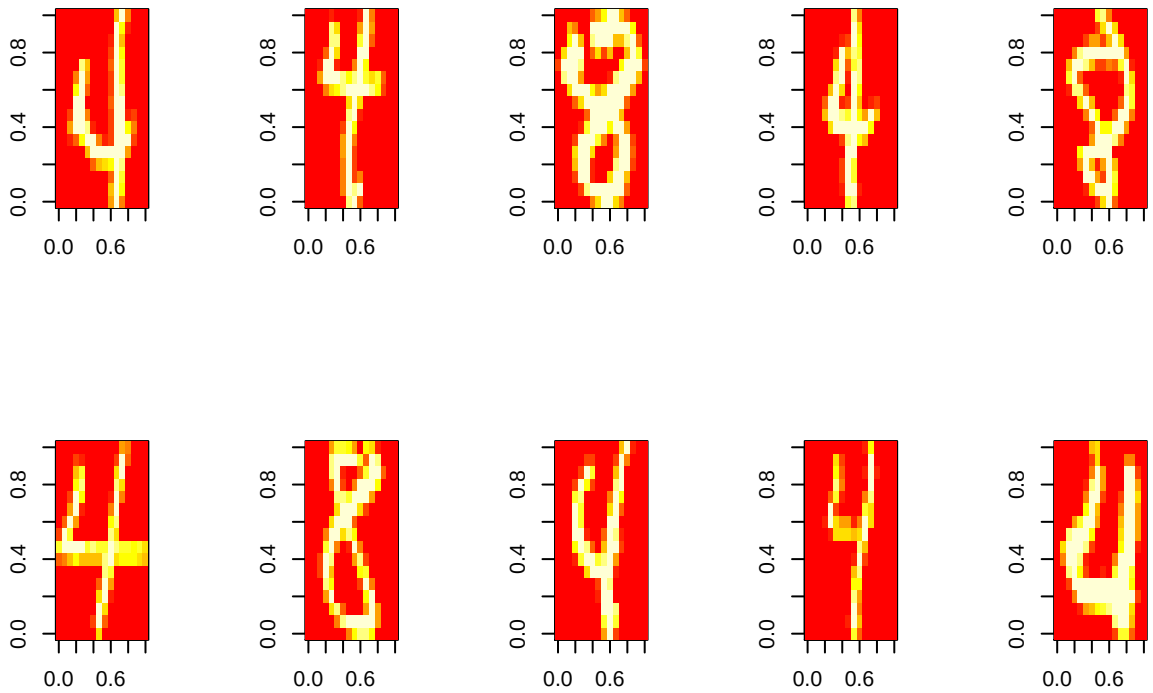
```r
# Se dibuja la recta generada con el algoritmo PLA_POCKET
abline(-pla[3]/pla[2], -pla[1]/pla[2], col=3)
```

Ejercicio 3




```r
#########
# Se repite el proceso para los datos de test
#########

# Se guarda en digit.test los datos que se leen del fichero de test
digit.test <- read.table("zip.test",
                         quote="\"", comment.char="", stringsAsFactors=FALSE)

# En digitos48se guardan los datos de test de los números 4 y 8
```

```
digitos48.test = digit.test[digit.test$V1==4 | digit.test$V1==8,]

# En la primera columna de la matriz que tiene todos los datos de test
# se indica el dígito al que corresponde. Se consigue un vector que indica en
# cada posición, el número correspondiente.
digitosTest = digitos48.test[,1]

# Se guarda el total de números 4 y 8 que hay
ndigitosTest = nrow(digitos48.test)

# Se retira la clase y se monta una matriz 3D: 599*16*16
grisesTest = array(unlist(subset(digitos48.test, select=-V1)), c(ndigitosTest, 16, 16))

# Ya no es necesario tener las matrices leídas con los 4 y 8 y se eliminan
rm(digit.test)
rm(digitos48.test)

# Se ajusta el plot para que muestre varios números a la vez
par(mfrow=c(2, 5))

# Se van a mostrar las "numImageToShow" primeras imágenes
for(i in 1:numImageToShow){

  # image necesita una matriz para mostrar la imagen
  # Se rota además, para mostrarla correctamente
  imagen = grises[i,,16:1]
  image(z=imagen)

}
```



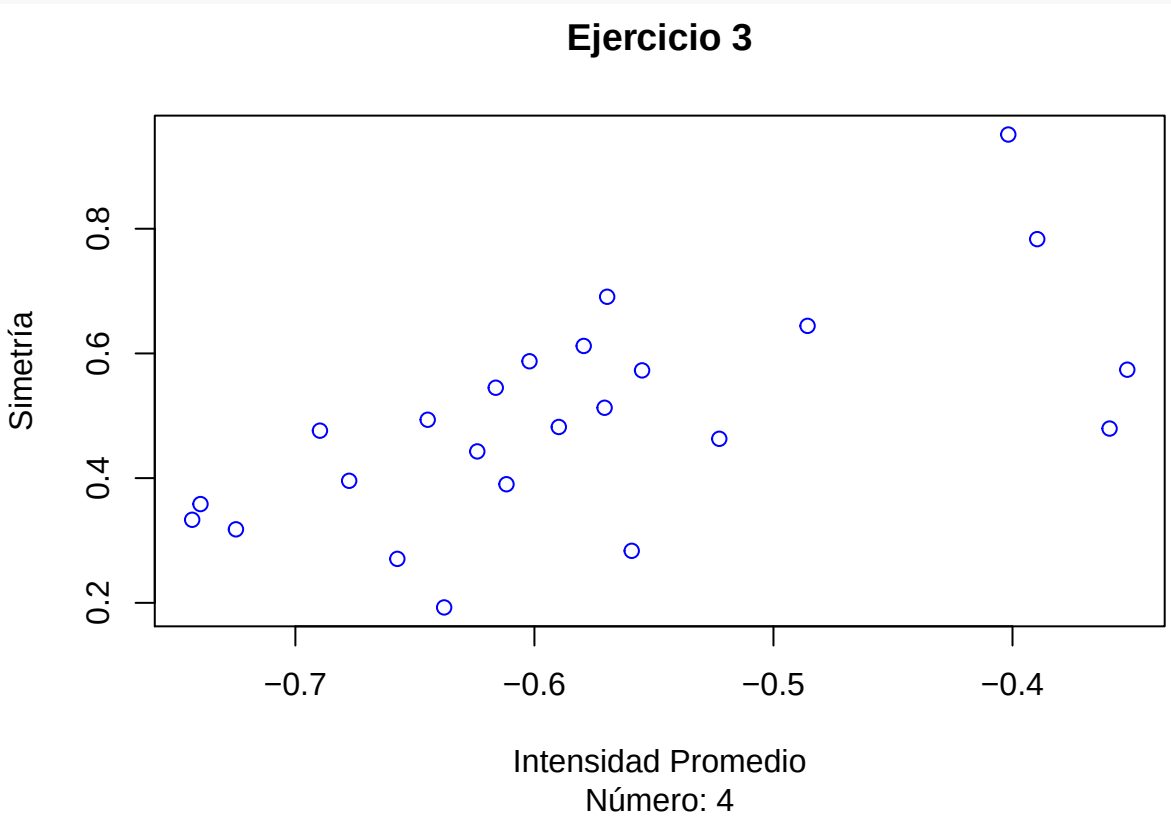

```
# Etiquetas correspondientes a las "numImageToShow" primeras imágenes
digitosTest[1:numImageToShow]
```

```
##  [1] 4 4 8 8 4 8 4 8 8 4
```

```r
# Se mostrará una sola gráfica a la vez
par(mfrow=c(1, 1))

# Se crea una matriz con 2 columnas para la media y la simetría y tantas
# filas como números 4 y 8 haya
matrixMeanSymTest = matrix(ncol = 2, nrow = ndigitosTest)

# Se le da nombre a las columnas
dimnames(matrixMeanSymTest) = list(c(), c("mean", "symmetry"))

# Se recorre la matriz 3d donde están todos los datos y se va creando
# la matriz con los resultados de los cálculos.
# He intentado hacerlo con apply por filas, obteniendo la matriz de cada
# número pero no conseguía reflejar los datos como debía
for(i in 1:ndigitosTest){

  matrixMeanSymTest[i, ] = getMeanSym(grisesTest[i, , ])

}
```

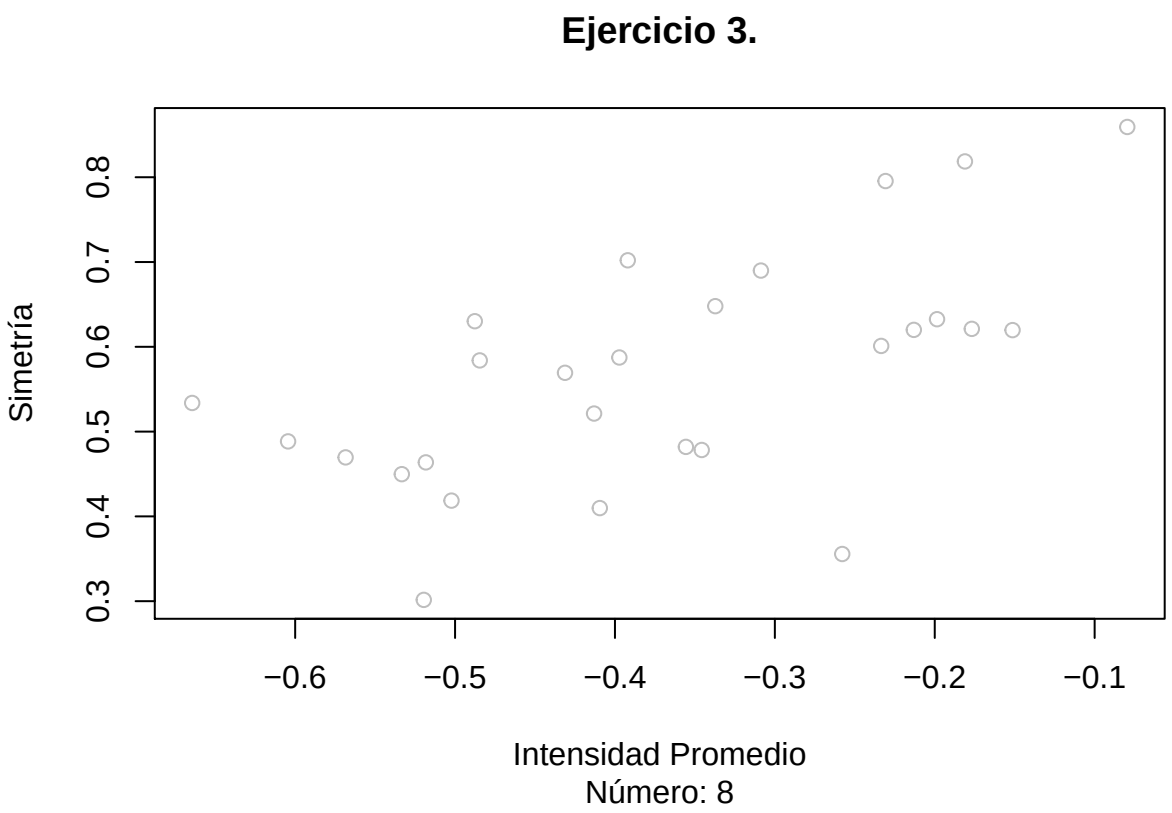
Gráfica con los dígitos *4* de los datos de test:

```r
# Gráfica con los 4
plot(matrixMeanSymTest[which(digitosTest==4), ],
     xlab="Intensidad Promedio", ylab="Simetría", col=4,
     main="Ejercicio 3", sub="Número: 4")
```



Gráfica con los dígitos *8* de los datos de test:

```
# Gráfica con los 8
plot(matrixMeanSymTest[which(digitosTest==8), ],
     xlab="Intensidad Promedio", ylab="Simetría", col=8,
     main="Ejercicio 3.", sub="Número: 8")
```



Se genera el gráfico con los datos de test.

La línea rosa ha sido generada con Regresión Lineal.

La línea verde ha sido generada con el algoritmo PLA Pocket.

```
# Gráfica con los dos números a la vez
plot(matrixMeanSymTest, col=digitosTest, xlab="Intensidad Promedio",
     ylab="Simetría", main="Ejercicio 3",
     sub="Datos de test")

# Se escribe en la gráfica a qué corresponde cada color
legend(x="topleft", legend=c("4","8"), col=c(4, 8), lwd=3)

# Se copian las etiquetas de los números para cambiar los 4 por -1 y los
# 8 por 1 y ajustarlas a Regress_lim con -1, 1
lD3Test = digitosTest
lD3Test[which(lD3Test==4)] = -1
lD3Test[which(lD3Test==8)] = 1

# Se calcula la regresión lineal con la matriz de medias y simetrias
# verticales y con las etiquetas de -1, 1 que representa los digitos
wLR3Test = Regress_Lin(cbind(matrixMeanSymTest, 1), lD3Test)

# Se dibuja la línea que ajustaría el error mínimo
```

```r
abline(-wLR3Test[3]/wLR3Test[2], -wLR3Test[1]/wLR3Test[2], col=6)

# Se guarda el error antes de calcular los pesos con PLA_POCKET
ETestPLA_POCKET = errorData(cbind(matrixMeanSymTest, 1), lD3Test, wLR3Test)

print(paste("ETest calculado con Regresión Lineal: ",
            round(ETestPLA_POCKET*100, digits = 3), "%"))
```

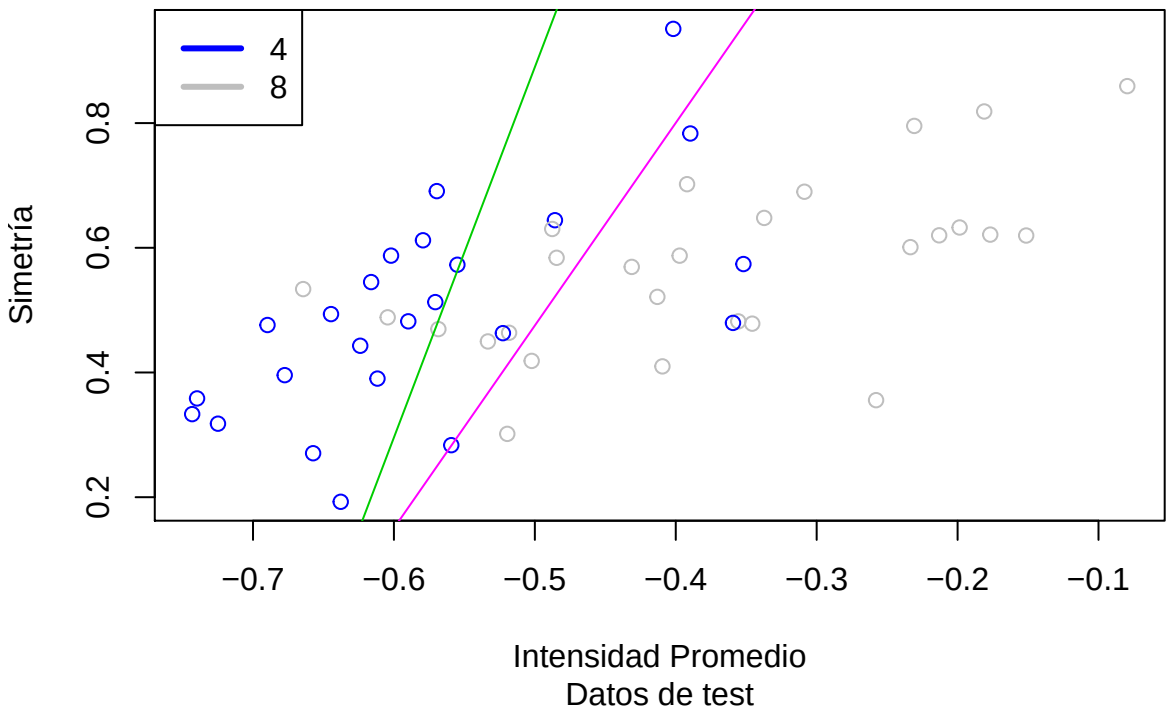
```
## [1] "ETest calculado con Regresión Lineal:  19.608 %"
```

```r
plaTest = ajusta_PLA_POCKET(matrixMeanSymTest, lD3Test, max_iter, wLR3Test)

# Se guarda el error después de calcular los pesos con PLA_POCKET
ETestPLA_POCKET = errorData(cbind(matrixMeanSymTest, 1), lD3Test, plaTest[1:3])

print(paste("ETest calculado con PLA_POCKET: ",
            round(ETestPLA_POCKET*100, digits = 3), "%"))
```

```
## [1] "ETest calculado con PLA_POCKET:  17.647 %"
```

```r
# Se dibuja la recta generada con el algoritmo PLA_POCKET
abline(-plaTest[3]/plaTest[2], -plaTest[1]/plaTest[2], col=3)
```

Ejercicio 3



Conclusión:

Se puede ver como el error de la muestra de entrenamiento con Regresión Lineal es 24.769% (línea rosa) y como con el algoritmo PLA modificado con Pocket (línea verde), se ha guardado la mejor solución, siendo ésta de 22.222%.

En la muestra de test ocurre igual, el error con Regresión Lineal (línea rosa) es del 19.608%, siendo superado

por el algoritmo PLA modificado con Pocket (línea verde), que es de 17.647%.

Estos datos nos indican que está funcionando correctamente el algoritmo PLA_Pocket y pese a que se obtienen errores muy superiores a los que se obtenían en la práctica 1, es debido a las muestras. Los 1 y 5 están muy diferenciados entre sí y, los 4 y 8 es más difícil encontrar una gran separación, lo que hace que aumente el error.

Apartado 3

Para calcular la cota de generalización se crea una función en la que se le pasen los parámetros y devuelva el valor que hay que sumar al error ya calculado

```
####################
# Calcula el valor que hay que sumar al error para obtener la cota de
# generalización
#
# Referencias:
# - Página 17, sesión 4
#
generalizationBound<-function(N, dvc, d){
  sqrt((8/N)*log((4*((2*N)^dvc+1))/0.05))
}

# N es el número de ejemplos de la muestra
NTrain = nrow(matrixMeanSym)
NTest = nrow(matrixMeanSymTest)

# La dimensión de Vapnik-Chervonenkis es el número de columnas que tiene
# la muestra más uno
dvc = ncol(matrixMeanSym)+1

# La tolerancia se indica que sea de 0.05
d = 0.05

# Se calcula el valor que se debe sumar al error
vEin = generalizationBound(NTrain, dvc, d)
vEtest = generalizationBound(NTest, dvc, d)

print(paste("El valor obtenido para calcular la cota a partir de Ein es: ",
            round(vEin, digits = 3), " a partir de ", NTrain,
            " ejemplos. La cota Eout es: ",
            round(EinPLA_POCKET+vEin, digits = 3)))
```

```
## [1] "El valor obtenido para calcular la cota a partir de Ein es:  0.676  a partir de  432  ejemplos.
```

```
print(paste("El valor obtenido para calcular la cota a partir de Etest es: ",
            round(vEtest, digits = 3), " a partir de ", NTest,
            " ejemplos. La cota Eout es: ",
            round(ETestPLA_POCKET+vEtest, digits = 3)))
```

```
## [1] "El valor obtenido para calcular la cota a partir de Etest es:  1.692  a partir de  51  ejemplos
```

```
vEaux = generalizationBound(10000, dvc, d)

print(paste("El valor obtenido para calcular la cota a partir de 10.000 ejemplos es: ",
            round(vEaux, digits = 3)))
```

```
## [1] "El valor obtenido para calcular la cota a partir de 10.000 ejemplos es:  0.165"
```

Conclusión:

Como se puede apreciar, el valor obtenido con la muestra de test es una cota muy pobre, siendo superior incluso a 1. Esto es debido a que la cantidad de ejemplos de la muestra de test es insuficiente (51 ejemplos).

Con la muestra de entrenamiento ya aparece un valor más lógico al tener 432 ejemplos, aunque sigue siendo elevado.

Suponiendo el caso de tener una muestra con 10.000 ejemplos se puede apreciar como se podría llegar a conclusiones más certeras con 0.165.

Se puede concluir que es mejor la cota de la muestra de entrenamiento, porque aunque los pesos están ajustados con esta muestra y podría no ser representativa de la realidad, no se disponen de suficientes ejemplos en la muestra de test.

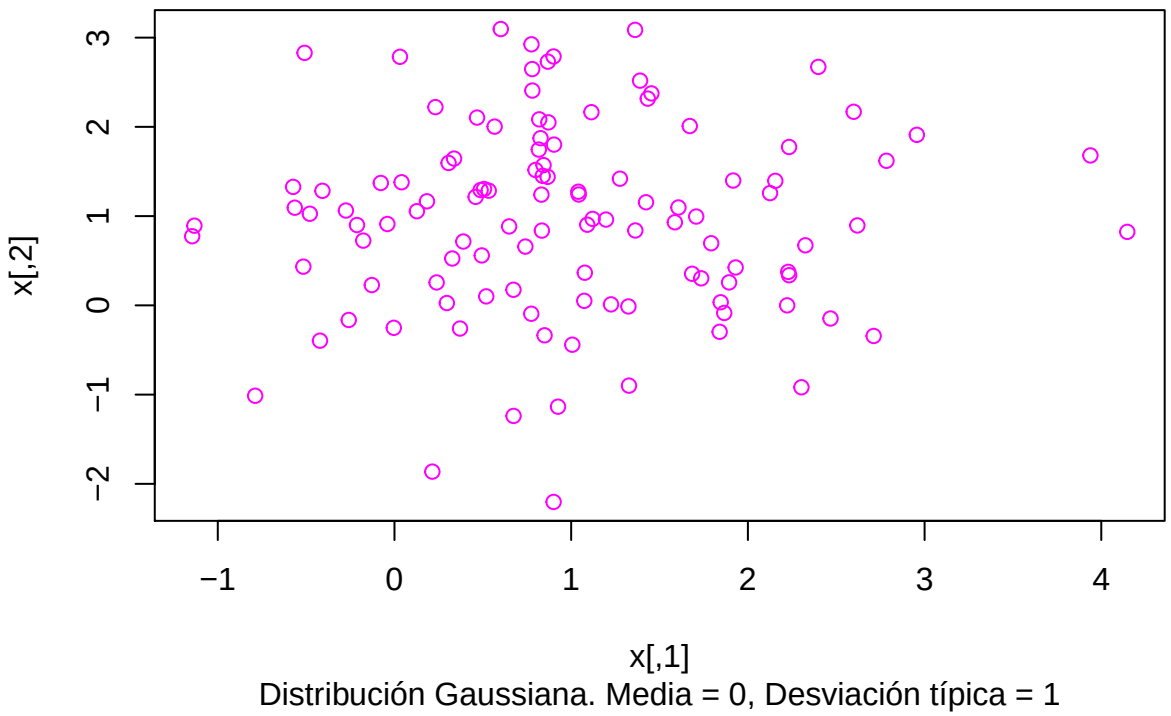
Ejercicio 4. Regularización en la selección de modelos.

Se va a hacer una modificación a la función Regress_Lin anteriormente implementada para incorporar la regularización "weight decay"

```
########################################
# Función Regress_Lin_Weight_decay(datos, label)
#
# Referencias:
# - página 14-16, sesión 3.pdf
# - http://stackoverflow.com/questions/36234136/apply-svd-linear-regression-in-r
Regress_Lin_Weight_Decay = function(datos, label, wd){

  # Para hacer esta versión y sumar lambda, se cambian las operaciones del
  # algoritmo por unas algo menos eficientes, aunque los resultados son
  # iguales, según se explica en la página 86 del libro de Mostafa y página
  # 17 de la sesión 6.

  # Primero se multiplica la matriz por su traspuesta y se le aplica
  # el parámetro de regularización 0.05/N
  #datos.aux<-(t(datos)%*%datos) + (wd*diag(nrow(t(datos)%*%datos)))
  datos.aux<-(t(datos)%*%datos)
  datos.aux = datos.aux + wd*diag(nrow(t(datos)%*%datos))

  # Se hace la descomposición en valores singulares
  svd<-svd(datos.aux)

  # Se calcula su inversa. (xT*x)^-1 = v*d(d^-1)*uT. Al haber aplicado ya
  # el parámetro de regularización realmente queda: (xT*x+l*I)^-1
  datos.inverse <- svd$v %*% diag(1/svd$d) %*% t(svd$u)

  # La pseudoinversa se calcula multiplicando la inversa calculada con la
  # traspuesta de la matriz original: (xT*x+l*I)^-1 * xT
  datos.pseudoinverse <- datos.inverse %*% t(datos)

  # Se termina obteniendo los pesos, multiplicando la pseudoinversa con las
  # etiquetas que se recibe por parámetro
  w <- datos.pseudoinverse %*% label
```

```
  w

}

# Para la generación de los valores de la distribución gaussiana se utiliza
# la función simula_gaus() de la práctica 1
########################################
# Función simula_gaus()
# Genera una matriz de "N" puntos con dimensión "dim". Contiene números
# aleatorios gaussianos de media 0 y desviación típica dada en "sigma".
#
simula_gaus = function(N=2, dim=2, sigma, mean = 0){

  # Si no existe sigma, se termina la ejecución de la función
  if (missing(sigma))
    stop("Debe dar un vector de varianzas")

  # Para la generación se usa sd, y no la varianza
  sigma = sqrt(sigma)

  # Se crea una función que genera 1 muestra con las desviaciones especificadas
  list_simula_gauss = function(){
    rnorm(dim, sd = sigma, mean = mean)
  }

  # Repite N veces, list_simula_gauss y se hace la traspuesta
  m = t(replicate(N, list_simula_gauss()))

  m

}

# Dimensión del vector de características
d = 3

# Se crearán d+110 datos aleatorios, ya que en el apartado a pide que se
# coja como máximo el ejemplo d+110
N = d+110

# Aunque la función rnorm por defecto utiliza estos valores, se crean
# por si hay futuras modificaciones
# Media
mean = 0

# Desviación típica
sd = 1

# Varianza del ruido
sigma = 0.5

# A la función rnorm() se le debe enviar el número de elementos que va a
# crear (1.000), la media (1) y desviación típica (1)
x = simula_gaus(N,d,sd, 1)
```

```r
# Los datos que se han generado son:
plot(x,
     type="p",
     main="Ejercicio 4",
     sub="Distribución Gaussiana. Media = 0, Desviación típica = 1",
     col=6)
```




```r
# Se le añade una columna a 1 para las operaciones
x = cbind(x, 1)

# Se crea el vector de pesos con d+1 valores, con la misma distribución normal
# que antes y sumándole 1 al último valor
w = rnorm(d+1, mean, sd)
w[length(w)] = w[length(w)] + 1

# Se crean las etiquetas asociadas a cada punto x. En la fórmula indica que
# se le aplique la traspuesta, pero no es necesario porque los pesos no los
# tenemos en una matriz de una columna, sino en un vector.
y = apply(x, 1, function(xi){(w%*%xi) + (sigma*rnorm(1, mean, sd))})

# El último dato necesario es el parámetro de regularización weight decay,
# que se fija en 0.05/N
wd = 0.05/N
```

Apartado a

Se deben calcular los errores de validación cruzada y Ecv y después repetir 1.000 el experimento. Para hacerlo más sencillo se crea una función donde lo automatice.

```
###################
# Calcula los errores de validación cruzada y Ecv
#
# x ->   datos de la muestra
# y ->   etiquetas de la muestra
# w ->   vector de pesos calculado con Regresión Lineal con regularización
#        weight decay
# pos -> vector con las posiciones a las que se le va a calcular el error
# d ->   valor que se le suma a la posición que se va a obtener
#
getECV = function(x, y, w, pos, add = 0){

  # Utilizando la fórmula que nos sirve para etiquetar la muestra, se
  # comprueba el error existente entre las etiquetas que ya se han creado
  # y los valores que nos ofrece la Regresión Lineal con Weight Decay.
  # Con sapply se recorren los números que contiene pos, para obtener
  # el error concreto de un ejemplo en concreto y su etiqueta
  e = sapply(pos, FUN = function(i){
    abs(t(w)%*%x[i+add,]-y[i+add])
  })

  # Se devuelve la lista de errores y la media de todos ellos
  list(errors = e, Ecv = mean(e))

}

# Se crean las posiciones que se indica que se deben tener en cuenta
posToCheck = seq(from=10, to=110, by=10)

# Se obtienen los errores calculados con las características que pide el
# apartado a.
res = getECV(x, y, Regress_Lin_Weight_Decay(x, y, wd), posToCheck, d)

print(paste("Los errores de validación cruzada de la muestra son: ",
            round(res$errors, digits = 4)))
```

```
##  [1] "Los errores de validación cruzada de la muestra son:  1.1601"
##  [2] "Los errores de validación cruzada de la muestra son:  0.0564"
##  [3] "Los errores de validación cruzada de la muestra son:  0.3119"
##  [4] "Los errores de validación cruzada de la muestra son:  0.7729"
##  [5] "Los errores de validación cruzada de la muestra son:  0.2721"
##  [6] "Los errores de validación cruzada de la muestra son:  0.0528"
##  [7] "Los errores de validación cruzada de la muestra son:  0.3697"
##  [8] "Los errores de validación cruzada de la muestra son:  0.8941"
##  [9] "Los errores de validación cruzada de la muestra son:  1.2872"
## [10] "Los errores de validación cruzada de la muestra son:  0.06"
## [11] "Los errores de validación cruzada de la muestra son:  0.3107"
```

```
print(paste("La media de los errores de validación cruzada es: ",
            round(mean(res$Ecv)*100, digits = 4),
```

```
            "%"))
```

```
## [1] "La media de los errores de validación cruzada es:  50.4349 %"
```

```
# Una vez que se ha hecho para una sola muestra, se debe repetir el experimento
# para 1.000.
repetitions = 1000

# Se guarda en cada iteración la media que se va calculando de los errores y los
# errores 1 y 2. Se crea un vector donde se irá añadiendo la información
errorMean = vector()
error1Mean = vector()
error2Mean = vector()

# Se repite 1.000 veces la operación anterior
for(i in 1:repetitions){

  # A la función rnorm() se le debe enviar el número de elementos que va a
  # crear (1.000), la media (1) y desviación típica (1)
  x = cbind(simula_gaus(N,d,sd, 1), 1)

  # Se crea el vector de pesos con d+1 valores, con la misma distribución normal
  # que antes y sumándole 1 al último valor
  w = rnorm(d+1, mean, sd)
  w[length(w)] = w[length(w)] + 1

  # Se crean las etiquetas asociadas a cada punto x. En la fórmula indica que
  # se le aplique la traspuesta, pero no es necesario porque los pesos no los
  # tenemos en una matriz de una columna, sino en un vector.
  y = apply(x, 1, function(xi){(w%*%xi) + (sigma*rnorm(1, mean, sd))})

  # Se obtienen los errores calculados con las características que pide el
  # apartado a.
  res = getECV(x, y, Regress_Lin_Weight_Decay(x, y, wd), posToCheck, d)

  errorMean = rbind(errorMean, res$Ecv)
  error1Mean = rbind(error1Mean, res$errors[1])
  error2Mean = rbind(error2Mean, res$errors[2])

}

varErrorMean = var(errorMean)
varError1Mean = var(error1Mean)
varError2Mean = var(error2Mean)

errorMean = mean(errorMean)
error1Mean = mean(error1Mean)
error2Mean = mean(error2Mean)

print(paste("El promedio de e1 es: ",
            round(error1Mean*100, digits = 4),
            "% y su varianza es: ",
            round(varError1Mean, digits = 4)))
```

```
## [1] "El promedio de e1 es:  38.734 % y su varianza es:  0.0864"
```

```r
print(paste("El promedio de e2 es: ",
            round(error2Mean*100, digits = 4),
            "% y su varianza es: ",
            round(varError2Mean, digits = 4)))
```

```
## [1] "El promedio de e2 es:  38.2243 % y su varianza es:  0.0763"
```

```r
print(paste("El promedio de Ecv es: ",
            round(errorMean*100, digits = 4),
            "% y su varianza es: ",
            round(varErrorMean, digits = 4)))
```

```
## [1] "El promedio de Ecv es:  39.0008 % y su varianza es:  0.0082"
```

Apartado b

Conclusión:

Ecv se compone de la media de todos los En de las 1.000 muestras que se han creado, por lo que Ecv tiene relación directa con todos los e1 al utilizar los datos de e1 en el cálculo de la media de Ecv. Esto indica que la media obtenida de Ecv debe ser muy parecida a la de e1 (siempre que el resto de En sea parecido). Se ve como es así: Ecv = 39.0008%, e1 = 38.734%.

La relación entre e1 y e2 viene dada porque los datos se han generado con la misma distribución gaussiana. Su porcentaje de error debería ser parecido y esto se confirma con los datos: e1: 38.734%, e2: 38.2243%.

Apartado c

Conclusión:

Lo que más contribuye a la varianza de e1 es la distribución gaussiana y el rango que puede tener el valor aleatorio que se genera, así como el vector de pesos creado, que influye en el error que se produce.

Apartado d

Conclusión:

La regularización consigue estabilizar los valores que se calculan con Regresión Lineal, lo que hace que de media todos los errores sean muy parecidos. El set de funciones que se pueden obtener es mas estable. Con este sistema se ha conseguido disminuir la varianza y dejarla en valores muy pequeños, perjudicando el sesgo. Hay que tener cuidado con el parámetro lambda que define la intensidad de regularización, porque puede provocar un sobreajuste de los datos y que se obtenga un error demasiado elevado.

Bonus 1. Coordenada descendente.

La función es igual que el Gradiente Descendente, pero que la generación de u, v se hace independientemente.

```r
####################
# Función Coordenada Descendente
# f ->       función original
# fduv ->    derivada de f
# w ->       valores iniciales de x,y o de u,v. Por defecto, (1,1)
# mu ->      tasa de aprendizaje
# tol ->     valor en el que debe parar el GD cuando f sea menor que él
```

```
# maxIter -> máximo número de iteraciones que el algoritmo tiene permitido
#            realizar. Por defecto, es el mayor entero que acepta la máquina.
#
CD = function(f, fduv, w=c(1,1), mu, tol, maxIter=.Machine$integer.max){

  # Se guarda el vector para hacer los cálculos intermedios. Se le suma
  # el máximo para que la primera vez entre en el ciclo
  wIni = w+.Machine$integer.max

  # Contador de iteraciones
  i = 0

  # Booleano para saber si debe seguir ejecutando el bucle while. Será FALSE
  # cuando el nuevo vector de pesos no mejore o no empeore más de un umbral
  # dado
  continue = TRUE

  # Como se deben devolver los pesos generados, se crea un vector donde estarán
  # todos inicializado vacío
  wGlobal = matrix(nrow = 0, ncol = 2)

  # Mientras que pueda seguir realizando iteraciones y no haya llegado al
  # umbral
  while(i<maxIter && continue){

    # Se calcula el valor de la función E con ambos pesos
    EIni = f(wIni[1], wIni[2])
    E = f(w[1], w[2])

    # Si los pesos cumplen la condición de que f(u,v) es menor que tol,
    # termina de calcular
    if(abs(EIni-E) < tol){

      continue = FALSE

    } else{

      wIni = w

      # Se guardan los pesos actuales
      wGlobal = rbind(wGlobal, c(w[1], w[2]))

      # Se calcula el gradiente para cambiar u
      g = fduv(w[1], w[2])

      # Se actualiza u
      w[1] = w[1] - mu*g[1]

      # Se vuelve a calcular el gradiente para cambiar v
      g = fduv(w[1], w[2])

      # Se actualiza v
      w[2] = w[2] - mu*g[2]
```

```
      # Se aumenta el contador de iteraciones
      i = i+1

    }

  }

  # Se devuelven aquí para no condicionar al resto con el formato de uno
  list(data = wGlobal, ini1 = wGlobal[1,1], ini2 = wGlobal[1,2], f = E, u = w[1], v= w[2], i = i)

}

# Ejecución del ejercicio Bonus 1
res = CD(f1a, fduv1a, c(1,1), 0.1, 10^-14)

print(paste("El número de iteraciones que tarda en obtener un valor inferior a 10^-4 es: ", res$i))
```

```
## [1] "El número de iteraciones que tarda en obtener un valor inferior a 10^-4 es:  3"
```

```
print(paste("Los valores obtenidos son: u=",
            round(res$u, digits = 5), ", v=", round(res$v, digits = 5),
            " con f = ", res$f))
```

```
## [1] "Los valores obtenidos son: u= 11903.21363 , v= -89355363163.5437  con f =  0"
```

```
# Ejecución del ejercicio 1a
res = GD(f1a, fduv1a, c(1,1), 0.001, 10^-4, 15)

print(paste("Los valores obtenidos con GD y 15 iteraciones son: u=",
            round(res$u, digits = 5), ", v=", round(res$v, digits = 5),
            " con f = ", round(res$f, digits = 5)))
```

```
## [1] "Los valores obtenidos con GD y 15 iteraciones son: u= 0.78545 , v= 0.97864  con f =  0.65628"
```

```
# Ejecución del ejercicio Bonus 1
res = CD(f1a, fduv1a, c(1,1), 0.001, 10^-4, 15)

print(paste("Los valores obtenidos con CD y 15 iteraciones son: u=",
            round(res$u, digits = 5), ", v=", round(res$v, digits = 5),
            " con f = ", round(res$f, digits = 5)))
```

```
## [1] "Los valores obtenidos con CD y 15 iteraciones son: u= 0.78487 , v= 0.98376  con f =  0.65075"
```

```
# Ejecución del ejercicio 1a
res = GD(f1a, fduv1a, c(1,1), 0.001, 10^-4, 30)

print(paste("Los valores obtenidos con GD y 30 iteraciones son: u=",
            round(res$u, digits = 5), ", v=", round(res$v, digits = 5),
            " con f = ", round(res$f, digits = 5)))
```

```
## [1] "Los valores obtenidos con GD y 30 iteraciones son: u= 0.70415 , v= 0.98476  con f =  0.14891"
```

```
# Ejecución del ejercicio Bonus 1
res = CD(f1a, fduv1a, c(1,1), 0.001, 10^-4, 30)

print(paste("Los valores obtenidos con CD 30 iteraciones son: u=",
            round(res$u, digits = 5), ", v=", round(res$v, digits = 5),
```

```
            " con f = ", round(res$f, digits = 5)))
```

```
## [1] "Los valores obtenidos con CD 30 iteraciones son: u= 0.70378 , v= 0.99014  con f =  0.14498"
```

Conclusión:

Se puede ver como la tasa de aprendizaje es demasiado alta y se producen muchas variaciones, al igual que ocurre en el ejercicio 1.b.1.

En solo 3 iteraciones termina el algoritmo y devuelve valor 0.

Si se cambia la tasa de aprendizaje a 0.001, se corrige esa irregularidad y se va encontrando una solución progresivamente.

Teniendo en cuenta este hecho, se utiliza la tasa de aprendizaje 0.001.

Comparando los resultados con Gradiente Descendente y Coordenada Descendente, la Coordenada Descendente ha obtenido mejores valores, aunque la diferencia es muy pequeña.

Bonus 2. Método de Newton.

Se deben hacer las segundas derivadas de la función del ejercicio 1.b

El resultado es:

$d''(x) = -8(pi^2)sin(2ypi)sin(2pix) + 2$

$d''(y) = -8(pi^2)sin(2xpi)sin(2piy) + 4$

$d''(xy) = 8(pi^2)cos(2pix)cos(2piy)$

$d''(yx) = 8(pi^2)cos(2pix)cos(2piy)$

```
###################
# Se realiza la segunda derivada parcial de la expresión:
# f(x,y) = (x-2)^2 + 2*(y-2)^2 + 2*sin(2*pi*x)*sin(2*pi*y)
#
# d''(x) = -8*(pi^2)*sin(2*y*pi)*sin(2*pi*x)+2
# d''(y) = -8*(pi^2)*sin(2*x*pi)*sin(2*pi*y)+4
#
# d''(xyyx) = 8*(pi^2)*cos(2*pi*x)*cos(2*pi*y)
#
# Implementación de las funciones de las derivadas
#
# La doble derivada primero con respecto a x y luego con respecto a y es igual
# a la doble derivada primero con respecto a y y luego con respecto a x, por lo
# que solo se calcula una vez
#
fdxxyy1b = function(x,y){

  # - Con respecto a x:
  dxx = function(x,y){
    -8*(pi^2)*sin(2*y*pi)*sin(2*pi*x)+2
  }

  # - Con respecto a y:
  dyy = function(x,y){
    -8*(pi^2)*sin(2*x*pi)*sin(2*pi*y)+4
  }
```

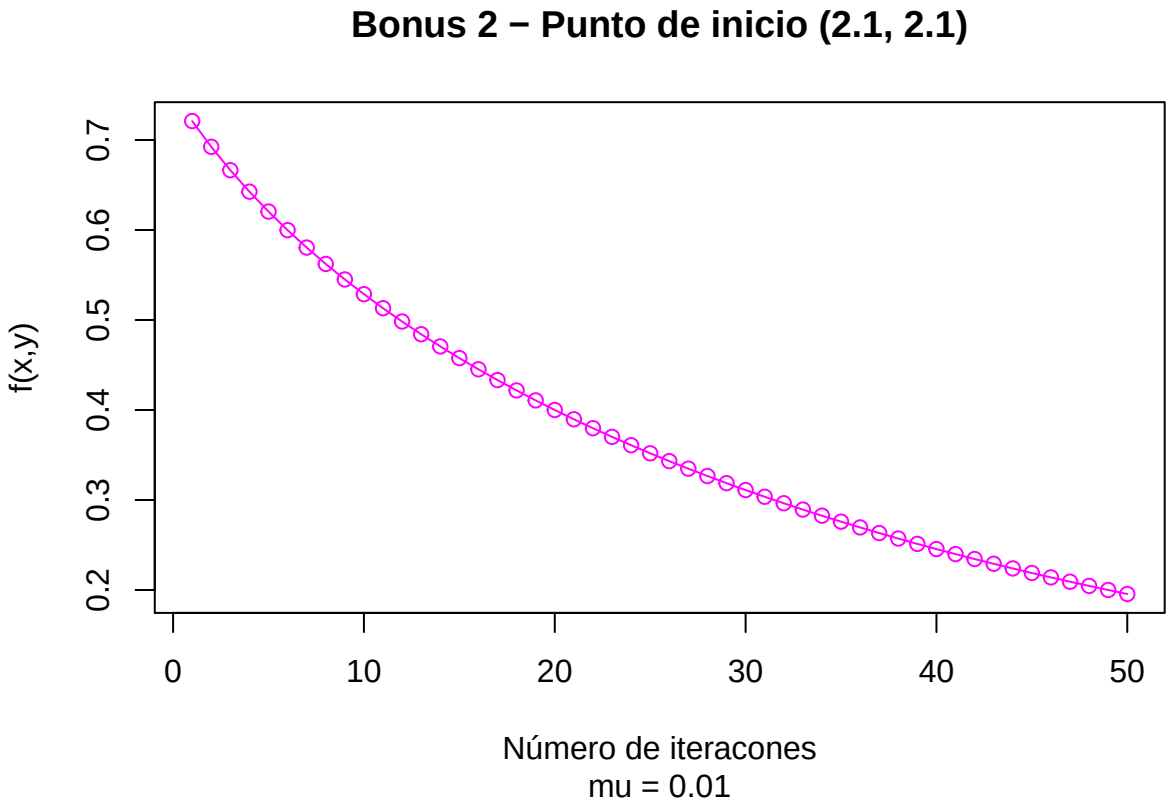
```
  dxyyx = function(x,y){
    8*(pi^2)*cos(2*pi*x)*cos(2*pi*y)
  }

  c(dxx(x,y), dyy(x,y), dxyyx(x,y))

}
```

Para hacer el algoritmo de minimización de Newton se utiliza como base el algoritmo de Gradiente Descendente

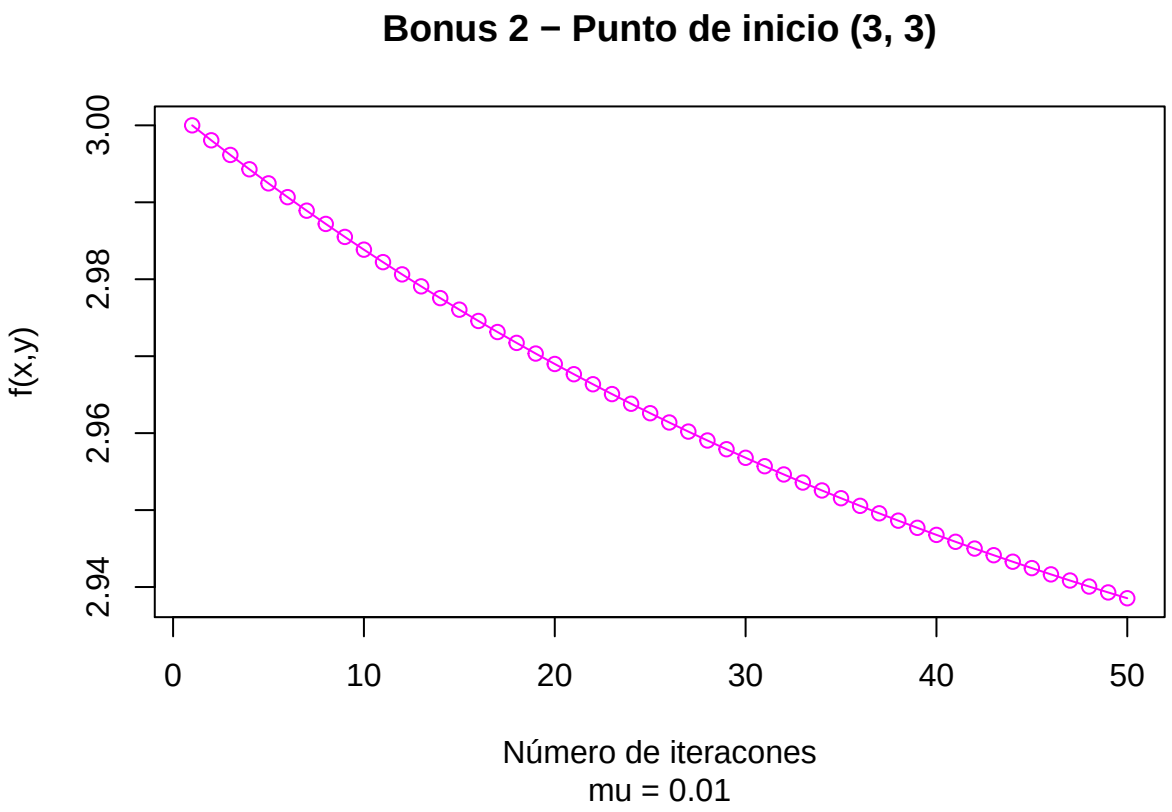
```
####################
# Función MinNewton
# f ->       función original
# fduv ->    derivada de f
# w ->       valores iniciales de x,y o de u,v. Por defecto, (1,1)
# mu ->      tasa de aprendizaje
# tol ->     valor en el que debe parar el GD cuando f sea menor que él
# maxIter -> máximo número de iteraciones que el algoritmo tiene permitido
#            realizar. Por defecto, es el mayor entero que acepta la máquina.
#
# Referencias:
# - Página 31 de la Sesión 5.
# - http://stackoverflow.com/questions/11995832/inverse-of-matrix-in-r
#
minNewton = function(f, fduv, fdduv, w=c(1,1), mu, tol, maxIter=.Machine$integer.max){

  # Se guarda el vector para hacer los cálculos intermedios. Se le suma
  # el máximo para que la primera vez entre en el ciclo
  wIni = w+.Machine$integer.max

  # Contador de iteraciones
  i = 0

  # Booleano para saber si debe seguir ejecutando el bucle while. Será FALSE
  # cuando el nuevo vector de pesos no mejore o no empeore más de un umbral
  # dado
  continue = TRUE

  # Como se deben devolver los pesos generados, se crea un vector donde estarán
  # todos inicializado vacío
  wGlobal = matrix(nrow = 0, ncol = 2)

  # Mientras que pueda seguir realizando iteraciones y no haya llegado al
  # umbral
  while(i<maxIter && continue){

    # Se calcula el valor de la función E con ambos pesos
    EIni = f(wIni[1], wIni[2])
    E = f(w[1], w[2])

    # Si los pesos cumplen la condición de que f(u,v) es menor que tol,
    # termina de calcular
    if(abs(EIni-E) < tol){
```

```
      continue = FALSE

    } else{

      wIni = w

      # Se guardan los pesos actuales
      wGlobal = rbind(wGlobal, c(w[1], w[2]))

      # Se calcula el gradiente
      g = fduv(w[1], w[2])

      # Se calcula la matriz Hessiana. Debe tener formato de matriz, por lo que
      # se crea de ese modo
      hessianMatrix = fdduv(w[1], w[2])

      # Se crea la matriz.
      hessianMatrix = matrix(c(hessianMatrix[1], hessianMatrix[3],
                               hessianMatrix[3], hessianMatrix[2]),
                             2)

      # Se calcula la dirección del gradiente multiplicando la inversa de la matriz
      # hessiana con el gradiente calculado
      gradW = solve(hessianMatrix) %*% g

      # Se calculan los nuevos pesos utilizando el Gradiente y la matriz Hessiana.
      # Se resta por el paso 4 de la página 28 de la sesión 5.
      w[1] = w[1] - mu*gradW[1]
      w[2] = w[2] - mu*gradW[2]

      # Se aumenta el contador de iteraciones
      i = i+1

    }

  }

  # Se devuelven aquí para no condicionar al resto con el formato de uno
  list(data = wGlobal, ini1 = wGlobal[1,1], ini2 = wGlobal[1,2], f = E, u = w[1], v= w[2], i = i)

}

# Se va a mostrar una tabla global con los resultados de las 4 ejecuciones.
# Se irá guardando en res1b2
res1b2 = matrix(ncol = 7, nrow = 0)

# Punto de inicio (2.1, 2.1)
gd1b2 = minNewton(f1b, fdxy1b, fdxxyy1b, c(2.1,2.1), 0.01, 10^-14, 50)

plot(f1b(gd1b2$data[,1], gd1b2$data[,2]),
     type="o",
     main="Bonus 2 - Punto de inicio (2.1, 2.1)",
     sub="mu = 0.01",
```

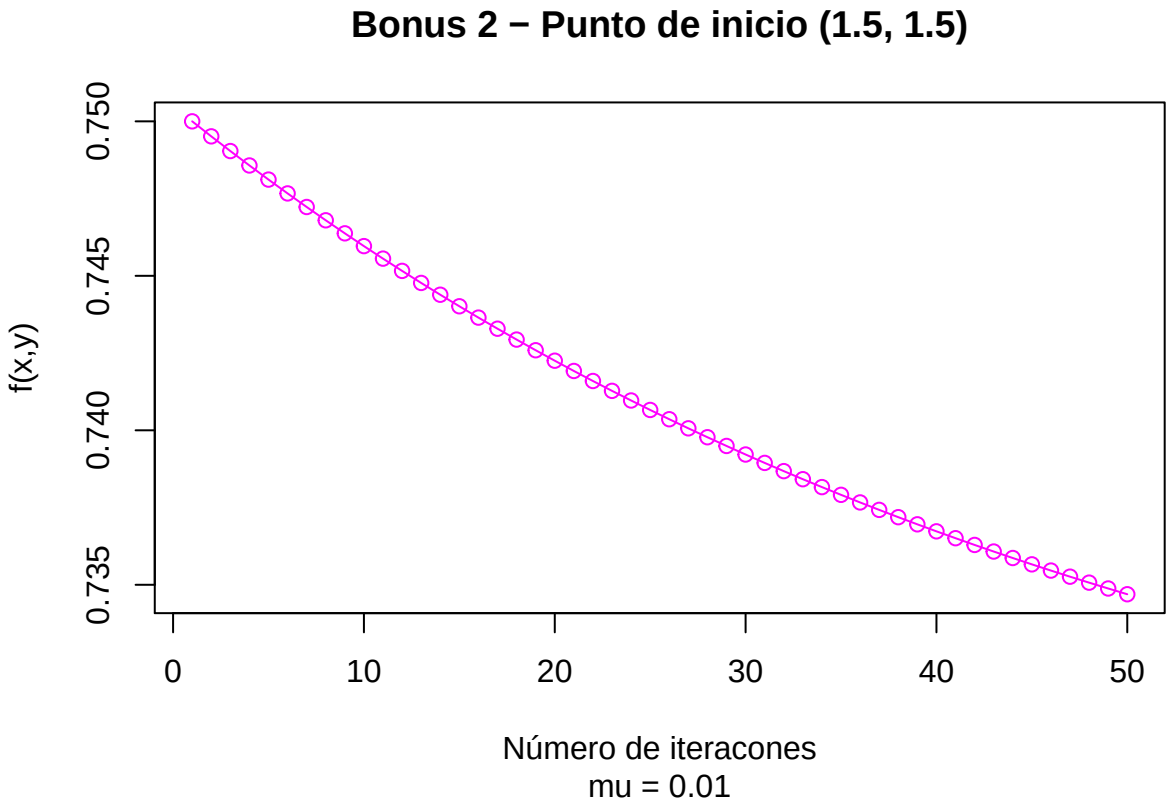
```
     col=6,
     xlab="Número de iteracones",
     ylab="f(x,y)")
```



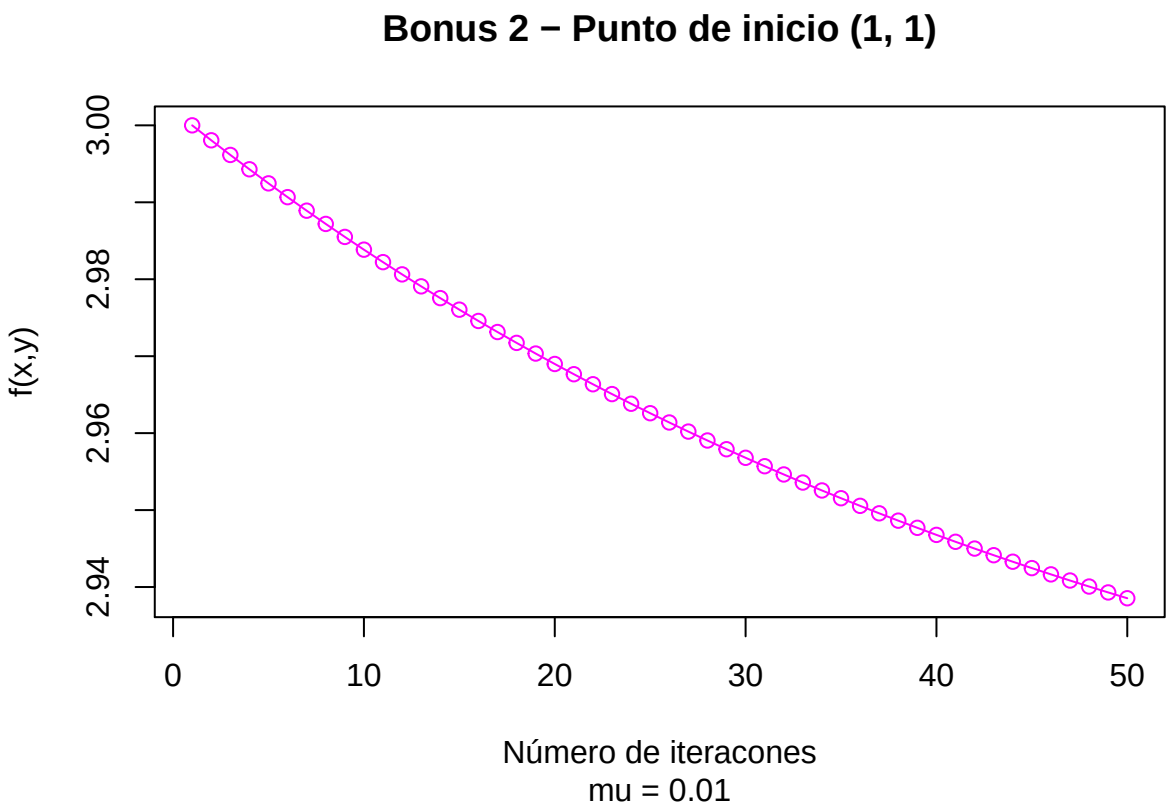

```
# Punto de inicio (2.1, 2.1)
res1b2 = rbind(res1b2, gd1b2)

# Punto de inicio (3, 3)
gd1b2 = minNewton(f1b, fdxy1b, fdxxyy1b, c(3,3), 0.01, 10^-14, 50)

plot(f1b(gd1b2$data[,1], gd1b2$data[,2]),
     type="o",
     main="Bonus 2 - Punto de inicio (3, 3)",
     sub="mu = 0.01",
     col=6,
     xlab="Número de iteracones",
     ylab="f(x,y)")
```




```
# Punto de inicio (3, 3)
res1b2 = rbind(res1b2, gd1b2)

# Punto de inicio (1.5, 1.5)
gd1b2 = minNewton(f1b, fdxy1b, fdxxyy1b, c(1.5, 1.5), 0.01, 10^-14, 50)

plot(f1b(gd1b2$data[,1], gd1b2$data[,2]),
     type="o",
     main="Bonus 2 - Punto de inicio (1.5, 1.5)",
     sub="mu = 0.01",
     col=6,
     xlab="Número de iteracones",
     ylab="f(x,y)")
```




```r
# Punto de inicio (1.5, 1.5)
res1b2 = rbind(res1b2, gd1b2)

# Punto de inicio (1, 1)
gd1b2 = minNewton(f1b, fdxy1b, fdxxyy1b, c(1,1), 0.01, 10^-14, 50)

plot(f1b(gd1b2$data[,1], gd1b2$data[,2]),
     type="o",
     main="Bonus 2 - Punto de inicio (1, 1)",
     sub="mu = 0.01",
     col=6,
     xlab="Número de iteracones",
     ylab="f(x,y)")
```

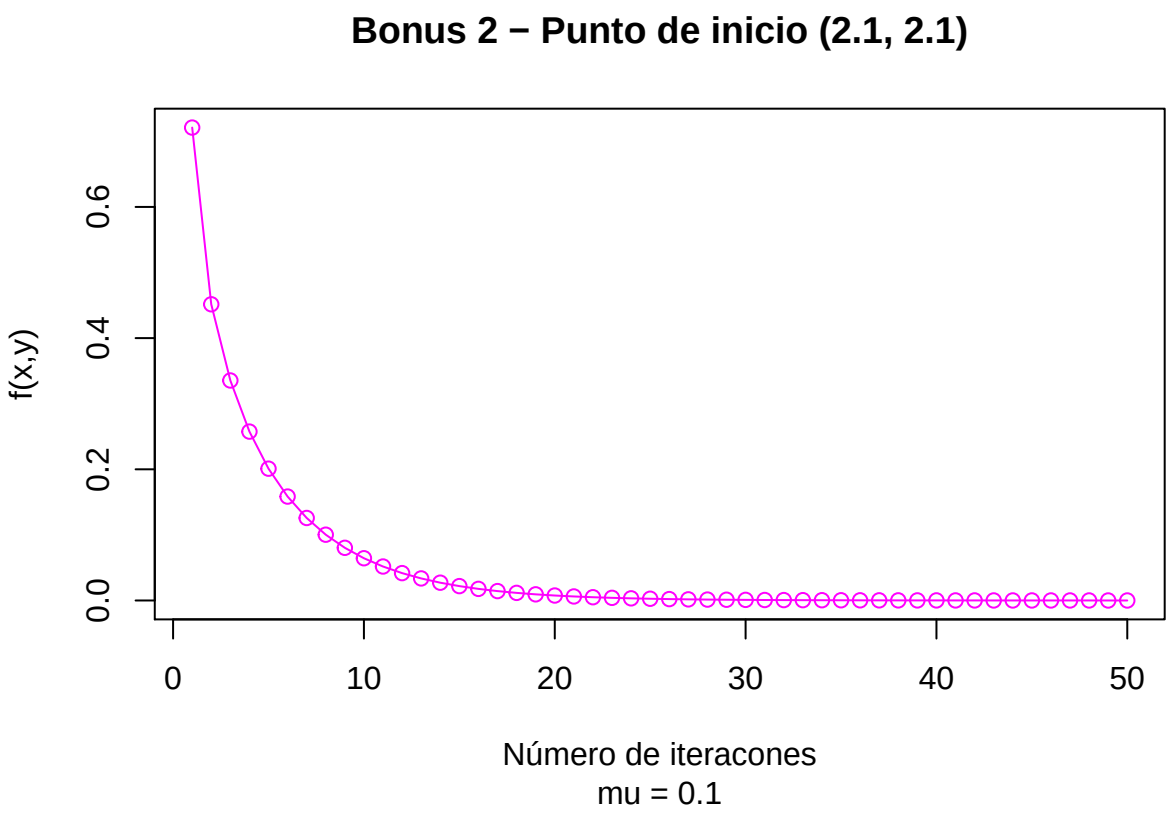



```
# Punto de inicio (1, 1)
res1b2 = rbind(res1b2, gd1b2)

# Se elimina la primera columna, que es la que contiene los pesos que ha
# ido generando en todo el proceso
res1b2 = res1b2[,-1]

dimnames(res1b2) = list(c(), c("Inicio 1", "Inicio 2", "f(x,y)", "x", "y", "Iteraciones"))

# Se muestra la tabla
print(res1b2)
```

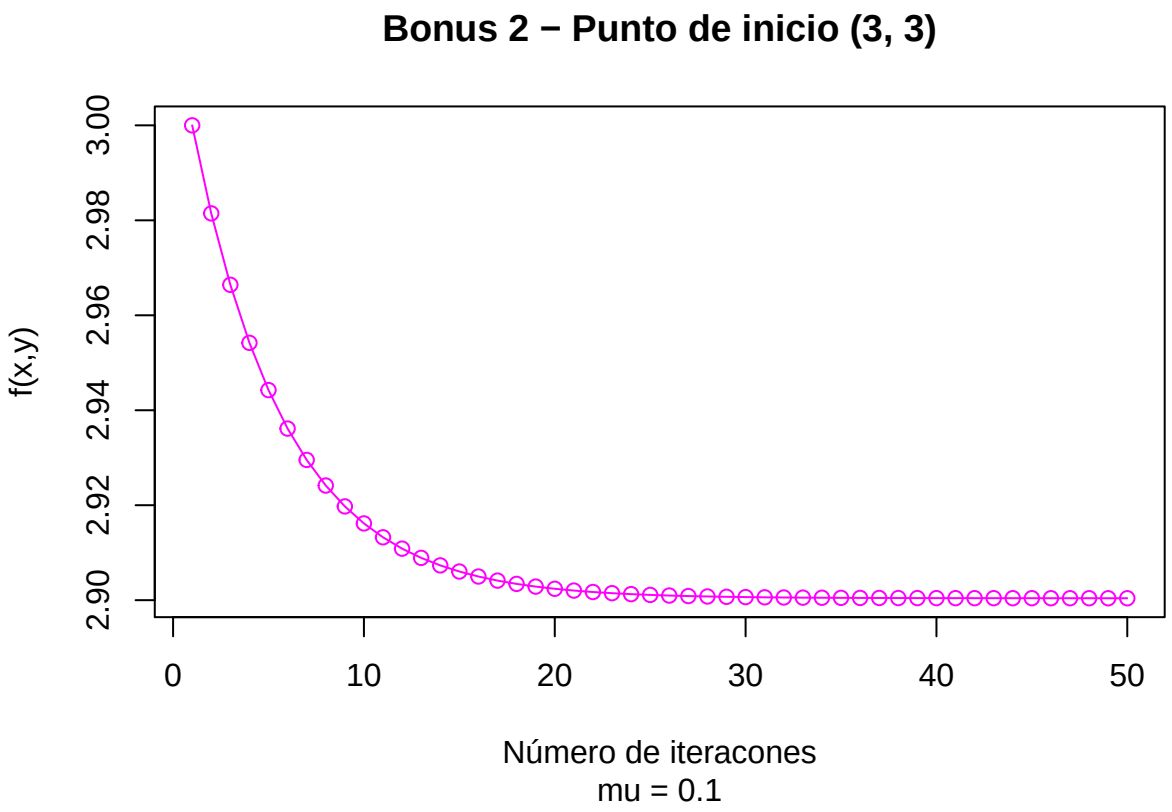
```
##      Inicio 1 Inicio 2 f(x,y)    x        y        Iteraciones
## [1,] 2.1      2.1      0.1955538 2.049202 2.0489   50
## [2,] 3        3        2.938543  2.980392 2.990415 50
## [3,] 1.5      1.5      0.7346939 1.509774 1.504765 50
## [4,] 1        1        2.938543  1.019608 1.009585 50
```

```
# La tasa de aprendizaje es muy pequeña, por lo que se van consiguiendo el mínimo
# muy lentamente.
# Se cambia la tasa de aprendizaje a 0.1 para que la tasa de aprendizaje sea
# mayor y se vea una progresión más rápida

# Se va a mostrar una tabla global con los resultados de las 4 ejecuciones.
# Se irá guardando en res1b2
res1b2 = matrix(ncol = 7, nrow = 0)

# Punto de inicio (2.1, 2.1)
gd1b2 = minNewton(f1b, fdxy1b, fdxxyy1b, c(2.1,2.1), 0.1, 10^-14, 50)
```

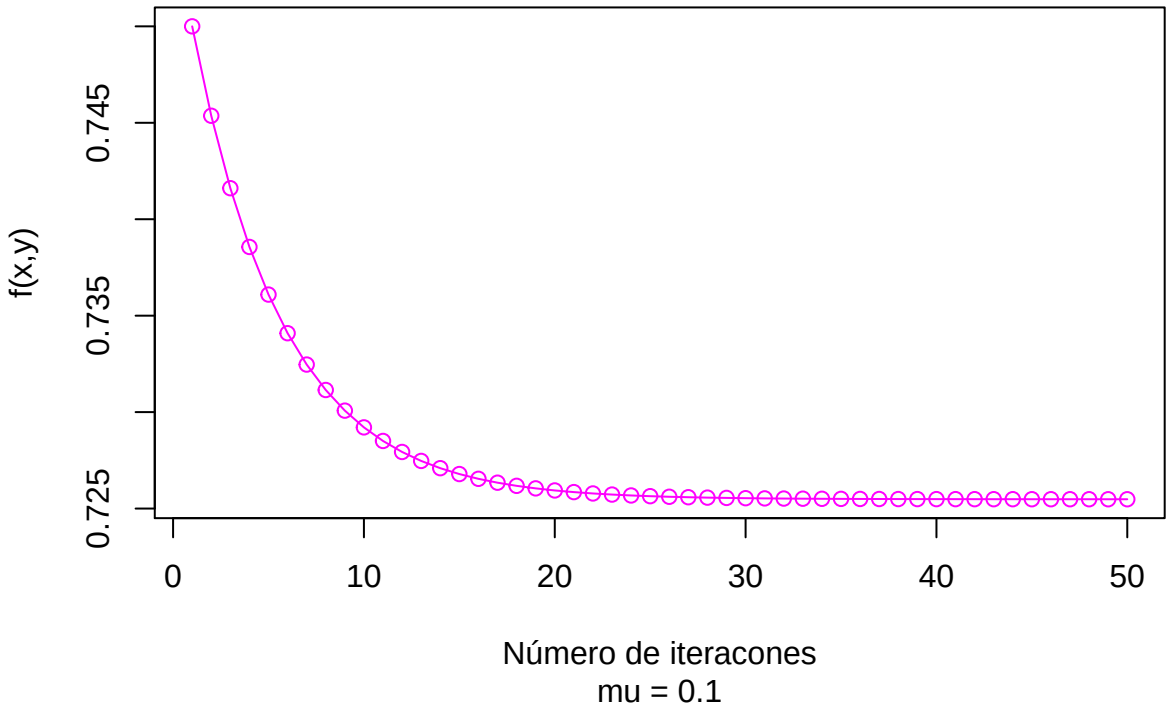
```r
plot(f1b(gd1b2$data[,1], gd1b2$data[,2]),
     type="o",
     main="Bonus 2 - Punto de inicio (2.1, 2.1)",
     sub="mu = 0.1",
     col=6,
     xlab="Número de iteracones",
     ylab="f(x,y)")
```



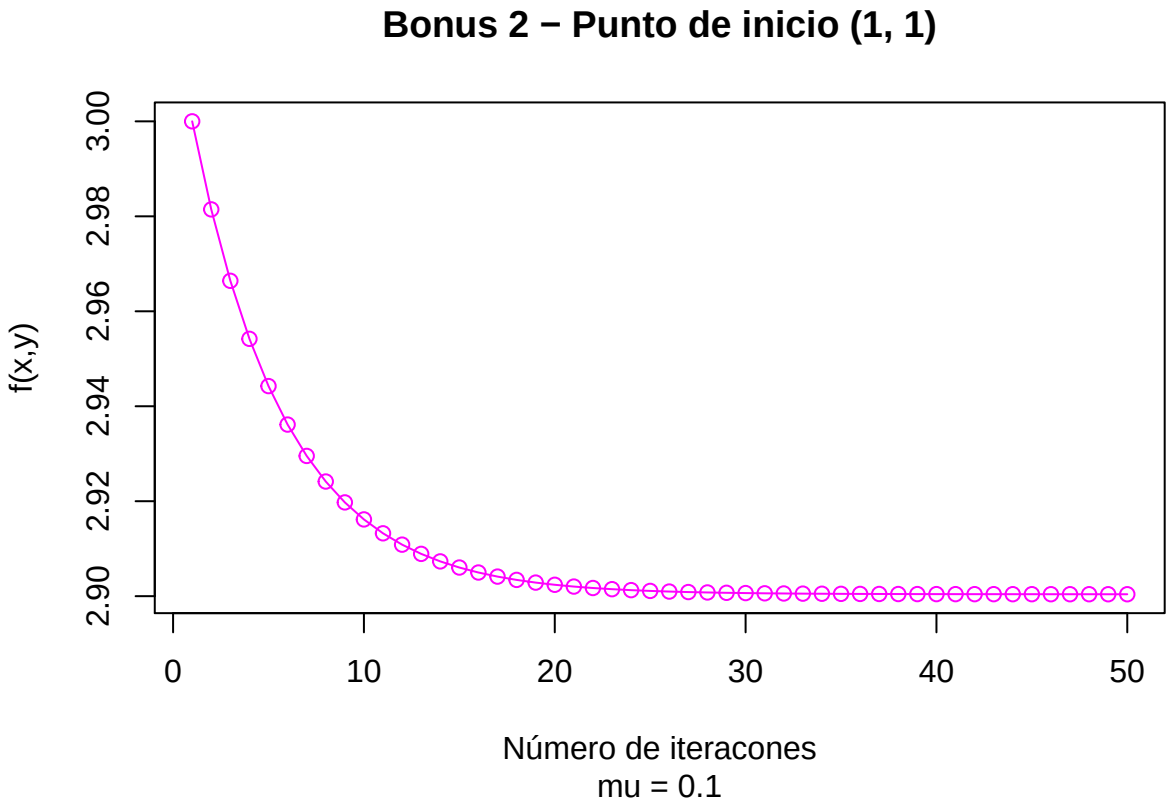

```r
# Punto de inicio (2.1, 2.1)
res1b2 = rbind(res1b2, gd1b2)

# Punto de inicio (3, 3)
gd1b2 = minNewton(f1b, fdxy1b, fdxxyy1b, c(3,3), 0.1, 10^-14, 50)

plot(f1b(gd1b2$data[,1], gd1b2$data[,2]),
     type="o",
     main="Bonus 2 - Punto de inicio (3, 3)",
     sub="mu = 0.1",
     col=6,
     xlab="Número de iteracones",
     ylab="f(x,y)")
```




```r
# Punto de inicio (3, 3)
res1b2 = rbind(res1b2, gd1b2)

# Punto de inicio (1.5, 1.5)
gd1b2 = minNewton(f1b, fdxy1b, fdxxyy1b, c(1.5, 1.5), 0.1, 10^-14, 50)

plot(f1b(gd1b2$data[,1], gd1b2$data[,2]),
     type="o",
     main="Bonus 2 - Punto de inicio (1.5, 1.5)",
     sub="mu = 0.1",
     col=6,
     xlab="Número de iteracones",
     ylab="f(x,y)")
```




```
# Punto de inicio (1.5, 1.5)
res1b2 = rbind(res1b2, gd1b2)

# Punto de inicio (1, 1)
gd1b2 = minNewton(f1b, fdxy1b, fdxxyy1b, c(1,1), 0.1, 10^-14, 50)

plot(f1b(gd1b2$data[,1], gd1b2$data[,2]),
     type="o",
     main="Bonus 2 - Punto de inicio (1, 1)",
     sub="mu = 0.1",
     col=6,
     xlab="Número de iteracones",
     ylab="f(x,y)")
```




```
# Punto de inicio (1, 1)
res1b2 = rbind(res1b2, gd1b2)

# Se elimina la primera columna, que es la que contiene los pesos que ha
# ido generando en todo el proceso
res1b2 = res1b2[,-1]

dimnames(res1b2) = list(c(), c("Inicio 1", "Inicio 2", "f(x,y)", "x", "y", "Iteraciones"))

# Se muestra la tabla
print(res1b2)
```

```
##      Inicio 1 Inicio 2 f(x,y)       x        y        Iteraciones
## [1,] 2.1      2.1      1.355796e-05 2.000368 2.000364 50
## [2,] 3        3        2.900408     2.949409 2.974715 50
## [3,] 1.5      1.5      0.7254829    1.524756 1.512131 50
## [4,] 1        1        2.900408     1.050591 1.025285 50
```

Conclusión: Tras analizar todos los datos, se puede decir que los resultados obtenidos con Gradiente Descendente son mejores que los obtenidos con el método de Newton. Aparte de llegar a un mejor mínimo, el número de iteraciones que necesita es menor.

Bonus 3

```
repetitions = 100

# Se guarda la media de iteraciones que necesita realizar y el valor de Eout
EoutBonus3 = vector()
```

```r
iBonus3 = vector()

# Se repite 1.000 veces la operación anterior
for(i in 1:repetitions){

  # Se generan los datos aleatorios como especifica el ejercicio.
  # Al igual que ocurría en la Regresión Lineal de la práctica 1, la matriz
  # debe coincidir con el vector de pesos, por lo que se le concatena
  # una columna con 1
  data = cbind(simula_unif(100, 2, c(0, 2)), 1)

  # Se crea la recta. Por defecto, los parámetros generan una recta de -1,1,
  # por lo que no se le pasa ningún vector
  line = simula_recta()

  # Se crean las etiquetas de la muestra teniendo en cuenta la recta
  valuesF = apply(data, 1, getValueF, line)

  # Lo que interesa es tener etiquetado cada punto, para eso se crea
  # un vector con -1, 1, teniendo en cuenta el signo de cada punto.
  label = sign(valuesF)

  # El vector de pesos debe estar inicializado a 0
  w = c(0,0,0)

  # Se hace la Regresión Logística con 1000 iteraciones y la tolerancia
  # especificada de 0.01
  res = LR_SGD(data, label, w, 1000, 0.01, 0.01)

  iBonus3 = rbind(iBonus3, res$i)

  # Se calcula ahora el Eout para esta iteración
  x <- replicate(1000, {
    getEOutData(100, 2, c(0,2), w)
  })

  EoutBonus3 = rbind(EoutBonus3, mean(x))

}

print(paste("El valor de Eout para N=100 es: ",
            round(mean(EoutBonus3), digits = 3), "%"))
```

```
## [1] "El valor de Eout para N=100 es:  0.693 %"
```

```r
print(paste("El número de iteraciones que tarda en converger para N=100 es: ",
            round(mean(iBonus3), digits = 3)))
```

```
## [1] "El número de iteraciones que tarda en converger para N=100 es:  56.66"
```