

Práctica 2

Anabel Gómez Ríos

Funciones de la práctica 1 que vamos a reutilizar.

```
set.seed(237)
# Simular números aleatorios uniformes
simula_unif = function (N=2, dims=2, rango = c(0,1)){
  matrix(runif(N*dims, min=rango[1], max=rango[2]), nrow = N, ncol=dims, byrow=T)
}

# Simular recta que corte a un intervalo dado
simula_recta <- function(intervalo) {
  m <- simula_unif(2, 2, intervalo)
  a <- (m[2,2] - m[1,2]) / (m[2,1] - m[1,1])
  b <- m[1,2] - a * m[1,1]
  c(a,b)
}

# Calcular la simetría de una matriz
calcular_simetria <- function(mat) {
  # Invertimos la matriz por columnas
  mat_invertida = apply(mat, 2, function(x) rev(x))
  # Calculamos el valor absoluto de la diferencia de cada elemento entre las dos
  # matrices
  dif = abs(mat - mat_invertida)
  # Sumamos los elementos de la matriz
  suma <- sum(dif)
  # Devolvemos el signo cambiado de la suma
  -suma
}

# Método de regresión lineal
Regress_Lin <- function(datos, label) {
  descomp <- La.svd(datos)
  vt <- descomp[[3]]
  # Creamos la inversa de la matriz diagonal al cuadrado
  diag <- matrix(0, length(descomp[[1]]), length(descomp[[1]]))
  for (i in 1:length(descomp[[1]])) {
    diag[i,i] = descomp[[1]][i]
    if (diag[i,i] != 0) {
      diag[i,i] = 1/(diag[i,i]^2)
    }
  }
  prod_inv <- t(vt) %*% diag %*% vt
  pseud_inv <- prod_inv %*% t(datos)
  w <- pseud_inv %*% label
  w
}
```

```

# Función para contar diferencias dados dos vectores necesaria en la siguiente
# función
cuenta_diferencias <- function(etiquetas1, etiquetas2) {
  vf <- etiquetas1 == etiquetas2
  length(vf[vf == FALSE])
}

# Función para contar errores necesaria en el PLA pocket
cuenta_errores <- function(w, etiquetas_originales, datos) {
  w <- -w/w[2]
  # Etiquetamos con la solución del PLA
  etiquetas_cambiadas <- unlist(lapply(1:nrow(datos), function(i) {
    # Obtenemos los puntos uno a uno y los etiquetamos
    p <- datos[i,]
    f <- -w[1]*p[1] + p[2] - w[3]
    sign(f)
  })))
  # Devolvemos el número de errores que da la solución
  cuenta_diferencias(etiquetas_originales, etiquetas_cambiadas)
}

# Algoritmo PLA pocket
ajusta_PLA_MOD <- function(datos, label, max_iter, vini) {
  parada <- F
  fin <- F
  w <- vini
  wmejor <- w
  iter <- 1
  errores_mejor <- cuenta_errores(wmejor, label, datos)
  # Mientras no hayamos superado el máximo de iteraciones o
  # no se haya encontrado solución
  while(!parada) {
    # iteramos sobre los datos
    for (j in 1:nrow(datos)) {
      if (sign(crossprod(w, datos[j,])) != label[j]) {
        w <- w + label[j]*datos[j,]
        # La variable fin controla si se ha entrado en el if
        fin <- F
      }
    }
    # Contamos el número de errores que hay en la solución actual y si
    # es menor que el número de errores en la mejor solución de las que
    # llevamos, nos quedamos con la actual
    errores_actual <- cuenta_errores(w, label, datos)
    if(errores_actual < errores_mejor) {
      wmejor <- w
      errores_mejor <- errores_actual
    }
    # Si no se ha entrado en el if, todos los datos estaban bien
    # clasificados y podemos poner a TRUE la variable parada.
    if(fin == T) {
      parada = T
    }
  }
}

```

```

else {
  fin = T
}
iter <- iter + 1
if (iter >= max_iter) parada = T
}

# Devolvemos el hiperplano, el número máximo de iteraciones al que hemos
# llegado y el número de errores de la mejor solución que hemos encontrado
list(w = wmejor, numIteraciones = iter, errores = errores_mejor)
}

```

1. MODELOS LINEALES

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

```

# Algoritmo del gradiente descendente. Le pasamos a la función la función de
# error, su gradiente (que serán funciones), el punto en el que se empieza, la
# tasa de aprendizaje, el número máximo de iteraciones a realizar, y el mínimo
# error al que queremos llegar, en orden.
# Devuelve los valores de la función de error por los que pasa junto con la
# iteración.
gradienteDescendente <- function(ferror, gradiente, pini, tasa, maxiter, umbral) {
  w <- pini
  i <- 1
  valoresError <- c(i, ferror(pini[1], pini[2]))
  mostrar <- TRUE
  mejora <- TRUE
  while (i <= maxiter && mejora) {
    g <- gradiente(w[1], w[2])
    # Le cambiamos la dirección al gradiente para ir hacia abajo
    v <- -g
    # Nos movemos tanto como indique la tasa
    wnew <- w + tasa*v
    valoresError <- rbind(valoresError, c(i, ferror(wnew[1], wnew[2])))

    if (((abs(ferror(wnew[1], wnew[2])) < umbral) || (i==maxiter)) && mostrar) {
      cat("He necesitado", i, "iteraciones para llegar al error", ferror(wnew[1], wnew[2]), "\n")
      cat("con valores de u y v:", wnew[1], ",", wnew[2])
      mostrar <- FALSE
    }

    if (abs(ferror(wnew[1], wnew[2]) - ferror(w[1], w[2])) < umbral ||
        ferror(wnew[1], wnew[2]) < umbral) {
      mejora <- FALSE
    }
    w <- wnew
    i <- i+1
  }
  return(valoresError)
}

```

a) Considerar la función no lineal de error $E(u, v) = (ue^v - 2ve^{-u})^2$. Usar gradiente descendente y minimizar esta función de error, comenzando desde el punto $(u, v) = (1, 1)$ y usando una tasa de aprendizaje $\eta = 0.1$

- 1) Calcular analíticamente y mostrar la expresión del gradiente de la función $E(u, v)$

Calculamos el gradiente de $E(u, v)$: $\nabla E(u, v) = (\frac{\partial E}{\partial u}, \frac{\partial E}{\partial v}) = (2(ue^v - 2ve^{-u})(e^v + 2ve^{-u}), 2(ue^v - 2ve^{-u})(ue^v - 2e^{-u})) = 2(ue^v - 2ve^{-u})(e^v + 2ve^{-u}, ue^v - 2e^{-u})$

- 2) ¿Cuántas iteraciones tarda el algoritmo en obtener por primera vez un valor de $E(u, v)$ inferior a 10^{-14} ? (Usar flotantes de 64 bits)

```
E <- function(u,v) (u*exp(v) - 2*v*exp(-u))^2
gradE <- function(u,v) {(2*(u*exp(v) - 2*v*exp(-u)))*c(exp(v) + 2*v*exp(-u),
                                                         u*exp(v) - 2*exp(-u))}

val <- gradienteDescendente(E, gradE, c(1,1), 0.1, 20, 10^{-14})
```

```
## He necesitado 10 iteraciones para llegar al error 1.208683e-15
## con valores de u y v: 0.04473629 , 0.02395871
```

- 3) ¿Qué valores de (u, v) obtuvo en el apartado anterior cuando alcanzó el error de 10^{-14}

Como podemos ver en la salida por pantalla anterior, el valor de u ha sido 0.04473628 y el de v ha sido 0.02395873

b) Considerar ahora la función $f(x, y) = x^2 + 2y^2 + 2 * \sin(2\pi x) * \sin(2\pi y)$

- 1) Usar gradiente descendente para minimizar esta función. Usar como valores iniciales $x_0 = 1, y_0 = 1$, la tasa de aprendizaje $\eta = 0.01$ y un máximo de 50 iteraciones. Generar un gráfico de cómo desciende el valor de la función con las iteraciones. Repetir el experimento pero usando $\eta = 0.1$. Comentar las diferencias.

Calculamos primero su gradiente: $\nabla f = (2x + 4\pi * \sin(2\pi y) * \cos(2\pi x), 4y + 4\pi * \sin(2\pi x) * \cos(2\pi y))$

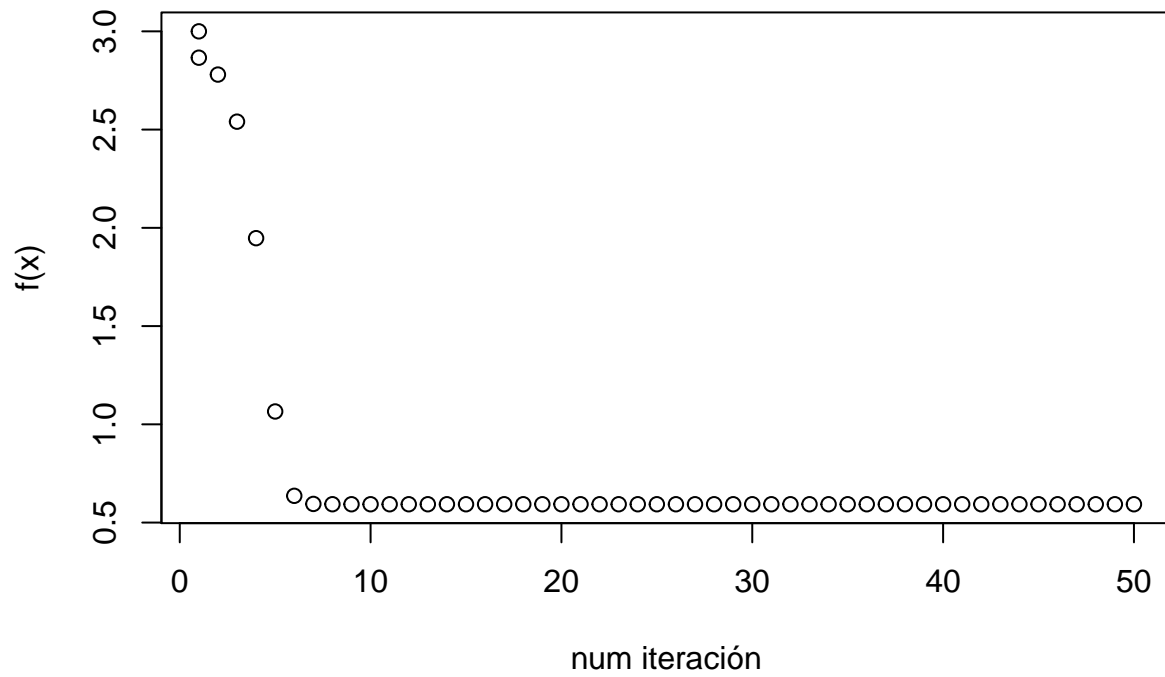
```
f <- function(x,y) x^2 + 2*y^2 + 2*sin(2*pi*x)*sin(2*pi*y)
gradF <- function(x,y) c(2*x + 4*pi*sin(2*pi*y)*cos(2*pi*x),
                        4*y + 4*pi*sin(2*pi*x)*cos(2*pi*y))

val <- gradienteDescendente(f, gradF, c(1,1), 0.01, 50, 0)
```

```
## He necesitado 50 iteraciones para llegar al error 0.5932694
## con valores de u y v: 1.21807 , 0.712812
```

```
plot(val[,1], val[,2], type="p", xlab="num iteración", ylab="f(x)", main="Gradiente Descendente")
```

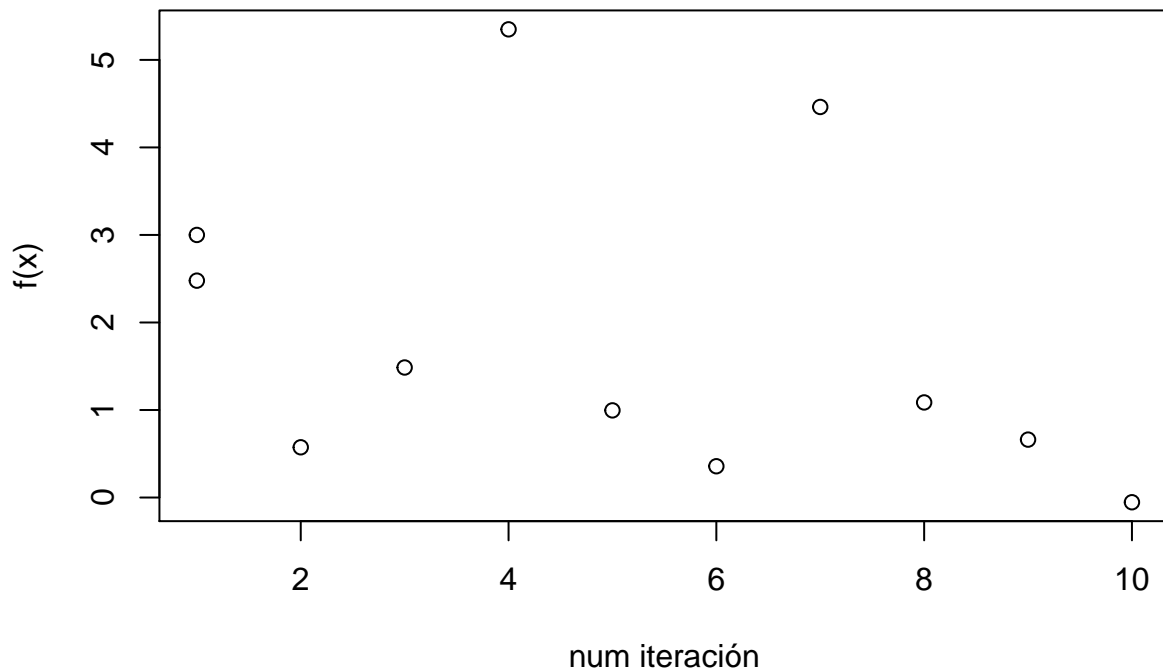
Gradiente Descendente



Repetimos con $\eta = 0.1$:

```
val <- gradienteDescendente(f, gradF, c(1,1), 0.1, 50, 0)
plot(val[,1], val[,2], type="p", xlab="num iteración", ylab="f(x)", main="Gradiente Descendente")
```

Gradiente Descendente



- 2) Obtener el valor mínimo y los valores de las variables que lo alcanzan cuando el punto de inicio se fija: (0.1,0.1), (1,1), (-0.5, -0.5), (-1, -1). Generar una tabla con los valores obtenidos. ¿Cuál sería su conclusión sobre la verdadera dificultad de encontrar el mínimo global de una función arbitraria?

PARA QUÉ TASA DE APRENDIZAJE

```
val <- gradienteDescendente(f, gradF, c(0.1,0.1), 0.01, 50, 0)
val <- gradienteDescendente(f, gradF, c(1,1), 0.01, 50, 0)
```

```
## He necesitado 50 iteraciones para llegar al error 0.5932694
## con valores de u y v: 1.21807 , 0.712812
```

```
val <- gradienteDescendente(f, gradF, c(-0.5,-0.5), 0.01, 50, 0)
val <- gradienteDescendente(f, gradF, c(-1,-1), 0.01, 50, 0)
```

```
## He necesitado 50 iteraciones para llegar al error 0.5932694
## con valores de u y v: -1.21807 , -0.712812
```

2. Coordenada descendente. En este ejercicio comparamos la eficiencia de la técnica de optimización de “coordenada descendente” usando la misma función del ejercicio 1.1.a. En cada iteración, tenemos dos pasos a lo largo de dos coordenadas. En el paso 1 nos movemos a lo largo de la coordenadas u para reducir el error (suponer que se verifica una aproximación de primer orden como en gradiente descendente), y el paso 2 es para reevaluar y movernos a lo largo de la coordenada v para reducir el error (hacer la misma hipótesis que en el paso 1). Usar una tasa de aprendizaje de $\eta = 0.1$.

```
# Algoritmo de coordenada descendente. Le pasamos a la función la función de
# error, su gradiente (que serán funciones), el punto en el que se empieza, la
# tasa de aprendizaje, el número máximo de iteraciones a realizar, y el mínimo
# error al que queremos llegar, en orden.
coordenadaDescendente <- function(ferror, gradiente, pini, tasa, maxiter, umbral) {
  w <- pini
  i <- 1
  mostrar <- TRUE
  mejora <- TRUE
  while (i <= maxiter && mejora) {
    # Paso 1
    g <- gradiente(w[1], w[2])
    # Le cambiamos la dirección al gradiente para ir hacia abajo
    v <- -g
    wnew <- w
    wnew[1] <- w[1] + tasa*v[1]

    # Paso 2
    g <- gradiente(wnew[1], wnew[2])
    # Le cambiamos la dirección al gradiente para ir hacia abajo
    v <- -g
    wnew[2] <- w[2] + tasa*v[2]

    if (((abs(ferror(wnew[1], wnew[2])) < umbral) || (i==maxiter)) && mostrar) {
      cat("He necesitado", i, "iteraciones para llegar al error", ferror(wnew[1], wnew[2]), "\n")
      cat("con valores de u y v:", w[1], ",", w[2])
      mostrar <- FALSE
    }

    if (abs(ferror(wnew[1], wnew[2]) - ferror(w[1], w[2])) < umbral &&
        ferror(wnew[1], wnew[2]) < umbral) {
      mejora <- FALSE
    }

    w <- wnew
    i <- i+1
  }
}
```

a) ¿Qué error $E(u, v)$ se obtiene después de 15 iteraciones completas (i.e. 30 pasos)?

```
val <- coordenadaDescendente(E, gradE, c(1,1), 0.1, 15, 0)
```

```
## He necesitado 15 iteraciones para llegar al error 0.1398138
## con valores de u y v: 6.300845 , -2.823852
```

b) Establezca una comparación entre esta técnica y la técnica de gradiente descendente.

POR HACER

3. Método de Newton. Implementar el algoritmo de minimización de Newton y aplicarlo a la función $f(x, y)$ dada en el ejercicio 1.b. Desarrolle los mismos experimentos usando los mismos puntos de inicio.

```
# Algoritmo del método de Newton. Le pasamos a la función la función de
# error, su gradiente y la matriz hessiana (que serán funciones), el punto
# en el que se empieza, la tasa de aprendizaje, el número máximo de iteraciones a
# realizar, y el mínimo error al que queremos llegar, en orden.
# Devuelve los valores de la función de error por los que pasa junto con la
# iteración.
metodoNewton <- function(ferror, gradiente, hessiana, pini, tasa, maxiter, umbral) {
  w <- pini
  i <- 1
  valoresError <- c(i, ferror(pini[1], pini[2]))
  mostrar <- TRUE
  mejora <- TRUE
  while (i <= maxiter && mejora) {
    hg <- solve(hessiana(w[1], w[2]))%*%gradiente(w[1], w[2])
    # Le cambiamos la dirección al gradiente para ir hacia abajo
    v <- -hg
    # Nos movemos tanto como indique la tasa
    wnew <- w + tasa*v
    valoresError <- rbind(valoresError, c(i, ferror(wnew[1], wnew[2])))

    if (((abs(ferror(wnew[1], wnew[2])) < umbral) || (i==maxiter)) && mostrar) {
      cat("He necesitado", i, "iteraciones para llegar al error", ferror(wnew[1], wnew[2]), "\n")
      cat("con valores de u y v:", wnew[1], ",", wnew[2])
      mostrar <- FALSE
    }

    if (abs(ferror(wnew[1], wnew[2]) - ferror(w[1], w[2])) < umbral ||
        ferror(wnew[1], wnew[2]) < umbral) {
      mejora <- FALSE
    }

    w <- wnew
    i <- i+1
  }
}
```



```

    return(valoresError)
}

```

Calculamos la matriz hessiana de la f . Recordemos que las derivadas parciales cruzadas (de existir y ser continuas, como es nuestro caso) son iguales, por el teorema de Schwarz.

```

f <- function(x,y) x^2 + 2*y^2 + 2*sin(2*pi*x)*sin(2*pi*y)
gradF <- function(x,y) c(2*x + 4*pi*sin(2*pi*y)*cos(2*pi*x),
                        4*y + 4*pi*sin(2*pi*x)*cos(2*pi*y))
d12 <- function(x,y) 8*pi^2*cos(2*pi*y)*cos(2*pi*x)
d11 <- function(x,y) 2 - 8*pi^2*sin(2*pi*y)*sin(2*pi*x)
d22 <- function(x,y) 4 - 8*pi^2*sin(2*pi*x)*sin(2*pi*y)

hess <- function(x,y) rbind(c(d11(x,y), d12(x,y)), c(d12(x,y), d22(x,y)))

val <- metodoNewton(f, gradF, hess, c(1,1), 0.01, 50, 0)

```

```

## He necesitado 50 iteraciones para llegar al error 2.937804
## con valores de u y v: 0.9803919 , 0.9904148

```

```

val2 <- metodoNewton(f, gradF, hess, c(1,1), 0.1, 50, 0)

```

```

## He necesitado 50 iteraciones para llegar al error 2.900408
## con valores de u y v: 0.9494086 , 0.9747153

```

ES ESTO O LO DE DESPUÉS TAMBIÉN?

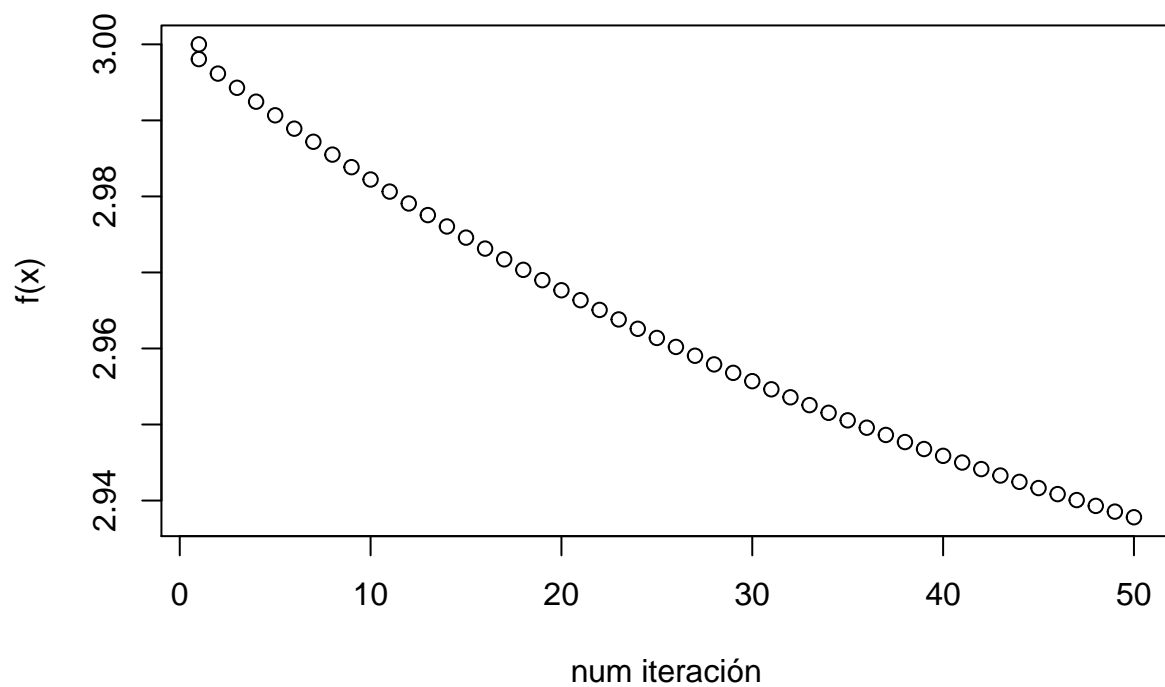
a) Generar un gráfico de cómo desciende el valor de la función con las iteraciones.

```

plot(val[,1], val[,2], type="p", xlab="num iteración", ylab="f(x)", main="Método Newton con tasa 0.01")

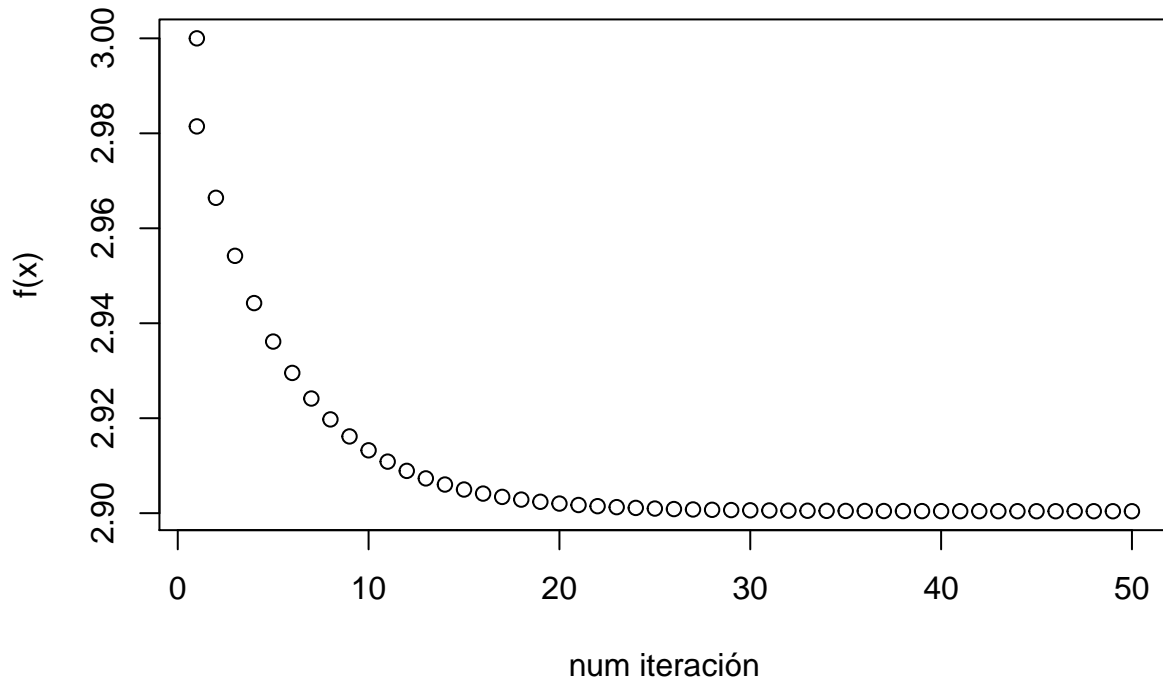
```

Método Newton con tasa 0.01



```
plot(val2[,1], val2[,2], type="p", xlab="num iteración", ylab="f(x)", main="Método Newton con tasa 0.1")
```

Método Newton con tasa 0.1



b) Extraer conclusiones sobre las conductas de los algoritmos comparando la curva de decrecimiento de la función calculada en el apartado anterior y la correspondiente obtenida con gradiente descendente.

POR HACER

4. Regresión Logística. En este ejercicio crearemos nuestra propia función objetivo f (probabilidad en este caso) y nuestro conjunto de datos D para ver cómo funciona regresión logística. Supondremos por simplicidad que f es una probabilidad con valores 0/1 y por tanto que y es una función determinista de x .

Consideremos $d = 2$ para que los datos sean visualizables, y sea $X = [-1, 1] \times [-1, 1]$ con probabilidad uniforme de elegir cada $x \in X$. Elegir una línea en el plano como la frontera entre $f(x) = 1$ (donde y toma valores +1) y $f(x) = 0$ (donde y toma valores -1), para ello seleccionar dos puntos aleatorios del plano y calcular la línea que pasa por ambos. Seleccionar $N = 100$ puntos aleatorios $\{x_n\}$ de X y evaluar las respuestas de todos ellos $\{y_n\}$ respecto de la frontera elegida.

Generamos la muestra de 100 puntos de manera uniforme:

```
muestra <- simula_unif(100, 2, c(-1,1))
```

Generamos la recta en el cuadrado dado que separa los datos:

```
recta <- simula_recta(c(-1,1))
```

Los etiquetamos con respecto a la recta que acabamos de generar:

```
etiquetas <- unlist(lapply(1:nrow(muestra), function(i) {  
  p <- muestra[i,]  
  sign(p[2] - recta[1]*p[1] - recta[2])  
}))
```

a) Implementar Regresión Logística (RL) con Gradiente Descendente Estocástico (SGD) bajo las siguientes condiciones:

1. Inicializar el vector de pesos con valores 0.
2. Parar el algoritmo cuando $\|w^{(t-1)} - w^{(t)}\| < 0.01$, donde $w^{(t)}$ denota el vector de pesos al final de la época t . Una época es un pase completo a través de los N datos.
3. Aplicar una permutación aleatoria de $1, 2, \dots, N$ a los datos antes de usarlos en cada época del algoritmo.
4. Usar una tasa de aprendizaje de $\eta = 0.01$.

Vamos a hacer primero una función auxiliar para calcular la norma euclídea de un vector

```
calcularNorma <- function(vec) {  
  sqrt(sum(vec^2))  
}
```

```
# Algoritmo RL con SGD. Recibe los datos en una matriz, las etiquetas de esos  
# datos, el vector inicial, la tasa de aprendizaje, el máximo número de  
# iteraciones y el umbral para la condición de parada.
```

```
Regress_LogSGD <- function(datos, label, vini, tasa, maxiter, umbral) {  
  parada <- F  
  w <- vini  
  iter <- 0  
  # Mientras no hayamos superado el máximo de iteraciones o  
  # no se hayan acercado lo suficiente w y wnew  
  while(!parada && iter < maxiter) {  
    # Hacemos una permutación al orden en el que vamos a utilizar los datos  
    pos <- sample(1:nrow(datos), nrow(datos))  
    # iteramos sobre los datos  
    wold <- w  
    for (j in pos) {  
      grad <- (-label[j]*datos[j,])/(1+exp(label[j]*crossprod(w, datos[j,])))  
      w <- w - tasa*grad  
    }  
    if(calcularNorma(wold-w) < umbral) {  
      parada <- TRUE  
    }  
  }  
}
```

```

    }
    iter <- iter+1
  }

  # Devolvemos los pesos finales
  return(list(pesos=w, iteraciones=iter))
}

```

b) Usar la muestra de datos etiquetada para encontrar g y estimar E_{out} (el error de entropía cruzada) usando para ello un número suficientemente grande de nuevas muestras.

QUÉ ES EL ERROR DE ENTROPÍA CRUZADA

```

g <- Regress_LogSGD(cbind(muestra,1), etiquetas, c(0,0,0), 0.01, 400, 0.01)
g

```

```

## $pesos
## [1] 1.668128 7.792767 5.084174
##
## $iteraciones
## [1] 371

```

c) Repetir el experimento 100 veces con diferentes funciones frontera y calcule el promedio.

- 1) ¿Cuál es el valor de E_{out} para $N = 100$?
- 2) ¿Cuántas épocas tarda en promedio RL en converger para $N = 100$, usando todas las condiciones anteriormente especificadas?

5. Clasificación de Dígitos. Considerar el conjunto de datos de los dígitos manuscritos y seleccionar las muestras de los dígitos 1 y 5. Usar los ficheros de entrenamiento (training) y test que se proporcionan. Extraer las características de *intensidad promedio* y *simetría* en la manera que se indicó en el ejercicio 3 del trabajo 1.

```

# Leemos los ficheros y extraemos los dígitos 1 y 5
train <- read.table("datos/zip.train", sep=" ")

```

```

## Warning in scan(file, what, nmax, sep, dec, quote, skip, nlines,
## na.strings, : número de items leídos no es múltiplo del número de columnas

```

```

test <- read.table("datos/zip.test", sep=" ")
numero_train <- train$V1
numero_test <- test$V1
frame_train <- train[numero_train==1 | numero_train==5,]
frame_test <- test[numero_test==1 | numero_test==5,]
numero_train <- numero_train[numero_train==1 | numero_train==5]
numero_test <- numero_test[numero_test==1 | numero_test==5]

```

```

# Hacemos los vectores de etiquetas
etiquetas_train = numero_train
etiquetas_train[numero_train==5] = -1
etiquetas_test = numero_test
etiquetas_test[numero_test==5] = -1

# Eliminamos de cada uno la primera columna, que guarda el número del
# que son los datos, y la última, que tiene NA
frame_train = frame_train[,-258]
frame_train = frame_train[,-1]
frame_test = frame_test[,-258]
frame_test = frame_test[,-1]

# Los pasamos a matrices
frame_train <- data.matrix(frame_train)
frame_test <- data.matrix(frame_test)

# Hacemos una lista de matrices
lista_train <- lapply(split(frame_train, seq(nrow(frame_train))), function(x) {
  matrix(x, 16, 16, T)
})
lista_test <- lapply(split(frame_test, seq(nrow(frame_test))), function(x) {
  matrix(x, 16, 16, T)
})

# Eliminamos lo que no vamos a utilizar
rm(train)
rm(test)
rm(frame_train)
rm(frame_test)
rm(numero_train)
rm(numero_test)

```

A continuación calculamos las intensidades y las simetrías y las metemos todas juntas (las de las instancias de 1 y las de 5, pero dejamos separados siempre los conjuntos de train y test)

```

# Calculamos primero las intensidades y las simetrías de todas las matrices,
# es decir, de todas las instancias de 1's y 5's
train_simetria <- unlist(lapply(lista_train, function(m) calcular_simetria(m)))
train_intensidad <- unlist(lapply(lista_train, function(m) mean(m)))

test_simetria <- unlist(lapply(lista_test, function(m) calcular_simetria(m)))
test_intensidad <- unlist(lapply(lista_test, function(m) mean(m)))

```

Plantear un problema de clasificación binaria que considere el conjunto de entrenamiento como datos de entrada para aprender la función g . Usando el modelo de Regresión Lineal para clasificación seguido por PLA-Pocket como mejora. Responder a las siguientes cuestiones:

Utilizamos primero el algoritmo de regresión lineal para obtener una buena solución inicial para dársela al PLA pocket.

Las etiquetas serán 1 o -1 según representen la intensidad o simetría de las instancias de números 1 o 5, respectivamente.

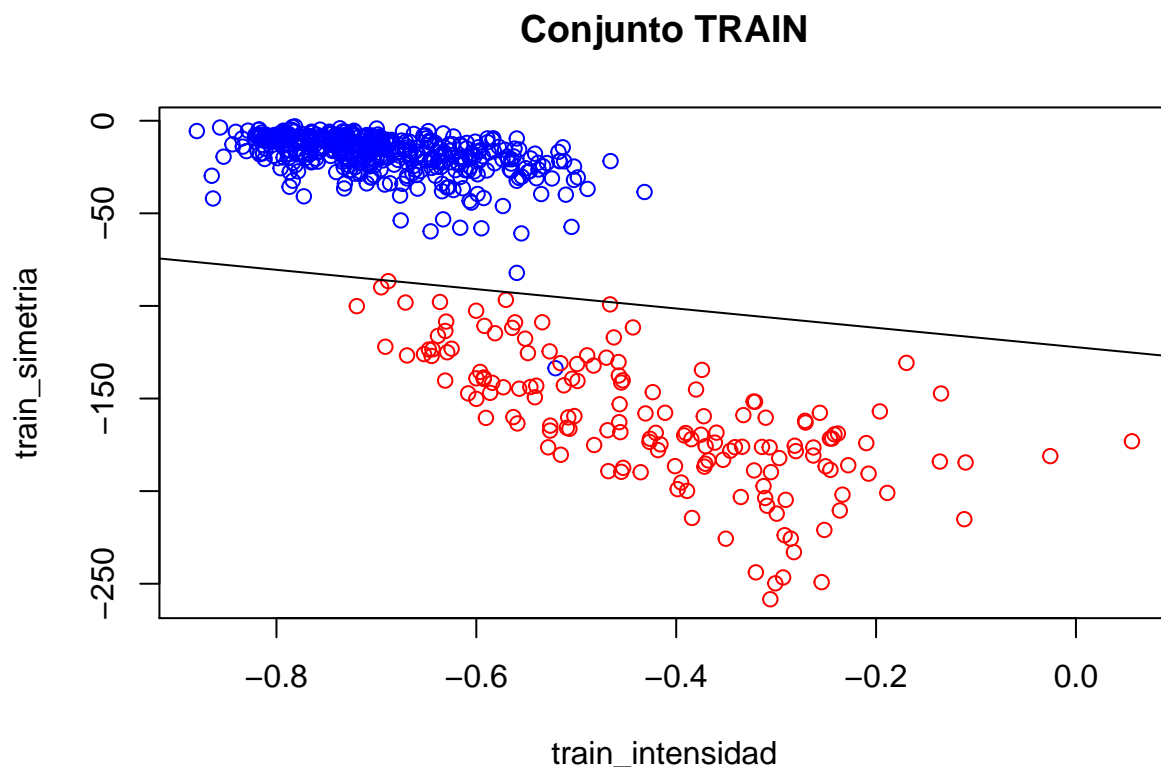
Necesitamos poner como tercera coordenada de la matriz de datos que le vamos a pasar al método de regresión lineal un 1, para ponerlo en coordenadas homogéneas. Nos quedará por tanto cada fila de la matriz de datos como (intensidad, simetría, 1).

```
datos <- cbind(train_intensidad, train_simetria, 1)
w <- Regress_Lin(datos, etiquetas_train)
# Le pasamos lo que nos devuelve la regresión lineal al PLA pocket como solución
# inicial
sol <- ajusta_PLA_MOD(datos, etiquetas_train, 100, w)
```

COMENTAR QUE LA FUNCIÓN ES LA MISMA PORQUE LO QUE DEVUELVE LA REGRESIÓN YA ES CANELA FINA Y EL POCKET NO LO PUEDE MEJORAR (ES LA MEJOR SOLUCIÓN A LA QUE PUEDE LLEGAR ÉL TAMBIÉN).

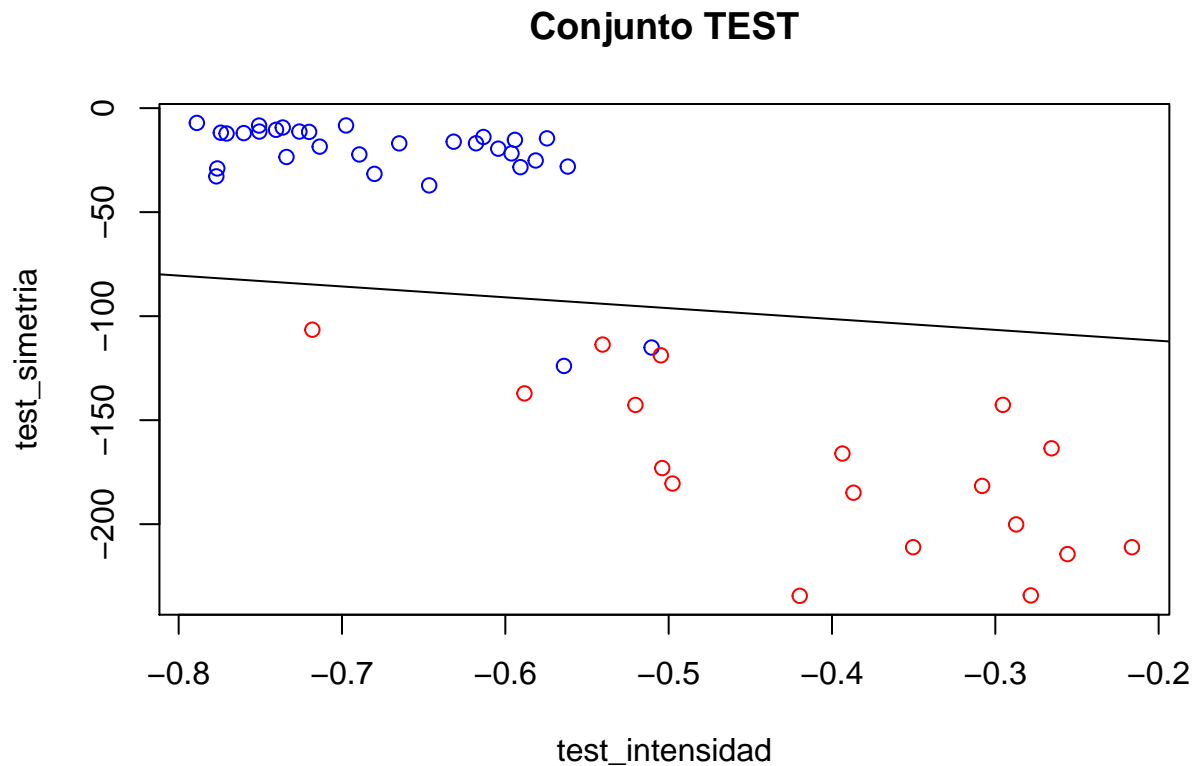
a) Generar gráficos separados (en color) de los datos de entrenamiento y test junto con la función estimada.

```
# Obtenemos la solución devuelta por el PLA
r <- sol[[1]]
r <- -r/r[2]
plot(train_intensidad, train_simetria, type="p", col=etiquetas_train+3, main="Conjunto TRAIN")
abline(r[3], r[1])
```



Dibujamos y mostramos el conjunto de test con la función que hemos calculado con el conjunto de train

```
plot(test_intensidad, test_simetria, type="p", col=etiquetas_test+3, main="Conjunto TEST")
abline(r[3], r[1])
```



b) Calcular E_{in} y E_{test} (error sobre los datos de test).

c) Obtener cotas sobre el verdadero valor de E_{out} . Pueden calcularse dos cotas, una basada en E_{in} y otra basada en E_{test} . Usar una tolerancia $\delta = 0.05$. ¿Qué cota es mejor?

d) Repetir los puntos anteriores pero usando una transformación polinómica de tercer orden ($\Phi_3(x)$ en las transparencias de teoría)

e) Si tuviera que usar los resultados para dárselos a un potencial cliente, ¿usaría la transformación polinómica? Explicar la decisión.

2. SOBREAJUSTE