Práctica 2

Anabel Gómez Ríos

Funciones de la práctica 1 que vamos a reutilizar.

```
set.seed(237)

######
# Funciones de la primera práctica
######

# 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 números aleatorios en una normal
simula_gauss <- function(N, dim, sigma) {
  lapply(1:N, function(x) rnorm(dim, mean = 0, sqrt(sigma)))
}

# 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))
  # Calulamos 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]))
  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])-ferror(w[1], w[2])) < umbral ||
        ferror(wnew[1], wnew[2]) < umbral || i==maxiter) {
      mejora <- FALSE
      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
    }
    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 $E(u,v)$: $\nabla E(u,v) = (\frac{\partial E}{\partial u}, \frac{\partial E}{\partial v}) = (2(ue^v - 2ve^{-u})(e^v + 2ve^{-u}), 2(ue^v - 2e^{-u})(ue^v - 2e^{-u})) = 2(ue^v - 2ve^{-u})(e^v + 2ve^{-u}, ue^v - 2e^{-u})$

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.

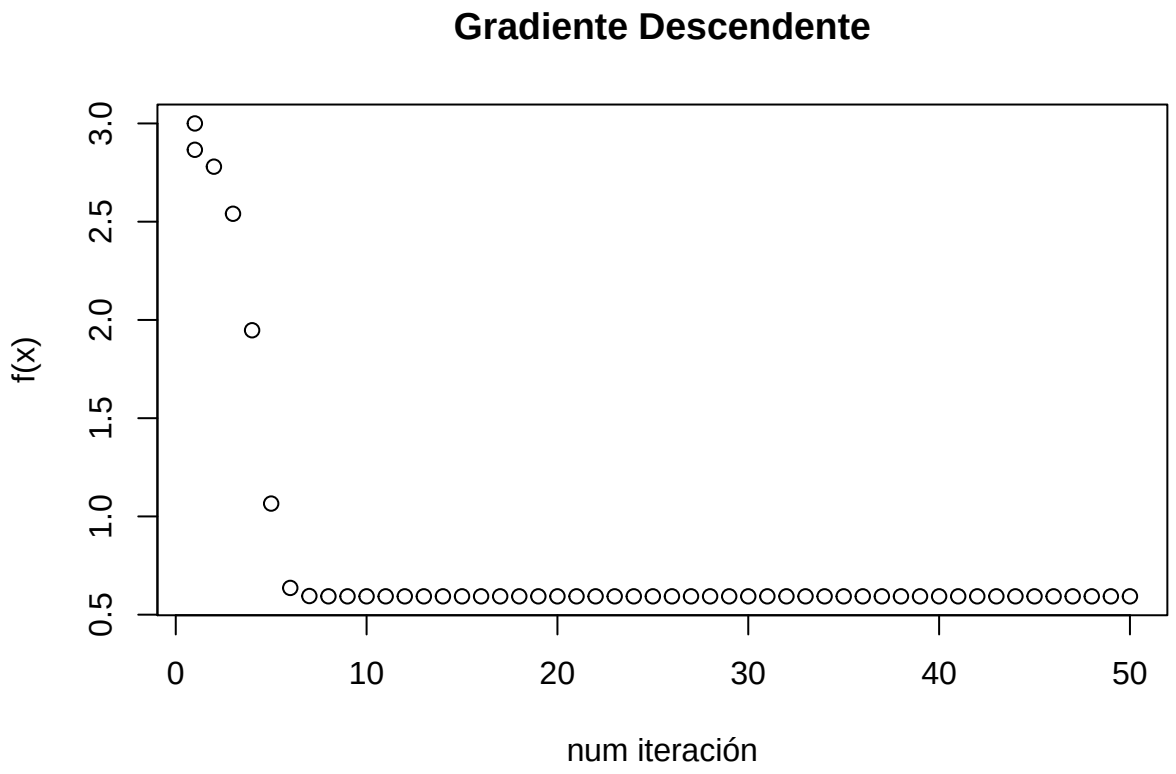
Calculamos primero su gradiente: $\nabla f = (2x + 4\pi * sin(2\pi y) * cos(2\pi x), 4y + 4\pi * sin(2\pi x) * cos(2\pi y))$

```
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")
```

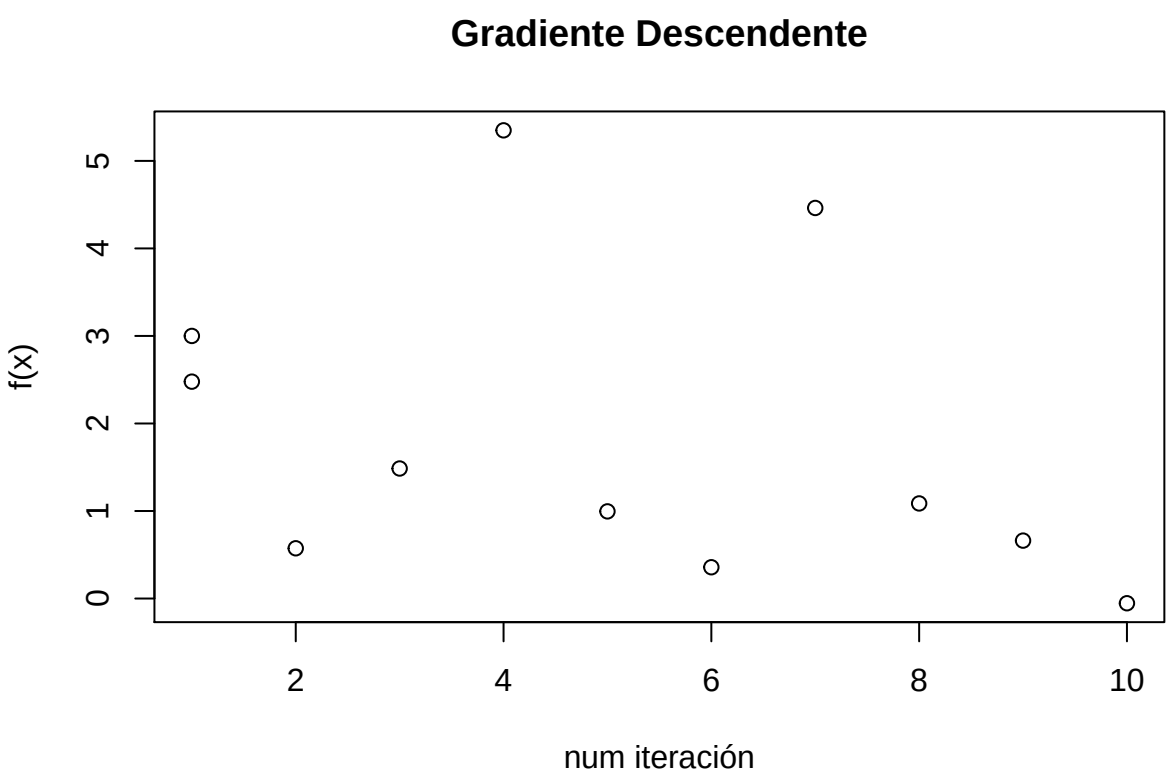


Repetimos con $\eta = 0.1$:

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

```
## He necesitado 10 iteraciones para llegar al error -0.05388005
## con valores de u y v: 0.3881225 , 0.6516421
```

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

Utilizamos la tasa de aprendizaje $\eta = 0.01$, ya que hemos visto que con la otra no converge.

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

```
## He necesitado 3 iteraciones para llegar al error -0.0009552139
## con valores de u y v: 0.005266095 , -0.002392069
```

```
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)
```

```
## He necesitado 5 iteraciones para llegar al error -0.01244002
## con valores de u y v: -0.5803234 , -0.3839919
```

```r
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
```

El problema de encontrar un mínimo global con esta técnica de gradiente descendente es que se queda estancada en mínimos locales. Esto significa que que encontremos el mínimo global o no depende únicamente del punto de inicio que le damos: si éste está lo suficientemente cerca del mínimo global como para que no haya mínimos locales entre ambos, lo encontrará, pero de haber algún mínimo local, se quedará en el primero que encuentre y será incapaz de salir de ahí.

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 $\eta = 0.1$.

```r
# 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
  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]) - ferror(w[1],w[2])) < umbral &&
        ferror(wnew[1],wnew[2]) < umbral || i==maxiter) {
      mejora <- FALSE
      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])
    }
```

```
    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.

Como vemos, es mucho mejor la técnica del gradiente descendente, ya que éste, para la misma función, con la misma tasa de aprendizaje y con el mismo punto inicial necesita 10 iteraciones para obtener un error de $1.2 * 10^{-15}$ (hecho en el apartado anterior) mientras que ésta en 15 iteraciones sólo llega a un error de 1.1398.

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]))
  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
    # Vamos guardando los valores de error por los que vamos pasando
    valoresError <- rbind(valoresError, c(i, ferror(wnew[1], wnew[2])))

    if (abs(ferror(wnew[1], wnew[2]) - ferror(w[1], w[2])) < umbral ||
        ferror(wnew[1], wnew[2]) < umbral || i==maxiter) {
      mejora <- FALSE
      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])
```

```
    }

    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)
```

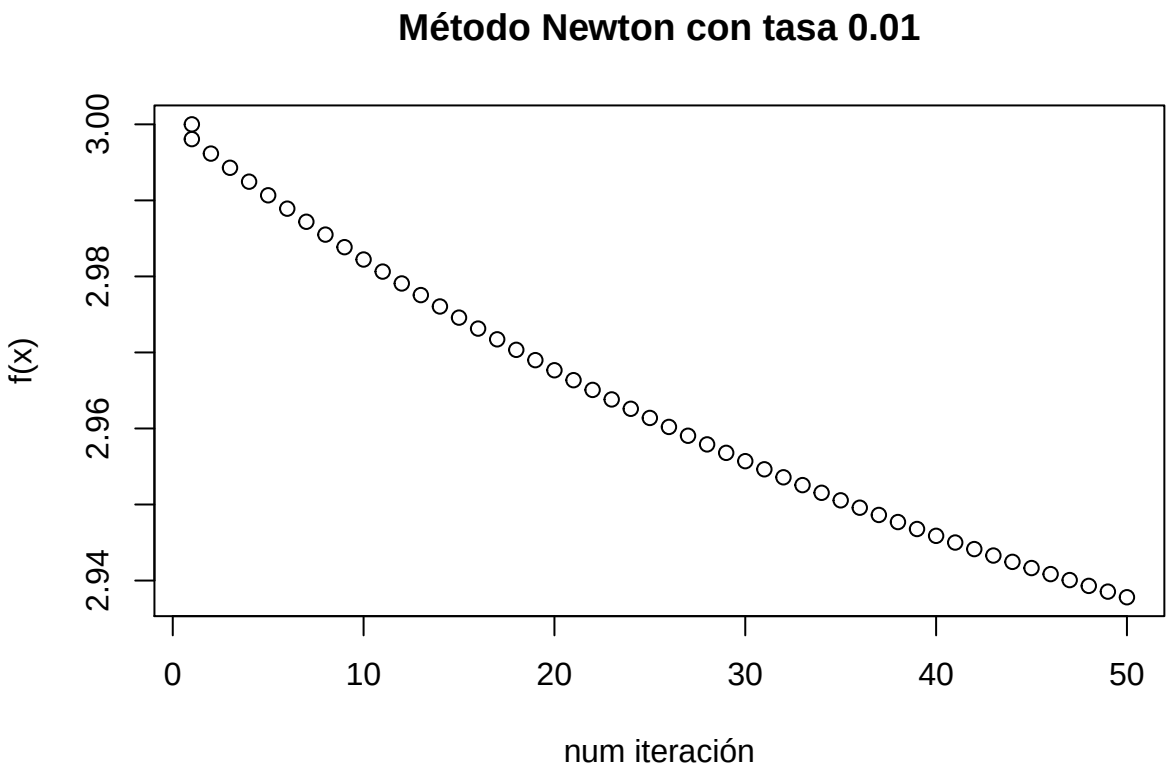
```
## He necesitado 50 iteraciones para llegar al error 2.937804
## con valores de u y v: 0.9803919 , 0.9904148
```

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

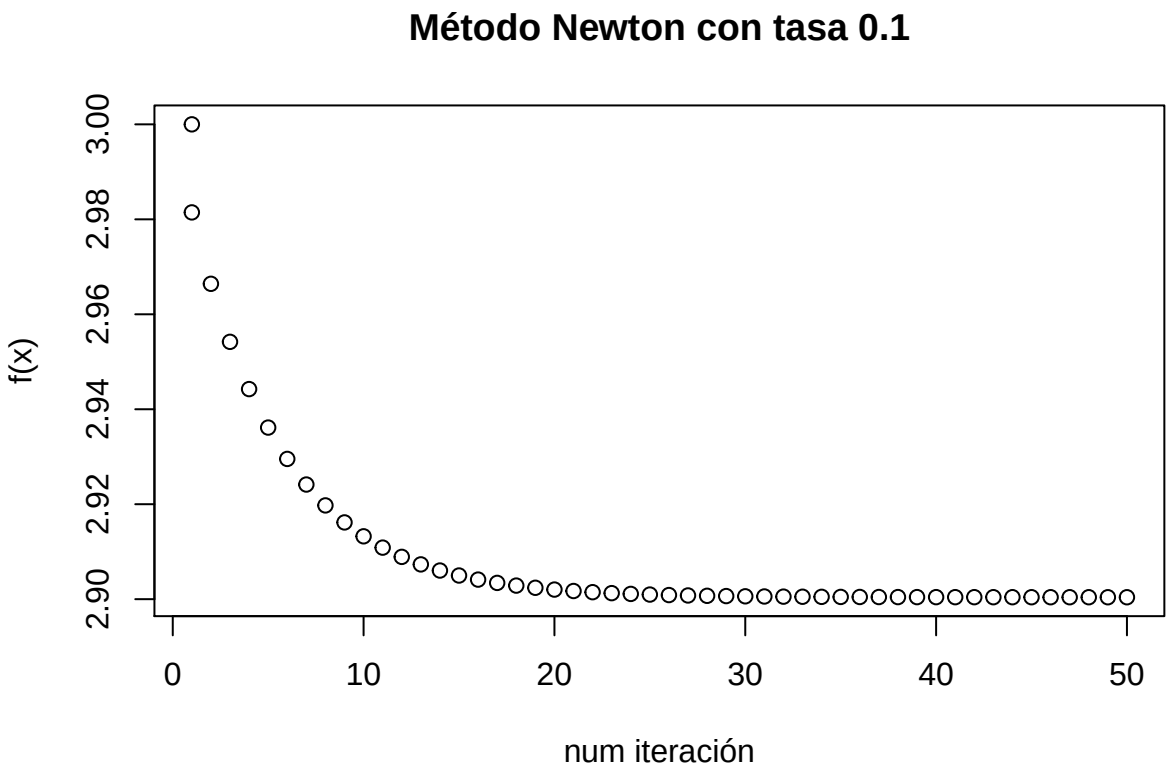
```
## He necesitado 50 iteraciones para llegar al error 2.900408
## con valores de u y v: 0.9494086 , 0.9747153
```

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")
```




```
plot(val2[,1], val2[,2], type="p", xlab="num iteración", ylab="f(x)", main="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.

Como vemos, para tasa 0.01, el método de gradiente descendente converge mucho más rápido a un error bastante más pequeño (0.5 frente a 2.94). En contraposición, con tasa de aprendizaje 0.1, el método de gradiente descendente no converge mientras que el de Newton sí, a error 2.90. Aun así, creo que es mejor el método de gradiente descendente, sólo hay que probar y ajustar bien la tasa de aprendizaje, pero converge más rápido y mejor.

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,...,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))
}
```

Implementamos ahora regresión logística con las condiciones dadas:

```
# 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} usando para ello un número suficientemente grande de nuevas muestras.

He usado un número de 1000 muestras, las he etiquetado con la verdadera función, he obtenido la g aproximada por regresión lineal y las he vuelto a etiquetar con esta g para poder contar los errores en estas 1000 muestras y poder calcular después el porcentaje de error:

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

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

```
g <- -sol[[1]]/sol[[1]][2]
muestra_out <- simula_unif(1000, 2, c(-1,1))
# Etiquetamos con la función original
etiquetas_originales <- unlist(lapply(1:nrow(muestra_out), function(i) {
  p <- muestra_out[i,]
  sign(p[2] - recta[1]*p[1] - recta[2])
}))
# Etiquetamos con la g estimada
```

```
etiquetas_g <- unlist(lapply(1:nrow(muestra_out), function(i) {
  p <- muestra_out[i,]
  sign(p[2] - g[1]*p[1] - g[3])
}))
# Contamos los errores
errores <- cuenta_diferencias(etiquetas_originales, etiquetas_g)
cat("El error Eout es: ", 100*(errores/nrow(muestra)))
```

```
## El error Eout es:  32
```

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

Hacemos un bucle de 1 a 100 generando en cada iteración una recta que divida a los datos distinta, estimamos la g con regresión logística y vamos acumulando los errores para hacer la media.

1) ¿Cuál es el valor de E_{out} para $N = 100$?

```
errores <- 0
iteraciones <- 0
for(i in 1:100) {
  # Generamos una nueva recta que separe
  recta_i <- simula_recta(c(-1,1))
  # Etiquetamos con esta recta
  etiquetas_i <- unlist(lapply(1:nrow(muestra), function(i) {
    p <- muestra[i,]
    sign(p[2] - recta_i[1]*p[1] - recta_i[2])
  }))

  # Estimamos g
  sol <- Regress_LogSGD(cbind(muestra,1), etiquetas_i, c(0,0,0), 0.01, 500, 0.01)
  g <- -sol[[1]]/sol[[1]][2]
  # Acumulamos las iteraciones que tarda en converger
  iteraciones <- iteraciones + sol[[2]]

  # Calculamos las etiquetas originales y las estimadas fuera de la muestra
  etiquetas_originales <- unlist(lapply(1:nrow(muestra_out), function(i) {
    p <- muestra_out[i,]
    sign(p[2] - recta_i[1]*p[1] - recta_i[2])
  }))
  etiquetas_g <- unlist(lapply(1:nrow(muestra_out), function(i) {
    p <- muestra_out[i,]
    sign(p[2] - g[1]*p[1] - g[3])
  }))
  # Contamos errores y acumulamos
  errores <- errores + cuenta_diferencias(etiquetas_originales, etiquetas_g)
}

cat("El número medio de Eout ha sido:", errores/100)
```

```
## El número medio de Eout ha sido: 24.61
```

2) ¿Cuántas épocas tada en promedio RL en converger para $N = 100$, usando todas las condiciones anteriormente especificadas?

En el trozo anterior, dentro del bucle, hemos ido acumulando el número de iteraciones que el método tarda en converger para las 100 iteraciones. Sólo nos queda hacer la media, es decir, dividir por 100:

```
cat("RL tarda en converger", iteraciones/100, "iteraciones en promedio")
```

```
## RL tarda en converger 331.41 iteraciones en promedio
```

```
# Borramos lo que no necesitamos
rm(muestra, muestra_out)
rm(errores, iteraciones)
rm(etiquetas, etiquetas_g, etiquetas_originales)
rm(g, i, recta, recta_i)
rm(val, val2, etiquetas_i, sol, hess)
rm(d11, d12, d22, E, f, gradE, gradF)
```

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)
cat(w)
```

```
## 0.7318291 0.01402067 1.714127
```

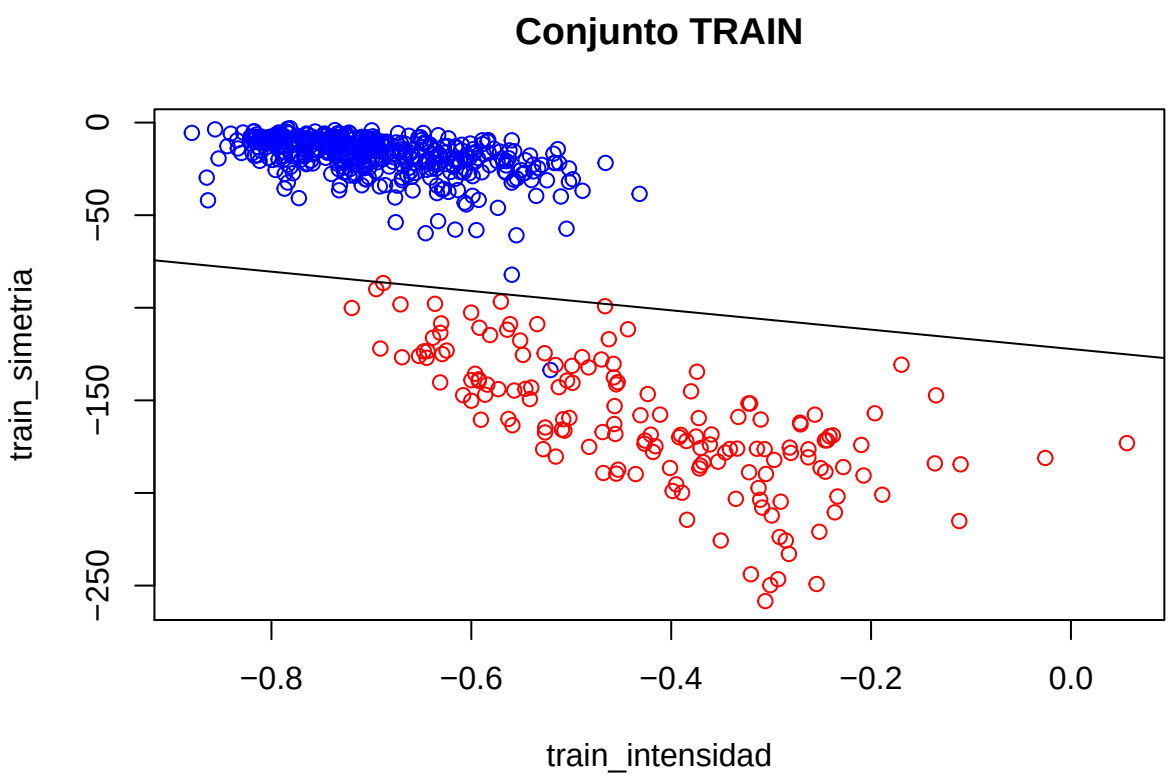
```r
# Le pasamos lo que nos devuelve la regresión lineal al PLA pocket como solución
# inicial
sol <- ajusta_PLA_MOD(datos, etiquetas_train, 100, w)
cat(sol[[1]])
```

```
## 0.7318291 0.01402067 1.714127
```

Como vemos, la solución que nos devuelve la regresión lineal es la misma que la que nos devuelve el PLA pocket, es decir, que el PLA pocket no puede mejorar la solución que da la regresión lineal. Esto es así porque como vemos en el siguiente apartado, el único error no hay forma de arreglarlo ya que no es separable y la regresión ya nos está ofreciendo la mejor solución.

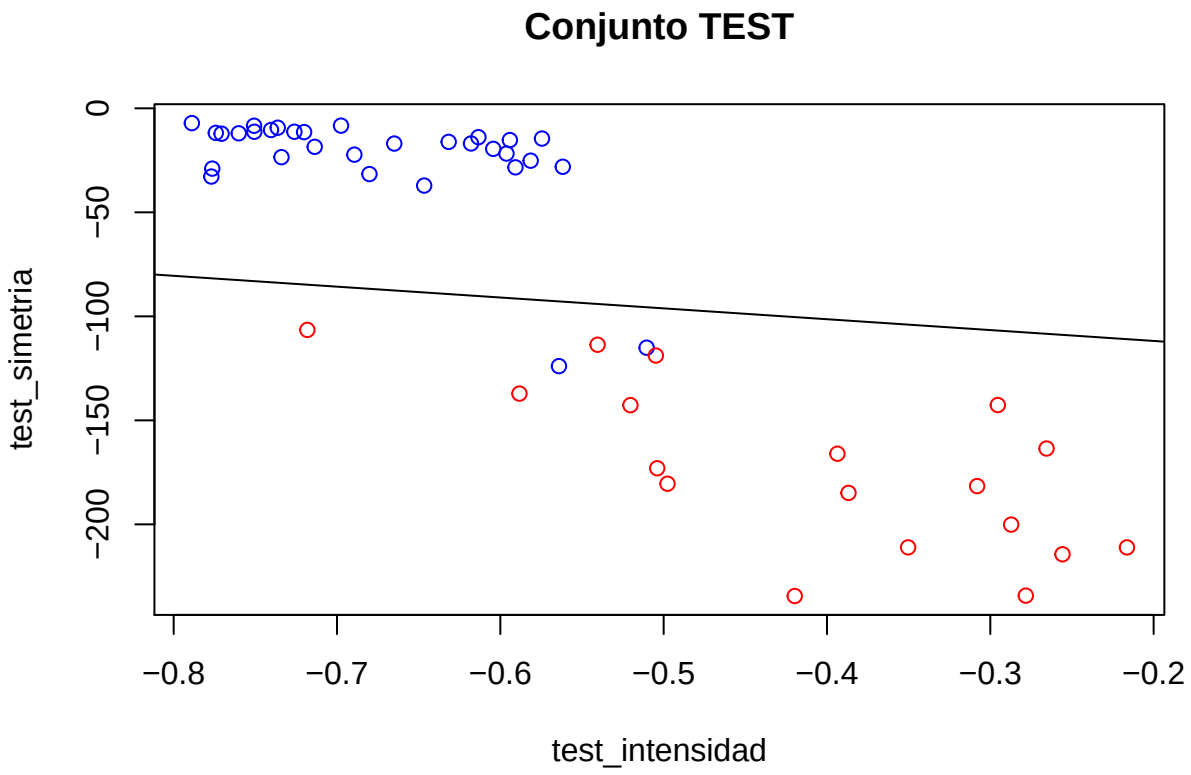
a) Generar gráficos separados (en color) de los datos de entrenamiento y test junto con la función estimada.

```r
# 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).

E_{in} es el error dentro de la muestra (datos de entrenamiento) y E_{test} el error en los datos de test con la recta que hemos obtenido previamente sobre los datos de entrenamiento. En ambos casos lo que hacemos es calcular las nuevas etiquetas que nos da la g (función obtenida de regresión logística + PLA pocket) y contar los errores de estas con las originales.

```
etiquetas_g_in <- unlist(lapply(1:nrow(datos), function(i) {
  p <- datos[i,]
  sign(p[2] - r[1]*p[1] - r[3])
}))
datos_test <- cbind(test_intensidad, test_simetria, 1)
etiquetas_g_test <- unlist(lapply(1:nrow(datos_test), function(i) {
  p <- datos_test[i,]
  sign(p[2] - r[1]*p[1] - r[3])
}))
Ein <- 100*cuenta_diferencias(etiquetas_train, etiquetas_g_in)/nrow(datos)
Etest <- 100*cuenta_diferencias(etiquetas_test, etiquetas_g_test)/nrow(datos_test)
cat("El error en la muestra Ein ha sido", Ein)
```

```
## El error en la muestra Ein ha sido 0.1669449
```

```r
cat("El error en el test Etest ha sido", Etest)
```

```
## El error en el test Etest ha sido 4.081633
```

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?

Empezamos obteniendo la cota basada en E_{in}: la cota de generalización de Vapnik-Chervonenkis para $M = \infty$, como es nuestro caso cuando calculamos la recta que separa los datos para los datos de entrenamiento. Para una tolerancia de $\delta = 0.05$, podemos afirmar que $E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{N} ln \frac{4m_H(2N)}{\delta}}$ con probabilidad mayor o igual $1 - \delta = 0.95$. Tenemos además una cota de $m_H(N)$: $m_H(N) \leq N^{d_{VC}} + 1$, y d_{VC} para este problema, como hemos visto en clase, es 3, con lo que $m_H(2N) \leq (2N)^3 + 1$ con N el número de datos, que en este caso es 599. Con todo esto podemos calcular dicha cota:

```r
# Cota sobre la función de crecimiento
mH_cota <- (2*nrow(datos))^3+1
# Cota basada en Ein
Eout_cota1 <- Ein + sqrt((8/nrow(datos))*log((4*mH_cota)/0.05))
cat("La cota basada en Ein es", Eout_cota1)
```

```
## La cota basada en Ein es 0.7522092
```

La cota basada en E_test es la cota de Hoeffding considerando $M = 1$, pues cuando llegamos al test ya tenemos escogida la h, que es la recta elegida con los datos de entrenamiento. Utilizamos por tanto la cota $P[|E_{test}(g) - E_{out}(g)| > \epsilon] \leq 2Me^{-2\epsilon^2 N}$ y queremos que esta probabilidad sea menor o igual que 0.05, con lo que vamos a obtener ϵ: $2e^{-2\epsilon^2 N} \leq 0.05 \Rightarrow e^{-2\epsilon^2 N} \leq 0.025 \Rightarrow -2\epsilon^2 N \leq ln(0.025) \Rightarrow \epsilon^2 \geq ln(0.025)/(-2N) \Rightarrow \epsilon \geq \sqrt{ln(0.025)/(-2N)}$, donde $N = 49$, el número de datos que tenemos en test, luego $\epsilon \geq 0.1940145$. Tomamos el ϵ más pequeño para obtener la cota más fina, y nos queda que la probabilidad de que $|E_{test} - E_{out}| \geq 0.1940145$ es 0.039682.

La cota basada en E_{test} es mejor ya que es más fina, es decir, nos restringe más el error fuera de la muestra, ya que nos dice que la diferencia entre el error que hemos obtenido en test y el que podemos obtener fuera de la muestra es muy probable (1-0.03968 de probabilidad) que sea muy pequeño (0.19). Como conocemos E_{test}, es fácil saber más o menos el error que tendremos fuera de la muestra.

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

$\Phi_3(x) = (1, x_1, x_2, x_1^2, x_2^2, x_1x_2, x_1^3, x_2^3, x_1x_2^2, x_1^2x_2)$

```r
x1 <- train_intensidad
x2 <- train_simetria
datos <- cbind(1, x1, x2, x1^2, x2^2, x1*x2, x1^3, x2^3, x1*x2^2, x2*x1^2)
etiquetas_g_in <- unlist(lapply(1:nrow(datos), function(i) {
  p <- datos[i,]
  sign(p[2] - r[1]*p[1] - r[3])
}))

x1 <- test_intensidad
x2 <- test_simetria
datos_test <- cbind(1, x1, x2, x1^2, x2^2, x1*x2, x1^3, x2^3, x1*x2^2, x2*x1^2)
```

```
etiquetas_g_test <- unlist(lapply(1:nrow(datos_test), function(i) {
  p <- datos_test[i,]
  sign(p[2] - r[1]*p[1] - r[3])
}))

Ein <- 100*cuenta_diferencias(etiquetas_train, etiquetas_g_in)/nrow(datos)
Etest <- 100*cuenta_diferencias(etiquetas_test, etiquetas_g_test)/nrow(datos_test)
cat("El error en la muestra Ein ha sido", Ein)
```

```
## El error en la muestra Ein ha sido 26.21035
```

```
cat("El error en el test Etest ha sido", Etest)
```

```
## El error en el test Etest ha sido 36.73469
```

```
# Cota sobre la función de crecimiento
mH_cota <- (2*nrow(datos))^3+1
# Cota basada en Ein
Eout_cota1 <- Ein + sqrt((8/nrow(datos))*log((4*mH_cota)/0.05))
cat("La cota basada en Ein es", Eout_cota1)
```

```
## La cota basada en Ein es 26.79561
```

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

No, es demasiado complejo para unos datos que son prácticamente separables y pueden resolverse de forma más cómoda con unas cotas sobre E_{out} buenas con una recta.

```
# Borramos lo que no necesitamos
rm(datos, datos_test, g, muestra_out, r, w)
```

```
## Warning in rm(datos, datos_test, g, muestra_out, r, w): objeto 'g' no
## encontrado
```

```
## Warning in rm(datos, datos_test, g, muestra_out, r, w): objeto
## 'muestra_out' no encontrado
```

```
rm(Ein, Eout_cota1, Etest, etiquetas_g, etiquetas_g_in)
```

```
## Warning in rm(Ein, Eout_cota1, Etest, etiquetas_g, etiquetas_g_in): objeto
## 'etiquetas_g' no encontrado
```

```
rm(etiquetas_g_test, etiquetas_test, etiquetas_train)
rm(lista_test, lista_train, mH_cota, sol, test_intensidad)
rm(test_simetria, train_intensidad, train_simetria, x1, x2)
```

2. SOBREAJUSTE

1. Sobreajuste. Vamos a construir un entorno que nos permita experimentar con los problemas de sobreajuste. Consideremos el espacio de entrada $X = [-1, 1]$ con una densidad de probabilidad uniforme $P(x) = \frac{1}{2}$. Consideramos dos modelos $\mathcal{H}_2$ y $\mathcal{H}_{10}$ representando el conjunto de todos los polinomios de grado 2 y grado 10 respectivamente. La función objetivo es un polinomio de grado Q_f que escribimos como $f(x) = \sum_{q=0}^{Q_f} a_q L_q(x)$, donde $L_q(x)$ son los polinomios de Legendre (ver la relación de ejercicios 2). El conjunto de datos es $D = \{(x_1, y_1), ..., (x_N, y_N)\}$ donde $y_n = f(x_n) + \sigma\epsilon_n$ y las $\{\epsilon_n\}$ son variables aleatorias i.i.d. $\mathcal{N}(0, 1)$ y σ^2 la varianza del ruido.

Comenzamos realizando un experimento donde suponemos que los valores de Q_f, N, σ están especificados. Para ello:

a) Generamos los coeficientes a_q a partir de muestras de una distribución $\mathcal{N}(0, 1)$ y escalamos dichos coeficientes de manera que $E_{a,x}[f^2] = 1$ (Ayuda: dividir los coeficientes por $\sqrt{\sum_{q=0}^{Q_f} \frac{1}{2q+1}}$

Fijamos $N = 10$, $\sigma = 0.2$, $Q_f = 3$ y generamos a_q con la función de la práctica anterior que extraía de una normal y los escalamos como nos pide.

```
# Fijamos parámetros y extraemos los aq
N <- 10
sigma <- 0.2^2
Qf <- 3
aq <- unlist(simula_gauss(N, 1, sigma))
# Escalamos
escalado <- unlist(lapply(0:(N-1), function(x) 1/(2*x+1)))
escalado <- sum(sqrt(escalado))
aq <- aq/escalado
```

b) Generamos un conjunto de datos $x_1, ..., x_N$ muestreando de forma independiente $P(x)$ y los valores $y_n = f(x_n) + \sigma\epsilon_n$

Los datos, como tenemos la probabilidad 1/2, los tenemos que extraer de la uniforme en el rango $[-1, 1]$, que es el espacio que nos han dado previamente. Generamos también y_n en órdenes distintas para hacerlo de forma independiente. Para esto último va a ser necesario primero crear los polinomios de Legendre y multiplicarlos por los coeficientes a_q que acabamos de calcular para obtener $f(x_n)$ y generar también los ϵ_n, extrayéndolos de una normal 0,1.

Vamos a hacer por tanto previamente una función para obtener los polinomios de Legendre de grado k.

```
polLegendre <- function(x, k) {
  L <- vector("numeric", k+1)
  L[1] <- 1
  L[2] <- x

}
```

```
datos <- simula_unif(N, 2, c(-1,1))
epsilon_n <- unlist(simula_gauss(N, 1, 1))
```

Sean g_2 y g_{10} los mejores ajustes a los datos usando $\mathcal{H}_2$ y $\mathcal{H}_{10}$ respectivamente, y sean $E_{out}(g_2)$ y $E_{out}(g_{10})$ sus respectivos errores fuera de la muestra.

a) Calcular g_2 y g_{10}

b) ¿Por qué normalizamos f? (Ayuda: interpretar el significado de σ)

c) ¿Cómo podemos obtener E_{out} analíticamente para una g_{10} dada?

2. Siguiendo con el punto anterior, generar múltiples combinaciones de Q_f, N, σ y para cada combinación de parámetros ejecutar un número grande de experimentos (>1000) calculando en cada caso $E_{out}(g_2)$ y $E_{out}(g_{10})$. Promediar todos los valores de error obtenidos para cada conjunto de hipótesis, es decir $E_{out}((H)_2)$ = promedio sobre experimentos$(E_{out}(g_2))$ y $E_{out}((H)_{10})$ = promedio sobre experimentos$(E_{out}(g_{10}))$.

Definimos una medida de sobreajuste como $E_{out}((H)_{10}) - E_{out}((H)_2)$.

a) Argumentar por qué la medida dada puede medir el sobreajuste.

b) ¿Bajo qué condiciones es esta medida significativamente positiva (i.e. sobreajuste alto) y lo opuesto, significativamente negativa? Usar los valores $Q_f \in \{1, 2, ..., 100\}$, $N \in \{20, 25, ..., 100\}$, $\sigma \in \{0 : 0.05; 0.1; ...; 2\}$. Explicar lo que se observa.

3. Repetir el experimento descrito en los puntos anteriores pero para el caso de clasificación donde la función objetivo es un perceptron ruidoso $f(x) = sign(\sum_{q=1}^{Q_f} a_q L_q(x) + \epsilon)$. Notemos que $a_0 = 0$ y que las a_q deben ser normalizadas para que $E_{a,x}[(\sum_{q=1}^{Q_f} a_q L_q(x))^2] = 1$. En este caso los modelos $\mathcal{H}_2$ y $\mathcal{H}_10$ contienen el signo de los polinomios de segundo y décimo orden respectivamente. (Atención: los datos no serán separables) (Ayuda: para la normalización adaptar la regla dada en el punto anterior).