Trabajo 2 Aprendizaje Automático

Jacinto Carrasco Castillo

30 de marzo de 2016

A - MODELOS LINEALES

Comenzamos estableciendo la semilla y definiendo la función `pause` para interrumpir la ejecución tras cada salida por pantalla.

```
set.seed(314159)
setwd("~/Documentos/Aprendizaje Automático/AA-1516/Trabajo2")
```

```
pause <- function(){
  cat("Pulse cualquier tecla")
  s <- scan()
}
```

Como en la práctica 1 y para facilitar la visualización de los resultados, se incluye un método para dibujar funciones:

```
draw_function <- function(interval_x, interval_y, f, levels = 0, col = 1,
                          add = FALSE){
  x <- seq(interval_x[1],interval_x[2],length=1000)
  y <- seq(interval_y[1],interval_y[2],length=1000)
  z <- outer(x,y, f) # Matriz con los resultados de hacer f(x,y)

  # Levels = 0 porque queremos pintar f(x,y)=0
  contour(x,y,z, levels=0, col = col, add = add)
}

hypplane_to_func <- function( vec, change_signs = FALSE ){
  if(change_signs){
    vec[2:length(vec)] <- -vec[2:length(vec)]
  }
  f <- function(x){
    return( vec[1]*x[1]+vec[2]*x[2]+vec[3]*x[3] )
  }
  return(f)
}

vec3D_to_func2D <- function( vec, change_signs = FALSE ){
  if(change_signs){
    vec[2:length(vec)] <- -vec[2:length(vec)]
  }
  f <- function(x,y){
    return( vec[1]*x+vec[2]*y+vec[3] )
  }
  return(f)
}
```

1.- Gradiente Descendente. Implementar el algoritmo de gradiente descendiente.

```
gradientDescent <- function(fun, v_ini, lr, max_iterations,
                            dif = 0, draw_iterations=FALSE,
                            print_iterations=FALSE){
  # Creación del vector donde guardaremos los valores obtenidos
  values <- vector(mode = "numeric", length = max_iterations+1)

  w <- v_ini   # Solución inicial
  aux <- fun(w)
  values[1] <- aux$value
  grad <- aux$derivative_f

  # Comenzamos el bucle
  iter <- 2
  stopped <- FALSE

  while(iter <= max_iterations+1 && !stopped){
    # Actualización de w
    w <- w - lr * grad

    # Evaluamos la función
    aux <- fun(w)
    values[iter]<- aux$value
    grad <-  aux$derivative_f

    #Comprobamos si la diferencia entre dos iteraciones es menor a un umbral
    if( abs(values[iter-1] - values[iter]) < dif){
      stopped <- TRUE
    }
    if(print_iterations){
      cat("Iteration ",iter-1,"\n")
      print(w)
      print(values[iter])
    }
    iter <- iter +1
  }

  if(draw_iterations){
    plot(1:iter,values[1:iter])
  }

  return(list('min' = w, 'iterations' = iter-2))
}
```

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 $\nu = 0.1$.

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

Debido a que es una función sencilla de derivar, lo incluimos directamente en la función en lugar de utilizar métodos numéricos de derivación:

$$w = (u, v); \quad \nabla E(w) = (\frac{\partial E}{\partial x}, \frac{\partial E}{\partial y}) = 2(ue^v - 2ve^{-u})(e^v + 2ve^{-u}, ue^v - 2e^{-u})$$

```
E_1a <- function(w){
  value <- (w[1]*exp(w[2]) - 2*w[2]*exp(-w[1]))^2
  derivative_f<-c(2*(w[1]*exp(w[2])-2*w[2]*exp(-w[1]))*(exp(w[2])+ 2*w[2]*exp(-w[1])),
                  2*(w[1]*exp(w[2])-2*w[2]*exp(-w[1]))*(w[1]*exp(w[2])- 2*exp(-w[1])))
  return(list('value' = value, 'derivative_f'= derivative_f))
}
```

```
results_1a <- gradientDescent(E_1a, v_ini = c(0,0.1), lr = 0.1, max_iterations = 15)
print(results_1a$iterations)
```

```
## [1] 15
```

```
print(results_1a$min)
```

```
## [1] 0.04864119 0.02621090
```

```
print(E_1a(results_1a$min)$value)
```

```
## [1] 4.814825e-35
```

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

```
print(gradientDescent(E_1a, v_ini = c(0,0.1), lr = 0.1,
                      max_iterations = 100, dif= 10^-14,
                      print_iterations = TRUE))
```

```
## Iteration  1
## [1] 0.05220684 0.02000000
## [1] 0.0002339732
## Iteration  2
## [1] 0.04896965 0.02564430
## [1] 1.971675e-06
## Iteration  3
## [1] 0.04866781 0.02616502
## [1] 1.294731e-08
## Iteration  4
## [1] 0.04864332 0.02620723
## [1] 8.293196e-11
## Iteration  5
## [1] 0.04864136 0.02621061
## [1] 5.301292e-13
## Iteration  6
## [1] 0.04864120 0.02621088
```

```
## [1] 3.388215e-15
## Iteration  7
## [1] 0.04864119 0.02621090
## [1] 2.165482e-17
## $min
## [1] 0.04864119 0.02621090
##
## $iterations
## [1] 7
```

Se comprueba que en la sexta iteración (séptima si contamos la evaluación del vector inicial) se obtiene un valor menor que 10^{-14}

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

Se obtuvo $u = 0.04864120$, $v = 0.02621088$. No es el mínimo esperado, $(0, 0)$, aunque se acerca. Esto se debe a que la función es muy inestable, esto es, para pequeños valores de u y v y al hacer la exponencial de v y $-u$, se obtienen valores muy grandes según el signo de las variables.

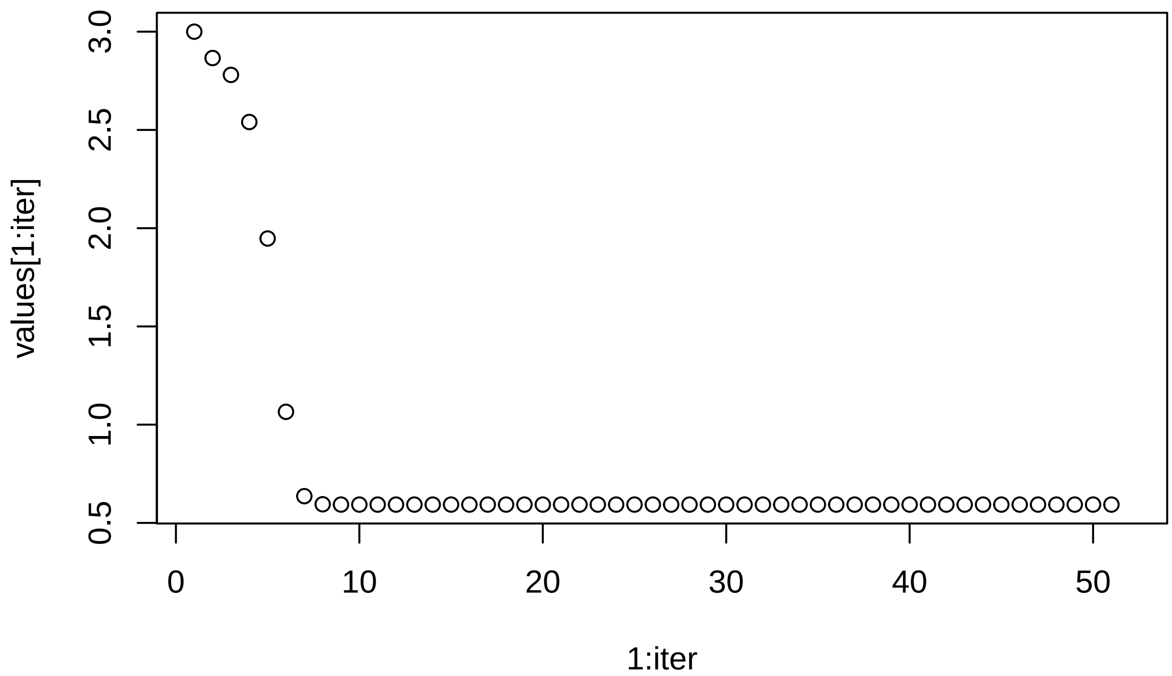
b) Considerar ahora la función $f(x, y) = x^2 + 2y^2 + 2\sin(2\pi x)\sin(2\pi y)$

Implementamos de nuevo la función y sus derivadas:

```r
E_1b <- function(w){
  value <- w[1]^2 + 2*w[2]^2 + 2*sin(2*pi*w[1])*sin(2*pi*w[2])
  derivative_f <- c(2*w[1] + 4*pi*cos(2*pi*w[1])*sin(2*pi*w[2]),
                    4*w[2] + 4*pi*sin(2*pi*w[1])*cos(2*pi*w[2]))
  hessian_f <- matrix(c(2 - 8*pi^2*cos(2*pi*w[1])*sin(2*pi*w[2]),
                        8*pi^2*cos(2*pi*w[1])*cos(2*pi*w[2]),
                        8*pi^2*cos(2*pi*w[1])*cos(2*pi*w[2]),
                        4 - 8*pi^2*sin(2*pi*w[1])*sin(2*pi*w[2])),2,2,TRUE)
  return(list('value' = value, 'derivative_f'= derivative_f, 'hessian_f'=hessian_f))
}
```

1)Usar gradiente descendente para minimizar esta función. Usar como valores iniciales $x_0 = 1, y_0 = 1$, la tasa de aprendizaje $\nu = 0.01$ 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 $\nu = 0.1$, comentar las diferencias.

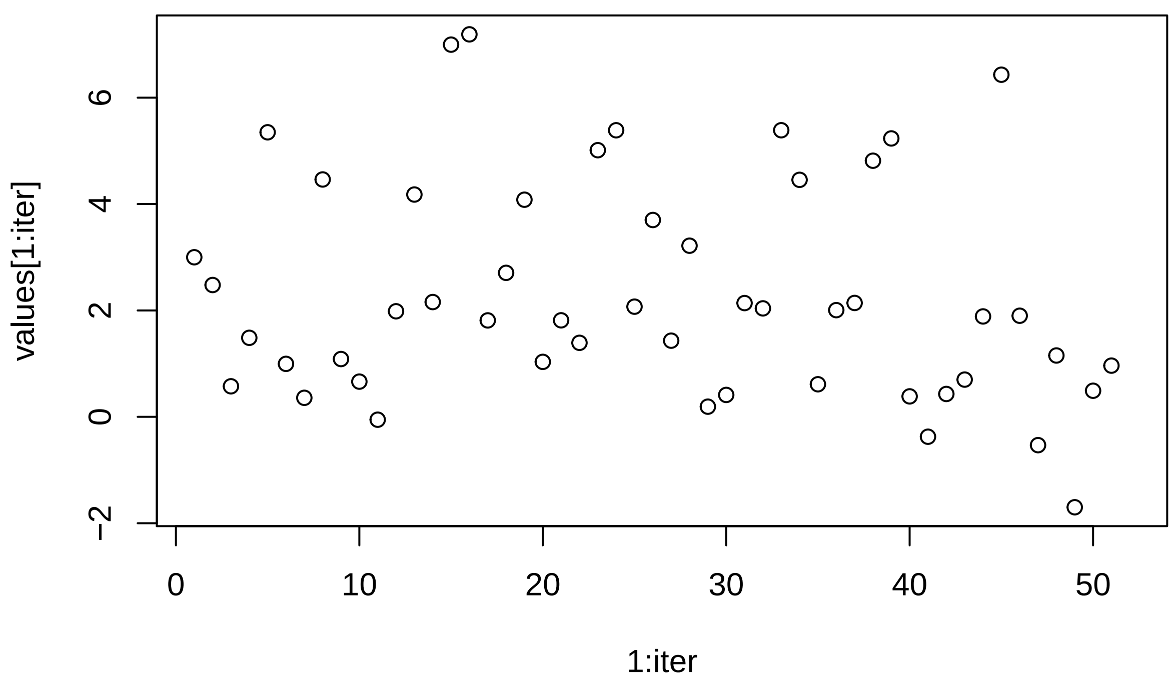
```r
print(gradientDescent(E_1b, v_ini = c(1,1), lr = 0.01,
                      max_iterations = 50, draw_iterations = TRUE))
```




```
## $min
## [1] 1.218070 0.712812
##
## $iterations
## [1] 50
```

Con este valor de la tasa de aprendizaje la función converge a 0.5.

```r
print(gradientDescent(E_1b, v_ini = c(1,1), lr = 0.1,
                      max_iterations = 50, draw_iterations = TRUE))
```



```
## $min
## [1] 0.1987232 0.4571101
##
## $iterations
## [1] 50
```

Con este valor de la tasa de aprendizaje, no existe la convergencia. Esto se debe a que la función tiene varios mínimos y máximos locales y la tasa de aprendizaje hace que el punto vaya de una región a otra, en lugar de, como en el caso anterior o en la primera función, converger hacia el mínimo más cercano.

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?

```r
vectors_ini <- matrix(c(0.1,0.1,1,1,-0.5,-0.5,-1,-1),ncol=2,byrow=TRUE)
vectors_min <- t(apply(vectors_ini,1,
                       function(x) gradientDescent(E_1b, v_ini =x ,
                          lr = 0.01, max_iterations= 100)$min))
print(vectors_min)
```

```
##            [,1]       [,2]
## [1,]  0.2438050 -0.2379258
## [2,]  1.2180703  0.7128120
## [3,] -0.7313775 -0.2378554
## [4,] -1.2180703 -0.7128120
```

```r
t(apply(vectors_min,1, function(x) E_1b(x)$value))
```

```
##           [,1]      [,2]      [,3]      [,4]
## [1,] -1.820079 0.5932694 -1.332481 0.5932694
```

Vemos como el mínimo es de entre los puntos es -1.8 y sin embargo un par de puntos han llegado al mismo mínimo (0.593) y el tercer punto se ha quedado en torno al -1.332. Por tanto, depende del punto inicial, además de la regularidad y concavidad de la función.

2. Coordenada descendente. En este ejercicio comparamos la eficiencia de la técnica de optimización de "coordenada descendente" usando la misma función del ejercicio 1.1a. En cada iteración, tenemos dos pasos a lo largo de dos coordenadas. En el Paso-1 nos movemos a lo largo de la coordenada 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 $\mu = 0.1$.

```r
coordinateDescent <- function(fun, v_ini, lr, max_iterations,
                              dif = 0, draw_iterations=FALSE,
                              print_iterations=FALSE){

  # Creación del vector donde guardaremos los valores obtenidos
```

```r
  values <- vector(mode = "numeric", length = max_iterations+1)

  w <- v_ini   # Solución inicial
  aux <- fun(w)
  values[1] <- aux$value
  grad_u <- aux$derivative_f[1]

  # Comenzamos el bucle
  iter <- 2
  stopped <- FALSE

  while(iter <= max_iterations+1 && !stopped){
    # Primer paso. Movimiento en u
    w[1] <- w[1] - lr * grad_u

    # Segundo paso. Movimiento en v
    grad_v <- fun(w)$derivative_f[2]
    w[2] <- w[2] - lr * grad_v

    # Evaluamos la función
    aux <- fun(w)
    values[iter]<- aux$value
    grad_u <-  aux$derivative_f[1]

    #Comprobamos si la diferencia entre dos iteraciones es menor a un umbral
    if( abs(values[iter-1] - values[iter]) < dif){
      stopped <- TRUE
    }
    if(print_iterations){
      cat("Iteration ",iter-1,"\n")
      print(w)
      print(values[iter])
    }

    iter <- iter +1
  }

  if(draw_iterations){
    plot(1:iter,values[1:iter])
  }

  return(list('min' = w, 'iterations' = iter-2))
}
```

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

```r
min_2a <- coordinateDescent(E_1a, v_ini = c(0,0.1), lr = 0.1,
                            max_iterations = 15)$min
print(E_1a(min_2a)$value)
```

```
## [1] 8.990612e-20
```

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

Obtenemos un error de $8.990612 \cdot 10^{-20}$. Vemos como también se acerca al mínimo. Sin embargo, lo hace más lento que el método del gradiente descendiente, que en las mismas 15 iteraciones llegaba a un valor de $4.814825 \cdot 10^{-35}$. Esto se debe a que con la técnica del gradiente descendiente estamos bajando en ambas coordenadas por la mayor pendiente para cada punto, en lugar de bajar primero por la mayor pendiente para una dimensión, volver a evaluar, y bajar por la mayor pendiente para la otra dimensión.

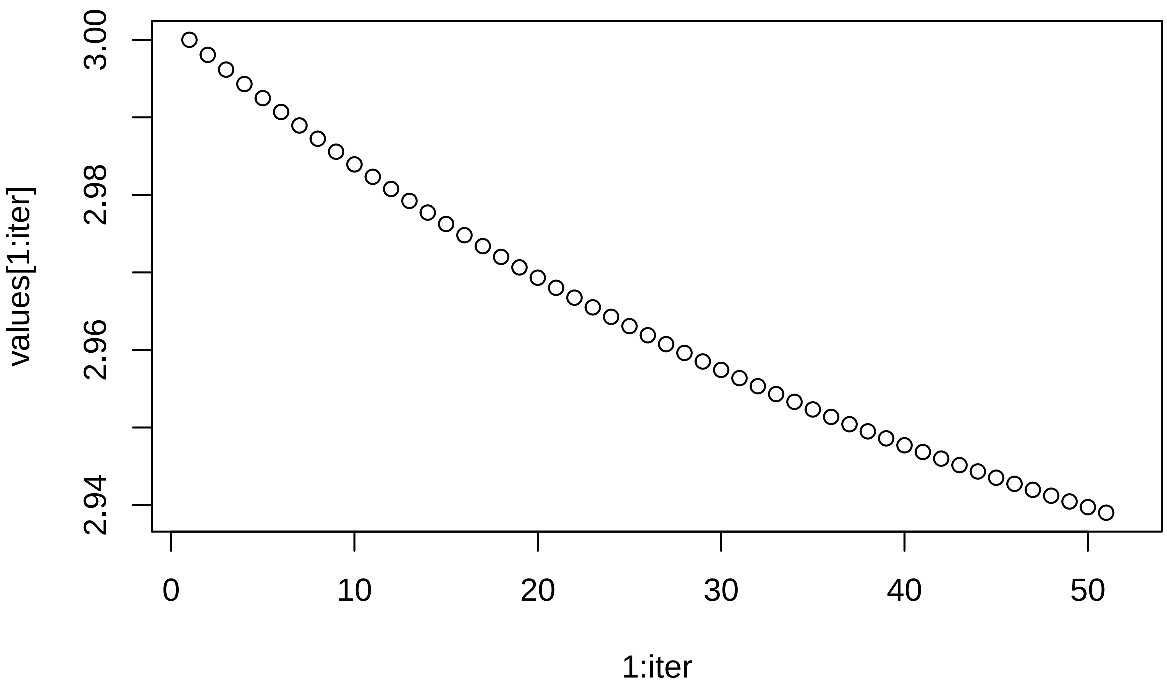
3.- Método de Newton: Implementar el algoritmo de minimización de Newton y aplicarlo a la función $f(x, y)$ dada en el ejercicio 1b. Desarrolle los mismos experimentos usando los mismos puntos de inicio.

```
newtonMethod <- function(fun, v_ini, lr, max_iterations, dif = 0,
                         draw_iterations=FALSE,
                         print_iterations=FALSE){
  # Creación del vector donde guardaremos los valores obtenidos
  values <- vector(mode = "numeric", length = max_iterations+1)

  w <- v_ini   # Solución inicial
  aux <- fun(w)
  values[1] <- aux$value
  increment <- solve(aux$hessian)%*%aux$derivative_f

  # Comenzamos el bucle
  iter <- 2
  stopped <- FALSE

  while(iter <= max_iterations+1 && !stopped){
    # Actualización de w
    w <- w - lr * increment

    # Evaluamos la función
    aux <- fun(w)
    values[iter]<- aux$value
    increment <- solve(aux$hessian)%*%aux$derivative_f

    #Comprobamos si la diferencia entre dos iteraciones es menor a un umbral
    if( abs(values[iter-1] - values[iter]) < dif){
      stopped <- TRUE
    }
    if(print_iterations){
      cat("Iteration ",iter-1,"\n")
      print(w)
      print(values[iter])
    }
    iter <- iter +1
  }

  if(draw_iterations){
    plot(1:iter,values[1:iter])
  }
```

```r
  return(list('min' = w, 'iterations' = iter-2))
}
```

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

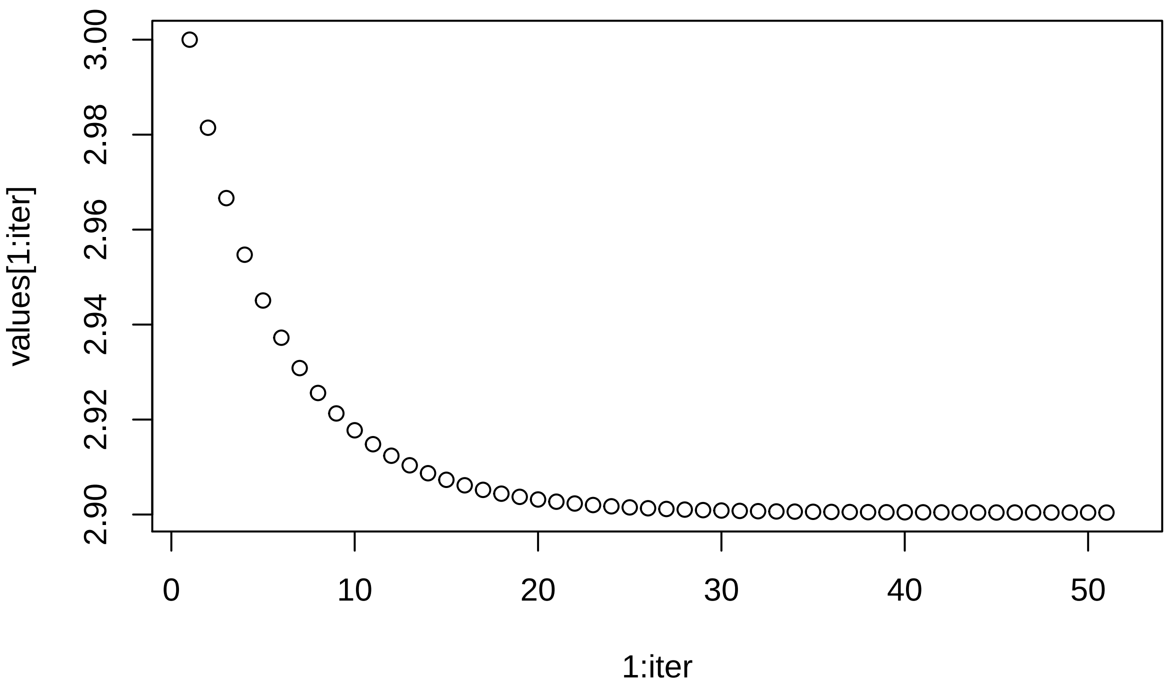
```r
print(newtonMethod(E_1b, v_ini = c(1,1), lr = 0.01,
                   max_iterations =50, draw_iterations = TRUE))
```




```
## $min
##           [,1]
## [1,] 0.9803694
## [2,] 0.9909248
##
## $iterations
## [1] 50
```

Con este valor de la tasa de aprendizaje la función converge a 2.94.

```r
print(newtonMethod(E_1b, v_ini = c(1,1), lr = 0.1,
                   max_iterations = 50, draw_iterations = TRUE))
```




```
## $min
##           [,1]
## [1,] 0.9494082
## [2,] 0.9749582
##
## $iterations
## [1] 50
```

Con esta tasa de aprendizaje hay convergencia a 2.90. Se comprueba que con este método es más difícil no quedarse en mínimos locales.

```
vectors_ini <- matrix(c(0.1,0.1,1,1,-0.5,-0.5,-1,-1),ncol=2,byrow=TRUE)
vectors_min <- t(apply(vectors_ini,1, function(x) newtonMethod(E_1b,
                        v_ini =x , lr = 0.01, max_iterations= 100)$min))
print(vectors_min)
```

```
##             [,1]        [,2]
## [1,]  0.02869492  0.02194543
## [2,]  0.96827827  0.98555464
## [3,] -0.48429794 -0.49208508
## [4,] -0.96834596 -0.98346603
```

```
t(apply(vectors_min,1, function(x) E_1b(x)$value))
```

```
##            [,1]     [,2]      [,3]     [,4]
## [1,] 0.05108201 2.916091 0.7286328 2.913082
```

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.

La conclusión que se puede obtener es que este método garantiza también para una mayor tasa de aprendizaje la convergencia a un mínimo. Sin embargo, el hecho de que tenga en cuenta la matriz hessiana hace que el punto se mueva según la forma de la función, no sólo con respecto a la máxima pendiente.

4. Regresión Logística: En este ejercicio crearemos nuestra propia función objetivo f (probabilidad en este caso) y nuestro conjunto de datos $\mathcal{D}$ para ver cómo funciona la 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* $\mathcal{X} = [-1,1] \times [-1,1]$ *con probabilidad uniforme de elegir cada* $x \in \mathcal{X}$. *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 $\mathcal{X}$ y evaluar las respuestas de todos ellos $\{y_n\}$ respecto de la frontera elegida.

Para este apartado usaré las funciones de la práctica 1 `simula_unif` y `simula_recta`:

```r
simula_unif <- function(N, dim, rango){
  # Tomamos N*dim muestras de una uniforme de rango dado
  lista <- matrix(runif(N*dim, min =  rango[1], max = rango[2]), N, dim)
  return(lista)
}

simula_recta <- function(intervalo){
  puntos <- simula_unif(2,2,intervalo)
  a <- (puntos[2,2]-puntos[1,2])/(puntos[2,1]-puntos[1,1])
  b <- puntos[1,2] - a * puntos[1,1]

  f <- vec3D_to_func2D(c(a,1,b), change_signs=TRUE)
  return(f)
}
```

```r
N <- 100
recta_4a <- simula_recta(c(-1,1))
muestra_4a <- simula_unif(N, 2, c(-1, 1))
etiquetas_4a <- apply(muestra_4a, 1, function(X) sign(recta_4a(X[1],X[2])))
```

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

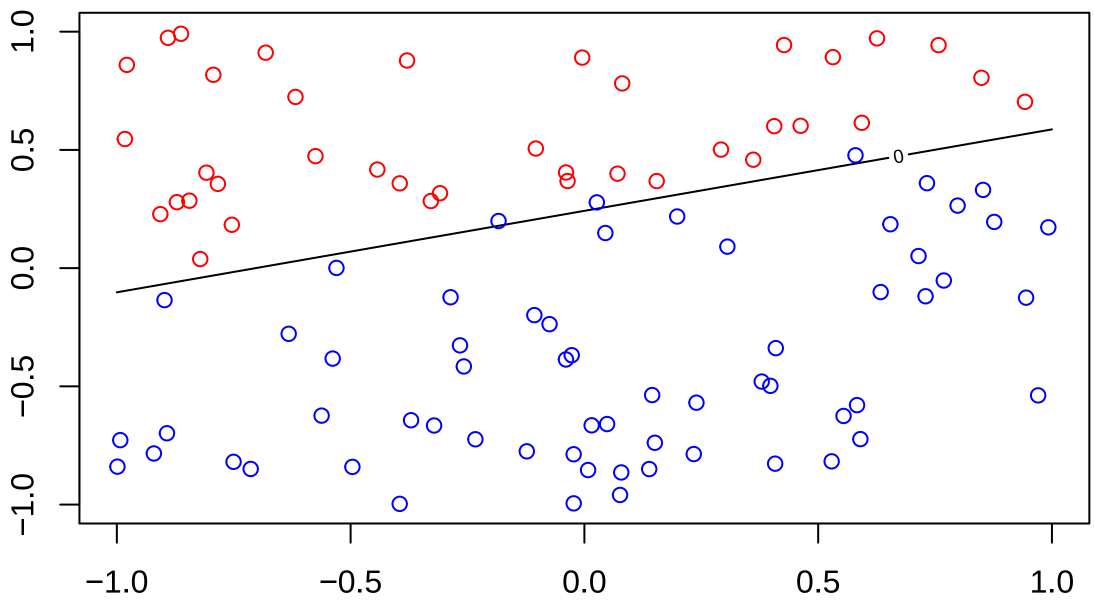
- Inicializar el vector de pesos con valores 0.
- Parar el algoritmo cuando $\|w(t-1) - w(t)\| < 0.01$, donde $w(t)$ denota el vector de pesos al final de la época t. Una época es un pase completo a través de los N datos.
- Aplicar una permutación aleatoria de $1, 2, \ldots, N$ a los datos antes de usarlos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

Implementación del descenso del gradiente estocástico:

```r
norm_v <- function(x) sqrt(sum(x^2))

SGD <- function(datos, valores, max_iter, vini= c(0,0,0),
                lr = 0.01, draw_iterations = FALSE){
  sol <- vini
  iter <- 0
  stopped <- FALSE

  while ( iter < max_iter && !stopped){
    permutation = sample(nrow(datos))
    prev_sol <- sol

    for( inner_iter in permutation ){
      # Añadimos coeficiente independiente
      x <- c(datos[inner_iter,],1)
      y <- valores[inner_iter]

      gr <- lr*y*x/(1+exp(y*(sol%*%x)))
      sol <- sol + gr
    }

    if(norm_v(sol - prev_sol) < 0.01)
      stopped = TRUE
    if(draw_iterations){
      draw_function(c(-1,1), c(-1,1), vec3D_to_func2D(sol),
                    col = 4)
      title(main=paste("Iteración ", iter,sep=" "))
      points(datos[,1], datos[,2], col = (valores+2))
      Sys.sleep(0.5)
    }

    iter <- iter+1
  }

  #Normalizamos la solución.
  sol <- sol/sol[length(sol)-1]

  #Devolvemos el hiperplano obtenido y el número de iteraciones realizadas
  return( list( hyperplane = sol, iterations = iter))
}
```

Cálculo de la función g mediante la regresión logística:

```r
sol_4a <- SGD(muestra_4a, etiquetas_4a, 15000)
g_4a <- vec3D_to_func2D(sol_4a$hyperplane)
draw_function(c(-1,1),c(-1,1), g_4a)
points(muestra_4a[,1],muestra_4a[,2],col=etiquetas_4a+3)
```




```r
print(sol_4a$iterations)
```

```
## [1] 330
```

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.

Generamos una muestra aleatoria, la etiquetamos según la función aleatoria y según la función obtenida de la regresión logística:

```r
  muestra_4a_out <- simula_unif(N, 2, c(-1,1))
  etiquetas_4a_out <- apply(muestra_4a_out, 1,
                            function(X) sign(recta_4a(X[1],X[2])))

  # Etiquetado según la regresión
  rl_etiqueta_4a_out <- apply(muestra_4a_out, 1, function(X) sign(g_4a(X[1],X[2])))

  # Cálculo del error fuera de la muestra usada para la regresión
  E_out <- sum(etiquetas_4a_out != rl_etiqueta_4a_out)/N*100
```

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

```r
E_out_4c <- vector(mode="numeric", length = N)

for( i in 1:100){
  recta_4c <- simula_recta(c(-1,1))
  muestra_4c <- simula_unif(N, 2, c(-1, 1))
  etiquetas_4c <- apply(muestra_4c, 1, function(X) sign(recta_4c(X[1],X[2])))

  sol_4c <- SGD(muestra_4c, etiquetas_4c, 1000)$hyperplane
  g_4c <- vec3D_to_func2D(sol_4c)

  muestra_4c_out <- simula_unif(N, 2, c(-1,1))
```

```r
  etiquetas_4c_out <- apply(muestra_4c_out, 1,
                            function(X) sign(recta_4c(X[1],X[2])))

  # Etiquetado según la regresión
  rl_etiquetas_4c_out <- apply(muestra_4c_out, 1,
                               function(X) sign(g_4c(X[1],X[2])))

  # Cálculo del error fuera de la muestra usada para la regresión
  E_out_4c[i] <- sum(etiquetas_4c_out != rl_etiquetas_4c_out)/N*100
}

cat("Media E_out: ", mean(E_out_4c))
```

```
## Media E_out:  97.88
```

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.

Lectura de datos y dotación de forma para datos de entrenamiento

```r
datos_train <- scan("datos/zip.train",sep=" ")
datos_train <- matrix(datos_train, ncol=257, byrow=T)
datos_train <- datos_train[datos_train[,1] == 1 | datos_train[,1]==5,]

number_train <- datos_train[,1]
pixels_train <- array(t(datos_train[,2:257]), dim=c(16,16,nrow(datos_train)))
```

Lectura de datos y dotación de forma para datos de test

```r
datos_test <- scan("datos/zip.test",sep=" ")
datos_test <- matrix(datos_test,ncol=257,byrow=T)
datos_test <- datos_test[datos_test[,1] == 1 | datos_test[,1]==5,]

number_test <- datos_test[,1]
pixels_test <- array(t(datos_test[,2:257]), dim=c(16,16,nrow(datos_test)))
```

Cálculo de la simetría vertical y la media de la intensidad.

```r
simetria_vertical <- function(M){
  -sum(apply(M, 1, function(x) sum(abs(x-x[length(x):1]))))
}


means_train <- apply(pixels_train, 3, mean)
means_test <- apply(pixels_test, 3, mean)

sim_vertical_train <- apply(pixels_train,3,simetria_vertical)
sim_vertical_test <- apply(pixels_test,3,simetria_vertical)
```

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.

Este problema consiste en, a partir de los datos de *train*, obtener una función mediante regresión lineal y PLA que nos separe los datos, y ver su tasa de acierto en el conjunto de test. Incluimos las funciones de la primera práctica para hacer regresión lineal y el algoritmo PLA modificado:

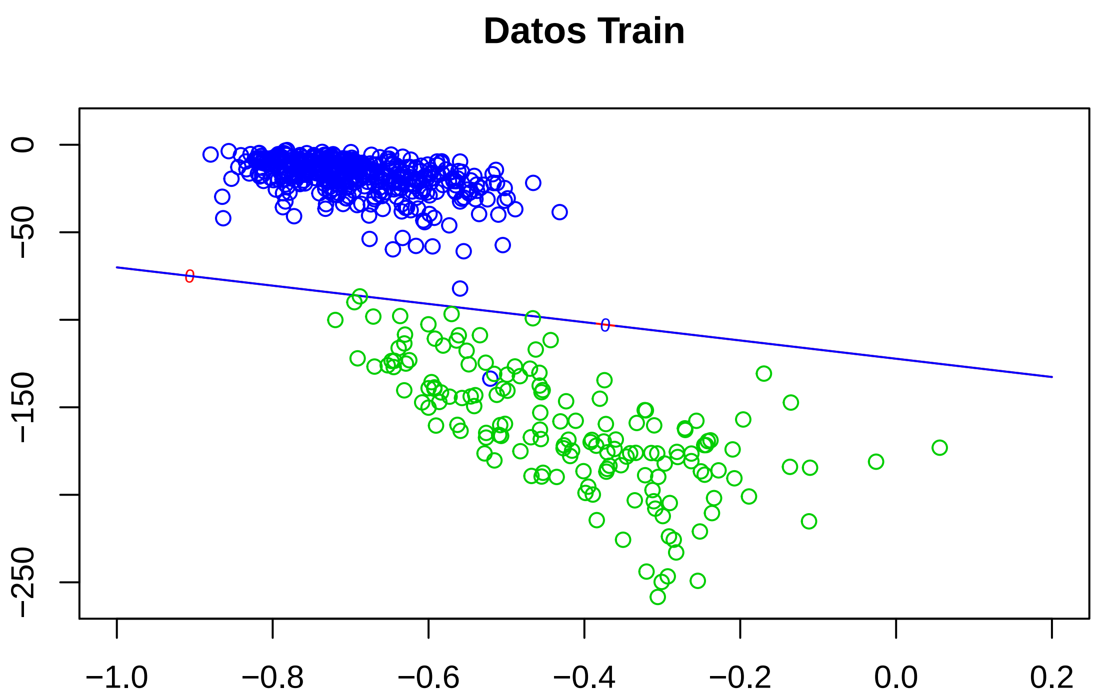
```
count_errors <- function(f, datos, label){
  #Conteo de errores
  signs <- apply(datos, 1, function(x) return(sign(f(c(x,1)))))
  return( sum(signs != label) )
}

ajusta_PLA_MOD <- function(datos, label, max_iter, vini,
                           draw_iterations = FALSE){
  sol <- vini
  iter <- 0
  changed <- TRUE
  current_sol <- sol

  while ( iter < max_iter && changed){
    # Comprobaremos en cada vuelta si hemos hecho algún cambio
    current_errors <- count_errors(hypplane_to_func(sol), datos, label )
    changed <- FALSE

    for( inner_iter in 1:nrow(datos) ){
      # Añadimos coeficiente independiente
      x <- c(datos[inner_iter,],1)

      if( sign(crossprod(x,current_sol)) != label[inner_iter]){
        current_sol <- current_sol + label[inner_iter] * x
        changed <- TRUE
      }
    }

    if(draw_iterations){
      draw_function(c(-50,50), c(-50,50),vec3D_to_func2D(sol), col = 4)
      draw_function(c(-50,50), c(-50,50),vec3D_to_func2D(current_sol),
                    col = 5, add=TRUE)
      points(lista_unif[,1], lista_unif[,2], col = (label+3))
      title(main=paste("Iteración ", iter,sep=" "))
      Sys.sleep(0.5)
    }

    if( current_errors >= count_errors(hypplane_to_func(current_sol),
                                       datos, label )){
      sol <- current_sol
    }

    iter <- iter+1
  }
  sol <- sol/sol[length(sol)-1]
```

```r
  return( list( hyperplane = sol, iterations = iter))
}

pseudoinv <- function(X){
  desc <- svd(X)
  d <- round(desc$d, digits=5)
  d[abs(d)>0] <- 1/d[abs(d)>0]
  pseudo_inv <- desc$v %*% diag(d) %*% t(desc$u)
  return(pseudo_inv)
}

regress_lin <- function(datos, label){
  pseudo_inv <- pseudoinv(t(datos)%*%datos)
  return(pseudo_inv%*%t(datos)%*%label)
}
```

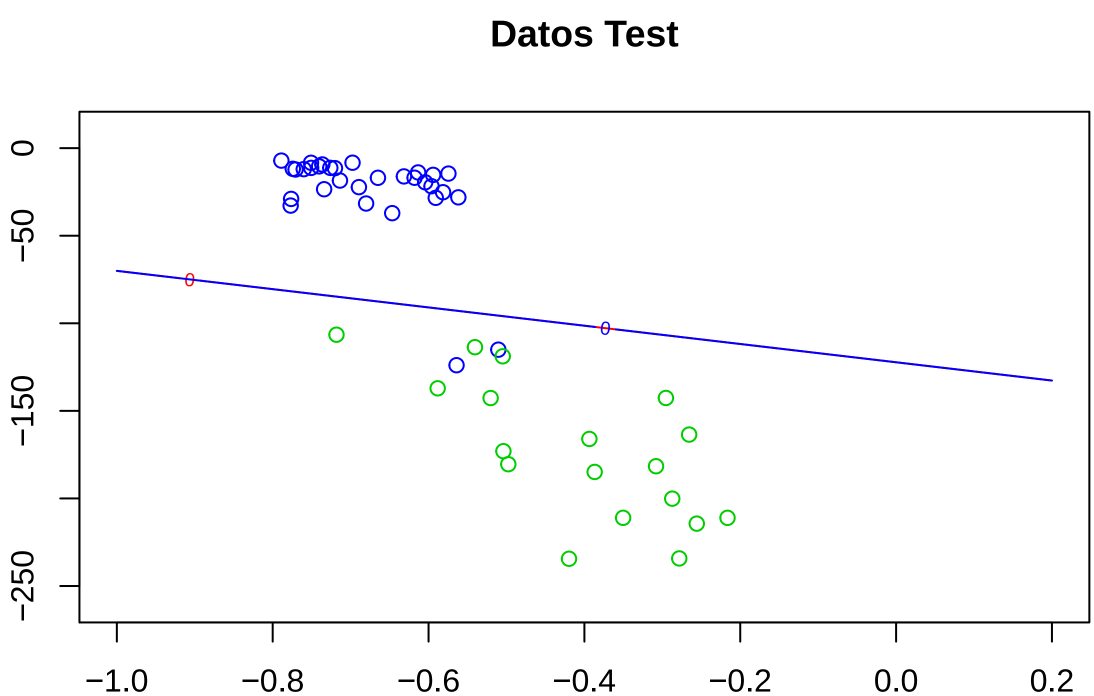
```r
label_5 <- numeric(length(number_train))
label_5[number_train==1] <- -1
label_5[number_train==5] <- 1

coefs_rl_5 <- regress_lin(cbind(means_train,sim_vertical_train,
                                rep(1,length(means_train))),
                          label_5)
coefs_5 <- ajusta_PLA_MOD(cbind(means_train,sim_vertical_train),
                          label_5, 100, coefs_rl_5)$hyperplane
```

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

```r
n_train_colors <- numeric(length(number_train))
n_train_colors[number_train==1] <- 4
n_train_colors[number_train==5] <- 3
data_list_train <- list('number'=number_train, 'pixels'=pixels_train, 'mean'=means_train,
                  'v_symmetry'=sim_vertical_train, 'colors'=n_train_colors )

draw_function(c(-1,0.2),c(-260,10),vec3D_to_func2D(coefs_rl_5), col = 2)
draw_function(c(-1,0.2),c(-260,10),vec3D_to_func2D(coefs_5), col=4, add=TRUE)
points(data_list_train$mean, data_list_train$v_symmetry, col = data_list_train$colors )
title(main="Datos Train")
```




```r
n_test_colors <- numeric(length(number_test))
n_test_colors[which(number_test==1)] <- 4
n_test_colors[which(number_test==5)] <- 3
data_list_test <- list('number'=number_test, 'pixels'=pixels_test,
                       'mean'=means_test,'v_symmetry'=sim_vertical_test,
                       'colors'=n_test_colors )
draw_function(c(-1,0.2),c(-260,10),vec3D_to_func2D(coefs_rl_5), col = 2)
draw_function(c(-1,0.2),c(-260,10),vec3D_to_func2D(coefs_5), col=4, add=TRUE)
points(data_list_test$mean, data_list_test$v_symmetry, col = data_list_test$colors )
title(main="Datos Test")
```



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

```
E_in_5b = count_errors(hypplane_to_func(coefs_5,TRUE),
                       cbind(means_train, sim_vertical_train),
                       label_5)/length(number_train)

label_test_5 <- numeric(length(number_test))
label_test_5[number_test==1] <- -1
label_test_5[number_test==5] <- 1

E_test_5b = count_errors(hypplane_to_func(coefs_5,TRUE),
                         cbind(means_test, sim_vertical_test),
                         label_test_5)/length(number_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?

Para obtener la cota basada en E_{in} usamos la desigualdad de Vapnik-Chervonenkis generalizada:

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{N} \log \frac{4m_{\mathcal{H}}(2N)}{\delta}}$$

Podemos, por mayor simplicidad, acotar $m_{\mathcal{H}}(2N)$ por $(2N)^{d_{VC}}$. Como en este caso d_{VC} es 3,

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{N} \log \frac{4(2N)^3}{\delta}}$$

$N = 599$, luego

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{599} \log \frac{4(2 \cdot 599)^3}{0.05}}$$
$$E_{out}(g) \leq 0.0951586 + 0.5852643 = 0.6804229$$

Para obtener la cota basada en E_{test} usamos la desigualdad de Hoeffding con una hipótesis, que será la producida por los datos de entrenamiento:

$$\mathbb{P}(|E_{in}(g)E_{out}(g)| \geq \varepsilon) \leq 2e^{-2\varepsilon^2 N}$$

Queremos hallar el ε tal que la probabilidad de que estén a una distancia mayor que ε sea menor que 0.05, luego despejamos:

$$\mathbb{P}(|E_{test}(g)E_{out}(g)| \geq \varepsilon) \leq 2e^{-2\varepsilon^2 N} = 0.05$$

Despejando llegamos a $\varepsilon = 0.194$. Por tanto, con probabilidad 0.95, $|E_{test}(g)E_{out}(g)| \leq 0.194$. Esto significa que $E_{out}(g) \leq E_{test}(g) + 0.194 = 0.2960553$.

Llegamos por tanto a una mejor cota de E_{out}

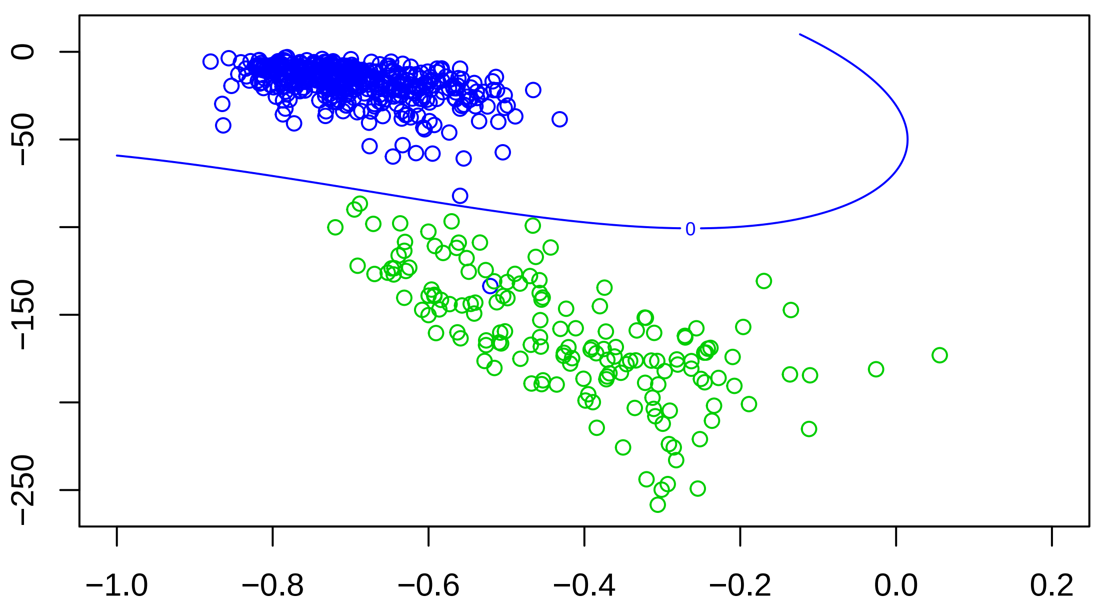
d) Repetir los puntos anteriores pero usando una transformación polinómica de tercer orden ($\Phi_3(x)$ en las transparencias de teoría).

Denotamos por $\Phi_3(x)$ a la transformación $\Phi_3(x) = 1, x_1, x_2, x_1^2, x_2^2, x_1x_2, x_1^3, x_2^3, x_1^2x_2, x_1x_2^2)$. Tenemos por tanto que realizar la regresión lineal para este modelo y ver el error dentro y fuera de la muestra:

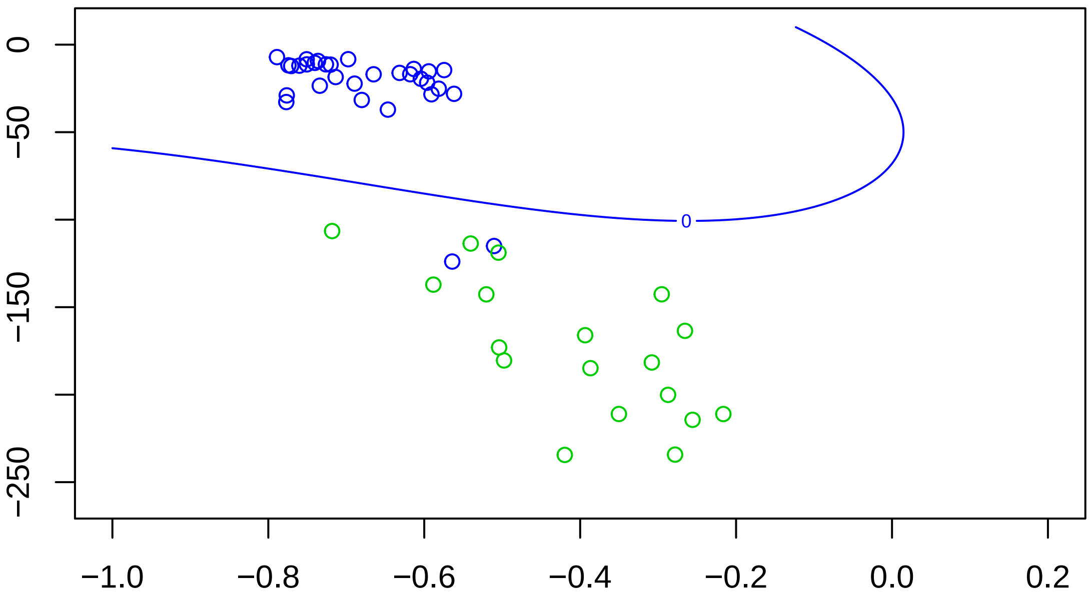
```r
data_transf_5d <- cbind(means_train,sim_vertical_train,
                        means_train^2,sim_vertical_train^2,
                        means_train*sim_vertical_train,
                        means_train^3,sim_vertical_train^3,
                        means_train^2*sim_vertical_train,
                        means_train*sim_vertical_train^2)

coefs_rl_5d <- regress_lin(cbind(data_transf_5d,
                                 rep(1,length(number_train))), label_5)

estimated_f_5d <- function(x,y){
  coefs_rl_5d[1]*x + coefs_rl_5d[2]*y + coefs_rl_5d[3]*x^2 +
    coefs_rl_5d[4]*y^2 + coefs_rl_5d[5]*x*y +
    coefs_rl_5d[6]*x^3 + coefs_rl_5d[7]*y^3 +
    coefs_rl_5d[8]* x^2*y + coefs_rl_5d[9]*x*y^2 + coefs_rl_5d[10]
}
```

Mostramos la gráfica obtenida con la regresión lineal

```r
draw_function(c(-1,0.2),c(-260,10),estimated_f_5d, col=4)
points(data_list_train$mean, data_list_train$v_symmetry,
       col = data_list_train$colors)
```




```r
draw_function(c(-1,0.2),c(-260,10),estimated_f_5d, col=4)
points(data_list_test$mean, data_list_test$v_symmetry,
       col = data_list_test$colors)
```



Contamos los errores en los datos de entrenamiento y en los datos de test:

```
E_in_5d = count_errors(hypplane_to_func(coefs_rl_5d,TRUE),
                       data_transf_5d,
                       label_5)/length(number_train)

data_transf_test_5d <- cbind(means_test, sim_vertical_test,
                        means_test^2,sim_vertical_test^2,
                        means_test*sim_vertical_test,
                        means_test^3,sim_vertical_test^3,
                        means_test^2*sim_vertical_test,means_test*sim_vertical_test^2)

E_test_5d = count_errors(hypplane_to_func(coefs_rl_5d,TRUE),
                         data_transf_test_5d,
                         label_test_5)/length(number_test)
```

Ahora calculamos las cotas para E_{out} de la misma manera que antes:

La dimensión es $\frac{Q(Q+3)}{2}+1$, para nuestro caso, $d_{VC}=10$ y entonces, acotando $m_{\mathcal{H}}(2N)$ por $(2N)^{d_{VC}}$:

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{N}\log\frac{4((2N)^{10}+1)}{\delta}}$$

$N = 599$, luego

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{599}\log\frac{4((2\cdot 599)^{10}+1)}{0.05}}$$
$$E_{out}(g) \leq 0.1285476 + 1.002608 = 1.131156$$

Pero esta cota sobrepasa 1, con lo que no nos aporta información. Miraremos ahora con la cota obtenida con E_{test}. Volvemos a usar la desigualdad de Hoeffding:

$$\mathbb{P}(|E_{in}(g)E_{out}(g)| \geq \varepsilon) \leq 2e^{-2\varepsilon^2 N}$$

Despejamos de nuevo ε y llegamos a $\varepsilon = 0.194$. Por tanto, con probabilidad 0.95, $|E_{test}(g)E_{out}(g)| \leq 0.194$. Esto significa que $E_{out}(g) \leq E_{test}(g) + 0.194 = 0.3368571$.

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

No usaría la transformación polinómica, pues añade demasiada complejidad a un modelo que, a la vista de los resultados, se puede resolver con el modelo lineal. Como vemos, las cotas son mayores para el caso con la transformación cúbica e incluso para hallar la cota sólo en función del error dentro de la muestra, nos da un error mayor que 1.

B - SOBREAJUSTE

1.-Sobreajuste.

Vamos a construir un entorno que nos permita experimentar con los problemas de sobreajuste. Consideremos el espacio de entrada $\mathcal{X} = [-1, 1]$ con una densidad de probabilidad uniforme, $f_U(x) = \frac{1}{2}$. 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 $\mathcal{D} = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ donde $y_n = f(x_n) + \sigma\varepsilon_n$ y las ε_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:

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

* Generamos un conjunto de datos $x_1, \ldots, x_N$ muestreando de forma independiente $\mathbb{P}(x)$ y los valores $y_n = f(x_n) + \sigma\varepsilon_n$.

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}.

Comenzamos definiendo una función para obtener los multiplicadores de Lagrange:

```
lagrange_poly <- function(x,k){
  if (k ==0){
    return( 1 )
  }
  else if (k ==1){
    return( c(1,x) )
  }

  l_poly <- vector(mode="numeric", length = k+1)
  l_poly[1] <- 1
  l_poly[2] <- x

  for( i in 2:k){
    l_poly[i+1] <- (2*i-1)/i*x*l_poly[i] -
      (i-1)/i*l_poly[i-1]
  }
```

```r
  return( l_poly )
}
```

Definimos los valores usados para el experimento.

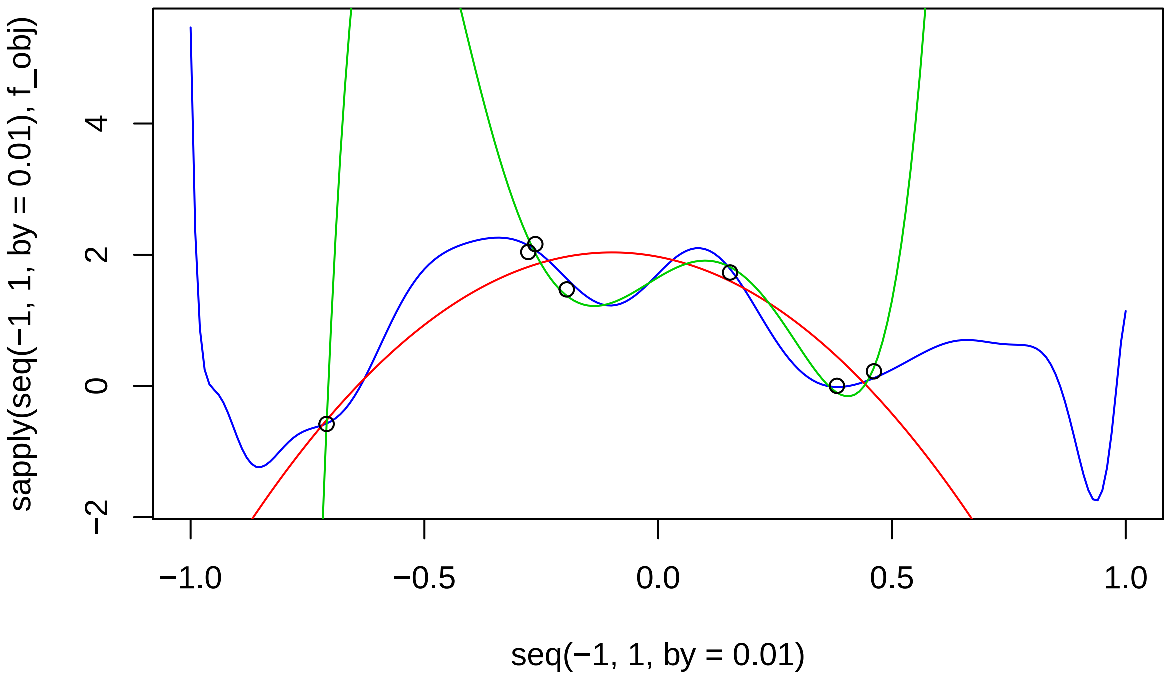
```r
Q_f <- 20
N <- 7
sigma <- 0.1
```

Generamos los coeficientes a_q:

```r
a_Q <- rnorm(Q_f+1)/sqrt(sum(1/(2*(0:Q_f)+1)))
f_obj <- function(x){  sapply(x, function(y) crossprod(a_Q,
                                 lagrange_poly(y,Q_f))) }
x <- runif(N, min=-1, max=1)
y <- f_obj(x) + sigma*rnorm(N)
```

Para calcular g_2 y g_{10}, hacemos regresión lineal sobre los datos.

```r
coefs_g2 <- regress_lin(t(sapply(x,function(i) i^(0:2))), y)
coefs_g10 <- regress_lin(t(sapply(x,function(i) i^(0:10))), y)

g2 <- function(x){
  sapply(x, function(y) crossprod(coefs_g2, y^(0:2)))
}

g10 <- function(x){
  sapply(x, function(y) crossprod(coefs_g10, y^(0:10)))
}

plot(seq(-1,1,by=0.01), sapply(seq(-1,1,by=0.01), f_obj), type ="l", col=4)
points(seq(-1,1,by=0.01), sapply(seq(-1,1,by=0.01), g2), type ="l", col=2)
points(seq(-1,1,by=0.01), sapply(seq(-1,1,by=0.01), g10), type ="l", col=3)
points(x,y)
```



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

σ representa el peso del error introducido en los datos. La normalización de los datos de f se realiza para reducir la variabilidad de los datos (especialmente en los extremos, donde se ve más la influencia de las mayores potencias o los polinomios de legendre de mayor grado).

c. ¿Cómo podemos obtener E_out analíticamente para una g_{10} dada?

La forma de obtener E_out será, dada g_{10}, calcular $\mathbb{E}_x[(g_{10}(x) - f(x))^2]$. Esto es, calcular el error fuera de la muestra, con varianza igual a 0, ya que g_{10} es fijo. Vemos el error para la g_{10} obtenida:

```
integrate(function(x) (g10(x)-f_obj(x))^2,-1,1,subdivisions = 10000L)
```

```
## 9880.207 with absolute error < 0.15
```

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

$$E_{out}(\mathcal{H}_{\in}) = promedio\ sobre\ experimentos(E_{out}(g_2))$$

$$E_{out}(\mathcal{H}_{\infty\prime}) = promedio\ sobre\ experimentos(E_{out}(g_{10}))$$

Definimos una medida de sobreajuste como $E_{out}(g_{10}) - E_{out}(g_2)$

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

Esta medida puede medir el sobreajuste debido a que se trata de la diferencia entre $E_{out}(\mathcal{H}_{10})$ y $E_{out}(\mathcal{H}_2)$, es decir, cuánto más hay de error al suponer la función objetivo está en $\mathcal{H}_{10}$ que suponer sólo que está en $\mathcal{H}_2$. Necesariamente el error dentro de la muestra será mejor o al menos igual de bueno para las funciones en $\mathcal{H}_{10}$ que en $\mathcal{H}_2$. Si hemos seleccionado el modelo $\mathcal{H}_{10}$ y la función pertenece a un espacio más pequeño, existirá ruido determinista.

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

```r
Q_f = 1:100
N = seq(20,100,by=5)
sigma = seq(0, 2, by=0.05)
```

```r
getEout <- function(f_obj, f, range = c(-1,1)){
  integ <- integrate(function(x) (f(x)-f_obj(x))^2, range[1], range[2],
                    subdivisions = subdivisions)
  return(integ)
}

getAproxPolynom <- function(x,y,grade){

  coefs <- regress_lin(t(sapply(x,function(i) i^(0:grade))), y)

  polynom <- function(x){
    sapply(x, function(y) crossprod(coefs, y^(0:grade)))
  }

  return(polynom)
}
```

```r
overfitting <- function(param, iterations){
  Q_f <- param[1]
  N <- param[2]
  sigma <- param[3]

  diff_E_outs = vector(mode = "numeric", length = iterations)

  for( i in 1:iterations){
    a_Q <- rnorm(Q_f+1)/sqrt(sum(1/(2*(0:Q_f)+1)))
    f_obj <- function(x){ sapply(x, function(y) crossprod(a_Q,
                                    lagrange_poly(y,Q_f))) }
    x <- runif(N, min=-1, max=1)
    y <- f_obj(x) + sigma*rnorm(N)

    g2 <- getAproxPolynom(x,y,2)
    g10 <- getAproxPolynom(x,y,10)

    E_out_g10 <- getEout(f_obj, g10)$value
    E_out_g2 <- getEout(f_obj, g2)$value

    diff_E_outs[i] <- E_out_g10 - E_out_g2
  }
```

```
  return(list('mean'=mean(diff_E_outs), 'aQ'=a_Q))
}
```

```
d <- expand.grid(x = Q_f, y = N, z = sigma)
sobreajuste_2b <- apply(d, 1, function(x) overfitting(x,1000) )
```

3.- Repetir el experimento descrito en los puntos anteriores pero para el caso de clasificación donde la función objetivo es un perceptron ruidoso $f(x) = sign(\sum_{q=1}^{Q_f} a_q L_1(x) + \varepsilon)$. Notemos que $a_0 = 0$ y que las a_q deben ser normalizadas para que $\mathbb{E}_{a,x}\left[(\sum q = 1^{Q_f} a_q L_q(x))^2\right] = 1$. 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)

```
overfitting_3 <- function(param, iterations){
  Q_f <- param[1]
  N <- param[2]
  sigma <- param[3]

  diff_E_outs = vector(mode = "numeric", length = iterations)

  for( i in 1:iterations){
    a_Q <- rnorm(Q_f)/sqrt(sum(1/(2*(1:Q_f)+1)))
    f_obj <- function(x){ sapply(x, function(y) sign(crossprod(a_Q,
                                      lagrange_poly(y,Q_f)[2:Qf]) +
                                      rnorm(1,sd=sigma)))}
    x <- runif(N, min=-1, max=1)
    y <- f_obj(x) + sigma*rnorm(N)

    g2 <- getAproxPolynom(x,y,2)
    g10 <- getAproxPolynom(x,y,10)

    E_out_g10 <- getEout(f_obj, g10)$value
    E_out_g2 <- getEout(f_obj, g2)$value

    diff_E_outs[i] <- E_out_g10 - E_out_g2
  }

  return(list('mean'=mean(diff_E_outs), 'aQ'=a_Q))
}
```

```
sobreajuste_3 <- apply(d, 1, function(x) overfitting(x,1000) )
```

C - REGULARIZACIÓN Y SELECCIÓN DE MODELOS

1.- Para $d = 3$ (dimensión) generar un conjunto de N datos aleatorios $\{x_n, y_n\}$de la siguiente forma. Para cada punto x_n generamos sus coordenadas muestreando de forma independiente una $N(0, 1)$. De forma similar generamos un vector de pesos de $d + 1$ dimensiones $\mathbf{w}_f$, y el conjunto de valores $y_n = \mathbf{w}_f^T \mathbf{x}_n + \sigma\varepsilon_n$, donde ε_n es un ruido que sigue también una $N(0, 1)$ y σ^2 es la varianza del ruido; fijar $\sigma = 0.5$.

Usar regresión lineal con regularización $*weight decay* para estimar $\mathbf{w}_f$ con $\mathbf{w}_{reg}$. Fijar el parámetro de regularización a $0.05/N$

```
simula_gauss <- function(N, dim, sigma){
  #Tomamos N*dim elementos de una normal, tomando la desviación estándar del vector sigma
  lista <- matrix(rnorm(N*dim, sd = sigma), N, dim, byrow=TRUE)
  return(lista)
}

N <- 100
sigma <- 0.05
d <- 3
x <- simula_gauss(N, d, 1)
w <- simula_gauss(1, d+1, 1)
y <- apply(x, 1, function(x_i) crossprod(t(w),c(x_i,1))+rnorm(1,sd=sigma))
wd_lamda <- sigma/N
```

```
regress_lin <- function(datos, label, weigth_decay = 1){
  pseudo_inv <- pseudoinv(t(datos)%*%datos +
                          weigth_decay*diag(ncol(datos)))
  return(pseudo_inv%*%t(datos)%*%label)
}
```

```
coefs_rl_1 <- regress_lin(cbind(x,1), y, wd_lamda)
```

a. Para $N \in \{d + 15, d + 25, \ldots, d + 115\}$ calcular los errores $e_1, \ldots, e_N$ de validación cruzada y E_{cv}.

```
for( N in seq(d+15, d+115, by=10) ){
  x <- simula_gauss(N, d, 1)
  w <- simula_gauss(1, d+1, 1)
  y <- apply(x, 1, function(x_i)
    crossprod(t(w),c(x_i,1))+rnorm(1,sd=sigma))
  wd_lamda <- sigma/N
  e <- vector(mode = "numeric", length = N)
  for( n in 1:N ){
    Dn = list('data'=cbind(x[1:N != rep(n,N),],1),
              'value'=y[1:N != rep(n,N)])
    coefs <- regress_lin(Dn$data, Dn$value, wd_lamda)
    e[n]= (y[n]-crossprod(coefs,c(x[n,],1)))^2
  }
  cat("For N=",N,"\t Ecv=",mean(e),"\n")
}
```

```
## For N= 18      Ecv= 0.00241717
```

```
## For N= 28     Ecv= 0.003337834
## For N= 38     Ecv= 0.003155914
## For N= 48     Ecv= 0.002909705
## For N= 58     Ecv= 0.003787424
## For N= 68     Ecv= 0.002230054
## For N= 78     Ecv= 0.002664281
## For N= 88     Ecv= 0.002104468
## For N= 98     Ecv= 0.002739491
## For N= 108    Ecv= 0.002616665
## For N= 118    Ecv= 0.002461887
```

b. Repetir el experimento 10^5 veces, anotando el promedio y la varianza de e_1 , e_2 y E_{cv} en todos los experimentos.

```r
e_1 = vector(mode="numeric", length = 10^4)
e_2 = vector(mode="numeric", length = 10^4)
E_cv = vector(mode="numeric", length = 10^4)

for( iteration in 1:10^4){


  for( N in seq(d+15, d+115, by=10) ){
    x <- simula_gauss(N, d, 1)
    w <- simula_gauss(1, d+1, 1)
    y <- apply(x, 1, function(x_i)
      crossprod(t(w),c(x_i,1))+rnorm(1,sd=sigma))
    wd_lamda <- sigma/N
    e <- vector(mode = "numeric", length = N)
    for( n in 1:N ){
      Dn = list('data'=cbind(x[1:N != rep(n,N),],1),
                'value'=y[1:N != rep(n,N)])
      coefs <- regress_lin(Dn$data, Dn$value, wd_lamda)
      e[n]= (y[n]-crossprod(coefs,c(x[n,],1)))^2
    }
  }


}
```

c. ¿Cuál debería de ser la relación entre el promedio de los valores de e_1 y el de los valores de E_{cv}? ¿y el de los valores de e_2? Argumentar la respuesta en base a los resultados de los experimentos.

d. ¿Qué es lo que contribuye a la varianza de los valores de e_1?

e. Si los errores de validación-cruzada fueran verdaderamente independientes, ¿cual sería la relación entre la varianza de los valores de e_1 y la varianza de los de E_{cv} ?

f. Una medida del número efectivo de muestras nuevas usadas en el cálculo de E_{cv} es el cociente entre la varianza de e_1 y la varianza de E_{cv} . Explicar por qué, y dibujar, respecto de N, el número efectivo de nuevos ejemplos(N_{eff}) como un porcentaje de N. NOTA: Debería de encontrarse que N_{eff} está cercano a N.

g. Si se incrementa la cantidad de regularización, ¿debería N_{eff} subir o bajar?. Argumentar la respuesta. Ejecutar el mismo experimento con $\lambda = 2.5/N$ y comparar los resultados del punto anterior para verificar la conjetura.

2.- La técnica de validación cruzada da una estimación precisa de $\hat{E}_{out}(N-1)$, pero puede ser demasiado inestable en problemas de selección de modelos. Una heurística común para regularizar validación cruzada en selección de modelos es usar una medida de su error $\sigma_{cv}(\mathcal{H})$.

a. Una elección para $\sigma_{cv}(\mathcal{H})$ es el uso de la desviación estandar de los errores por LOO ("leave-one-out") dividida por $\sqrt{N}$, $\sigma_{cv}(\mathcal{H}) \approx \frac{1}{\sqrt{N}}\sqrt{var(e_1,\ldots,e_n)}$. ¿Por qué se divide por $\sqrt{N}$?

b. Analizamos dos aproximaciones para la estimación:

   i. Dado el mejor modelo $\mathcal{H}^\star$, la aproximación conservadora de $1-\sigma$ selecciona el modelo más simple a una distancia $\sigma_{cv}(\mathcal{H}^\star)$ del mejor.
   ii. La aproximación que minimiza una cota selecciona el modelo que minimiza $E_{out}(\mathcal{H}) + \sigma_{cv}(\mathcal{H})$.

Usar el diseño experimental del ejercicio sección 2.1 (sobreajuste) para comparar estas aproximaciones con la estimación "no regularizada" de validación cruzada. Para ello hacer lo siguiente:
I. Fijar $Q_f = 15,\ Q = 20,\ \sigma = 1$.
II. Aplicar cada una de las dos aproximaciones propuestas así como el estimador estándar de validación cruzada para seleccionar el valor óptimo del parámetro de regularización λ en el conjunto $\{0.05, 0.1, 0.15, \ldots, 5\}$ usando regularización por "weight decay", $\Sigma(w) = \frac{\lambda}{N} w^T w$.
III. Dibujar el error fuera de la muestra para cada una de las técnicas, usando cada una de ellas como una función de N, con $N \in \{2 \times Q, 3 \times Q, \ldots, 10 \times Q, \}$. ¿Cuáles son sus conclusiones?.