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)
}
hiperplane_to_function <- function( vec ){
  f <- function(x){
    return( crossprod(vec,x) )
  }
  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_iterati
```

```
## 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.

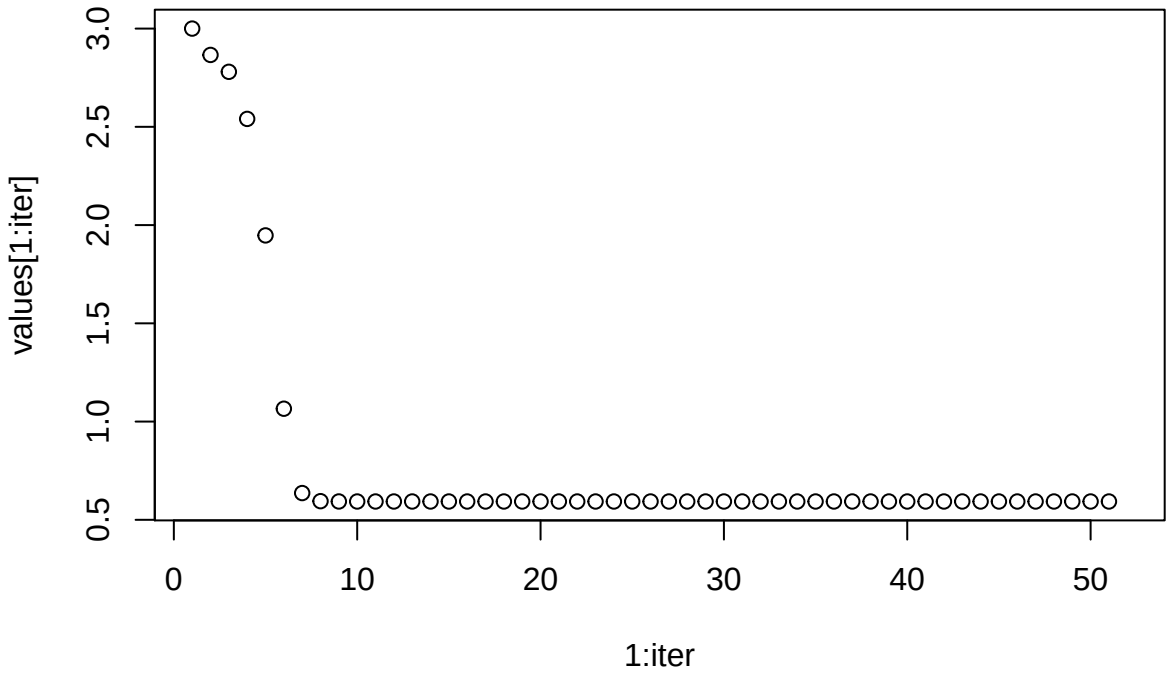
b) Considerar ahora la función $f(x, y) = x^2 + 2y^2 + 2\sin(2\pi x)\sin(2\pi y)$

Implementamos de nuevo la función y sus derivadas:

```
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.

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

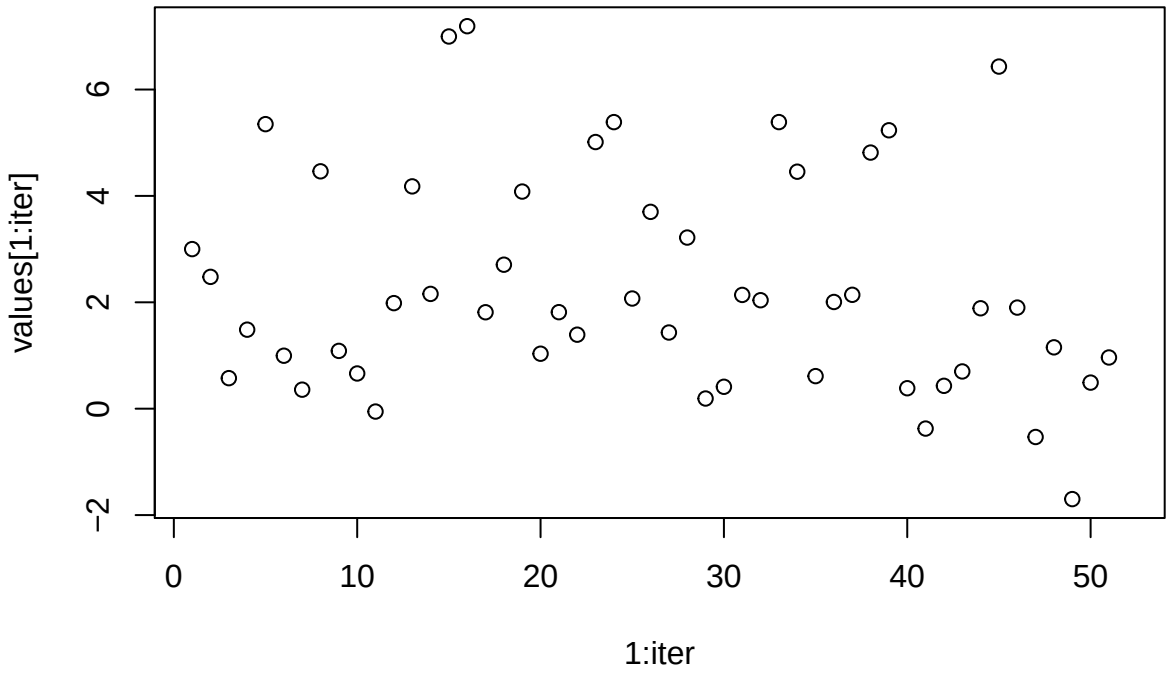



```
## $min
## [1] 1.218070 0.712812
```

```
##
## $iterations
## [1] 50
```

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

```
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?

```
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_itera
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
```

```
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 $\mu = 0.1$.

```
coordinateDescent <- 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$value
  grad_u <- aux$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) ?

```
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.

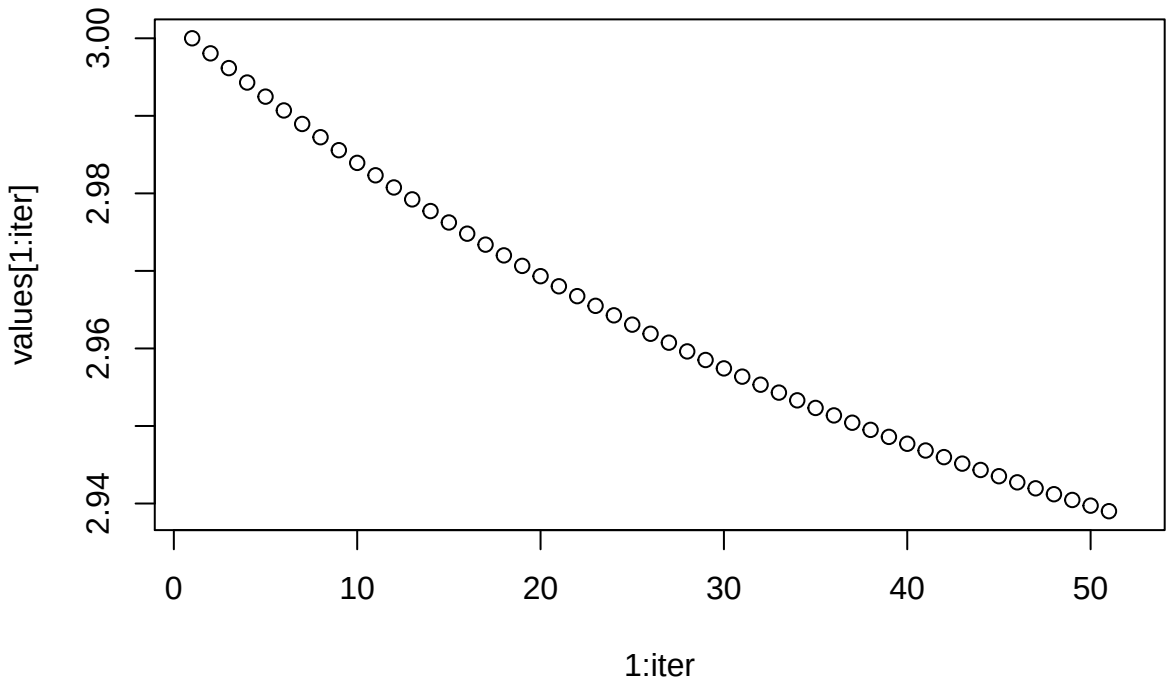
3.- Método de Newton: Implementar el algoritmo de minimización de Newton y aplicarlo a la función $f(x, y)$ dada en el ejercicio 1b. Desarrolle los mismos experimentos usando los mismos puntos de inicio.

```
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])
  }

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

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

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

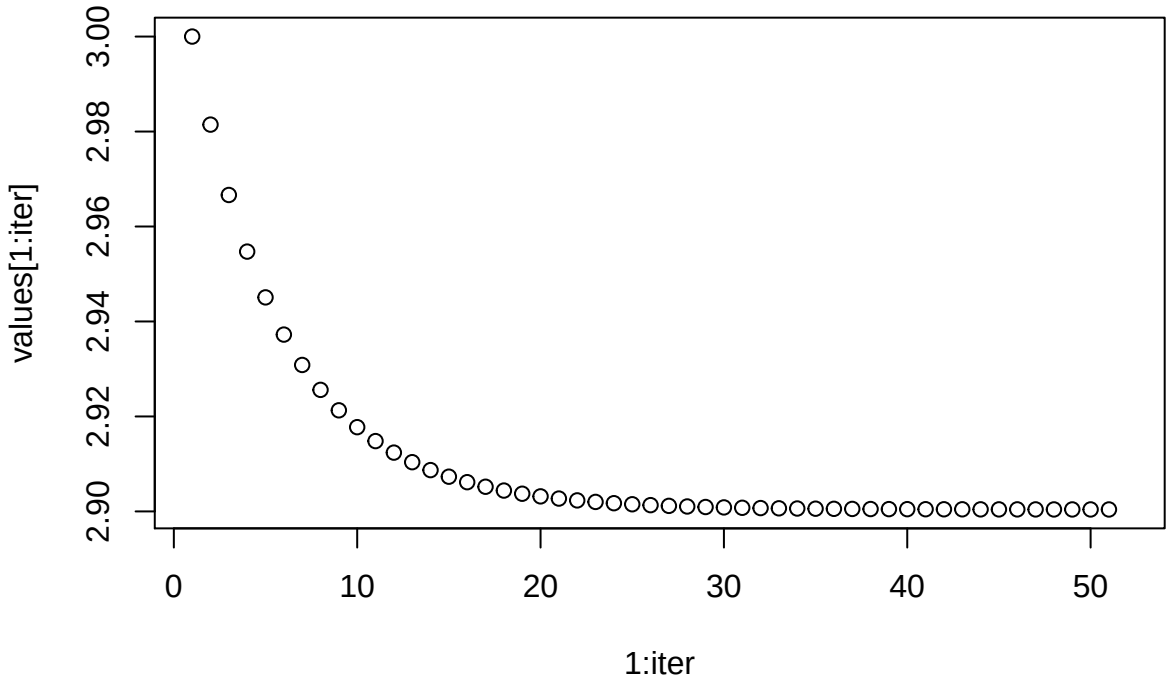



```
## $min
##           [,1]
```

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

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

```
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_iteratio
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`:

```
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 <- function(x,y){
    return(y - a*x -b)
  }
  return(f)
}
```

```
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])))
etiquetas_4a[etiquetas_4a<0] <- 0
```

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 $\nu = 0.01$

Definición de la función logística:

```
logistic <- function(x){ exp(x)/(1+exp(x)) }
```

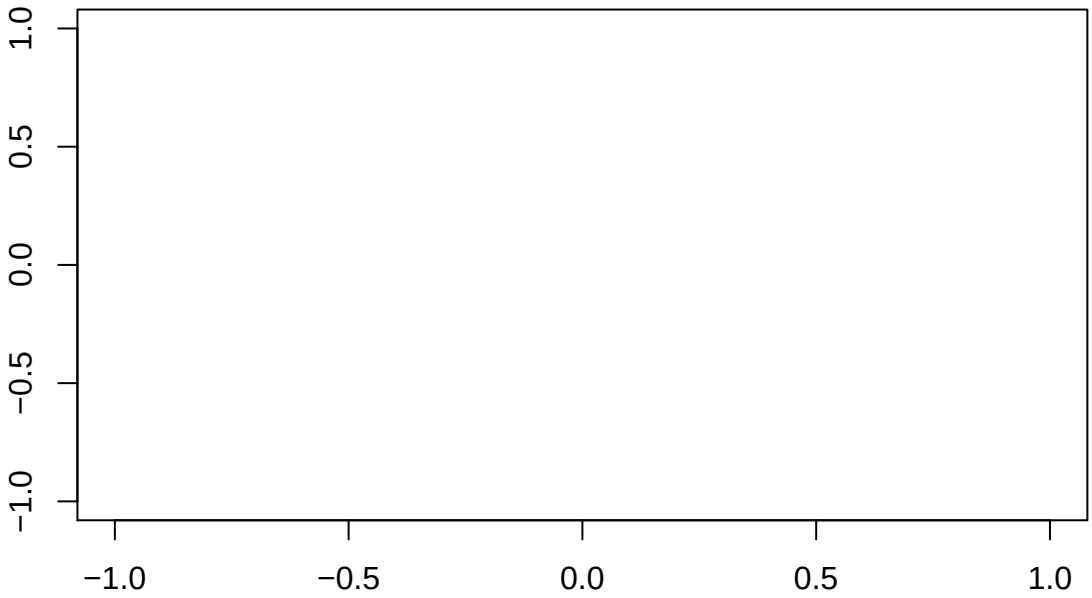
Implementación del descenso del gradiente estocástico:

```
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){
  prev_sol <- vini
  iter <- 0
  stopped <- FALSE

  while ( iter < max_iter && !stopped){
    permutation = sample(nrow(datos),nrow(datos))
    sol <- prev_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(-3,3), c(-3,3), function(x,y) y +sol[1]/sol[2]*x
                    +sol[3]/sol[2], 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( hiperplane = sol, iterations = iter))
}
```

Cálculo de la función g mediante la regresión logística:

```
sol_4a <- SGD(muestra_4a, etiquetas_4a, 1000)$hiperplane
g_4a <- function(x,y) sol_4a[1]*x + sol_4a[2]*y + sol_4a[3]
draw_function(c(-1,1),c(-1,1), g_4a)
```



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:

```
  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])))
  etiquetas_4a_out[etiquetas_4a_out<0] <- 0

  # 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.

```
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])))
  etiquetas_4c[etiquetas_4c<0] <- 0

  sol_4c <- SGD(muestra_4c, etiquetas_4c, 1000)$hiperplane
  g_4c <- function(x,y) sol_4c[1]*x + sol_4c[2]*y + sol_4c[3]

  muestra_4c_out <- simula_unif(N, 2, c(-1,1))
  etiquetas_4c_out <- apply(muestra_4c_out, 1, function(X) sign(recta_4c(X[1],X[2])))
  etiquetas_4c_out[etiquetas_4c_out<0] <- 0

  # Etiquetado según la regresión
  rl_etiquetas_4c_out <- apply(muestra_4c_out, 1, function(X) sign(g_4c(X[1],X[2])))
  rl_etiquetas_4c_out[rl_etiquetas_4c_out<0] <- 0

  # 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:  47.14
```

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

```
datos_train <- scan("datos/zip.train",sep=" ")
datos_train <- matrix(datos_train,ncol=257,byrow=T)
datos_train <- datos_train[-(which(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

```
datos_test <- read.table("datos/zip.test",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
```

```
number_test <- datos_train[,1]
datos_test <- datos_test[datos_test$V1 == 1 || datos_test$V1==5, -c(1,258)]
pixels_test <- array(t(datos_train[,2:257]), dim=c(16,16,nrow(datos_train)))
```

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

```
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:

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

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)