Trabajo 2: Programación

Gema Correa Fernández

27 de Abril de 2017

Nota: *Antes de comenzar a ejecutar todo el código, inicializamos una semilla a un valor por defecto, convenientemente a un número primo.*

```
set.seed(3) # Inicializamos la semilla a un número por defecto
```

Modelos Lineales

Ejercicio 1

Gradiente Descendente. Implementar el algoritmo de gradiente descendente.

a) Considerar la función no lineal $E(u,v) = (u^2e^v - 2v^2e^{-u})^2$. Usar gradiente descendente para encontrar un mínimo de esta función, 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 $E(u,v)$

El gradiente de $E(u,v)$ es la derivada con respecto cada una de sus variables. En este caso, calcularemos la derivada con respecto u y v. Por tanto, lo primero que haremos será calcular la derivada de E respecto de u y luego respecto de v.

Derivada de E respecto u: Tratamos v como constante y partimos de:

$$\frac{\partial((u^2e^v - 2v^2e^{-u})^2)}{\partial u}$$

Aplicamos la regla de la cadena $\frac{df(u)}{dx} = \frac{df}{du}\frac{du}{dx}$ y asignamos a $w = (u^2e^v - 2v^2e^{-u})^2$. Por tanto, nos quedará algo así:

$$\frac{\partial}{\partial w}(w^2)\frac{\partial}{\partial u}(u^2e^v - 2v^2e^{-u})$$

Para que resulte más fácil resolverla, vamos a derivar cada multiplicando por separado. Luego más tarde, los juntaremos.

1. Partimos de $\frac{\partial}{\partial w}(w^2)$ y aplicamos la regla de la potencia:

$$\frac{\partial}{\partial w}(w^2) = 2w^{(2-1)} = 2w^1 = 2w$$

2. Partimos de $\frac{\partial}{\partial u}(u^2e^v - 2v^2e^{-u})$ y aplicamos la regla de la suma o diferencia, que consiste en derivar cada sumando por separado:

$$\frac{\partial}{\partial u}(u^2e^v - 2v^2e^{-u}) = \frac{\partial}{\partial u}(u^2e^v) - \frac{\partial}{\partial u}(2v^2e^{-u})$$

Para que nos resulte más sencillo resolverla, volvemos a resolverla en dos partes.

2.1. Partimos de $\frac{\partial}{\partial u}(u^2e^v)$, donde sacamos la constante fuera $e^v\frac{\partial}{\partial u}(u^2)$ y aplicamos la regla de la potencia $e^v 2u^{(2-1)} = e^v 2u = 2e^v u$

2.2. Partimos de $\frac{\partial}{\partial u}(2v^2e^{-u})$, donde sacamos la constante fuera $2v^2\frac{\partial}{\partial u}(e^{-u})$ y volvemos a aplicar la regla de la cadena donde $w = -u$, por lo que nos quedaría $2v^2\frac{\partial}{\partial w}(e^w)\frac{\partial}{\partial u}(-u)$ que es lo mismo que $2v^2e^w(-1)$. Sustituimos en la ecuación $w = -u \longrightarrow 2v^2e^{-u}(-1)$ y la simplificamos $-2v^2e^{-u}$

Ahora juntamos ambas partes (puntos 2.1 y 2.2):

$$\frac{\partial}{\partial u}(u^2e^v) - \frac{\partial}{\partial u}(2v^2e^{-u}) = 2e^v u - (-2v^2e^{-u}) = 2e^v u + 2v^2e^{-u}$$

Luego juntamos el punto 1 y el punto 2 (***Nota****: Recordad que hemos sustituido w por $u^2e^v - 2v^2e^{-u})^2$)*

$$\frac{\partial}{\partial w}(w^2)\frac{\partial}{\partial u}(u^2e^v - 2v^2e^{-u}) = 2w(2e^v u + 2v^2e^{-u})$$

Finalmente, la derivada de E respecto de u es:

$$\frac{\partial((u^2e^v - 2v^2e^{-u})^2)}{\partial u} = 2\left(u^2e^v - 2v^2e^{-u}\right)\left(2e^v u + 2e^{-u}v^2\right)$$

Derivada de E respecto v: Tratamos u como constante y partimos de:

$$\frac{\partial((u^2e^v - 2v^2e^{-u})^2)}{\partial v}$$

Aplicamos la regla de la cadena $\frac{df(u)}{dx} = \frac{df}{du}\frac{du}{dx}$ y asignamos $w = (u^2e^v - 2v^2e^{-u})^2$. Por tanto, nos quedará algo así:

$$\frac{\partial}{\partial w}(w^2)\frac{\partial}{\partial v}(u^2e^v - 2v^2e^{-u})$$

Para que resulte más fácil resolverla, vamos a derivar cada multiplicando por separado. Luego más tarde, los juntaremos.

1. Partimos de $\frac{\partial}{\partial w}(w^2)$ y aplicando la regla de la potencia, obtenemos:

$$\frac{\partial}{\partial w}(w^2) = 2w^{(2-1)} = 2w^1 = 2w$$

2. Partimos de $\frac{\partial}{\partial v}(u^2e^v - 2v^2e^{-u})$ y aplicamos la regla de la suma o diferencia, que consiste en derivar cada sumando por separado:

$$\frac{\partial}{\partial v}(u^2e^v - 2v^2e^{-u}) = \frac{\partial}{\partial v}(u^2e^v) - \frac{\partial}{\partial u}(2v^2e^{-u})$$

Para que nos resulte más sencillo resolverla, volvemos a resolverla en dos partes.

2.1. Partimos de $\frac{\partial}{\partial u}(u^2e^v)$, donde sacamos la constante fuera $u^2\frac{\partial}{\partial v}(e^v)$ y aplicamos la regla de derivación, entonces $\frac{\partial}{\partial u}(u^2e^v) = u^2e^v$

2.2. Partimos de $\frac{\partial}{\partial u}(2v^2e^{-u})$, donde sacamos la constante fuera $2e^{-u}\frac{\partial}{\partial v}(v^2)$ y aplicamos la regla de la potencia, entonces $\frac{\partial}{\partial u}(2v^2e^{-u}) = 2e^{-u}2v^{(2-1)} = 2e^{-u}2v^1 = 4e^{-u}v$.

Ahora juntamos ambas partes (puntos 2.1 y 2.2):

$$\frac{\partial}{\partial v}(u^2e^v) - \frac{\partial}{\partial u}(2v^2e^{-u}) = u^2e^v - 4e^{-u}\mathring{u}v$$

Luego juntamos el punto 1 y el punto 2 (***Nota:*** *Recordad que hemos sustituido* w *por* $u^2e^v - 2v^2e^{-u})^2$)

$$\frac{\partial}{\partial w}(w^2)\frac{\partial}{\partial v}(u^2e^v - 2v^2e^{-u}) = 2w(u^2e^v - 4e^{-u}v)$$

Finalmente, la derivada de E respecto de v:

$$\frac{\partial((u^2e^v - 2v^2e^{-u})^2)}{\partial v} = 2\left(u^2e^v - 2v^2e^{-u}\right)\left(u^2e^v - 4e^{-u}v\right)$$

Así el gradiente de la función $E(u,v) = (u^2e^v - 2v^2e^{-u})^2$ es:

$$\nabla E(u,v) = 2\left(u^2e^v - 2v^2e^{-u}\right)\left(2e^vu + 2e^{-u}v^2\right), 2\left(u^2e^v - 2v^2e^{-u}\right)\left(u^2e^v - 4e^{-u}v\right)$$

Para comprobar que ambas derivadas son correctas, lo que he hecho ha sido comprobarlo con un paquete de cálculo matemático denominado wxMaxima y también de manera online en la página Symbolab.

Teniendo en cuenta esto, nos creamos una función para su gradiente y para la función no lineal proporcionada en el apartado a).

```
# Creamos una función para la funcion no lineal E(u,v)
E <- function(u, v){
  ((u^2)*(exp(v)) - 2*(v^2)*exp(-u))^2
}

# Creamos una función para su gradiente (derivadas parciales)
E_dev <- function(u, v){
    Eu <- 2 * ( (u^2)*exp(v) - 2*(v^2)*exp(-u) ) * ( 2*exp(v)*u + 2*exp(-u)*(v^2) )
    Ev <- 2 * ( (u^2)*exp(v) - 2*(v^2)*exp(-u) ) * ( (u^2)*exp(v) - 4*exp(-u)*v )
    c(Eu, Ev)
}
```

2) ¿Cuántas iteraciones tarda el algoritmo en obtener por primera vez un valor de $E(u,v)$ inferior a 10^{-4}. (Usar flotantes de 64 bits)

Para ello, procedemos a implementar el **Algoritmo de Gradiente Descendente**. Haciendo uso del pseudocódigo proporcionado por el libro *Learning from data: A short course* o por la *Sesión 5 de Teoría*, página 28.

Pseudocódigo Gradiente Descendente

1. Inicializar los pesos $t = 0$ a $w(0)$
2. **Para** $t = 0, 1, 2, ...$ **hacer**
   2.1. Calcular el gradiente: $g_t = \nabla E_{in}(w(t))$
   2.2. Establecer la dirección a moverse: $v_t = -g_t$
   2.3. Actualizar los pesos: $w(t+1) = w(t) + \eta v_t$
   2.4. Iterar al paso siguiente hasta que se rompa la condición de parada
3. **Devolver** los pesos finales

Por tanto debemos pasar como entrada al **Algoritmo de Gradiente Descendiente**:

- La función que le pasamos ($funcion$)
- Las dos derivadas parciales de la función ($funcion\_gradiente$)
- Un tasa de aprendizaje (η)
- El punto de inicio ($vini$)
- Un umbral de diferencia entre dos puntos ($umbral$)
- El número de iteraciones máximas, el cual será una de nuestras condiciones de parada ($iteraciones\_max$)

Este algoritmo hace uso del gradiente de una función y cuando busca el mínimo hace uso del negativo del gradiente para así aproximarse al mínimo (óptimo). Se utiliza para minimizar una función de coste y encontrar el mejor ajuste para esos datos. Además, puede que el método dé una mala aproximación, ya que depende de un parámetro de entrada para avanzar.

Para detener el algoritmo, usaremos tres condiciones de parada:

- Un número máximo de iteraciones.
- El valor de la funcion sea menor o igual a un umbral.
- La diferencia entre dos puntos en valor absoluto sea muy cercana.

***Nota**: No he usado la norma, por lo que al principio irá más rapido y luego más lento.*

```
# Algoritmo Gradiente Descendente
GD <- function(funcion, funcion_gradiente, eta=0.1, vini=c(0,0), umbral=10^(-4),
               iteraciones_max=500){

  # Inicializar los pesos
  w_old <- c(0,0)
  w_new <- vini

  # Variable que lleva el número de iteraciones realizadas
  iter <- 0

  # Array creado donde guardaremos el valor de la funcion, para luego crear la gráfica
  f <- rep(iteraciones_max)
  f[iter] <- funcion(w_new[1],w_new[2])

  # Mostramos el punto de partida
  cat("Punto de partida: ", w_new[1], w_new[2])

  # Mostramos los valores del gradiente al principio
  cat("\nGradiente (inicial): ", funcion_gradiente(w_new[1],w_new[2]))

  # Mostramos el valor de la función al principio
  cat("\nValor función (inicial): ", funcion(w_new[1], w_new[2]))

  # La condición de parada será cuando el valor de la funcion sea menor o igual a
  # un umbral, cuando se hayan completado un número máximo de iteraciones o cuando
  # la diferencia entre dos puntos en valor absoluto sea muy cercana.
  # La condición de la diferencia entre dos puntos en valor absoluto sea muy cercana,
  # descarta muchos valores, ya que avanza muy lentamente
  while ( (iter <= iteraciones_max) & (funcion(w_new[1], w_new[2]) > umbral)
         & (abs(funcion(w_new[1], w_new[2]) - funcion(w_old[1], w_old[2])) > umbral) ) {

    # Asignamos el valor nuevo, al antiguo
    w_old <- w_new
```

```
    # Calculamos el gradiente
    gradiente <- funcion_gradiente(w_new[1], w_new[2])

    # Establecemos la dirección a moverse
    direccion <- -gradiente

    # Actualizamos los pesos
    w_new <- w_old + eta * direccion

    # Incrementamos el valor de la iteración (época)
    iter <- iter + 1

    # Guardamos el nuevo valor de f
    f[iter] = funcion(w_new[1],w_new[2])
  }

  # Mostramos los valores del gradiente al terminar
  cat("\nGradiente (final): ", funcion_gradiente(w_new[1],w_new[2]))

  # Mostramos la solución final
  cat("\nSolución: ", w_new[1], w_new[2])

  # Mostramos el valor de la función al final
  cat("\nValor función (final): ", funcion(w_new[1], w_new[2]))

  # Devolvemos el número de iteraciones necesarias para encontrar la solución
  cat("\nIteraciones: ",iter-1)
  cat("\n")

  # Mostramos el valor de la funcion
  list(valor_funcion = f[1:iter])
}
```

Vamos a ejecutar el algoritmo, con las condiciones especificadas en el enunciado. Pararemos cuando el valor de la funcion sea menor o igual a 10^{-4}, o cuando la diferencia en valor absoluto de dos puntos seguidos sea menor o igual a 10^{-4} o bien, cuando se hayan completado 500 iteraciones.

```
GD (E, E_dev, eta=0.1, vini=c(1,1), umbral=10^(-4), iteraciones_ma=500)
```

```
## Punto de partida:  1 1
## Gradiente (inicial):  24.47354 4.943477
## Valor función (inicial):  3.930397
## Gradiente (final):  -0.007711036 -0.0006310625
## Solución:  9.864573 -24.43828
## Valor función (final):  0.003855518
## Iteraciones:  3
```

```
## $valor_funcion
## [1] 1.687693e+00 1.174763e+02 3.861518e-03 3.855518e-03
```

Para comprobar que el gradiente comienza en esa solución, vamos a comprobarlo con el código proporcionado por la profesora:

```r
# Guardamos la expresion
expresion = expression((u^2*exp(v) - 2*v^2*exp(-u))^2)

# Vamos hacer la derivada a esa función, dependiendo de los valores 'u' y 'v'
duv = deriv(expresion,c("u","v"))

# Establecemos el punto donde realizar la derivada
u = 1
v = 1

# Evaluamos la derivada
eval(duv)
```

```
## [1] 3.930397
## attr(,"gradient")
##             u        v
## [1,] 24.47354 4.943477
```

```r
# Evaluamos la expresion
eval(expresion)
```

```
## [1] 3.930397
```

Como acabamos de comprobar, el valor de la derivada en el punto de inicio coincide $(24.47354, 4.943477)$

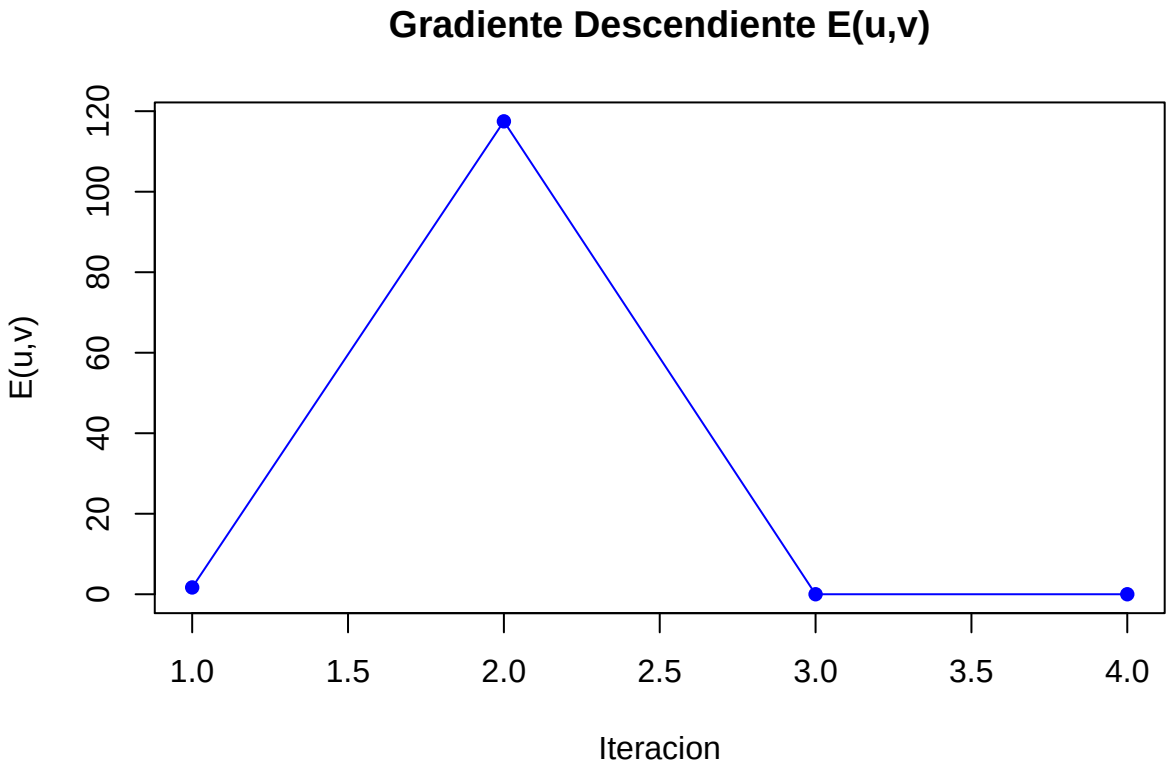
Volviendo a nuestra ejecución del *Gradiente Descendete* de antes, podemos ver que el algoritmo tarda 3 iteraciones en converger, ya que la última condición nos descarta valores que tienen una diferencia despreciable con el valor anterior.

Para entender mejor el resultado, vamos a mostrarlo gráficamente.

```r
# Guardamos el valor de la función
resultado01 = GD(E, E_dev, eta=0.1, vini=c(1,1), umbral=10^(-4), iteraciones_ma=500)
```

```
## Punto de partida:  1 1
## Gradiente (inicial):  24.47354 4.943477
## Valor función (inicial):  3.930397
## Gradiente (final):  -0.007711036 -0.0006310625
## Solución:  9.864573 -24.43828
## Valor función (final):  0.003855518
## Iteraciones:  3
```

```r
# La mostramos gráficamente
plot(resultado01$valor_funcion, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "E(u,v)", main = "Gradiente Descendiente E(u,v)")
```



Vemos que en tan solo 3 iteraciones (quitando el punto de inicio), el algoritmo ha alcanzado el mínimo local más cercano.

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

3) ¿Qué valores de (u, v) obtuvo en el apartado anterior cuando alcanzó el error de 10^{-4}?.

Para contestar a esta preguna sólo tenemos que ver el último valor de $w\_new[1]$ y de $w\_new[2]$.

```
GD(E, E_dev, eta=0.1, vini=c(1,1), umbral=10^(-4), iteraciones_ma=500)
```

```
## Punto de partida:  1 1
## Gradiente (inicial):  24.47354 4.943477
## Valor función (inicial):  3.930397
## Gradiente (final):  -0.007711036 -0.0006310625
## Solución:  9.864573 -24.43828
## Valor función (final):  0.003855518
## Iteraciones:  3
```

```
## $valor_funcion
## [1] 1.687693e+00 1.174763e+02 3.861518e-03 3.855518e-03
```

Por tanto, el último punto del gradiente descenciente es $(w\_new[1], w\_new[2]) = (9.864573, -24.43828) \longrightarrow 0.003855518$

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

Calculamos el gradiente de la misma manera que antes. En este caso, calcularemos la derivada de f con respecto x e y.

***Nota**: Omitiré algunos pasos de la resolución de la derivada, puesto que pueden hacer muy pesada la lectura.*

Derivada de f respecto x: Tratamos y como constante y partimos de:

$$\frac{\partial}{\partial\, x}\left((x-2)^2+2\,(y-2)^2+2\sin(2\pi x)\sin(2\pi y)\right)$$

Aplicamos la regla de la suma/diferencia. Por tanto, nos quedará algo así:

$$\frac{\partial}{\partial\, x}\left((x-2)^2\right)+\frac{\partial}{\partial\, x}\left(2\,(y-2)^2\right)+\frac{\partial}{\partial\, x}\left(2\sin(2\pi x)\sin(2\pi y)\right)$$

Para que resulte más fácil resolverla, vamos a derivar cada sumando por separado y una vez hecho volveremos a juntar las derivadas.

1. Partimos de $\frac{\partial}{\partial\, x}\left((x-2)^2\right)$ y aplicamos la regla de la cadena donde $u=(x-2)$:

$$\frac{\partial}{\partial\, x}\left((x-2)^2\right)=\frac{\partial}{\partial\, u}\left(u^2\right)\frac{\partial}{\partial\, x}(x-2)$$

Vamos a resolver la derivada de cada multiplicando por separado.

1.1. Partimos de $\frac{\partial}{\partial\, u}\left(u^2\right)$ y aplicamos la regla de la potencia $2u^{(2-1)}=2u$.

1.2. Partimos de $\frac{\partial}{\partial\, x}(x-2)$ y aplicamos la regla de la suma/diferencia $\frac{\partial}{\partial\, x}(x)-\frac{\partial}{\partial\, x}(2)$, resolvemos $\frac{\partial}{\partial\, x}(x)=1$ y $\frac{\partial}{\partial\, x}(2)=0$. Por tanto, nos quedaría $\frac{\partial}{\partial\, x}(x-2)=1-0=1$.

Ahora juntamos ambas partes (puntos 1.1 y 1.2): (***Nota**: Sustituir en la ecuación $u=(x-2)$*)

$$\frac{\partial}{\partial\, u}\left(u^2\right)\frac{\partial}{\partial\, x}(x-2)=2u\cdot\ 1=2\,(x-2)\cdot\ 1=2\,(x-2)$$

2. Partimos de $\frac{\partial}{\partial\, x}\left(2\,(y-2)^2\right)$ y como sabemos que la derivada de una constante es 0. Por tanto, $\frac{\partial}{\partial\, x}\left(2\,(y-2)^2\right)=0$

3. Partimos de $\frac{\partial}{\partial\, x}\left(2\sin(2\pi x)\sin(2\pi y)\right)$ cuya derivada es $4\pi\sin(2y\pi)\cos(2\pi x)$.

Luego juntamos el punto 1, el punto 2 y el punto 3:

$$\frac{\partial}{\partial\, x}\left((x-2)^2\right)+\frac{\partial}{\partial\, x}\left(2\,(y-2)^2\right)+\frac{\partial}{\partial\, x}\left(2\sin(2\pi x)\sin(2\pi y)\right)=2\,(x-2)+0+4\pi\sin(2y\pi)\cos(2\pi x)$$

Finalmente, la derivada de f respecto de x es:

$$\frac{\partial}{\partial\, x}\left((x-2)^2+2\,(y-2)^2+2\sin(2\pi x)\sin(2\pi y)\right)=4\pi\cos(2\pi x)\sin(2\pi y)+2\,(x-2)$$

Derivada de f respecto y: Tratamos x como constante y partimos de:

$$\frac{\partial}{\partial\, x}\left((x-2)^2+2\,(y-2)^2+2\sin(2\pi x)\sin(2\pi y)\right)$$

Aplicamos la regla de la suma/diferencia. Por tanto, nos quedará algo así:

$$\frac{\partial}{\partial y}\left((x-2)^2\right)+\frac{\partial}{\partial y}\left(2(y-2)^2\right)+\frac{\partial}{\partial y}(2\sin(2\pi x)\sin(2\pi y))$$

Para que resulte más fácil resolverla, vamos a derivar cada sumando por separado. Luego más tarde, se juntarán.

1. Partimos de $\frac{\partial}{\partial x}\left((x-2)^2\right)$ y derivamos una constante $\frac{\partial}{\partial y}\left((x-2)^2\right)=0$

2. Partimos de $\frac{\partial}{\partial y}\left(2(y-2)^2\right)$ cuya derivada es $4(y-2)$.

3. Partimos de $\frac{\partial}{\partial y}(2\sin(2\pi x)\sin(2\pi y))$ cuya derivada es $4\pi\sin(2x\pi)\cos(2\pi y)$ que simplificada es $4\pi\sin(2y\pi)\cos(2\pi x)$.

Luego juntamos el punto 1, el punto 2 y el punto 3:

$$\frac{\partial}{\partial y}\left((x-2)^2\right)+\frac{\partial}{\partial y}\left(2(y-2)^2\right)+\frac{\partial}{\partial y}(2\sin(2\pi x)\sin(2\pi y))=0+4(y-2)+4\pi\sin(2x\pi)\cos(2\pi y)$$

Finalmente, la derivada de f respecto de y es:

$$\frac{\partial}{\partial y}\left((x-2)^2+2(y-2)^2+2\sin(2\pi x)\sin(2\pi y)\right)=4\pi\sin(2\pi x)\cos(2\pi y)+4(y-2)$$

Así el gradiente de la función $f(x,y)=(x-2)^2+2(y-2)^2+2sin(2\pi x)sin(2\pi y)$ es:

$$\nabla f(x,y)=4\pi\cos(2\pi x)\sin(2\pi y)+2(x-2),4\pi\sin(2\pi x)\cos(2\pi y)+4(y-2)$$

Para comprobar que ambas derivadas son correctas, lo que he hecho ha sido comprobarlo con un paquete de cálculo matemático como wxMaxima y de manera online en la página Symbolab.

Teniendo en cuenta esto, nos creamos una función para su gradiente y para la función proporcionada en el apartado.

```
# Creamos una función para la funcion f(x,y)
f <- function(x, y){
  (x-2)^2 + 2*(y-2)^2 + 2*sin(2*pi*x)*sin(2*pi*y)
}

# Creamos una función para su gradiente (derivadas parciales)
f_dev <- function(x, y){
    fx <- 4*pi*cos(2*pi*x)*sin(2*pi*y) + 2*(x-2)
    fy <- 4*pi*sin(2*pi*x)*cos(2*pi*y) + 4*(y-2)
    c(fx, fy)
}
```

1) Usar gradiente descendente para minimizar esta función. Usar como punto inicial $(x_0=1, y_0=1)$**, 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.**

Realizamos el mismo procedimiento que para el apartado anterior:

```r
# Guardamos el valor de la función con la tasa de aprendizaje de 0.1
resultado02 = GD(f, f_dev, vini = c(1,1), eta = 0.1, umbral = 10^(-4),
                 iteraciones_max = 50)
```

```
## Punto de partida:  1 1
## Gradiente (inicial):  -2 -4
## Valor función (inicial):  3
## Gradiente (final):  -8.590336 2.1004
## Solución:  1.611878 1.348358
## Valor función (final):  -0.05388005
## Iteraciones:  9
```

```r
resultado02
```

```
## $valor_funcion
##  [1]  2.47803399  0.57401499  1.48564469  5.34911006  0.99575320
##  [6]  0.35731952  4.46231280  1.08628879  0.66165817 -0.05388005
```

```r
# Guardamos el valor de la función con la tasa de aprendizaje de 0.01
resultado03 = GD(f, f_dev, vini = c(1,1), eta = 0.01, umbral = 10^(-4),
                 iteraciones_max = 50)
```

```
## Punto de partida:  1 1
## Gradiente (inicial):  -2 -4
## Valor función (inicial):  3
## Gradiente (final):  0.01574209 -0.007690211
## Solución:  0.7821293 1.2871
## Valor función (final):  0.5932713
## Iteraciones:  8
```

```r
resultado03
```

```
## $valor_funcion
## [1] 2.8659382 2.7799956 2.5403700 1.9474671 1.0654046 0.6360725 0.5948140
## [8] 0.5933222 0.5932713
```

Como se aprecia, con una tasa de aprendizaje $\eta = 0.01$ vemos que alcanza un mínimo local más cercano e incluso tarda menos iteraciones que para $\eta = 0.1$. Sin embargo, para una tasa de aprendizaje de $\eta = 0.1$, el algoritmo no converge, es decir, al multiplicar por 0.1 tiene que dar una gran avance y esto puede que en algún momento nos saque del entorno del mínimo. Pero en este caso, alcanza un mínimo en 10 iteraciones.

Para comprobar que el gradiente comienza en esa solución, vamos a comprobarlo con el código proporcionado por la profesora:

```r
# Guardamos la expresion
expresion = expression((x-2)^2 + 2*(y-2)^2 + 2*sin(2*pi*x)*sin(2*pi*y))

# Vamos hacer la derivada a esa función, dependiendo de los valores 'u' y 'v'
dxy = deriv(expresion,c("x","y"))

# Establecemos el punto donde realizar la derivada
x = 1
y = 1

# Evaluamos la derivada
eval(dxy)
```

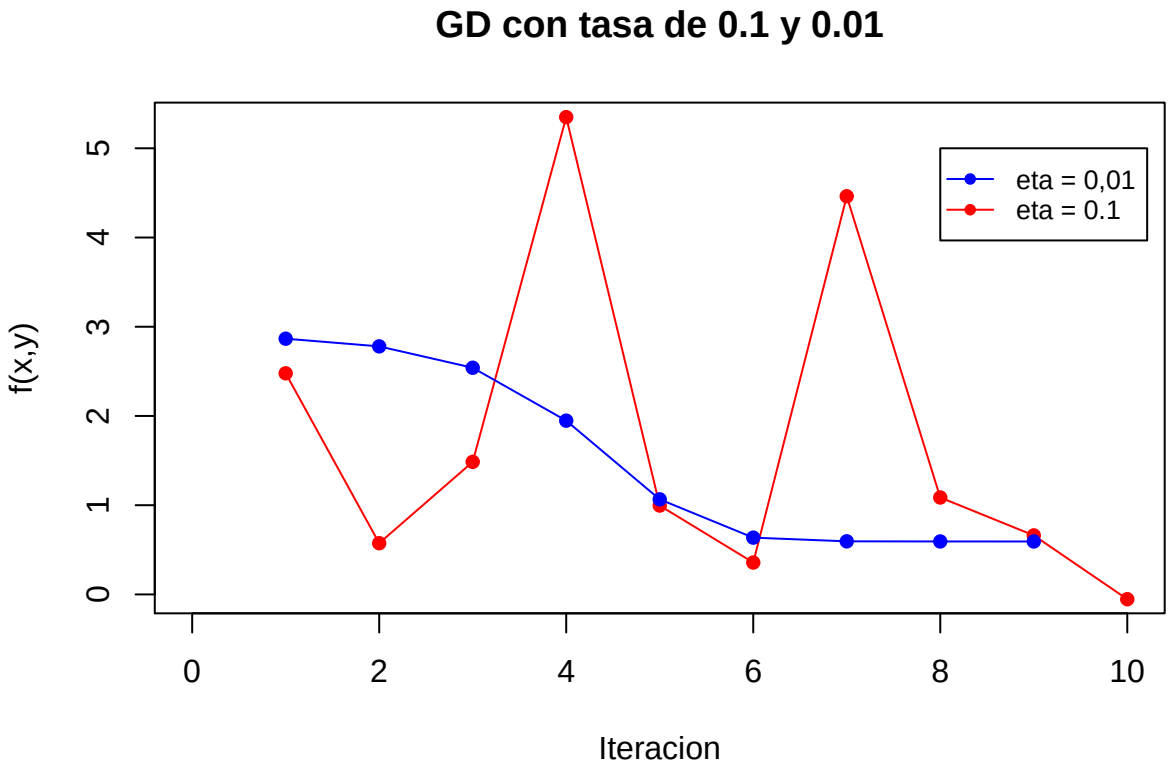
```
## [1] 3
## attr(,"gradient")
##       x  y
## [1,] -2 -4
```

```r
# Evaluamos la expresion
eval(expresion)
```

```
## [1] 3
```

Para entender el resultado de antes, vamos a mostrar gráficamente ambos resultados:

```r
# Abrimos plot
plot(c(0,10), c(0,5.3), type = "n", xlab = "Iteracion", ylab = "f(x,y)",
     main = "GD con tasa de 0.1 y 0.01")

# Mostramos gráficamente la recta para la tasa  0.1
lines(resultado02$valor_funcion, type = "o", pch = 16, col = 2)

# Mostramos gráficamente la recta para la tasa  0.01
lines(resultado03$valor_funcion, type = "o", pch = 16, col = 4)

# Agregamos una leyenda
legend(8, 5, c("eta = 0,01", "eta = 0.1"), cex = 0.8, col = c("blue", "red"),
       pch=16, lty=c(1,1))
```



Como hemos comentado anteriormente, con una tasa de aprendizaje de 0.1 se encuentra mínimos locales pero se sale de ellos, es decir, el gradiente diverge en este caso pero se encuentra muy cerca del mínimo local. Sin embargo para una tasa de aprendizaje de 0.01, nos vamos aproximando al mínimo local mientras vamos disminuyendo el valor de la función.

```r
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

2) Obtener el valor mínimo y los valores de las variables que lo alcanzan cuando el punto de inicio se fija: (2.1, 2.1), (3, 3), (1.5, 1,5), (1, 1). Generar una tabla con los valores obtenidos.

Como se especifica nada acerca de la tasa de aprendizaje, haré una comparación parecida al apartado anterior usando $\eta = 0.1$ y $\eta = 0.01$.

Punto(2.1,2.1)

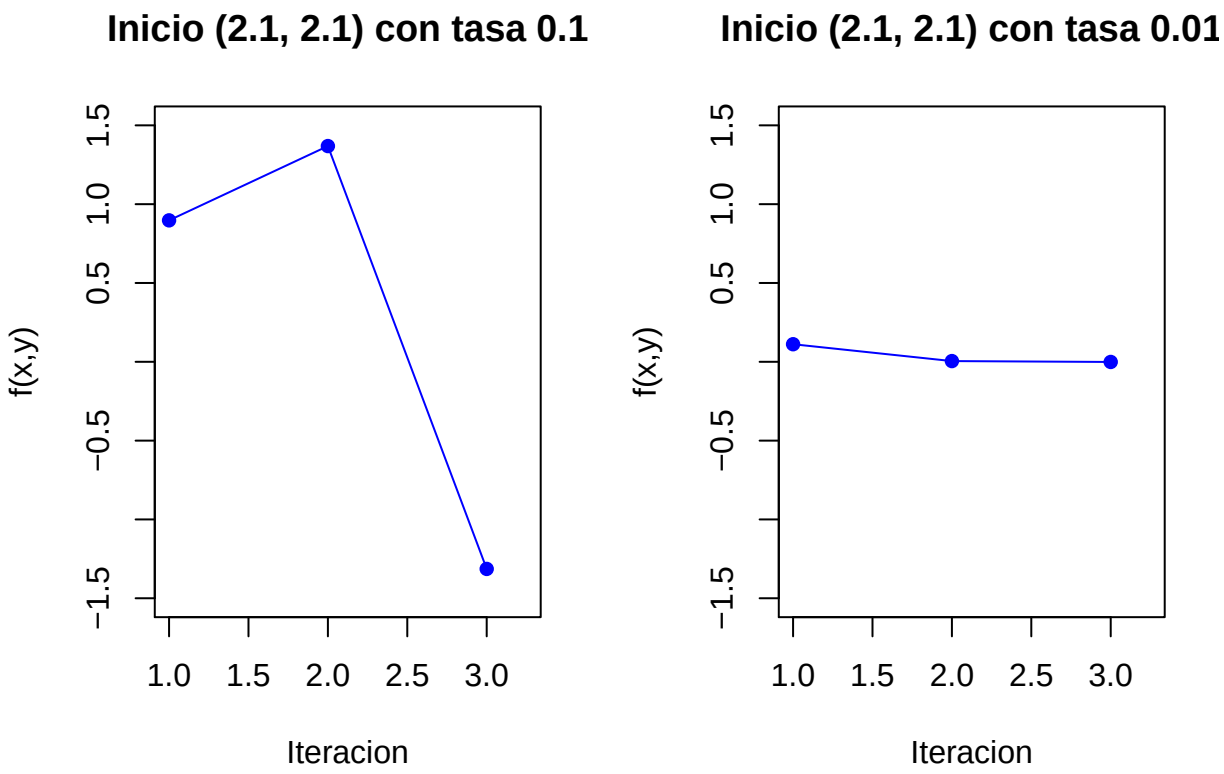
```r
# Dividimos la región de dibujo en dos partes
par(mfrow=c(1:2))

# Punto de inicio (2.1, 2.1) con tasa de aprendizaje de 0.1:
# ----------------------------------------------------------------------------
# Nos creamos una función con la tasa de aprendizaje de 0.1
resultado04 = GD(f, f_dev, vini = c(2.1,2.1), eta = 0.1, umbral = 10^(-4),
                 iteraciones_max = 50)
```

```
## Punto de partida:  2.1 2.1
## Gradiente (inicial):  6.175664 6.375664
## Valor función (inicial):  0.720983
## Gradiente (final):  0.6913102 -1.598891
## Solución:  2.739809 2.218414
## Valor función (final):  -1.313997
## Iteraciones:  2
```

```r
# Lo mostramos gráficamente
plot(resultado04$valor_funcion, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "f(x,y)", main = "Inicio (2.1, 2.1) con tasa 0.1", xlim = c(1,3.25),
     ylim = c(-1.5, 1.5))

# Punto de inicio (2.1, 2.1) con tasa de aprendizaje de 0.01:
# ----------------------------------------------------------------------------
# Nos creamos una función con la tasa de aprendizaje de 0.01
resultado05 = GD(f, f_dev, vini = c(2.1,2.1), eta = 0.01, umbral = 10^(-4),
                 iteraciones_max = 50)
```

```
## Punto de partida:  2.1 2.1
## Gradiente (inicial):  6.175664 6.375664
## Valor función (inicial):  0.720983
## Gradiente (final):  -0.1782276 0.4061031
## Solución:  2.005266 1.997608
## Valor función (final):  -0.0009552139
## Iteraciones:  2
```

```r
# Lo mostramos gráficamente
plot(resultado05$valor_funcion, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "f(x,y)", main = "Inicio (2.1, 2.1) con tasa 0.01", xlim = c(1,3.25),
     ylim = c(-1.5, 1.5))
```



Como se ve con $\eta = 0.01$, el algoritmo converge en pocas iteraciones. Pero ambos tardan el mismo número de iteraciones en encontrar un mínimo local. Aunque, como he comentado antes, para la tasa de aprendizaje 0.01 se obtiene un mínimo mientras la función va decreciendo. Y para $\eta = 0.1$, se da un gran salto (*por eso aparece el pico de arriba*), sacándonos así del entorno del mínimo, aunque en este caso encuentre un mínimo local.

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

Punto(3,3)

```
# Dividimos la región de dibujo en dos partes
par(mfrow=c(1:2))

# Punto de inicio (3, 3) con tasa de aprendizaje de 0.1:
# ------------------------------------------------------------------------------
# Nos creamos una función con la tasa de aprendizaje de 0.1
resultado06 = GD(f, f_dev, vini = c(3,3), eta = 0.1, umbral = 10^(-4),
                 iteraciones_max = 50)
```

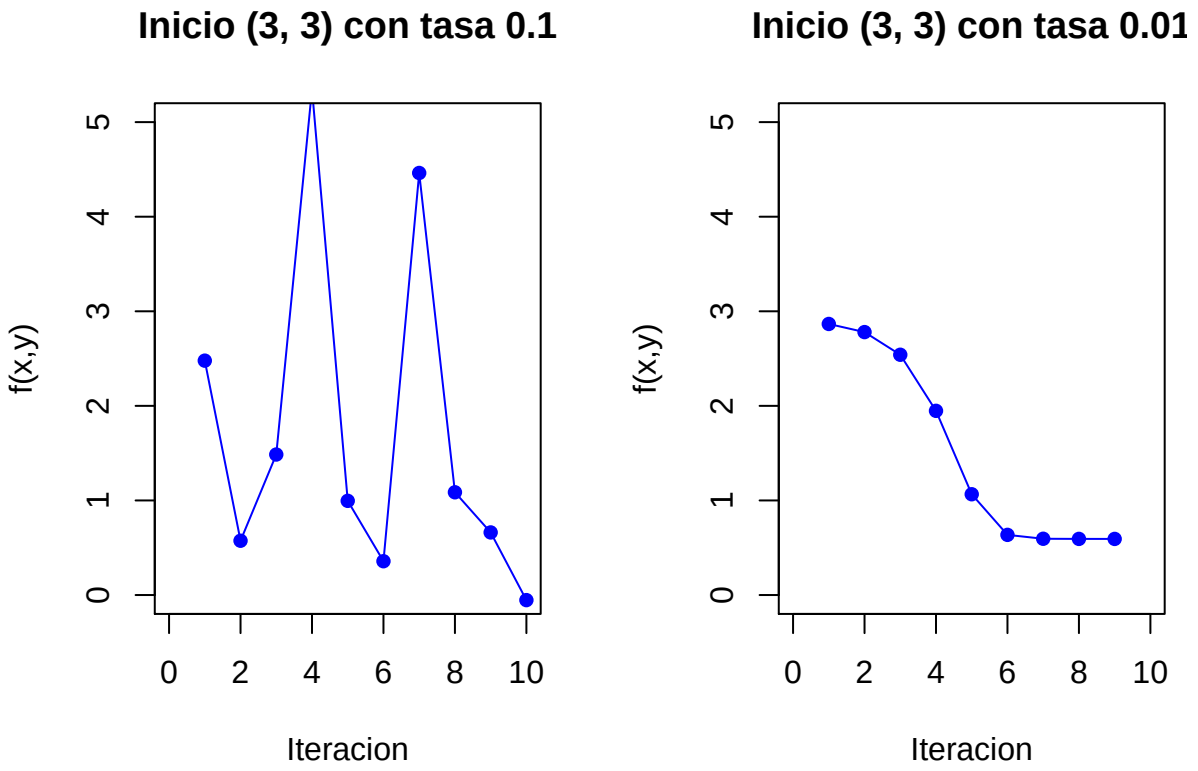
```
## Punto de partida:  3 3
## Gradiente (inicial):  2 4
## Valor función (inicial):  3
## Gradiente (final):  8.590336 -2.1004
## Solución:  2.388122 2.651642
## Valor función (final):  -0.05388005
## Iteraciones:  9
```

```
# Lo mostramos gráficamente
plot(resultado06$valor_funcion, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "f(x,y)", main = "Inicio (3, 3) con tasa 0.1", xlim = c(0,10),
```

```
     ylim = c(0,5))

# Punto de inicio (3, 3) con tasa de aprendizaje de 0.01:
# -------------------------------------------------------------------------------
# Nos creamos una función con la tasa de aprendizaje de 0.01
resultado07 = GD(f, f_dev, vini = c(3,3), eta = 0.01, umbral = 10^(-4),
                 iteraciones_max = 50)
```

```
## Punto de partida:  3 3
## Gradiente (inicial):  2 4
## Valor función (inicial):  3
## Gradiente (final):  -0.01574209 0.007690211
## Solución:  3.217871 2.7129
## Valor función (final):  0.5932713
## Iteraciones:  8
```

```
# Lo mostramos gráficamente
plot(resultado07$valor_funcion, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "f(x,y)", main = "Inicio (3, 3) con tasa 0.01", xlim = c(0,10),
     ylim = c(0,5))
```



Como se aprecia, con una tasa de aprendizaje $\eta = 0.01$ vemos que alcanza un mínimo local de manera descendente e incluso tarda menos iteraciones que para el otro caso. Sin embargo, para una tasa de aprendizaje de $\eta = 0.1$, vemos que al multiplicar por 0.1, nos está sacando de la zona donde se encuentra el mínimo.

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

Punto(1.5,1.5)

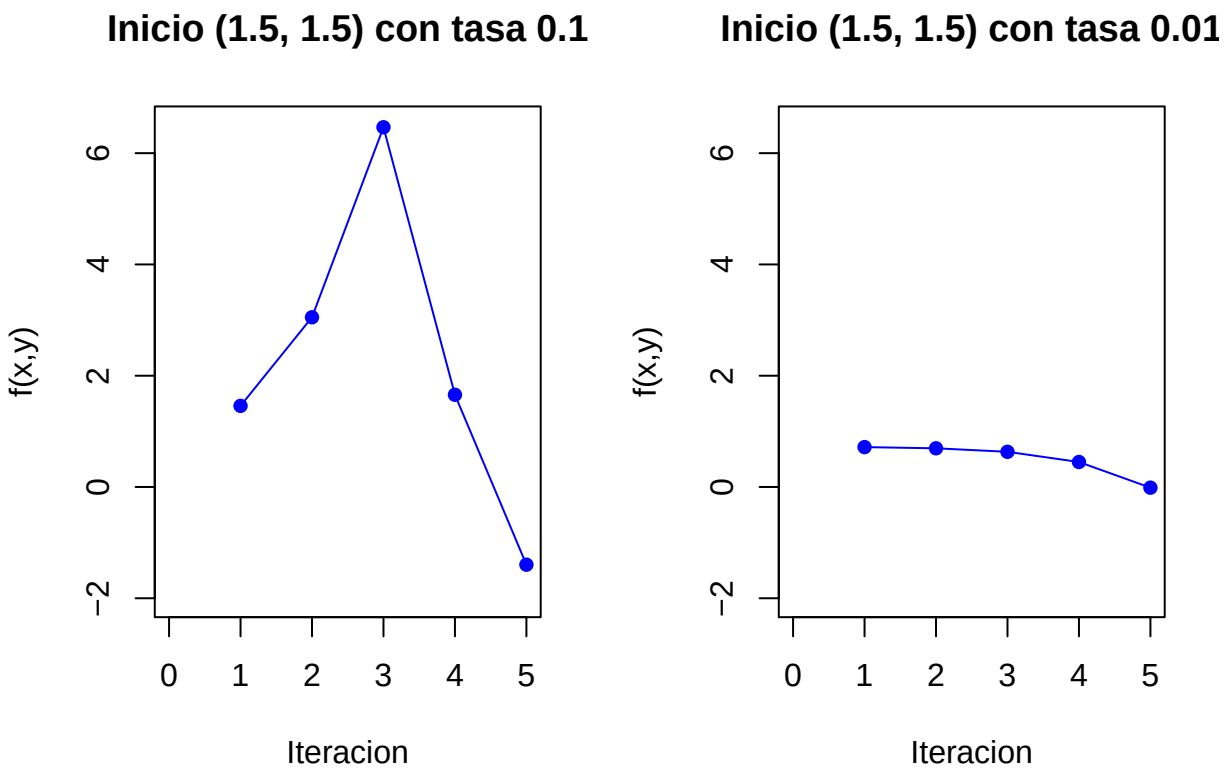
```r
# Dividimos la región de dibujo en dos partes
par(mfrow=c(1:2))

# Punto de inicio (1.5, 1.5) con tasa de aprendizaje de 0.1:
# ----------------------------------------------------------------------------
# Nos creamos una función con la tasa de aprendizaje de 0.1
resultado08 = GD(f, f_dev, vini = c(1.5,1.5), eta = 0.1, umbral = 10^(-4),
                 iteraciones_max = 50)
```

```
## Punto de partida:  1.5 1.5
## Gradiente (inicial):  -1 -2
## Valor función (inicial):  0.75
## Gradiente (final):  3.048934 -7.325005
## Solución:  2.286294 1.668486
## Valor función (final):  -1.396467
## Iteraciones:  4
```

```r
# Lo mostramos gráficamente
plot(resultado08$valor_funcion, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "f(x,y)", main = "Inicio (1.5, 1.5) con tasa 0.1", xlim = c(0,5),
     ylim = c(-2,6.5))

# Punto de inicio (1.5, 1.5) con tasa de aprendizaje de 0.01:
# ----------------------------------------------------------------------------
# Nos creamos una función con la tasa de aprendizaje de 0.01
resultado09 = GD(f, f_dev, vini = c(1.5,1.5), eta = 0.01, umbral = 10^(-4),
                 iteraciones_max = 50)
```

```
## Punto de partida:  1.5 1.5
## Gradiente (inicial):  -1 -2
## Valor función (inicial):  0.75
## Gradiente (final):  6.165684 -6.068292
## Solución:  1.419677 1.616008
## Valor función (final):  -0.01244002
## Iteraciones:  4
```

```r
# Lo mostramos gráficamente
plot(resultado09$valor_funcion, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "f(x,y)", main = "Inicio (1.5, 1.5) con tasa 0.01", xlim = c(0,5),
     ylim = c(-2,6.5))
```



Ambos encuentran el mínimo en el mismo número de iteraciones. Se ve que para $\eta = 0.01$, el algoritmo converge. Aunque como he comentado antes para la tasa de aprendizaje 0.01 se comprueba que se acerca al óptimo lentamente ya que realiza un menor salto que para la tasa 0.1.

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

Punto(1,1)

```
# Dividimos la región de dibujo en dos partes
par(mfrow=c(1:2))

# Punto de inicio (1, 1) con tasa de aprendizaje de 0.1:
# -------------------------------------------------------------------------------
# Nos creamos una función con la tasa de aprendizaje de 0.1
resultado09 = GD(f, f_dev, vini = c(1,1), eta = 0.1, umbral = 10^(-4),
                 iteraciones_max = 50)
```

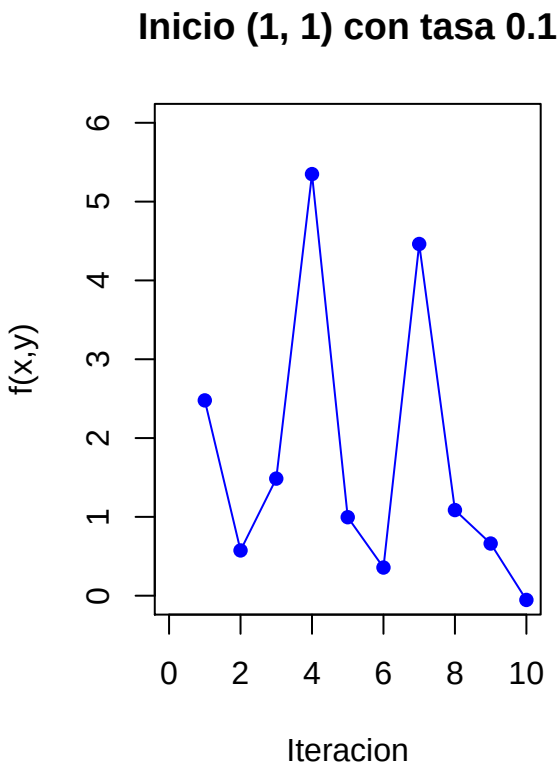
```
## Punto de partida:  1 1
## Gradiente (inicial):  -2 -4
## Valor función (inicial):  3
## Gradiente (final):  -8.590336 2.1004
## Solución:  1.611878 1.348358
## Valor función (final):  -0.05388005
## Iteraciones:  9
```

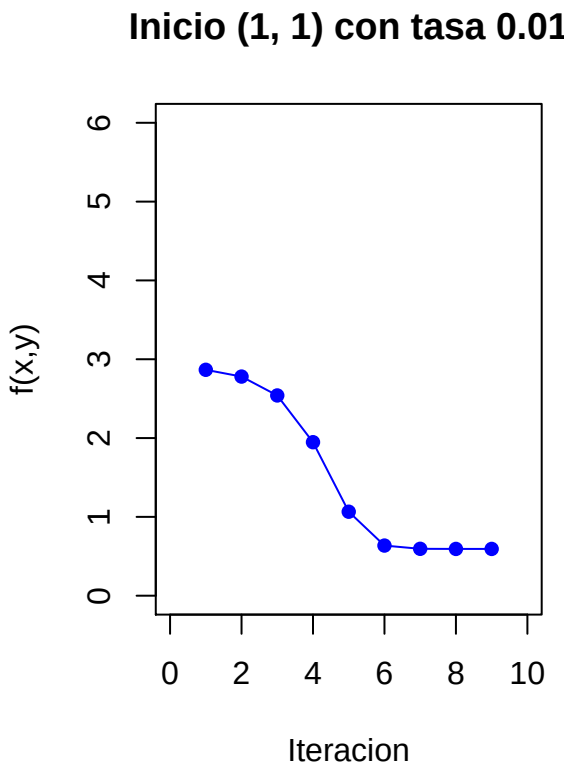
```
# Lo mostramos gráficamente
plot(resultado09$valor_funcion, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "f(x,y)", main = "Inicio (1, 1) con tasa 0.1", xlim = c(0,10),
     ylim = c(0,6))
```

```r
# Punto de inicio (1, 1) con tasa de aprendizaje de 0.01:
# ------------------------------------------------------------------------------
# Nos creamos una función con la tasa de aprendizaje de 0.01
resultado10 = GD(f, f_dev, vini = c(1,1), eta = 0.01, umbral = 10^(-4),
                 iteraciones_max = 50)
```

```
## Punto de partida:  1 1
## Gradiente (inicial):  -2 -4
## Valor función (inicial):  3
## Gradiente (final):  0.01574209 -0.007690211
## Solución:  0.7821293 1.2871
## Valor función (final):  0.5932713
## Iteraciones:  8
```

```r
# Lo mostramos gráficamente
plot(resultado10$valor_funcion, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "f(x,y)", main = "Inicio (1, 1) con tasa 0.01", xlim = c(0,10),
     ylim = c(0,6))
```





Como vemos, ocurre lo mismo que explicábamos al principio del ejercicio para el punto de inicio $(1, 1)$.

Para comprender mejor, vamos a recoger todos estos valores en una tabla.

η	(x_0, y_0)	(x_f, y_f)	$f(x_0, y_0)$	$f(x_f, y_f)$
0.01	$(2.1, 2.1)$	$(2.005266, 1.997608)$	0.720983	-0.0009552139
0.1	$(2.1, 2.1)$	$(2.739809, 2.218414)$	0.720983	-1.313997
0.01	$(3, 3)$	$(3.217871, 2.7129)$	3	0.5932713
0.1	$(3, 3)$	$(2.388122, 2.651642)$	3	-0.05388005
0.01	$(1.5, 1.5)$	$(1.419677, 1.61600)$	0.75	-0.01244002
0.1	$(1.5, 1)$	$(2.286294, 1.668486)$	0.75	-1.396467

η	(x_0, y_0)	(x_f, y_f)	$f(x_0, y_0)$	$f(x_f, y_f)$
0.01	$(1, 1)$	$(0.7821293, 1.2871)$	3	0.5932713
0.1	$(1, 1)$	$(1.611878, 1.348358)$	3	-0.05388005

c) ¿Cuál sería su conclusión sobre la verdadera dificultad de encontrar el mínimo global de una función arbitraria?

A la vista de la tabla, vemos que encontrar un mínimo local es díficil, ya que depende de varios parámetros como el punto de inicio o la tasa de aprendizaje. Acabamos de comprobar que con una tasa de aprendizaje muy grande podemos diverger y con una tasa de aprendizaje muy pequeña podemos llegar a converger o no converger, ya que nuestro gradiente descendente apenas avanzaría. Básicamente, lo que acabamos de ver en las gráficas de antes, es que con un η pequeño, nos vamos acercando al óptimo lentamente. Pero con un η más grande, como estamos dando saltos más amplios, puede que en uno de esos avances nos salgamos de la región del mínimo o puede que gracias a esos avances podamos encontrar el óptimo más rápido.

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

Ejercicio 2

Regresión Logística: En este ejercicio crearemos nuestra propia función objetivo f (una 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 la etiqueta y es una función determinista de x.

Consideremos $d = 2$ para que los datos sean visualizables, y sea $X = [0, 2] \times [0, 2]$ con probabilidad uniforme de elegir cada $x \in X$. Elegir una línea en el plano que pase por X 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 y_n de todos ellos respecto de la frontera elegida.

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, $1, 2, ..., N$, en el orden de los datos antes de usarlos en cada época del algoritmo.
- Usar una tasa de aprendizaje de $\eta = 0.01$

Primero, vamos a implementar **Regresión Logística**, utilizando el **Algoritmo de Gradiente Descendente Estocástico (SGD)**, para eso hacemos uso de las transparencias de clase (*página 29 de la Sesión 5 de Teoría*) y del libro *Learning from data: A short course*.

El método SGD, lo que hace es corregir el error, para ello en vez de recalcular los pesos usando todas las muestras, utiliza solo una muestra elegida de manera aleatoria de la muestra global. Nuestro algoritmo parará cuando alcancemos un máximo de iteraciones o cuando la norma de la diferencia entre $w^{(t-1)} - w^{(t)}$ sea menor que un determinado valor. Hay que saber que la norma de un vector es la distancia euclídea (en línea recta) entre dos puntos A y B que delimitan dicho vector. Además, el hecho de que introduzcamos permutaciones, es lo que hace que sea que el gradiente sea *estocástico*.

Para la regresión logística aplicamos la siguiente expresión que nos calculará el error:

$$g_t = \frac{1}{t}\sum_{n=1}^{N}\frac{y_n x_n}{1+e^{y_n w(t)^T x_n}}$$

Antes de implementar el algoritmo, debemos cargar las funciones para generar una recta *simula_recta* y para generar la gráfica de nube de puntos $simula\_unif$. Ambas funciones ya fueron creadas y explicadas en la práctica 1, por lo que no me detendré en su explicación.

```
# Generación de nube de puntos de manera uniforme
simula_unif = function (N = 2, dim = 2, rango = c(0,1)){
 m = matrix(runif(N*dim, min=rango[1], max=rango[2]),nrow = N, ncol=dim, byrow=T)
 m
}

# Generaración de una recta
simula_recta = function (intervalo = c(-1,1), visible=F){

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

  # calculamos la pendiente
  a = (ptos[1,2] - ptos[2,2]) / (ptos[1,1]-ptos[2,1])

  # Calculamos el punto de corte
  b = ptos[1,2]-a*ptos[1,1]

  # Se pinta la recta y los 2 puntos
  if (visible) {

    # Si no esta abierto el dispositivo lo abre con plot
    if (dev.cur()==1)
      plot(1, type="n", xlim=intervalo, ylim=intervalo)

    # Pinta en verde los puntos
    points(ptos,col=3)

    # y la recta
    abline(b,a,col=3)
  }

  # Devuelve el par pendiente y punto de corte
  c(a,b)
}
```

Hecho esto, ya podemos empezar a implementar el algoritmo de **Regresión Logística**, basándonos en el **Gradiente Descendente Estocástico**.

```
# Función que calcula la norma
norma <- function(w_old, w_new){
  sqrt(sum((w_old - w_new)^2))
}
```

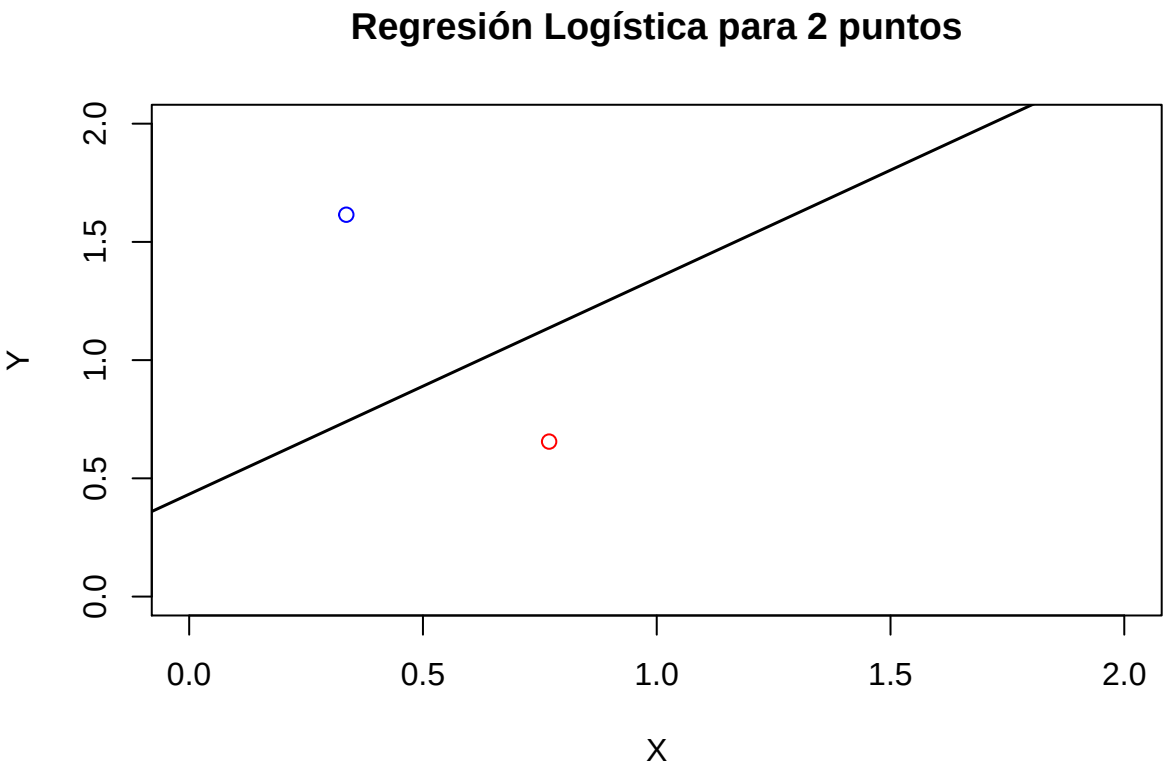
```r
# Algoritmo de Regresión Logística
RL <- function(datos, etiquetas, vini = c(0,0,0), eta = 0.01, iteraciones_max = 500){

  # Inicializar los pesos
  w_new <- vini
  w_old <- c(0,0,0)

  # Variable que lleva el número de iteraciones realizadas
  iter <- 0

  # Añadimos una columna para poder hacer el producto vectorial
  datos <- cbind(1, datos)

  # Variable booleana que nos servirá para salir o continuar en el bucle
  continuar <- FALSE

  # La condición de parada será cuando se hayan completado un número máximo de
  # iteraciones o cuando la norma de la diferencia entre dos puntos sea menor
  # que un determinado valor
  while (iter <= iteraciones_max | !continuar){

    # Asignamos el valor nuevo, al antiguo
    w_old <- w_new

    # Hacemos una permutación aleatoria de los datos
    permutacion <- sample(1:length(etiquetas))

    # Recorremos la permutación
    for(i in permutacion){

      # Calculamos el gradiente (basándonos en la fórmula explicada anteriormente)
      gradiente <- (-etiquetas[i]*datos[i,]) /
        (1 + exp(etiquetas[i] * w_new %*%datos[i,]))

      # Actualizamos los pesos
      w_new <- w_new - eta * gradiente
    }

    # Incrementamos el valor de la iteración, cuando hacemos una pasada a las muestras
    iter <- iter + 1

    # Si la norma de la diferencia entre dos puntos es menor que un determinado valor
    # salirnos del bucle (condición de parada)
    # Esta condición, no la meto dentro del while, ya que al principio ambos valores
    # son 0 por lo que nunca entraría en el bucle
    if(norma(w_old, w_new) < eta ){
      continuar <- TRUE
    }
  }

  cat("Pesos: ", w_new, "\nValores de la Recta: ", -w_new[1]/w_new[3], -w_new[2]/w_new[3],
       "\nIteraciones: ", iter-1)
}
```

Primero vamos a comprobar si para dos muestras el algoritmos las separa linealmente.

```
# Cuando se utilizan números aleatorios, es recomedable establecer una semilla, para
# que al repetir la generación de los números, siempre obtengamos los mismos
set.seed(3)

# Generamos aleatoriamente una lista de números aleatorios
datos01 = simula_unif(N=2, dim=2, rango=c(0,2))

# Generamos una recta
recta01 = simula_recta(intervalo=c(0,2))

# Aplicamos la función a cada punto y almacenamos el signo del resultado en una lista
etiquetas01 = sign(datos01[,2] - recta01[1]*datos01[,1] - recta01[2])

# Obtenemos los valores de la Regresión Logística
RL(datos01, etiquetas01, c(0,0,0), 0.01, 500)
```

```
## Pesos:  -0.5411461 -1.142061 1.249963
## Valores de la Recta:  0.4329298 0.9136762
## Iteraciones:  500
```

```
# Mostramos gráficamente los puntos
plot (datos01, col = etiquetas01+3, xlab = "X", ylab = "Y",  xlim = c(0,2),
      ylim = c(0,2), main = "Regresión Logística para 2 puntos")
# Dibujamos la recta
abline(0.4329298, 0.9136762, lwd = 1.5)
```



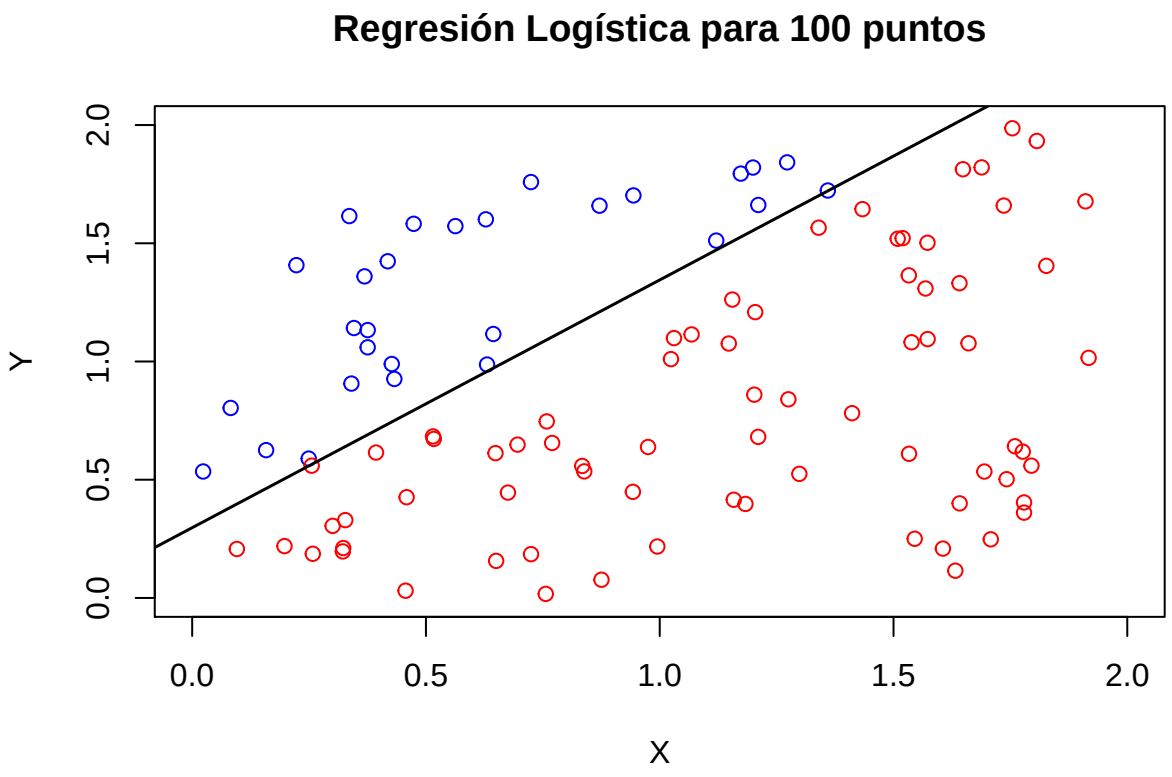
```r
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

Una vez comprobado para $N = 2$, vamos a comprobarlo para $N = 100$:

```r
# Cuando se utilizan números aleatorios, es recomedable establecer una semilla, para
# que al repetir la generación de los números, siempre obtengamos los mismos
set.seed(3)

# Generamos aleatoriamente una lista de números aleatorios
datos02 = simula_unif(N=100, dim=2, rango=c(0,2))

# Generamos una recta
recta02 = simula_recta(intervalo=c(0,2))

# Aplicamos la función a cada punto y almacenamos el signo del resultado en una lista
etiquetas02 = sign(datos02[,2] - recta02[1]*datos02[,1] - recta02[2])

# Obtenemos los valores de la Regresión Logística
RL(datos02, etiquetas02, c(0,0,0), 0.01, 500)
```

```
## Pesos:  -2.313951 -8.1601 7.78831
## Valores de la Recta:  0.2971057 1.047737
## Iteraciones:  500
```

```r
# Lo mostramos gráficamente
plot (datos02, col = etiquetas02+3, xlab = "X", ylab = "Y",  xlim = c(0,2),
      ylim = c(0,2), main = "Regresión Logística para 100 puntos")
# Dibujamos la recta
abline(0.2971057, 1.047737, lwd = 1.5)
```



```r
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

b) Usar la muestra de datos etiquetada para encontrar nuestra solución g y estimar E_{out} usando para ello un número suficientemente grande de nuevas muestras (>999).

Ahora simplemente, volvemos a escribir nuestra función de Regresión Logística de nuevo para que nos devuelva solo los pesos y así poder utilizarlos para estimar el error. Ya que ahora no queremos que nos muestre por pantalla todos los resultados que imprimíamos antes. Debemos tener claro que la función será la misma que la de antes. Lo único que difiere es que guardo los pesos en un vector, al final.

```r
# Algoritmo de Regresión Logística (1)
RL1 <- function(datos, etiquetas, vini = c(0,0,0), eta = 0.01, iteraciones_max = 500){

  # Inicializar los pesos
  w_new <- vini
  w_old <- c(0,0,0)

  # Variable que lleva el número de iteraciones realizadas
  iter <- 0

  # Añadimos a una columna para poder hacer el producto vectorial
  datos <- cbind(1, datos)

  # Variable booleana que nos servirá para salir o continuar en el bucle
  continuar <- FALSE

  # La condición de parada cuando se hayan completado un número máximo de iteraciones
  # o cuando la norma de la diferencia entre dos puntos sea menor que un determinado
  # valor
  while (iter <= iteraciones_max | !continuar){

    # Asignamos el valor nuevo, al antiguo
    w_old <- w_new

    # Hacemos una permutación aleatoria de los datos
    permutacion <- sample(1:length(etiquetas))

    # Recorremos la permutación
    for(i in permutacion){

      # Calculamos el gradiente (basándonos en la fórmula explicada anteriormente)
      gradiente <- (-etiquetas[i]*datos[i,]) / (1 + exp(etiquetas[i] * w_new %*%datos[i,]))

      # Actualizamos los pesos
      w_new <- w_new - eta * gradiente
    }

    # Incrementamos el valor de la iteración, cuando hacemos una pasada a las muestras
    iter <- iter + 1

    # Si la norma de la diferencia entre dos puntos es menor que un determinado valor
```

```
    # salirnos del bucle (condición de parada)
    # Esta condición, no la meto dentro del while, ya que al principio ambos valores son 0
    # por lo que nunca entraría en el bucle
    if(norma(w_old, w_new) < eta ){
      continuar <- TRUE
    }
  }

  c(w_new, iter-1)
}
```

Ahora vamos a generar una nueva muestra de test y calculamos los errores con respecto la muestra y con respecto al conjunto de test.

Para obtener el error dentro de la muestra E_{in}, partimos de los datos de entrenamiento, es decir, de las etiquetas que son distintas de entre todas las etiquetas. Lo que queremos hacer es predecir cuántas veces nos equivocamos al predecir la etiqueta.

$$E_{in} = etiquetas.train! = clasificador(train)/etiquetas.total$$

```
# Cuando se utilizan números aleatorios, es recomedable establecer una semilla, para
# que al repetir la generación de los números, siempre obtengamos los mismos
set.seed(3)

# Generamos las etiquetas aleatoriamente
datos03 <- simula_unif(N = 1000, dim = 2, rango = c(0,2))

# Generamos una recta
recta03 <- simula_recta(intervalo = c(0,2))

# Etiquetamos los puntos a partir de la función de la recta
etiquetas03 <- sign(datos03[,2] - recta03[1]*datos03[,1] - recta03[2])

# Calculamos la recta de regresión que separa ambas clases
recta_regresion01 <- RL1(datos03, etiquetas03)
recta_regresion02 = c(-recta_regresion01[1]/recta_regresion01[3],
                      - recta_regresion01[2]/recta_regresion01[3])

# Etiquetamos los puntos a partir de la recta de regresión
etiquetas_regresion01 <- sign(datos03[,2] - recta_regresion02[1]*datos03[,1]
                              - recta_regresion02[2])

# Calculamos Ein como las etiquetas que son distintas, partido el total de las etiquetas
Ein01 <- 0
Ein01 = sum(etiquetas03 != etiquetas_regresion01) / length(etiquetas03)

Ein01
```

```
## [1] 0.688
```

Cuando haya obtenido el valor de E_{out}, explicaré ambos resultado. Por eso, procedemos a calcular E_{out}, que es el error fuera de la muestra y se calcula a partir de los datos de train con el conjunto de test. Para E_{out}, usaremos la recta de regresión usada en la muestra de entrenamiento.

$$E_{out} = etiquetas.test! = clasificador(test)/etiquetas.total$$

```
# Cuando se utilizan números aleatorios, es recomedable establecer una semilla, para
# que al repetir la generación de los números, siempre obtengamos los mismos
set.seed(3)

# Generamos los datos aleatoriamente
datos03.test <- simula_unif(N = 1000, dim = 2, rango = c(0,2))

# Generamos una recta
recta03.test <- simula_recta(intervalo = c(0,2))

# Etiquetamos las etiquetas a partir de la función de la recta
etiquetas03.test <- sign(datos03.test[,2] - recta03.test[1]*datos03.test[,1]
                         - recta03.test[2])

# Usamos la recta de regresión creada en los datos de entrenamiento
etiquetas_regresion01.test <- sign(datos03.test[,2]
                                   - recta_regresion02[2]*datos03.test[,1]
                                   - recta_regresion02[2])

# Calculamos Eout como las etiquetas que son distintas, partido el total de las etiquetas
Eout01 <- 0
Eout01 = sum(etiquetas03.test != etiquetas_regresion01.test) / length(etiquetas03.test)

Eout01
```

```
## [1] 0.515
```

Vemos como obtenemos un error elevado, esto puede sugerirnos que la regresión logística no es adecuada con esa tasa de aprendizaje. Como vemos, obtenemos un error parecido dentro que fuera de la muestra, esto puede ser lógico ya que para sacar E_{in} partimos de unos datos distribuidos de una manera y que no tienen ruido. Y para sacar E_{out} partimos de unos nuevos datos que siguen la misma distribución que E_{in} y también son sin ruido. Por tanto, la función que hace la regresión es prácticamente la verdadera f. Entonces, al ser ambos errores parecidos, podemos decir que las muestras son i.i.d (independientes e idénticamente distribuidas).

***Nota**: Hay que tener en cuenta, que cuando obtenemos los errores en una sola iteración, los resultados obtenidos puedes ser incoherentes.*

Ejercicio 3

Clasificación de Dígitos. Considerar el conjunto de datos de los dígitos manuscritos y seleccionar las muestras de los dígitos 4 y 8. 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.

a) Plantear un problema de clasificación binaria que considere el conjunto de entrenamiento como datos de entrada para aprender la función g.

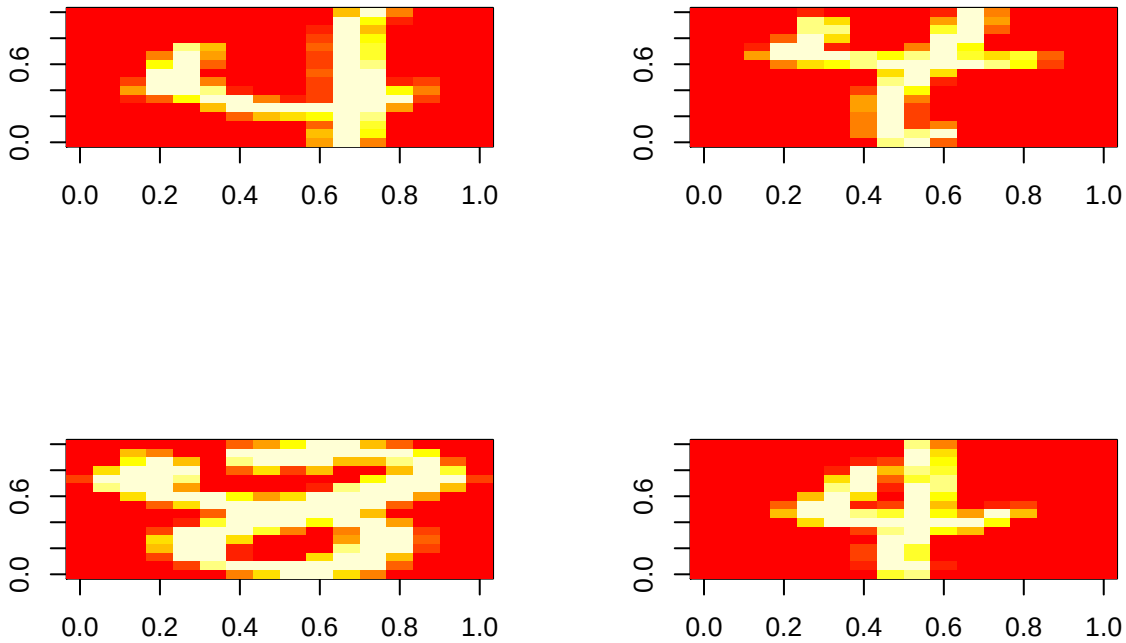
Para plantear el problema de clasificación binaria, partimos del código proporcionado por la profesora que aparece en *paraTrabajo.Rmd*. Básicamente, lo que hacemos es leer el fichero *Zip.train* y *Zip.test* y visualizar las imágenes (guardándolas como matrices de tamaño 16 x 16). Lo mismo que hicimos en la anterior práctica.

Para leer el fichero *Zip.train*

```
# Leemos el fichero con los datos de entrenamiento
digit.train <- read.table("datos/zip.train", quote="\"", comment.char="",
                          stringsAsFactors=FALSE)
```

```
## Warning in scan(file = file, what = what, sep = sep, quote = quote, dec =
## dec, : número de items leídos no es múltiplo del número de columnas
```

```
# Guardamos los 4 y los 8
digitos48.train = digit.train[digit.train$V1==4 | digit.train$V1==8,]

# Etiquetas
digitos.train = digitos48.train[,1]
ndigitos.train = nrow(digitos48.train)

# Se retira la clase y se monta una matriz de tamanio 16x16
grises = array(unlist(subset(digitos48.train,select=-V1)),c(ndigitos.train,16,16))
grises.train = lapply(seq(dim(grises)[1]), function(m) {matrix(grises[m,,],16)})

# Visualizamos las imágenes
par(mfrow=c(2,2))

for(i in 1:4){
  imagen = grises[i,,16:1] # Se rotan para verlas bien
  image(z=imagen)
}
```




```
# Etiquetas correspondientes a las 4 imágenes
digitos.train[1:4]
```

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

```
# Guardamos las etiquetas
etiquetas48.train <- digitos48.train$V1
```

```r
# Convertimos las etiquetas de 4 a -1 y de 8 a 1
# [(4-6) / 2 = -1] y [(8-6) / 2 = 1]
etiquetas48.train <- (etiquetas48.train-6)/2
```

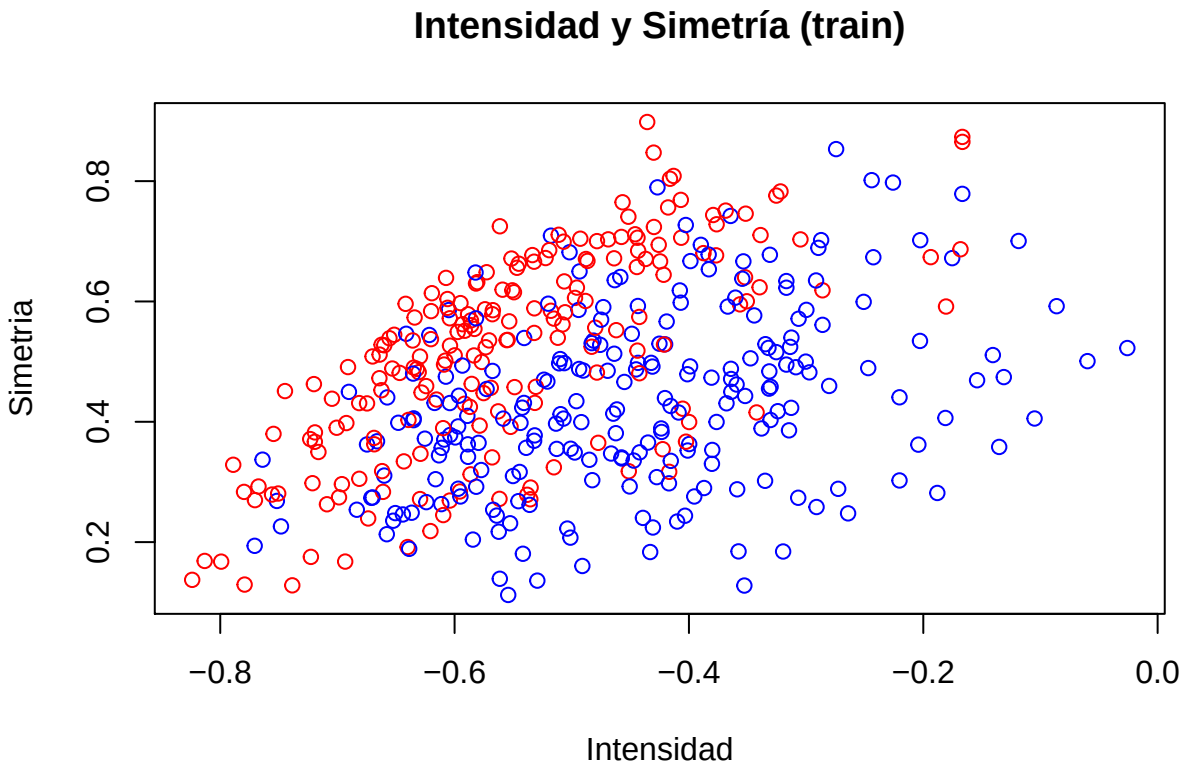
```r
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

A continuación, vamos a extraer las características de **intensidad promedio** y **simetría** para los datos *train* de la misma manera que hicimos en la práctica 1.

```r
# INTENSIDAD ------------------------------------------------------------------
# Creamos un array con la media de la imagen
# Para calcular la intensidad, se devuelve el número medio de blancos y
# negros (grises).
intensidad.train = unlist(lapply(grises.train, FUN=mean))

# SIMETRÍA --------------------------------------------------------------------
# Creamos una función para calcular la simetría
simetria <- function(matriz){

  # Matriz_original
  matriz_original = matriz[1:256]

  # Calculamos una nueva imagen invirtiendo el orden de las columnas
  matriz_invertida = matriz[,ncol(matriz):1]

  # Calculamos la diferencia entre la matriz original y la matriz invertida
  diferencia_matriz = matriz_original - matriz_invertida

  # Calculamos la media global de los valores absolutos de la matriz
  simetria = mean(abs(diferencia_matriz))

  simetria
}

# Guardamos la simetria de la imagen
simetria.train = unlist(lapply(grises.train, simetria))
```

Ahora, procedemos a representar en los ejes $X = Intensidad\ Promedio$ e $Y = Simetría$, las instacias seleccionadas de $4's$ y $8's$ para *train*.

```r
# Dibujamos la nube de puntos
plot(x=intensidad.train, y=simetria.train, col=etiquetas48.train+3, xlab = "Intensidad",
     ylab = "Simetria", main="Intensidad y Simetría (train)")
```



Vemos que la instancia 8, tiene el color azul y que la instancia 4 es el color rojo. A simple vista, ya podemos ver que los datos no van a ser separables linealmente.

***Nota**: No entraré en más detalles en este apartado y no representaré la recta que 'separa' ambas instancias, puesto que el siguiente aparatado se explicará.*

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

Ahora hacemos el mismo procedimiento para los datos *test*.

Para leer el fichero *Zip.test*

```
# Leemos el fichero con los datos de test
digit.test <- read.table("datos/zip.test", quote="\"", comment.char="",
                         stringsAsFactors=FALSE)
```

```
## Warning in scan(file = file, what = what, sep = sep, quote = quote, dec =
## dec, : número de items leídos no es múltiplo del número de columnas
```

```
# Guardamos los 1 y los 5
digitos48.test = digit.test[digit.test$V1==4 | digit.test$V1==8,]

# Etiquetas
digitos.test = digitos48.test[,1]
ndigitos.test = nrow(digitos48.test)

# Se quita la clase y se monta una matriz 16x16
grises = array(unlist(subset(digitos48.test,select=-V1)),c(ndigitos.test,16,16))
grises.test = lapply(seq(dim(grises)[1]), function(m) {matrix(grises[m,,],16)})
```

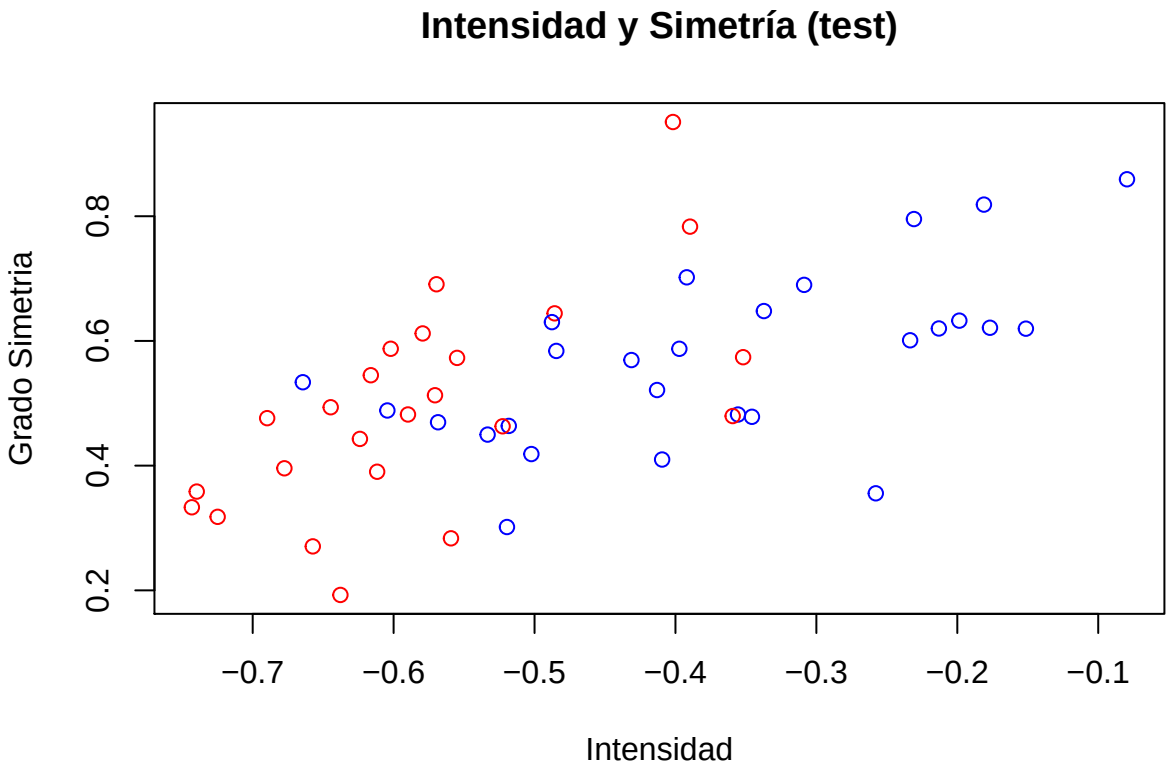
```
# Recordamos que las etiquetas que tenemos son 4 y 8
# por tanto las pasamos a -1 y 1
etiquetas48.test <- digitos48.test$V1
etiquetas48.test <- (etiquetas48.test-6)/2
```

Volvemos a extraer las características de **intensidad promedio** y **simetría** para esta vez para los datos *test*.

```
# INTENSIDAD ------------------------------------------------------------
# Creamos un array con la media de la imagen
intensidad.test = unlist(lapply(grises.test, FUN=mean))

# SIMETRÍA --------------------------------------------------------------
# Creamos uan función para calcular la simetría
simetria <- function(matriz){
  # Matriz_original
  matriz_original = matriz[1:256]

  # Calculamos una nueva imagen invirtiendo el orden de las columnas
  matriz_invertida = matriz[,ncol(matriz):1]

  # Calculamos la diferencia entre la matriz original y la matriz invertida
  diferencia_matriz = matriz_original - matriz_invertida

  # Calculamos la media global de los valores absolutos de la matriz
  simetria = mean(abs(diferencia_matriz))

  simetria
}

# Creamos un array con la simetría de la imagen
simetria.test = unlist(lapply(grises.test, simetria))
```

Ahora, procedemos a representar en los ejes $X = Intensidad\ Promedio$ e $Y = Simetría$, las instacias seleccionadas de $4's$ y $8's$ para el *test*.

```
# Dibujamos los puntos
plot(x=intensidad.test, y=simetria.test, col=etiquetas48.test+3, xlab="Intensidad",
     ylab="Grado Simetria", main="Intensidad y Simetría (test)")
```



Vemos que la instancia 8, tiene el color azul y que la instancia 4 es el color rojo.

***Nota**: No entraré en más detalles en este apartado y no representaré la recta que 'separa' ambas instancias, puesto que el siguiente aparatado se explicará.*

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

b) Usar un modelo de Regresión Lineal y aplicar PLA-Pocket como mejora. Responder a las siguientes cuestiones.

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

Primero cargamos el método de **Regresión Lineal** que hicimos en la anterior práctica:

```
Regress_Lin <- function(datos, label) {
  # Añadimos una columna a los datos, para hacer el producto
  x <- cbind(1, data.matrix(datos))

  # Obtenemos la transformación SVD
  x.svd <- svd(x)

  # Descomposición de X en valores singulares
  v <- x.svd$v
  d <- x.svd$d

  # Comprobamos todos los términos de la diagonal
```

```
  diagonal <- diag(ifelse(d>0.0001, 1/d, d))

  # Calculamos (X^T) * X = V * (D^2) * (V^T)
  xx <- v %*% (diagonal^2) %*% t(v)

  # Calculamos la pseudoinversa: (X^T) * etiqueta
  pseudoinversa <- xx %*% t(x)

  # Calculamos wlin
  w <- pseudoinversa %*% as.numeric(label)

  # Devolvemos los coeficientes del hiperplano: w1 + w2*x + w3*y = 0
  c(w)
}
```

Luego, procedemos a implementar el **PLA_Pocket**, que básicamente consiste en implementar el **PLA**. Se basa en que cuando coge un w que es distinto, va recalculando el $w\_current$; pero no lo podemos dar por bueno a la primera de cambio, y solo lo cambiaremos si $E\_in = 0$. Es decir, no cambiamos los pesos hasta que no sepamos con seguridad que mejora.

Primero procedemos a calcular el E_{in} para la función *PLA Pocket*, que consiste en calcular la etiqueta de sus datos con respecto a los pesos, y comparar cuantas veces nos equivocamos con respecto a la verdadera etiqueta.

```
# Función que calcula Ein
Ein <- function(datos, label, pesos){

    # Se obtiene el signo de los datos con respecto a los pesos
    signo =  sign(datos%*% pesos)

    # Comparamos en cuantas te equivocas con respecto a la verdadera etiqueta
    # nrow(datos) -> para que sea tanto por uno
    error <- sum (signo != sign(label)) / nrow(datos)

    # Devolvemos el error
    error
}
```

Vamos a implementar nuestro **PLA_Pocket**:

```
PLA_Pocket <- function(datos, label, vini, iteraciones_max) {

  # Asigamos a w el valor inicial del vector
  w <- vini

  # Variable que contará el número de iteraciones para parar el PLA
  # en caso de que no sea linealmente separable
  iteraciones = 1

  # Variable booleana que nos permitirá seguir o no,
  # viendo cuando se ha cambiado el valor o no
  continuar = TRUE

  # Añadimos a la matriz de datos, una tercera columna para poder hacer
  # el producto
```

```
  datos = cbind(datos,1)

  # Calculamos el error al principio
  error = Ein(datos, label, w)

  # Iterar en las muestra, mejorando la solución
  while ((iteraciones <= iteraciones_max) & continuar) {
    continuar = FALSE

    # Hacemos un bucle interno iterando sobre cada dato
    # cogiendo para ello los índices aleatoriamente
    for(i in (sample(nrow(datos)))) {

      # Obtenemos el signo del producto vectorial de los datos por los pesos
      signo = sign(datos[i,] %*% w)

      # Comparamos si el signo obtenido antes, coincide con el de la etiqueta
      if (signo != label[i]) { # Sino coincide

        # Actualizamos el peso: w_new = w_old + x*etiqueta
        w_current = w + datos[i,]*label[i]

        # Cambiamos el valor de la variable booleana
        continuar <- TRUE

        # Después de actualizar los pesos, obtenemos el nuevo error
        nuevo_error = Ein(datos, label, w_current)

        # Comparamos si el error del principio es menor o igual
        # que el nuevo
        if(error > nuevo_error){

          # Los mostramos por pantalla
          cat("Antiguo error = ", error, "\n")
          cat("Nuevo error = ", nuevo_error, "\n")

          # Actualizamos los valores
          w = w_current
          error = nuevo_error
        }
      }
    }

    # Incrementamos el número de iteraciones
    iteraciones = iteraciones + 1
  }

  # Devolvemos los parámetros del hiperplano del perceptrón
  # y el numero de iteraciones
  c(w)
}
```

Para comprobar si nuestro *PLA Pocket* sirve como mejora, lo que haremos será:

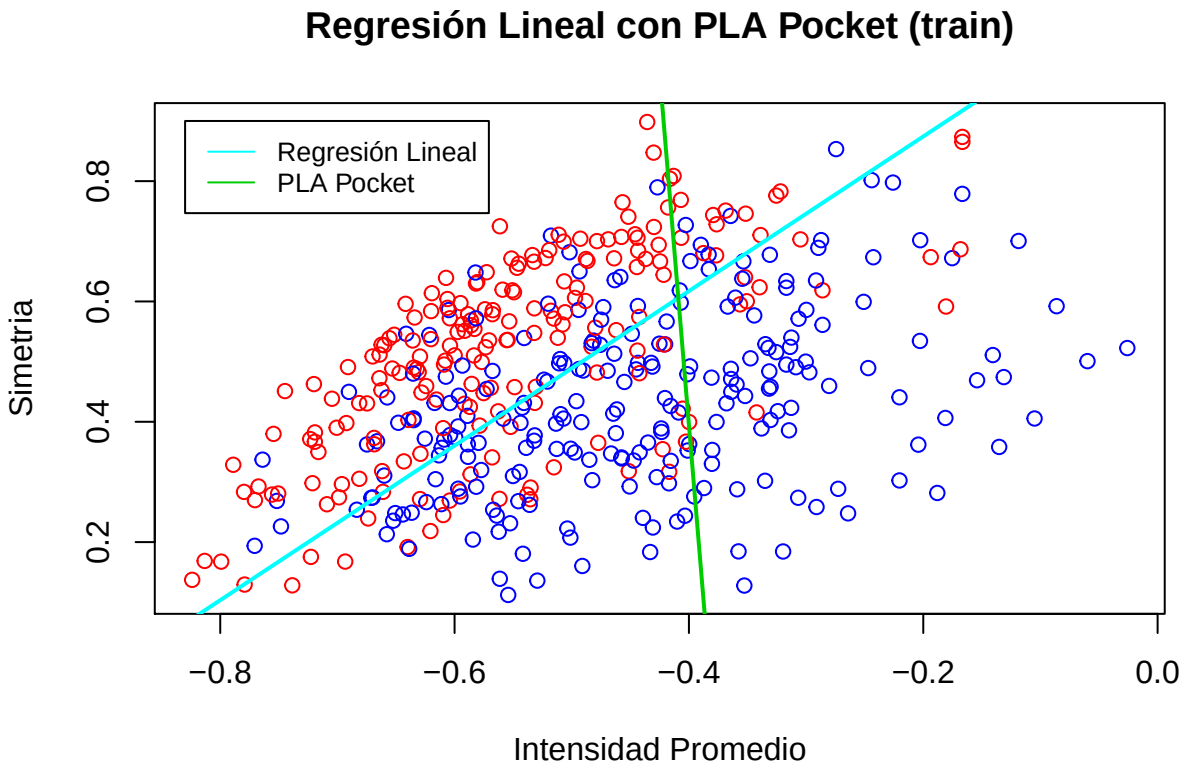
1. Aplicar *Regresión Lineal*, obteniendo así una aproximación para la separación de datos.
2. Los pesos obtenidos de la regresión lineal, son introducidos al *PLA Pocket*. Obteniendo así una separación mejor.

1. Generaremos la gráfica para los datos de entrenamiento:

```
# Cuando se utilizan números aleatorios, es recomedable establecer una semilla, para
# que al repetir la generación de los números, siempre obtengamos los mismos
set.seed(3)

# Juntamos los valores obtenidos de simetria e intensidad del train
datos.train <- cbind(intensidad.train, simetria.train)

# Dibujamos los puntos del train
plot(datos.train, xlab = "Intensidad Promedio", ylab = "Simetria",
     col = etiquetas48.train+3, main = "Regresión Lineal con PLA Pocket (train)")

# Hacemos Regresión Lineal
pesos01.train <- Regress_Lin(datos.train, etiquetas48.train)

# -pesos01.train[1]/pesos01.train[3], -pesos01.train[2]/pesos01.train[3], es mi 'g',
# es decir, la función que intenta aproximar a la verdadera funcion 'f'

# Dibujamos la línea que separa los puntos (mediante regresión lineal)
abline(-pesos01.train[1]/pesos01.train[3], -pesos01.train[2]/pesos01.train[3],
       col = 5, lwd = 2) # Línea color azul es de regresión lineal

# Hacemos el PLA Pocket
pesos02.train <- PLA_Pocket(datos.train, etiquetas48.train, pesos01.train, 100)
```

```
## Antiguo error =  0.5347222
## Nuevo error =  0.5324074
## Antiguo error =  0.5324074
## Nuevo error =  0.5277778
## Antiguo error =  0.5277778
## Nuevo error =  0.4606481
```

```
# Dibujamos la línea que separa los puntos (mediante PLA Pocket)
abline(-pesos02.train[1]/pesos02.train[3], -pesos02.train[2]/pesos02.train[3],
       col = 3, lwd = 2) # Línea color verde es de PLA Pocket

# Agregamos una leyenda
legend(-0.83, 0.9, c("Regresión Lineal", "PLA Pocket"), cex = 0.8, col = c(5, 3),
      lty=c(1,1))
```



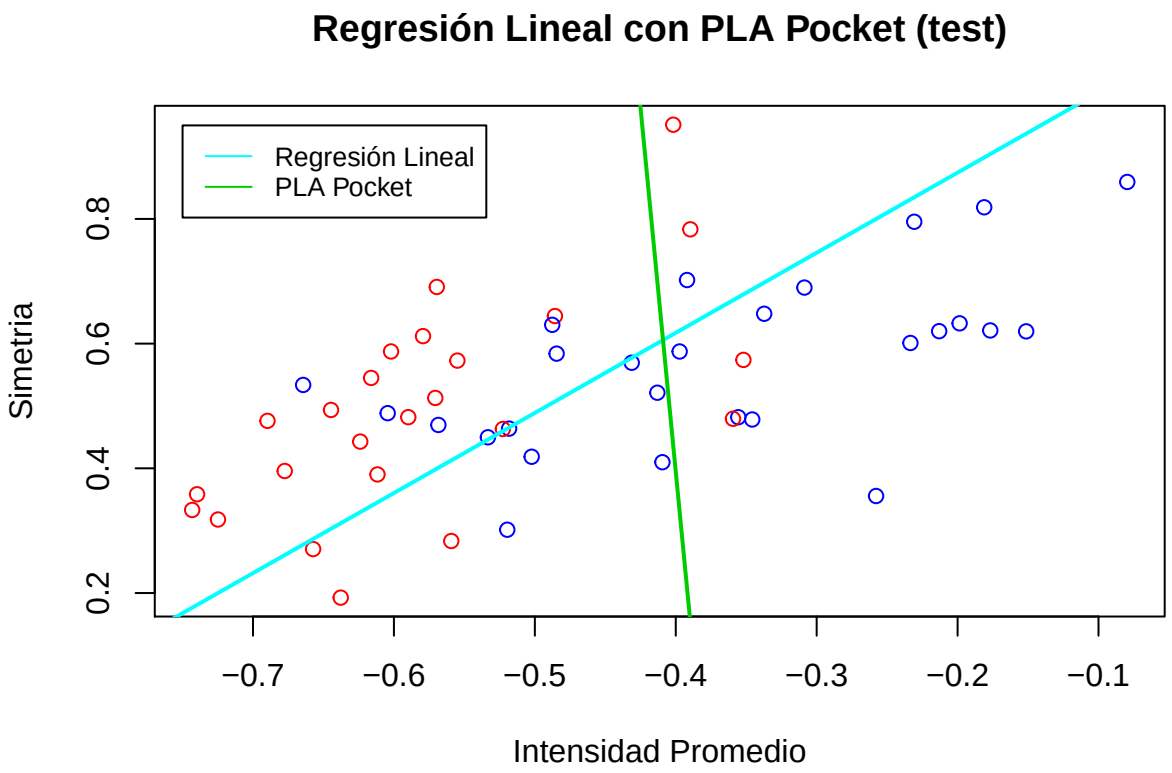
Como podemos ver en la salida, vamos mejorando el error, por lo que se intenta encontrar una recta que *separe* los datos linealmente. Como se aprecia en la gráfica, los datos no son linealmente separables, aunque la recta verde si que parece que obtiene una recta más adecuada. Aunque es de esperar que nos salga un error alto, porque como ya sabíamos, los datos no son linealmente separables. Esto también nos puede decir que nuestro clasificador es malo. Ya que tenemos la mitad de probabilidad de acertar.

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

2. Generaremos la gráfica para los datos de test. Realizamos el mismo procedimiento que antes.

```
# Cuando se utilizan números aleatorios, es recomedable establecer una semilla, para
# que al repetir la generación de los números, siempre obtengamos los mismos
set.seed(3)

# Juntamos los valores obtenidos de simetria e intensidad del test
datos.test <- cbind(intensidad.test, simetria.test)

# Dibujamos los puntos del test
plot(datos.test, xlab = "Intensidad Promedio", ylab = "Simetria",
     col = etiquetas48.test+3, main = "Regresión Lineal con PLA Pocket (test)")

# Dibujamos la línea que separa los puntos (mediante regresión lineal)
abline(-pesos01.train[1]/pesos01.train[3], -pesos01.train[2]/pesos01.train[3],
       col = 5, lwd = 2) # Línea color amarillo es de regresión lineal

# Dibujamos la línea que separa los puntos (mediante PLA Pocket)
abline(-pesos02.train[1]/pesos02.train[3], -pesos02.train[2]/pesos02.train[3],
       col = 3, lwd = 2) # Línea color verde es de PLA Pocket
```

```
# Agregamos una leyenda
legend(-0.75, 0.95, c("Regresión Lineal", "PLA Pocket"), cex = 0.8, col = c(5, 3),
       lty=c(1,1))
```



Como podemos ver en la salida, los datos parecen que están mejor separados utilizando el PLA Pocket. Pero siguen sin ser separables linealmente.

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

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

Vamos a proceder a calcular el error dentro y fuera de la muestra. Para ello, usamos los pesos obtenidos en el *train* del *PLA Pocket*. Ya que la recta del *PLA Pocket*, es nuestra función g.

Para E_{in}

```
# Calculamos la recta de regresión que separa ambas clases (PLA Pocket)
recta_regresion03 <- c(-pesos02.train[1]/pesos02.train[3],
                       - pesos02.train[2]/pesos02.train[3])

# Etiquetamos los puntos a partir de la recta de regresión
etiquetas_regresion01 <- sign(datos.train[,2]
                              - recta_regresion03[1]*datos.train[,1]
                              - recta_regresion03[2])

# Calculamos Ein como las etiquetas que son distintas, partido el total de las etiquetas
Ein <- 0
```

```r
Ein <- sum(etiquetas48.train != etiquetas_regresion01) / length(etiquetas48.train)

Ein
```

```
## [1] 0.4652778
```

Para E_{test}:

Calculamos E_{test} considerando como si fuera E_{out}. Esta conclusión fue sacada del libro *Learning from Data: A short course de Abu Mostafa, Magdon Ismail, Lin* en la *página 59-60.*

_**Nota**: E_{test} es un subconjunto de E_{out}

```r
## ya tienes arriba la recta creada
# Calculamos la recta de regresión que separa ambas clases (PLA Pocket)
recta_regresion03 = c(-pesos02.train[1]/pesos02.train[3],
                      - pesos02.train[2]/pesos02.train[3])

# Etiquetamos los puntos a partir de la recta de regresión
etiquetas_regresion01 <- sign(datos.test[,2] -
                                recta_regresion03[1]*datos.test[,1] -
                                recta_regresion03[2])

# Calculamos Ein como las etiquetas que son distintas, partido el total de las etiquetas
Etest <- 0
Etest = sum(etiquetas48.test != etiquetas_regresion01) / length(etiquetas48.test)

Etest
```

```
## [1] 0.4705882
```

Hemos obtenido que $E_{in} = 0.46$ y que $E_{out} = 0.47$, esto puede tener cierta lógica ya que nuestros datos no son separables linealmente. Además, el error fuera de la muestra es algo mayor al de dentro, esto quiere decir que no hemos sobreajustado la muestra.

3) 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 realizar este apartado tenemos que tener el valor de E_{in} y E_{out}.

Primero, vamos a obtener la cota E_{out} basada en E_{in}, para ello, nos vamos a la página 15 de las transparencias de la *Sesión 5.* Y copiamos la siguiente fórmula:

$$E_{out}(h) <= E_{in}(h) + \sqrt{\frac{8}{N} log \frac{4((2N)^{d_{vc}} + 1)}{\delta}}$$

Por tanto, vamos a usar la dimensión de *Vapnik-Chervonenkis*, y sabemos que para las rectas en el plano (*Perceptron*), dicha dimensión es 3, es decir, $d_{vc} = d + 1$, siendo d el tamaño de la entrada, en nuestro caso 2.

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{N} ln \left(\frac{4((2N)^{d_{VC}} + 1)}{\delta} \right)}$$

Así que simplemente tenemos que sustituir en la ecuación y resolver:

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{N} ln\left(\frac{4((2N)^3+1)}{0.05}\right)}$$

```
# Obtenemos el tamaño de los datos
N <- nrow(datos.train)

# Calculamos el segundo término de la fórmula
x <- sqrt( (8/N) * log((4*((2*N)^(3) + 1))/0.05) )

# Obtenemos el valor de Eout
cota_Ein_Eout <- Ein + x
cat ("Cota de generalizacion basada en E_in: ", cota_Ein_Eout)
```

```
## Cota de generalizacion basada en E_in:  1.141141
```

Segundo, vamos a obtener la cota E_{out} basada en E_{test}, para ello hacemos uso del libro *Learning from Data*, en concreto en la *página 40*, nos viene la ecuación que necesitamos, que está basada en la *desigualdad de Hoeffding* :

$$P[|E_{in}(g) - E_{out}(g)| > \epsilon] \leq 2Me^{-2N\epsilon^2}$$

Resolviendo esta fórmula, llegamos a:

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{1}{2N} ln\left(\frac{2M}{\delta}\right)}$$

Debemos cambiar E_{in} por E_{test}, ya que esta ecuación es para cuando E_{in} tiene un conjunto finito de hipótesis de tamaño M y en este caso, nuestro conjunto de hipótesis es infinito. Por eso, debemos tomar $M = 1$.

$$E_{out}(g) \leq E_{test}(g) + \sqrt{\frac{1}{2N} ln\left(\frac{2M}{\delta}\right)}$$

$$E_{out}(g) \leq E_{test}(g) + \sqrt{\frac{1}{2N} ln\left(\frac{2}{0.05}\right)}$$

```
# Obtenemos el tamaño de los datos
N <- nrow(datos.test)

# Calculamos el segundo término de la fórmula
x <- sqrt( (1/(2*N)) * log( 2 / 0.05) )

# Obtenemos el valor de Eout
cota_Ein_Eout <- Etest + x
cat ("Cota de generalizacion basada en E_test: ", cota_Ein_Eout)
```

```
## Cota de generalizacion basada en E_test:  0.6607605
```

La cota que obtenemos con el conjunto de test es menor que la que obtenemos con el train. Esto nos quiere decir, que la cota obtenida con el conjunto de test es más que la que obtenemos con el conjunto de entrenamiento, ya que las muestras para *test* están menos manipuladas que para *train*. Por eso, la cota

basada en E_{test} es mejor. Además, E_{out} basada en E_{test} es mejor para generalizar el error fuera de la muestra, ya que su valor es más pequeño.

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

Ejercicio 4

En este ejercicio evaluamos el papel de la regularización en la selección de modelos. Para $d = 3$ (dimensión del vector de características) generar un conjunto de N datos aleatorios x_n, y_n de la siguiente forma:

- Las coordenadas de los puntos x_n se generarán como valores aleatorios extraidos de una Gaussiana de media 1 y desviación típica 1.
- Para definir el vector de pesos w_f de la función f generamos $d+1$ valores de una Gaussiana de media 0 y desviación típica 1. Al último valor le sumaremos 1.
- Usando los valores anteriores generamos la etiqueta asociada a cada punto x_n a partir del valor $y_n = w_f^T x_n + \sigma \epsilon n$, donde ϵn es un ruido que sigue también una Gaussiana de media 0 y desviación típica 1 y σ^2 es la varianza del ruido; fijar $\sigma = 0,5$

Para generar la gráfica de nube de puntos, partimos de la función *simula_gaus*() desarrollada por la profesora. En ella, se genera un conjunto de longitud N de vectores de dimensión *dim*, conteniendo números aleatorios gaussianos de media 0 y varianzas, dadas por el vector *sigma*. Por defecto, genera 2 puntos de 2 dimensiones. Además, debemos incluir a la función un nuevo parámetro *mean*.

```
simula_gaus = function(N = 2, mean = 0, dim = 2, sigma) {
  if (missing(sigma))
    stop("Debe dar un vector de varianzas")

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

  #if(dim != length(sigma))
   # stop ("El numero de varianzas es distinto de la dimensión")

  # Genera 1 muestra, con las desviaciones especificadas
  simula_gauss1 = function() rnorm(dim, mean = mean, sd = sigma)

  # Repite N veces, simula_gauss1 y se hace la traspuesta
  m = t(replicate(N,simula_gauss1()))
  m
}
```

Ahora vamos a estimar el valor de w_f usando w_{reg}, es decir, los pesos de un modelo de regresión lineal con regularización "weight decay". Fijar el parámetro de regularización a $0,05/N$.

Entonces, vamos a implementar el algoritmo de Regresión lineal usando "weight decay". Para ello partimos del código implmentado de **Regresión Lineal** de la práctica 1. Para completar este método nos basaremosen el algoritmo descrito en el libro *Learning from Data: A short course* y de las transparencias de la *Sesión 6 página 16* y anotaciones de clase. Usaremos la transformación SVD.

Para calcular w_{reg} uso la siguiente fórmula:

$$w_{reg} = (Z^T Z + \lambda I)^{-1} \; Z^T y$$

Para calcular el error, uso la la fórmula de la *página 134* del libro *Learning from Data*:

$$E_{in}(w_{reg}) = \frac{1}{N} y^T (I - H(\lambda))^2 y$$

Ya solo nos queda implementar la función, y mostrar un pequeño ejemplo:

```
# Regresión Lineal con "weight decay"
Regress_Lin_weight_decay = function(datos,label,lambda) {

  # Calculamos la inversa
  inversa <- solve(t(datos) %*% datos + lambda * diag(nrow = ncol(datos), ncol = ncol(datos)))

  # Calculamos pseudoinversa
  pseudoinversa <- inversa %*% t(datos)

  # Calculamos los pesos
  w <- pseudoinversa %*% label

  # Devolvemos los pesos
  w
}
```

Ahora, vamos a realizar un pequeño experimento para comprobar que funciona, siguiendo las indicaciones del enunciado.

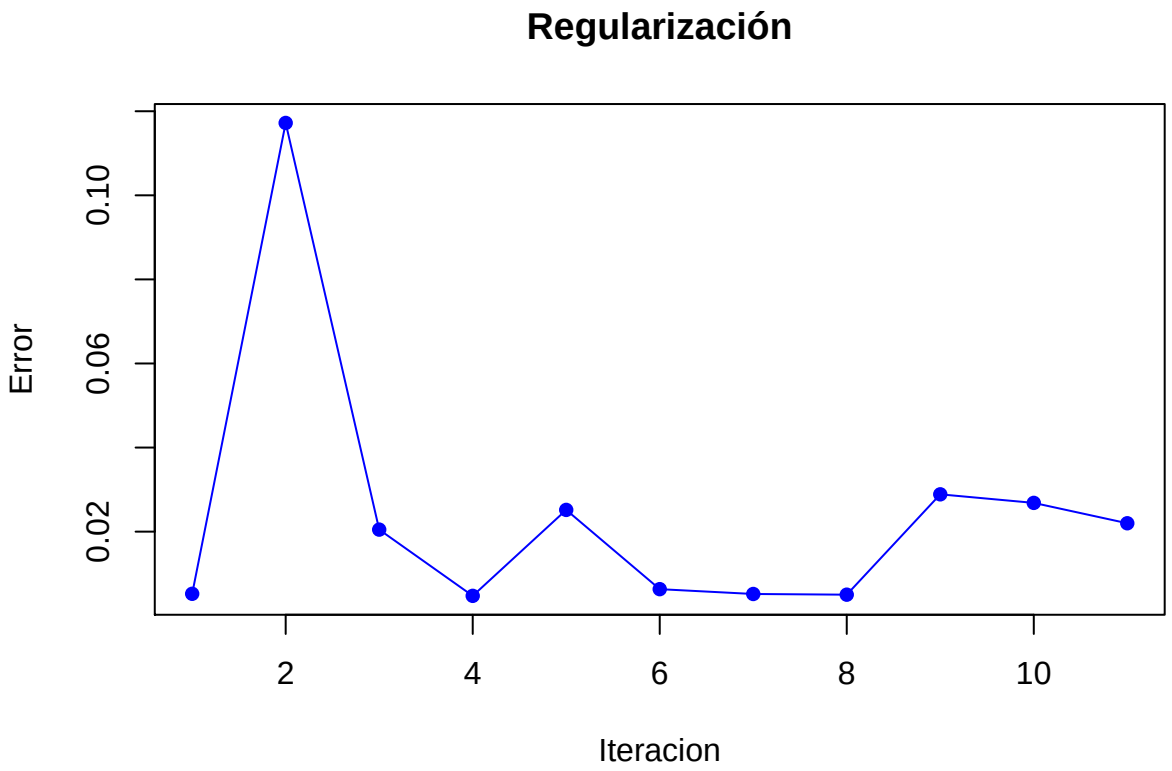
```
# Cuando se utilizan números aleatorios, es recomedable establecer una semilla, para
# que al repetir la generación de los números, siempre obtengamos los mismos
set.seed(3)

# Para realizar el ejemplo, introducimos todo en una función
# Como entrada le pasaremos a la función el número de datos,
# la dimensión, un sigma y un lambda
# (Todos definidos en el enunciado del ejercicio)
Validación_Cruzada <- function(N = 11, d = 3, sigma = 0.5, lambda = 0.05) {

  # Vector que guardar todos los errores
  ecv = vector()
  # Vector que guardar errores del primer conjunto [13]
  e1 = vector()
  # Vector que guardar errores del segundo conjunto [23]
  e2 = vector()

  # Definimos el vector de pesos genernado d+1 valores de una Gaussiana
  #de media 0 y desviación típica 1
  wf = as.vector(simula_gaus(N=1, dim=d+1, mean = 0, sigma=c(1)))

  # La lista (d+10, d+20, ..., d+115)
  tams = seq(from=10, to=110, by=10)

  # Recorremos los valores e iniciamos la validación cruzada
```

```
  for(i in tams) {

    # Las coordenadas 'x', se generará como valores extraídos de una
    # gaussiana de media 1 y de desviación típica 1
    datos = simula_gaus(N, mean = 1, dim = d, sigma = c(1))

    # Añadimos una nueva columna, para poder hacer el producto escalar
    datos = cbind(datos,1)

    # Generamos las etiquetas a partir de wf*xn + sigma*ruido
    # Donde ruido es una gaussiana de media 0 y desviación típica
    # 1, fijar sigma a 0.5
    etiquetas = apply (datos, 1,

                       FUN = function(x){

                         # Calculamos la fórmula
                         wf %*% x + sigma * rnorm(1, 0, 0.5)
                         }
                       )

    # Calculamos el error
    error = sapply(1:N,

                   FUN = function(x){

                     # Calculamos wreg como la fórmula explicada al
                     # principio del ejercicio
                     # Cuando pongamos [-x,] haremos referencia a los
                     # datos del train y cuando pogamos [x,], haremos
                     # referencia a los datos del test

                     wreg = Regress_Lin_weight_decay(datos[-x,], etiquetas[-x],
                                                     lambda/N)

                     (1/N) * ( ( etiquetas[x] - (datos[x,] %*% wreg) )^2 )
                     }
                   )

    # Guardamos la media de todos lo errores
    ecv = c(mean(error), ecv)
    # Guardamos la media de los primeros errores (e1)
    e1 = c(error[1],e1)
    # Guardamos la media de los segundos errores (e2)
    e2 = c(error[2],e2)

  }

  list(errores = error, Ecv = mean(ecv), E1 = mean(e1), E2 = mean(e2))
}

# Probamos el algoritmo
Validación_Cruzada()
```

```
## $errores
##  [1] 0.005214523 0.117211565 0.020478197 0.004726887 0.025168134
##  [6] 0.006316337 0.005168514 0.005005409 0.028875179 0.026834047
## [11] 0.021990140
##
## $Ecv
## [1] 0.01027156
##
## $E1
## [1] 0.006615844
##
## $E2
## [1] 0.01555503
```

Como se ve, obtenemos 11 errores, esto quiere decir que el primer error coincide a [13], el segundo a [23], y así hasta [113]. Ahora vamos a crear una gráfica donde se vea la evolución del error:

```r
# Cuando se utilizan números aleatorios, es recomedable establecer una semilla, para
# que al repetir la generación de los números, siempre obtengamos los mismos
set.seed(3)

# Guardamos el resultado
resultado <- Validación_Cruzada()

# Lo mostramos gráficamente
plot(resultado$errores, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "Error", main = "Regularización")
```



El error de validación cruzada ha sido 0.01027156. Como se aprecia, la tendencia de la gráfica es a disminuir el error. Pero aún así, debemos repetir el experimento más veces, para poder dar una conclusión más firme.

```r
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

Ahora, vamos a repetir el mismo proceso que antes, pero 1000 veces, y con ello anotaremos el promedio y la varianza de e_1, e_2 y E_{cv}. Pero antes, deberemos cambiar la salida de nuestra función, ya que para realizar la varianza, necesitamos un vector y no una lista. Además, de que ahora se nos pide la media de tres valores, y antes nos pedían que mostráramos todos los errores. Copiaré el mismo algoritmo que antes, solo cambiando la salida.

```r
# Cuando se utilizan números aleatorios, es recomedable establecer una semilla, para
# que al repetir la generación de los números, siempre obtengamos los mismos
set.seed(3)

# Para realizar el ejemplo, introducimos todo en una función
# Como entrada le pasaremos a la función el número de datos,
# la dimensión, un sigma y un lambda
# (Todos definidos en el enunciado del ejercicio)
Validación_Cruzada1 <- function(N = 11, d = 3, sigma = 0.5, lambda = 0.05) {

  # Vector que guardar todos los errores
  ecv = vector()
  # Vector que guardar errores del primer conjunto [13]
  e1 = vector()
  # Vector que guardar errores del segundo conjunto [23]
  e2 = vector()

  # Definimos el vector de pesos genernado d+1 valores de una Gaussiana
  #de media 0 y desviación típica 1
  wf = as.vector(simula_gaus(N=1, dim=d+1, mean = 0, sigma=c(1)))

  # La lista (d+10, d+20, ..., d+115)
  tams = seq(from=10, to=110, by=10)

  # Recorremos los valores e iniciamos la validación cruzada
  for(i in tams) {

    # Las coordenadas 'x', se generará como valores extraídos de una
    # gaussiana de media 1 y de desviación típica 1
    datos = simula_gaus(N, mean = 1, dim = d, sigma = c(1))

    # Añadimos una nueva columna, para poder hacer el producto escalar
    datos = cbind(datos,1)

    # Generamos las etiquetas a partir de wf*xn + sigma*ruido
    # Donde ruido es una gaussiana de media 0 y desviación típica
    # 1, fijar sigma a 0.5
    etiquetas = apply (datos, 1,

                       FUN = function(x){

                         # Calculamos la fórmula
                         wf %*% x + sigma * rnorm(1, 0, 0.5)
```

```
                      }
                    )

    # Calculamos el error
    error = sapply(1:N,

                   FUN = function(x){

                     # Calculamos wreg como la fórmula explicada al
                     # principio del ejercicio
                     # Cuando pongamos [-x,] haremos referencia a los
                     # datos del train y cuando pogamos [x,], haremos
                     # referencia a los datos del test

                     wreg = Regress_Lin_weight_decay(datos[-x,], etiquetas[-x],
                                                     lambda/N)

                     (1/N) * ( ( etiquetas[x] - (datos[x,] %*% wreg) )^2 )
                     }
                   )

    # Guardamos la media de todos lo errores
    ecv = c(mean(error), ecv)
    # Guardamos la media de los primeros errores (e1)
    e1 = c(error[1],e1)
    # Guardamos la media de los segundos errores (e2)
    e2 = c(error[2],e2)
  }

   c(mean(ecv), mean(e1), mean(e2))
}
```

Una vez, modificado el algoritmo, comenzaré a realizar el experimento. Debemos repetir la misma función de antes 1000 veces.

```
# Cuando se utilizan números aleatorios, es recomedable establecer una semilla, para
# que al repetir la generación de los números, siempre obtengamos los mismos
set.seed(3)

# Lanzamos 1000 el algoritmo
res <- replicate(1000, Validación_Cruzada1())

# Guardamos los valores de Ecv (para hacer media y varianza)
Ecv = res[1,]
cat ("La media de Ecv es", mean(Ecv))
```

```
## La media de Ecv es 0.009981508
```

```
cat ("\nLa varianza de Ecv es", var(Ecv))
```

```
##
## La varianza de Ecv es 3.526194e-06
```

```
# Guardamos los valores de E1 (para hacer media y varianza)
E1 = res[2,]
```

```r
cat ("\n\nLa media de E1 es", mean(E1))
```

```
##
##
## La media de E1 es 0.009715803
```

```r
cat ("\nLa varianza de E1 es", var(E1))
```

```
##
## La varianza de E1 es 2.25495e-05
```

```r
# Guardamos los valores de E2 (para hacer media y varianza)
E2 = res[3,]
cat ("\n\nLa media de E2 es", mean(E2))
```

```
##
##
## La media de E2 es 0.009981066
```

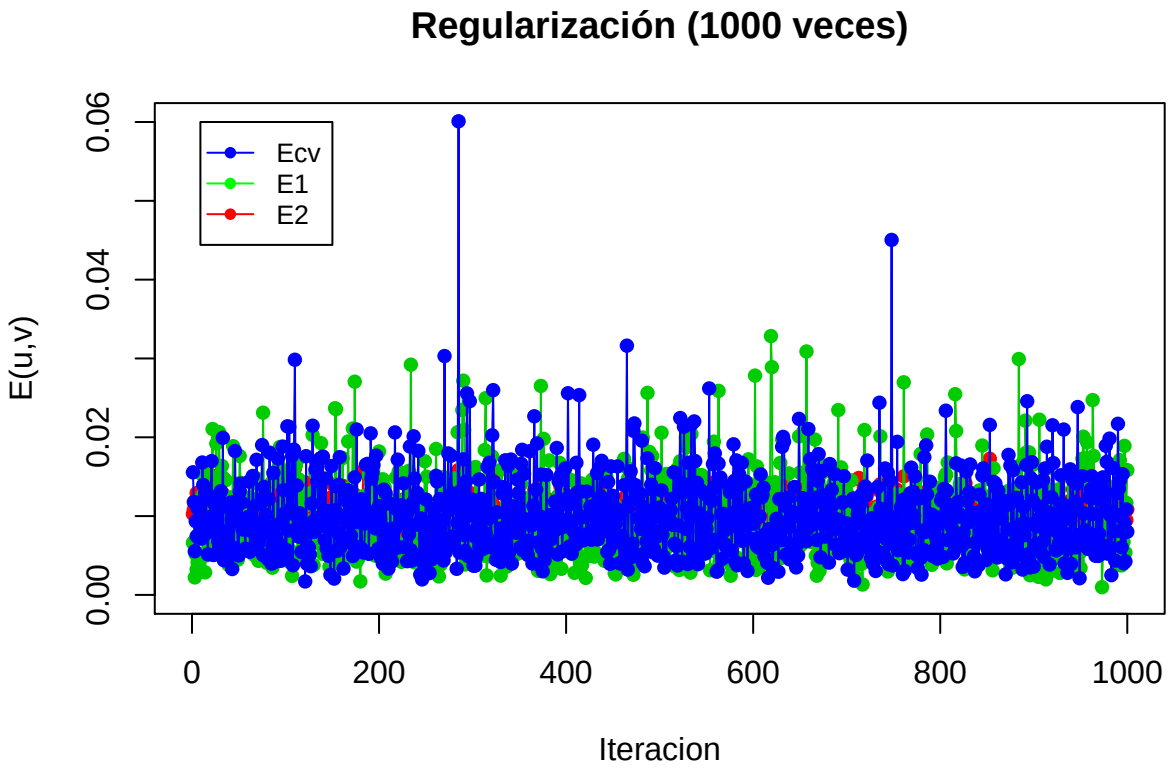
```r
cat ("\nLa varianza de E2 es", var(E2))
```

```
##
## La varianza de E2 es 2.583383e-05
```

Para poder interpretar los datos, hay que tener en cuenta que la *varianza* mide qué tan dispersos están los datos alrededor de su media. Por lo que podemos ver con los resultados obtenidos, que nuestros valores no están casi dispersos. Además, las 3 *medias* que se obtienen, tienen sentido, por lo que es razonable pensar que tienen que tener valores aproximados.

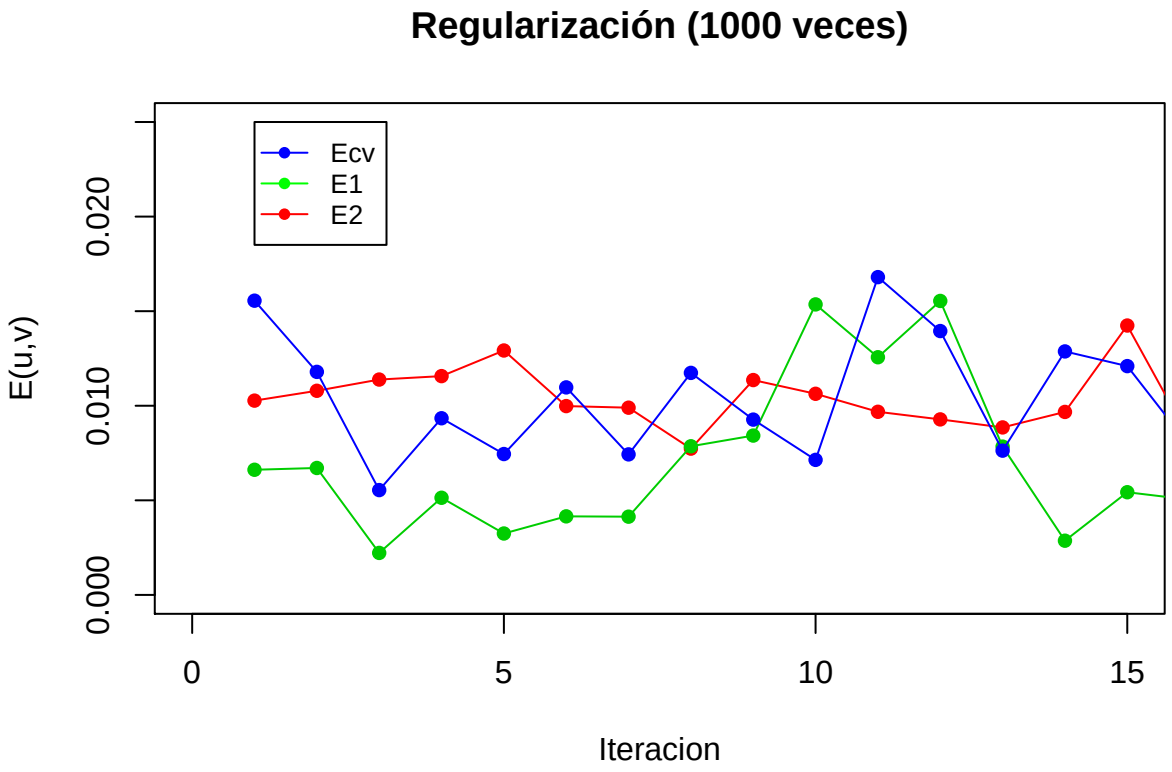
Pero vamos a mostrar gráficamente los valores que hemos obtenido para las variables. Primero mostraré la gráfica con las 1000 iteraciones.

```r
# Abrimos plot
plot(c(0.2,1000), c(0,0.06), type = "n", xlab = "Iteracion", ylab = "E(u,v)",
     main = "Regularización (1000 veces)")

# Mostramos gráficamente la recta para Ecv
lines(Ecv, type = "o", pch = 16, col = 2)
# Mostramos gráficamente la recta para E1
lines(E1, type = "o", pch = 16, col = 3)
# Mostramos gráficamente la recta para E2
lines(E2, type = "o", pch = 16, col = 4)

# Agregamos una leyenda
legend(9, 0.06, c("Ecv", "E1", "E2"),
       cex = 0.8, col = c("blue", "green", "red"), pch=16, lty=c(1,1))
```



Como se puede ver, no es que podamos interpretar los datos muy bien, por tanto realizaré un zoom sobre una pequeña porción de la gráfica.

```
# Abrimos plot
plot(c(0,15), c(0,0.025), type = "n", xlab = "Iteracion", ylab = "E(u,v)",
     main = "Regularización (1000 veces)")

# Mostramos gráficamente la recta para Ecv
lines(Ecv, type = "o", pch = 16, col = 2)
# Mostramos gráficamente la recta para E1
lines(E1, type = "o", pch = 16, col = 3)
# Mostramos gráficamente la recta para E2
lines(E2, type = "o", pch = 16, col = 4)

# Agregamos una leyenda
legend(1, 0.025, c("Ecv", "E1", "E2"),
       cex = 0.8, col = c("blue", "green", "red"), pch=16, lty=c(1,1))
```



Con esta gráfica, ya sí podremos contestar mejor las respuestas que vienen a continuación. En vez de explicarlo ahora, iré contestando los apartados siguientes, ya que se basan en la interpretación de todos estos datos que hemos obtenido.

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

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

La media de E_{cv} es \$ 0.0099815085\$, la media de E_1 es 0.009715803 y la media de E_2 es 0.009981066. Por lo que la lógica nos dice que los valores promedio deben de ser similares.

La relación que deben de tener el valor promedio de e_1 y el de E_{cv}, es que E_1 debe estar contenido en E_{cv}. Para ver la relación entre el valor promedio de e_1 y el de e_2, nos basamos en que ambos se obtienen de la misma distribución. Por lo que para un número de experimentos grande, la media de estos valores se parecerán. E_cv no es más que la media de los e_i.

c) ¿Qué es lo que más contribuye a la varianza de los valores de e_1?

Para la varianza de e_1, contribuye que los valores de cada ejecución son independientes, es decir, hay que tener en cuenta que la generación aleatoria del punto, da lugar a un w_{reg} distinto, por eso la indenpendencia de los valores de e_1.

d) Diga que conclusiones sobre regularización y selección de modelos ha sido capaz de extraer de esta experimentación.

Como conclusión, podemos definir que este proceso es un buen estimador. Además, la varianza juega mucho a su favor, ya que obtenemos que los datos no están muy dispersos. También, acabamos de ver que la relación que guardan e_1 y e_2 con respecto E_{cv} es la misma.

Una dato importante que hemos sacado de este experimento, es que al principio, cuando ejecutamos solo una vez el algoritmo, obteníamos medias muy dispares o distintas entre sí, sin embargo, cuando hemos realizado 1000 veces el algoritmo, las medias entre los valores se mantienen al unísono.

Bonus

Ejercicio 1

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, minimizamos a lo largo de cada una de las coordenadas individualmente. 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 $\eta = 0.1$.

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

Para hacer el algoritmo de **Coordenada Descendente**, partimos del algoritmo gradiente descendente. La única diferencia con él, es que ahora se deriva primero la variable u y luego se deriva la variable v a partir del nuevo valor de u, es decir, avanzamos sólo en una dirección.

***Nota**: Tuve que quitar del algoritmo la condición de parada donde la diferencia entre dos puntos en valor absoluto sea muy cercana y la variable donde guardaba el valor anterior ya que ocasionaba error en las iteraciones del bucle.*

Veamos cómo se implementaría:

```
CD <- function(funcion, funcion_gradiente, eta=0.1, vini=c(0,0), umbral=10^(-4),
               iteraciones_max=500){

  # Inicializar los pesos
  w_new <- vini

  # Variable que lleva el número de iteraciones realizadas
  iter <- 0

  # Array creado donde guardaremos el valor de la funcion, para luego crear la gráfica
  f <- rep(iteraciones_max)
  f[iter] <- funcion(w_new[1],w_new[2])

  # Mostramos el punto de partida
  cat("Punto de partida: ", w_new[1], w_new[2])

  # Mostramos los valores del gradiente al principio
  cat("\nGradiente (inicial): ", funcion_gradiente(w_new[1],w_new[2]))

  # Mostramos el valor de la función al principio
```

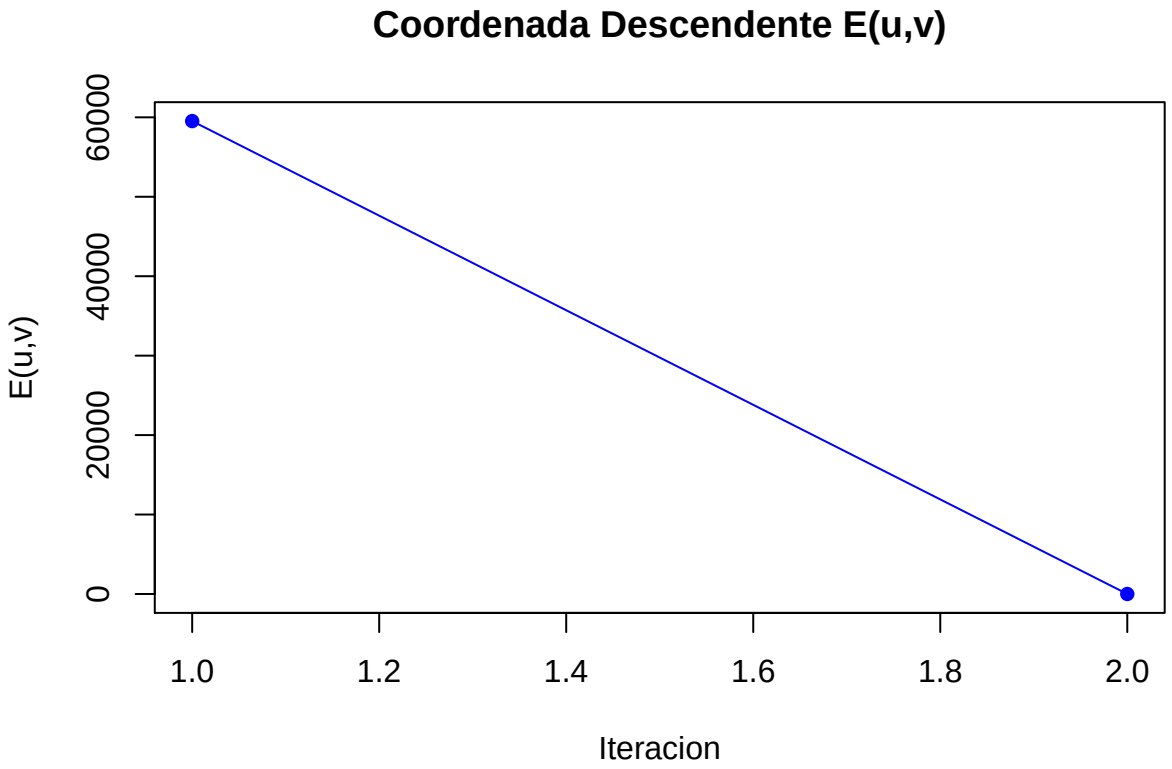
```
  cat("\nValor función (inicial): ", funcion(w_new[1], w_new[2]))

  # La condición de parada será cuando el valor de la funcion sea menor o igual a
  # un umbral, cuando se hayan completado un número máximo de iteraciones
  while ((iter <= iteraciones_max) & (funcion(w_new[1], w_new[2]) > 10^(-4))) {

    # Paso 1: nos movemos a lo largo de la coordenada u
    gradiente <- funcion_gradiente(w_new[1], w_new[2])
    direccion <- -gradiente
    w_new[1] <- w_new[1] + eta * direccion[1]

    # Paso 2: reevaluamos y nos movemos a lo largo de la coordenada v
    gradiente <- funcion_gradiente(w_new[1], w_new[2])
    direccion <- -gradiente
    w_new[2] <- w_new[2] + eta * direccion[2]

    # Incrementamos el valor de la iteración
    iter <- iter + 1

    # Guardamos el nuevo valor de f
    f[iter] = funcion(w_new[1],w_new[2])
  }

  # Mostramos los valores del gradiente al terminar
  cat("\nGradiente (final): ", funcion_gradiente(w_new[1],w_new[2]))

  # Mostramos la solución final
  cat("\nSolución: ", w_new[1], w_new[2])

  # Mostramos el valor de la función al final
  cat("\nValor función (final): ", funcion(w_new[1], w_new[2]))

  # Devolvemos el número de iteraciones necesarias para encontrar la solución
  cat("\nIteraciones: ",iter-1)
  cat("\n")

  # Mostramos el valor de la funcion
  list(valor_funcion = f[1:iter])
}
```

Procedemos a ejecutar el algoritmo:

```
# Guardamos el valor con las condiciones específicadas en el enunciado
resultado11 = CD(E, E_dev, eta=0.1, vini=c(1,1), umbral=10^(-4), iteraciones_ma=15)
```

```
## Punto de partida:  1 1
## Gradiente (inicial):  24.47354 4.943477
## Valor función (inicial):  3.930397
## Gradiente (final):  0 0
## Solución:  11903.21 -89355363164
## Valor función (final):  0
## Iteraciones:  1
```

```r
# Lo mostramos gráficamente
plot(resultado11$valor_funcion, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "E(u,v)", main = "Coordenada Descendente E(u,v)")
```



Se obtiene un valor de la funcion $E(u, v)$ cero. En el siguiente apartado, explico lo ocurrido.

```r
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

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

Para poder establecer una comparación, debo tener implementado la misma idea para ambos algoritmos, y como he comentado antes he modificado el algoritmo de coordenada descendente debido a varios errores que me daba. Por tanto, voy a implementar coordenada descendente pero usando gradiente descendiente.

***Nota**: Omito la condición de parada donde la diferencia entre dos puntos en valor absoluto sea muy cercana y la variable donde guardaba el valor anterior, que aparecían en el primer diseño del gradiente descendente.*

```r
# Gradiente Descendente Modificado
GD1 <- function(funcion, funcion_gradiente, eta=0.1, vini=c(0,0), umbral=10^(-4),
                iteraciones_max=500){

  # Inicializar los pesos
  w_new <- vini

  # Variable que lleva el número de iteraciones realizadas
  iter <- 0

  # Array creado donde guardaremos el valor de la funcion, para luego crear la gráfica
```

```
  f <- rep(iteraciones_max)
  f[iter] <- funcion(w_new[1],w_new[2])

  # Mostramos el punto de partida
  cat("Punto de partida: ", w_new[1], w_new[2])

  # Mostramos los valores del gradiente al principio
  cat("\nGradiente (inicial): ", funcion_gradiente(w_new[1],w_new[2]))

  # Mostramos el valor de la función al principio
  cat("\nValor función (inicial): ", funcion(w_new[1], w_new[2]))

  # La condición de parada será cuando el valor de la funcion sea menor o igual a
  # un umbral, cuando se hayan completado un número máximo de iteraciones
  while ((iter <= iteraciones_max) & (funcion(w_new[1], w_new[2]) > 10^(-4))) {

    # Calculamos el gradiente
    gradiente <- funcion_gradiente(w_new[1], w_new[2])

    # Establecemos la dirección a moverse
    direccion <- -gradiente

    # Actualizamos los pesos
    w_new <- w_new + eta * direccion

    # Incrementamos el valor de la iteración
    iter <- iter + 1

    # Guardamos el nuevo valor de f
    f[iter] = funcion(w_new[1],w_new[2])
  }

  # Mostramos los valores del gradiente al terminar
  cat("\nGradiente (final): ", funcion_gradiente(w_new[1],w_new[2]))

  # Mostramos la solución final
  cat("\nSolución: ", w_new[1], w_new[2])

  # Mostramos el valor de la función al final
  cat("\nValor función (final): ", funcion(w_new[1], w_new[2]))

  # Devolvemos el número de iteraciones necesarias para encontrar la solución
  cat("\nIteraciones: ",iter-1)
  cat("\n")

  # Mostramos el valor de la funcion
  list(valor_funcion = f[1:iter])
}
```

Ejecuto las mismas condiciones para gradiente descendente y comparo:

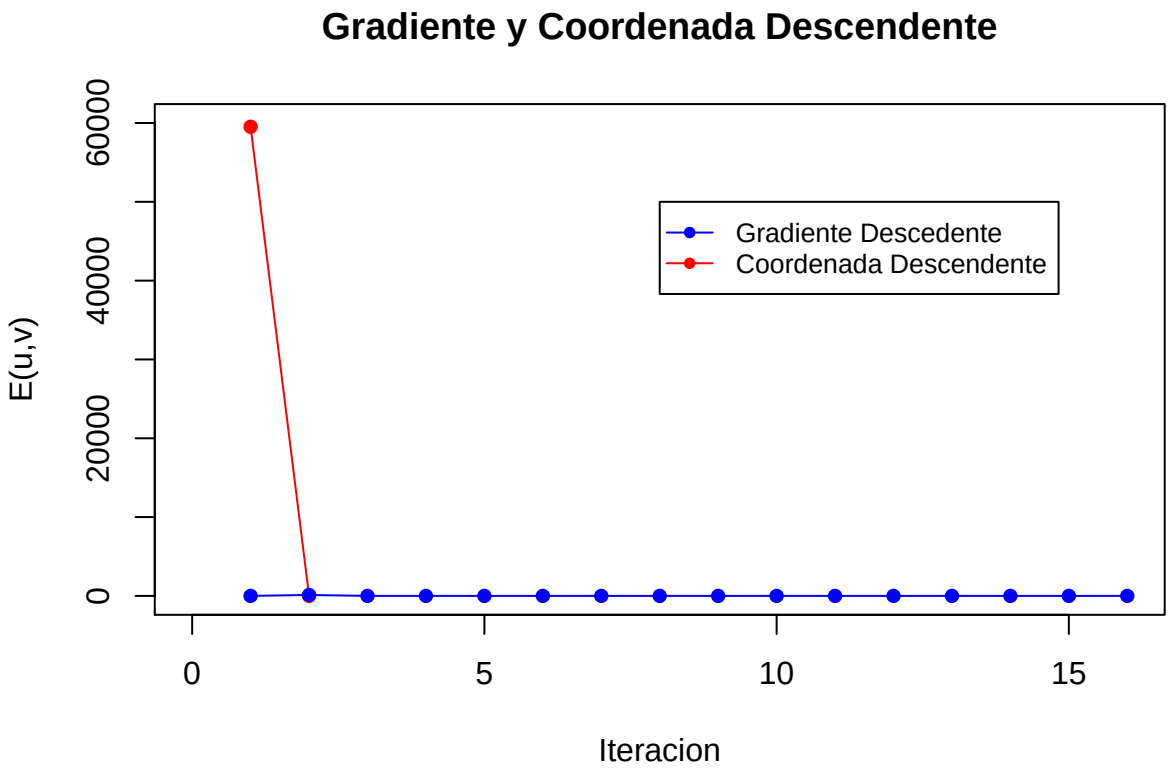
```
resultado12 = GD1(E, E_dev, eta=0.1, vini=c(1,1), umbral=10^(-4), iteraciones_ma=15)
```

```
## Punto de partida:  1 1
```

```
## Gradiente (inicial):  24.47354 4.943477
## Valor función (inicial):  3.930397
## Gradiente (final):  -0.007569897 -0.0006195309
## Solución:  9.873748 -24.43753
## Valor función (final):  0.003784948
## Iteraciones:  15
```

Muestro las dos gráficas para comprobar como llegan al mínimo ambos algoritmos:

```
# Abrimos plot
plot(c(0,16), c(0,60000), type = "n", xlab = "Iteracion", ylab = "E(u,v)",
     main = "Gradiente y Coordenada Descendente")

# Mostramos gráficamente la recta para CD
lines(resultado11$valor_funcion, type = "o", pch = 16, col = 2)

# Mostramos gráficamente la recta para GD
lines(resultado12$valor_funcion, type = "o", pch = 16, col = 4)

# Agregamos una leyenda
legend(8, 50000, c("Gradiente Descedente", "Coordenada Descendente"),
       cex = 0.8, col = c("blue", "red"), pch=16, lty=c(1,1))
```



Como acabamos de ver, se obtiene una gráfica poco realista (debido a la dimensión de los números), pero ambos llegan a un mínimo (o se intenta). Por tanto, la calidad de la solución obtenida es aceptable. Vemos que la trayectoria seguida por Coordenada Descendente tiene sentido, y además consigue un mínimo en una sola iteración. Aunque nos lleva a otro mínimo, distinto al que habíamos encontrado con gradiente descendente.

Por tanto, Coordenada Descendente puede que cuando llegue a un mínimo se quede estancando ahí y no consiga avanzar, mientras que con Gradiente Descendente si lograríamos seguir

Hay que tener en cuenta, que Coordenada Descendente se centra en minimizar primero una variable y luego la otra, mientras que Gradiente Descendente se centra en minimizar las dos variables a la vez, por lo que en casos particulares como he comentado anteriormente podría llegar a no ser mejor que Gradiente Descendente.

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

Pongamos otra comparación entre ambos algoritmos, minimicemos la tasa de aprendizaje a 0.001 para ambos algoritmos.

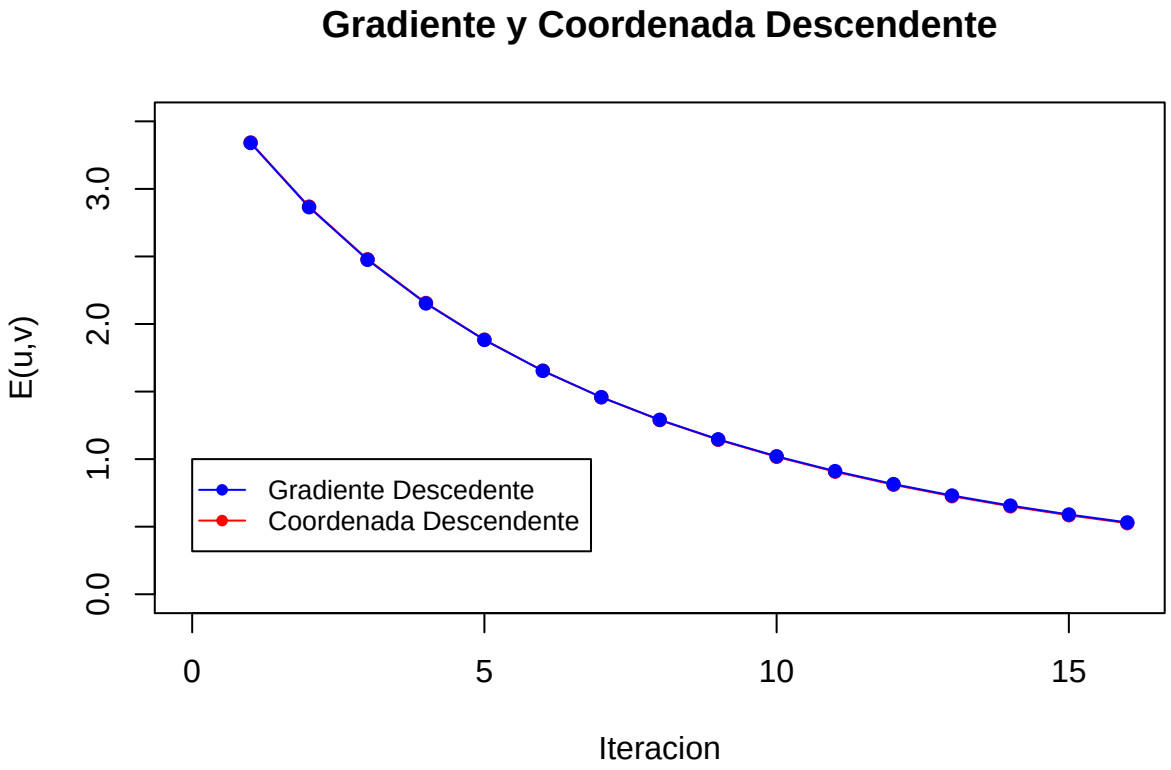
```
# Coordenada Descedente con tasa de 0.001
resultado13 = CD(E, E_dev, eta=0.001, vini=c(1,1), umbral=10^(-4), iteraciones_ma=15)
```

```
## Punto de partida:  1 1
## Gradiente (inicial):  24.47354 4.943477
## Valor función (inicial):  3.930397
## Gradiente (final):  7.321647 -0.28147
## Solución:  0.7770985 0.9840421
## Valor función (final):  0.5258717
## Iteraciones:  15
```

```
# Gradiente Descedente con tasa de 0.001
resultado14 = GD1(E, E_dev, eta=0.001, vini=c(1,1), umbral=10^(-4), iteraciones_ma=15)
```

```
## Punto de partida:  1 1
## Gradiente (inicial):  24.47354 4.943477
## Valor función (inicial):  3.930397
## Gradiente (final):  7.3205 -0.2761781
## Solución:  0.7776887 0.978857
## Valor función (final):  0.5316383
## Iteraciones:  15
```

```
# Abrimos plot
plot(c(0,16), c(0,3.5), type = "n", xlab = "Iteracion", ylab = "E(u,v)",
     main = "Gradiente y Coordenada Descendente")

# Mostramos gráficamente la recta para la tasa  0.1
lines(resultado13$valor_funcion, type = "o", pch = 16, col = 2)

# Mostramos gráficamente la recta para la tasa  0.01
lines(resultado14$valor_funcion, type = "o", pch = 16, col = 4)

# Agregamos una leyenda
legend(0, 1, c("Gradiente Descedente", "Coordenada Descendente"),
       cex = 0.8, col = c("blue", "red"), pch=16, lty=c(1,1))
```



Como acabamos de ver, para una tasa de aprendizaje muy baja, ambos algoritmos tienen el mismo comportamiento. Siendo ambos, capaces de encontrar un óptimo.

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

Ejercicio 2

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.

Generar un gráfico de como desciende el valor de la función con las iteraciones.

Antes de nada, debemos saber que el **método de Newton** introduce el uso de la matriz Hessiana, que no es más que la matriz definida por cuatro elementos, en nuestro caso:

***Nota**: Usamos la fórmula del ejercicio 1 apartado b).*

$$\begin{pmatrix} \frac{\partial^2 f}{\partial x x} & \frac{\partial^2 f}{\partial x y} \\ \frac{\partial^2 f}{\partial y x} & \frac{\partial^2 f}{\partial y y} \end{pmatrix}$$

Por tanto, debemos de calcular las segundas derivadas de la función con respecto todas las posibles combinaciones de variables. Para realizar las derivadas usaré un paquete de cálculo matemático como wxMaxima o de manera online mediante la página Symbolab.

$$f(x,y) = (x-2)^2 + 2(y-2)^2 + 2sin(2\pi x)sin(2\pi y)$$

$$\frac{\partial^2 f}{\partial xx}(x-2)^2 + 2(y-2)^2 + 2sin(2\pi x)sin(2\pi y) = 2 - 8\pi^2 sin(2\pi x)sin(2\pi y)$$

$$\frac{\partial^2 f}{\partial xy}(x-2)^2 + 2(y-2)^2 + 2sin(2\pi x)sin(2\pi y) = 8\pi^2 cos(2\pi x)cos(2\pi y)$$

$$\frac{\partial^2 f}{\partial yx}(x-2)^2 + 2(y-2)^2 + 2sin(2\pi x)sin(2\pi y) = 8\pi^2 cos(2\pi x)cos(2\pi y)$$

$$\frac{\partial^2 f}{\partial yy}(x-2)^2 + 2(y-2)^2 + 2sin(2\pi x)sin(2\pi y) = 4 - 8\pi^2 sin(2\pi x)sin(2\pi y)$$

```
# Derivada Hessiana
f_hess = function(x,y){

  dxx = 2-8*pi^2*sin(2*pi*x)*sin(2*pi*y)
  dyy = 4-8*pi^2*sin(2*pi*x)*sin(2*pi*y)
  dyx = 8*pi^2*cos(2*pi*x)*cos(2*pi*y)
  dxy = 8*pi^2*cos(2*pi*x)*cos(2*pi*y)

  matrix(c(dxx,dxy,dxy,dyy),2)
}
```

Para entender este algoritmo nos vamos a la *página 35* de la *Sesión 5* de Teoría. Por lo que el método consiste en aplicar de forma iterativa la expresión:

$$w(t+1) = -H^{-1}\ w(t)\ \eta$$

Este algoritmo lo podemos tratar como la misma idea con la que hicimos el **Gradiente Descendente**, pero ahora usando la **matriz Hessiana**. Nuestra condición de parada serán las mismas que para el gradiente descendente.

```
# Método de Newton
Metodo_Newton = function (funcion, funcion_gradiente, funcion_hessiana,
                          vini = c(0,0), umbral = 10^(-5), iteraciones_max = 50) {

  # Inicializar los pesos
  w_old <- c(0,0)
  w_new <- vini


  # Variable que lleva el número de iteraciones realizadas
  iter <- 0

  # Array creado donde guardaremos el valor de la funcion, para luego crear la gráfica
  f <- rep(iteraciones_max)
  f[iter] <- funcion(w_new[1],w_new[2])

  # Mostramos el punto de partida
  cat("Punto de partida: ", w_new[1], w_new[2])

  # Mostramos el valor de la función al principio
```

```r
  cat("\nValor función (inicial): ", funcion(w_new[1], w_new[2]))

  # La condición de parada será cuando el valor de la funcion sea menor o igual a
  # un umbral, cuando se hayan completado un número máximo de iteraciones o cuando
  # la diferencia entre dos puntos en valor absoluto sea muy cercana.
  # La condición de la diferencia entre dos puntos en valor absoluto sea muy cercana,
  # descarta muchos valores, ya que avanza muy lentamente

  while(funcion(w_new[1],w_new[2]) > umbral & (iter < iteraciones_max )
        & (abs(funcion(w_new[1], w_new[2])
               - funcion(w_old[1], w_old[2])) > umbral) ){

    # Asignamos el valor nuevo, al antiguo
    w_old <- w_new

    # Calculamos el gradiente
    gradiente = funcion_gradiente(w_new[1],w_new[2])

    # Calculamos la hessiana
    hessiana = funcion_hessiana(w_new[1],w_new[2])

    # Calculamos la dirección
    direccion = - solve(hessiana) %*% gradiente

    # Actualizamos los pesos
    w_new = w_old + direccion

    # Incrementamos el valor de la iteración (época)
    iter = iter + 1

    # Guardamos el nuevo valor de f
    f[iter] = funcion(w_new[1],w_new[2])
  }

  # Mostramos el valor de la función al final
  cat("\nValor función (final): ", funcion(w_new[1], w_new[2]))

  # Mostramos la solución final
  cat("\nSolución: ", w_new[1], w_new[2])
  cat("\n")

  # Mostramos el valor de la funcion
  list(valor_funcion = f[1:iter])
}

# Ejecutamos el algoritmo
Metodo_Newton(f, f_dev, f_hess, vini=c(1,1))
```

```
## Punto de partida:  1 1
## Valor función (inicial):  3
## Valor función (final):  2.900405
## Solución:  1.050871 1.025432
```

```
## $valor_funcion
```

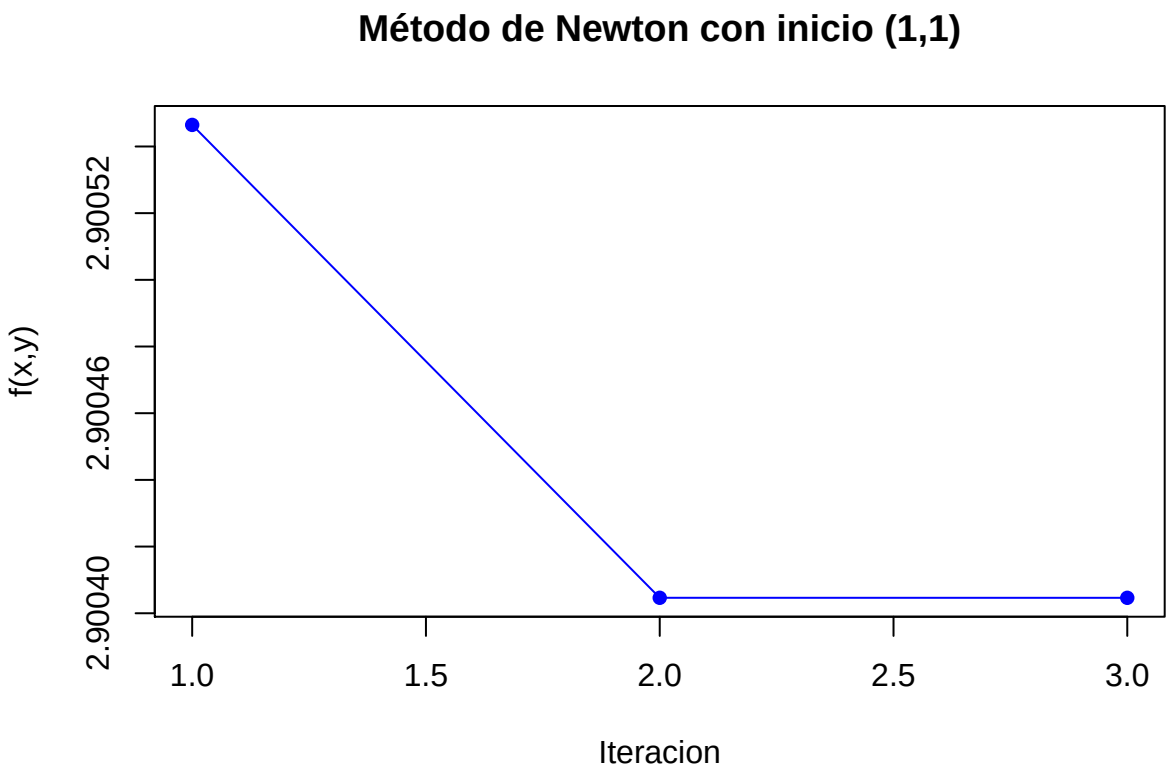
```
## [1] 2.900546 2.900405 2.900405
```

Vamos a visualizar los resultados:

```
# Guardamos el resultado
data = Metodo_Newton(f, f_dev, f_hess, vini=c(1,1))
```

```
## Punto de partida:  1 1
## Valor función (inicial):  3
## Valor función (final):  2.900405
## Solución:  1.050871 1.025432
```

```
# Los mostramos por pantalla
plot(data$valor_funcion, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "f(x,y)", main = "Método de Newton con inicio (1,1)",)
```



Como se ve se obtiene un buen comportamiento a la hora de converger y de encontrar el mínimo. Ya que desciende buscando el óptimo.

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

Vamos a realizar el mismo experimento, pero ahora cambiaremos el punto de inicio.

```
# Dividimos la región de dibujo en 4 partes
par(mfrow=c(2,2))

## Punto (2.1, 2.1) ----------------------------------------------------------------
resultado04 = Metodo_Newton(f, f_dev, f_hess, vini = c(2.1,2.1), iteraciones_max = 30)
```

```
## Punto de partida:  2.1 2.1
## Valor función (inicial):  0.720983
## Valor función (final):  1727302
## Solución:  673.221 800.9884
```

```
# Lo mostramos gráficamente
plot(resultado04$valor_funcion, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "f(x,y)", main = "Inicio (2.1, 2.1)")

## Punto (3, 3) ----------------------------------------------------------------------
resultado04 = Metodo_Newton(f, f_dev, f_hess, vini = c(3,3), iteraciones_max = 30)
```

```
## Punto de partida:  3 3
## Valor función (inicial):  3
## Valor función (final):  2.900405
## Solución:  2.949129 2.974568
```

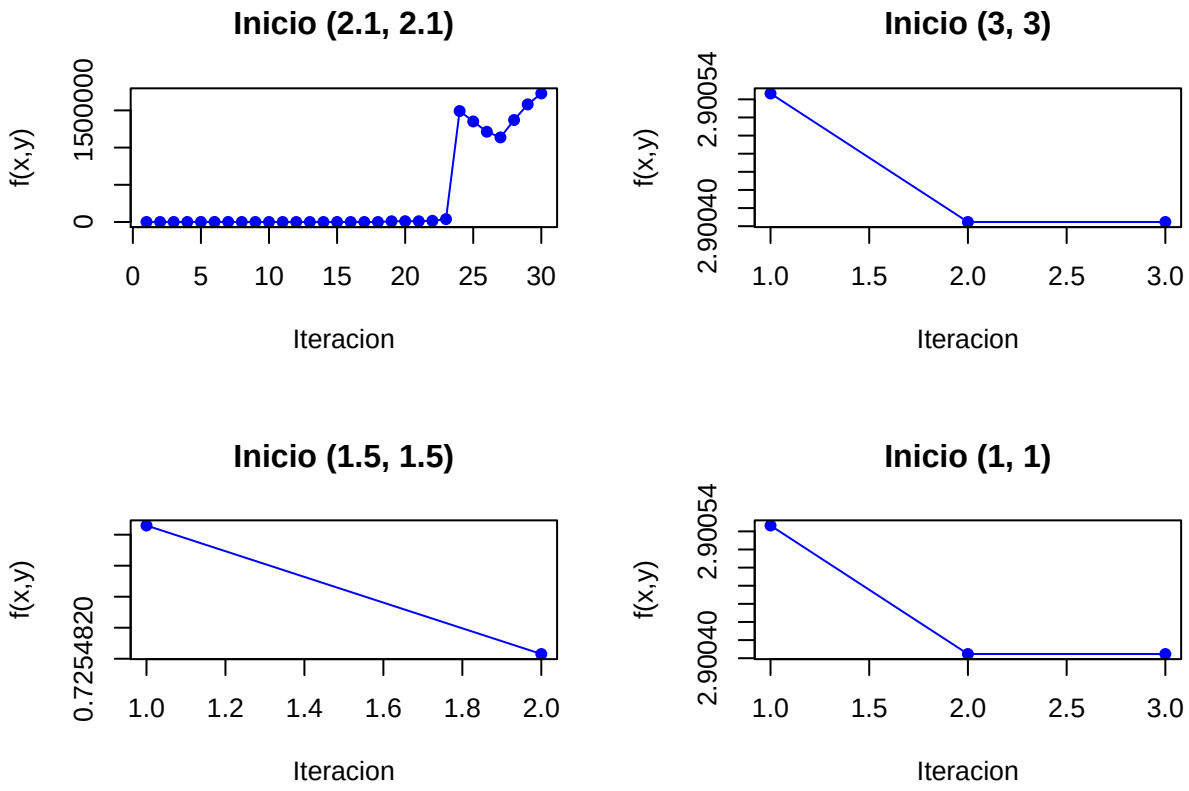
```
# Lo mostramos gráficamente
plot(resultado04$valor_funcion, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "f(x,y)", main = "Inicio (3, 3)")

## Punto (1.5, 1.5) ------------------------------------------------------------------
resultado04 = Metodo_Newton(f, f_dev, f_hess, vini = c(1.5,1.5), iteraciones_max = 30)
```

```
## Punto de partida:  1.5 1.5
## Valor función (inicial):  0.75
## Valor función (final):  0.7254821
## Solución:  1.524886 1.512195
```

```
# Lo mostramos gráficamente
plot(resultado04$valor_funcion, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "f(x,y)", main = "Inicio (1.5, 1.5)")
## Punto (1, 1) ----------------------------------------------------------------------
resultado04 = Metodo_Newton(f, f_dev, f_hess, vini = c(1,1), iteraciones_max = 30)
```

```
## Punto de partida:  1 1
## Valor función (inicial):  3
## Valor función (final):  2.900405
## Solución:  1.050871 1.025432
```

```
# Lo mostramos gráficamente
plot(resultado04$valor_funcion, type = "o", pch = 16, col = 4, xlab = "Iteracion",
     ylab = "f(x,y)", main = "Inicio (1, 1)")
```



Como podemos ver, este método siempre llega a un punto de forma directa, por lo que podemos considerar ese punto como un mínimo local.

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

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.

Para comparar los dos algoritmos, vamos a escoger por ejemplo, el punto de inicio (1,1) y vamos a ver el resultado que muestran ambos algoritmos.

```
# Newton
resultado05 = Metodo_Newton(f, f_dev, f_hess, vini = c(1,1), iteraciones_max = 30)
```

```
## Punto de partida:  1 1
## Valor función (inicial):  3
## Valor función (final):  2.900405
## Solución:  1.050871 1.025432
```

```
# Gradiente Descendente con 0.1
resultado06 = GD(f, f_dev, vini =c(1,1), eta = 0.1, umbral = 10^(-4),
                 iteraciones_max = 30)
```

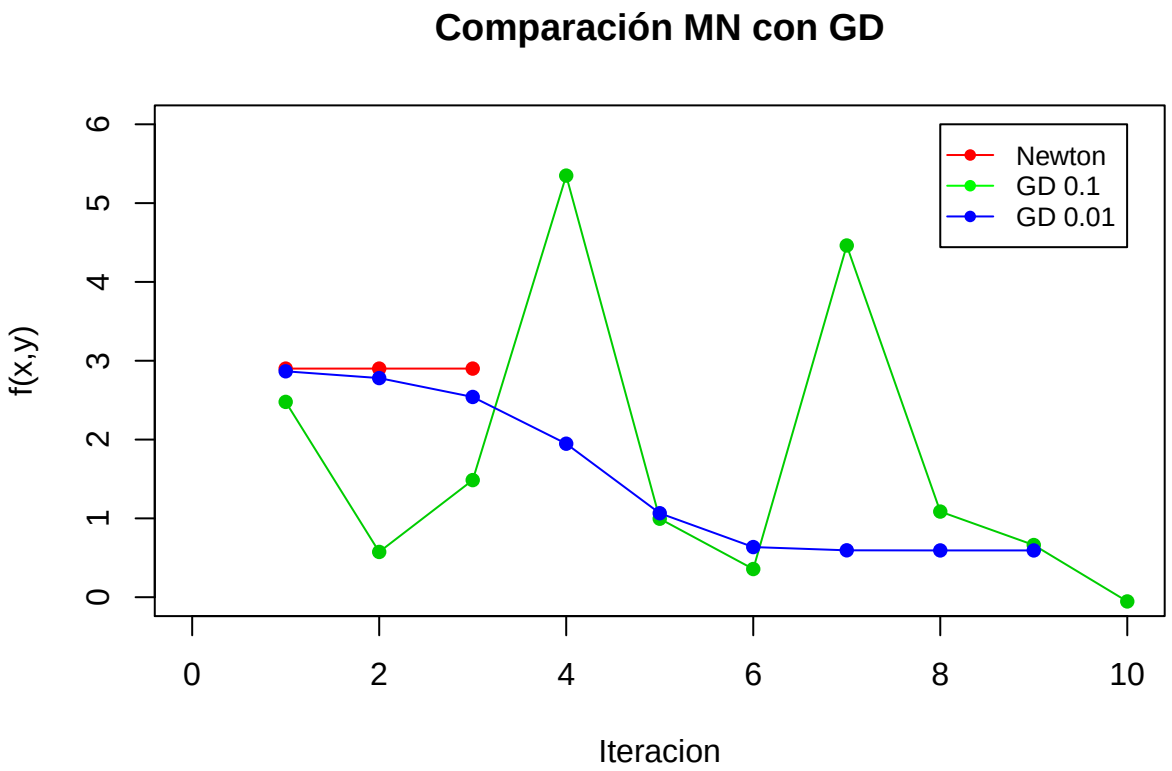
```
## Punto de partida:  1 1
## Gradiente (inicial):  -2 -4
## Valor función (inicial):  3
## Gradiente (final):  -8.590336 2.1004
## Solución:  1.611878 1.348358
## Valor función (final):  -0.05388005
```

```
## Iteraciones:  9
```

```
# Gradiente Descendente con 0.01
resultado07 = GD(f, f_dev, vini =c(1,1), eta = 0.01, umbral = 10^(-4),
                 iteraciones_max = 30)
```

```
## Punto de partida:  1 1
## Gradiente (inicial):  -2 -4
## Valor función (inicial):  3
## Gradiente (final):  0.01574209 -0.007690211
## Solución:  0.7821293 1.2871
## Valor función (final):  0.5932713
## Iteraciones:  8
```

```
# Abrimos plot
plot(c(0,10), c(0,6), type = "n", xlab = "Iteracion", ylab = "f(x,y)",
     main = "Comparación MN con GD")

# Mostramos gráficamente las tres líneas
# Newton
lines(resultado05$valor_funcion, type = "o", pch = 16, col = 2)
# Gradiente Descendente con 0.1
lines(resultado06$valor_funcion, type = "o", pch = 16, col = 3)
# Gradiente Descendente con 0.01
lines(resultado07$valor_funcion, type = "o", pch = 16, col = 4)

# Agregamos una leyenda
legend(8, 6,c("Newton", "GD 0.1", "GD 0.01"), cex = 0.8,
       col =c("red", "green", "blue"),pch=16, lty=c(1,1,1))
```



Como se puede apreciar, el algoritmo de gradiente descendente alcanza el entorno del mínimo local. Por eso

es mejor, es decir, se queda menos estancado y consigue alcanzar valores más bajos de la función con las mismas iteraciones. De todas maneras, el método de Newton, converge en muy pocas iteraciones.

Para comprender mejor el resultado, vamos a recoger todos estos valores en una tabla. Para ello vamos hacer una comparación, basándonos en el método de newton contra el gradiente descedente con $\eta = 0.01$.

Algoritmo	(x_0, y_0)	(x_f, y_f)	$f(x_0, y_0)$	$f(x_f, y_f)$
Gradiente Descendente	$(3, 3)$	$(3.217871, 2.7129)$	3	0.5932713
Método Newton	$(3, 3)$	$(2.949129, 2.974568)$	3	2.900405
Gradiente Descendente	$(1.5, 1.5)$	$(1.419677, 1.61600)$	0.75	-0.01244002
Método Newton	$(1.5, 1.5)$	$(1.524886, 1.512195)$	0.75	0.7254821
Gradiente Descendente	$(1, 1)$	$(0.7821293, 1.2871)$	3	0.5932713
Método Newton	$(1, 1)$	$(1.050871, 1.025432)$	3	2.900405

Como podemos ver en la tabla, el poder de minimización del método de Newton es un poco menor que el del gradiente descendiente, pero tiene la ventaja de que realiza menos iteraciones que el gradiente descendiente, y nos da un mínimo con cierta calidad, pero muy dependiente del punto de inicio de la función.

```
# Después de crear una gráfica o iniciar un apartado paramos la ejecución 3 segundos
Sys.sleep(3)
```

Ejercicio 3

Repetir el experimento de RL (punto 2) 100 veces con diferentes funciones frontera y calcule el promedio.

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

Básicamente este ejercicio es repetir el apartado b del ejercicio dos, 100 veces.

```
# Establecemos una semilla por defecto, para la generación de números aleatorios
set.seed(3)

Eout02 <- 0 # Establecemos eout a cero
Ein02 <- 0 # Establecemos Ein a cero

# Ejecutar el experimento 100 veces
for (i in seq(1,100)){

  # Ein
  # ------------------------------------------------------------------------
  # Generamos aleatoriamente una lista de números aleatorios
  datos04 = simula_unif(N = 100, dim = 2, rango = c(0,2))
  # Generamos una recta
  recta04 <- simula_recta(intervalo = c(0,2))
  # Etiquetamos los puntos a partir de la función de la recta
  etiquetas04 <- sign(datos04[,2] - recta04[1]*datos04[,1] - recta04[2])
  # Calculamos la recta de regresión que separa ambas clases
  recta_regresion03 <- RL1(datos04, etiquetas04)
  recta_regresion04 = c(-recta_regresion03[1]/recta_regresion03[3],
                        - recta_regresion03[2]/recta_regresion03[3])
```

```
  # Etiquetamos los puntos a partir de la recta de regresión
  etiquetas_regresion05 <- sign(datos04[,2] - recta_regresion04[1]*datos04[,1]
                                - recta_regresion04[2])
  # Calculamos el valor de Ein como las etiquetas que son distintas,
  # partido el total de las etiquetas
  Ein02 = Ein02 + sum(etiquetas04 != etiquetas_regresion05) / length(etiquetas04)

  # Eout
  # ----------------------------------------------------------------------
  # Generamos aleatoriamente una lista de números aleatorios
  datos04.test = simula_unif(N = 100, dim = 2, rango = c(0,2))
  # Generamos una recta
  recta04.test <- simula_recta(intervalo = c(0,2))
  # Etiquetamos las etiquetas a partir de la función de la recta
  etiquetas04.test <- sign(datos04.test[,2] - recta04.test[1]*datos04.test[,1]
                           - recta04.test[2])

  # Etiquetamos los puntos a partir de la recta de regresión ya creada
  etiquetas_regresion02.test <- sign(datos04.test[,2]
                                     - recta_regresion04[1]*datos04.test[,1]
                                     - recta_regresion04[2])
  # Calculamos el valor de Eout
  Eout02 = Eout02 + sum(etiquetas04.test != etiquetas_regresion02.test) /
    length(etiquetas04.test)
}

cat ("Ein:", Ein02/100)
```

```
## Ein: 0.3858
```

```
cat ("\nEout: ", Eout02/100)
```

```
##
## Eout:  0.4298
```

Al ejecutar el experimento 100 veces obtenemos que $E_{in} = 0.38$ y que $E_{out} = 0.42$. Vemos que ambos valores siguen estando próximos, como comentamos en el ejercicio 2. Pero ahora el *test* sube, ya que hay nuevos datos que se intentan ajustar a los datos del *train*. Como vemos, obtenemos un error medio-alto, ya que estamos distribuyendo los datos de una maneras que no tienen ruido, por eso la función que hace la regresión es prácticamente la verdadera f.

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

Se nos pide que digamos las épocas que tarda nuestro algoritmo en converger, que no es más que las iteraciones que ha tardado. Ya que una *época* es un pase completo a través de los N datos. Por tanto, solo nos hace falta obtener las iteraciones y hacer la media.

```
sol <- replicate(100, RL1(datos04, etiquetas04)[4])
mean(sol)
```

```
## [1] 506.95
```

Como se aprecia, se obtiene el máximo de iteraciones que se le pone al algoritmo por defecto. Puesto que la condición de la norma no sirve como parada, para este caso.