Práctica 2 - Aprendizaje Automático

Marta Gómez Macías

8 de abril de 2016

Índice

1 Modelos lineales

1.1 Gradiente descendente.

Implementar el algoritmo de gradiente descendente

a) Considerar la función no lineal de error $E(u,v) = (ue^v - 2ve^{-u})^2$. Usar gradiente descendente y minimizar esta función de error, comenzando desde el punto $(u,v) = (1,1)$ y usando una tasa de aprendizaje $\eta = 0,1$.

1. **Calcular analíticamente y mostrar la expresión del gradiente de la función $E(u,v)$.**
2. **¿Cuántas iteraciones tarda el algoritmo en obtener por primera vez un valor de $E(u,v)$ inferior a 10^{-14} . (Usar flotantes de 64 bits).**
3. **¿Qué valores de (u,v) obtuvo en el apartado anterior cuando alcanzo el error de 10^{-14}.**

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

1. **Usar gradiente descendente para minimizar esta función. Usar como valores iniciales $x_0 = 1$, $y_0 = 1$, la tasa de aprendizaje $\eta = 0,01$ y un máximo de 50 iteraciones. Generar un gráfico de cómo desciende el valor de la función con las iteraciones. Repetir el experimento pero usando $\eta = 0,1$, comentar las diferencias.**
2. **Obtener el valor mínimo y los valores de las variables que lo alcanzan cuando el punto de inicio se fija: $(0,1,\ 0,1), (1,\ 1), (-0,5,\ 0,5), (-1,\ -1)$. Generar una tabla con los valores obtenidos. ¿Cuál sería su conclusión sobre la verdadera dificultad de encontrar un mínimo global de una función arbitraria?**

Para calcular la derivada de la expresión (*apartado a1*), he calculado $E'(u)$ y $E'(v)$. En cada una he tomado la otra como constante, es decir, cuando he derivado u he tomado a v como una constante y cuando he derivado v he tomado a u como una constante. Así, he obtenido el siguiente resultado:

$$E'(u) = 2(ue^v - 2ve^{-u}) \cdot (e^v + 2ve^{-u})$$

$$E'(v) = 2(ue^v - 2ve^{-u}) \cdot (ue^v - 2e^{-u})$$

La implementación realizada del **Gradiente descendiente** es:

```
source("../P1/p1.R")
library("Deriv")
library(ggplot2)
library("orthopolynom") # para polinomios de Legendre

# Version con las derivadas hechas por mí
fau <- function(u,v) 2*(u*exp(v) - 2*v*exp(-u)) * (exp(v) + 2*v*exp(-u))
fav <- function(u,v) 2*(u*exp(v) - 2*v*exp(-u)) * (u*exp(v) - 2*exp(-u))

funciones <- c(function(x,y) (x*exp(y) - 2*y*exp(-x))^2,
    function(x,y) x^2 + 2*y^2 + 2*sin(6.28*x)*sin(6.28*y))

gradient_descent <- function(eta=0.1, precision=10^(-14), init=c(1,1),
    fx = fau, fy=fav, fu = funciones[[1]], max_iter=50) {
    x_old = 0
    y_old = 0
    x_new = init[1]
    y_new = init[2]
    iter = 0
    while(abs(fx(x_new,y_new)) > precision & iter < max_iter
        & abs(x_new - x_old) > precision) {
        iter = iter + 1
        x_old = x_new
        y_old = y_new
        x_new = x_old - eta * fx(x_old, y_old)
        y_new = y_old - eta * fy(x_old, y_old)
    }
    c(fu(x_new,y_new),x_new,y_new,iter)
}
```

Para no tener que hacer las derivadas a mano, he usado el paquete Deriv. A parte de usar el paquete `Deriv` también he cambiado la condición de parada para poder tener en cuenta tanto el valor de x como el de y:

```
# Version con las derivadas hechas por Deriv
gradient_descent_deriv <- function(eta=0.1, precision=10^(-14), init=c(1,1),
    fu = funciones[[1]], max_iter=50) {

    df = Deriv(f=fu,x=c('x','y'))
    x_old = 0
    y_old = 0
    x_new = init[1]
    y_new = init[2]
```

```
    iter = 0
    while(norm(as.matrix(df(x_new,y_new)), type="F") > precision & iter < max_iter
            & abs(fu(x_new, y_new) - fu(x_old, y_old)) > precision) {
        iter = iter + 1
        x_old = x_new
        y_old = y_new
        derivada = df(x_old, y_old)
        x_new = x_old - eta * derivada[1]
        y_new = y_old - eta * derivada[2]
    }
    c(fu(x_new,y_new),x_new,y_new,iter)
}
```

Con ambos obtengo un resultado parecido:

```
## [1] "Versión sin deriv"
```

```
## [1] 1.850819e-31 4.473628e-02 2.395873e-02 1.700000e+01
```

```
## [1] "Versión con Deriv"
```

```
##            x            x            y
## 6.635388e-18 4.473628e-02 2.395873e-02 1.100000e+01
```

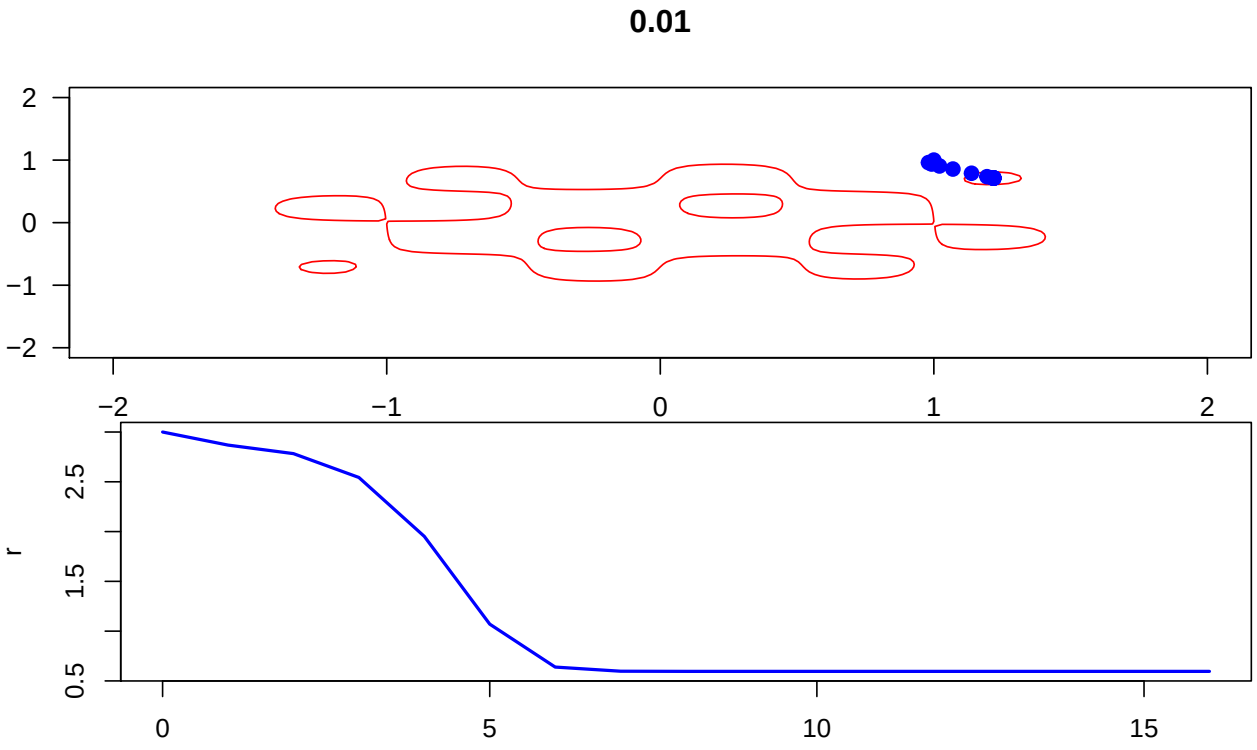
Respondiendo a la pregunta del apartado *a2*, el número de iteraciones que se necesitan para converger es 17 para el caso en el que no usamos `Deriv` y 11 para el caso en el que sí. El número de iteraciones cambia porque con ésta última versión se cumple antes la condición de parada. Por último, respecto a la pregunta del apartado *a3* sobre los valores obtenidos, han sido $x = 0,04473628$ e $y = 0,02395873$ en ambos casos.

Para realizar el apartado *b*, he añadido funcionalidad para pintar la función y ver cómo desciende el gradiente:

```
# Versión con las derivadas hechas por Deriv y que hace un gráfico de cómo va
# evolucionando el gradiente
dibuja_funcion_sin_puntos <- function(funcion, titulo,intervalo=c(-3,3)) {
    x <- y <- seq(intervalo[1],intervalo[2],length=100)
    z <- outer(x,y,funcion)             # calculamos los puntos de la funcion
    contour (                           # la representamos
        x=x, y=x, z=z,
        levels=1, las=1, drawlabels=FALSE, main=titulo, col="red"
    )
}

dibuja_funcion_puntos <- function (puntos = puntos_funcion, func=funciones[[2]]) {
    p = matrix(puntos[!is.na(puntos)], ncol=3)
    r = apply(X=p, MARGIN=1, FUN=function(vec) func(vec[1],vec[2]))
    datos = cbind(p[,3], r)
    plot(datos, type="l", lwd=2, col="blue")
    abline(a=0,b=0, lwd=2, col="red")
}

gradient_descent_deriv_grafica <- function(eta=0.01, precision=10^(-14), init=c(1,1),
    fu = funciones[[2]], max_iter=50, intervalo_g=c(-2,2)) {
```

```
    df = Deriv(f=fu,x=c('x','y'))
    x_old = 0
    y_old = 0
    x_new = init[1]
    y_new = init[2]
    iter = 0
    puntos_funcion = matrix(ncol=3, nrow=max_iter) # matriz vacia para guardar puntos
    dibuja_funcion_sin_puntos(funcion=fu, titulo=eta, intervalo=intervalo_g)

    while(norm(as.matrix(df(x_new,y_new)), type="F") > precision & iter < max_iter
            & abs(fu(x_new, y_new) - fu(x_old, y_old)) > precision) {
        points(x=x_new, y=y_new, col="blue", lwd=2, pch=19)
        puntos_funcion[iter+1,] = c(x_new, y_new, iter)
        iter = iter + 1
        x_old = x_new
        y_old = y_new
        derivada = df(x_old, y_old)
        x_new = x_old - eta * derivada[1]
        y_new = y_old - eta * derivada[2]
    }
    # print("Pulsa s para ejecutar la siguiente gráfica...")
    # scan(what=character(), n=1)
    dibuja_funcion_puntos(puntos_funcion, fu)
    c(fu(x_new,y_new),x_new,y_new,iter)
}
```

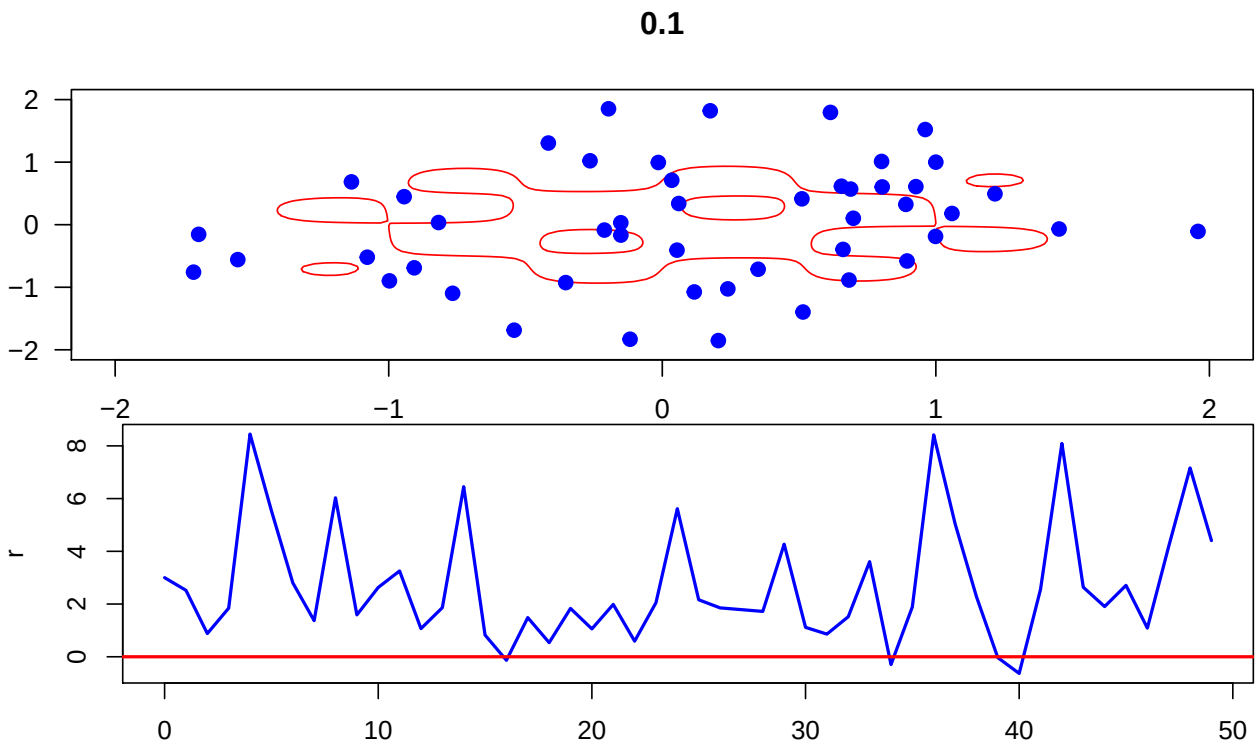
Los resultados obtenidos son:




```
##          x          x          y
##  0.5958059  1.2186543  0.7131354 17.0000000
```

Como se ve en la gráfica, el gradiente se ha quedado en un mínimo local. Además, también se ve como los puntos están muy juntos unos con otros, esto se debe a que η es muy pequeño.

En el caso de $\eta = 0,1$ obtenemos el siguiente resultado:




```
##          x          x          y
##  0.9795583  1.0127282 -0.0552796 50.0000000
```

En este caso hemos elegido un η demasiado grande, por lo que el gradiente diverge y supera el número máximo de iteraciones. El gradiente diverge en este caso porque se encuentra muy cerca del mínimo local.

Para el apartado *b2* he obtenido los siguientes resultados

```
## [1] "Punto de inicio x=0,1 e y=0,1"
```

```
##          x          x          y
## -1.8199034  0.2439225 -0.2380348 21.0000000
```

```
## [1] "Punto de inicio x=1 e y=1"
```

```
##          x          x          y
##  0.5958059  1.2186543  0.7131354 17.0000000
```

```
## [1] "Punto de inicio x=-0,5 e y=-0,5"
```

```
##          x          x          y
## -1.3318235 -0.7317298 -0.2379641 17.0000000
```

```
## [1] "Punto de inicio x=-1 e y=-1"
```

```
##          x          x          y
##  0.5958059 -1.2186543 -0.7131354 17.0000000
```

Estos resultados los reflejo en la siguiente tabla:

punto de inicio	$f(x, y)$	x	y	iteraciones
$(0,1,\ 0,1)$	$-1,8199034$	$0,2439225$	$-0,2380348$	21
$(1,\ 1)$	$0,5958059$	$1,2186543$	$0,7131354$	17
$(-0,5,\ -0,5)$	$-1,3318235$	$-0,7317298$	$-0,2379641$	17
$(-1,\ -1)$	$0,5958059$	$-1,2186543$	$-0,7131354$	17

El poder encontrar un el mínimo global depende tanto de la complejidad de la función como del punto de partida. En este caso, al ser la función una especie de función elíptica es mucho más complicado poder encontrar un buen punto en el que comenzar. Además, esta analogía de función elíptica se aprecia en la tabla, cuando el punto de partida es (1, 1) obtenemos los mismos resultados que con el punto de partida $(-1, -1)$ pero con x e y de signo contrario. Para poder encontrar realmente el mínimo global tendríamos que conocer la estructura de la función para saber exactamente un buen punto de partida, si no, es prácticamente imposible.

1.2 Coordenada descendente

En este ejercicio comparamos la eficiencia de la técnica de optimización de "coordenada descendente" usando la misma función del ejercicio 1.1a. En cada iteración, tenemos dos pasos a lo largo de dos coordenadas. En el Paso-1 nos movemos a lo largo de la coordenada u para reducir el error (suponer que se verifica una aproximación de primer orden como en gradiente descendente), y el Paso-2 es para reevaluar y movernos a lo largo de la coordenada v para reducir el error (hacer la misma hipótesis que en el Paso-1). Usar una tasa de aprendizaje $\eta = 0,1$.

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

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

La función implementada para la *Coordenada Descendente* es:

```
coordenada_descendente <- function(max_iter=15,fu=funciones[[1]], eta=0.1,
    init=c(1,1), precision=10^(-14)) {

    df = Deriv(f=fu,x=c('x','y'))
    x_new = init[1]
    y_new = init[2]
    x_old = 0
    y_old = 0
    iter = 0
    while(iter < max_iter & norm(as.matrix(df(x_new,y_new)), type="F") > precision) {
        iter = iter + 1
        # Paso 1: nos movemos a lo largo de la coordenada u
        x_old = x_new
        derivada = df(x_old, y_old)
        x_new = x_old - eta * derivada[1]
        # Paso 2: reevaluamos y nos movemos a lo largo de la coordenada v
        derivada = df(x_new, y_old)
        y_new = y_old - eta * derivada[2]
```

```
    }
    c(fu(x_new,y_new),x_new,y_new, iter)
}
```

En ella, derivamos en el Paso-1 sólo la coordenada u (`x_old`) y en el siguiente, Paso-2, derivamos la coordenada v a partir del nuevo valor de la coordenada u obtenido en el paso anterior:

$$u' = f'(u, v) \qquad v' = f'(u', v)$$

Tras 15 iteraciones, obtenemos el siguiente resultado:

```
##            x            x          y.x
## 9.801081e-05 3.518437e-02 1.333959e-02 1.500000e+01
```

Mientras que con *Gradiente descendente*, obtenemos esta otra, realizando 30 iteraciones:

```
##            x            x            y
## 6.635388e-18 4.473628e-02 2.395873e-02 1.100000e+01
```

La calidad de la solución obtenida por el *Gradiente descendente* es peor, ya que el valor de la función f es mayor y además, converge en más iteraciones que la *Coordenada descendente*.

Si probamos con $\eta = 0,01$ obtenemos los siguientes resultados para la *Coordenada Descendente*:

```
##            x            x          y.x
##  0.544462720  0.738569103  0.003205761 15.000000000
```

y para el *Gradiente Descendente*:

```
##            x            x            y
## 6.630464e-07 4.589310e-01 8.427461e-01 3.000000e+01
```

En este caso, obtenemos unos resultados mejores con *Gradiente Descendente*, pero peores que con $\eta = 0,1$.

1.3 Método de Newton

Implementar el algoritmo de minimización de Newton y aplicarlo a la función $f(x, y)$ dada en el ejercicio 1.1b. Desarrolle los mismos experimentos usando los mismos puntos de inicio.

- **Generar un gráfico de cómo desciende el valor de la función con las iteraciones.**
- **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 descendiente.**

La función del *Método de minimización de Newton* implementada es:

```
newton <- function(fu=funciones[[2]], init=c(1,1), precision=10^(-14), max_iter=50,
    intervalo_g=c(-2,2)) {

    df = Deriv(f=fu,x=c('x','y'), nderiv=1:2)
    x_old = 0
    y_old = 0
    x_new = init[1]
    y_new = init[2]
    iter = 0

    puntos_funcion = matrix(ncol=3, nrow=max_iter)
    dibuja_funcion_sin_puntos(funcion=fu, titulo=paste("x=",init[1]," y=",init[2]),
        intervalo = intervalo_g)

    while(iter < max_iter & abs(fu(x_old, y_old) - fu(x_new, y_new)) > precision &
            norm(as.matrix(df(x_new, y_new)$`1`), type="F") > precision) {
        points(x=x_new, y=y_new, col="blue", lwd=2, pch=19)
        puntos_funcion[iter+1,] = c(x_new, y_new, iter)
        iter = iter + 1
        x_old = x_new
        y_old = y_new
        derivada = df(x_old, y_old)
        d2inversa = solve(matrix(derivada$`2`, ncol=2))
        # formula: http://web.stanford.edu/class/msande311/lecture13.pdf pag 6
        x_new = x_old - d2inversa[1,] %*% derivada$`1`
        y_new = y_old - d2inversa[2,] %*% derivada$`1`
    }
    # print("Pulsa s para ejecutar la siguiente gráfica...")
    # scan(what=character(), n=1)
    dibuja_funcion_puntos(puntos_funcion, fu)
    c(fu(x_new,y_new),x_new,y_new,iter)
}
```

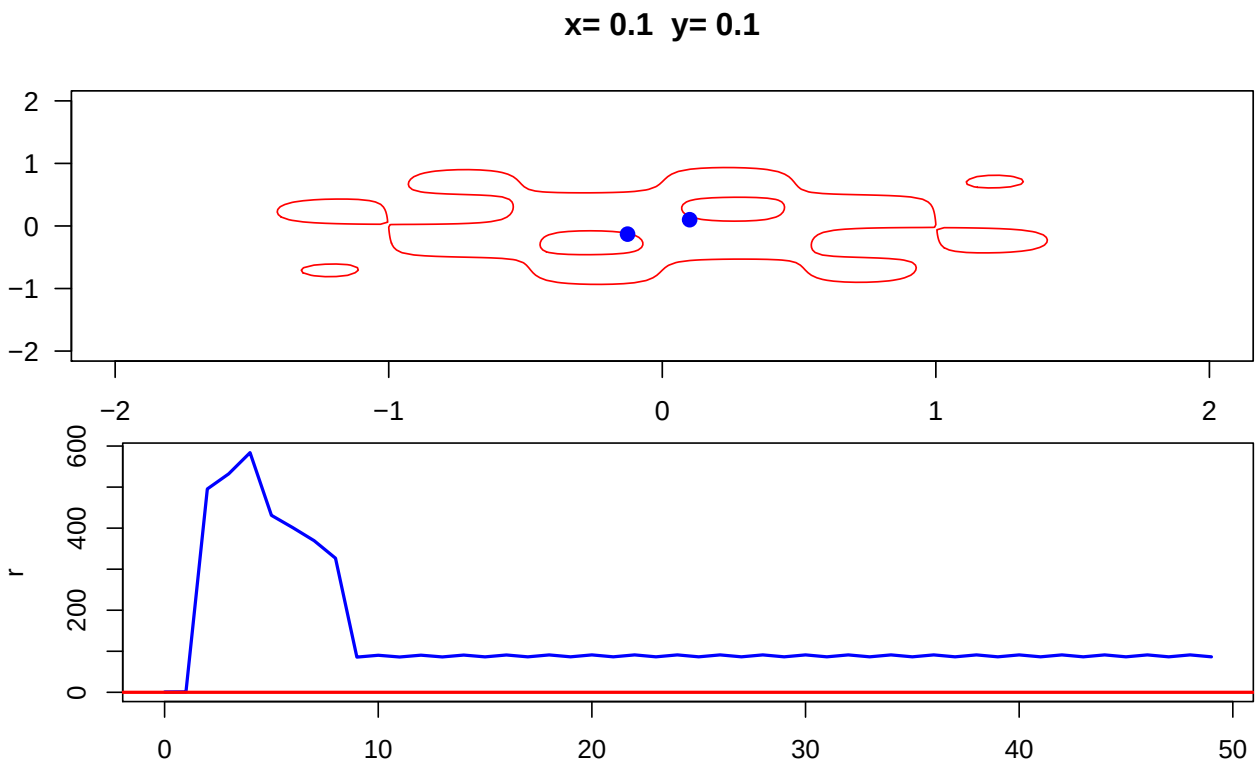
En ella, aplicamos el mismo proceso que en el *Gradiente Descendente* pero cambiando la fórmula usada, que, como se indica en esta presentación, en este caso sería:

$$x^{k+1} = x^k - (\nabla^2 f(x^k))^{-1} \cdot \nabla f(x^k)$$

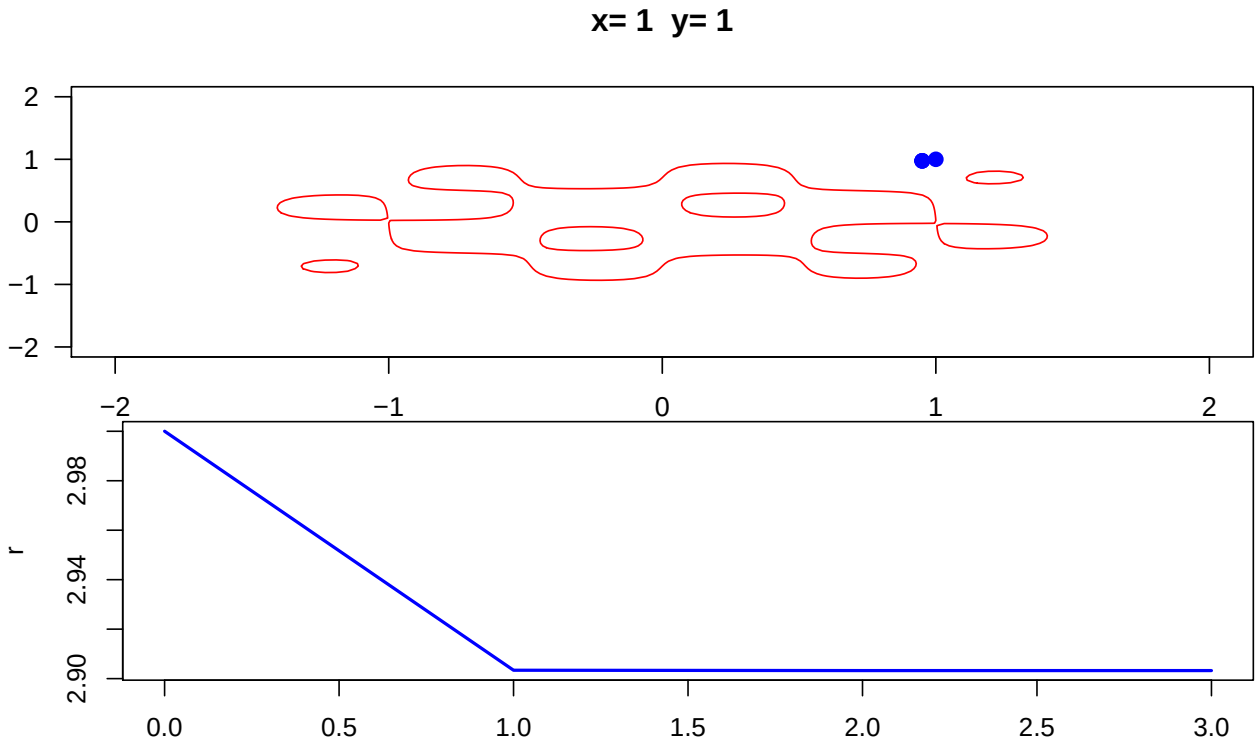
$$y^{k+1} = y^k - (\nabla^2 f(y^k))^{-1} \cdot \nabla f(y^k)$$

En nuestro caso, al tener la segunda derivada cuatro componentes, hemos tenido que trabajar con ellos como si fueran una matriz.

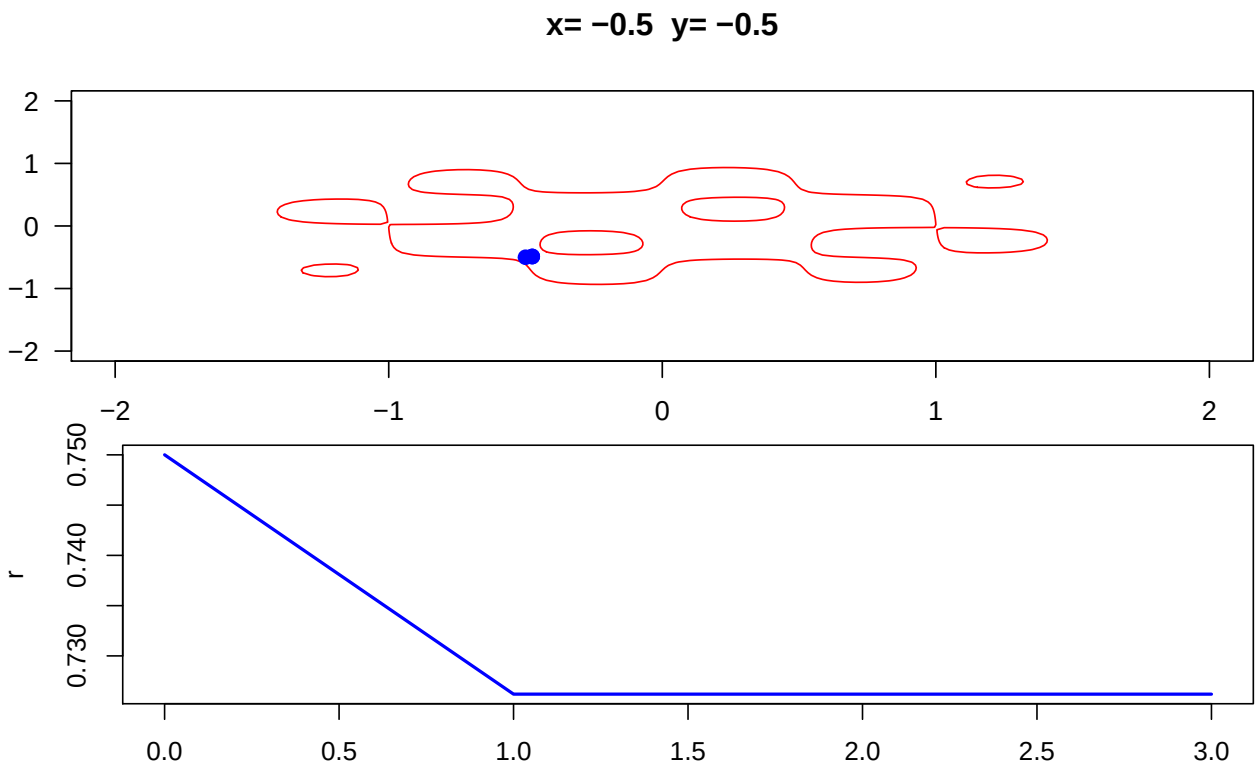
Para cada uno de los puntos de inicio, he obtenido las siguientes gráficas:



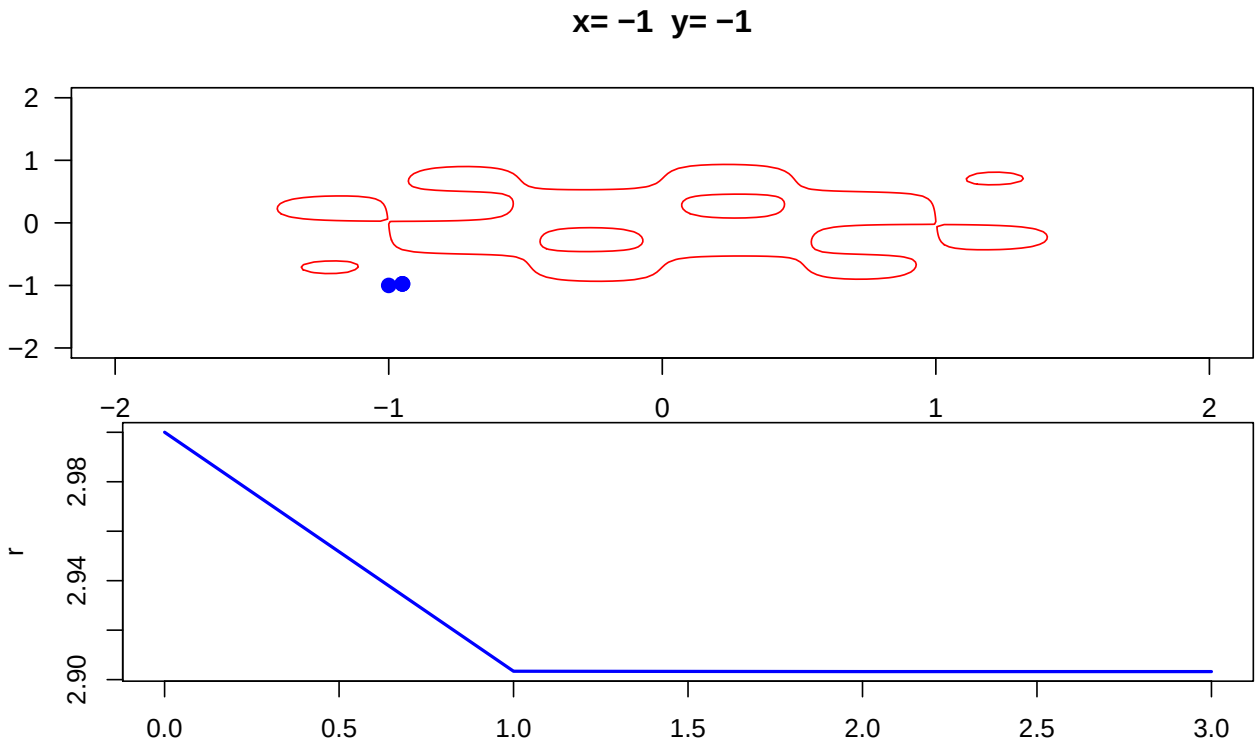

```
## [1] 91.140162 -5.347845 -5.650946 50.000000
```




```
## [1] 2.9032461 0.9495569 0.9750344 4.0000000
```




```
## [1]  0.7261940 -0.4753295 -0.4880400  4.0000000
```




```
## [1]  2.9032461 -0.9495569 -0.9750344  4.0000000
```

En resumen, la siguiente tabla refleja los resultados obtenidos:

punto de inicio	$f(x, y)$	x	y	iteraciones
$(0,1,\ 0,1)$	$91,140162$	$-5,347845$	$-5,650946$	50
$(1,\ 1)$	$2,9032461$	$0,9495569$	$0,9750344$	4
$(-0,5,\ -0,5)$	$0,7261940$	$-0,4753295$	$-0,4880400$	4
$(-1,\ -1)$	$2,9032461$	$-0,9495569$	$-0,9750344$	4

Con *Gradiente Descendente* obtenemos mejores resultados cuando empezamos desde $\pm(1,\ 1)$ pero, obtenemos un mejor resultado con el *Método de Newton* cuando empezamos con $(-0,5,\ -0,5)$. En el caso del punto de inicio $(0,1,\ 0,1)$, podemos concluir que es malísimo, ya que el método no llega a converger.

1.4 Regresión Logística

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

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

Seleccionar $N = 100$ puntos aleatorios $\{x_n\}$ de $\mathcal{X}$ y evaluar las respuestas de todos ellos $\{y_n\}$ respecto de la frontera elegida.

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

- **Inicializar el vector de pesos con valores 0.**
- **Parar el algoritmo cuando $||w^{(t-1)} - w^{(t)}|| < 0,01$, donde $w^{(t)}$ denota el vector de pesos al final de la época t. Una época es un pase completo a través de los N datos.**
- **Aplicar una permutación aleatoria de $1, 2, \cdots, N$ a los datos antes de usarlos en cada época del algoritmo.**
- **Usar una tasa de aprendizaje de $\eta = 0,01$.**

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

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

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

La función implementada es:

```
cross_entropy_error <- function(wlin, datos, recta) {
    etiquetas = obten_param_recta(puntos=datos, re=recta)
    datos = cbind(rep(1, nrow(datos)), datos)
    aux = apply(X=datos, FUN = function(d) d%*%wlin, MARGIN=1)
    error = mapply(FUN = function(d, l) log (1 + exp(-l*d)), d = aux,
        l = etiquetas)
```

```
    mean(error)
}

logistic_regression_stochastic_gradient_descent <- function(eta = 0.01, max_iter=150,
    datos = simula_unif(dim=2, N=100, rango=-1:1), etiquetas) {
    w_new = c(0,0,0)
    w_old = w_new+1 # inicializamos el vector de pesos
    # añadimos a la matriz datos una tercera columna para poder hacer el producto vectorial
    datos = cbind(rep(1, nrow(datos)), datos)
    i = 0

    mod <- function(w) {
        raiz = sqrt(w[1]*w[1] + w[2]*w[2] + w[3]*w[3])
        raiz
    }

    # parada: modulo de la diferencia de los dos vectores
    while (mod(w_old - w_new) >= 0.01 & i < max_iter) {
        i = i + 1
        # hacemos una permutación aleatoria de los datos
        w_old = w_new
        datos = datos[sample.int(nrow(datos)),]
        for (j in 1:nrow(datos)) {
            # calculamos el gradiente de error para cada punto con la fórmula
            # de la página 98 del libro de teoría
            error = (-etiquetas[j]*datos[j,])/(1 + exp(etiquetas[j]*(w_new%*%datos[j,])))
            # actualizamos el peso usando la probabilidad de error de ese punto
            w_new = w_new - eta/nrow(datos) * error
        }
    }
    w_new
}
```

Para implementar la función he seguido la descripción de la página 98 del libro de teoría y el enunciado del ejercicio. Le he dado 150 iteraciones en vez de sólo 50 para que el algoritmo pueda converger correctamente.

Para calcular el *error de entropía cruzada* he usado la siguiente fórmula:

$$E_{out} = \frac{1}{N}\sum_{n=1}^{N} ln\left(1 + e^{-y_n w^T x_n}\right)$$

El conjunto de datos y etiquetas usado para calcular dicho error, es generado de la misma forma que se generó el conjunto de prueba.

Los resultados obtenidos son:

```
## [1] 2.001147e-04 4.889764e-04 7.609002e-05
```

```
## Cross entropy error:  0.6930442
```

Obtenemos un error grande, ya que la probabilidad de error es del 69 %.

1.5 Clasificación de dígitos

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

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

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

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

c) **Obtener cotas sobre el verdadero valor de Eout. Pueden calcularse dos cotas, una basada en E_{in} y otra basada en E_{test}. Usar una tolerancia $\delta = 0,05$. ¿Qué cota es mejor?**

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

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

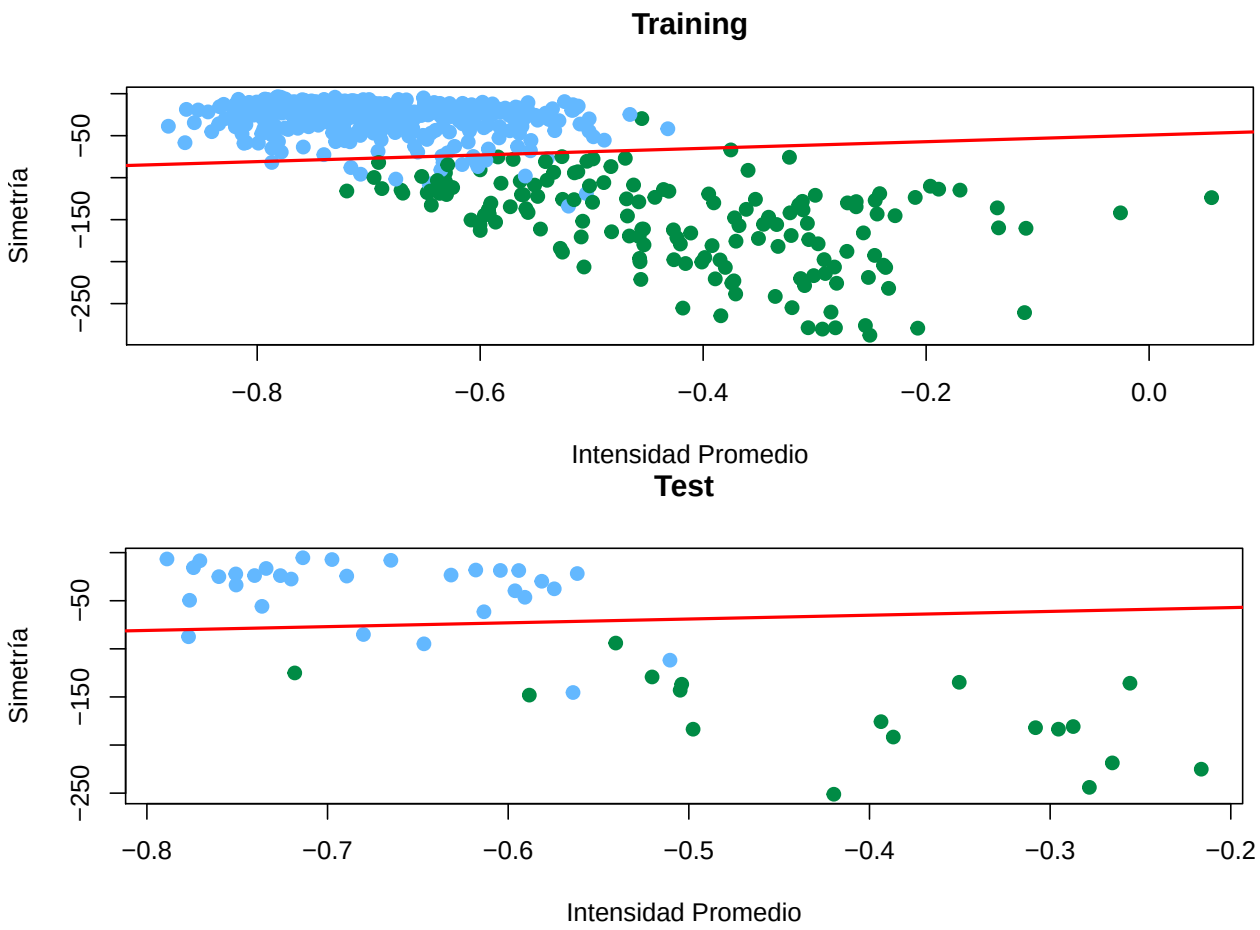
He hecho varias funciones: una para procesar los datos leídos de un fichero y otra para representación. Me he apoyado en las funciones hechas en la práctica 1 pero, en vez de usar el PLA Pocket, he usado el PLA normal porque convergía más rápido.

El código desarrollado es:

```
ajusta_fichero <- function(fichero="zip.train") {
    # obtenemos los datos
    fichero <- lee_fichero(fichero)
    # calculamos las etiquetas en función de si el número es 1 o no
    etiquetas <- apply(X=fichero, MARGIN=1,
        FUN=function(fila) {
            s = 0
            if (fila[1] == 1) {
                s = 1
            } else {
                s = -1
            }
            s
        })
    datos <- fichero[,2:257]
    # calculamos la media y la simetria
    sim <- calcula_sim(datos)
    med <- calcula_media(datos)

    list(med,sim,etiquetas)
}

regresion_pla <- function(parametros) {
    ajuste = Regress_Lin(parametros[[2]], parametros[[1]])
    # con el pocket tarda demasiado en converger
```

```
    pla = ajusta_PLA(datos=matrix(c(parametros[[1]],parametros[[2]]), ncol=2),
        label=parametros[[3]],vini=c(ajuste, 1), max_iter=50)
    list(ajuste, pla)
}

cota_eout <- function(N, dvc, delta=0.05, e) {
    eout = e + sqrt(8/N * log((4 * (2*N)^dvc + 1)/delta))
    eout
}

ej_5 <- function() {
    # ajustamos los datos de entrenamiento
    train <- ajusta_fichero()
    trainpla <- regresion_pla(train)

    # ajustamos los datos de test
    test <- ajusta_fichero(fichero="zip.test")

    # representamos la regresión
    representa_param(parametros=list(train[[1]],train[[2]]),titulo="Training",
        etiquetas=train[[3]])
    abline(a=trainpla[[2]][1], b=trainpla[[2]][2], col="red", lwd=2)
    # print("Pulsa s para ejecutar la siguiente gráfica...")
    # scan(what=character(), n=1)
    representa_param(parametros=list(test[[1]],test[[2]]),titulo="Test",
        etiquetas=test[[3]])
    abline(a=trainpla[[2]][1], b=trainpla[[2]][2], col="red", lwd=2)

    # calculamos E_in y E_test:
    ein = apply(X=cbind(train[[1]], train[[2]], train[[3]]), MARGIN=1,
        FUN=function(vec) experimento_a(wlin=trainpla[[1]], datos=cbind(vec[1],
            vec[2]), etiquetas=vec[3]))
    etest = apply(X=cbind(test[[1]], test[[2]], test[[3]]), MARGIN=1,
        FUN=function(vec) experimento_a(wlin=trainpla[[1]], datos=cbind(vec[1],
            vec[2]), etiquetas=vec[3]))
    # etest = experimento_b(mean(ein), 1, length(train[[1]]))
    cat("Ein = ", mean(ein),"\n")
    cat("Etest = ", mean(etest),"\n")

    # y por último, la cota de eout. Como estamos en un modelo lineal, dvc = 3
    eout_in = cota_eout(N=length(train[[1]]), dvc=3, e=mean(ein))
    eout_test = cota_eout(N=length(train[[1]]), dvc=3, e=mean(etest))

    cat("Eout (basado en Ein) = ", eout_in,"\n")
    cat("Etest (basado en Etest) = ", eout_test, "\n")
}
```

Así, obtenemos las siguiente gráficas. La primera hecha con los datos de training y la segunda, hecha con los datos de test. En ambas representamos la misma recta de regresión.




```
## Ein =  0.8221383
## Etest =  1.102169
## Eout (basado en Ein) =  1.407403
## Etest (basado en Etest) =  1.687433
```

Como se ve, la función obtenida clasifica correctamente los puntos según su simetría e intensidad media, aunque deja algunos puntos mal clasificados, los cuales podríamos considerar ruido.

Para calcular E_{in} y E_{test} he usado la siguiente fórmula para calcular el error dentro de la muestra. Para el caso de E_{in} he usado la muestra de entrenamiento y para E_{test}, la de entrenamiento. En ambas he usado los mismos pesos:

$$E_{in}(w) = \frac{1}{N}(w^T \cdot X^T \cdot X \cdot w - 2 \cdot w^T \cdot X^T \cdot y + y^T \cdot y)$$

Los resultados obtenidos son: $E_{in} = 0,8221383$ y $E_{test} = 1,102169$. Tal y como se esperaba, el E_{test} es mayor al E_{in}, pero no mucho más. Hemos hecho un modelo que se ha ajustado bastante bien al conjunto de test.

Por último, para calcular la cota sobre el verdadero valor de E_{out} con tolerancia $\delta = 0,05$ he usado el método descrito en la página 58 del libro de teoría:

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

Las cotas obtenidas han sido, basándome en E_{in} he obtenido $E_{out} = 1,407403$ y, basándome en E_{test}, he obtenido $E_{out} = 1,687433$. Ambas cotas son bastante similares, mucho más que los respectivos E_{in} y E_{test}, por lo que podemos concluir que hemos obtenido un buen modelo para clasificar los datos.

2 Sobreajuste

2.1 Sobreajuste

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

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

- **Generamos los coeficientes a_q a partir de muestras de una distribución $\mathcal{N}(0, 1)$ y escalamos dichos coeficientes de manera que $\mathbb{E}_{a,x}[f^2] = 1$ (Ayuda: Dividir los coeficientes por $\sqrt{\sum_{q=0}^{Q_f} \frac{1}{2q+1}}$).**
- **Generamos un conjunto de datos, $x_1, \cdots, x_N$ muestreando de forma independiente $\mathbb{P}(x)$ y los valores $y_n = f(x_n) + \sigma\epsilon_n$.**

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

a) Calcular g_2 y g_{10}.

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

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

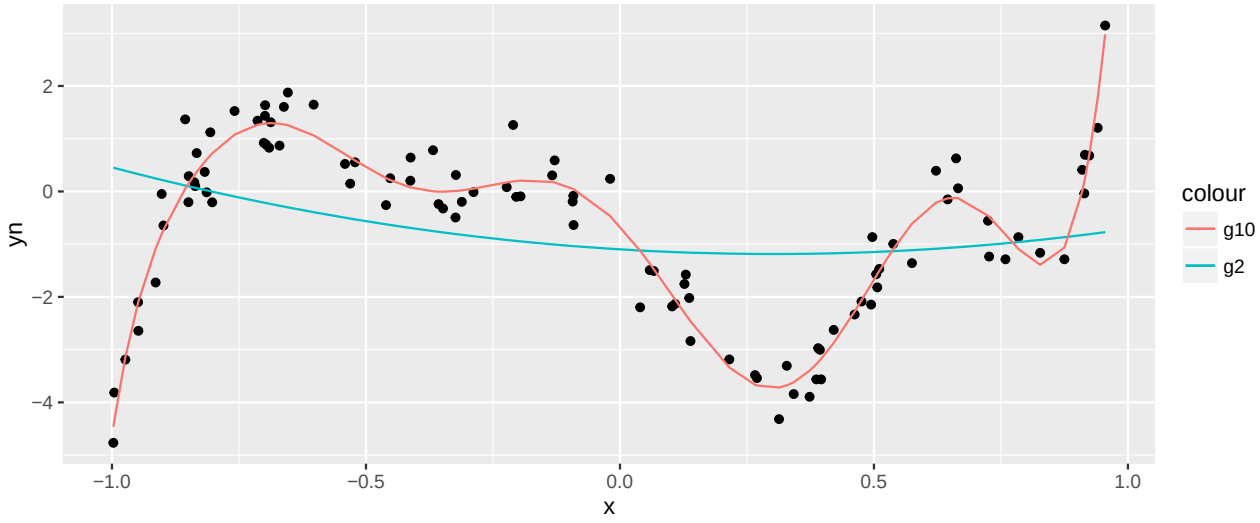
La función implementada es:

```
pintar_ggplot <- function(datos, x, yn, g2, g10) {
    graph <- (ggplot() + geom_point(data=as.data.frame(datos), aes(x=x, y=yn))
        + geom_line(aes(x=x, y=predict(g2), color="g2"))
        + geom_line(aes(x=x, y=predict(g10), color="g10")))
}

ej_2_1 <- function(intervalo=-1:1, prob=0.5, N=100, sigma=1, Qf=10) {
    # generamos los coeficientes con la distribución normal N(0,1)
    aq = as.vector(simula_gaus(N=Qf+1, dim=1, sigma=c(1)))
    d = sqrt(sum(sapply(X=0:Qf, FUN=function(q) 1/(2*q+1))))
    # escalamos los coeficientes generados
    aq = aq/d
    # generamos la función objetivo, como la suma de los polinomios de legendre
    # desde grado 0 hasta 20
```

[1] *independent and identically distributed*

```
    legendre = legendre.polynomials(n=Qf, normalized=T) # generamos los 20 primeros
    f = legendre[[1]]*aq[1]
    # sumamos los polinomios de legendre. NO se puede hacer con sum
    for (i in 2:length(legendre))
        f = f + legendre[[i]] * aq[i]
    # generamos los datos
    x = as.vector(simula_unif(N=N, dim=1, rango=intervalo))
    # generamos epsilon_n
    epsilon_n = as.vector(simula_gaus(N=N, dim=1, sigma=c(0.5)))
    # calculamos la funcion de etiquetado como f_e = f(x) + sigma * epsilon_n
    # para ello: 1. calculamos todos los f(x)
    fx = sapply(X=x, FUN=as.function(f))
    # 2. le sumamos a los fx calculados, sigma*epsilon
    yn = mapply(FUN=function(valor, en) valor + sigma*en, valor=fx, en=epsilon_n)
    # generamos la matriz de datos
    datos = cbind(x, yn)
    # y los datos que queremos ajustar
    datos_predecir = poly(x, degree=Qf)
    # hayamos los modelos para cada h
    g2 <- lm(yn ~ poly(x, degree=2),data=datos_predecir)
    g10 <- lm(yn ~ poly(x, degree=10),data=datos_predecir)

    list(datos, x, yn, g2, g10)
}
```

Con ella, obtenemos el siguiente resultado:



Como vemos, $g10$ se ajusta perfectamente a los datos, mientras que $g2$ se ajusta peor, pero capta muy bien la tendencia que tienen.

Cada punto es de la forma

$$\{x_n, y_n\} \qquad \text{donde } y_n = f(x_n) + \sigma \cdot \epsilon_n$$

Por tanto, cada punto tiene una componente de ruido σ. Debemos de normalizar f para poder decir que σ^2 es la verdadera cantidad de ruido que tienen nuestros datos.

2.2 Experimentación

Siguiendo con el punto anterior, usando la combinación de parámetros $Q_f = 20$, $N = 50$, $\sigma = 1$ ejecutar un número de experimentos (> 100) calculando en cada caso $E_{out}(g_2)$ y $E_{out}(g_{10})$. Promediar todos los valores de error obtenidos para cada conjunto de hipótesis, es decir

- $E_{out}(\mathcal{H}_2)$ **= promedio sobre experimentos (**$E_{out}(g_2)$**)**
- $E_{out}(\mathcal{H}_{10})$ **= promedio sobre experimentos (**$E_{out}(g_{10})$**)**

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

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

b) (Opcional) Usando la combinación de valores $Q_f \in \{1, 2, \cdots, 100\}$, $N \in \{20, 25, \cdots, 100\}$, $\sigma \in \{0,\ 0,05,\ 0,1,\ \cdots, 2\}$, se obtiene una gráfica como la que aparece en la figura 4.3 del libro "Learning from data", capítulo 4. Interpreta la gráfica respecto a las condiciones en las que se da el sobreajuste. (Nota: no es necesario la implementación).

El código desarrollado para realizar el experimento es:

```
ein <- function(w, x, y) {
    # calculamos Ein(w) como se indica en la página 91:
    # Ein(w) = 1/N sum_n=1^N (w^T z_n - y_n)^2
    errores = apply(X=x, MARGIN=1, FUN=function(dat) t(w)%*%dat)
    ein = mapply(FUN=function(e,yn) log(1+exp(-yn*e)), e=errores, yn=y)
    mean(ein)
}

calcula_errores <- function(x, yn, g, grado, N) {
    datos = cbind(1, poly(x, degree=grado))
    ein = ein(w=g, x=datos, y=yn)
    eout = cota_eout(e=ein, dvc=grado+1, N=N)
    c(eout,ein)
}

experimento <- function() {
    pesos = ej_2_1(Qf=15, N=50, sigma=1)
    errores_g2 = calcula_errores(x=pesos[[2]], yn=pesos[[3]], grado=2,
        N=50, g=as.vector(pesos[[4]]$'coefficients'))
    errores_g10 = calcula_errores(x=pesos[[2]], yn=pesos[[3]], grado=10,
        N=50, g=as.vector(pesos[[5]]$'coefficients'))
    list(errores_g2,errores_g10)
}

ej_2_2 <- function() {
    eg2 = 0
    eg10 = 0
    eing2 = 0
    eing10 = 0
    for (i in 1:100) {
        errores = experimento()
        eg2 = eg2 + errores[[1]][1]
        eg10 = eg10 + errores[[2]][1]
```

```
        eing2 = eing2 + errores[[1]][2]
        eing10 = eing10 + errores[[2]][2]
    }
    eg2 = eg2/100
    eg10 = eg10/100
    eing2 = eing2/100
    eing10 = eing10/100
    cat("Eout(g2) = ", eg2, "\n")
    cat("Eout(g10) = ", eg10, "\n")
    cat("Ein(g2) = ",eing2,"\n")
    cat("Ein(g10) = ",eing10,"\n")
    eg10 - eg2
}
```

En este código, uso el error dentro de la muestra explicado en la página 91 del libro "Learning from Data: a short course":

$$E_{in}(w) = \frac{1}{N}\sum_{n=1}^{N} ln\left(1 + e^{-y_n w^T x_n}\right)$$

Este error dentro de la muestra, lo uso después para calcular E_{out} con el error de entropía cruzada:

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

Los resultados obtenidos son:

```
## Eout(g2) =  2.330956
## Eout(g10) =  3.32677
## Ein(g2) =  0.6246127
## Ein(g10) =  0.3592421
```

```
## [1] 0.9958147
```

Si estamos haciendo sobreajuste de los datos, $E_{out}(\mathcal{H}_{10})$ será mayor que $E_{out}(\mathcal{H}_2)$ y por tanto, la resta de ambos será positiva. Sin embargo, si la resta de ambos es negativa, $E_{out}(\mathcal{H}_{10})$ será menor que $E_{out}(\mathcal{H}_2)$ y así, estaremos haciendo un ajuste mejor de los datos, sin sobreajuste. Si la resta de ambos es 0, significa que los dos coeficientes son iguales. Por tanto, esta medida nos sirve para saber si realmente estamos haciendo sobreajuste o hemos dado con un buen modelo para aproximar la función objetivo.

Teniendo esto en cuenta, podemos concluir que el g_{10} calculado sobreajusta los datos, ya que obtenemos una medida positiva. Como se ve, $E_{in}(g10)$ es menor al $E_{in}(g2)$. Sin embargo, obtenemos un resultado contrario al salir de la muestra. Se dan todas las condiciones descritas en la teoría para que se dé el sobreajuste.

En la figura 4.3, vemos dos gráficas. En la primera, se compara el nivel de ruido con el tamaño de la muestra. Este tipo de ruido se denomina **ruido estocástico**. Cuando tenemos muy pocos datos, el sobreajuste está práticamente garantizado, aunque haya poco ruido. A medida que crece el tamaño de la muestra, con un ruido pequeño, disminuye el sobreajuste. Ahora bien, conforme crece el ruido, crece el sobreajuste. Cuando tenemos muchos datos, tendremos algo de sobreajuste en nuestro modelo, a pesar de tener mucho ruido. En la siguiente tabla vemos un resumen de lo explicado en este párrafo:

$\sigma \setminus N$	0	100	150
0	↑	↘	↓
1	↑	↗	↓
2.25	↑	↑	↘

En la segunda gráfica, se compara la complejidad de la función objetivo con el tamaño de la muestra. Este tipo de ruido se denomina **ruido determinístico**. En este caso, si la función objetivo tiene una complejidad baja no tendremos sobreajuste, aunque tengamos pocos datos. A medida que crece la complejidad de la función objetivo, tendremos más sobreajuste en nuestro modelo. Si tenemos un tamaño de muestra grande, tendremos poco sobreajuste a pesar de de la función objetivo tenga mucha complejidad. La siguiente tabla muestra un resumen de este párrafo:

$Qf \setminus N$	0	100	150
10	↘	↓	↓
50	↑	↘	↓
100	↑	↗	↓

En ambos casos vemos que al aumentar el tamaño de la muestra, se reduce el sobreajuste.

3 Regularización y selección de modelos

3.1 Regularización "weight decay"

Para $d = 3$ (dimensión) generar un conjunto de N datos aleatorios $\{x_n, y_n\}$ de la siguiente forma. Para cada punto, x_n generamos sus coordenadas muestreando de forma independiente una $\mathcal{N}(0, 1)$. De forma similar generamos un vector de pesos de $d+1$ dimensiones w_f, y el conjunto de valores $y_n = w_f^T x_n + \sigma\epsilon_n$, donde ϵ_n es el ruido que sigue también una $\mathcal{N}(0, 1)$ y σ^2 es la varianza del ruido; fijar $\sigma = 0, 5$.

Usar regresión lineal con regularización "weight decay" para estimar w_f con w_{reg}. Fijar el parámetro de regularización a $0, 05/N$.

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

b) Repetir el experimento 1000 veces, anotando el promedio y la varianza de e_1, e_2 y E_{cv} en todos los experimentos.

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

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

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

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

g) Si se incrementa la cantidad de regularización, ¿debería N_{eff} subir o bajar? Argumentar la respuesta. Ejecutar el mismo experimento con $\lambda = 25/N$ y comparar los resultados del punto anterior para verificar la conjetura.

a) Para el apartado a he hecho el siguiente código:

```
weight_decay <- function(datos, y, lambda){
    as.vector(solve(t(datos)%*%datos+lambda*diag(ncol(datos)))%*%t(datos)%*%y)
}

cross_validation_n <- function(d=2, sigma=0.5, lambda=0.05) {
    # generamos los tamaños de N en los que haremos el experimento
    tams = seq(from=15, to=115, by=10)
    lista_ecv = vector() # vector para guardar todos los errores de validación cruzada
    e1 = vector() # vector para guardar todos los e1 obtenidos
    e2 = vector() # vector para guardar todos los e2 obtenidos
    wf = as.vector(simula_gaus(N=1, dim=d+1, sigma=c(1))) # los pesos
    for (i in tams) {
        n = i+d # generamos el tamaño de los datos en esta iteración
        lambda = lambda/n
        x = simula_gaus(N=n, dim=d, sigma=c(1)) # generamos los datos
        epsilon = as.vector(simula_gaus(N=n, dim=1, sigma=c(1))) # el ruido
        # y por último, las etiquetas
        x_aux = cbind(rep(1,n), x)
        fx = apply(X=x_aux, MARGIN=1, FUN=function(fila) wf%*%fila)
        yn = mapply(FUN=function(f, e) f + e*sigma, f=fx, e=epsilon)
        # definimos el error en regresión.
        squared_error <- function(w, x, y) {
            hi = w%*%x
            (hi-y)*(hi-y)
        }
        # y calculamos el error para cada punto
        e = sapply(X=1:n, FUN=function(i) squared_error(w=weight_decay(datos=x_aux[-i,],
            y=yn[-i], lambda=lambda), x=x_aux[i,], y=yn[i]))
        lista_ecv = append(x=lista_ecv, values=mean(e))
        e1 = append(x=e1, values=e[1])
        e2 = append(x=e2, values=e[2])
    }
    list(lista_ecv, e1, e2)
}
```

En él, genero un vector con todos los tamaños que se especifican en el enunciado y, con dichos tamaños, repito el experimento de generar datos, calcular etiquetas, calcular el modelo y el E_{cv}. Finalmente, devuelvo los E_{cv} calculados para cada N.

Para calcular w_{reg} uso la siguiente fórmula:

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

A la hora de realizar los experimentos, ajusto la función usando $N-1$ datos, y valido el modelo encontrado usando sólo 1. Repito esto N veces, usando cada vez un punto distinto para validar:

$$\mathcal{D}_n = (x_1, y_1), \cdots, (x_{n-1}, y_{n-1}), \cancel{(x_n, y_n)}, (x_{n+1}, y_{n+1}), \cdots, (x_N, y_N)$$

Así, obtenemos un buen modelo, ya que entrenamos con muchos datos y además, obtenemos una buena validación, ya que validamos el modelo N veces, o mejor dicho, con $K = N$ puntos.

Los errores E_{cv} obtenidos son:

```
##  [1] 0.1843964 0.4110072 0.1988736 0.2574779 0.2089674 0.2015252 0.3234056
##  [8] 0.2800179 0.2876109 0.2275862 0.2876048
```

b) Para realizar el experimento, he realizado el siguiente código:

```
experimento_3_1_b <- function(veces=1000) {
    resultados = matrix(ncol=6, nrow=veces)

    for (i in 1:veces) {
        l = cross_validation_n()
        resultados[i,] = c(mean(l[[1]]), var(l[[1]]), mean(l[[2]]), var(l[[2]]),
            mean(l[[3]]), var(l[[3]]))
    }

    colnames(resultados)=c("Media Ecv", "Var Ecv", "Media e1", "Var e1", "Media e2", "Var e2")
    resultados
}
```

Por simplicidad, en el PDF realizo 5 experimentos en lugar de 1000, y obtengo los siguientes resultados:

```
##      Media Ecv     Var Ecv  Media e1      Var e1  Media e2     Var e2
## [1,] 0.2329453 0.002109421 0.1379033 0.009238904 0.1069829 0.02196777
## [2,] 0.2771993 0.002262608 0.1769218 0.017543663 0.1391508 0.05246897
## [3,] 0.2696571 0.003741661 0.4045670 0.185131327 0.1197111 0.03936558
## [4,] 0.2886625 0.003788674 0.1879916 0.054178329 0.4318992 0.10189532
## [5,] 0.3000702 0.010992608 0.2847102 0.035413233 0.4502807 0.23951068
```

c) Al ser E_{cv} la media de todos los e_n obtenidos, tanto el promedio de e_1 como el de e_2 y como el de cada e_n debe de ser muy similar al de E_{cv}. Como se ve en los experimentos, los promedios de e_1 y e_2 obtenidos son muy parecidos a los E_{cv}.

d) En cada iteración del experimento, generamos conjuntos de datos con $d + N$ donde $N \in \{15 + d, 25 + d, \cdots, 115 + d\}$. Cada conjunto de datos da lugar a un w_{reg} diferente y, por tanto, a un e_1 diferente. Por tanto, la variación de e_1 (y de todos los e_n) viene dada por el D_{train} generado y por los pesos w_{reg}.

e) E_{cv} es un error de validación hecho con N puntos mientras que e_1 es un error de validación hecho con sólo uno. Por tanto, la varianza de e_1 debe ser mucho mayor, ya que nos da una estimación peor del error fuera de la muestra mientras que la varianza de E_{cv} debe de ser muy pequeña, ya que nos da una buena estimación del error fuera de la muestra al usar N puntos independientes para validar en vez de uno sólo.

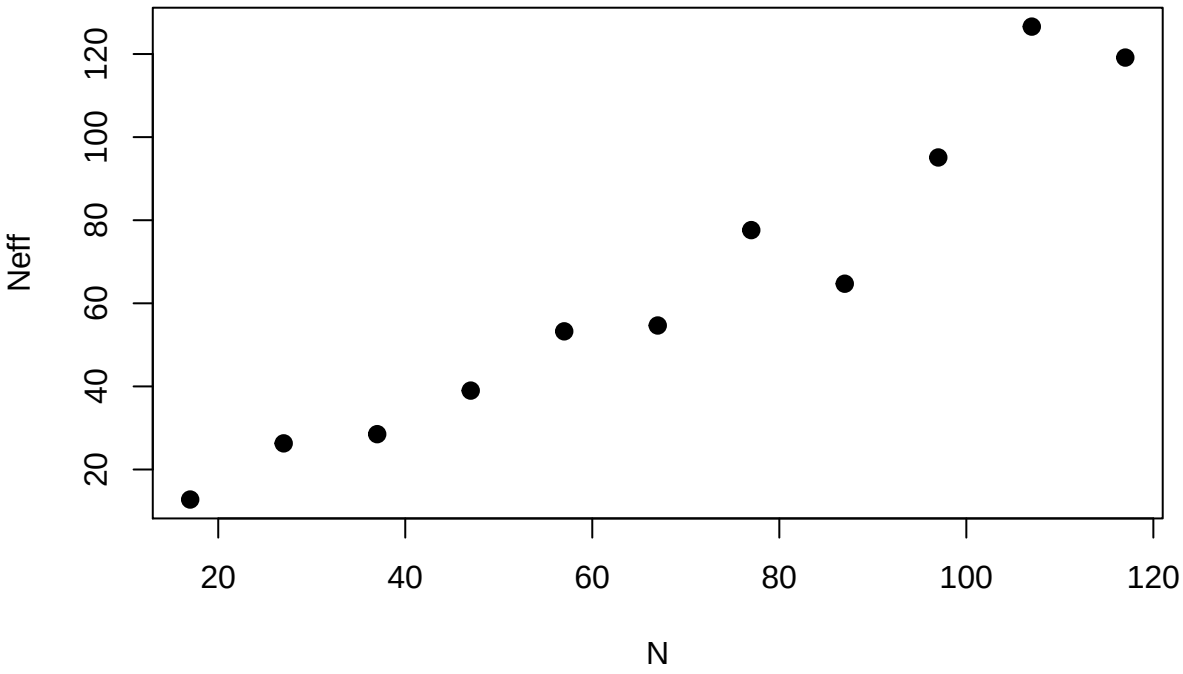
f) Para el apartado f he realizado el siguiente código:

```
nuevos_ejemplos <- function(veces=1000, d=2, lambda=0.05) {
    # ejecutamos el apartado a varias veces para obtener listas completas de Ecv y e1
    # para ello: creamos una matriz para guardar los distintos valores de Ecv y e1
    # para cada N usado.
    valores_ecv = matrix(ncol=11, nrow=veces)
    valores_e1 = matrix(ncol=11, nrow=veces)
```

```
    for (i in 1:veces) {
        listas = cross_validation_n(lambda=lambda)
        valores_ecv[i,] = as.vector(listas[[1]])
        valores_e1[i,] = as.vector(listas[[2]])
    }
    # una vez obtenidas las listas de valores de e1 y Ecv calculamos la varianza
    var_ecv = apply(X=valores_ecv, MARGIN=2, FUN=function(col) var(col))
    var_e1 = apply(X=valores_e1, MARGIN=2, FUN=function(col) var(col))
    # hacemos el cociente de los elementos de ambos vectores
    var_cociente = var_e1/var_ecv
    # representamos en una gráfica en el eje X los N y en el y, los cocientes:
    tams = seq(from=15+d, to=115+d, by=10)
    plot(x=tams, y=var_cociente, lwd=2, pch=19, ylab="Neff", xlab="N")
    (var_cociente/tams)*100
}
```

En ella, ejecuto la validación cruzada 1000 veces, y calculo la varianza de E_{cv} y e_1 para cada N.

Los N_{eff} en tanto por ciento respecto de N son:

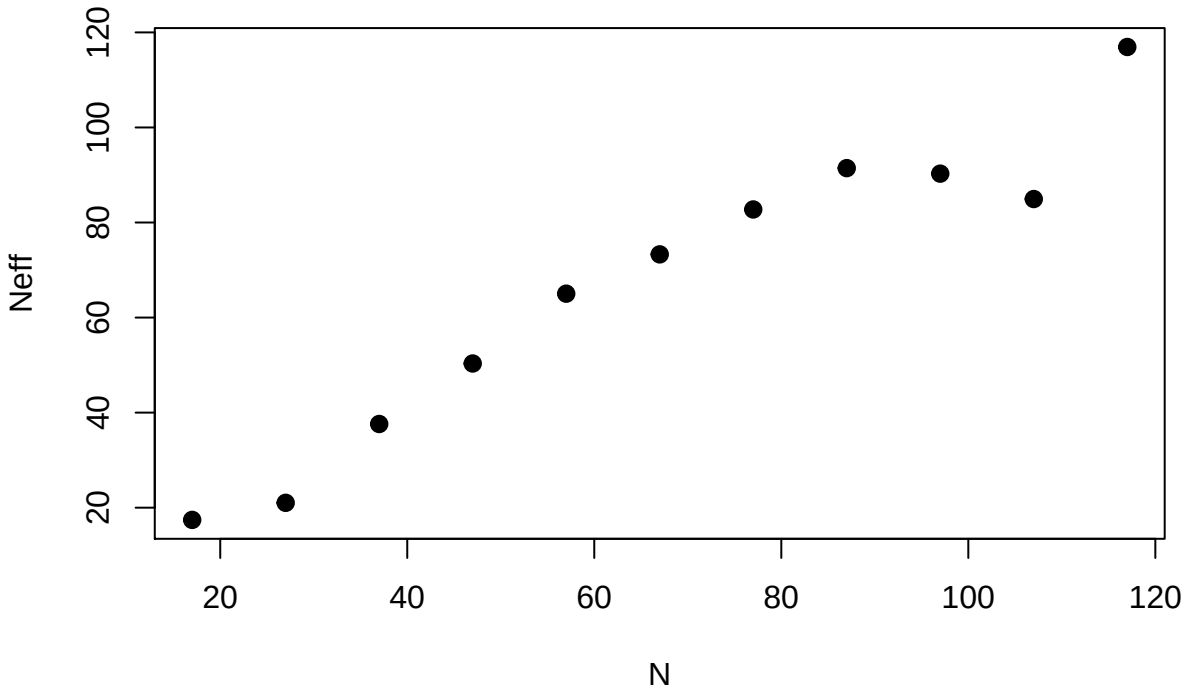



```
##  [1]  75.16241  97.37025  77.04426  82.97023  93.43415  81.58889 100.80326
##  [8]  74.38803  98.04780 118.31622 101.83752
```

Tal y como se esperaba, obtenemos un N_{eff} cercano a N, siempre superior al 70 % del valor de N.

g) Al aumentar la regularización, la varianza decrece (tanto para e_1 como para E_{cv}) y, por tanto, el cociente de ambas varianzas, es decir, N_{eff} permanecerá igual, debido a que la proporción será parecida. En algunos casos, N_{eff} crecerá y en otros, bajará.

Con $\lambda = 25/N$ obtenemos los siguientes resultados:




```
##  [1] 102.52830  77.85089 101.62810 107.09400 114.07800 109.39834 107.46129
##  [8] 105.10751  93.07245  79.38127  99.94033
```

Tal y como se esperaba, los N_{eff} obtenidos han permaneceido parecidos a los del apartado anterior.