Trabajo 2

Alejandro García Montoro

7 de abril de 2016

Modelos lineales

Ejercicio 1 - Gradiente descendente

Apartado 1.a

Apartado 1.a.1

La función $E(u,v) = (ue^v - 2ve^{-u})^2$ tiene las siguientes derivadas parciales:

$$\frac{\delta}{\delta u}E(u,v) = 2(ue^v - 2ve^{-u})(e^v + 2ve^{-u})$$
$$\frac{\delta}{\delta v}E(u,v) = 2(ue^v - 2ve^{-u})(ue^v - 2e^{-u})$$

luego su gradiente es

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

Apartado 1.a.2

Para implementar el algoritmo del gradiente descendente necesitamos definir la función E y su gradiente, ∇E:

```
# Devuelve el valor de E en (u,v) = `punto`
E <- function(punto){
  # Tomamos las componentes del punto
  u <- punto[1]
  v <- punto[2]

  # Devolvemos el valor de E
  return((u*exp(v) - 2*v*exp(-u))^2)
}


# Devuelve el valor del gradiente de E en (u,v) = `punto`


E.gradiente <- function(punto){
  # Tomamos las componentes del punto
  u <- punto[1]
  v <- punto[2]

  # Calculamos el coeficiente común
  coeff <- 2*(u*exp(v) - 2*v*exp(-u))
```

```
  # Devolvemos el valor del gradiente de E
  return(coeff * c(exp(v) + 2*v*exp(-u), u*exp(v) - 2*exp(-u)))
}
```

Definimos ahora ya la función del gradiente descendente:

```
# Ejecuta el método del gradiente descendente para encontrar el mínimo
# de la función f.
gradienteDescendente <- function(f, f.gradiente, w_0 = c(1,1),
                                 eta = 0.1, tol = 1e-14, maxIter = 50){
  # Inicializamos solución y vectores de puntos/valores intermedios
  w <- w_0
  variables.x <- w[1]
  variables.y <- w[2]
  valores <- f(w)

  # Criterio de parada: el valor de f es menor que la tolerancia
  # o se ha alcanzado el máximo de iteraciones
  while(f(w) >= tol && length(valores) < maxIter){
    # Calculamos la dirección de menor descenso
    direccion <- -f.gradiente(w)

    # Avanzamos en esa dirección
    w <- w + eta*direccion

    # Actualizamos el valor de los vectores de puntos/valores intermedios
    variables.x <- c(variables.x, w[1])
    variables.y <- c(variables.y, w[2])
    valores <- c(valores, f(w))
  }

  # Devolvemos el punto encontrado, el número de iteraciones
  # empleadas en encontrarlo y un data frame con los puntos intermedios
  return(list("Pesos"=w, "Iter"=length(valores)-1,
              "Res"=data.frame(X = variables.x, Y = variables.y, Value = valores)))
}
```

Ejecutamos la función anteriormente implementada:

```
# Ejecución del gradiente descendente
resultado.E <- gradienteDescendente(E, E.gradiente)
```

Comprobamos así que el algoritmo ha ejecutado 10 iteraciones hasta obtener un valor de $E(u, v)$ inferior a 10^{-14}.

Apartado 1.a.3

De la ejecución anterior podemos ver también que los valores de u y v obtenidos tras las 10 iteraciones son los siguientes:

$$u = 0.0447363$$
$$v = 0.0239587$$

Apartado 1.b

Sea ahora la función $f(x, y) = x^2 + 2y^2 + 2\sin(2\pi x)\sin(2\pi y)$, cuyo gradiente es:

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

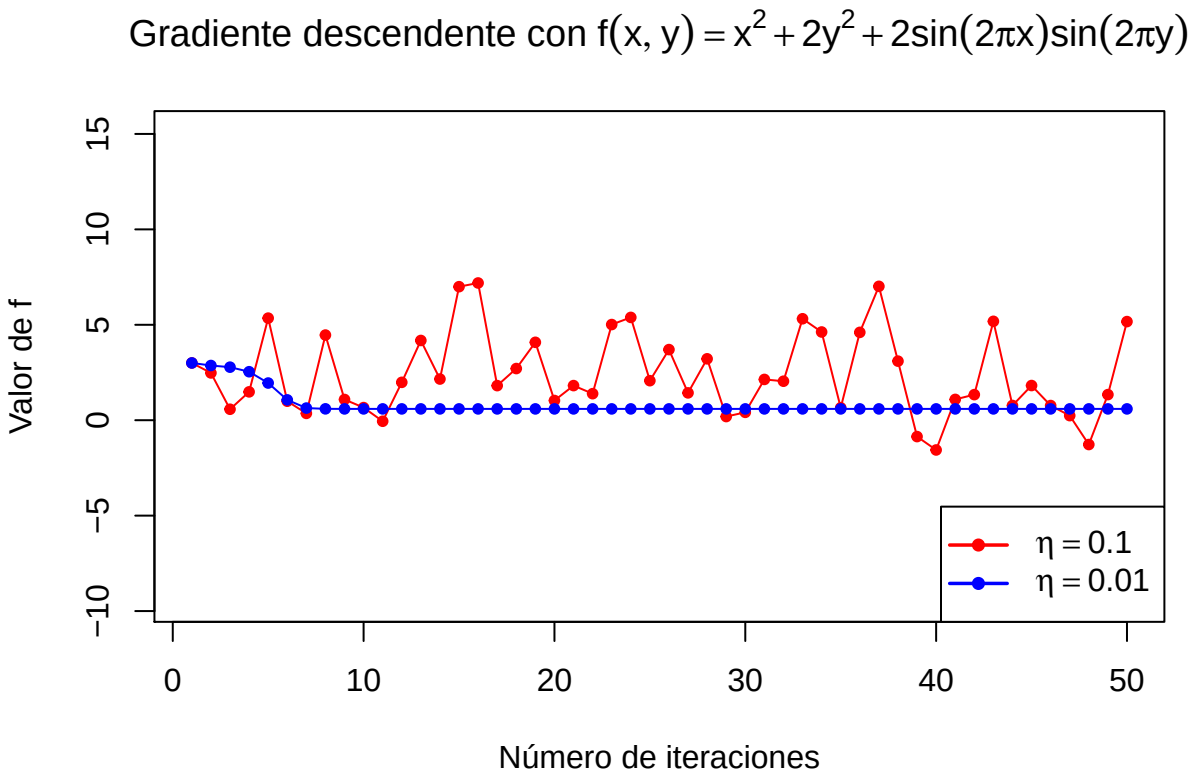
Apartado 1.b.1

Definimos las funciones f y ∇f en R:

```
# Devuelve el valor de f en (x,y) = `punto`
f <- function(punto){
  # Tomamos las componentes del punto
  x <- punto[1]
  y <- punto[2]

  # Devolvemos el valor de f
  return( x*x + 2*y*y + 2*sin(2*pi*x)*sin(2*pi*y) )
}

# Devuelve el valor del gradiente de f en (x,y) = `punto`
f.gradiente <- function(punto){
  # Tomamos las componentes del punto
  x <- punto[1]
  y <- punto[2]

  # Calculamos las componentes del gradiente de f
  f.grad.x <- 2*x + 4*pi*sin(2*pi*y)*cos(2*pi*x)
  f.grad.y <- 4*y + 4*pi*sin(2*pi*x)*cos(2*pi*y)

  # Devolvemos el valor del gradiente de f
  return(c(f.grad.x, f.grad.y))
}
```

Generamos ahora los datos y el gráfico solicitado con la función anteriormente implementada y valores de tasa de aprendizaje $\eta = 0.01$ y $\eta = 0.1$:

```
# Ejecutamos el experimento con ambos parámetros
resultado.f.0.01 <- gradienteDescendente(f, f.gradiente, eta = 0.01,
                                         w_0 = c(1,1), maxIter = 50, tol=-Inf)
resultado.f.0.1 <- gradienteDescendente(f, f.gradiente, eta = 0.1,
                                        w_0 = c(1,1), maxIter = 50, tol=-Inf)

# Generamos las gráficas de ambos descensos
plot(resultado.f.0.1$Res$Value, col = 'red', pch = 20, asp=1, type='o',
     main=expression(paste("Gradiente descendente con ",
                           f(x,y) == x^2 + 2*y^2 + 2*plain(sin)(2*pi*x)*plain(sin)(2*pi*y))),
     xlab="Número de iteraciones", ylab="Valor de f")
points(resultado.f.0.01$Res$Value, col = 'blue', pch = 20, asp=1, type='o')

# Añadimos leyenda de los datos dibujados.
legend("bottomright", c(expression(eta == 0.1), expression(eta == 0.01)),
       bg="white", col=c("red", "blue"), lty=1, pch=20, lwd=1.75)
```



Con la tasa de aprendizaje a 0.1, los cambios en la región de búsqueda del método son demasiado bruscos, y por eso el comportamiento es tan inestable: al tomar cada dirección con tanta velocidad, se explora más espacio de búsqueda y los descensos son muy dispares. Sin embargo, al bajar la tasa de aprendizaje a 0.01, el método encuentra rápidamente un mínimo local del que no puede salir, ya que las direcciones que va encontrando son siempre iguales y no es capaz de saltar del valle en el que se encuentra.

Apartado 1.b.2

Generamos los datos desde los datos iniciales indicados y tomamos los mínimos, tomando como criterio de parada únicamente el número máximo de iteraciones, que ponemos a 50:

```
# Definimos una función que, dado p, devuelve el valor mínimo del gradiente descendente
# con punto inicial = (p,p) y dónde se alcanza este:
minimoGradiente <- function(p){
  valores <- gradienteDescendente(f, f.gradiente, c(p, p), tol = -Inf, maxIter = 50)$Res
  unlist(valores[which.min(valores$Value),])
}

# Definimos los puntos iniciales sobre los que iterar con el apply siguiente
puntosIniciales <- c(0.1, 1, -0.5, -1)

# Usamos la funcion minimoGradiente sobre cada uno de los puntos anteriores
tablaMinimos <- lapply(puntosIniciales, minimoGradiente)

# Convertimos la lista devuelta en un data frame y ponemos nombres a columnas y filas
tablaMinimos <- data.frame(matrix(unlist(tablaMinimos), nrow=4, byrow=T))
colnames(tablaMinimos) <- c("X", "Y", "Valor")
rownames(tablaMinimos) <- c("Pto. inicial = (0.1, 0.1)", "Pto. inicial = (1, 1)",
                            "Pto. inicial = (-0.5, 0.5)", "Pto. inicial = (-1, -1)")
```

Podemos ahora crear una tabla con los datos obtenidos, en la que se especifica, para cada punto inicial, el punto (x, y) donde se alcanza el mínimo y el valor del mismo:

```
# Generamos la tabla solicitada
kable(tablaMinimos, digits=5)
```

	X	Y	Valor
Pto. inicial = (0.1, 0.1)	0.20646	-0.17155	-1.59489
Pto. inicial = (1, 1)	0.26879	-0.31392	-1.55870
Pto. inicial = (-0.5, 0.5)	0.28629	-0.33151	-1.39647
Pto. inicial = (-1, -1)	-0.26879	0.31392	-1.55870

Ejercicio 2 - Coordenada descendente

Redefinimos la función `gradienteDescendente` del primer apartado para realizar la técnica de *coordenada descendente*:

```
# Ejecuta el método de la coordenada descendente para encontrar el mínimo
# de la función f.
coordenadaDescendente <- function(f, f.grad, w_0 = c(1,1), eta = 0.1,
                                  tol = 1e-14, maxIter = 50){
  # Inicializamos solución y número de iteraciones
  w <- w_0
  iteraciones <- 0

  # Criterio de parada: el valor de f es menor que la tolerancia
  # o se ha alcanzado el máximo de iteraciones
  while(f(w) >= tol && iteraciones < maxIter){
    # Calculamos la dirección de menor descenso en la 1a coordenada
    direccion.x <- c(-f.grad(w)[1], 0)
    # Avanzamos en esa dirección
    w <- w + eta*direccion.x

    # Calculamos la dirección de menor descenso en la 2a coordenada
    direccion.y <- c(0, -f.grad(w)[2])
    # Avanzamos en esa dirección
    w <- w + eta*direccion.y

    # Aumentamos el contador de iteraciones
    iteraciones <- iteraciones + 1
  }

  # Devuelve la solución encontrada y el número de iteraciones
  return(list("Pesos"=w, "Iter"=iteraciones))
}
```

Apartado 2.a

Ejecutamos la función anteriormente implementada para ver el valor conseguido:

```
# Ejecutamos coordenada descendente
res.coordDesc <- coordenadaDescendente(E, E.gradiente, maxIter = 15)
```

De la ejecución anterior podemos ver que los valores de u y v obtenidos tras las 15 iteraciones son los siguientes:

$$u = 6.2970759$$
$$v = -2.852307$$

El valor alcanzado es además $f(6.2970759, -2.852307) = 0.1398138$.

Apartado 2.b

Como primera aproximación a la comparación, vamos a ejecutar ambos métodos sobre las funciones f y E con un máximo de 20 iteraciones cada uno, con el parámetro `tol` puesto a un valor $-\infty$ para asegurarnos de que las iteraciones se realizan:

```
# Tomamos los resultados para f
comp.f.grad <- gradienteDescendente(f, f.gradiente, maxIter = 20, tol=-Inf)
comp.f.coor <- coordenadaDescendente(f, f.gradiente, maxIter = 20, tol=-Inf)

# Tomamos los resultados para E
comp.E.grad <- gradienteDescendente(E, E.gradiente, maxIter = 20, tol=-Inf)
comp.E.coor <- coordenadaDescendente(E, E.gradiente, maxIter = 20, tol=-Inf)

# Obtenemos el valor de f en el mínimo devuelto por ambos algoritmos
min.f.grad <- f(comp.f.grad$Pesos)
min.f.coor <- f(comp.f.coor$Pesos)

# Obtenemos el valor de E en el mínimo devuelto por ambos algoritmos
min.E.grad <- E(comp.E.grad$Pesos)
min.E.coor <- E(comp.E.coor$Pesos)
```

Podemos hacer una tabla con la que poder comparar estos mínimos:

```
# Generamos un data frame con los datos obtenidos
tablaComp <- data.frame(c(min.f.coor, min.E.coor),
                        c(min.f.grad, min.E.grad))

colnames(tablaComp) <- c("Valor de f", "Valor de E")
rownames(tablaComp) <- c("Coordenada descendente", "Gradiente descendente")
```

```
# Generamos la tabla a partir del data frame
kable(tablaComp, digits=15)
```

	Valor de f	Valor de E
Coordenada descendente	2.5500174	1.033149
Gradiente descendente	0.1103108	0.000000

Ante estos datos parece que es claro que el método del gradiente descendente es mejor que el de coordenada descendente. Sin embargo, esta comparación no ha sido justa, ya que el método de coordenada está haciendo en realidad el doble de iteraciones que el de gradiente.

Así, para ser justos en la comparación, tenemos que ejecutar el método inicial de gradiente el doble de veces que el de la coordenada. Vamos a comparar entonces con 20 y 10 iteraciones, respectivamente, ambos métodos con las mismas funciones anteriores

```
# Tomamos los resultados para f
comp.f.grad <- gradienteDescendente(f, f.gradiente, maxIter = 20, tol=-Inf)
comp.f.coor <- coordenadaDescendente(f, f.gradiente, maxIter = 10, tol=-Inf)

# Tomamos los resultados para E
comp.E.grad <- gradienteDescendente(E, E.gradiente, maxIter = 20, tol=-Inf)
comp.E.coor <- coordenadaDescendente(E, E.gradiente, maxIter = 10, tol=-Inf)

# Obtenemos el valor de f en el mínimo devuelto por ambos algoritmos
min.f.grad <- f(comp.f.grad$Pesos)
min.f.coor <- f(comp.f.coor$Pesos)

# Obtenemos el valor de E en el mínimo devuelto por ambos algoritmos
min.E.grad <- E(comp.E.grad$Pesos)
min.E.coor <- E(comp.E.coor$Pesos)
```

Rehacemos la tabla:

```
# Generamos un data frame con los datos obtenidos
tablaComp <- data.frame(c(min.f.coor, min.E.coor),
                        c(min.f.grad, min.E.grad))

colnames(tablaComp) <- c("Valor de f", "Valor de E")
rownames(tablaComp) <- c("Coordenada descendente", "Gradiente descendente")
```

```
# Generamos la tabla a partir del data frame
kable(tablaComp, digits=15)
```

	Valor de f	Valor de E
Coordenada descendente	7.8390825	1.033149
Gradiente descendente	0.1926057	0.000000

Como vemos, en el caso de E no ha habido cambio, pero en el caso de f sí: el gradiente, en este estudio, que es mucho más justo que el anterior, da unos valores muchísimo mejores, como era de esperar.

Ejercicio 3 - Método de Newton

El método de Newton es igual que el de gradiente descendente, pero esta vez la dirección de cambio viene dada por $H^{-1}\nabla f(w)$, donde H es la matriz Hessiana de f. Por tanto, basta adaptar la función anterior para que use esta nueva dirección. Además, vamos ahora a facilitar la llamada a la función: haremos uso del paquete `numDeriv`, que se debe instalar antes de ejecutar el siguiente trozo de código; con él, tenemos a disposición las funciones `grad(func, x)`, que devuelve el valor de la función `func` en el punto `x` y `hessian(func, x)` que, del mismo modo, devuelve el valor de la matriz hessiana de `func` en el punto `x`. Así, podemos obviar el

parámetro `f.grad` de los métodos anteriores y pasar sólo la función; el código se encargará de calcular los gradientes y hessianas necesarios.

Como además vamos a comparar el funcionamiento del método de Newton con lo analizado del gradiente descendente en el apartado 1.b, vamos a fijar como único criterio de parada el número de iteraciones —poniendo por defecto la tolerancia a $-\infty$, para estudiar qué pasa en cada momento:

```
# Ejecuta el método del gradiente descendente para encontrar el mínimo
# de la función f.
metodoNewton <- function(f, w_0 = c(1,1), eta = 0.1, tol = -Inf, maxIter = 50){
  # Inicializamos solución y vectores de puntos/valores intermedios
  w <- w_0
  variables.x <- w[1]
  variables.y <- w[2]
  valores <- f(w)

  # Criterio de parada: el valor de f es menor que la tolerancia
  # o se ha alcanzado el máximo de iteraciones
  while(f(w) >= tol && length(valores) < maxIter){
    # Calculamos la dirección con la hessiana
    direccion <- -solve(hessian(f, w)) %*% grad(f, w)

    # Avanzamos en esa dirección
    w <- w + eta*direccion

    # Actualizamos el valor de los vectores de puntos/valores intermedios
    variables.x <- c(variables.x, w[1])
    variables.y <- c(variables.y, w[2])
    valores <- c(valores, f(w))
  }

  # Devolvemos el punto encontrado, el número de iteraciones
  # empleadas en encontrarlo y un data frame con los puntos intermedios
  return(list("Pesos"=w, "Iter"=length(valores)-1,
              "Res"=data.frame(X = variables.x, Y = variables.y, Value = valores)))
}
```

Estudiamos ahora de nuevo la mejor solución encontrada por el Método de Newton tras 50 iteraciones con los puntos iniciales $(0.1, 0.1)$, $(1, 1)$, $(-0.5, -0.5)$ y $(-1, -1)$, creando una tabla comparativa en la que vemos el mínimo alcanzado y el punto donde se alcanza:

```
# Definimos una función que, dado p, devuelve el valor mínimo del método de Newton
# con punto inicial = (p,p) y dónde se alcanza este:
minimoGradiente <- function(p){
  valores <- metodoNewton(f, c(p, p))$Res
  unlist(valores[which.min(valores$Value),])
}

# Definimos los puntos iniciales
puntosIniciales <- c(0.1, 1, -0.5, -1)

# Usamos la funcion minimoGradiente sobre cada uno de los puntos anteriores
tablaMinimos.N <- lapply(puntosIniciales, minimoGradiente)
```

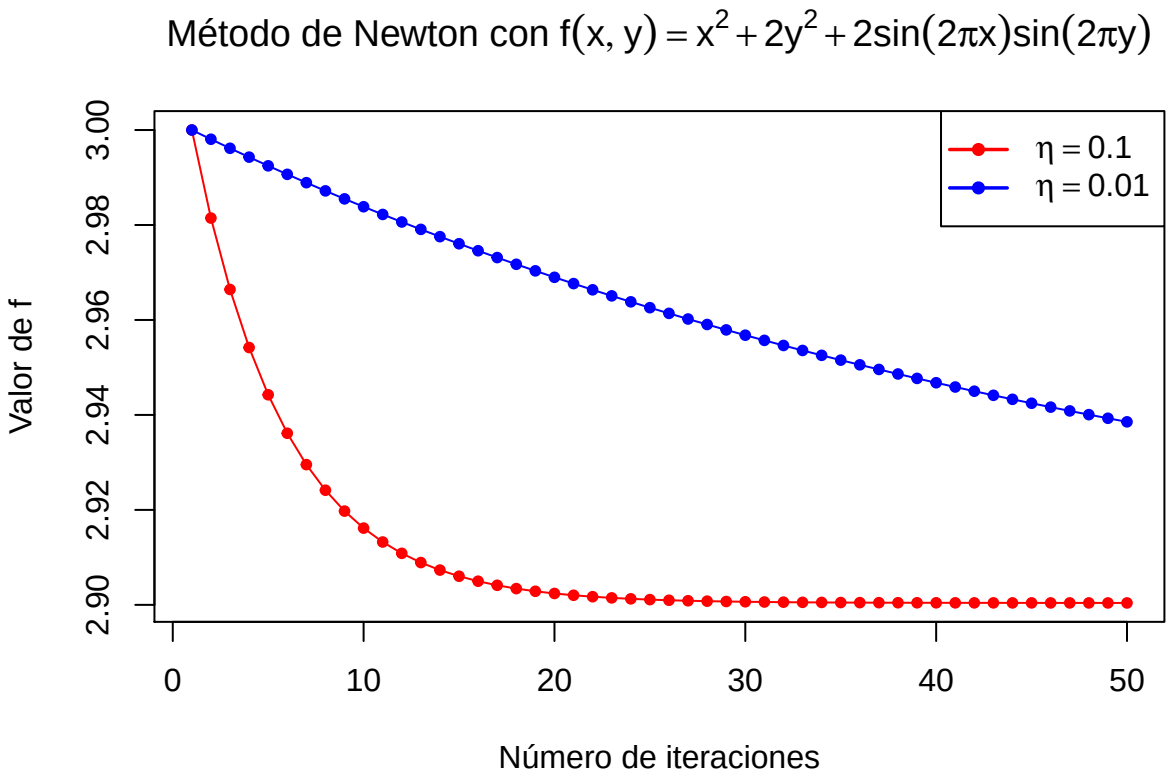
```r
# Convertimos la lista devuelta en un data frame y ponemos nombres a columnas y filas
tablaMinimos.N <- data.frame(matrix(unlist(tablaMinimos.N), nrow=4, byrow=T))
colnames(tablaMinimos.N) <- c("X", "Y", "Valor")
rownames(tablaMinimos.N) <- c("(0.1, 0.1)", "(1, 1)", "(-0.5, 0.5)", "(-1, -1)")
```

```r
# Generamos la tabla a partir del data frame
kable(tablaMinimos.N, digits=15)
```

	X	Y	Valor
(0.1, 0.1)	0.0004089448	0.0004045771	1.355796e-05
(1, 1)	0.9494396307	0.9747316958	2.900408e+00
(-0.5, 0.5)	-0.4752581069	-0.4878764660	7.254829e-01
(-1, -1)	-0.9494396307	-0.9747316958	2.900408e+00

Por último, del mismo modo en el que en el ejercicio 1.b estudiamos el comportamiento ante tasas de aprendizaje distintas, generamos ahora un gráfico parecido para determinar cómo se comporta el método de Newton:

```r
# Ejecutamos el experimento con valores de eta = 0.1, 0.01
newton01 <- unlist(metodoNewton(f, eta = 0.1)$Res$Value)
newton001 <- unlist(metodoNewton(f, eta = 0.01)$Res$Value)

# Generamos los dos plots
plot(newton01, col = 'red', pch = 20, type='o',
     main=expression(paste("Método de Newton con ",
                           f(x,y) == x^2 + 2*y^2 + 2*plain(sin)(2*pi*x)*plain(sin)(2*pi*y))),
     xlab="Número de iteraciones", ylab="Valor de f")
points(newton001, col = 'blue', pch = 20, type='o')

# Añadimos leyenda de los datos dibujados.
legend("topright", c(expression(eta == 0.1), expression(eta == 0.01)),
       bg="white", col=c("red", "blue"), lty=1, pch=20, lwd=1.75)
```

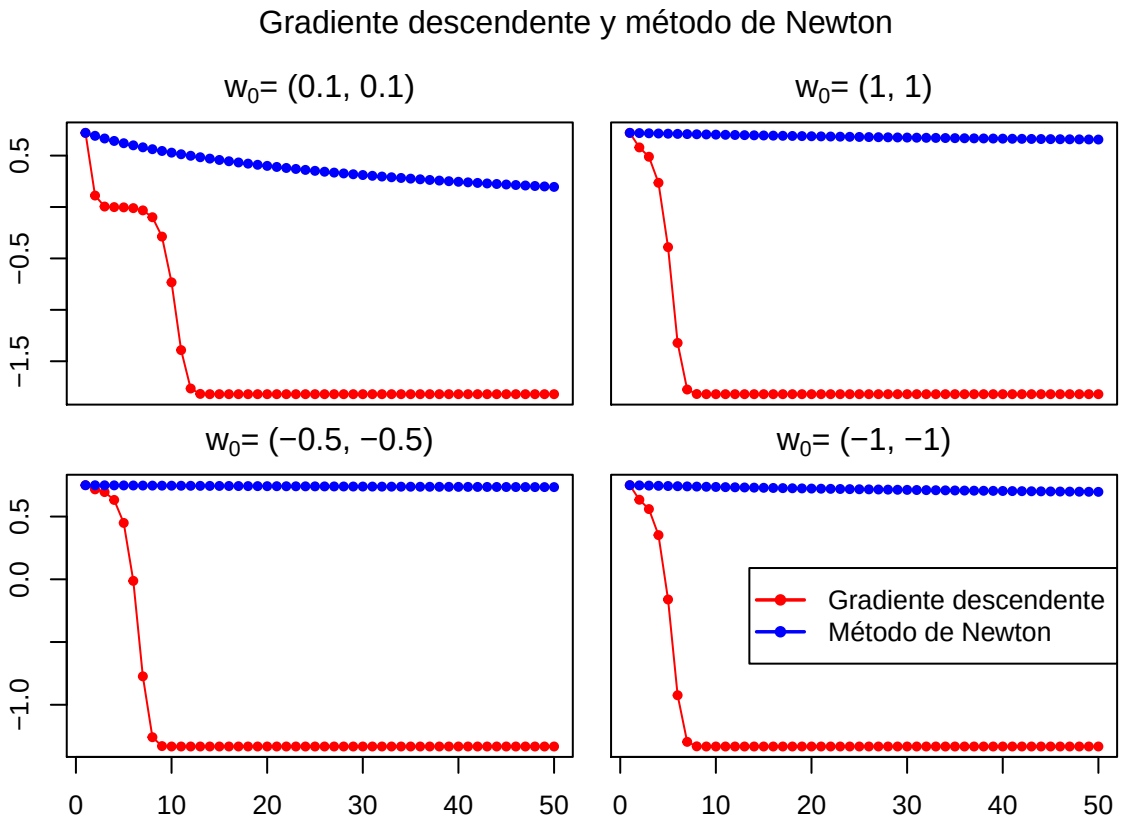


En este caso es claro que la tasa de aprendizaje más alta encuentra más rápido el óptimo local, ya que se mueve más rápido. En contraste con el método del gradiente descendente, el método de Newton, que tiene en cuenta hasta la segunda derivada de la función, se comporta de una forma mucho más suave aun con tasas de aprendizaje altas: de hecho, con $\eta = 0.1$ encuentra el mínimo local mucho antes y se estabiliza rápidamente.

Comparemos ahora ambos métodos, mostrando las curvas de descenso con cada punto inicial considerado y con una tasa de aprendizaje $\eta = 0.1$. Definimos para esto una función que recibe un punto inicial, una función y una tasa de aprendizaje, de manera que genere el gráfico correspondiente:

```
# Dibuja una gráfica comparando gradiente descendente y método de Newton sobre la función f
# con tasa de aprendizaje `tasa`
dibujarComparacion <- function(p, foo = f, tasa = 0.01, ...){
  # Generamos los resultados de ambos métodos
  res.grad <- unlist(gradienteDescendente(f, f.gradiente, w_0 = c(p,p), eta = tasa, tol = -Inf)$Res$Valu
  res.newt <- unlist(metodoNewton(f, w_0 = c(p,p), eta = tasa, tol = -Inf)$Res$Value)

  # Generamos el gráfico de ambos descensos
  plot(res.grad, pch = 20, type='o', col="red", main=bquote(paste(w[0],"= (",.(p),", ",.(p), ")")), ...
  points(res.newt, pch = 20, type='o', col="blue", ...)
}

# Definimos una rejilla 2x2 para los plots
prev_par <- par(no.readonly = T)

# Ajustamos los márgenes
par(mfrow=c(2, 2), mar=c(0.1, 1.1, 2.1, 0.1), oma=2.5*c(1,1,1,1))

# Generamos los cuatro gráficos
dibujarComparacion(0.1, xaxt="n")
dibujarComparacion(1, xaxt="n", yaxt="n")
```

```r
dibujarComparacion(-0.5)
dibujarComparacion(-1, yaxt="n")

# Ponemos título al gráfico conjunto
mtext("Gradiente descendente y método de Newton", outer=TRUE, line=0.5)

# Añadimos leyenda
legend("right", c("Gradiente descendente", "Método de Newton"),
       bg="white", col=c("red", "blue"), lty=1, pch=20, lwd=1.75)
```




```r
# Dejamos los parámetros como estaban anteriormente
par(prev_par)
```

Es claro que el gradiente descendente converge y consigue unos resultados muchísimo mejores que los del método de Newton con las mismas iteraciones. El último se queda estancado muy rápido y el primero consigue descender rápidamente para converger a un estado estable muy pronto, normalmente entre las 10 primeras iteraciones.

Ejercicio 4 - Regresión logística

Para este ejercicio vamos a reusar código de la práctica anterior:

```r
# Devuelve una lista de N vectores de dimensión dim cuyos valores están cogidos de
# una distribución uniforme en el intervalo `rango`
simula_unif <- function(N, dim, rango){
```

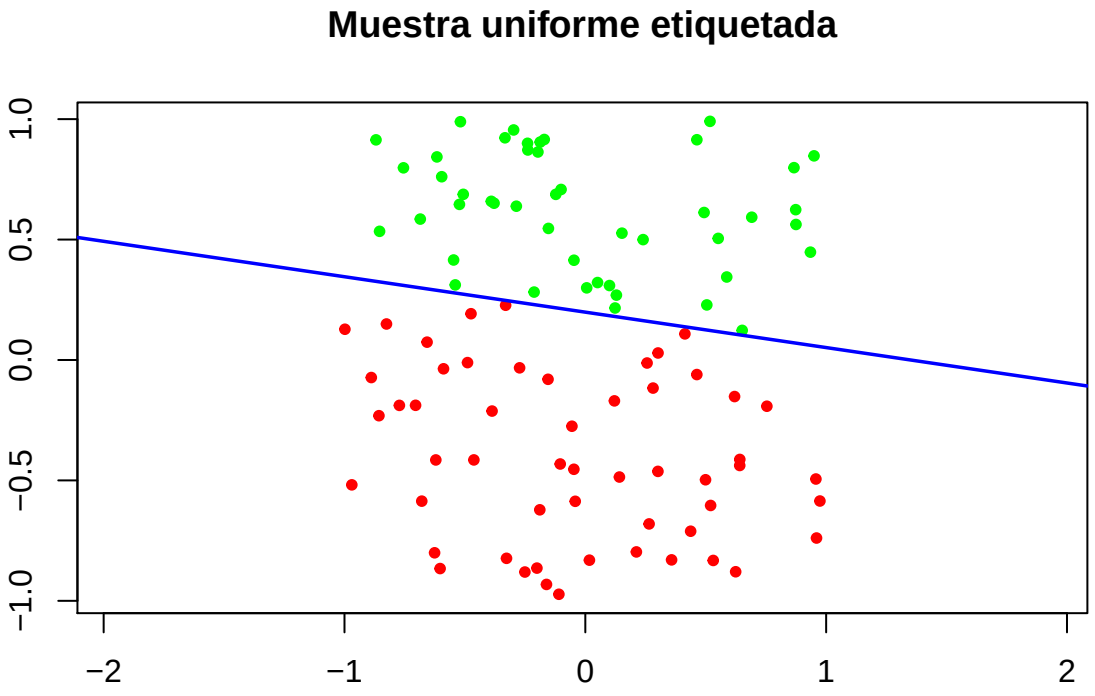
```
  lapply(rep(dim, N), runif, min = rango[1], max = rango[2])
}

# Devuelve los parámetros de una recta aleatoria que cruza el cuadrado
# intervalo x intervalo
simula_recta <- function(intervalo){
  # Simulamos dos puntos dentro del cuadrado intervalo x intervalo
  punto1 <- runif(2, min=intervalo[1], max=intervalo[2])
  punto2 <- runif(2, min=intervalo[1], max=intervalo[2])

  # Generamos los parámetros que definen la recta
  a <- (punto2[2] - punto1[2]) / (punto2[1] - punto1[1])
  b <- -a * punto1[1] + punto1[2]

  # Devolvemos un vector concatenando ambos parámetros
  c(a,b)
}

# Devuelve una función etiquetadora basada en f
generador_etiquetados <- function(f){
  function(x,y){
    sign(f(x,y))
  }
}
```

Definimos la función f y el conjunto $\mathcal{D}$:

```
# Generamos la frontera que define f
regLog.frontera <- simula_recta(c(-1,1))

# Generamos los 100 datos aleatorios
datos <- simula_unif(100, 2, c(-1,1))
regLog.datos <- matrix(unlist(datos), ncol = 2, byrow = T)

# Generamos la función f objetivo
regLog.f <- function(x,y){
  y - regLog.frontera[1]*x - regLog.frontera[2]
}
```

Echémosle un vistazo a nuestros datos y a sus etiquetas

```
# Generamos la función etiquetadora y generamos las etiquetas
regLog.etiquetado <- generador_etiquetados(regLog.f)
regLog.etiquetas <- regLog.etiquetado(regLog.datos[,1], regLog.datos[,2])

# Generamos un vector de colores basado en las etiquetas
colores <- ifelse(regLog.etiquetas == 1, "green", "red")

# Generamos la gráfica
plot(regLog.datos, asp = 1, col = colores, pch = 20,
     main="Muestra uniforme etiquetada", xlab="", ylab="")
abline(rev(regLog.frontera), col="blue", lwd=1.75)
```



Apartado 4.a

Definimos una función que implementa regresión logística con gradiente descendente estocástico:

```
# Devuelve la distancia euclídea entre los puntos x e y
distance <- function(x,y){
  return(sqrt(sum((x-y)^2)))
}

# Ejecuta regresión logística con gradiente descendente estocástica sobre los datos
# (x_n, y_n) = (datos, etiquetas), cno punto inicial w_0, tasa de aprendizaje
# eta y criterio de parada basado en mejoras mayores de 0.01
RL.SGD <- function(datos, etiquetas, w_0 = c(0,0), eta = 0.01, tol= 0.01){
  # Inicializamos contador de iteraciones
  iteraciones <- 0

  # Definimos la solución actual como el vector inicial. Si tiene
  # una posición más que el número de datos, lo dejamos como está;
  # si no, añadimos un 0.
  w.curr <- ifelse(length(w_0) == ncol(datos) + 1, w_0, c(w_0, 0))

  # Añadimos una columna de unos a los datos para manejar el término independiente
  datos <- cbind(datos, 1)

  # Criterio de parada: que la distancia entre la solución actual y la anterior
  # sea menor que la tolerancia. Hacemos el OR lógico con iteraciones == 0 al
  # principio para entrar la primera vez
  while(iteraciones == 0 || distance(w.prev, w.curr) > tol){
    # Aumentamos el contador de iteraciones
    iteraciones <- iteraciones+1
```

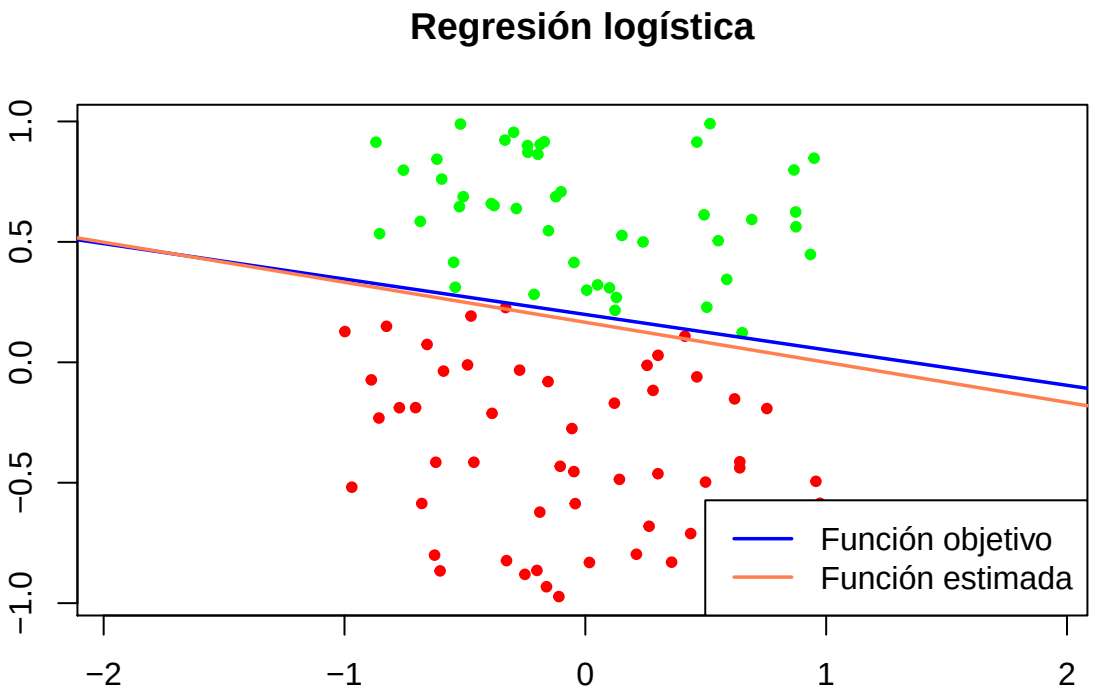
```
    # La solución anterior es la actual del último bucle
    w.prev <- w.curr

    # Comportamiento estocástico: tomamos una permutación aleatoria de los datos
    indices <- sample(length(etiquetas))

    # Bucle sobre toda la muestra
    for(index in indices){
      # Tomamos el dato y la etiqueta
      dato <- datos[index,]
      etiq <- etiquetas[index]

      # Calculamos la dirección de descenso con la función logística
      g_t <- -(etiq*dato)/(1+exp(etiq*w.curr%*%dato))

      # Nos movemos en esa dirección
      w.curr <- w.curr - eta*g_t
    }
  }

  # Definimos la recta obtenida
  recta <- -c(w.curr[1], w.curr[3]) / w.curr[2]

  # Definimos la función estimada a partir del resultado obtenido
  g <- function(x, y){
    return(y - recta[1]*x - recta[2])
  }


  # Devolvemos la solución (los pesos), la recta generada, la función estimada y el número de iteracion
  return(list("Pesos"=w.curr, "Recta" = recta,
              "g"=g, "Iter"=iteraciones))
}
```

Apartado 4.b

Si hacemos la regresión logística sobre los datos generados anteriormente, podemos ver cómo de bien se ajusta la g estimada a la f objetivo:

```
# Hacemos regresión logística sobre los datos generados anteriormente
regLog.res <- RL.SGD(regLog.datos, regLog.etiquetas)

# Dibujamos la muestra etiquetada
plot(regLog.datos, asp = 1, col = colores, pch = 20,
     main="Regresión logística", xlab="", ylab="")

# Añadimos al gráfico la función objetivo
abline(rev(regLog.frontera), col="blue", lwd=1.75)
# Añadimos al gráfico la función estimada
abline(rev(regLog.res$Recta), col="coral", lwd=1.75)

# Añadimos leyenda
```

```
legend("bottomright", c("Función objetivo", "Función estimada"),
       bg="white", col=c("blue", "coral"), lty=1, lwd=1.75)
```



Estimamos ahora E_{out} midiendo E_{test} sobre 100 muestrasde test aleatorias de tamaño 1000, contando el número de puntos mal etiquetados y tomando la media de todos ellos.

```
# Definimos una función para generar muestras de tamaño tam en el
# cuadrado intervalo X intervalo
generar_muestra <- function(tam = 100, intervalo = c(-1,1)){
  # Generamos la muestra uniforme
  datos.muestra <- simula_unif(N = tam, dim = 2, rango = intervalo)

  # Devolvemos una matriz con la coordenada x en la primera columna
  # y la y en la segunda
  return(matrix(unlist(datos.muestra), ncol = 2, byrow = T))
}

# Definimos función etiquetadora basada en la función objetivo
regLog.g.etiquetado <- generador_etiquetados(regLog.res$g)

# Definimos una función para medir el error en una muestra de test
medirEtest <- function(etiquetadoObjetivo, N = 100){
  # Generamos una nueva muestra de test de tamaño 100
  regLog.test <- generar_muestra(100)

  # Generamos etiquetado con f de la nueva muestra
  regLog.test.etiq <- etiquetadoObjetivo(regLog.test[, 1], regLog.test[, 2])

  # Generamos etiquetado con g de la nueva muestra
  regLog.g.etiquetas <- regLog.g.etiquetado(regLog.test[, 1], regLog.test[, 2])
```

```
  # Devolvemos el porcentaje de muestras mal etiquetadas
  return(sum(regLog.g.etiquetas != regLog.test.etiq) / nrow(regLog.test))
}

# Realizamos el conteo sobre 100 muestras de test y tomamos la media
E_out <- mean(replicate(100, medirEtest(regLog.etiquetado)))
```

Tenemos así una estimación del error fuera de la muestra $E_{out}(g) = 1.45\%$.

Apartado 4.c

```
experimentoEout <- function(N = 100){
  ################# GENERACIÓN DE DATOS #################
  #######################################################

  # Generamos la frontera que define f
  frontera <- simula_recta(c(-1,1))

  # Generamos los N datos aleatorios
  datos <- simula_unif(N, 2, c(-1,1))
  datos <- matrix(unlist(datos), ncol = 2, byrow = T)

  # Generamos la función f objetivo
  f <- function(x,y){
    y - frontera[1]*x - frontera[2]
  }

  # Generamos la función etiquetadora y generamos las etiquetas
  etiquetado <- generador_etiquetados(f)
  etiquetas <- regLog.etiquetado(datos[,1], datos[,2])

  ################### REGRESIÓN LINEAL ##################
  #######################################################

  # Hacemos regresión logística sobre los datos generados anteriormente
  res <- RL.SGD(datos, etiquetas)

  ################### MEDICIÓN DEL ERROR ################
  #######################################################

  # Realizamos el conteo sobre 100 muestras de test y tomamos la media
  E_out <- mean(replicate(100, medirEtest(etiquetado)))

  return(list("Error" = E_out, "Iteraciones" = res$Iter))
}

# Repetimos el experimento del error fuera de la muestra 100 veces
resExp <- replicate(100, experimentoEout())

# Tomamos los valores de E_out medios y el número medio de iteraciones
E_out <- mean(unlist(resExp["Error",]))
epoch <- mean(unlist(resExp["Iteraciones",]))
```

Vemos que el error medio tras 100 repeticiones con funciones objetivo distintas y estimando el error fuera de la muestra con 100 muestras de test de tamaño 100 es de $E_{out} = 32.6778\%$. El número medio de iteraciones necesarias con el criterio de parada indicado ha sido de 332.6.

Ejercicio 5 - Clasificación de dígitos

Cargamos los datos del fichero con `read.table()`. Así, tenemos en `digitos` una matriz cuyas filas son las instancias de los datos de entrenamiento. Cada fila tiene en su primera posición el dígito del que se trata, así que filtramos con ese dato y nos quedamos con los unos y los cincos:

```
# Leemos el fichero de entrenamiento
digitos.train <- read.table("datos/zip.train")

# Leemos el fichero de test
digitos.test <- read.table("datos/zip.test")

# Nos quedamos únicamente con los números 1 y 5
digitos.train <- digitos.train[ is.element(digitos.train[,1], c(1,5)), ]
digitos.test <- digitos.test[ is.element(digitos.test[,1], c(1,5)), ]
```

Para el cálculo de la intensidad media y de la simetría se ha usado la función de la anterior práctica, que recibe una instancia de los digitos como parámetro y calcula la intensidad media de los valores y la simetría, obteniendo la diferencia entre la matriz original y la que tiene sus columnas invertidas:

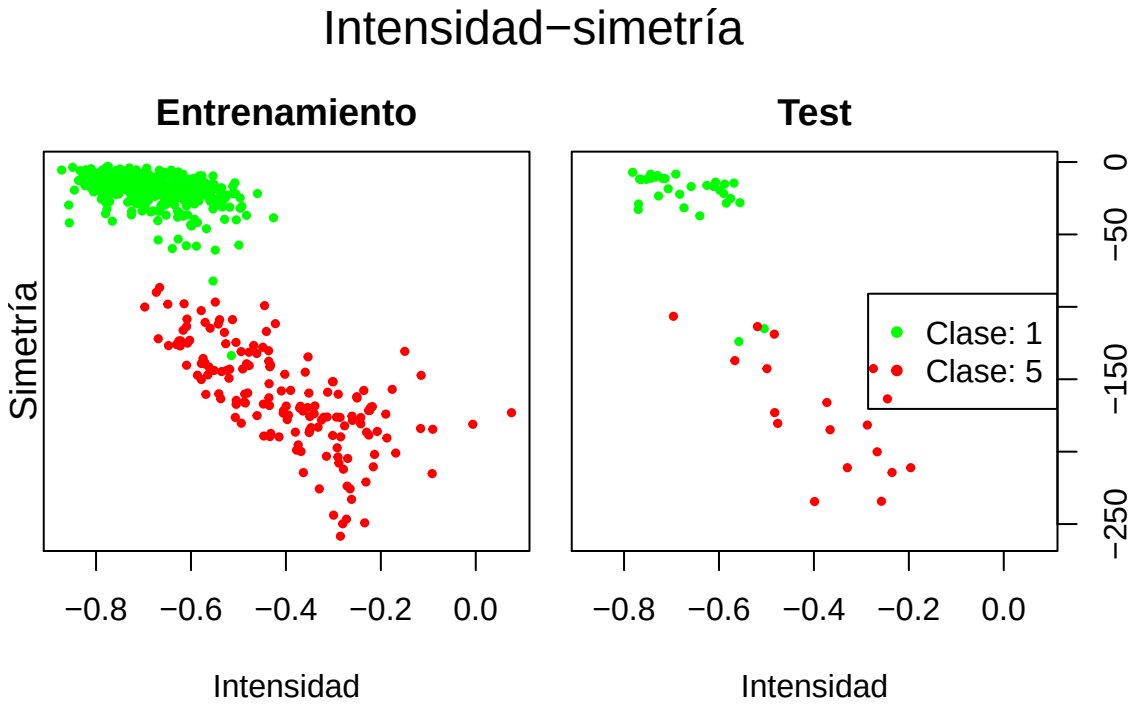
```
# Análisis de la intensidad y la simetría de los dígitos.
analisis_digito <- function(digito){
  # Nos aseguramos que el tipo de `digito` es el correcto
  digito <- as.numeric(digito)

  # Quitamos el primer valor (la clase) y ponemos los restantes
  # en una matriz de 16x16
  numero <- matrix(digito[-1], nrow = 16)

  # Tomamos también la matriz con las columnas invertidas
  numero.inv <- numero[, ncol(numero):1]

  # Calculamos intensidad y simetria
  intensidad <- mean(digito)
  simetria <- -sum( abs(numero.inv - numero) )

  # Devolvemos un vector con: la clase, la intensidad y la simetría
  c(digito[1], intensidad, simetria)
}
```

Esta función devuelve un vector concatenando la etiqueta del número, su intensidad media y su valor de simetría.

Recordemos el aspecto que tenían nuestros datos, tanto los de entrenamiento como los de test:

```
# Analizamos las filas de la matriz dígitos
digitos.analisis.train <- t(apply(digitos.train, 1, analisis_digito))
digitos.analisis.test <- t(apply(digitos.test, 1, analisis_digito))
```

```r
# Generamos un vector de colores basado en las etiquetas
colores.train <- ifelse(digitos.analisis.train[, 1] == 1, "green", "red")
colores.test <- ifelse(digitos.analisis.test[, 1] == 1, "green", "red")

# Tomamos el rango máximo de ambos datos (train y test) para que las gráficas
# tengan la misma escala
limX <- range(c(digitos.analisis.train[,2], digitos.analisis.test[,2]))
limY <- range(c(digitos.analisis.train[,3], digitos.analisis.test[,3]))

# Definimos una rejilla 1x2 para los plots
prev_par <- par(no.readonly = T)

# Ajustamos los márgenes
par(mfrow=c(1, 2), mar=c(5.0, 1.0, 2.1, 0.1), oma=2.5*c(1,1,1,1))

# Generamos las gráfica de los datos de entrenamiento
plot(digitos.analisis.train[, 2:3], col = colores.train, pch = 20, cex=0.75,
     main="Entrenamiento", xlab="Intensidad", yaxt="n", xlim = limX, ylim = limY)

# Añadimos nombre al eje Y
title(ylab="Simetría", mgp=c(1,1,10), line=0, cex.lab=1.2)

# Generamos las gráfica de los datos de test
plot(digitos.analisis.test[, 2:3], col = colores.test, pch = 20, cex=0.75,
     main="Test", xlab="Intensidad", ylab="", yaxt="n", xlim = limX, ylim = limY)

# Llevamos la escala del eje Y a la derecha
axis(4)
mtext("", side=4, line=3)

# Asignamos título a la gráfica global
mtext("Intensidad-simetría", outer=TRUE, line=0.5, cex=1.5)

# Añadimos leyenda
legend("right", c("Clase: 1", "Clase: 5"), col=c("green", "red"), pch=20)
```




```
# Dejamos los parámetros como estaban anteriormente
par(prev_par)
```

Reutilizamos la regresión lineal y el algoritmo PLA Pocket de la práctica anterior:

```
# Calcula regresión lineal de los datos (x_n, y_n) = (datos, label)
regresLin <- function(datos, label){
  # Añadimos una columna de 1 a los datos
  X <- cbind(datos, 1)
  Y <- label

  # Calculamos la descomposición de X
  SVD <- svd(X)
  D <- diag(SVD$d)
  U <- SVD$u
  V <- SVD$v

  # Calculamos la pseudo-inversa de X
  D[abs(D)>0] <- 1/D[abs(D)>0]
  X_pseudoinv <- V %*% D**2 %*% t(V)

  # Devolvemos la solución
  return(X_pseudoinv %*% t(X) %*% Y)
}

# Implementa el algoritmo Perceptron. Devuelve una lista con tres valores:
# Coefs: Pesos
# Sol: Recta solución
# Iter: Número de iteraciones que han sido necesarias
PLA <- function(datos, label, vini, max_iter = 100){
  # Definimos w como el vector inicial. Si tiene una posición más que
```

```r
# el número de datos, lo dejamos como está; si no, añadimos un 0.
w <- ifelse(length(vini) == ncol(datos) + 1, vini, c(vini, 0))

# Variables usadas en el bucle
changing <- T
iteraciones <- 0

# Añadimos una columna de unos a los datos
datos <- cbind(datos, 1)

# Inicializamos el número de errores al total de datos recibidos
# y la mejor solución al vector de pesos inicial
mejor_error <- length(datos[,1])
mejor_solucion <- w

# Bucle principal, del que salimos si no ha habido cambios tras una
# pasada completa a los datos (solución encontrada) o si se ha llegado
# al máximo de iteraciones permitidas (solución no encontrada)
while(changing && iteraciones < max_iter){
  iteraciones <- iteraciones+1

  changing <- F

  # Bucle sobre toda la muestra
  for(index in seq(label)){
    dato <- datos[index,]
    etiq <- label[index]

    # Comportamiento principal: si la muestra está mal etiquetada,
    # recalcular el hiperplano para etiquetarla bien.
    if(sign(sum(w * dato)) != etiq) {
      w <- w + etiq*dato
      changing <- T
    }
  }

  # Definimos la recta generada por los pesos
  recta <- -c(w[1], w[3]) / w[2]

  # Etiquetamos la muestra con la nueva recta
  nuevo_etiquetado <- generador_etiquetados(function(x, y){ y - recta[1]*x - recta[2] })
  nuevas_etiquetas <- nuevo_etiquetado(datos[,1], datos[,2])

  # Calculamos el número de muestras mal etiquetadas con la recta actual
  error_actual <- sum(nuevas_etiquetas != label)

  # Si el error actual es mejor que el mejor encontrado hasta ahora, guardamos
  # los pesos actuales y actualizamos el mejor error
  if(error_actual < mejor_error){
    mejor_error <- error_actual
    mejor_solucion <- w
  }
}
```

```
  # Actualizamos w con la mejor solución
  w <- mejor_solucion

  # Definimos la función estimada
  g <- function(x, y){
    return(y - recta[1]*x - recta[2])
  }

  # Devolvemos los pesos, el vector (a,b), coeficientes que determinan la
  # recta y = ax + b y el número de iteraciones.
  return(list("Coefs" = w, "Recta" = recta, "Iter" = iteraciones, "g" = g))
}
```

Podemos ya hacer la regresión lineal seguida del algoritmo PLA Pocket para hacer la clasificación:

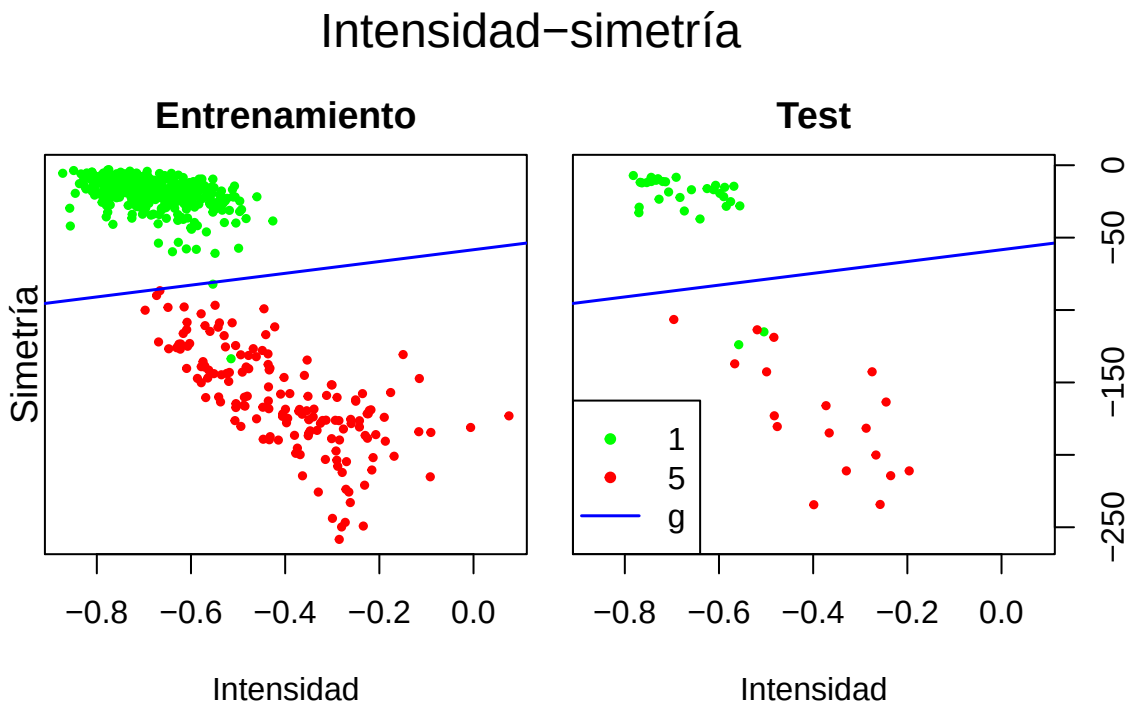
```
# Preparamos los datos
digitos.train.data <- digitos.analisis.train[,2:3]
digitos.train.etiq <- ifelse(digitos.analisis.train[,1] == 1, 1, -1)

digitos.test.data <- digitos.analisis.test[,2:3]
digitos.test.etiq <- ifelse(digitos.analisis.test[,1] == 1, 1, -1)

# Hacemos regresión lineal
digitos.reg <- regresLin(digitos.train.data, digitos.train.etiq)

# Usamos el resultado de la regresión como valor inicial para el PLA Pocket
digitos.pla <- PLA(digitos.train.data, digitos.train.etiq, vini = digitos.reg)
```

Apartado 5.a

Visualicemos la función g estimada junto con los datos de test y los de entrenamiento:

```
# Definimos una rejilla 1x2 para los plots
prev_par <- par(no.readonly = T)

# Ajustamos los márgenes
par(mfrow=c(1, 2), mar=c(5.0, 1.1, 2.1, 0.1), oma=2.5*c(1,1,1,1))

# Generamos las gráficas
plot(digitos.train.data, col = colores.train, pch = 20, cex=0.75,
     main="Entrenamiento", xlab="Intensidad", yaxt="n", xlim=limX, ylim=limY)
abline(rev(digitos.pla$Recta), col="blue", lwd=1.5)

# Añadimos etiqueta del eje Y a la izquierda
title(ylab="Simetría", mgp=c(1,1,10), line=0, cex.lab=1.2)

plot(digitos.test.data, col = colores.test, pch = 20, cex=0.75,
     main="Test", xlab="Intensidad", ylab="", yaxt="n", xlim=limX, ylim=limY)
abline(rev(digitos.pla$Recta), col="blue", lwd=1.5)

# Añadimos escala del eje Y a la derecha
axis(4)
```

```r
mtext("", side=4, line=3)

# Añadimos título global
mtext("Intensidad-simetría", outer=TRUE, line=0.5, cex=1.5)

# Añadimos leyenda
legend("bottomleft", c("1", "5", "g"),
       col=c("green", "red", "blue"), pch=c(20, 20, NA), lwd=c(NA, NA, 1.5))
```




```r
# Dejamos los parámetros como estaban anteriormente
par(prev_par)
```

Apartado 5.b

Hacemos el cálculo de E_{in} y E_{test}:

```r
# Definimos función etiquetadora
digitos.g.etiquetado <- generador_etiquetados(digitos.pla$g)

# Etiquetamos los datos de entrenamiento y de test
digitos.g.train.etiq <- digitos.g.etiquetado(digitos.train.data[,1], digitos.train.data[,2])
digitos.g.test.etiq <- digitos.g.etiquetado(digitos.test.data[,1], digitos.test.data[,2])

# Calculamos el error dentro de la muestra y en la muestra de test
E_in <- sum(digitos.g.train.etiq != digitos.train.etiq) / length(digitos.train.etiq)
E_test <- sum(digitos.g.test.etiq != digitos.test.etiq) / length(digitos.test.etiq)
```

Los errores obtenidos, $E_{in} = 0.3338898\%$ y $E_{test} = 4.0816327\%$ son excelentes, y nos dan una idea de lo bueno que este análisis: los datos son linealmente separables y un método sencillo como el de regresión lineal junto con PLA consiguen grandes resultados.

Apartado 5.c

Para obtener la cota de E_{out} basada en E_{in} podemos usar la cota de generalización de Vapnik-Chervonenkis, que afirma que para todo $\delta > 0$:

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

En este caso, la tolerancia es de $\delta = 0.05$, el número de muestras es $N = 599$ y la dimensión de Vapnik-Chervonenkis es $d_{VC} = 3$, ya que estamos separando muestras bidimensionales con una recta. Teniendo todo esto en cuenta, y que $E_{in} = 0.0033389$, podemos acotar el error fuera de la muestra como sigue:

$$E_{out}(g) \leq 0.5886032$$

Esto es, el error fuera de la muestra es como mucho de un 59%.

Para estudiar una cota del error fuera de la muestra basada en el error en la muestra de test, podemos usar la desigualdad de Hoeffding:

$$P(|E_{test}(g) - E_{out}(g)| > \varepsilon) \leq 2e^{-2N\varepsilon^2}$$

Como queremos estudiar cómo de parecidos son ambos errores, vamos a igualar la parte de la derecha de la desigualdad a $\delta = 0.05$, de manera que despejando el ε sabremos, con un 95% de seguridad, que $E_{out} \leq E_{test}(g) + \varepsilon$.

Por tanto:

$$\begin{aligned} 2e^{-2N\varepsilon^2} &= \delta \\ e^{-2N\varepsilon^2} &= \frac{\delta}{2} \\ -2N\varepsilon^2 &= ln(\frac{\delta}{2}) \\ \varepsilon &= \sqrt{-\frac{ln(\frac{\delta}{2})}{2N}} \end{aligned}$$

Tomando $N = 49$ y $\delta = 0.05$, obtenemos que $\varepsilon = 0.1940145$; es decir:

$$E_{out} \leq E_{test}(g) + \varepsilon = 0.2348308$$

Hemos conseguido así una cota mejor del error fuera de la muestra, ya que ahora tenemos que es como mucho de un 24%.

Apartado 5.d

La transformación polinómica de tercer orden es la siguiente:

$$\phi_3(x, y) = (1, x, y, x^2, y^2, xy, x^3, y^3, xy^2, x^2y)$$

Transformamos, por tanto, los datos de intensidad-simetría de los dígitos de la siguiente forma:

```
# Definimos la transformación polinómica de tercer orden
phi3 <- function(dato){
  x <- dato[1]
  y <- dato[2]

  return(c(x, y, x^2, y^2, x*y, x^3, y^3, x*y^2, x^2*y))
}

# Aplicamos la transformación a cada fila de la matriz de datos
digitos.train.dataTrans <- t(apply(digitos.train.data, 1, phi3))
```

Hacemos regresión lineal como antes:

```
# Hacemos regresión lineal
digitos.reg <- regresLin(digitos.train.dataTrans, digitos.train.etiq)
```

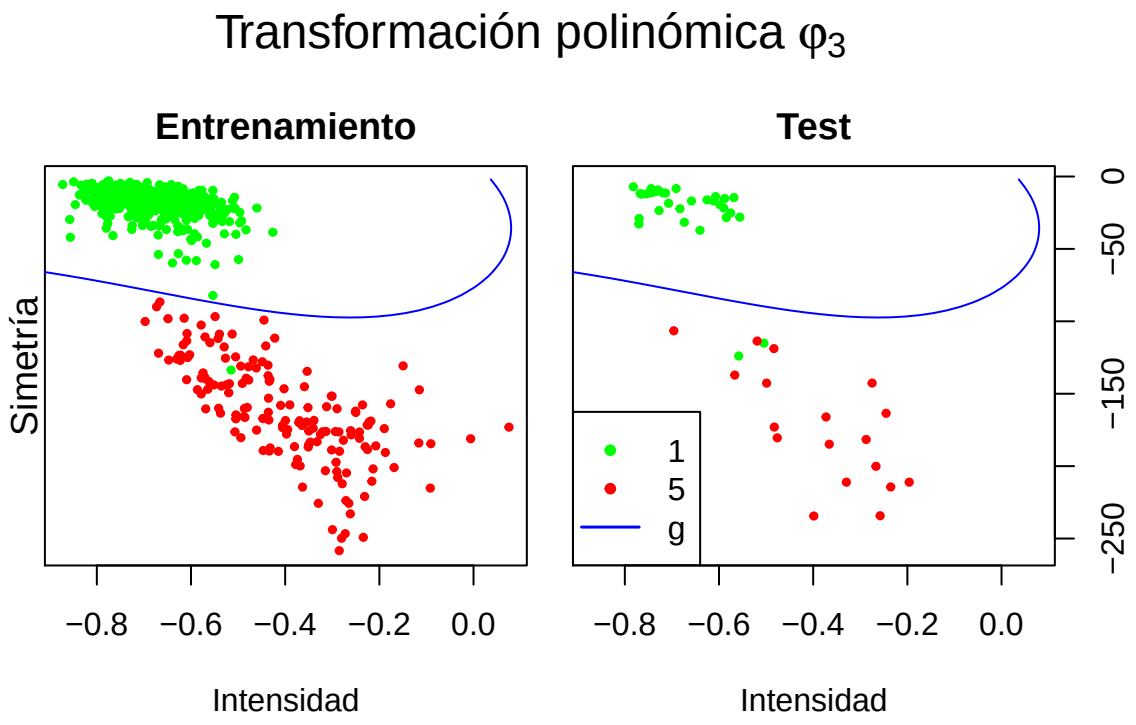
Ahora no podemos tomar directamente la función g devuelta por PLA; esta sólo nos sirve cuando los datos son de dos dimensiones. Definimos por tanto la función g en base a la transformación polinómica realizada:

```
# Función g estimada de la regresión. Devuelve <phi(x,y), coefs de regresión>
digitos.g.trans <- function(x,y){
  coef <- digitos.reg
  res <- x*coef[1] + y*coef[2] + x^2*coef[3] + y^2*coef[4] + x*y*coef[5] + x^3*coef[6] + y^3*coef[7] + x
  return(res)
}
```

Echémosle un vistazo a cómo ha cambiado la función estimada:

```
# Definimos una rejilla 1x2 para los plots
prev_par <- par(no.readonly = T)

# Ajustamos los márgenes
par(mfrow=c(1, 2), mar=c(5.0, 1.1, 2.1, 0.1), oma=2.5*c(1,1,1,1))

# Cálculo de la gráfica de la función
g.x <- seq(limX[1]-1, limX[2]+1, length=100)
g.y <- seq(limY[1]-1, limY[2]+1, length=100)
g.z <- outer(g.x, g.y, digitos.g.trans)

# Generamos las gráficas
plot(digitos.train.data, col = colores.train, pch = 20, cex=0.75,
     main="Entrenamiento", xlab="Intensidad", yaxt="n", xlim=limX, ylim=limY)

# Dibujamos la función estimada
contour(g.x, g.y, g.z, levels=0, col = "blue", add=T, drawlabels=F)

# Añadimos etiqueta del eje Y a la izquierda
title(ylab="Simetría", mgp=c(1,1,10), line=0, cex.lab=1.2)

plot(digitos.test.data, col = colores.test, pch = 20, cex=0.75,
     main="Test", xlab="Intensidad", ylab="", yaxt="n", xlim=limX, ylim=limY)
```

```r
# Dibujamos la función estimada
contour(g.x, g.y, g.z, levels=0, col = "blue", add=T,  drawlabels=F)

# Añadimos escala del eje Y a la derecha
axis(4)
mtext("", side=4, line=3)

# Añadimos título global
mtext(expression(paste("Transformación polinómica ", phi1[3])), outer=TRUE, line=0.5, cex=1.5)

# Añadimos leyenda
legend("bottomleft", c("1", "5", "g"),
       col=c("green", "red", "blue"), pch=c(20, 20, NA), lwd=c(NA, NA, 1.5))
```




```r
# Dejamos los parámetros como estaban anteriormente
par(prev_par)
```

Apartado 5.e

La transformación polinómica es muchísimo más costosa que la lineal, así que podría recomendarse sólo si la mejora en los errores dentro y fuera de la muestra son mucho mejores. Para tomar la decisión, midamos esos errores como hicimos con la transformación lineal:

```r
# Definimos función etiquetadora
digitos.g.etiquetado <- generador_etiquetados(digitos.g.trans)

# Etiquetamos los datos de entrenamiento y de test
digitos.g.train.etiq <- digitos.g.etiquetado(digitos.train.data[,1], digitos.train.data[,2])
digitos.g.test.etiq <- digitos.g.etiquetado(digitos.test.data[,1], digitos.test.data[,2])
```

```
# Calculamos el error dentro de la muestra y en la muestra de test
E_in.trans <- sum(digitos.g.train.etiq != digitos.train.etiq) / length(digitos.train.etiq)
E_test.trans <- sum(digitos.g.test.etiq != digitos.test.etiq) / length(digitos.test.etiq)
```

Vemos que $E_{in}(g_\Phi) = 0.1669449\% < E_{in}(g) = 0.3338898\%$; esto es, el error dentro de la muestra es algo mejor con la transformación polinómica. Sin embargo, al inspeccionar el error fuera de la muestra, observamos que $E_{test}(g_\Phi) = 4.0816327\% = E_{test}(g) = 4.0816327\%$: ¡el error fuera de la muestra de entrenamiento es igual! Este caso está muy cerca del sobreajuste, en el que intentamos dar más flexibilidad a la clase de funciones de ajuste y terminamos teniendo iguales o peores resultados en las muestras que no participan del entrenamiento.

Por tanto, nunca usaría esta transformación para dársela a un potencial cliente: es más costosa computacionalmente y arroja los mismos resultados.

Sobreajuste

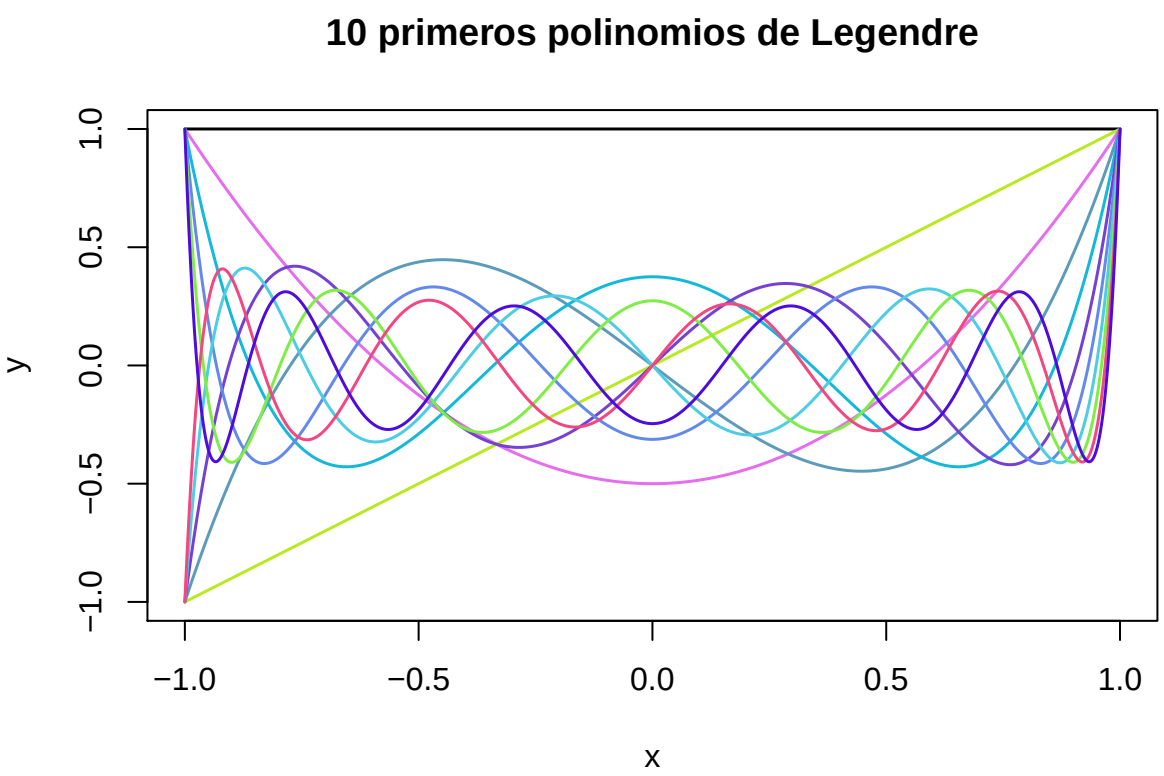
Ejercicio 1 - Sobreajuste

Lo primero que vamos a hacer es definir una función para calcular el polinomio de Legendre de orden k en el punto x. Se han implementado dos versiones: una recursiva, más compacta y elegante, y una iterativa, mucho más eficiente, y con la que trabajaremos a lo largo de este ejercicio. Veamos el código de ambas:

```
# Definición recursiva del polinomio de Legendre de orden k en el punto x
legendre <- function(k, x){
  if(k == 0 || k == 1){
    return(x^k)
  }
  else{
    L_k1 <- legendre(k-1,x)
    L_k2 <- legendre(k-2, x)
    L_k <- ((2*k - 1) / k) * x * L_k1 - ((k - 1) / k) * L_k2
    return(L_k)
  }
}

# Definición iterativa (menos elegante pero mucho más eficiente) del pol. de Legendre
legendre <- function(k,x){
  res = c(1,x)
  d = 3

  while(d <= k+1){
    res[d] = ((2*(d-1) - 1) / (d-1)) * x * res[d-1] - (((d-1) - 1) / (d-1)) * res[d-2]
    d <- d+1
  }

  return(res[k+1])
}
```

Podemos comprobar que todo ha salido bien haciendo una gráfica de los primeros 10 polinomios de Legendre, que podemos comparar con una de los numerosos gráficos iguales que se pueden encontrar en la literatura que cubre este tema:

```r
x <- seq(-1,1,by=0.001)

f <- function(x, k){
  legendre(k,x)
}

y <- sapply(x, f, 0)
plot(x,y, type='l', lwd=1.5, xlim=c(-1,1), ylim=c(-1,1), main="10 primeros polinomios de Legendre")

colores <- rainbow(10)

for (k in seq(10)) {
  color <- rgb(runif(1,0,1),runif(1,0,1),runif(1,0,1))

  y <- sapply(x, f, k)
  points(x,y, type='l', col=color, lwd=1.5, xlim=c(-1,1), ylim=c(-1,1))
}
```



Definimos ahora los valores de Q_f, N y σ:

```r
Qf <- 20
N <- 50
sigma <- 1
```

Generamos los coeficientes a_q:

```r
# Generamos los coeficientes normalizados
a_q <- rnorm(Qf+1) / sqrt(sum(sapply(seq(0,Qf), function(q){1 / (2*q + 1)})))
```

```
legendre.f <- function(x){
  res <- 0
  for (q in seq(0,Qf)) {
    res <- res + a_q[q+1] * legendre(q,x)
  }

  return(res)
}

legendre.eps = rnorm(N)
legendre.X = runif(N, -1, 1)
legendre.Y = sapply(legendre.X, legendre.f) + sigma*legendre.eps
```

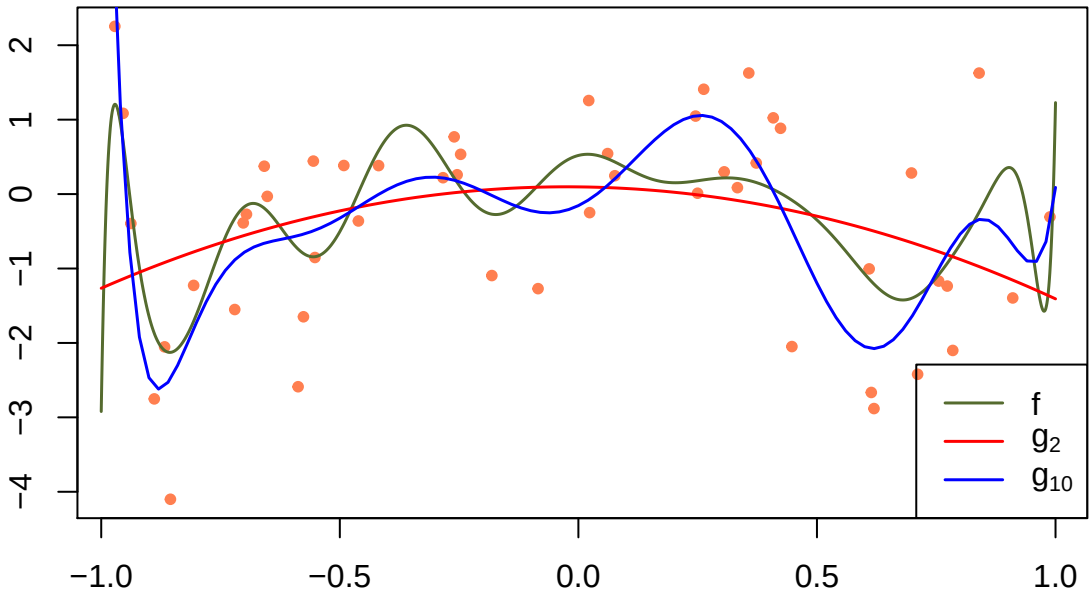
Apartado 1.a

Para ajustar los datos a $\mathcal{H}_2$ y a $\mathcal{H}_{10}$ usaremos regresión lineal; buscamos así los coeficientes de los polinomios de segundo y décimo grado, respectivamente, que ajustan mejor los datos:

```
# Aplicamos las transformaciones a los datos para ajustarlos a polinomios de grado 2 y 10
legendre.g2.dat = t(sapply(legendre.X, function(x){ sapply(seq(1,2), function(n){x^n}) }))
legendre.g10.dat = t(sapply(legendre.X, function(x){ sapply(seq(1,10), function(n){x^n}) }))

# Ajustamos con regresión lineal
legendre.g2.pesos <- regresLin(legendre.g2.dat, legendre.Y)
legendre.g10.pesos <- regresLin(legendre.g10.dat, legendre.Y)
```

Podemos ver cómo se ajustan ambas funciones g_2 y g_{10} a la función objetivo f:

```
legendre.g2 <- function(x){
  l <- legendre.g2.pesos
  return(l[3] + l[1]*x + l[2]*x^2)
}

legendre.g10 <- function(x){
  l <- legendre.g10.pesos
  return(l[11] + l[1]*x + l[2]*x^2 + l[3]*x^3 + l[4]*x^4 + l[5]*x^5
         + l[6]*x^6 + l[7]*x^7 + l[8]*x^8 + l[9]*x^9 + l[10]*x^10)
}

plot(legendre.X, legendre.Y, pch=20, col="coral", xlab="", ylab="", main="Ajuste")
points(x, sapply(x,legendre.f), col="darkolivegreen", type='l', lwd=1.5)
plot(legendre.g2, -1, 1, col="red", lwd=1.5, add=T)
plot(legendre.g10, -1, 1, col="blue", lwd=1.5, add=T)

legend("bottomright", c(expression(f), expression(g[2]), expression(g[10])),
       col=c("darkolivegreen", "red", "blue"), lwd=1.5)
```



Podemos estimar el error cuadrático medio fuera de la muestra midiendo el error en una muestra de test sin ruido:

```
test.X = runif(100, -1, 1)
leg.test.Y = sapply(test.X, legendre.f)

g2.test.Y = sapply(test.X, legendre.g2)
g10.test.Y = sapply(test.X, legendre.g10)

Eout.g2 <- sum((leg.test.Y - g2.test.Y)^2)/100
Eout.g10 <- sum((leg.test.Y - g10.test.Y)^2)/100
```

En este caso, estimamos que el error cuadrático medio fuera de la muestra es $E_{out}(g_2) \approx 0.4602546$ y $E_{out}(g_{10}) \approx 0.6499936$.

Apartado 1.b

En este caso, el valor de σ modela el ruido, de manera que este desciende si el valor es bajo y aumenta si es alto. La normalización trata de reducir la influencia de este ruido en la f.

Ejercicio 2 - Medida de sobreajuste

Vamos a definir ahora una función para medir el error en una muestra de tamaño m de las funciones estimadas g_2 y g_{10}:

```
# m : Tamaño de la muestra de test
# g : Función estimada
medirEout <- function(m, g){
  # Generamos una nueva muestra de tamaño m
  test.X = runif(m, -1, 1)
```

```
  # Generamos los datos reales
  test.f.Y = sapply(test.X, legendre.f)


  # Generamos los datos estimados
  test.g.Y = sapply(test.X, g)

  # Medimos el error cuadrático medio
  Eout <- sum((test.f.Y - test.g.Y)^2)/m

  return(Eout)
}
```

Repetimos ahora el experimento 100 veces con muestras de tamaño $m = 50$, tanto con g_2 como con g_{10}:

```
# TODO: Poner 1 a 100 y 5 a 50
Eout.g2 <- mean(replicate(100,medirEout(50, legendre.g2)))
Eout.g10 <- mean(replicate(100,medirEout(50, legendre.g10)))
```

Vemos que el error cuadrático medio con g_2 es de $E_{out}(g_2) = 0.4125448$ y, con g_{10}, de $E_{out}(g_{10}) = 0.8566231$. El sobreajuste es más que evidente tras estos experimentos.

Apartado 2.a

Es evidente que el ajuste con funciones de $\mathcal{H}_{10}$ es muchísimo más poderoso que el que podamos hacer con funciones de $\mathcal{H}_2$, debido a que el primero tiene muchos más grados de libertad que el último. Aun trabajando con datos ruidosos como los que tenemos entre manos, es claro que g_{10} tendrá un error dentro de la muestra siempre menor al que pueda tener g_2. Por tanto, podemos afirmar que $\mathcal{H}_{10}$ ajusta mejor los datos *de la muestra* que $\mathcal{H}_2$.

Sin embargo, lo que a nosotros nos interesa es el error *fuera* de la muestra, y es aquí donde se produce el sobreajuste: al haber ajustado la función a los datos ruidosos, como $\mathcal{H}_{10}$ es más flexible, con ella estamos aprendiendo muchísimo del ruido; como $\mathcal{H}_2$ es menos flexible, el ajuste a los datos de la muestra es peor pero puede comportarse mejor en puntos nuevos. De hecho, y en contraste con lo dicho en el párrafo anterior —esto es, que normalmente $E_{in}(g_{10}) < E_{in}(g_2)$—, es muy posible que en el error fuera de la muestra la desigualdad se dé vuelta y tengamos $E_{out}(g_{10}) > E_{out}(g_2)$; es decir, $E_{out}(g_{10}) - E_{out}(g_2) > 0$.

Este comportamiento es uno de los síntomas del sobreajuste, así que parece sensato tomar esa diferencia como medida de sobreajuste: cuando más grande —en positivo— sea, mayor será el sobreajuste.

Regularización y selección de modelos

Ejercicio 1 - Estudio

Fijamos el número de muestras $N = 100$, la dimensión $d = 3$ y generamos el conjunto de datos que se nos pide, reutilizando la función `simula_gauss` implementada en la primera práctica

```
# Devuelve una lista de N vectores de dimensión dim con una muestra
# gaussiana de media cero y desviación sigma
simula_gauss <- function(N, dim, sd_){
    t(sapply(rep(dim, N), rnorm, mean = 0, sd = sd_))
```

```
}

# Fijamos las constantes del experimento
N <- 100
d <- 3
sigma <- 0.5

# Generamos la muestra
reg.X <- simula_gauss(N, d, sd = 1)

# Generamos los pesos
reg.W <- simula_gauss(1, d+1, sd = 1)

# Definimos las etiquetas
reg.Y <- sapply(1:N, function(i){ reg.W %*% c(reg.X[i,],1) }) + sigma * rnorm(N)

# Definimos el parametro de regularizacion
reg.lambda <- 0.05 / N
```

Estimamos ahora w_f con w_{reg} usando regularización *weight decay*:

```
# Regresión lineal con weight decay
weightDecay <- function(X,Y, lambda){
  # Calculamos Z, su traspuesta y la diagonal lambda*I
  Z <- cbind(X,1)
  Zt <- t(Z)
  lambdaI <- diag(lambda, ncol(Z))

  # Calculamos la solución
  w <- solve(Zt %*% Z + lambdaI) %*% Zt %*% Y

  return(w)
}

reg.Wreg <- weightDecay(reg.X, reg.Y, reg.lambda)
```

Apartado 1.a

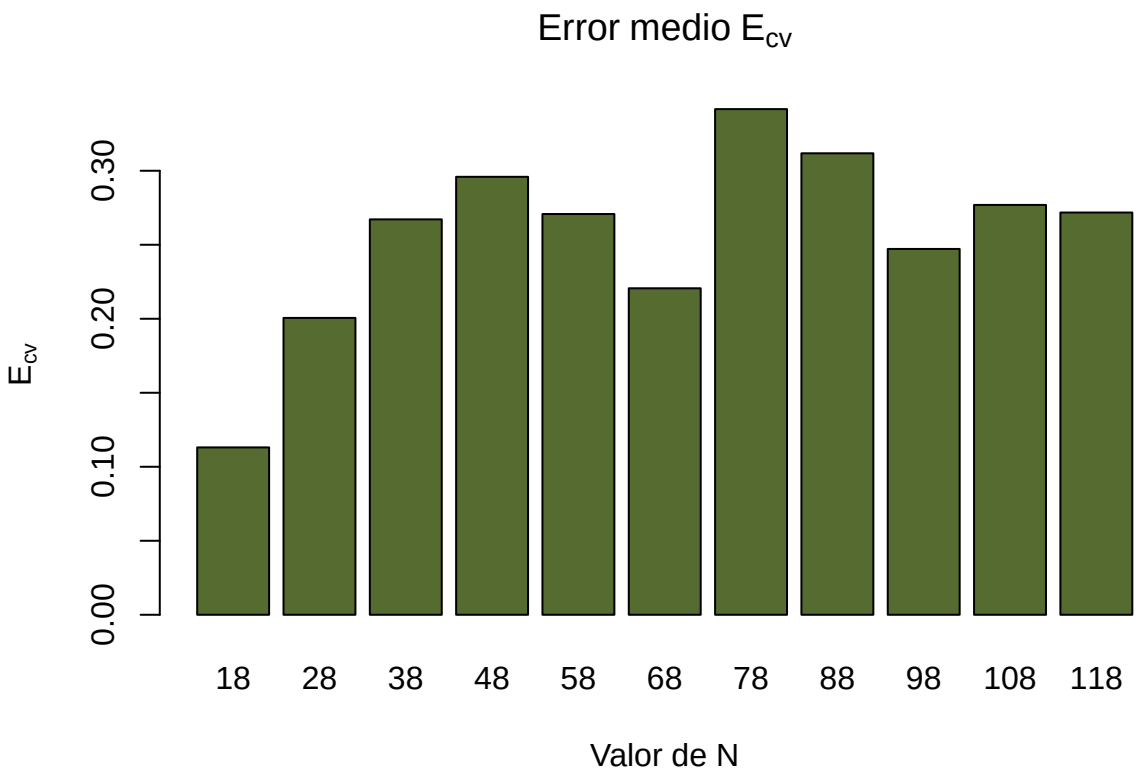
Definimos los valores de N como se nos indica e implementamos una función que, dado N, devuelve un vector con los errores de *leaev-one-out* $e_1, e_2, \ldots, e_N$:

```
valoresN <- seq(d+15, d+115, by = 10)

# Definimos una función para medir e_{loo}; es decir, el error usando
# leave-one-out con el conjunto de test = {x_{loo}}
errorLOO <- function(loo, N, X, Y, lambda){
  indices <- setdiff(1:N, loo)

  Wreg <- weightDecay(X[indices,], Y[indices], lambda)

  return((Y[loo] - (t(Wreg) %*% c(X[loo,],1)))^2)
}
```

```r
# Devuelve un vector con los errores e_i, con i \in {1, ..., N}
medirErrores <- function(N, numLambda = 0.05){
  # Generamos la muestra
  X <- simula_gauss(N, d, sd = 1)

  # Generamos los pesos
  W <- simula_gauss(1, d+1, sd = 1)

  # Definimos las etiquetas
  Y <- sapply(1:N, function(i){ W %*% c(X[i,],1) }) + sigma * rnorm(N)

  # Definimos el parametro de regularización
  lambda <- numLambda / N

  # Devolvemos todos los e_i
  return(sapply(1:N, errorLOO, N, X, Y, lambda))
}
```

Podemos ya medir los errores $e_1, \ldots, e_{N_i}$ para $N_i \in \{d+15, d+25, \ldots, d+115\}$ y obtener sus medias, E_{cv_i} y visualizar los errores medios para cada N_i:

```r
# Medimos los errores e_i y calculamos su media E_{cv}
errores <- sapply(valoresN, medirErrores)
erroresMedios <- sapply(errores, mean)

# Generamos gráfico de barras
barplot(erroresMedios,col = "darkolivegreen", names.arg=valoresN,
        xlab = "Valor de N", ylab = expression(E[cv]),
        main=expression(Error~medio~E[cv]))
```



Apartado 1.b

Vamos a definir una función que devuelva, del experimento anterior, la media y varianza de e_1, e_2 y E_{cv}. La ejecutamos 1000 veces para obtener una matriz en cuyas fila i se encuentran los resultados del experimento i-ésimo:

```
experimento <- function(N, numLambda = 0.05){
  # Medimos los errores e_i y calculamos su media E_{cv}
  e_i <- replicate(1000, medirErrores(N, numLambda))
  E_cv <- apply(e_i, 2, mean)

  e1 <- e_i[1,]
  e2 <- e_i[2,]

  return(c(mean(e1), mean(e2), mean(E_cv), var(e1), var(e2), var(E_cv)))
}

exp.res <- t(sapply(valoresN, experimento))
colnames(exp.res)  <- c("MediaE_1", "MediaE_2", "MediaE_cv", "VarE_1", "VarE_2", "VarE_cv")
rownames(exp.res) <- c(valoresN)
```

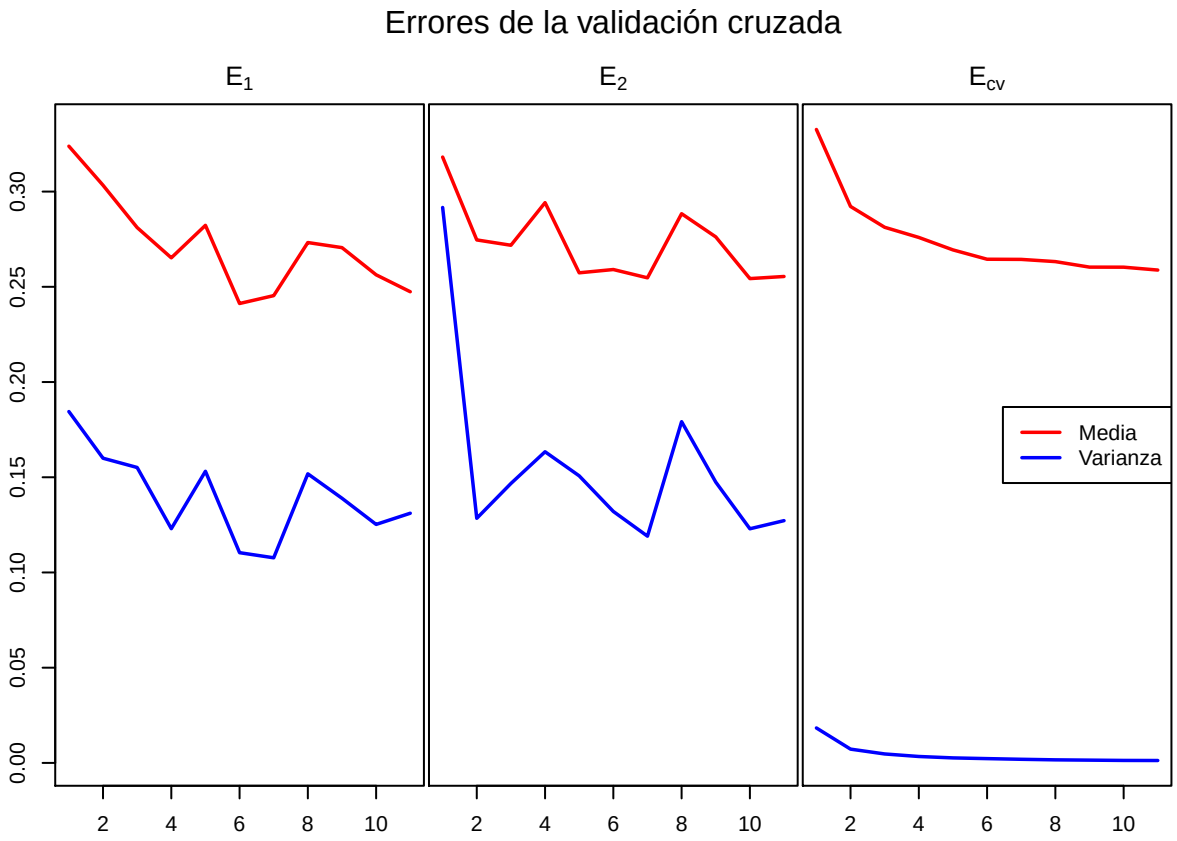
La siguienta tabla resume los valores de medias y varianzas de cada uno de los errores medidos para cada $N \in \{18, 28, 38, \ldots, 118\}$:

	MediaE_1	MediaE_2	MediaE_cv	VarE_1	VarE_2	VarE_cv
18	0.3238252	0.3181299	0.3326058	0.1844896	0.2916615	0.0183588
28	0.3033316	0.2746176	0.2921936	0.1599966	0.1284426	0.0072376

	MediaE_1	MediaE_2	MediaE_cv	VarE_1	VarE_2	VarE_cv
38	0.2811118	0.2718166	0.2812530	0.1551271	0.1467176	0.0047032
48	0.2652226	0.2941924	0.2759142	0.1229932	0.1633765	0.0033801
58	0.2822781	0.2573457	0.2693772	0.1531264	0.1507341	0.0026363
68	0.2412311	0.2590724	0.2644734	0.1103957	0.1320358	0.0022579
78	0.2453619	0.2547174	0.2643908	0.1077054	0.1190629	0.0018976
88	0.2732445	0.2883742	0.2632251	0.1518031	0.1791836	0.0016164
98	0.2705696	0.2762063	0.2603267	0.1389634	0.1474478	0.0014624
108	0.2562947	0.2543167	0.2603069	0.1252378	0.1229474	0.0013028
118	0.2474010	0.2554127	0.2587882	0.1311037	0.1272062	0.0012548

Podemos ver también cómo se comportan estos valores dibujando su valor con respecto al N:

```
# Definimos una rejilla 1x3 para los plots
prev_par <- par(no.readonly = T)

par(mfrow=c(1, 3), mar=c(0.1, 0.1, 2.1, 0.1), oma=2.5*c(1,1,1,1))

yrange = range(exp.res)

plot(exp.res[,"MediaE_1"], type='l', col="red", lwd=1.75, main=expression(E[1]), ylim = yrange)
lines(exp.res[,"VarE_1"], type='l', col="blue", lwd=1.75)

plot(exp.res[,"MediaE_2"], type='l', col="red", lwd=1.75, main=expression(E[2]), ylim = yrange, yaxt="n
lines(exp.res[,"VarE_2"], type='l', col="blue", lwd=1.75)

plot(exp.res[,"MediaE_cv"], type='l', col="red", lwd=1.75, main=expression(E[cv]), ylim = yrange, yaxt=
lines(exp.res[,"VarE_cv"], type='l', col="blue", lwd=1.75)

mtext("Errores de la validación cruzada", outer=TRUE, line=0.5)

legend("right", c("Media", "Varianza"),
       bg="white", col=c("red", "blue"), lty=1, lwd=1.75)
```




```r
# Dejamos los parámetros como estaban anteriormente
par(prev_par)
```

Apartado 1.c

Los valores medios de e_1, e_2 y E_{cv} deben ser muy parecidos: cada e_i se calculando eliminando uno y sólo uno de los datos de entrenamiento, luego este valor y la media global de todos los e_i debe ser parecida, como así se muestra en los datos obtenidos en el anterior experimento.

Apartado 1.d

De la misma manera que las medias son muy similares entre los e_i y el E_{cv}, es claro que la varianza en el último es muchísimo más pequeña que en los primeros. La causa de la baja varianza del último está clara: los datos de cada experimento están tomados de una $\mathcal{N}(0,1)$, luego de forma global el error va a ser siempre parecido, y únicamente dependiente de lo bueno que es el modelo y cómo de bien ajusta los datos. Sin embargo, las varianzas de los e_1 y e_2 van a ser siempre superiores: cada e_i reprenseta el error de la muestra i-ésima con respecto a lo aprendido de las demás; esto es muy dependiente de la muestra en cuestión, del ruido y de cómo se ajusta el modelo a ese punto concreto. Es evidente que el modelo puede ajustarse bien de forma global pero no en todos y cada uno de los puntos, y esto es lo que contribuye a la mayor varianza de los e_i aislados.

Apartado 1.e

Si los errores de validación cruzada fueran independientes, la varianza de los e_i y de E_{cv} debería ser similar. No es así porque en realidad los e_i no son independientes: en el cálculo de e_i se usa la muestra (x_j, y_j) como

entrenamiento, mientras que para el cálculo de e_j estamos usando justo esa muestra como test; en definitiva, los errores individuales no son independientes. Podríamos conseguir que fueran individuales calculando cada e_i con una muestra de puntos completamente diferente, pero esto es inviable en la práctica. Esta situación nos lleva, por tanto, a que las varianzas sean muy distintas, al contrario de lo que debería pasar si los e_i fueran independientes.

Apartado 1.f

Si desarrollamos el cociente descrito veremos que el número de muestras N_{eff} es precisamente N:

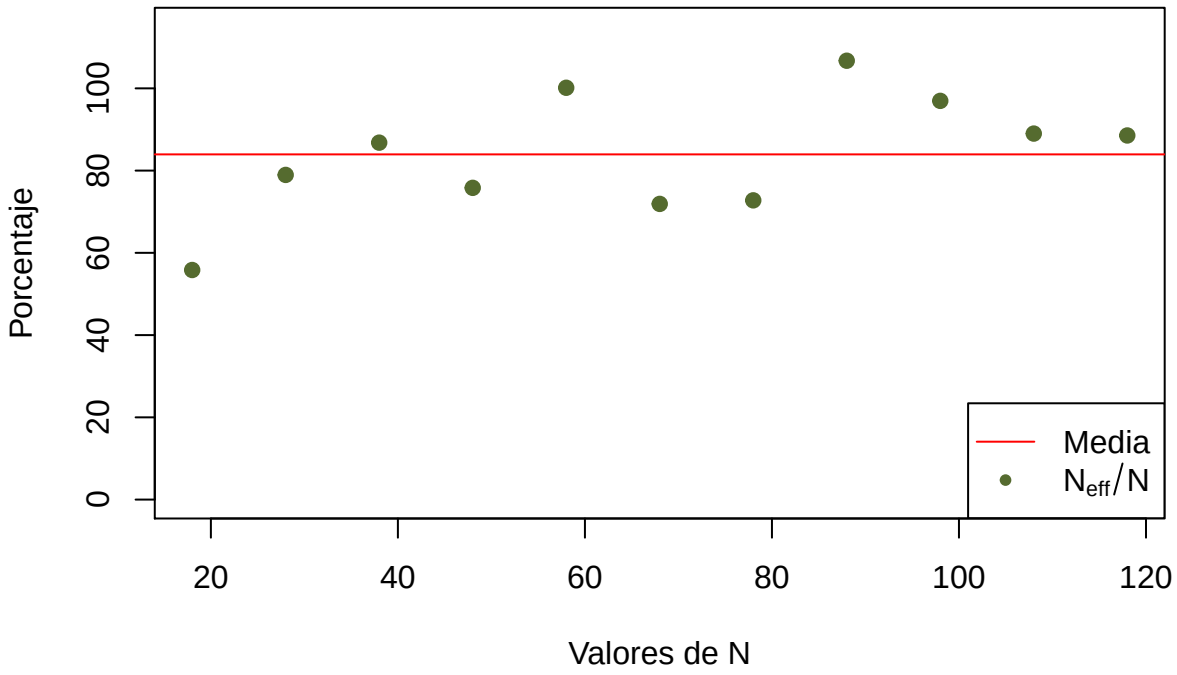
$$\frac{Var(e_1)}{Var(E_{cv})} = \frac{Var(e_1)}{Var(\frac{1}{N}\sum_{i=1}^{N} e_i)} = \frac{Var(e_1)}{\frac{1}{N^2}\sum_{i=1}^{N} Var(e_i)}$$
$$= N^2 \frac{Var(e_1)}{\sum_{i=1}^{N} Var(e_i)} \approx N^2 \frac{Var(e_1)}{NVar(e_1)} = N$$

donde hemos usado que los errores e_i deben ser variables i.i.d. —en este caso sabemos que no son totalmente independientes, por lo explicado en el anterior apartado, así que hemos notado esa igualdad como una aproximación— luego deben tener la misma varianza.

El desarrollo matemático anterior prueba entonces que este cociente modela el número de muestras nuevas usadas en el cálculo de E_{cv}.

Podemos comprobar de forma experimentalmente que la teoría es cierta:

```
calcularNeff <- function(datos){
  Neff <- datos[,"VarE_1"] / datos[,"VarE_cv"]

  plot(valoresN, 100*Neff/valoresN, ylim=c(0,115), pch=20, cex=1.5, col="darkolivegreen",
       xlab="Valores de N", ylab = "Porcentaje")
  abline(c(mean(100*Neff/valoresN),0), col = "red")

  legend('bottomright',c('Media', expression(N[eff]/N)),
         lty=c(1,NA), pch=c(NA,20),
         col=c("red", "darkolivegreen"))

  return(mean(100*Neff/valoresN))
}

exp.res.Neff <- calcularNeff(exp.res)
```



Vemos que el porcentaje de N_{eff} con respecto a N es de un 83.9480344%.

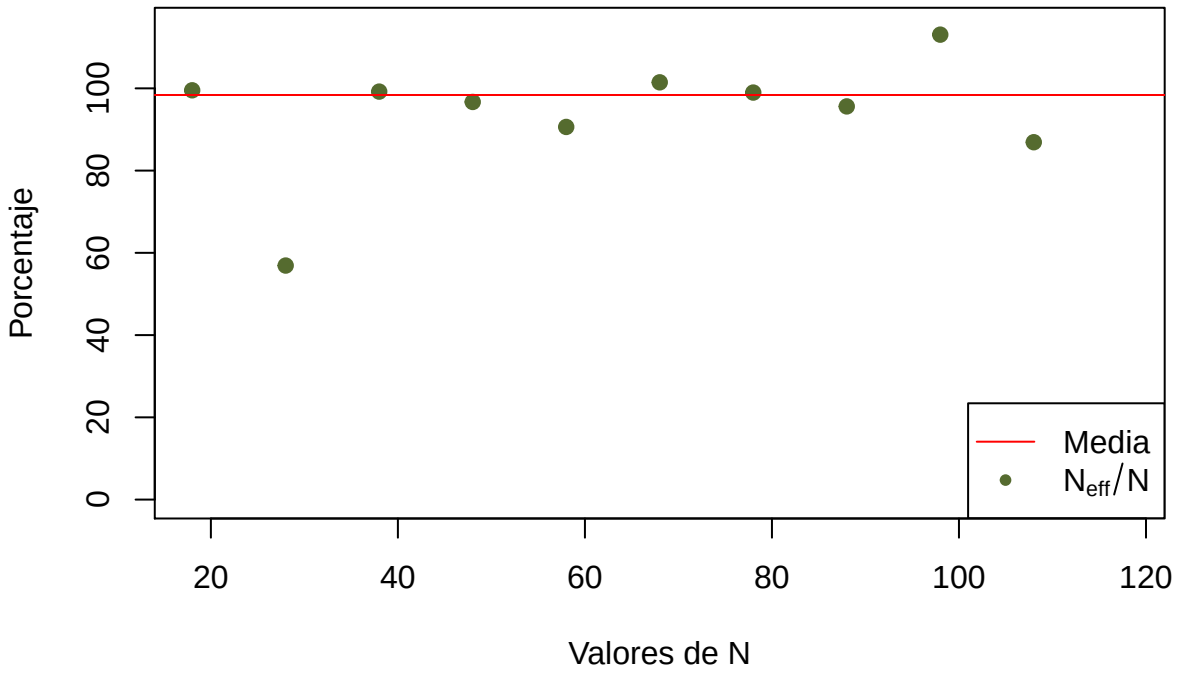
Apartado 1.e

Al aumentar el parámetro de regularización lo que hacemos es relajar el ajuste a los datos de la muestra para intentar mejorar el error fuera de ella. Sin embargo, si lo aumentamos mucho, el error de cada uno de los datos por separado va a oscilar demasiado —el ajuste a cada punto es más flexible—; por otro lado, el error medio de validación cruzada va a quedarse más o menos estable, así que la varianza se mantendrá también estable.

Como el número de muestras efectivas lo hemos tomado como el cociente $N_{eff} = \frac{Var(e_1)}{Var(E_{cv})}$ y acabamos de ver que el numerador aumenta y el denominador se queda fijo, N_{eff} aumentará cuando aumentemos el parámetro de regularización λ.

Repetimos el experimento tomando $\lambda = \frac{2.5}{N}$:

```
exp.res.2.5 <- t(sapply(valoresN, experimento, 2.5))
colnames(exp.res.2.5)  <- c("MediaE_1", "MediaE_2", "MediaE_cv", "VarE_1", "VarE_2", "VarE_cv")
rownames(exp.res.2.5) <- c(valoresN)
exp.res.2.5.Neff <- calcularNeff(exp.res.2.5)
```



Comprobamos así, corroborando el argumento anterior, que el porcentaje de N_{eff} con respecto a N con $\lambda = \frac{2.5}{N}$ es de un 98.3867337%, 14.4386993 puntos mejor que el anterior.