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 gradiente siguiente:

```
# Devuelve el valor del gradiente de E en el punto (u,v)
E.gradiente <- function(punto){
  u <- punto[1]
  v <- punto[2]

  coeff <- 2*(u*exp(v) - 2*v*exp(-u))

  return(coeff * c(exp(v) + 2*v*exp(-u), u*exp(v) - 2*exp(-u)))
}
```

```
E <- function(punto){
  u <- punto[1]
  v <- punto[2]

  return((u*exp(v) - 2*v*exp(-u))^2)
}

gradienteDescendente <- function(f, f.gradiente, w_0 = c(1,1), eta = 0.1, tol = 1e-14, maxIter = 100){
  w <- w_0
  iteraciones <- 0
```

```
  while(f(w) >= tol && iteraciones < maxIter){
    direccion <- -f.gradiente(w)
    w <- w + eta*direccion
    iteraciones <- iteraciones + 1
  }

  return(list("Pesos"=w, "Iter"=iteraciones))
}
```

Ejecutamos la función anteriormente implementada:

```
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:

```
f <- function(punto){
  x <- punto[1]
  y <- punto[2]

  return( x*x + 2*y*y + 2*sin(2*pi*x)*sin(2*pi*y) )
}
```

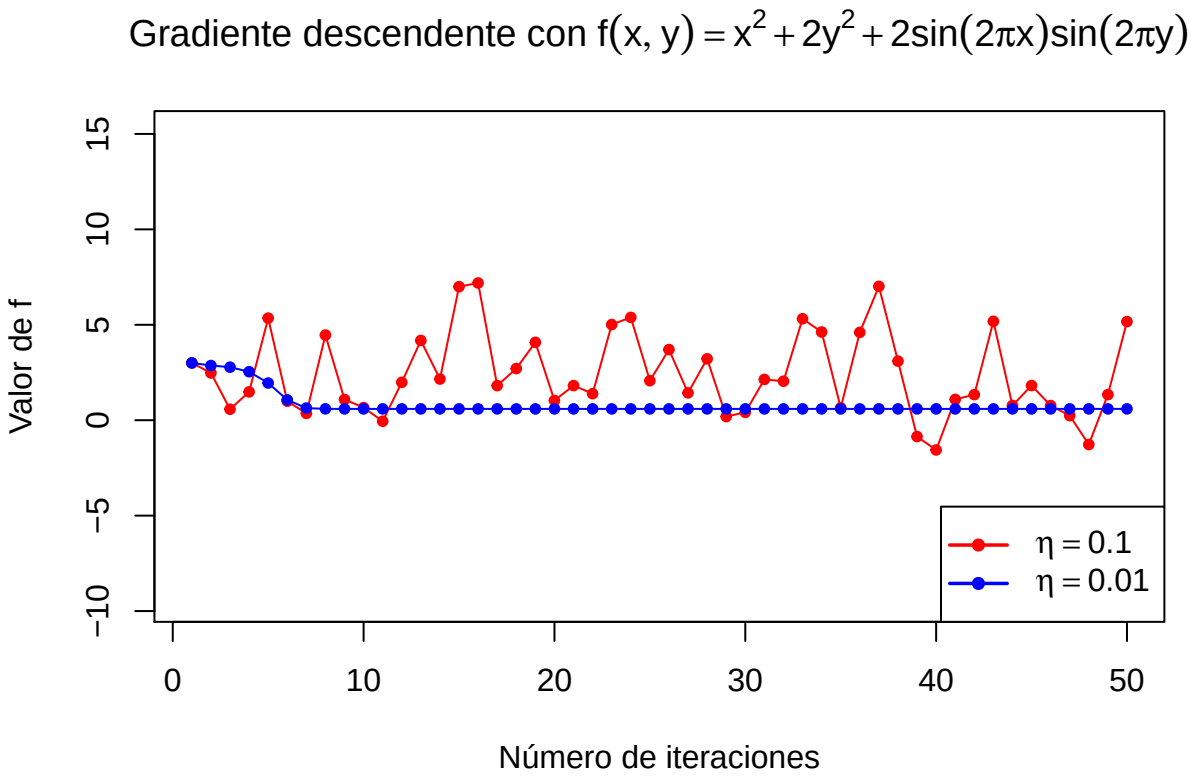
```
# Devuelve el valor del gradiente de f en el punto (x,y)
f.gradiente <- function(punto){
  x <- punto[1]
  y <- punto[2]

  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)

  return(c(f.grad.x, f.grad.y))
}
```

Para poder obtener el gráfico solicitado tenemos que modificar la función del gradiente descendente, de manera que devuelva un vector con el valor de la función en las sucesivas iteraciones y el criterio de parada sea ahora únicamente el número de iteraciones:

```
gradienteDescendente.log <- function(f, f.gradiente, w_0 = c(1,1), eta = 0.1, maxIter = 50){
  w <- w_0
  valorIter <- f(w)


  while(length(valorIter) < maxIter){
    direccion <- -f.gradiente(w)
    w <- w + eta*direccion
    valorIter <- c(valorIter, f(w))
  }

  return(list("Pesos"=w, "Iter"=length(valorIter)-1, "Valores"=valorIter))
}
```

Generamos ahora los datos y el gráfico con la función modificada:

```
# Ejecutamos el experimento con ambos parámetros
resultado.f.0.01 <- gradienteDescendente.log(f, f.gradiente, eta = 0.01)
resultado.f.0.1 <- gradienteDescendente.log(f, f.gradiente, eta = 0.1)

# Generamos los dos plots
plot(resultado.f.0.1$Valores, col = 'red', pch = 20, asp=1, type='o',
     main=expression(paste("Gradiente descendente con ",
                           f(x,y) == x^2 + 2*y^2 + 2*plain(sin)(2*pi*x)*plain(sin)(2*pi*y))),
     xlab="Número de iteraciones", ylab="Valor de f")
points(resultado.f.0.01$Valores, 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)
```



Apartado 1.b.2

Redefinimos la función, de nuevo, para que devuelva también un vector de los puntos por los que pasa. El criterio de parada sigue siendo el número máximo —cincuenta en este caso— de iteraciones:

```
gradienteDescendente.log2 <- function(f, f.gradiente, w_0 = c(1,1), eta = 0.1, maxIter = 50){
  w <- w_0
  variables.x <- w[1]
  variables.y <- w[2]
  valores <- f(w)


  while(length(valores) < maxIter){
    direccion <- -f.gradiente(w)
    w <- w + eta*direccion
    variables.x <- c(variables.x, w[1])
    variables.y <- c(variables.y, w[2])
    valores <- c(valores, f(w))
  }

  return(list("Pesos"=w, "Iter"=length(valores)-1,
              "Res"=data.frame(X = variables.x, Y = variables.y, Value = valores)))
}
```

Generamos ahora los datos desde los datos iniciales indicados y tomamos los mínimos

```r
# 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.log2(f, f.gradiente, 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 <- 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("(0.1, 0.1)", "(1, 1)", "(-0.5, 0.5)", "(-1, -1)")

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

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

Ejercicio 2 - Coordenada descendente

Apartado 2.1

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

```r
coordenadaDescendente <- function(f, f.grad, w_0 = c(1,1), eta = 0.1, tol = 1e-14, maxIter = 50){
  w <- w_0
  iteraciones <- 0

  while(f(w) >= tol && iteraciones < maxIter){
    direccion.x <- c(-f.grad(w)[1], 0)
    w <- w + eta*direccion.x

    direccion.y <- c(0, -f.grad(w)[2])
    w <- w + eta*direccion.y

    iteraciones <- iteraciones + 1
  }

  return(list("Pesos"=w, "Iter"=iteraciones))
}
```

```
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 virtualmente $-\infty$ para asegurarnos de que las iteraciones se realizan:

```
# Tomamos los resultados para f
comp.f.grad <- gradienteDescendente(f, f.gradiente, maxIter = 20, tol=-999999)
comp.f.coor <- coordenadaDescendente(f, f.gradiente, maxIter = 20, tol=-999999)

# Tomamos los resultados para E
comp.E.grad <- gradienteDescendente(E, E.gradiente, maxIter = 20, tol=-999999)
comp.E.coor <- coordenadaDescendente(E, E.gradiente, maxIter = 20, tol=-999999)

# 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 rápida para 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	1.814305
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=-999999)
comp.f.coor <- coordenadaDescendente(f, f.gradiente, maxIter = 10, tol=-999999)

# Tomamos los resultados para E
comp.E.grad <- gradienteDescendente(E, E.gradiente, maxIter = 20, tol=-999999)
comp.E.coor <- coordenadaDescendente(E, E.gradiente, maxIter = 10, tol=-999999)

# 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	1.814305
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 caso, 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 las funciones 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, para estudiar qué pasa en cada momento:

```
metodoNewton <- function(f, w_0 = c(1,1), eta = 0.1, maxIter = 50){
  w <- w_0
  variables.x <- w[1]
  variables.y <- w[2]
  valores <- f(w)


  while(length(valores) < maxIter){
    direccion <- -solve(hessian(f, w)) %*% grad(f, w)
    w <- w + eta*direccion
    variables.x <- c(variables.x, w[1])
    variables.y <- c(variables.y, w[2])
    valores <- c(valores, f(w))
  }

  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:

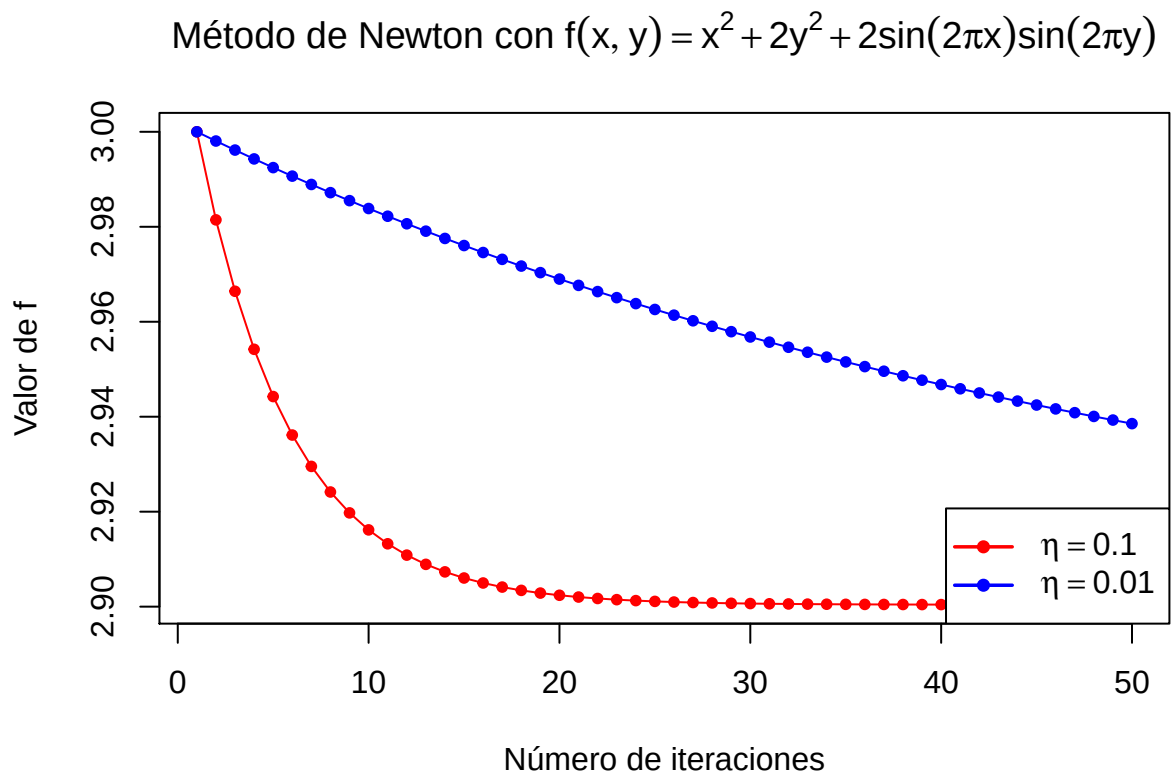
```
# Definimos una función que, dado p, devuelve el valor mínimo del método de Newton
# con punto inicial = (p,p) y dónde se alcanza este:
minimoGradiente <- function(p){
  valores <- metodoNewton(f, c(p, p))$Res
  unlist(valores[which.min(valores$Value),])
}

# 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)

# 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)")

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

	X	Y	Valor
(0.1, 0.1)	0.00041	0.00040	0.00001

	X	Y	Valor
(1, 1)	0.94944	0.97473	2.90041
(-0.5, 0.5)	-0.47526	-0.48788	0.72548
(-1, -1)	-0.94944	-0.97473	2.90041

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:

```
# Ejecutamos el experimento con ambos parámetros
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("bottomright", c(expression(eta == 0.1), expression(eta == 0.01)),
       bg="white", col=c("red", "blue"), lty=1, pch=20, lwd=1.75)
```

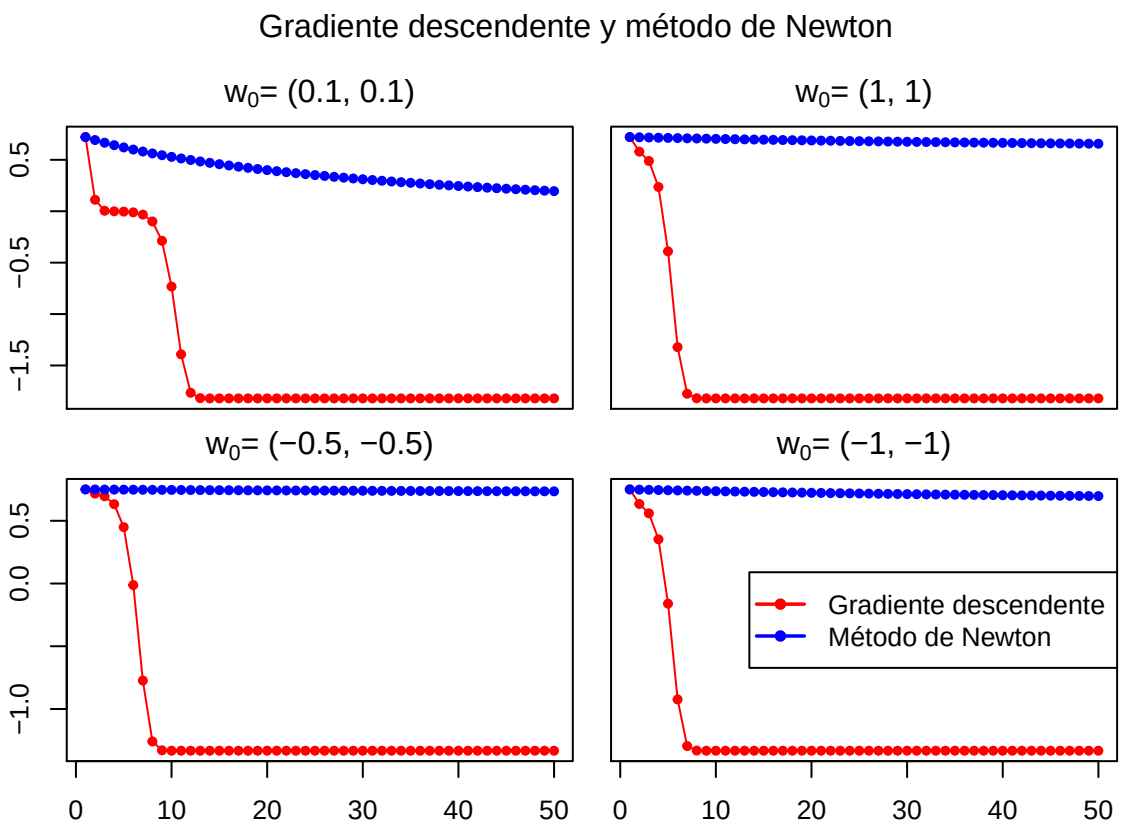


En este caso es claro que la tasa de aprendizaje más alta encuentra más rápido el óptimo local, ya que se mueve más rápido.

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, un método de descenso y una función, de manera que genere el gráfico correspondiente:

```
dibujarComparacion <- function(p, foo = f, tasa = 0.01, ...){
  res.grad <- unlist(gradienteDescendente.log2(f, f.gradiente, w_0 = c(p,p), eta = tasa)$Res["Value"])
  res.newt <- unlist(metodoNewton(f, w_0 = c(p,p), eta = tasa)$Res["Value"])

  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)

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

# . <- lapply(puntosIniciales, dibujarComparacion)
dibujarComparacion(0.1, xaxt="n")
dibujarComparacion(1, xaxt="n", yaxt="n")
dibujarComparacion(-0.5)
dibujarComparacion(-1, yaxt="n")

mtext("Gradiente descendente y método de Newton", outer=TRUE, line=0.5)

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



```
# 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:

```
simula_unif <- function(N, dim, rango){
  lapply(rep(dim, N), runif, min = rango[1], max = rango[2])
}

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)
}

generador_etiquetados <- function(f){
  function(x,y){
    sign(f(x,y))
  }
}
```

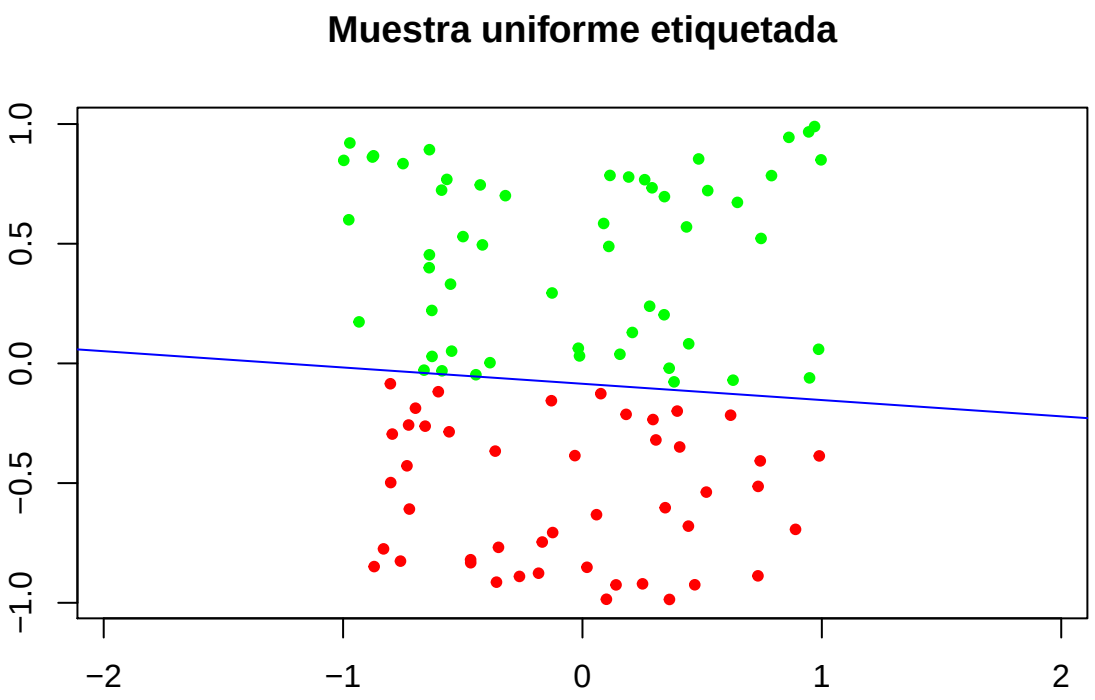
Definimos la función f y el conjunto $\mathcal{D}$:

```
regLog.frontera <- simula_recta(c(-1,1))
datos <- simula_unif(100, 2, c(-1,1))
regLog.datos <- matrix(unlist(datos), ncol = 2, byrow = T)

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

Echémosle un vistazo a nuestros datos:

```
regLog.etiquetado <- generador_etiquetados(regLog.f)
regLog.etiquetas <- regLog.etiquetado(regLog.datos[,1], regLog.datos[,2])

# Genera vector de colores basado en las etiquetas
```

```r
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")
```



Apartado 4.a

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

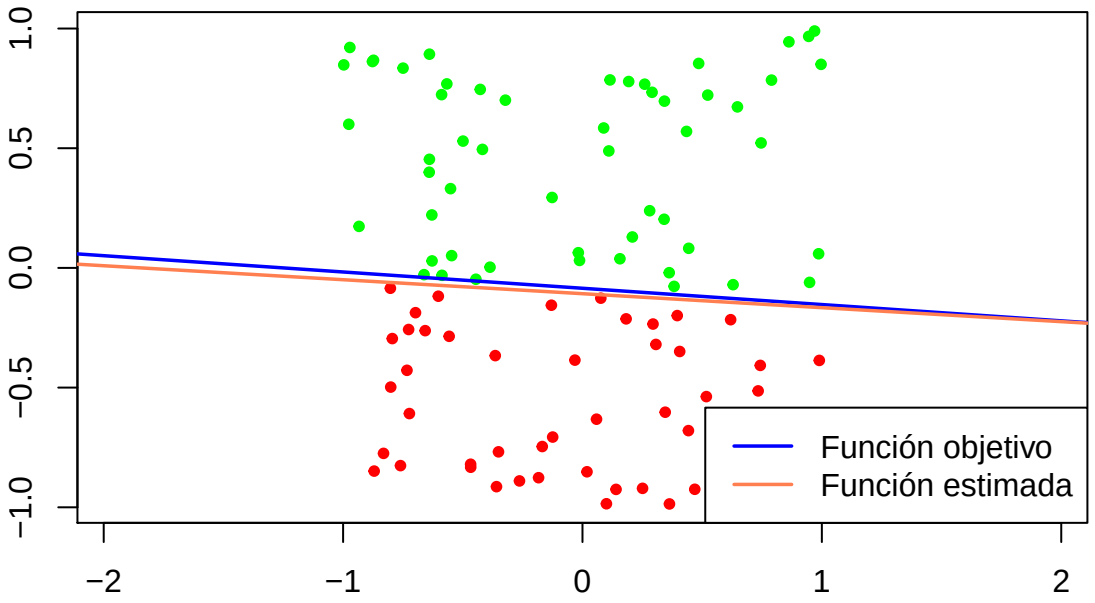
```r
distance <- function(x,y){
  return(sqrt(sum((x-y)^2)))
}

RL.SGD <- function(datos, etiquetas, w_0 = c(0,0), eta = 0.01, tol= 0.01){
  iteraciones <- 0
  n <- length(etiquetas)



  # Definimos w 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
  datos <- cbind(datos, 1)

  while(iteraciones == 0 || distance(w.prev, w.curr) > tol){
    iteraciones <- iteraciones+1
```

```
    w.prev <- w.curr

    # Tomamos una permutación de los datos
    indices <- sample(length(etiquetas))

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

      g_t <- -(etiq*dato)/(1+exp(etiq*w.curr%*%dato))
      w.curr <- w.curr - eta*g_t
    }

    iteraciones <- iteraciones + 1
  }

  return(list("Pesos"=w.curr, "Recta" = -c(w.curr[1], w.curr[3]) / w.curr[2], "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
regLog.res <- RL.SGD(regLog.datos, regLog.etiquetas)

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

# Función objetivo
abline(rev(regLog.frontera), col="blue", lwd=1.75)
# 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} usando una muestra de test de tamaño 10000, contando el número de puntos mal etiquetados:

```
# Definimos la función estimada a partir del resultado obtenido
regLog.g <- function(x, y){
  return(y - regLog.res$Recta[1]*x - regLog.res$Recta[2])
}

# 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 las x en la primera columna y las y en la segunda
  return(matrix(unlist(datos.muestra), ncol = 2, byrow = T))
}


# Generamos una nueva muestra
regLog.test <- generar_muestra(1000)

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

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

# Calcula el número de muestras mal etiquetadas
E_out <- sum(regLog.g.etiquetas != regLog.test.etiq) / nrow(regLog.test)
```

Apartado 4.c

```
# # medir_Eout <- function(etiquetado.objetivo, etiquetado.estimado){
# medir_Eout <- function(muestra){
#     # Genera un etiquetado aleatorio de la muestra
#     f.etiquetado <- generar_etiquetado(muestra)
#
#     # Estima la recta que mejor aproxima la clasificación usando regresión
#     regresion <- Regress_Lin(muestra, f.etiquetado$Etiquetas)
#     g.recta <- -c(regresion[1], regresion[3]) / regresion[2]
#
#     # Define la función clasificadora estimada
#     g <- function(x,y){
#         y - g.recta[1]*x - g.recta[2]
#     }
#
#     # Generamos una nueva muestra
#     test.muestra <- generar_muestra(1000)
#
#     # Generamos etiquetado con f de la nueva muestra
#     test.etiquetas <- f.etiquetado$Etiquetador(test.muestra[, 1], test.muestra[, 2])
#
#     # Generamos etiquetado con g de la nueva muestra
#     g.etiquetador <- generador_etiquetados(g)
#     g.etiquetas <- g.etiquetador(test.muestra[, 1], test.muestra[, 2])
#
#     # Calcula el número de muestras mal etiquetadas
#     E_out <- sum(g.etiquetas != test.etiquetas)
#
#     # Devuelve el porcentaje de error fuera de la muestra
#     return(E_out / nrow(test.muestra))
# }
#
# ## ----3.7.b - Analisis-----------------------------------------------------
# # Generamos una muestra aleatoria de tamaño 100
# reg.muestra <- generar_muestra(100)
#
# # Definimos el número de repiticiones
# num_rep <- 1000
#
# # Repetimos el experimento num_rep veces y tomamos la media de los valores devueltos
# E_out <- sum(replicate(num_rep, medir_Eout(reg.muestra))) / num_rep
#
# ## ----3.7.b - Salida-------------------------
# mensaje <- paste("EJERCICIO 3.7.b: Error medio fuera de la muestra tras", num_rep, "repeticiones:", E
# print(mensaje)
```

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:

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

```
## Warning in scan(file, what, nmax, sep, dec, quote, skip, nlines,
## na.strings, : número de items leídos no es múltiplo del número de columnas
```

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

```
## Warning in scan(file, what, nmax, sep, dec, quote, skip, nlines,
## na.strings, : número de items leídos no es múltiplo del número de columnas
```

```r
# 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 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:

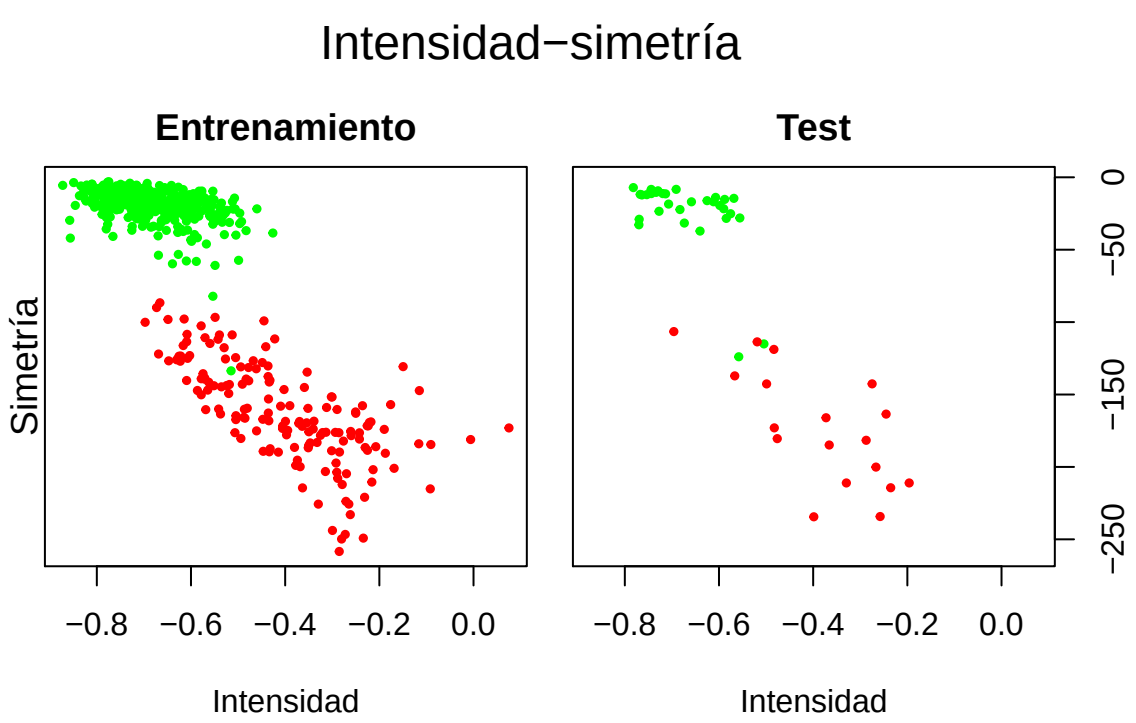
```r
# Análisis de la intensidad y la simetría de los dígitos.
analisis_digito <- function(digito){
  digito <- as.numeric(digito)
  numero <- matrix(digito[-1], nrow = 16)
  numero.inv <- numero[, ncol(numero):1]

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

  c(head(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:

```r
# 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))

# 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)
```

```r
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.analisis.train[, 2:3], col = colores.train, pch = 20, cex=0.75,
     main="Entrenamiento", xlab="Intensidad", yaxt="n", xlim = limX, ylim = limY)

title(ylab="Simetría", mgp=c(1,1,10), line=0, cex.lab=1.2)

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)

axis(4)
mtext("", side=4, line=3)

mtext("Intensidad-simetría", outer=TRUE, line=0.5, cex=1.5)
```




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

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

```r
# Necesario para llamar a la función ginv, que calcula la pseudo-inversa
library(MASS)

# Calcula regresión lineal
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 inversa de X^tX
  XtX_inv <- V %*% ginv(D ** 2) %*% t(V)
  X_pseudoinv <- XtX_inv %*% t(X)

  # Devolvemos la solución
  return(X_pseudoinv %*% 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
  # 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
  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

  # 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 = -c(w[1], w[3]) / w[2], Iter = iteraciones))
}
```

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

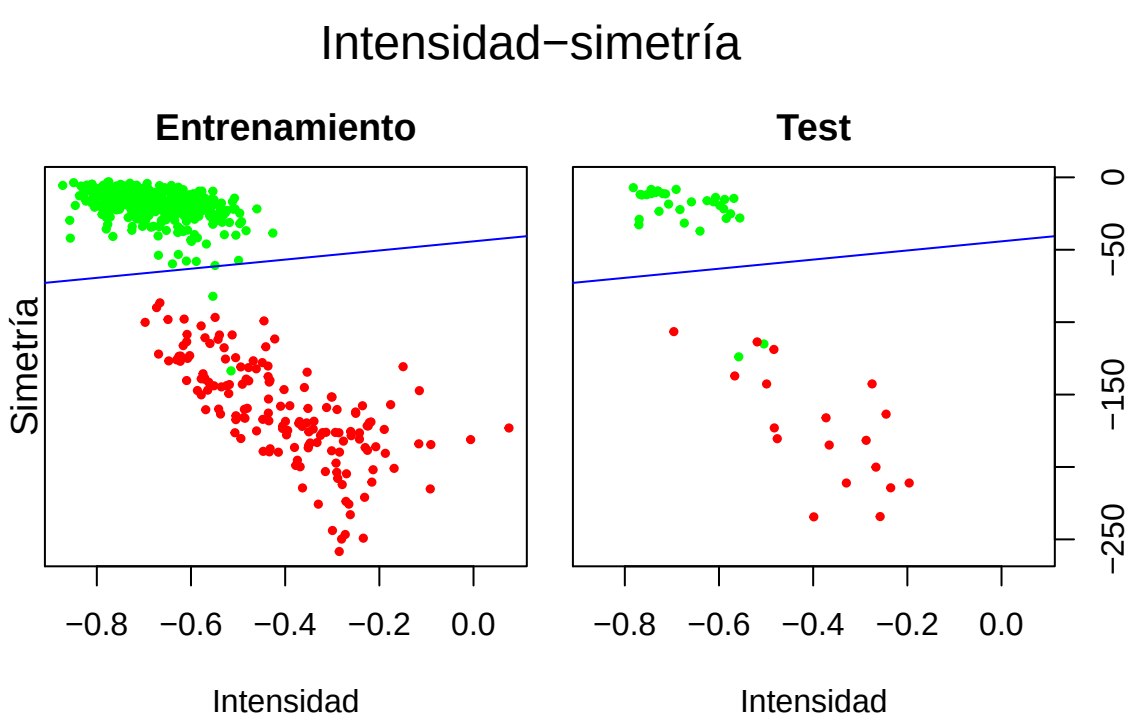
```
# Preparamos los datos
digitos.train.data <- digitos.analisis.train[,2:3]
digitos.train.etiq <- ifelse(digitos.analisis.train[,1] == 1, 1, -1)

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)

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.analisis.train[, 2:3], 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")

title(ylab="Simetría", mgp=c(1,1,10), line=0, cex.lab=1.2)

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)
abline(rev(digitos.pla$Recta), col="blue")

axis(4)
mtext("", side=4, line=3)

mtext("Intensidad-simetría", outer=TRUE, line=0.5, cex=1.5)
```




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

Apartado 5.b

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

```
digitos.g <- function(x, y){
  return(y - digitos.pla$Recta[1]*x - digitos.pla$Recta[2])
}

digitos.g.etiquetado <- generador_etiquetados(digitos.g)
```

```
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])


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%.

Sobreajuste

Ejercicio 1 - Sobreajuste

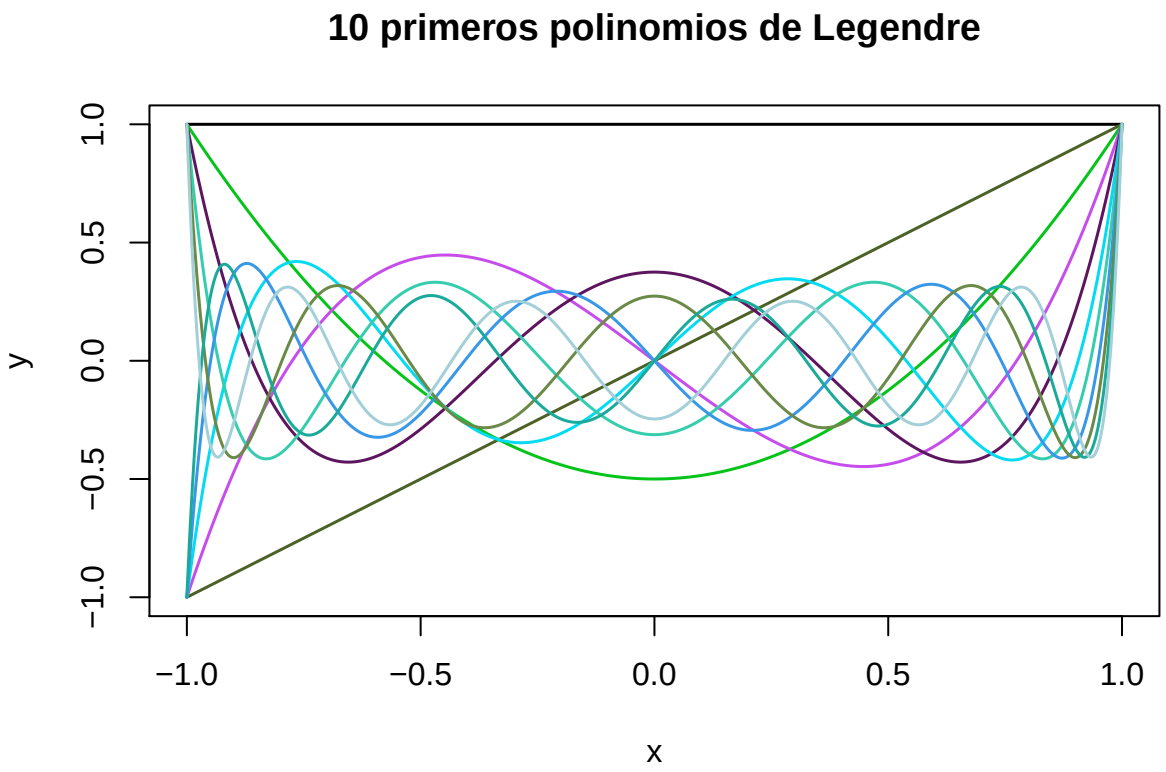
Lo primero que vamos a hacer es definir una función para calcular el polinomio de Legendre de orden k en el punto x. Se han implementado dos versiones: una recursiva, más compacta y elegante, y una iterativa, mucho más eficiente, y con la que trabajaremos a lo largo de este ejercicio. Veamos el código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

```
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:

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

Generamos los coeficientes a_q:

```
# 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
```

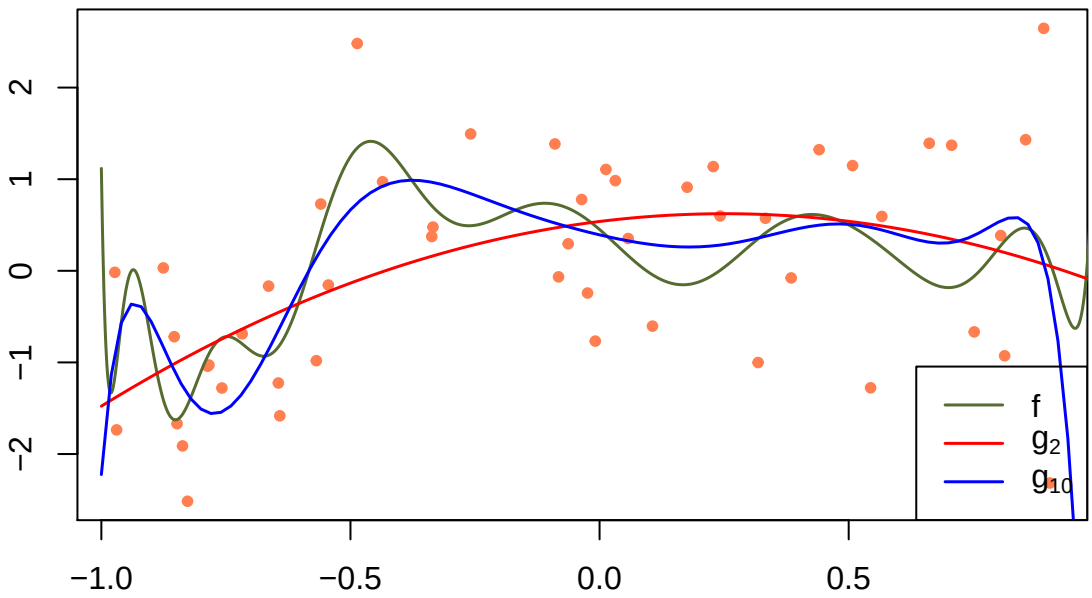
Apartado 1.a

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 $E_{out}(g_2) \approx 0.2459364$ y $E_{out}(g_{10}) \approx 0.5115419$.

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 $E_{out}(g_2) = 0.3458799$ y, con g_{10}, de $E_{out}(g_{10}) = 1.2471385$. 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(N, d+1, sd = 1)

# Definimos las etiquetas
reg.Y <- sapply(1:N, function(i){ reg.W[i,] %*% 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 <- t(cbind(X,1))
  Zt <- t(Z)
  lambdaI <- diag(lambda, ncol(Z))

  # Calculamos la solución
  w <- t(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 y calculamos, para cada uno de ellos, los `N` errores con la técnica del *leave-one-out*: