Práctica 2

Anabel Gómez Ríos

Funciones de la práctica 1 que vamos a reutilizar.

```
set.seed(237)

######
# Funciones de la primera práctica
######

# Simular números aleatorios uniformes
simula_unif = function (N=2, dims=2, rango = c(0,1)){
 matrix(runif(N*dims, min=rango[1], max=rango[2]), nrow = N, ncol=dims, byrow=T)
}

# Simular números aleatorios en una normal
simula_gauss <- function(N, dim, sigma) {
  lapply(1:N, function(x) rnorm(dim, mean = 0, sqrt(sigma)))
}

# Simular recta que corte a un intervalo dado
simula_recta <- function(intervalo) {
  m <- simula_unif(2, 2, intervalo)
  a <- (m[2,2] - m[1,2]) / (m[2,1] - m[1,1])
  b <- m[1,2] - a * m[1,1]
  c(a,b)
}

# Calcular la simetría de una matriz
calcular_simetria <- function(mat) {
  # Invertimos la matriz por columnas
  mat_invertida = apply(mat, 2, function(x) rev(x))
  # Calulamos el valor absoluto de la diferencia de cada elemento entre las dos
  # matrices
  dif = abs(mat - mat_invertida)
  # Sumamos los elementos de la matriz
  suma <- sum(dif)
  # Devolvemos el signo cambiado de la suma
  -suma
}
```

```
# Método de regresión lineal
Regress_Lin <- function(datos, label) {
  descomp <- La.svd(datos)
  vt <- descomp[[3]]
  # Creamos la inversa de la matriz diagonal al cuadrado
  diagonal <- matrix(0, length(descomp[[1]]), length(descomp[[1]]))
  for (i in 1:length(descomp[[1]])) {
    diagonal[i,i] = descomp[[1]][i]
```

```
    if (diagonal[i,i] != 0) {
      diagonal[i,i] = 1/(diagonal[i,i]^2)
    }
  }
  prod_inv <- t(vt) %*% diagonal %*% vt
  pseud_inv <- prod_inv %*% t(datos)
  w <- pseud_inv %*% label
  return(w)
}

# Función para contar diferencias dados dos vectores necesaria en la siguiente
# función
cuenta_diferencias <- function(etiquetas1, etiquetas2) {
  vf <- etiquetas1 == etiquetas2
  length(vf[vf == FALSE])
}

# Función para contar errores necesaria en el PLA pocket
cuenta_errores <- function(w, etiquetas_originales, datos) {
  # Etiquetamos con la solución del PLA
  etiquetas_cambiadas <- unlist(lapply(1:nrow(datos), function(i) {
    # Obtenemos los puntos uno a uno y los etiquetamos
    p <- datos[i,]
    f <- crossprod(w,p)
    sign(f)
  }))
  # Devolvemos el número de errores que da la solución
  cuenta_diferencias(etiquetas_originales, etiquetas_cambiadas)
}

# Algoritmo PLA pocket
ajusta_PLA_MOD <- function(datos, label, max_iter, vini) {
  parada <- F
  fin <- F
  w <- vini
  wmejor <- w
  iter <- 1
  errores_mejor <- cuenta_errores(wmejor, label, datos)
  # Mientras no hayamos superado el máximo de iteraciones o
  # no se haya encontrado solución
  while(!parada) {
    # iteramos sobre los datos
    for (j in 1:nrow(datos)) {
      if (sign(crossprod(w, datos[j,])) != label[j]) {
        w <- w + label[j]*datos[j,]
        # La variable fin controla si se ha entrado en el if
        fin <- F
      }
    }
    # Contamos el número de errores que hay en la solución actual y si
    # es menor que el número de errores en la mejor solución de las que
    # llevamos, nos quedamos con la actual
    errores_actual <- cuenta_errores(w, label, datos)
```

```
    if(errores_actual < errores_mejor) {
      wmejor <- w
      errores_mejor <- errores_actual
    }
    # Si no se ha entrado en el if, todos los datos estaban bien
    # clasificados y podemos poner a TRUE la variable parada.
    if(fin == T) {
      parada = T
    }
    else {
      fin = T
    }
    iter <- iter + 1
    if (iter >= max_iter) parada = T
  }

  # Devolvemos el hiperplano, el número máximo de iteraciones al que hemos
  # llegado y el número de errores de la mejor solución que hemos encontrado
  list(w = wmejor, numIteraciones = iter, errores = errores_mejor)
}
```

```
# Función para pintar datos junto con gráficas
pinta_particion <- function(coordX, coordY, etiquetas=NULL, visible=FALSE,
                            f=NULL, main="", xlab="Eje X", ylab="Eje Y",
                            rango=1) {
  if(is.null(etiquetas))
    etiquetas=1
  else etiquetas = etiquetas+3

  plot(coordX, coordY, type = "p", col = etiquetas, xlab = xlab, ylab = ylab,
       main = main)

  # Si queremos pintar función junto con los datos
  if(visible) {
    sec <- seq(-rango, rango, length.out = 1500)
    z <- outer(seq(-1,1,length.out=1000), sec, f)
    contour(seq(-1,1,length.out=1000), sec, z, col = "blue", levels = 0, add = TRUE, drawlabels = F)
  }
}
```

1. MODELOS LINEALES

Todos los algoritmos dentro de este apartado tienen los siguientes criterios de parada (a no ser que se especifiquen otros en el enunciado): que se llegue a un número máximo de iteraciones prefijado, que la función de error en algún w calculado esté por debajo de un umbral (teniendo en cuenta que las funciones dadas tienen los mínimos en cero) o que la diferencia de dos valores consecutivos de w calculados esté por debajo del mismo umbral anterior.

1. Gradiente Descendente. Implementar el algoritmo de gradiente descendente.

Lo he implementado de forma que se le pueda pasar la función de error y la función gradiente como funciones, de forma que se evalúan dentro de la función con los valores w que se van calculando. Hay que pasarle el punto inicial, la tasa de aprendizaje, el número máximo de iteraciones a realizar y el umbral para las otras dos condiciones de parada.

```
# Algoritmo del gradiente descendente. Le pasamos a la función la función de
# error, su gradiente (que serán funciones), el punto en el que se empieza, la
# tasa de aprendizaje, el número máximo de iteraciones a realizar, y el mínimo
# error al que queremos llegar, en orden.
# Devuelve los valores de la función de error por los que pasa junto con la
# iteración.
gradienteDescendente <- function(ferror, gradiente, pini, tasa, maxiter, umbral) {
  w <- pini
  i <- 1
  valoresError <- c(i, ferror(pini[1], pini[2]))
  mejora <- TRUE
  while (i <= maxiter && mejora) {
    g <- gradiente(w[1], w[2])
    # Le cambiamos la dirección al gradiente para ir hacia abajo
    v <- -g
    # Nos movemos tanto como indique la tasa
    wnew <- w + tasa*v
    valoresError <- rbind(valoresError, c(i, ferror(wnew[1], wnew[2])))

    if (abs(ferror(wnew[1], wnew[2])-ferror(w[1], w[2])) < umbral ||
        ferror(wnew[1], wnew[2]) < umbral || i==maxiter) {
      mejora <- FALSE
      cat("He necesitado", i, "iteraciones para llegar al error",
          ferror(wnew[1], wnew[2]),"\n")
      cat("con valores de u y v:", wnew[1],",", wnew[2])
      mostrar <- FALSE
    }
    w <- wnew
    i <- i+1
  }
  return(valoresError)
}
```

a) Considerar la función no lineal de error $E(u,v) = (ue^v - 2ve^{-u})^2$. Usar gradiente descendente y minimizar esta función de error, comenzando desde el punto $(u,v) = (1,1)$ y usando una tasa de aprendizaje $\eta = 0.1$

1) Calcular analíticamente y mostrar la expresión del gradiente de la función $E(u,v)$

Calculamos el gradiente de $E(u,v)$: $\nabla E(u,v) = (\frac{\partial E}{\partial u}, \frac{\partial E}{\partial v}) = (2(ue^v - 2ve^{-u})(e^v + 2ve^{-u}), 2(ue^v - 2e^{-u})(ue^v - 2e^{-u})) = 2(ue^v - 2ve^{-u})(e^v + 2ve^{-u}, ue^v - 2e^{-u})$

2) ¿Cuántas iteraciones tarda el algoritmo en obtener por primera vez un valor de $E(u,v)$ inferior a 10^{-14}? (Usar flotantes de 64 bits)

```
E <- function(u,v) (u*exp(v) - 2*v*exp(-u))^2
gradE <- function(u,v) {(2*(u*exp(v) - 2*v*exp(-u)))*c(exp(v) + 2*v*exp(-u),
                                                       u*exp(v) - 2*exp(-u))}

val <- gradienteDescendente(E, gradE, c(1,1), 0.1, 20, 10^{-14})
```

```
## He necesitado 10 iteraciones para llegar al error 1.208683e-15
## con valores de u y v: 0.04473629 , 0.02395871
```

3) ¿Qué valores de (u,v) obtuvo en el apartado anterior cuando alcanzó el error de 10^{-14}

Como podemos ver en la salida por pantalla anterior, el valor de u ha sido 0.04473629 y el de v ha sido 0.02395871

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

1) Usar gradiente descendente para minimizar esta función. Usar como valores iniciales $x_0 = 1$, $y_0 = 1$, la tasa de aprendizaje $\eta = 0.01$ y un máximo de 50 iteraciones. Generar un gráfico de cómo desciende el valor de la función con las iteraciones. Repetir el experimento pero usando $\eta = 0.1$. Comentar las diferencias.

Calculamos primero su gradiente: $\nabla f = (2x + 4\pi * sin(2\pi y) * cos(2\pi x), 4y + 4\pi * sin(2\pi x) * cos(2\pi y))$

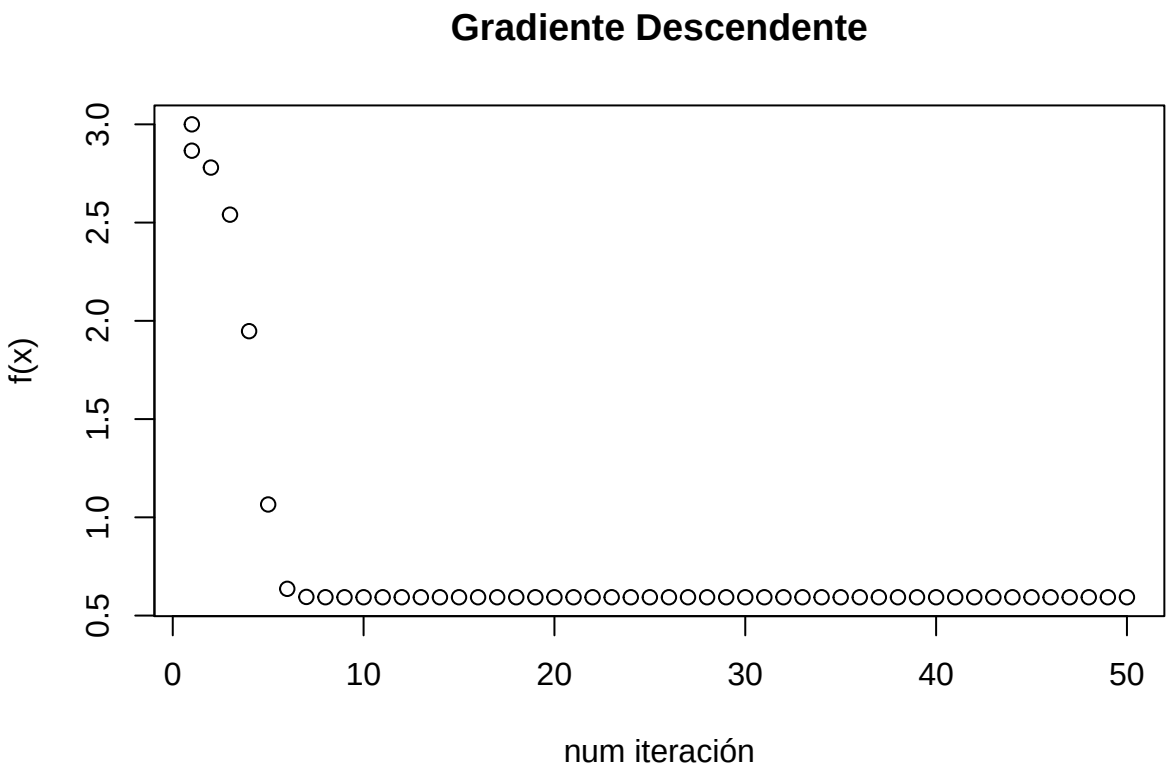
Le pasamos la función f y su grafiente al método de gradiente descendente y pintamos por dónde va pasando en las iteraciones, que es algo que nos devuelve el método:

```
f <- function(x,y) x^2 + 2*y^2 + 2*sin(2*pi*x)*sin(2*pi*y)
gradF <- function(x,y) c(2*x + 4*pi*sin(2*pi*y)*cos(2*pi*x),
                         4*y + 4*pi*sin(2*pi*x)*cos(2*pi*y))

val <- gradienteDescendente(f, gradF, c(1,1), 0.01, 50, 0)
```

```
## He necesitado 50 iteraciones para llegar al error 0.5932694
## con valores de u y v: 1.21807 , 0.712812
```

```
pinta_particion(val[,1], val[,2], xlab="num iteración", ylab="f(x)",
                main="Gradiente Descendente")
```

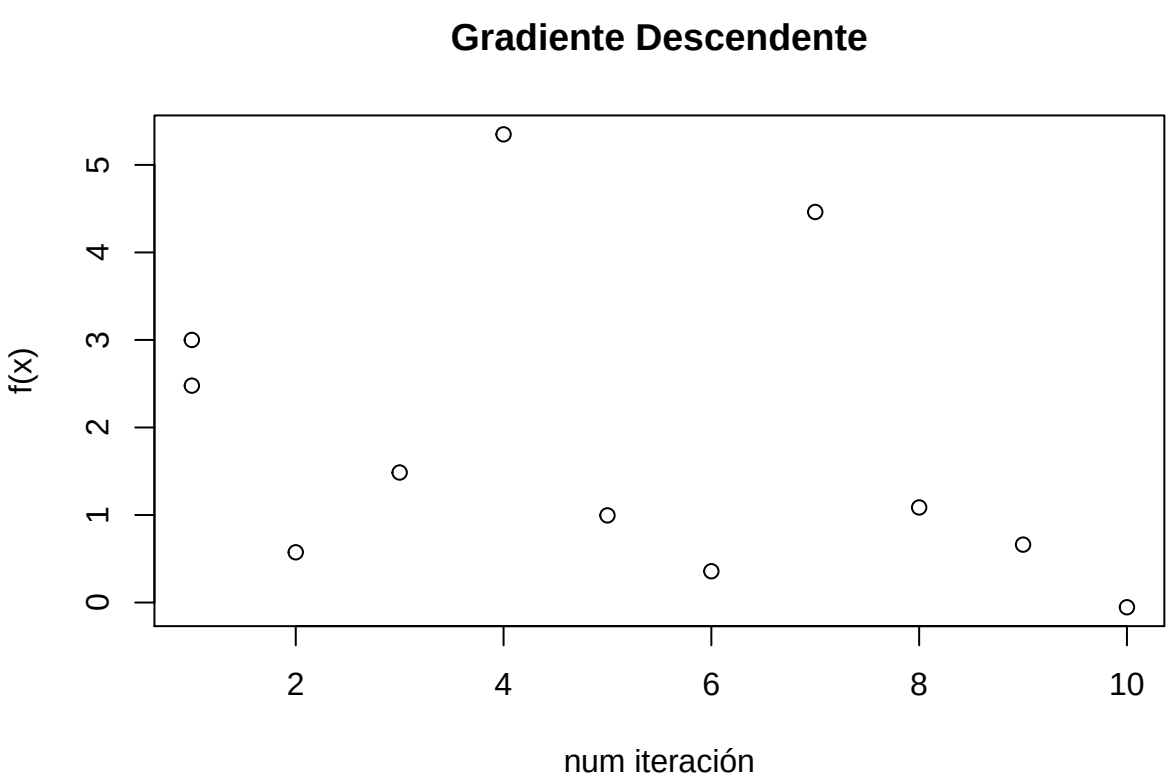


Repetimos con $\eta = 0.1$:

```
val <- gradienteDescendente(f, gradF, c(1,1), 0.1, 50, 0)
```

```
## He necesitado 10 iteraciones para llegar al error -0.05388005
## con valores de u y v: 0.3881225 , 0.6516421
```

```
pinta_particion(val[,1], val[,2], xlab="num iteración", ylab="f(x)",
                main="Gradiente Descendente")
```



Como vemos, las diferencias son enormes puesto que para el caso $\eta = 0.1$ el método no converge, mientras que para $\eta = 0.01$ converge muy pronto a un valor muy bajo (0.5). Esto nos da una idea de lo importante que es elegir bien la tasa de aprendizaje.

2) Obtener el valor mínimo y los valores de las variables que lo alcanzan cuando el punto de inicio se fija: (0.1,0.1), (1,1), (-0.5, -0.5), (-1, -1). Generar una tabla con los valores obtenidos. ¿Cuál sería su conclusión sobre la verdadera dificultad de encontrar el mínimo global de una función arbitraria?

Utilizamos la tasa de aprendizaje $\eta = 0.01$, ya que hemos visto que con la otra no converge.

```
val <- gradienteDescendente(f, gradF, c(0.1,0.1), 0.01, 50, 0)
```

```
## He necesitado 3 iteraciones para llegar al error -0.0009552139
## con valores de u y v: 0.005266095 , -0.002392069
```

```
val <- gradienteDescendente(f, gradF, c(1,1), 0.01, 50, 0)
```

```
## He necesitado 50 iteraciones para llegar al error 0.5932694
## con valores de u y v: 1.21807 , 0.712812
```

```
val <- gradienteDescendente(f, gradF, c(-0.5,-0.5), 0.01, 50, 0)
```

```
## He necesitado 5 iteraciones para llegar al error -0.01244002
## con valores de u y v: -0.5803234 , -0.3839919
```

```
val <- gradienteDescendente(f, gradF, c(-1,-1), 0.01, 50, 0)
```

```
## He necesitado 50 iteraciones para llegar al error 0.5932694
## con valores de u y v: -1.21807 , -0.712812
```

El problema de encontrar un mínimo global con esta técnica de gradiente descendente es que se queda estancada en mínimos locales. Esto significa que que encontremos el mínimo global o no depende únicamente del punto de inicio que le damos: si éste está lo suficientemente cerca del mínimo global como para que no haya mínimos locales entre ambos, lo encontrará, pero de haber algún mínimo local, se quedará en el primero que encuentre y será incapaz de salir de ahí.

2. Coordenada descendente. En este ejercicio comparamos la eficiencia de la técnica de optimización de "coordenada descendente" usando la misma función del ejercicio 1.1.a. En cada iteración, tenemos dos pasos a lo largo de dos coordenadas. En el paso 1 nos movemos a lo largo de la coordenadas u para reducir el error (suponer que se verifica una aproximación de primer orden como en gradiente descendente), y el paso 2 es para reevaluar y movernos a lo largo de la coordenada v para reducir el error (hacer la misma hipótesis que en el paso 1). Usar una tasa de aprendizaje de $\eta = 0.1$.

De nuevo, tenemos que pasarle al método la función de error y la gradiente, como funciones, para poder evaluarlas dentro.

```
# Algoritmo de coordenada descendente. Le pasamos a la función la función de
# error, su gradiente (que serán funciones), el punto en el que se empieza, la
# tasa de aprendizaje, el número máximo de iteraciones a realizar, y el mínimo
# error al que queremos llegar, en orden.
coordenadaDescendente <- function(ferror, gradiente, pini, tasa, maxiter, umbral) {
  w <- pini
  i <- 1
  mejora <- TRUE
  while (i <= maxiter && mejora) {
    # Paso 1
    g <- gradiente(w[1], w[2])
    # Le cambiamos la dirección al gradiente para ir hacia abajo
    v <- -g
    wnew <- w
    wnew[1] <- w[1] + tasa*v[1]

    # Paso 2
    g <- gradiente(wnew[1], wnew[2])
    # Le cambiamos la dirección al gradiente para ir hacia abajo
    v <- -g
    wnew[2] <- w[2] + tasa*v[2]

    if (abs(ferror(wnew[1],wnew[2]) - ferror(w[1],w[2])) < umbral &&
        ferror(wnew[1],wnew[2]) < umbral || i==maxiter) {
      mejora <- FALSE
      cat("He necesitado", i, "iteraciones para llegar al error",
          ferror(wnew[1], wnew[2]),"\n")
      cat("con valores de u y v:", w[1],",", w[2])
```

```
    }

    w <- wnew
    i <- i+1
  }
}
```

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

```
val <- coordenadaDescendente(E, gradE, c(1,1), 0.1, 15, 0)
```

```
## He necesitado 15 iteraciones para llegar al error 0.1398138
## con valores de u y v: 6.300845 , -2.823852
```

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

Como vemos, es mejor la técnica del gradiente descendente, ya que éste, para la misma función, con la misma tasa de aprendizaje y con el mismo punto inicial necesita 10 iteraciones para obtener un error de $1.2 * 10^{-15}$ (hecho en el apartado anterior) mientras que ésta en 15 iteraciones sólo llega a un error de 0.1398138.

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

En este caso, además de la función de error y la gradiente, tenemos que pasarle la función hessiana, también para que se pueda evaluar dentro del método. En el método de Newton cómo nos vemos lo da el producto matricial entre la hessiana (que será una matriz 2x2) y el gradiente (que será un vector 2x1), de forma que obtenemos un vector 2x1, como en los métodos anteriores.

```
# Algoritmo del método de Newton. Le pasamos a la función la función de
# error, su gradiente y la matriz hessiana (que serán funciones), el punto
# en el que se empieza, la tasa de aprendizaje, el número máximo de iteraciones
# a realizar, y el mínimo error al que queremos llegar, en orden.
# Devuelve los valores de la función de error por los que pasa junto con la
# iteración.
metodoNewton <- function(ferror, gradiente, hessiana, pini, tasa, maxiter, umbral) {
  w <- pini
  i <- 1
  valoresError <- c(i, ferror(pini[1], pini[2]))
  mejora <- TRUE
  while (i <= maxiter && mejora) {
    hg <- solve(hessiana(w[1], w[2]))%*%gradiente(w[1], w[2])
    # Le cambiamos la dirección a la hessiana por el gradiente para ir hacia abajo
    v <- -hg
    # Nos movemos tanto como indique la tasa
    wnew <- w + tasa*v
    # Vamos guardando los valores de error por los que vamos pasando
    valoresError <- rbind(valoresError, c(i, ferror(wnew[1], wnew[2])))
```

```
    if (abs(ferror(wnew[1], wnew[2]) - ferror(w[1], w[2])) < umbral ||
        ferror(wnew[1], wnew[2]) < umbral || i==maxiter) {
      mejora <- FALSE
      cat("He necesitado", i, "iteraciones para llegar al error",
          ferror(wnew[1], wnew[2]),"\n")
      cat("con valores de u y v:", wnew[1],",", wnew[2])
    }

    w <- wnew
    i <- i+1
  }
  return(valoresError)
}
```

Calculamos la matriz hessiana de la f. Recordemos que las derivadas parciales cruzadas (de existir y ser continuas, como es nuestro caso) son iguales, por el teorema de Schwarz.

```
f <- function(x,y) x^2 + 2*y^2 + 2*sin(2*pi*x)*sin(2*pi*y)
gradF <- function(x,y) c(2*x + 4*pi*sin(2*pi*y)*cos(2*pi*x),
                         4*y + 4*pi*sin(2*pi*x)*cos(2*pi*y))
d12 <- function(x,y) 8*pi^2*cos(2*pi*y)*cos(2*pi*x)
d11 <- function(x,y) 2 - 8*pi^2*sin(2*pi*y)*sin(2*pi*x)
d22 <- function(x,y) 4 - 8*pi^2*sin(2*pi*x)*sin(2*pi*y)

hess <- function(x,y) rbind(c(d11(x,y), d12(x,y)), c(d12(x,y), d22(x,y)))

val <- metodoNewton(f, gradF, hess, c(1,1), 0.01, 50, 0)
```

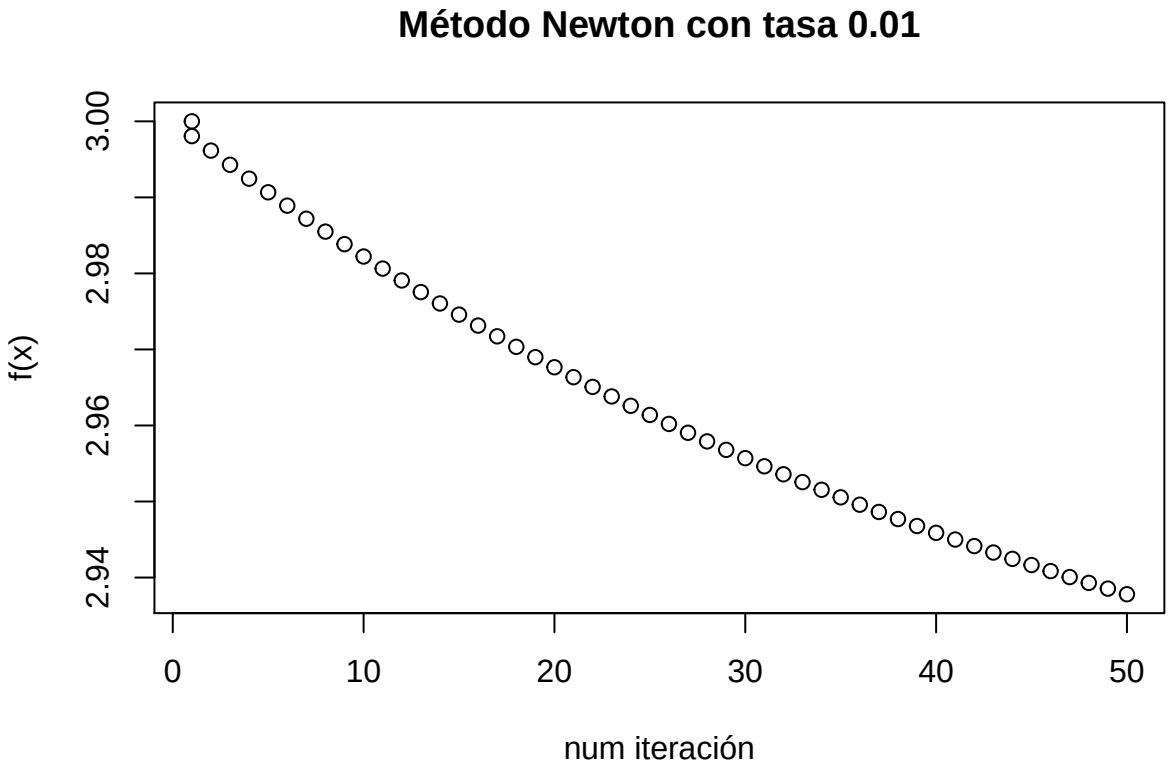
```
## He necesitado 50 iteraciones para llegar al error 2.937804
## con valores de u y v: 0.9803919 , 0.9904148
```

```
val2 <- metodoNewton(f, gradF, hess, c(1,1), 0.1, 50, 0)
```

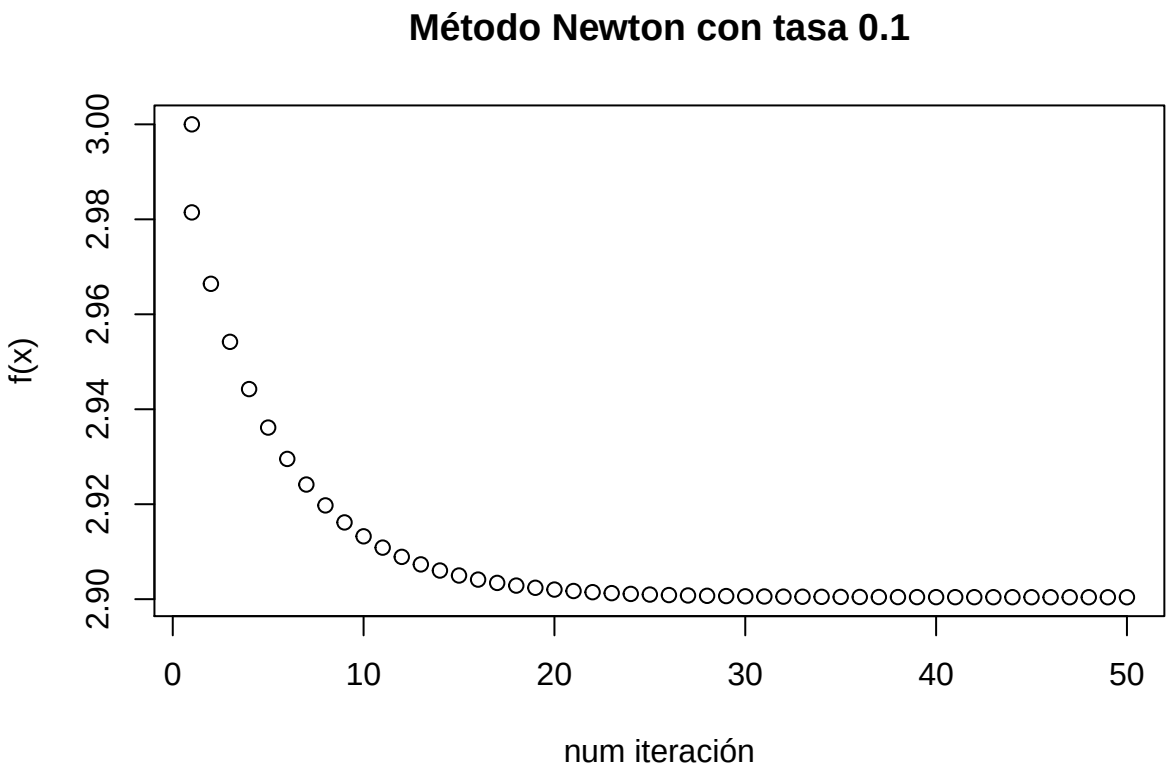
```
## He necesitado 50 iteraciones para llegar al error 2.900408
## con valores de u y v: 0.9494086 , 0.9747153
```

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

```
pinta_particion(val[,1], val[,2], xlab="num iteración", ylab="f(x)",
                main="Método Newton con tasa 0.01")
```




```
pinta_particion(val2[,1], val2[,2], xlab="num iteración", ylab="f(x)",
                main="Método Newton con tasa 0.1")
```



b) Extraer conclusiones sobre las conductas de los algoritmos comparando la curva de decrecimiento de la función calculada en el apartado anterior y la correspondiente obtenida con gradiente descendente.

Como vemos, para tasa 0.01, el método de gradiente descendente converge mucho más rápido a un error bastante más pequeño (0.5 frente a 2.94). En contraposición, con tasa de aprendizaje 0.1, el método de gradiente descendente no converge mientras que el de Newton sí, a error 2.90. Aun así, creo que es mejor el método de gradiente descendente, sólo hay que probar y ajustar bien la tasa de aprendizaje, pero converge más rápido y mejor.

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

Consideremos $d = 2$ para que los datos sean visualizables, y sea $X = [-1, 1] \times [-1, 1]$ con probabilidad uniforme de elegir cada $x \in X$. Elegir una línea en el plano como la frontera entre $f(x) = 1$ (donde y toma valores +1) y $f(x) = 0$ (donde y toma valores -1), para ello seleccionar dos puntos aleatorios del plano y calcular la línea que pasa por ambos. Seleccionar $N = 100$ puntos aleatorios $\{x_n\}$ de X y evaluar las respuestas de todos ellos $\{y_n\}$ respecto de la frontera elegida.

Generamos la muestra de 100 puntos de manera uniforme:

```
muestra <- simula_unif(100, 2, c(-1,1))
```

Generamos la recta en el cuadrado dado que separa los datos:

```
recta <- simula_recta(c(-1,1))
```

Los etiquetamos con respecto a la recta que acabamos de generar:

```
etiquetas <- unlist(lapply(1:nrow(muestra), function(i) {
  p <- muestra[i,]
  sign(p[2] - recta[1]*p[1] - recta[2])
}))
```

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

1. Inicializar el vector de pesos con valores 0.
2. Parar el algoritmo cuando $||w^{(t-1)} - w^{(t)}|| < 0.01$, donde $w^{(t)}$ denota el vector de pesos al final de la época t. Una época es un pase completo a través de los N datos.
3. Aplicar una permutación aleatoria de 1,2,...,N a los datos antes de usarlos en cada época del algoritmo.
4. Usar una tasa de aprendizaje de $\eta = 0.01$.

Vamos a hacer primero una función auxiliar para calcular la norma euclídea de un vector:

```
# Función para calcular la norma euclídea de un vector
calcularNorma <- function(vec) {
  sqrt(sum(vec^2))
}
```

Implementamos ahora regresión logística con las condiciones dadas:

```r
# Algoritmo RL con SGD. Recibe los datos en una matriz, las etiquetas de esos
# datos, el vector inicial, la tasa de aprendizaje, el máximo número de
# iteraciones y el umbral para la condición de parada.
Regress_LogSGD <- function(datos, label, vini, tasa, maxiter, umbral) {
  parada <- F
  w <- vini
  iter <- 0
  # Mientras no hayamos superado el máximo de iteraciones o
  # no se hayan acercado lo suficiente w y wnew
  while(!parada && iter < maxiter) {
    # Hacemos una permutación al orden en el que vamos a utilizar los datos
    pos <- sample(1:nrow(datos), nrow(datos))
    # iteramos sobre los datos
    wold <- w
    for (j in pos) {
      grad <- (-label[j]*datos[j,])/(1+exp(label[j]*crossprod(w, datos[j,])))
      w <- w - tasa*grad
    }
    if(calcularNorma(wold-w) < umbral) {
      parada <- TRUE
    }
    iter <- iter+1
  }

  # Devolvemos los pesos finales
  return(list(pesos=w, iteraciones=iter))
}
```

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

He usado un número de 1000 muestras, las he etiquetado con la verdadera función, he obtenido la g aproximada por regresión lineal y las he vuelto a etiquetar con esta g para poder contar los errores en estas 1000 muestras y poder calcular después el porcentaje de error:

```r
sol <- Regress_LogSGD(cbind(muestra,1), etiquetas, c(0,0,0), 0.01, 400, 0.01)
sol
```

```
## $pesos
## [1] 1.668128 7.792767 5.084174
##
## $iteraciones
## [1] 371
```

```r
g <- -sol[[1]]/sol[[1]][2]
muestra_out <- simula_unif(1000, 2, c(-1,1))
# Etiquetamos con la función original
etiquetas_originales <- unlist(lapply(1:nrow(muestra_out), function(i) {
  p <- muestra_out[i,]
  sign(p[2] - recta[1]*p[1] - recta[2])
}))
# Etiquetamos con la g estimada
```

```r
etiquetas_g <- unlist(lapply(1:nrow(muestra_out), function(i) {
  p <- muestra_out[i,]
  sign(p[2] - g[1]*p[1] - g[3])
}))
# Contamos los errores
errores <- cuenta_diferencias(etiquetas_originales, etiquetas_g)
cat("El error Eout es: ", 100*(errores/nrow(muestra)))
```

```
## El error Eout es:  32
```

c) Repetir el experimento 100 veces con diferentes funciones frontera y calcular el promedio.

Hacemos un bucle de 1 a 100 generando en cada iteración una recta que divida a los datos distinta, estimamos la g con regresión logística y vamos acumulando los errores para hacer la media.

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

```r
errores <- 0
iteraciones <- 0
for(i in 1:100) {
  # Generamos una nueva recta que separe
  recta_i <- simula_recta(c(-1,1))
  # Etiquetamos con esta recta
  etiquetas_i <- unlist(lapply(1:nrow(muestra), function(i) {
    p <- muestra[i,]
    sign(p[2] - recta_i[1]*p[1] - recta_i[2])
  }))

  # Estimamos g
  sol <- Regress_LogSGD(cbind(muestra,1), etiquetas_i, c(0,0,0), 0.01, 500, 0.01)
  g <- -sol[[1]]/sol[[1]][2]
  # Acumulamos las iteraciones que tarda en converger
  iteraciones <- iteraciones + sol[[2]]

  # Calculamos las etiquetas originales y las estimadas fuera de la muestra
  etiquetas_originales <- unlist(lapply(1:nrow(muestra_out), function(i) {
    p <- muestra_out[i,]
    sign(p[2] - recta_i[1]*p[1] - recta_i[2])
  }))
  etiquetas_g <- unlist(lapply(1:nrow(muestra_out), function(i) {
    p <- muestra_out[i,]
    sign(p[2] - g[1]*p[1] - g[3])
  }))
  # Contamos errores y acumulamos
  errores <- errores + cuenta_diferencias(etiquetas_originales, etiquetas_g)
}

cat("El número medio de Eout ha sido:", errores/100)
```

```
## El número medio de Eout ha sido: 24.61
```

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

En el trozo anterior, dentro del bucle, hemos ido acumulando el número de iteraciones que el método tarda en converger para las 100 iteraciones. Sólo nos queda hacer la media, es decir, dividir por 100:

```
cat("RL tarda en converger", iteraciones/100, "iteraciones en promedio")
```

```
## RL tarda en converger 331.41 iteraciones en promedio
```

```
# Borramos lo que no necesitamos
rm(muestra, muestra_out)
rm(errores, iteraciones)
rm(etiquetas, etiquetas_g, etiquetas_originales)
rm(g, i, recta, recta_i)
rm(val, val2, etiquetas_i, sol, hess)
rm(d11, d12, d22, E, f, gradE, gradF)
```

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

(*AVISO*: he tenido que borrar la última fila del fichero de test a mano pues tenía menos datos que los demás y hacía que me diera fallo lo demás).

Comenzamos leyendo los datos tanto de test como de train, quedándonos sólo con las instancias de de unos y cincos y calculando simetrías e intensidad. Formateamos los datos de forma que al final nos quede una lista de matrices para las instancias de test y otra lista de matrices para las instancias de train:

```
# Leemos los ficheros y extraemos los dígitos 1 y 5
train <- read.table("datos/zip.train", sep=" ")
```

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

```
test <- read.table("datos/zip.test", sep=" ")
numero_train <- train$V1
numero_test <- test$V1
frame_train <- train[numero_train==1 | numero_train==5,]
frame_test <- test[numero_test==1 | numero_test==5,]
numero_train <- numero_train[numero_train==1 | numero_train==5]
numero_test <- numero_test[numero_test==1 | numero_test==5]

# Hacemos los vectores de etiquetas
etiquetas_train = numero_train
etiquetas_train[numero_train==5] = -1
etiquetas_test = numero_test
etiquetas_test[numero_test==5] = -1
```

```r
# Eliminamos de cada uno la primera columna, que guarda el número del
# que son los datos, y la última, que tiene NA
frame_train = frame_train[,-258]
frame_train = frame_train[,-1]
frame_test = frame_test[,-258]
frame_test = frame_test[,-1]

# Los pasamos a matrices
frame_train <- data.matrix(frame_train)
frame_test <- data.matrix(frame_test)

# Hacemos una lista de matrices
lista_train <- lapply(split(frame_train, seq(nrow(frame_train))), function(x) {
  matrix(x, 16, 16, T)
})
lista_test <- lapply(split(frame_test, seq(nrow(frame_test))), function(x) {
  matrix(x, 16, 16, T)
})

# Eliminamos lo que no vamos a utilizar
rm(train)
rm(test)
rm(frame_train)
rm(frame_test)
rm(numero_train)
rm(numero_test)
```

A continuación calculamos las intensidades y las simetrías y las metemos todas juntas (las de las instancias de 1 y las de 5, pero dejamos separados siempre los conjuntos de train y test)

```r
# Calculamos primero las intensidades y las simetrías de todas las matrices,
# es decir, de todas las instancias de 1's y 5's
train_simetria <- unlist(lapply(lista_train, function(m) calcular_simetria(m)))
train_intensidad <- unlist(lapply(lista_train, function(m) mean(m)))

test_simetria <- unlist(lapply(lista_test, function(m) calcular_simetria(m)))
test_intensidad <- unlist(lapply(lista_test, function(m) mean(m)))
```

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

Utilizamos primero el algoritmo de regresión lineal para obtener una buena solución inicial para dársela al PLA pocket.
Las etiquetas serán 1 o -1 según representen la intensidad o simetría de las instancias de números 1 o 5, respectivamente.
Necesitamos poner como tercera coordenada de la matriz de datos que le vamos a pasar al método de regresión lineal un 1, para ponerlo en coordenadas homogéneas. Nos quedará por tanto cada fila de la matriz de datos como (intensidad, simetría, 1).

```r
datos <- cbind(train_intensidad, train_simetria, 1)
w <- Regress_Lin(datos, etiquetas_train)
cat(w)
```

```
## 0.7318291 0.01402067 1.714127
```

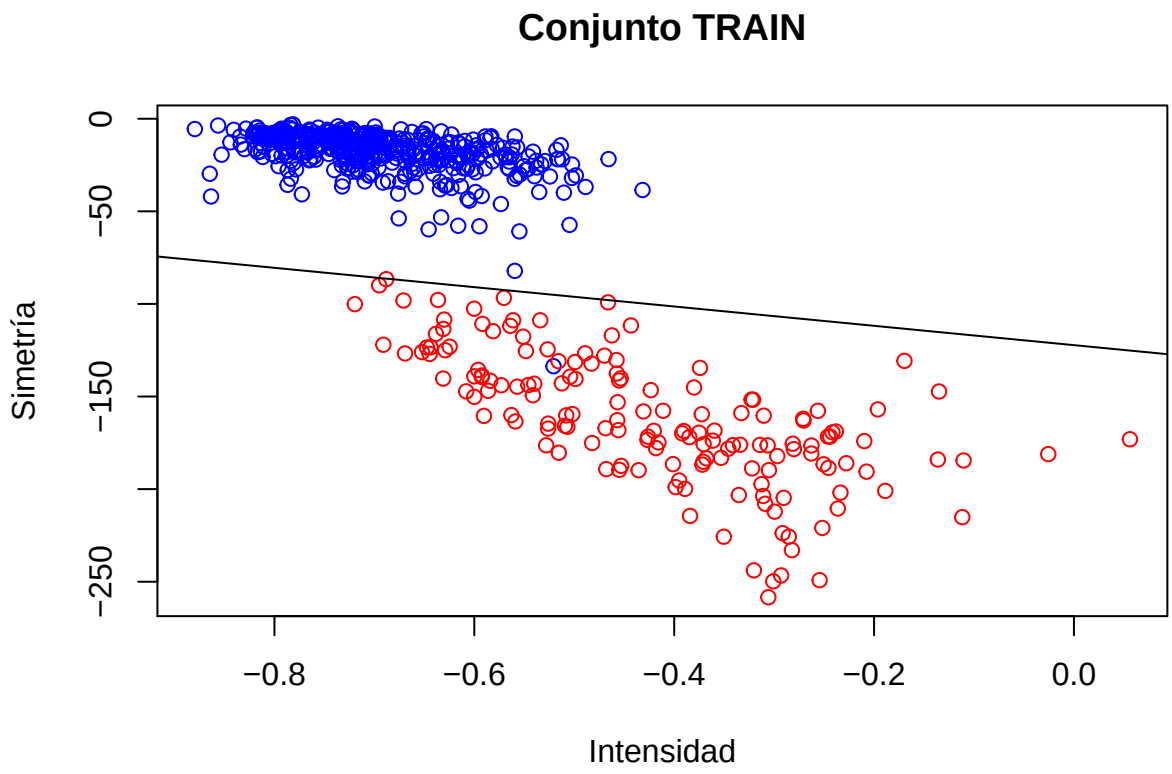
```r
# Le pasamos lo que nos devuelve la regresión lineal al PLA pocket como solución
# inicial
sol <- ajusta_PLA_MOD(datos, etiquetas_train, 100, w)
cat(sol[[1]])
```

```
## 0.7318291 0.01402067 1.714127
```

Como vemos, la solución que nos devuelve la regresión lineal es la misma que la que nos devuelve el PLA pocket, es decir, que el PLA pocket no puede mejorar la solución que da la regresión lineal. Esto es así porque como vemos en la gráfica del siguiente apartado, el único error que hay no hay forma de arreglarlo ya que los datos no son separables y la regresión ya nos está ofreciendo la mejor solución.

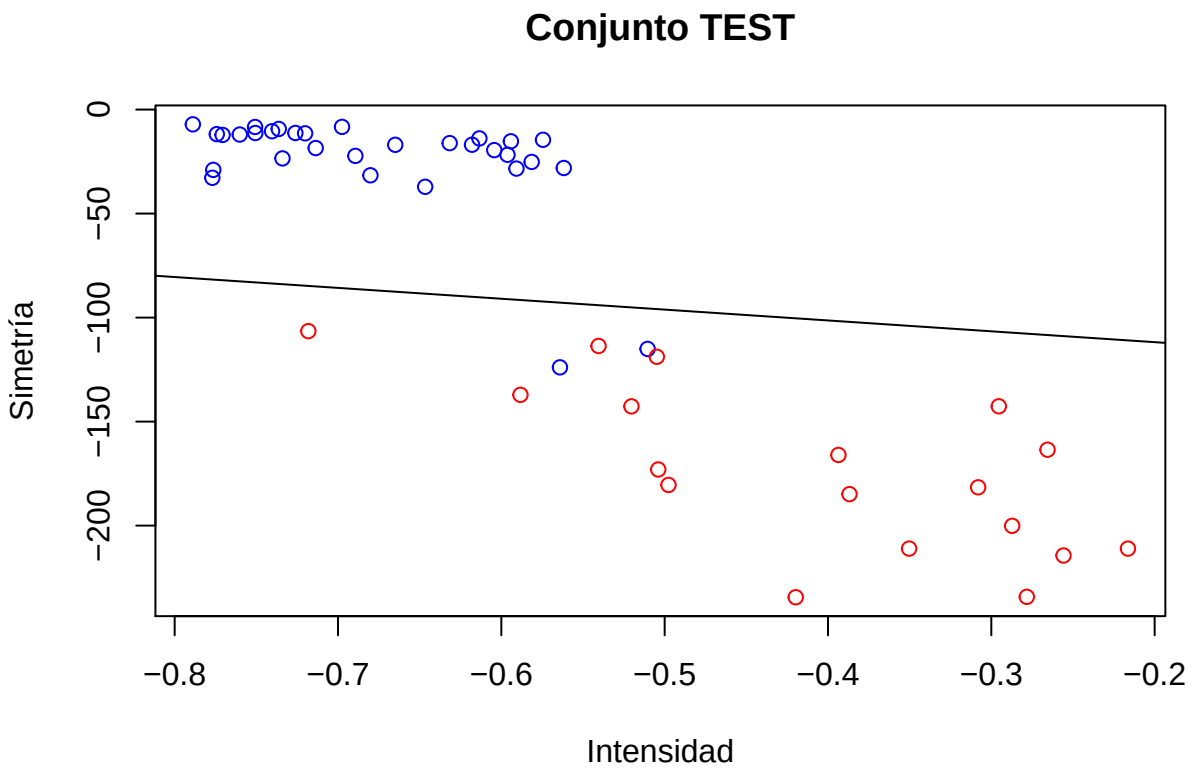
a) Generar gráficos separados (en color) de los datos de entrenamiento y test junto con la función estimada.

```r
# Obtenemos la solución devuelta por el PLA
r <- sol[[1]]
r <- -r/r[2]
plot(train_intensidad, train_simetria, col=etiquetas_train+3, type="p",
     main="Conjunto TRAIN", xlab="Intensidad", ylab="Simetría")
abline(r[3],r[1])
```



Dibujamos y mostramos el conjunto de test con la función que hemos calculado con el conjunto de train

```
plot(test_intensidad, test_simetria, type="p", col=etiquetas_test+3,
     main="Conjunto TEST", xlab="Intensidad", ylab="Simetría")
abline(r[3], r[1])
```



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

E_{in} es el error dentro de la muestra (datos de entrenamiento) y E_{test} el error en los datos de test con la recta que hemos obtenido previamente sobre los datos de entrenamiento. En ambos casos lo que hacemos es calcular las nuevas etiquetas que nos da la g (función obtenida de regresión logística + PLA pocket) y contar los errores de estas con las originales, para después hacer el porcentaje, de forma que son el porcentaje de puntos mal etiquetados dentro de la muestra y el porcentaje de puntos mal etiquetados en los datos de test, respectivamente.

```
etiquetas_g_in <- unlist(lapply(1:nrow(datos), function(i) {
  p <- datos[i,]
  sign(p[2] - r[1]*p[1] - r[3])
}))
datos_test <- cbind(test_intensidad, test_simetria, 1)
etiquetas_g_test <- unlist(lapply(1:nrow(datos_test), function(i) {
  p <- datos_test[i,]
  sign(p[2] - r[1]*p[1] - r[3])
}))
Ein <- 100*cuenta_diferencias(etiquetas_train, etiquetas_g_in)/nrow(datos)
Etest <- 100*cuenta_diferencias(etiquetas_test, etiquetas_g_test)/nrow(datos_test)
cat("El error en la muestra Ein ha sido", Ein)
```

```
## El error en la muestra Ein ha sido 0.1669449
```

```
cat("El error en el test Etest ha sido", Etest)
```

```
## El error en el test Etest ha sido 4.081633
```

c) Obtener cotas sobre el verdadero valor de E_{out}. Pueden calcularse dos cotas, una basada en E_{in} y otra basada en E_{test}. Usar una tolerancia $\delta = 0.05$. ¿Qué cota es mejor?

Empezamos obteniendo la cota basada en E_{in}: la cota de generalización de Vapnik-Chervonenkis para $M = \infty$, como es nuestro caso cuando calculamos la recta que separa los datos para los datos de entrenamiento. Para una tolerancia de $\delta = 0.05$, podemos afirmar que $E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{N} ln \frac{4m_H(2N)}{\delta}}$ con probabilidad mayor o igual $1 - \delta = 0.95$. Tenemos además una cota de $m_H(N)$: $m_H(N) \leq N^{d_{VC}} + 1$, y d_{VC} para este problema, como hemos visto en clase, es 3, con lo que $m_H(2N) \leq (2N)^3 + 1$ con N el número de datos, que en este caso es 599. Con todo esto podemos calcular dicha cota:

```
# Cota sobre la función de crecimiento
mH_cota <- (2*nrow(datos))^3+1
# Cota basada en Ein
Eout_cota1 <- Ein + sqrt((8/nrow(datos))*log((4*mH_cota)/0.05))
cat("La cota basada en Ein es", Eout_cota1)
```

```
## La cota basada en Ein es 0.7522092
```

La cota basada en E_test es la cota de Hoeffding considerando $M = 1$, pues cuando llegamos al test ya tenemos escogida la h, que es la recta elegida con los datos de entrenamiento. Utilizamos por tanto la cota $P[|E_{test}(g) - E_{out}(g)| > \epsilon] \leq 2Me^{-2\epsilon^2 N}$ y queremos que esta probabilidad sea menor o igual que 0.05, con lo que vamos a obtener ϵ: $2e^{-2\epsilon^2 N} \leq 0.05 \Rightarrow e^{-2\epsilon^2 N} \leq 0.025 \Rightarrow -2\epsilon^2 N \leq ln(0.025) \Rightarrow \epsilon^2 \geq ln(0.025)/(-2N) \Rightarrow \epsilon \geq \sqrt{ln(0.025)/(-2N)}$, donde $N = 49$, el número de datos que tenemos en test, luego $\epsilon \geq 0.1940145$. Tomamos el ϵ más pequeño para obtener la cota más fina, y nos queda que la probabilidad de que $|E_{test} - E_{out}| \geq 0.1940145$ es 0.039682.

La cota basada en E_{test} es mejor ya que es más fina, es decir, nos restringe más el error fuera de la muestra, ya que nos dice que la diferencia entre el error que hemos obtenido en test y el que podemos obtener fuera de la muestra es muy probable (1-0.03968 de probabilidad) que sea muy pequeño (0.19). Como conocemos E_{test}, es fácil saber más o menos el error que tendremos fuera de la muestra.

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

$\Phi_3(x) = (1, x_1, x_2, x_1^2, x_2^2, x_1x_2, x_1^3, x_2^3, x_1x_2^2, x_1^2x_2)$

Combinamos los datos según esta transformación, donde en nuestro caso x_1 es la intensidad y x_2 es la simetría, obtenemos los mejores pesos por regresión lineal para después intentar mejorarlos con el PLA pocket y contamos los errores cometidos tanto en el conjunto de train como en el de test.

```
x1 <- train_intensidad
x2 <- train_simetria
datos <- cbind(1, x1, x2, x1^2, x2^2, x1*x2, x1^3, x2^3, x1*x2^2, x2*x1^2)

# Hacemos regresión lineal
w <- Regress_Lin(datos, etiquetas_train)
# Le pasamos el PLA
```

```r
g_pol <- ajusta_PLA_MOD(datos, etiquetas_train, 100, w)[[1]]

etiquetas_g_in <- unlist(lapply(1:nrow(datos), function(i) {
  p <- datos[i,]
  sign(g_pol[1]+g_pol[2]*p[2]+g_pol[3]*p[3]+g_pol[4]*p[4]+g_pol[5]*p[5]+
         g_pol[6]*p[6]+g_pol[7]*p[7]+g_pol[8]*p[8]+g_pol[9]*p[9]+
         g_pol[10]*p[10])
}))

x1 <- test_intensidad
x2 <- test_simetria
datos_test <- cbind(1, x1, x2, x1^2, x2^2, x1*x2, x1^3, x2^3, x1*x2^2, x2*x1^2)

etiquetas_g_test <- unlist(lapply(1:nrow(datos_test), function(i) {
  p <- datos_test[i,]
 sign(g_pol[1]+g_pol[2]*p[2]+g_pol[3]*p[3]+g_pol[4]*p[4]+g_pol[5]*p[5]+
         g_pol[6]*p[6]+g_pol[7]*p[7]+g_pol[8]*p[8]+g_pol[9]*p[9]+
         g_pol[10]*p[10])
}))

Ein <- 100*cuenta_diferencias(etiquetas_train, etiquetas_g_in)/nrow(datos)
Etest <- 100*cuenta_diferencias(etiquetas_test, etiquetas_g_test)/nrow(datos_test)
cat("El error en la muestra Ein ha sido", Ein)
```

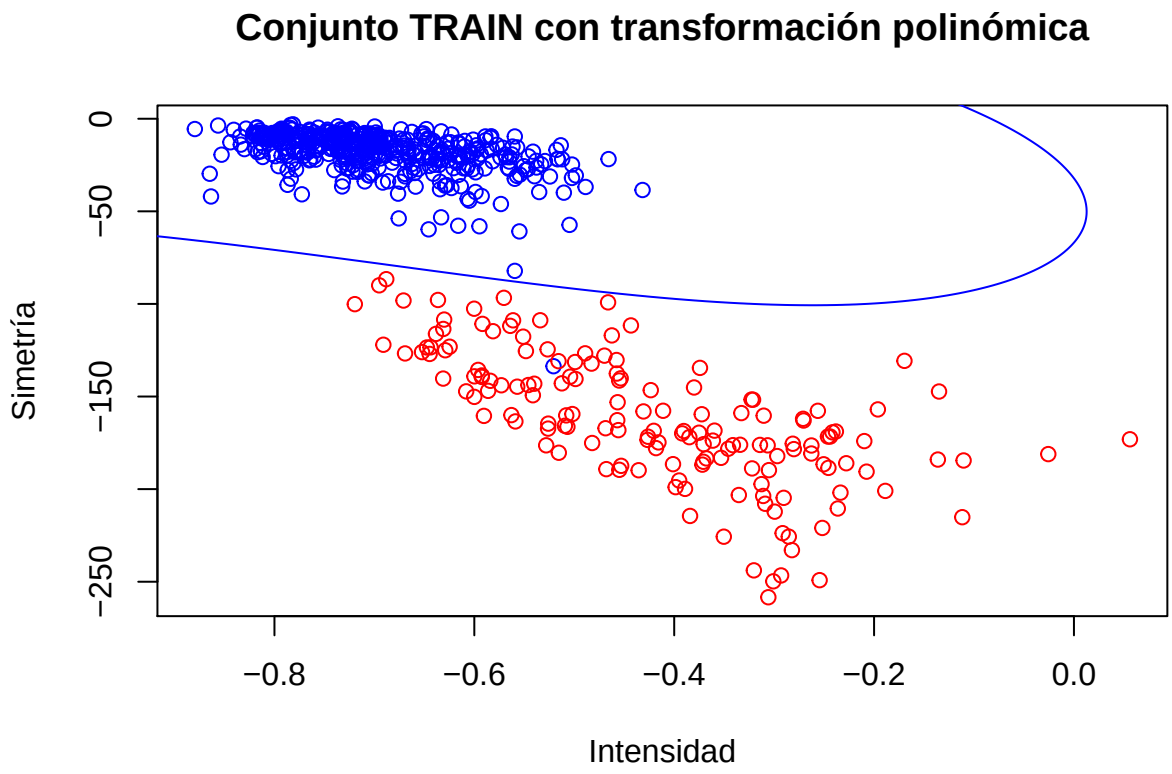
```
## El error en la muestra Ein ha sido 0.1669449
```

```r
cat("El error en el test Etest ha sido", Etest)
```

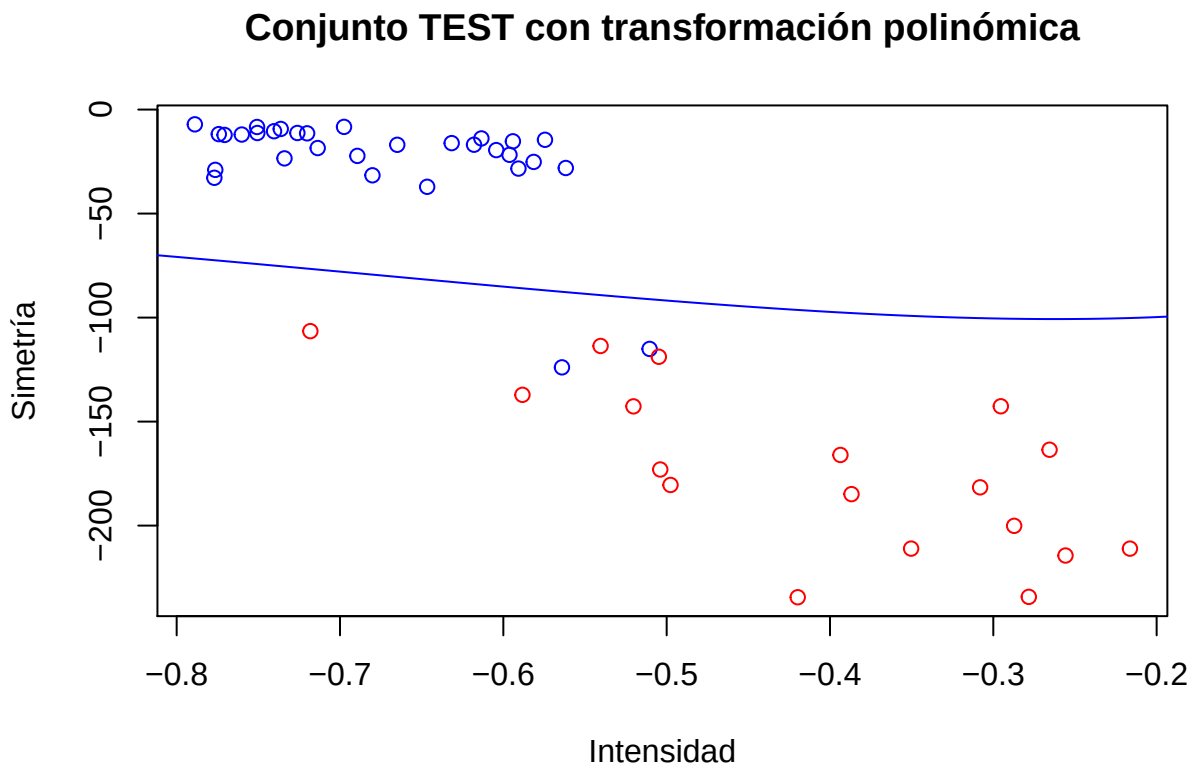
```
## El error en el test Etest ha sido 4.081633
```

Pintamos también esta transformación por ver cómo está clasificando:

```r
# Pintamos el conjunto de train
funcion <- function(x,y) g_pol[1]+g_pol[2]*x+g_pol[3]*y+g_pol[4]*x^2+
  g_pol[5]*y^2+g_pol[6]*x*y+g_pol[7]*x^3+g_pol[8]*y^3+g_pol[9]*x*y^2+
  g_pol[10]*y*x^2
pinta_particion(train_intensidad, train_simetria, etiquetas_train, T, f=funcion,
                main="Conjunto TRAIN con transformación polinómica",
                xlab="Intensidad", ylab="Simetría", rango=250)
```




```
# Pintamos el conjunto de test
pinta_particion(test_intensidad, test_simetria, etiquetas_test, T, f=funcion,
                main="Conjunto TEST con transformación polinómica",
                xlab="Intensidad", ylab="Simetría", rango=250)
```



Hacemos ahora las cotas pedidas en el apartado anterior:

```
# Cota sobre la función de crecimiento
mH_cota <- (2*nrow(datos))^3+1
# Cota basada en Ein
Eout_cota1 <- Ein + sqrt((8/nrow(datos))*log((4*mH_cota)/0.05))
cat("La cota basada en Ein es", Eout_cota1)
```

```
## La cota basada en Ein es 0.7522092
```

La cota sobre E_{test} es la misma que la que hemos calculado previamente, ya que ahí lo único que influye es el número de datos que tenemos en test, que es algo que no ha cambiado.

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

No, es demasiado complejo para unos datos que son prácticamente separables y pueden resolverse de forma más cómoda con una recta que con la transformación polinómica, cuando además no se está obteniendo mejora con la transformación, ya que los errores sobre train y test son los mismos y las cotas sobre E_{out} son mejores con la recta que con la transformación polinómica.

```
# Borramos lo que no necesitamos
rm(datos, datos_test, r, w)
rm(Ein, Eout_cota1, Etest, etiquetas_g_in)
rm(etiquetas_g_test, etiquetas_test, etiquetas_train)
```

```
rm(lista_test, lista_train, mH_cota, sol, test_intensidad)
rm(test_simetria, train_intensidad, train_simetria, x1, x2)
rm(g_pol)
```

2. SOBREAJUSTE

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

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

a) Generamos los coeficientes a_q a partir de muestras de una distribución $\mathcal{N}(0, 1)$ y escalamos dichos coeficientes de manera que $E_{a,x}[f^2] = 1$ (Ayuda: dividir los coeficientes por $\sqrt{\sum_{q=0}^{Q_f} \frac{1}{2q+1}}$

Fijamos $N = 10$, $\sigma = 0.2$, $Q_f = 3$ y generamos a_q con la función de la práctica anterior que extraía valores aleatorios de una normal y los escalamos como nos pide.

```
# Reinicializamos la semilla
set.seed(27172)
# Fijamos parámetros y extraemos los aq
N <- 10
sigma <- 0.2^2
Qf <- 3
aq <- unlist(simula_gauss(Qf+1, 1, sigma))
# Escalamos
escalado <- sapply(0:Qf, function(q) 1/(2*q+1))
escalado <- sum(sqrt(escalado))
aq <- aq/escalado
```

b) Generamos un conjunto de datos $x_1, ..., x_N$ muestreando de forma independiente $P(x)$ y los valores $y_n = f(x_n) + \sigma\epsilon_n$

Los datos, como tenemos la probabilidad 1/2, los tenemos que extraer de la uniforme en el rango $[-1, 1]$, que es el espacio que nos han dado previamente. Generamos también y_n en órdenes distintas para hacerlo de forma independiente. Para esto último va a ser necesario primero crear los polinomios de Legendre y multiplicarlos por los coeficientes a_q que acabamos de calcular para obtener $f(x_n)$ y generar también los ϵ_n, extrayéndolos de una normal 0,1.

Vamos a hacer por tanto previamente una función para obtener los polinomios de Legendre de grado k.

```
# Función para calcular los polinomios de Legendre, con una ecuación en
# diferencias, de hasta grado k en función del valor en el que evaluemos x.
polLegendre <- function(x, k) {
  L <- vector("numeric", k+1)
  if(k == 0) {
    L[1] <- 1
  }
  else {
    L[1] <- 1
    L[2] <- x
    if (k > 2){
      for(i in 2:k) {
        L[i+1] <- ((2*i-1)/i)*x*L[i] - ((i-1)/i)*L[i-1]
      }
    }
  }
  return(L)
}
```

Hacemos también una función auxiliar para calcular f, dados los coeficientes escalados aq y Qf:

```
f <- function(x, aq, Qf) {
  sapply(x, function(x_i){
    crossprod(aq, polLegendre(x_i,Qf))
  })
}
```

Calculamos los y_n como el valor real de los datos en f más un ruido dado por un escalado σ por unos datos extraídos de una normal 0,1.

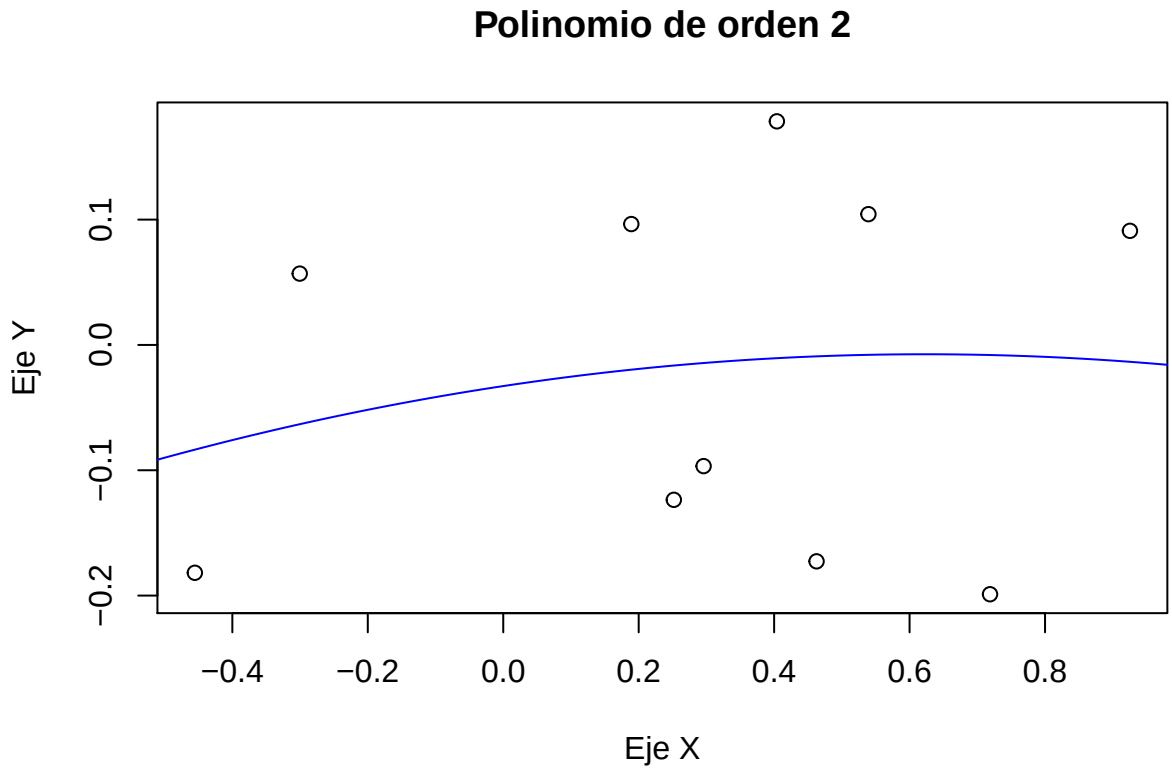
```
datos <- unlist(simula_unif(N, 1, c(-1,1)))
epsilon_n <- unlist(simula_gauss(N, 1, 1))
y_n <- f(datos, aq, Qf) + sqrt(sigma)*epsilon_n
```

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

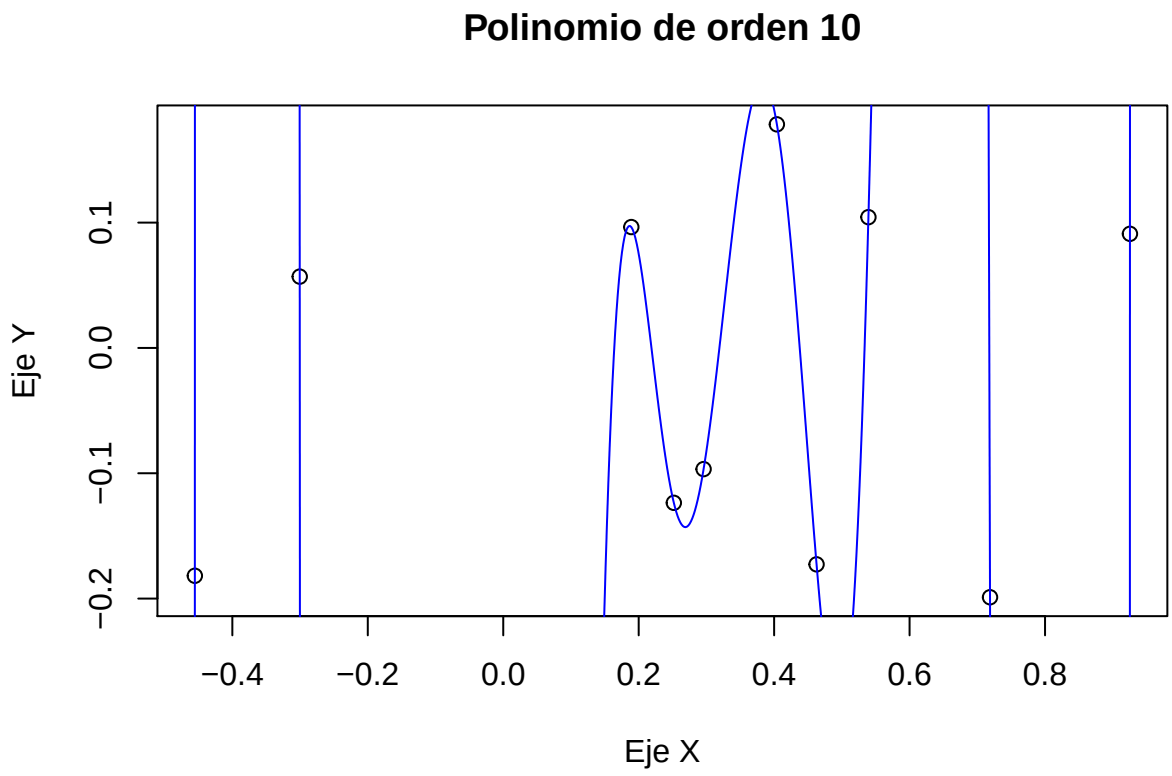
a) Calcular g_2 y g_{10}

Como hemos visto que la regresión lineal da buenos resultados, vamos a calcular g_2 y g_{10} por regresión lineal, ajustando las y_n calculadas a los datos x_n

```
w <- Regress_Lin(cbind(1, datos, datos^2), y_n)
# Pintamos el resultado
pinta_particion(datos, y_n, etiquetas=NULL, visible=T,
                            function(x,y) w[1]+w[2]*x+w[3]*x^2-y,
                main="Polinomio de orden 2", xlab="Eje X", ylab="Eje Y")
```




```
w10 <- Regress_Lin(cbind(1, datos, datos^2, datos^3, datos^4, datos^5, datos^6,
                         datos^7, datos^8, datos^9, datos^10), y_n)
pinta_particion(datos, y_n, etiquetas=NULL, visible=T,
                            function(x,y) w10[1]+w10[2]*x+w10[3]*x^2+w10[4]*x^3+
                  w10[5]*x^4+w10[6]*x^5+w10[7]*x^6+w10[8]*x^7+w10[9]*x^8+
                  w10[10]*x^9+w10[11]*x^10-y, main="Polinomio de orden 10",
                xlab="Eje X", ylab="Eje Y")
```



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

σ es el factor por el que multiplicamos unos datos uniformes entre -1 y 1, que juntos hacen el ruido que le damos a las y_n. Al calibrar f de forma que $\mathcal{E}[f^2] = 1$, estamos calibrando el ruido σ^2 con la función f.

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

La forma de obtener E_{out} analíticamente para una g_{10} dada es calcular la integral definida en el conjunto donde varían las coordenadas x de los puntos, de la diferencia entre g_{10} y la función f original (sin ruido).

2. Siguiendo con el punto anterior, usando la combinación de parámetros $Q_f = 20$, $N = 50$, $\sigma = 1$ ejecutar un número grande de experimentos (>100) calculando en cada caso $E_{out}(g_2)$ y $E_{out}(g_{10})$. Promediar todos los valores de error obtenidos para cada conjunto de hipótesis, es decir $E_{out}((H)_2)$ = promedio sobre experimentos$(E_{out}(g_2))$ y $E_{out}((H)_{10})$ = promedio sobre experimentos$(E_{out}(g_{10}))$.

Cambiamos los parámetros como nos pide, volvemos a calcular aq y normalizarlos y obtenemos una muestra de 50 datos.

```
Qf <- 20
N <- 50
sigma <- 1
aq <- unlist(simula_gauss(Qf+1, 1, sigma))
```

```r
# Escalamos
escalado <- sapply(0:Qf, function(q) 1/(2*q+1))
escalado <- sum(sqrt(escalado))
aq <- aq/escalado
datos <- unlist(simula_unif(N, 1, c(-1,1)))
epsilon_n <- unlist(simula_gauss(N, 1, 1))
y_n <- f(datos, aq, Qf) + sqrt(sigma)*epsilon_n

# Obtenemos g2 y g10 con estos datos
w2 <- Regress_Lin(cbind(1, datos, datos^2), y_n)
w10 <- Regress_Lin(cbind(1, datos, datos^2, datos^3, datos^4, datos^5, datos^6,
                         datos^7, datos^8, datos^9, datos^10), y_n)
```

Como vamos a calcular el error fuera de la muestra, estamos suponiendo que conocemos la función f original, por lo que para calcular error lo vamos a hacer tomando la diferencia al cuadrado entre el valor real de f y la función obtenida.
Lo que hacemos es un bucle exterior de 101 iteraciones en el que en cada uno generamos unos datos distintos que serán los datos fuera de la muestra y en cada iteración de este bucle pasamos por todos los datos calculando el error de cada uno de estos datos para g_2 y para g_{10}, acumulándolos para después hacer la media.

```r
error2 <- 0
error10 <- 0
for(i in 0:100) {
  # Generamos 100 datos fuera de la muestra inicial
  datos_out <- unlist(simula_unif(100, 1, c(-1,1)))

  for(j in 1:100) {
    # Calculamos lo que vale cada punto en realidad con la función objetivo
    valor_real <- f(datos_out[j], aq, Qf)
    # Calculamos ahora la diferencia al cuadrado de las diferencias con los
    # polinomios de grado 2 y grado 10 y las acumulamos
    valorg2 <- w2[1]+w2[2]*datos_out[j]+w2[3]*datos_out[j]^2
    valorg10 <- w10[1]+w10[2]*datos_out[j]+w10[3]*datos_out[j]^2+
      w10[4]*datos_out[j]^3+w10[5]*datos_out[j]^4+w10[6]*datos_out[j]^5+
      w10[7]*datos_out[j]^6+w10[8]*datos_out[j]^7+w10[9]*datos_out[j]^8+
      w10[10]*datos_out[j]^9+w10[11]*datos_out[j]^10
    error2 <- error2 + (valor_real - valorg2)^2
    error10 <- error10 + (valor_real - valorg10)^2
  }
}
Eout2 <- error2/100
Eout10 <- error10/100

cat("Eout para el polinomio de grado 2 ha sido", Eout2)
```

```
## Eout para el polinomio de grado 2 ha sido 8.681256
```

```r
cat("Eout para el polinomio de grado 10 ha sido", Eout10)
```

```
## Eout para el polinomio de grado 10 ha sido 33.80701
```

Como vemos, hay mucho más error en el polinomio de grado 10 que en el polinomio de grado 2, pese a que como hemos visto en las gráficas anteriores, el polinomio de grado 10 pasaba por todos los puntos que teníamos y el de grado 2 no pasaba por ninguno de ellos: esta es la principal consecuencia del sobreajuste.

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

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

Calculamos la medida de sobreajuste:

```
sobreajuste <- error10 - error2
cat("La medida de sobreajuste ha salido:", sobreajuste)
```

```
## La medida de sobreajuste ha salido: 2512.575
```

Dicha medida puede medir el sobreajuste de g_{10} porque estamos comparando su error fuera de la muestra con el error fuera de la muestra de otro polinomio también ajustado, no el original. Si este polinomio g_{10} tiene menos error dentro de la muestra que otro pero a la vez más error fuera que éste mismo (es decir, dicha medida anterior es positiva), entonces hay sobreajuste, pues estamos ajustando tanto para los datos dentro de la muestra que estamos obviando la función real que queríamos ajustar (debido al error presente en las y_n), de forma que el error fuera de la muestra aumenta mucho, como es el caso que tenemos ahora mismo. Además, cuando mayor sea la medida anterior, más sobreajuste habrá.

b) Usando la combinación de valores $Q_f \in \{1, 2, ..., 100\}$, $N \in \{20, 25, ..., 100\}$, $\sigma \in \{0; 0.05; 0.1; ...; 2\}$, se obtiene una gráfica como la que aparece en la figura 4.3 del libro "Learning from data", capítulo 4. Interpreta la gráfica respecto a las condiciones en las que se da el sobreajuste. (Nota: No es necesario la implementación).

En la página 124 tenemos dos gráficas, puesto que hay tres variables distintas. En una de ellas está fijada σ^2 y en la otra Q_f.
Pensemos primero en qué se supone que debería pasar. Para un número pequeño de datos (como por ejemplo, para 10 datos, como hemos hecho el experimento en los apartados anteriores) el polinomio de orden 10 va a ir a pasar exactamente por los datos (o muy cerca, si aumentamos un poco el número de datos) sobreajustando así mucho, mientras que el polinomio de orden 2 no puede hacer esto y ajustará mejor la función original. Sin embargo si aumentamos el número de datos de forma considerable el polinomio de orden 10 se irá comportando cada vez mejor ya que no puede pasar por todos los datos. Si ahora variamos también la complejidad, el polinomio de orden 10 ajustará mejor que el de orden 2 para mayor complejidad y muchos datos.

Vamos a fijarnos ahora en la primera gráfica, que deja fijo $Q_f = 20$. La zona roja oscura significa que hay sobreajuste, mientras que la azul oscura, que el polinomio de orden 10 da mejor resultado fuera de la muestra que el de grado 2. Efectivamente, cuando crece N g_{10} acaba comportándose mejor que g_2 cuando hay poco ruido. Cuando el ruido crece sigue comportándose mejor el polinomio de orden 2 ya que el de orden 10 intentará ajustar los datos que tienen mucho ruido más que el de grado 2.
En la segunda gráfica se deja fijo $\sigma^2 = 0.1$. Para complejidad menor que 10, como no hay mucho error, el polinomio de orden 10 puede acercarse más a los datos y por tanto ajusta mejor que el de orden 2. Sin embargo para mayor complejidad y pocos datos ocurre lo mismo que hemos mencionado antes, y es que el polinomio de orden 10 se ve más afectado por el ruido que el de orden 2 y por tanto el de orden 2 ajusta mejor. Como hay poco error el número de datos para que el polinomio de orden 10 ajuste mejor es menor que antes: con 100 datos ya ajusta mejor y con 120 es mejor para cualquier complejidad.

```
# Borramos lo que no necesitamos
rm(datos, datos_out, w, w10, w2)
rm(aq, epsilon_n, error10, error2, escalado)
rm(i, j, N, Qf, sigma, sobreajuste, valor_real)
rm(valorg10, valorg2, y_n, f)
rm(Eout2, Eout10)
```

3. REGULARIZACIÓN Y SELECCIÓN DE MODELOS

1. Para $d = 3$ (dimensión) generar un conjunto de N datos aleatorios $\{x_n, y_n\}$ de la sigiuente forma. Para cada punto x_n generamos sus coordenadas muestreando de forma independiente una $\mathcal{N}(0, 1)$. De forma similar generamos un vector de pesos de $(d + 1)$ dimensiones $\mathbf{w}_f$, y el conjunto de valores $y_n = \mathbf{w}_f^T x_n + \sigma\epsilon_n$, donde ϵ_n es un ruido que sigue también una $\mathcal{N}(0, 1)$ y σ^2 es la varianza del ruido; fijar $\sigma = 0.5$

Usar regresión lineal con regularización "weight decay" para estimar $\mathbf{w}_f$ con $\mathbf{w}_{reg}$. Fijar el parámetro de regularización a $0.05/N$.

Creamos un método nuevo de regresión lineal con weigth decay:

```
# Volvemos a fijar la semilla
set.seed(12345)

# Método de regresión lineal con weigth decay. Recibe los datos y las etiquetas
# por parámetros, junto con el weigth decay lambda.
Regress_Lin_WD <- function(datos, label, lambda) {
  Z <- t(datos)%*%datos
  dim <- nrow(Z)
  Z <- Z + diag(lambda, dim, dim)
  inversa <- solve(Z)
  wreg <- inversa %*% t(datos) %*% label
  return(wreg)
}
```

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

Como ahora estamos calculando errores de validación cruzada con los datos de dentro de la muestra, nosotros en realidad no conocemos la función objetivo real, sólo conocemos las y_n calculadas con ruido y es con estas y_n con las que tenemos que calcular la diferencia al cuadrado para calcular los e_i.

```
d <- 3
sigma <- 0.5

wf <- unlist(simula_gauss(1, d+1, 1))
N <- 18
errores <- vector("list", 11)
i <- 1
while (N <= 118) {
  lambda <- 0.05/N
  datos <- simula_gauss(N, 3, sigma^2)
  datos <- matrix(unlist(datos), N, 3, T)
  datos <- cbind(datos,1)
  # Generamos los y_n
  y_n <- sapply(1:N, function(i) crossprod(datos[i,],wf)+sigma*rnorm(1))

  e <- vector("numeric", N)
```

```
  # Hacemos leave one out
  for(j in 1:N) {
    datos_loo <- datos[-j,]
    yn_loo <- y_n[-j]
    xn <- datos[j,]
    # Estimamos wreg con regresión lineal con weigth decay
    wreg <- Regress_Lin_WD(datos_loo, yn_loo, lambda)
    # Calculamos el error de validación cruzada
    valor_estimado <- crossprod(wreg,xn)
    e_n <- (y_n[j] - valor_estimado)^2
    e[j] <- e_n
  }
  errores[[i]] <- e

  N <- N+10
  i <- i+1
}
```

Los valores $e_1, ..., e_N$ pedidos están en la lista `errores`, donde cada elemento de la lista se corresponde con los $e_1, ..., e_N$ para cada N. Calculamos ahora E_{cv} como la media de cada elemento de la lista:

```
listaEcv <- sapply(errores, function(i) mean(i))
cat("La media de los errores de validación cruzada, Ecv, son los siguientes (para cada N)"
    , listaEcv)
```

```
## La media de los errores de validación cruzada, Ecv, son los siguientes (para cada N) 0.6145654 0.259
```

b) Repetir el experimento 10^3 veces, anotando el promedio y la varianza de e_1, e_2 y E_{ev} en todos los experimentos.

Para poder ir guardando todos los valores lo que hacemos es crear varios vectores previamente e ir guardando los valores en estos vectores.

```
wf <- unlist(simula_gauss(1, d+1, 1))
N <- 18
errores <- vector("list", 11)
varianzas <- vector("list", 11)
i <- 1
while (N <= 118) {
  e1 <- vector("numeric", 10^3)
  e2 <- vector("numeric", 10^3)
  Ecv <- vector("numeric", 10^3)
  for(k in 1:10^3) {
    lambda <- 0.05/N
    datos <- simula_gauss(N, 3, sigma^2)
    datos <- matrix(unlist(datos), N, 3, T)
    datos <- cbind(datos,1)
    # Generamos los y_n
    y_n <- sapply(1:N, function(i) crossprod(datos[i,],wf)+sigma*rnorm(1))

    e <- vector("numeric", N)
```

```
    # Hacemos leave one out
    for(j in 1:N) {
      datos_loo <- datos[-j,]
      yn_loo <- y_n[-j]
      xn <- datos[j,]
      # Estimamos wreg con regresión lineal con weigth decay
      wreg <- Regress_Lin_WD(datos_loo, yn_loo, lambda)
      # Calculamos el error de validación cruzada
      valor_estimado <- crossprod(wreg,xn)
      e_n <- (y_n[j] - valor_estimado)^2
      e[j] <- e_n
    }
    e1[k] <- e[1]
    e2[k] <- e[2]
    Ecv[k] <- mean(e)
  }

  errores[[i]] <- c(mean(e1), mean(e2), mean(Ecv))
  varianzas[[i]] <- c(var(e1), var(e2), var(Ecv))
  i <- i+1
  N <- N+10
}
```

```
print("La media de e1, e2 y Ecv ha sido, para cada N y en este orden:")
```

```
## [1] "La media de e1, e2 y Ecv ha sido, para cada N y en este orden:"
```

```
print(errores)
```

```
## [[1]]
## [1] 0.3196860 0.3490673 0.3250890
##
## [[2]]
## [1] 0.3215812 0.2958883 0.2917521
##
## [[3]]
## [1] 0.2731672 0.2742570 0.2830277
##
## [[4]]
## [1] 0.2734939 0.2842869 0.2766671
##
## [[5]]
## [1] 0.2957754 0.2602127 0.2714315
##
## [[6]]
## [1] 0.2856189 0.2588999 0.2654714
##
## [[7]]
## [1] 0.2510682 0.2588083 0.2643169
##
## [[8]]
## [1] 0.2640747 0.2720883 0.2632301
```

```
##
## [[9]]
## [1] 0.2727506 0.2656432 0.2598345
##
## [[10]]
## [1] 0.2552232 0.2491069 0.2605229
##
## [[11]]
## [1] 0.2634994 0.2594971 0.2567083
```

```
print("La varianza de e1, e2 y Ecv ha sido, para cada N y este orden:")
```

```
## [1] "La varianza de e1, e2 y Ecv ha sido, para cada N y este orden:"
```

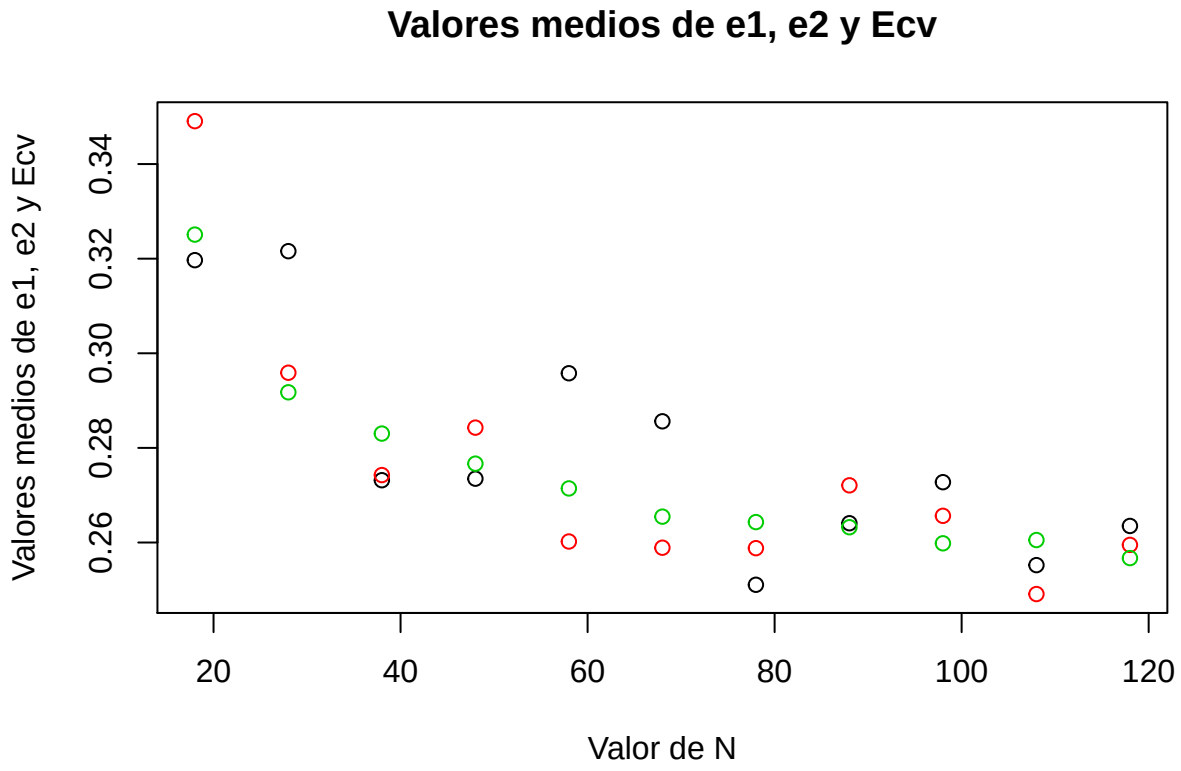
```
print(varianzas)
```

```
## [[1]]
## [1] 0.22169391 0.25543493 0.01739098
##
## [[2]]
## [1] 0.197271095 0.191119425 0.007589383
##
## [[3]]
## [1] 0.152941704 0.149164851 0.004716632
##
## [[4]]
## [1] 0.155924314 0.160497206 0.003395157
##
## [[5]]
## [1] 0.179336694 0.152185525 0.002502404
##
## [[6]]
## [1] 0.157390541 0.149708302 0.002237586
##
## [[7]]
## [1] 0.106083293 0.122267221 0.001961178
##
## [[8]]
## [1] 0.12909624 0.15021842 0.00161002
##
## [[9]]
## [1] 0.138347857 0.121085986 0.001418174
##
## [[10]]
## [1] 0.128970683 0.108707859 0.001342008
##
## [[11]]
## [1] 0.140770775 0.132052610 0.001170227
```

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

Como tenemos que argumentar en base a los resultados de los experimentos anteriores, vamos a mostrar una gráfica con los valores e_1, e_2 y E_{cv} en media por cada N (el negro es el color para los valores de e_1, el rojo es el color para los valores de e_2 y el verde el color para los valores de E_{cv}):

```
color <- rep(6:8,11)
N <- c(rep(18,3), rep(28,3), rep(38,3), rep(48, 3), rep(58,3), rep(68,3),
       rep(78,3), rep(88,3), rep(98,3), rep(108,3), rep(118,3))
pinta_particion(N, unlist(errores), color, xlab="Valor de N",
                ylab="Valores medios de e1, e2 y Ecv",
                main="Valores medios de e1, e2 y Ecv")
```



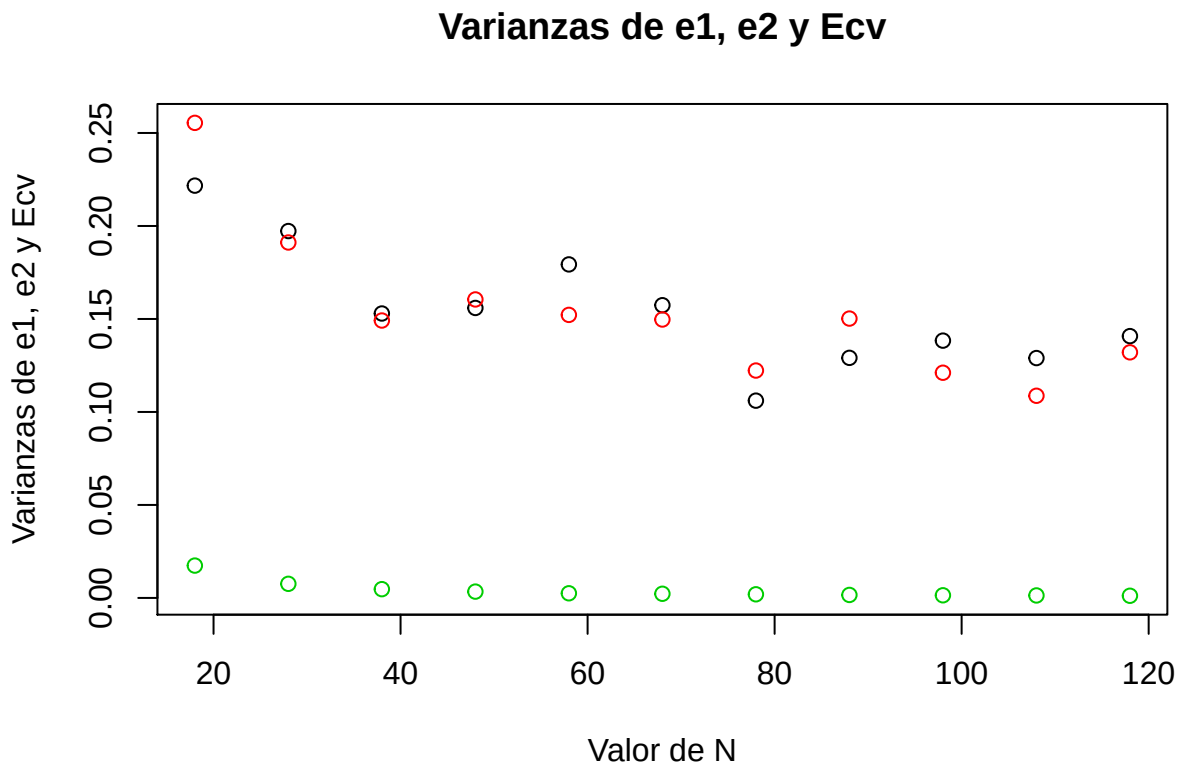
Como vemos, la diferencia entre el promedio de los valores de e_1, e_2 y E_{cv} es pequeña (y se va haciendo más pequeña cuanto mayor es la N), que es algo que tiene sentido ya que lo que aquí se está representando son las medias de los 1000 valores que nos han salido de e_1, e_2 y E_{cv} para cada valor de N. Es lógico que las medias de e_1 y e_2 se parezcan (todo lo que el ruido nos permite) si pensamos en lo que estamos haciendo: para cada valor en la muestra, lo sacamos de la misma y calculamos una función que ajuste los demás, y e_i es el valor de error cuadrático que tiene el punto que hemos sacado fuera. Puede que algún punto esté muy lejos de la función estimada o que alguno caiga dentro, pero en media quedarán más o menos todos a la misma distancia, sin importar que el punto que hemos sacado sea el primero, el segundo o el último, por lo que las medias de e_i serán parecidas y, como es lógico, cuanto mayor sea el número de datos en la muestra (N), mejor estimaremos la función y menor error habrá, como se puede apreciar en la gráfica, de hecho, esto se ve muy bien cuando tomamos la media de todos los e_i, es decir, E_{cv}, que son los puntos verdes, y vemos que se

van acercando cada vez más a cero cuando N crece de forma más o menos regular, dando menos saltos que e_1 y e_2.

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

Vamos a mostrar otra gráfica, ahora con los valores para las varianzas, con el mismo código de colores.

```
pinta_particion(N, unlist(varianzas), color, xlab="Valor de N",
                ylab="Varianzas de e1, e2 y Ecv",
                main="Varianzas de e1, e2 y Ecv")
```



Lo que contribuye a la varianza tanto de e_1 como de e_2 y E_{cv} son los valores que se quedan lejos de la media. Como vemos en la gráfica, las varianzas de e_1 y e_2 están más o menos cercanas y las varianzas de E_{cv} están siempre cercanas a cero. Esto también es lógico. Vamos a centrarnos primero en valores pequeños de N: los valores que se quedan lejos de la media influyen más porque tenemos pocos datos, que es por lo que las varianzas son un poco más altas para $N = 18$. Ahora, las varianzas de e_i van a ser más altas que la varianza de E_{cv} porque E_{cv} es la media de todos los e_i y en media los errores, por lo que hemos comentado en el punto anterior, son parecidos (muy parecidos, como vemos en la gráfica, ya que la varianza se queda muy cerca de cero), es decir, ovbiamente va a haber menos varianza entre dos medias (caso de E_{cv}) que entre varios datos (caso de e_i).

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

La relación es fácil de sacar si recordamos dos propiedades de la varianza: la primera es que la varianza de un escalar por una variable aleatoria es ese escalar al cuadrado por la varianza de la variable aleatoria y la segunda que la varianza de la suma de dos variables aleatorias es la suma de las varianzas más dos veces la covarianza entre ambas variables aleatorias.
Como estamos suponiendo que las variables son de verdad independientes, la covarianza entre dos e_i o cualquier e_i y E_{cv} es cero, con lo que nos queda que la varianza de la suma es la suma de las varianzas.

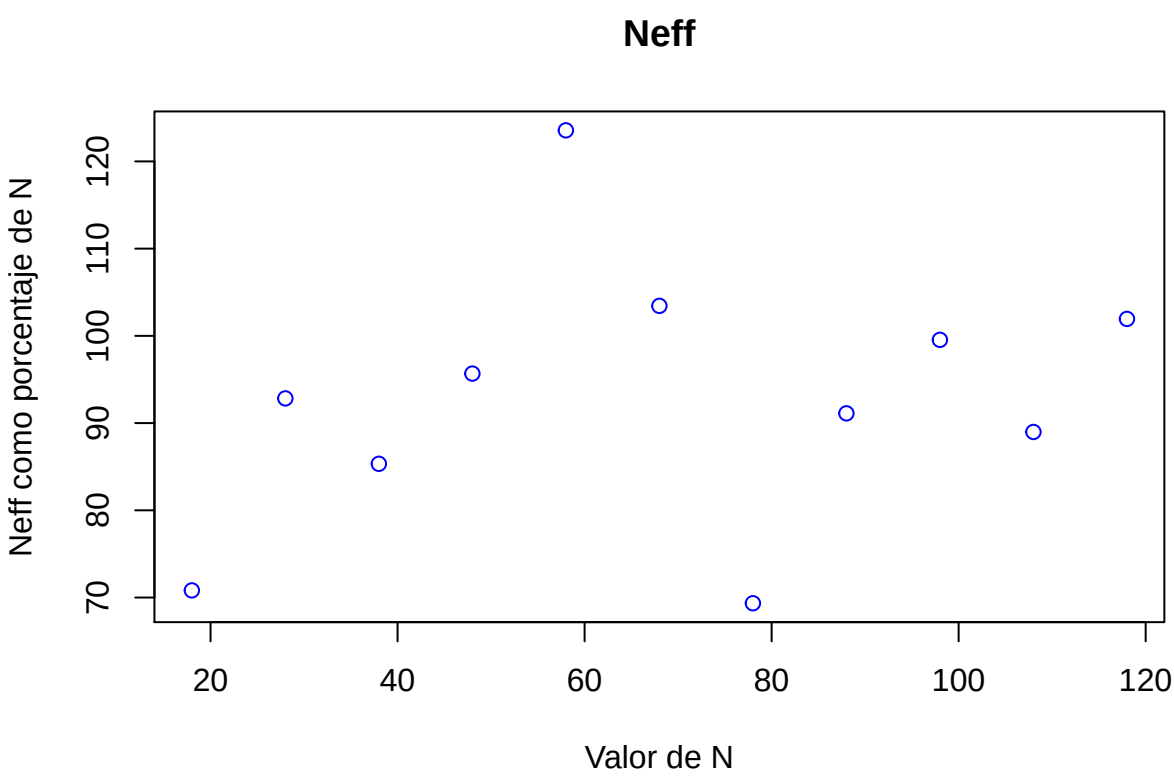
Vamos ahora a partir de la varianza de E_{cv}: $Var(E_{cv}) = Var(\frac{1}{N}\sum_{i=1}^{N} e_i)$ por ser E_{cv} la media de los e_i y $Var(\frac{1}{N}\sum_{i=1}^{N} e_i) = \frac{1}{N^2}Var(\sum_{i=0}^{N} e_i)$ por la primera propiedad y $\frac{1}{N^2}Var(\sum_{i=0}^{N} e_i) = \frac{1}{N^2}\sum_{i=0}^{N} Var(e_i)$ por la segunda propiedad.

Hemos llegado así a una relación entre la varianza de E_{cv} y la suma de las varianzas de e_i. Supongamos que las varianzas de e_i son lo suficientemente parecidas (algo que experimentalmente hemos visto que ocurre en el apartado anterior) como para poder suponer que $Var(e_1) = Var(e_2) = ... = Var(e_N)$ de forma que $\sum_{i=0}^{N}(Var(e_i)) = NVar(e_1)$, entonces encontramos una relación entre las dos varianzas pedidas: $Var(E_{cv}) = \frac{1}{N}Var(e_1)$.

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

Hacemos una gráfica como las de los apartados anteriores pero ahora dibujando el cociente entre e_1 y E_{cv} para cada N.

```
e_1 <- rapply(varianzas, function(x) x[1])
Ecv <- rapply(varianzas, function(x) x[3])
N <- seq(18, 118, by=10)
Neff <- 100*(e_1/Ecv)/N
pinta_particion(N, Neff, rep(1,11), main="Neff", xlab="Valor de N",
                ylab="Neff como porcentaje de N")
```



Podemos ver que N_{eff} como porcentaje de N se queda en una franja alredador del 100%, que es lo que el ejercicio nos decía que debíamos encontrar, en concreto entre el 70% y el 120%.

g) Si se incrementa la cantidad de regularización, ¿debería N_{eff} subir o bajar?. Argumentar la respuesta. Ejecutar el mismo experimento con $\lambda = 2.5/N$ y comparar los resultados del punto anterior para verificar la conjetura.

Que suba o baje depende de cuánto aumentemos λ, puesto que hay un tope en la ganancia a partir del cual cuanto más aumentemos λ peor estaremos ajustando la función y mayores van a ser los errores, ya que cuanto más aumentemos la penalización menos grado tendrá la solución escogida y más se parecerá la solución a una recta, que seguramente no sea la función que queremos ajustar.
Teniendo en cuenta que N_{eff} es el cociente entre la varianza de e_1, éste subirá cuando la varianza de e_1 suba y bajará cuando la varianza de e_1 baje, ya que la varianza de E_{cv} no debería variar mucho con respecto al experimento anterior ya que sigue siendo la varianza de medias.
Entonces el cociente e_1/E_{cv} será mayor o menor según aumentemos λ mucho o poco.

Vamos a verlo experimentalmente para $\lambda = 2.5/N$.

```
wf <- unlist(simula_gauss(1, d+1, 1))
N <- 18
errores <- vector("list", 11)
varianzas <- vector("list", 11)
i <- 1
while (N <= 118) {
  e1 <- vector("numeric", 10^3)
  e2 <- vector("numeric", 10^3)
```

```
  Ecv <- vector("numeric", 10^3)
  for(k in 1:10^3) {
    lambda <- 2.5/N
    datos <- simula_gauss(N, 3, sigma^2)
    datos <- matrix(unlist(datos), N, 3, T)
    datos <- cbind(datos,1)
    # Generamos los y_n
    y_n <- sapply(1:N, function(i) crossprod(datos[i,],wf)+sigma*rnorm(1))

    e <- vector("numeric", N)

    # Hacemos leave one out
    for(j in 1:N) {
      datos_loo <- datos[-j,]
      yn_loo <- y_n[-j]
      xn <- datos[j,]
      # Estimamos wreg con regresión lineal con weigth decay
      wreg <- Regress_Lin_WD(datos_loo, yn_loo, lambda)
      # Calculamos el error de validación cruzada
      valor_estimado <- crossprod(wreg,xn)
      e_n <- (y_n[j] - valor_estimado)^2
      e[j] <- e_n
    }
    e1[k] <- e[1]
    e2[k] <- e[2]
    Ecv[k] <- mean(e)
  }

  errores[[i]] <- c(mean(e1), mean(e2), mean(Ecv))
  varianzas[[i]] <- c(var(e1), var(e2), var(Ecv))
  i <- i+1
  N <- N+10
}
```

```
print("La media de e1, e2 y Ecv ha sido, para cada N y en este orden:")
```

```
## [1] "La media de e1, e2 y Ecv ha sido, para cada N y en este orden:"
```

```
print(errores)
```

```
## [[1]]
## [1] 0.3174694 0.3364280 0.3264335
##
## [[2]]
## [1] 0.2723716 0.2694061 0.2897515
##
## [[3]]
## [1] 0.2952725 0.2600080 0.2782929
##
## [[4]]
## [1] 0.2810709 0.2641499 0.2733715
##
```

```
## [[5]]
## [1] 0.2725355 0.2746333 0.2697573
##
## [[6]]
## [1] 0.2713064 0.2735469 0.2651213
##
## [[7]]
## [1] 0.2760106 0.2718873 0.2638023
##
## [[8]]
## [1] 0.2872527 0.2586385 0.2620949
##
## [[9]]
## [1] 0.2854015 0.2452233 0.2591895
##
## [[10]]
## [1] 0.2573986 0.2575120 0.2609892
##
## [[11]]
## [1] 0.2641310 0.2494651 0.2595596
```

```
print("La varianza de e1, e2 y Ecv ha sido, para cada N y este orden:")
```

```
## [1] "La varianza de e1, e2 y Ecv ha sido, para cada N y este orden:"
```

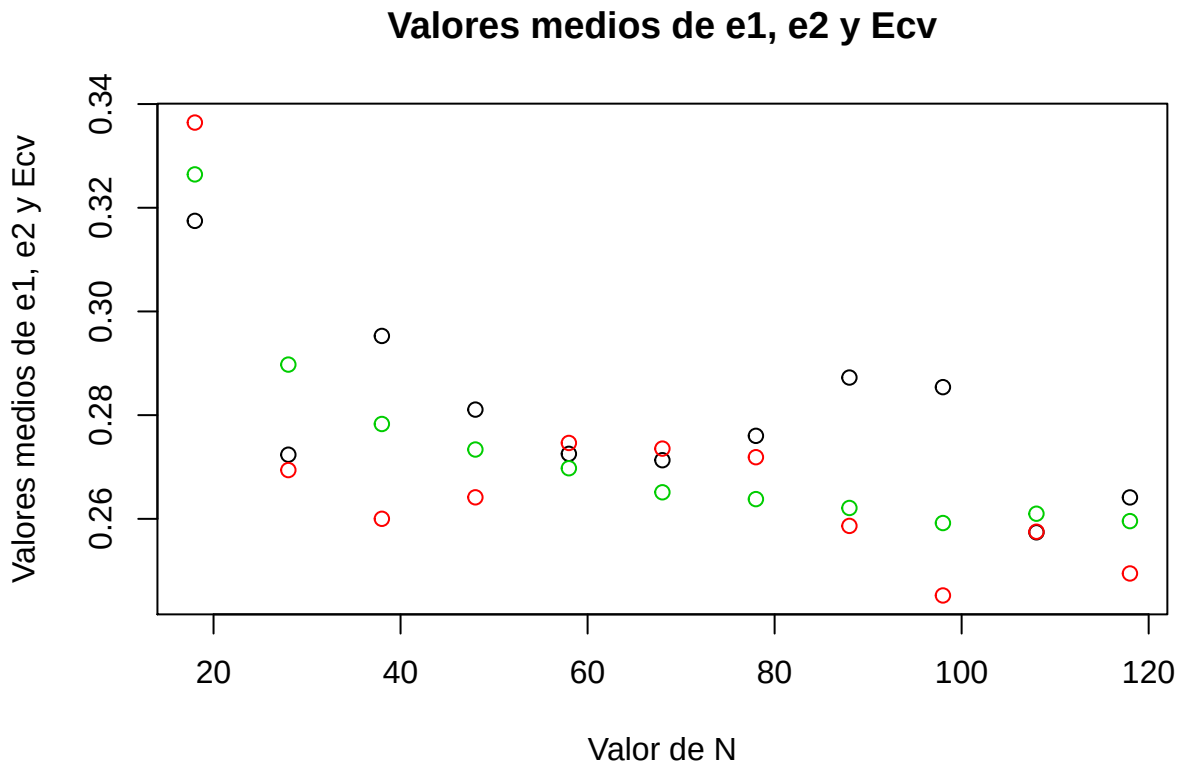
```
print(varianzas)
```

```
## [[1]]
## [1] 0.20172296 0.26792991 0.01520387
##
## [[2]]
## [1] 0.151061377 0.150594529 0.006897617
##
## [[3]]
## [1] 0.166551726 0.141524080 0.004604044
##
## [[4]]
## [1] 0.159697646 0.148022593 0.003404038
##
## [[5]]
## [1] 0.1493753 0.1449348 0.0029299
##
## [[6]]
## [1] 0.16096367 0.14833842 0.00216321
##
## [[7]]
## [1] 0.148931421 0.163607587 0.001907182
##
## [[8]]
## [1] 0.171162637 0.143686626 0.001484442
##
## [[9]]
## [1] 0.16736419 0.12337452 0.00142021
```

```
##
## [[10]]
## [1] 0.12265991 0.13895775 0.00132971
##
## [[11]]
## [1] 0.140132727 0.104557613 0.001256362
```

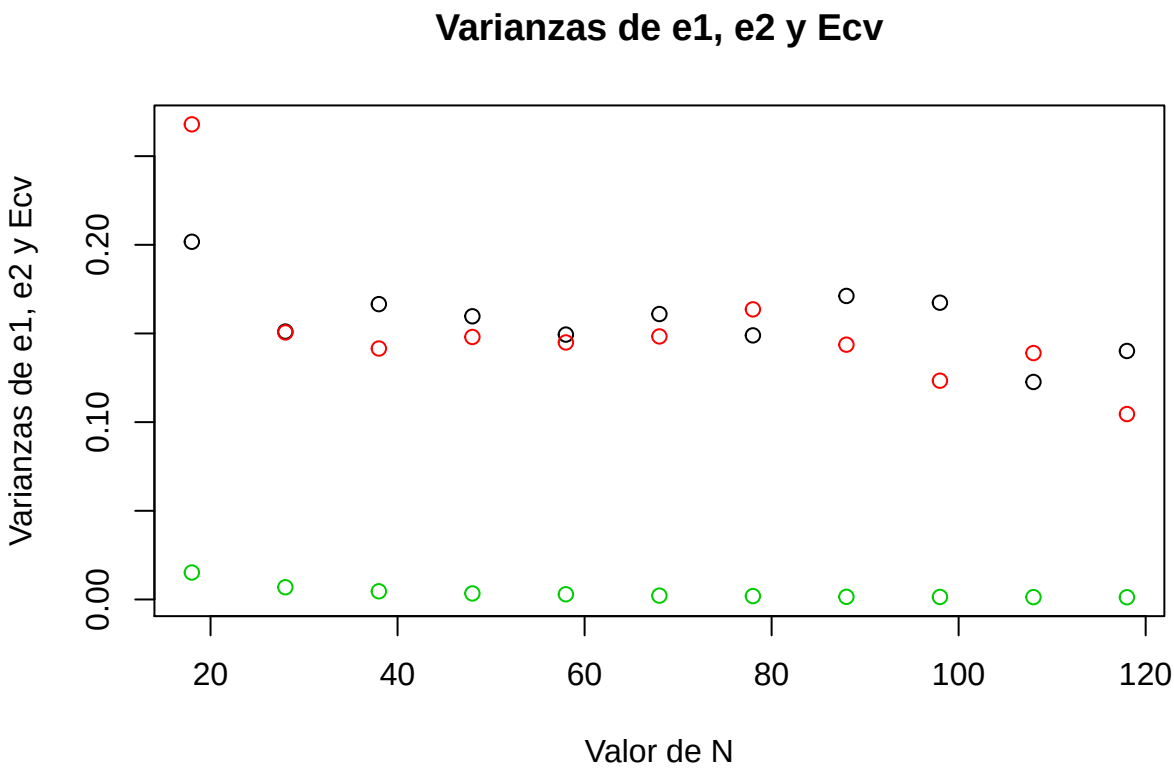
Pintamos los valores medios:

```
N <- c(rep(18,3), rep(28,3), rep(38,3), rep(48, 3), rep(58,3), rep(68,3),
       rep(78,3), rep(88,3), rep(98,3), rep(108,3), rep(118,3))
pinta_particion(N, unlist(errores), color, xlab="Valor de N",
                ylab="Valores medios de e1, e2 y Ecv",
                main="Valores medios de e1, e2 y Ecv")
```



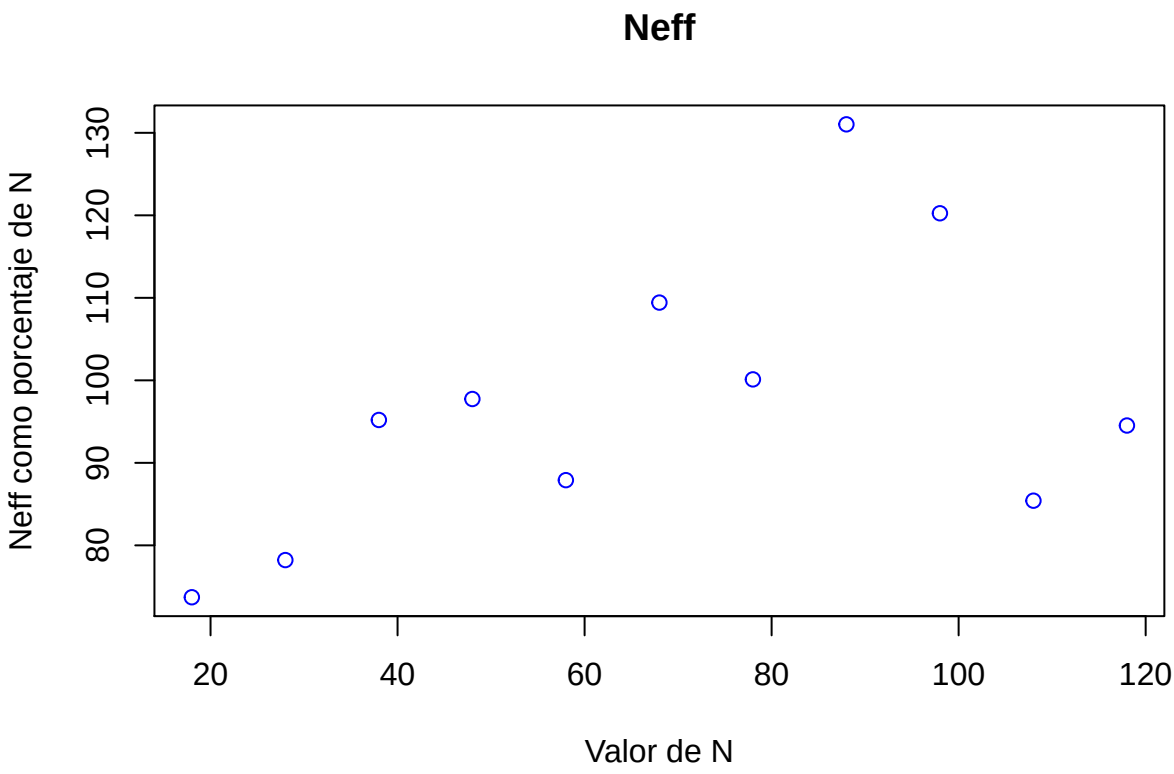
Como vemos los valores medios han aumentado un poco, mientras que antes los valores mínimos estaban en 0.26 o un poco por debajo ahora se quedan un poco por encima, con lo que hemos aumentado λ mucho, de forma que estamos ajustando un poco peor la función objetivo.
Veamos ahora las varianzas:

```
pinta_particion(N, unlist(varianzas), color, xlab="Valor de N",
                ylab="Varianzas de e1, e2 y Ecv",
                main="Varianzas de e1, e2 y Ecv")
```



Vemos que la varianza de e_1 ha subido un poco con respecto al experimento anterior, y para valores pequeños de N vemos que E_{cv} ha bajado un poco, de forma que el cociente aumenta, pero es muy poquito, con lo que no vamos a ver mucha diferencia en la gráfica de N_{eff}:

```
e_1 <- rapply(varianzas, function(x) x[1])
Ecv <- rapply(varianzas, function(x) x[3])
N <- seq(18, 118, by=10)
Neff <- 100*(e_1/Ecv)/N
pinta_particion(N, Neff, rep(1,11), main="Neff", xlab="Valor de N",
                ylab="Neff como porcentaje de N")
```



Efectivamente, ahora N_{eff} ha subido un poco, los valores están entre 70 y 130, pero es muy poquito y el valor que está en 130 es uno sólo.