Practica 1

Adrián Jesús Peña Rodríguez

March 23, 2018

Ejercicio 1:

Apartado 1

En este apartado se nos pedía implementar el algoritmo de gradiente descendente, en esta implementación he puesto como umbral de parada que el error con los pesos calculados sea menor que un umbral de parada pasado por parámetro a la función y también un número máximo de iteraciones. La implementación usa como función de error la derivada de f

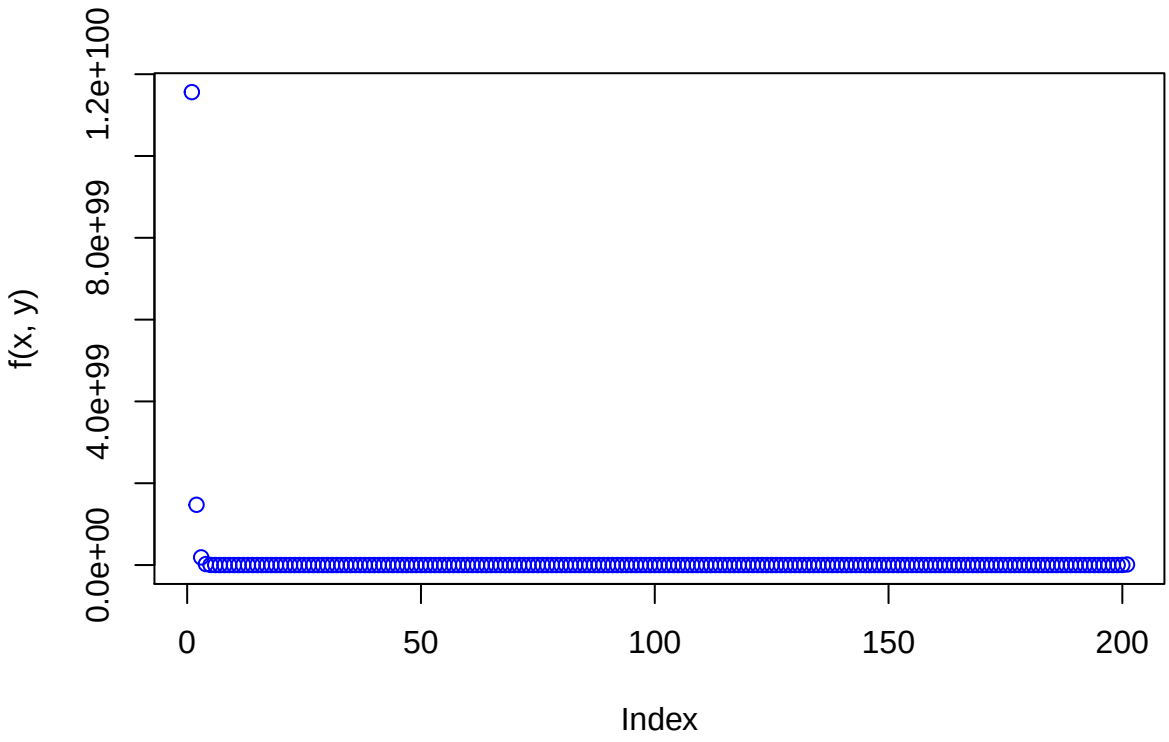
```
# Ejercicio 1.1:

# APARTADO A

# Algoritmo gradiente descendente para una variable, ejercicio 1.a

GD = function(f, fp, wini, nitr = 10000, nu, umbral_parada) {
  numero_de_iteraciones = 0

  while(numero_de_iteraciones < nitr & f(wini) > umbral_parada) {
    wini = wini - nu * fp(wini)

    numero_de_iteraciones = numero_de_iteraciones + 1
  }

  print(sprintf("Iteraciones = %s", numero_de_iteraciones))
  print(sprintf("Minimo en %s , %s", wini))
  print(sprintf("Mínimo encontrado = %s", f(wini)))
}


##################################################
```

Apartado 2

Para empezar declaramos la función que nos indican en el ejercicio y su derivada respecto a u y v pues nuestro gradiente descendente necesita de las derivadas para calcular el error. fdu = funcionDerivadaU, fdv = funcionDerivadaV

```
# Ejercicio 1.2

# Declaramos la función y sus derivadas parciales

f = function(u, v) {
  (u^3 * exp(v-2) - 4 * v^3 * exp(-u))^2
}

fdu = function(u, v) {
```

```
  2*(u^3 * exp(v-2) - 4*v^3 * exp(-u)) * (3*exp(v-2) * u^2 + 4*v^3*exp(-u))
}

fdv = function(u, v) {
  2*(u^3 * exp(v-2) - 4*v^3 * exp(-u)) * (u^3*exp(v-2) - 12*exp(-u)*v^2)
}

x = c(-100:100)
y = c(-100:100)
plot(f(x,y),col="blue")
```



Ahora reajustamos nuestro algoritmo de gradiente descendente para que este acepte nuestra nueva función, para ello simplemente tenemos en cuenta que ahora son dos derivadas y no una, como umbral de para tenemos que es hasta que nuestra funcion con nuestros pesos como u y v sea menor que el umbral puesto por parámetro. Usamos la propia función como error, el error a revasar será 10*10^-14, minimizaremos nuestro error a medida que hagamos iteraciones. Llamamos a la función con una tase de aprendizaje de 0.05 pues con 0.1 nunca llega a converger, debido a esto tambien le ponemos otra condición de parada que es número de iteraciones máximas. Podemos ver en cuantas iteraciones el error es menor que el seleccionado así como el mínimo encontrado y sus pesos asociados. Esta función converge entorno a 0 como podemos apreciar en la gráfica anterior y como nos hace ver nuestro gradiente descendente.

```
# Implementamos el algoritmo de gradiente descendente para dos variables

GD2 = function(f, fdu, fdv, wini = c(0, 0), nitr = 100000, nu, umbral_parada) {
  numero_de_iteraciones = 0
  wini_x = c()
  wini_y = c()
  guardarwini = wini

  while(numero_de_iteraciones < nitr & umbral_parada < f(wini[1], wini[2])) {
    wini_ahora = wini
```

```
    wini[1] = wini_ahora[1] - nu * fdu(wini_ahora[1], wini_ahora[2])
    wini[2] = wini_ahora[2] - nu * fdv(wini_ahora[1], wini_ahora[2])

    wini_x[numero_de_iteraciones + 1] = wini[1]
    wini_y[numero_de_iteraciones + 1] = wini[2]

    numero_de_iteraciones = numero_de_iteraciones + 1
  }

  print(sprintf("Punto de comienzo = %s , %s", guardarwini[1], guardarwini[2]))
  print(sprintf("Iteraciones = %s", numero_de_iteraciones))
  print(sprintf("Minimo en %s , %s", wini[1], wini[2]))
  print(sprintf("Mínimo encontrado = %s", f(wini[1], wini[2])))

  cbind(wini_x, wini_y)
}

borrar = GD2(f, fdu, fdv, c(1, 1), 100000, 0.05, 10*10^(-14))
```

```
## [1] "Punto de comienzo = 1 , 1"
## [1] "Iteraciones = 36"
## [1] "Minimo en 1.11954394631441 , 0.653988010291244"
## [1] "Mínimo encontrado = 4.9960111522217e-14"
```

```
##################################################
```

Apartado 3

En este apartado volvemos a declarar las nuevas funciones reescribiendo las anteriores

```
# Ejercicio 1.3

# Declaramos la función y sus derivadas parciales

f = function(x, y) {
  (x - 2)^2 + 2*(y + 2)^2 + 2 * sin(2*pi*x)*sin(2*pi*y)
}

fdx = function(x, y) {
  4*pi*cos(2*pi*x)*sin(2*pi*y)+2*(x-2)
}

fdy = function(x, y) {
  4*pi*sin(2*pi*x)*cos(2*pi*y)+4*(y+2)
}
```

Aquí básicamente llamamos a nuestro gradiente descendente con nuestra nueva función y su derivada y una tasa de aprendizaje de 0.01 y un máximo de 50 iteraciones y luego con una tasa de 0.1, como mínimo a alcanzar ponemos -5 para que así llegue a hacer las 50 iteraciones pues ese mínimo nunca lo podrá alcanzar.

```
old.par = par(mfrow=c(1,2))
print("Punto de comienzo (1, 1)")
```

```
## [1] "Punto de comienzo (1, 1)"
```

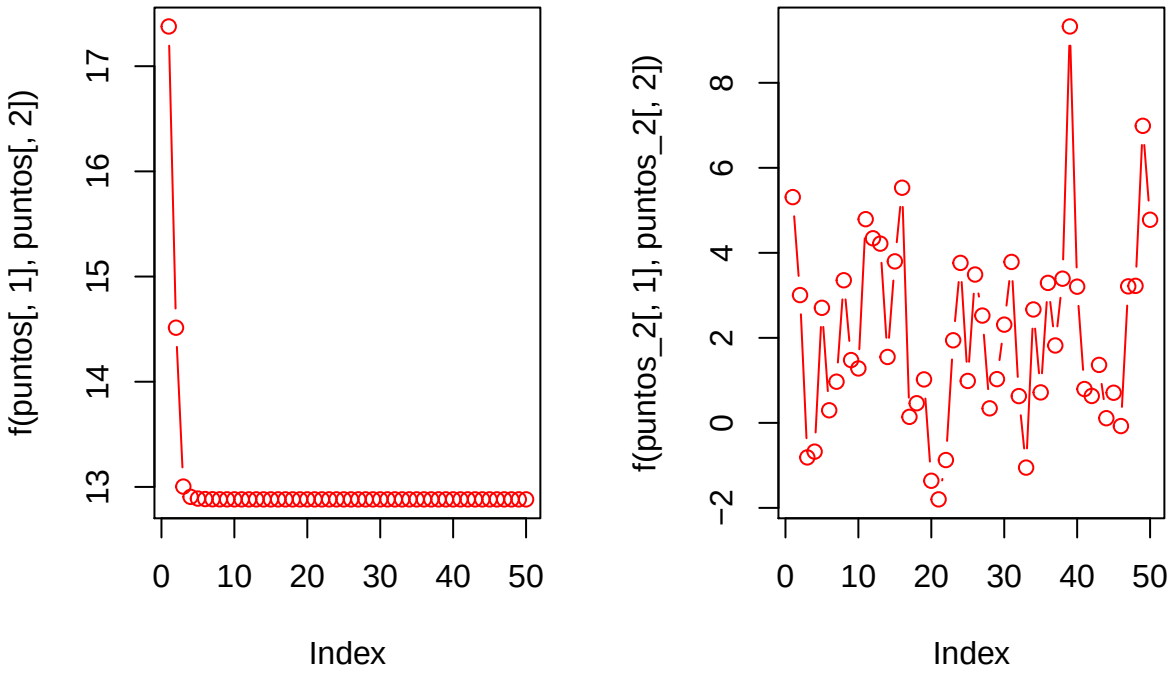
```r
puntos = GD2(f, fdx, fdy, c(1, 1), 50, 0.01, -5)
```

```
## [1] "Punto de comienzo = 1 , 1"
## [1] "Iteraciones = 50"
## [1] "Minimo en 1.28404319975679 , 0.590183743073683"
## [1] "Mínimo encontrado = 12.8815621880287"
```

```r
plot(f(puntos[,1], puntos[,2]), col = "red", type = "b")
puntos_2 = GD2(f, fdx, fdy, c(1, 1), 50, 0.1, -5)
```

```
## [1] "Punto de comienzo = 1 , 1"
## [1] "Iteraciones = 50"
## [1] "Minimo en 0.159665587078859 , -1.61187257831042"
## [1] "Mínimo encontrado = 4.77828220217578"
```

```r
plot(f(puntos_2[,1], puntos_2[,2]), col = "red", type = "b")
```



En el gráfico de la izquierda podemos ver como la el valor de la función desciende con una tasa de aprendizaje de 0.01, esto quiere decir que vamos bajando muy poco a poco con la ventaja de explorar muy detenidamente una sección de la función(hasta que llegue al número de iteraciones máximo o revasemos negativamente el umbral de parada). Por otro lado en el gráfico de la derecha vemos como con una tasa de aprendizaje de 0.1 nos saltamos el mínimo que en tasa de 0.01 si que explorabamos y empieza a variar entre distintos puntos para al final en 50 iteraciones dar un mejor resultado que con nu = 0.01. Esto depende de como sea nuestra función deberemos de ajustar nuestra tasa de aprendizaje. En la gráfica de la derecha llegamos a explorar si nos fijamos en los límites mucho más allá que con tasa 0.01. Básicamente con tasa 0.1 avanzamos 10 veces más rápido que con 0.01 y esto en este caso se traduce en una mayor exploración y un mínimo mejor en este caso. Podemos apreciar que esta función no converge y tiene forma de "huevera", al no converger dependiendo de la tasa de aprendizaje obtendremos un mínimo u otro.

Ahora vamos a ver con otros puntos de comienzo y vemos como se comporta la función con los mismos parámetros pero distinto punto de inicio para así compararlos

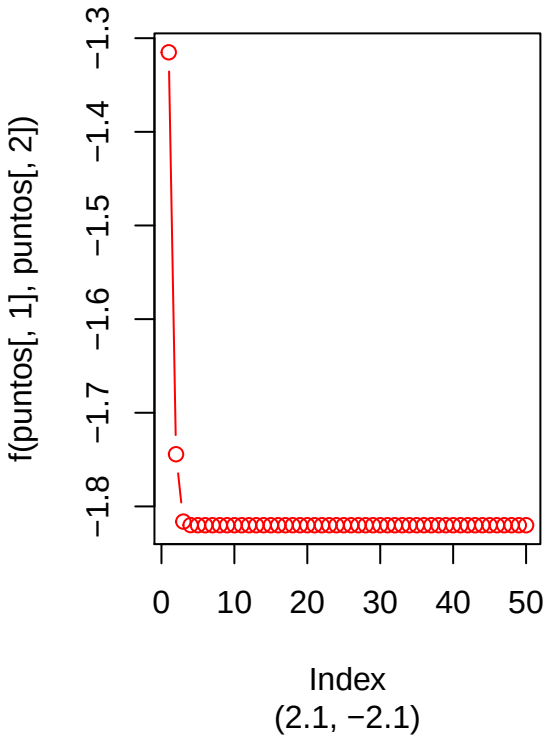
```r
old.par = par(mfrow=c(1,2))
puntos = GD2(f, fdx, fdy, c(2.1, -2.1), 50, 0.01, -5)
```

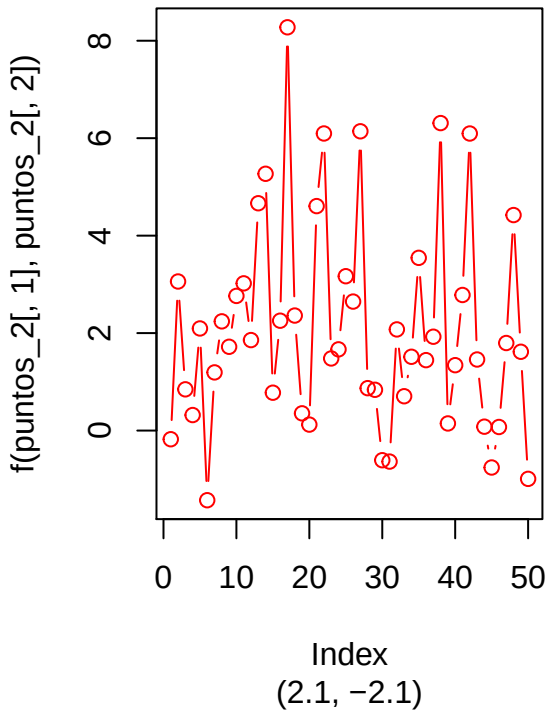
```
## [1] "Punto de comienzo = 2.1 , -2.1"
## [1] "Iteraciones = 50"
## [1] "Minimo en 2.24380496936479 , -2.23792582148618"
## [1] "Mínimo encontrado = -1.82007854154716"
```

```
plot(f(puntos[,1], puntos[,2]), col = "red", type = "b", sub="(2.1, -2.1)")
puntos_2 = GD2(f, fdx, fdy, c(2.1, -2.1), 50, 0.1, -5)
```

```
## [1] "Punto de comienzo = 2.1 , -2.1"
## [1] "Iteraciones = 50"
## [1] "Minimo en 1.64812722359196 , -1.65879794436579"
## [1] "Mínimo encontrado = -0.991195045742926"
```

```
plot(f(puntos_2[,1], puntos_2[,2]), col = "red", type = "b", sub="(2.1, -2.1)")
```





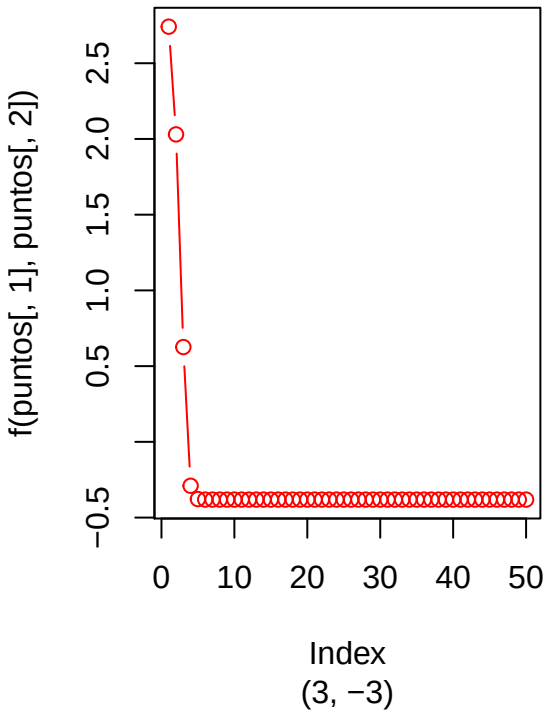

```
puntos = GD2(f, fdx, fdy, c(3, -3), 50, 0.01, -5)
```

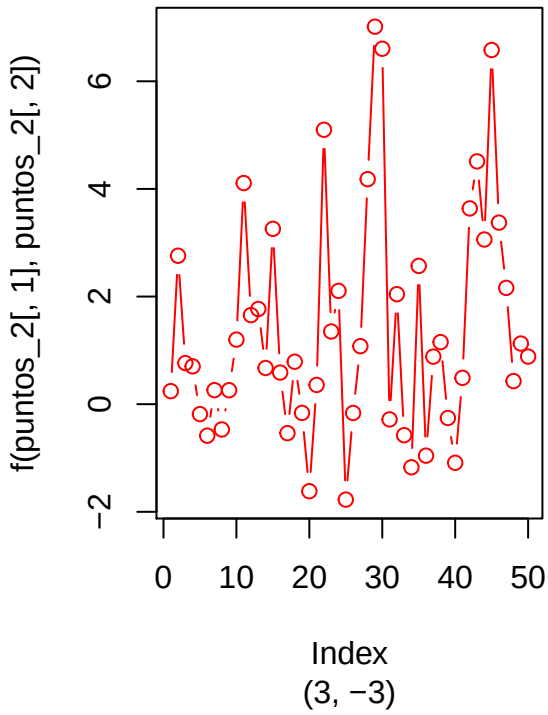
```
## [1] "Punto de comienzo = 3 , -3"
## [1] "Iteraciones = 50"
## [1] "Minimo en 2.73093564824811 , -2.7132791261667"
## [1] "Mínimo encontrado = -0.3812494974381"
```

```
plot(f(puntos[,1], puntos[,2]), col = "red", type = "b", sub="(3, -3)")
puntos_2 = GD2(f, fdx, fdy, c(3, -3), 50, 0.1, -5)
```

```
## [1] "Punto de comienzo = 3 , -3"
## [1] "Iteraciones = 50"
## [1] "Minimo en 2.32256315518421 , -2.52110723603622"
## [1] "Mínimo encontrado = 0.884602729550448"
```

```
plot(f(puntos_2[,1], puntos_2[,2]), col = "red", type = "b", sub="(3, -3)")
```





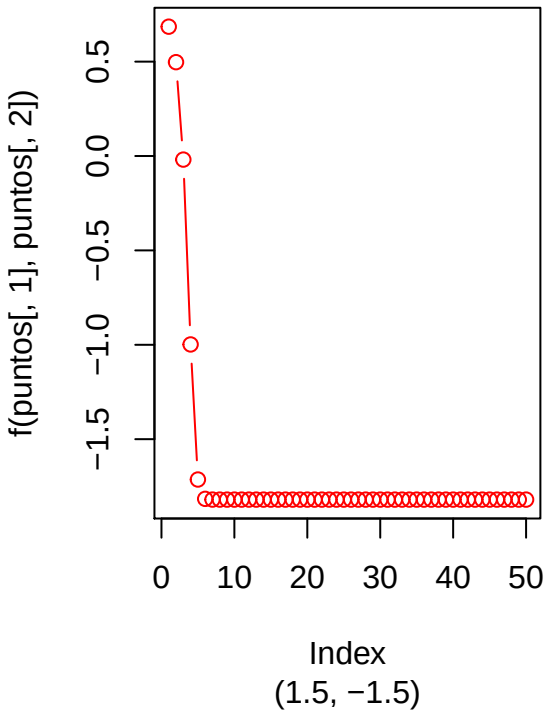

```
puntos = GD2(f, fdx, fdy, c(1.5, -1.5), 50, 0.01, -5)
```

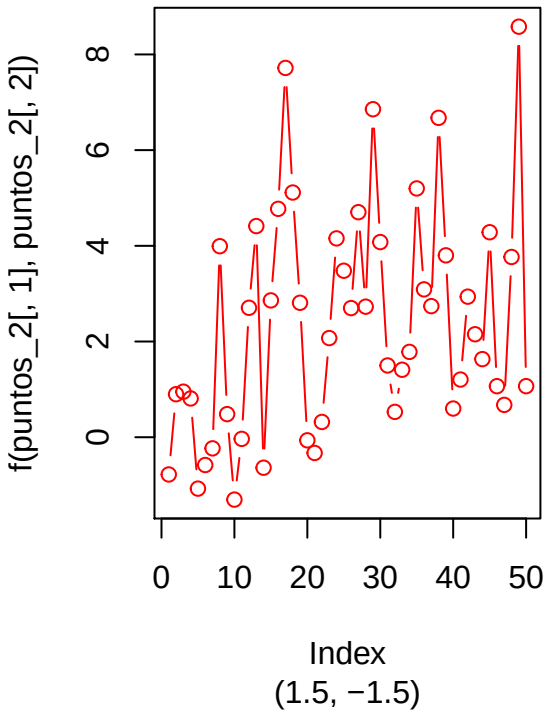
```
## [1] "Punto de comienzo = 1.5 , -1.5"
## [1] "Iteraciones = 50"
## [1] "Minimo en 1.75619503063521 , -1.76207417851382"
## [1] "Mínimo encontrado = -1.82007854154716"
```

```
plot(f(puntos[,1], puntos[,2]), col = "red", type = "b", sub="(1.5, -1.5)")
puntos_2 = GD2(f, fdx, fdy, c(1.5, -1.5), 50, 0.1, -5)
```

```
## [1] "Punto de comienzo = 1.5 , -1.5"
## [1] "Iteraciones = 50"
## [1] "Minimo en 3.31090270916109 , -2.4105979427332"
## [1] "Mínimo encontrado = 1.06739493154394"
```

```
plot(f(puntos_2[,1], puntos_2[,2]), col = "red", type = "b", sub="(1.5, -1.5)")
```





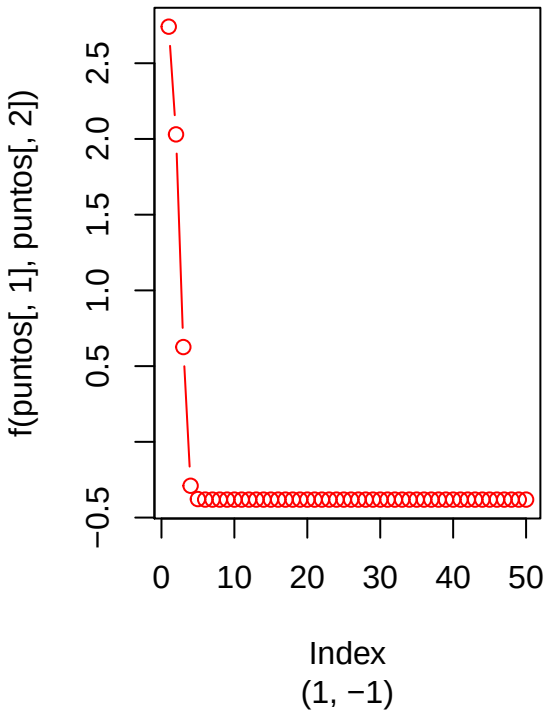

```
puntos = GD2(f, fdx, fdy, c(1, -1), 50, 0.01, -5)
```

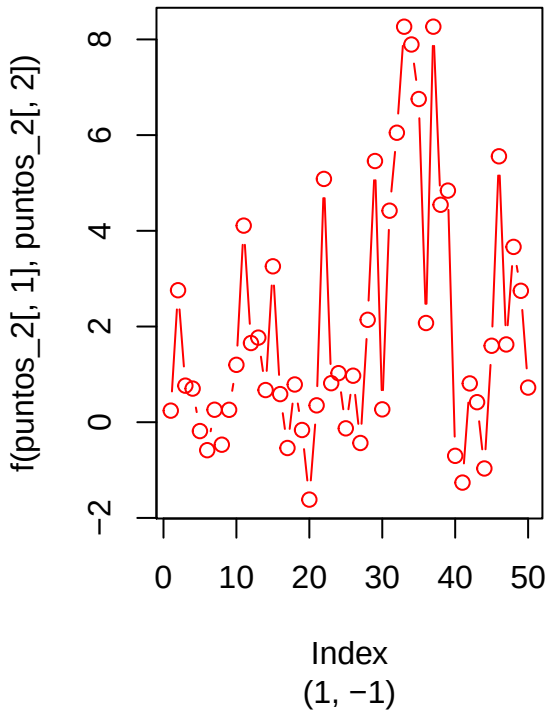
```
## [1] "Punto de comienzo = 1 , -1"
## [1] "Iteraciones = 50"
## [1] "Minimo en 1.26906435175189 , -1.2867208738333"
## [1] "Mínimo encontrado = -0.3812494974381"
```

```
plot(f(puntos[,1], puntos[,2]), col = "red", type = "b", sub="(1, -1)")
puntos_2 = GD2(f, fdx, fdy, c(1, -1), 50, 0.1, -5)
```

```
## [1] "Punto de comienzo = 1 , -1"
## [1] "Iteraciones = 50"
## [1] "Minimo en 2.9393764798167 , -1.60779542243539"
## [1] "Mínimo encontrado = 0.724114942499606"
```

```
plot(f(puntos_2[,1], puntos_2[,2]), col = "red", type = "b", sub="(1, -1)")
```






```
rm(puntos, puntos_2, old.par)

####################################################
```

Apartado 4

La conclusión es que es díficil encontrar un mínimo global para una función arbitraria porque es posible saltarlo o nunca llegar a el pues nunca tenemos la certeza de que el mínimo que encontremos sea global o no. Solo en algunas funciones podremos encontrar un mínimo global y asegurar que es ese, por ejemplo en f(x,y) = x^2+y^2. En los casos que hemos visto anteriormente vemos que cuando variamos la tasa de aprendizaje los mínimos obtenidos para una misma función varián, esto se debe a que en función de ese nu avanzaremos en el dominio de la función de una manera u otra. La idea es encontrar el nu que mejores resultados nos de, teniendo en cuenta que un nu más alto da cambios más bruscos y un nu más pequeño cambios menos bruscos. Digamos que un nu bajo puede no ser capaz de salir de un mínimo y perderse un mínimo mejor y en contra posición un nu alto puede saltarse el mejor mínimo.

Ejercicio 2:

Funciones útiles y proporcionadas

La primera función nos sirve para calcular la simetría de nuestros datos. La siguiente es básicamente para a traves de unos pesos sacar la recta que divide a nuestros datos. A continuación tenemos una función que nos devuelve los pesos calculados mediante la pseudo inversa, para ello le aplica la pseudo inversa a los datos y posteriormente multiplicandolá por los dígitos. Por último, tenemos la función que nos devuelve la predicción que hacemos a traves de los datos y los pesos calculados, tan solo devolvemos el signo pues es lo que nos interesa para clasificar nuestros datos con etiquetas (-1,1)

```
# Ejercicio 2:
# Función que calcula la simetria de cada ejemplo de nuestro dataset
fSimetria <- function(A){
```

```
  A = abs(A-A[,ncol(A):1])
  -sum(A)
}

pasoARecta = function(w){
  if(length(w)!= 3)
    stop("Solo tiene sentido con 3 pesos")
  a = -w[1]/w[2]
  b = -w[3]/w[2]
  c(a,b)
}

# Regress_Lin, para la obtención de pesos del modelo lineal
Regress_Lin = function(datos, digitos){
  pseudo = svd(datos)
  pinversa = pseudo$v %*% diag(1/pseudo$d) %*% t(pseudo$u) # Formula para obtener la pseudo inversa
  return (pinversa %*% digitos)
}

# Funcion que devuelve unas predicciones en funcion de los pesos hallados anteriormente
h = function(w, x) {
  sign(t(w) %*% t(x))
}
```

Primero preparamos los datos que vamos a usar, abrimos nuestros datos de entrenamiento y test y seleccionamos solo los 1 y 5 de ambos, borramos las variables que ya no necesitamos.

```
# Preparacion de los datos
# 1º. Lectura zip del train y del test y asignación a una variable

digit.train <- read.table("datos/zip.train", quote="\"", comment.char="", stringsAsFactors=FALSE)
```

```
## Warning in scan(file = file, what = what, sep = sep, quote = quote, dec =
## dec, : number of items read is not a multiple of the number of columns
```

```
digit.test <- read.table("datos/zip.test", quote="\"", comment.char="", stringsAsFactors=TRUE)
```

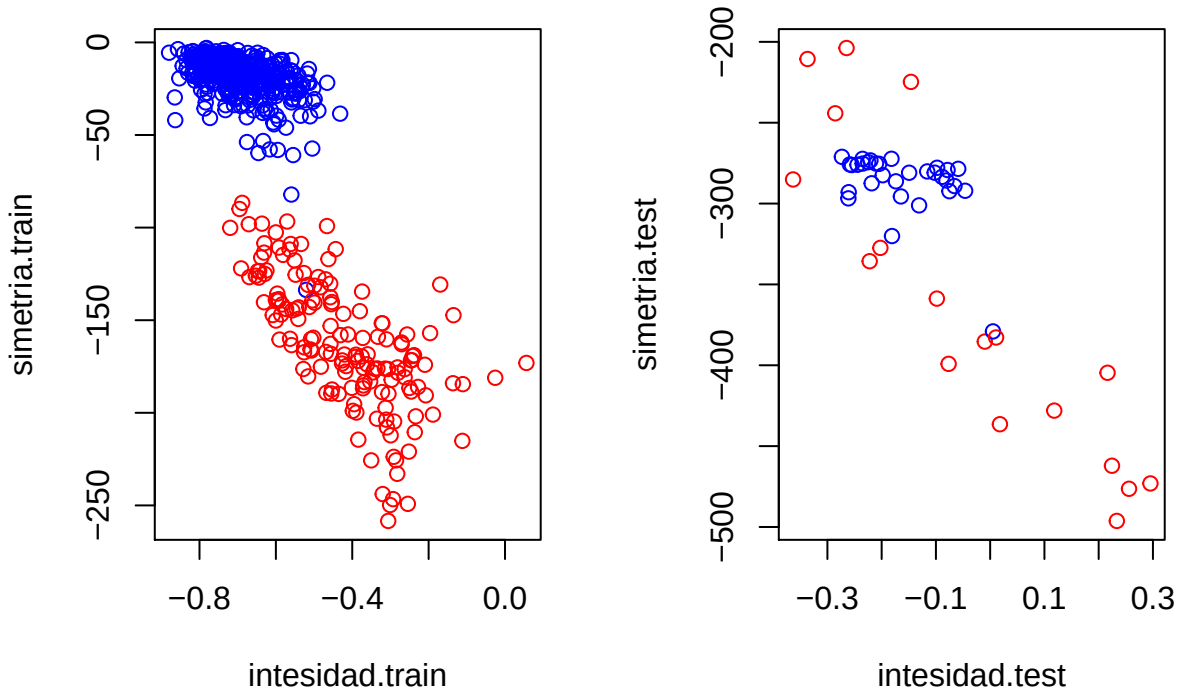
```
## Warning in scan(file = file, what = what, sep = sep, quote = quote, dec =
## dec, : number of items read is not a multiple of the number of columns
```

```
# Ahora nos quedamos solamente con los 1 y los 5
digitos15.train = digit.train[digit.train$V1==1 | digit.train$V1==5,]
digitos15.test = digit.test[digit.test$V1==1 | digit.test$V1==5,]

digitos.train = digitos15.train[,1]    # vector de etiquetas(clases) del train, en este caso columna 1
ndigitos.train = nrow(digitos15.train)  # numero de muestras del train
digitos.test = digitos15.test[,1]    # vector de etiquetas(clases) del test, en este caso columna 1
ndigitos.test = nrow(digitos15.test)  # numero de muestras del test

# se retira la clase y se monta una matriz 3D: 599*16*16
grises.train = array(unlist(subset(digitos15.train,select=-V1)),c(ndigitos.train,16,16)) # Cada numero
rm(digit.train)
rm(digitos15.train)
# Ahora con el test
grises.test = array(unlist(subset(digitos15.test,select=-V1)),c(ndigitos.test,16,16))
rm(digit.test)
```

```
rm(digitos15.test)
```

Ahora calculamos la media de las dos muestras y la simetría usando la función que se nos proporciono. Luego cambiamos la etiqueta de 5 por -1 para así clasificar entre 1 y -1 y usar dicha etiqueta en nuestras fórmulas y por último unimos los datos de intensidad y simetría de cada uno de los números los cuales están almacenados cada uno en una matriz 16x16. Y ya trabajamos con estos datos. Por último, introdicimos el término independiente como una columna nueva de nuestro dataset, este serán todo 1.

```
# 2º. Vamos a obtener la intensidad media de nuestro dataset:

intesidad.train = apply(X = grises.train, MARGIN = 1, FUN = mean) # Intensidad de cada uno de los numer
intesidad.test = apply(X = grises.test, MARGIN = 1, FUN = mean)

# 3º. Vamos a obtener la simetria respecto al eje vertical de nuestro dataset:

simetria.train = apply(X = grises.train, MARGIN = 1, FUN = fSimetria) # Simetria de cada uno de los num
simetria.test = apply(X = grises.test, MARGIN = 1, FUN = fSimetria) # Simetria de cada uno de los numer

rm(grises.train) # Liberamos memoria pues ya no necesitamos a grises
rm(grises.test)

# 4º. Vamos a recodificar nuestras etiquetas, cambiando el 5 por -1

digitos.train[digitos.train == 5] = -1
digitos.test[digitos.test == 5] = -1

# 5º. Componemos datosTr, que es la unión por columnas de la intensidad con la simetria, columna 1 = in

datosTr = as.matrix(cbind(intesidad.train, simetria.train))
datosTst = as.matrix(cbind(intesidad.test, simetria.test))

# Introducimos la variable independiente (1) en nuestros datos de intesidad y simetria
indep = vector("numeric", length = ndigitos.train)
indep_test = vector("numeric", length = ndigitos.test)
indep[indep == 0] = 1
indep_test[indep_test == 0] = 1
datosTr = cbind(datosTr, indep)
datosTst = cbind(datosTst, indep_test)

old.par = par(mfrow=c(1,2))
plot(intesidad.train, simetria.train, col = digitos.train + 3)
plot(intesidad.test, simetria.test, col = digitos.test + 3)
```



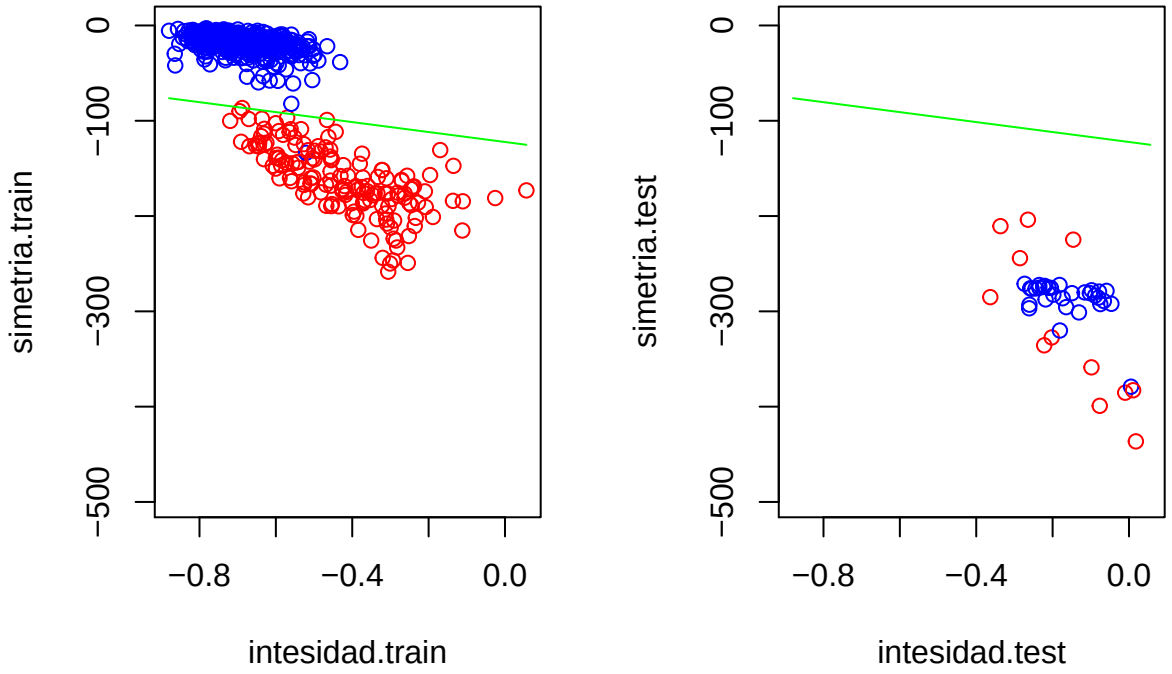

```
rm(indep, indep_test)
```

Ahora vamos a calcular la resta de regresión que nos dividirá de la mejor manera posible y consiguiendo clasificar correctamente los maximos digitos posibles. Para ello vamos a usar la pseudo inversa la cual he descrito anteriormente. Una vez calculados los pesos a traves de la pseudo inversa, vemos cuantos aciertos se pueden tener con esos pesos, esta será nuestra predicción. Luego en base a nuestra predicción vemos el número de fallos que hemos tenido y obtenemos el error de los datos del train, el cual es realmente bajo pues solo clasifica mal uno de los dígitos. Hacemos lo mismo para el test y comprobamos que tiene un error muy alto, esto se debe a que los datos del test están totalmente alejados de los datos del train. Debido a esto nuestra recta de regresión no funciona bien en los datos de test y sin embargo esta perfectamente ajustada para los datos de train.

Para poder predecir con un error aceptable estos datos, tendríamos que hacer trasformaciones a los datos como puede ser elevar al cuadrado o alguna otra ya que si nos fijamos para englobar a los datos azules, necesitamos que la recta de regresión sea un círculo que englobe a estos datos y deje a lo otros fuera, nos valdría tanto para el train como para el test.

```
# REGRESION CON PSEUDO-INVERSA

pesos_2a_svd = Regress_Lin(datosTr, digitos.train)  # Calculo de los pesos mediante la pseudo-inversa

prediccion_train_2a_svd = h(pesos_2a_svd, datosTr) # Atraves de los pesos hallados con anterioridad dam

fallos_train_2a_svd = prediccion_train_2a_svd == digitos.train   # Vemos cuantos aciertos hemos tenido

Ein_2a_svd = length(fallos_train_2a_svd[fallos_train_2a_svd != 1]) / ndigitos.train  # Elegimos solo lo

prediccion_test_2a_svd = h(pesos_2a_svd, datosTst) # Volvemos a repetir para el test y así poder obtene

fallos_test_2a_svd = prediccion_test_2a_svd == digitos.test

Eout_2a_svd = length(fallos_test_2a_svd[fallos_test_2a_svd != 1]) / ndigitos.test
```

```r
# Imprimimos por pantalla las dos gráficas y la recta de regresión donde podemos ver que ocurre
old.par = par(mfrow=c(1,2))
plot(intesidad.train, simetria.train, col = digitos.train + 3, type = "p", xlim = c(min(datosTr[,1]),ma
x = c(min(datosTr[,1]),max(datosTr[,1]))
y = c(pasoARecta(pesos_2a_svd)[1]*x[1] + pasoARecta(pesos_2a_svd)[2], pasoARecta(pesos_2a_svd)[1]*x[2]
lines(x, y, type = "l", col = "green")

plot(datosTst, col = digitos.test + 3, type = "p", xlim = c(min(datosTr[,1]),max(datosTr[,1])), ylim =
x = c(min(datosTr[,1]),max(datosTr[,1]))
y = c(pasoARecta(pesos_2a_svd)[1]*x[1] + pasoARecta(pesos_2a_svd)[2], pasoARecta(pesos_2a_svd)[1]*x[2]
lines(x, y, type = "l", col = "green")
```

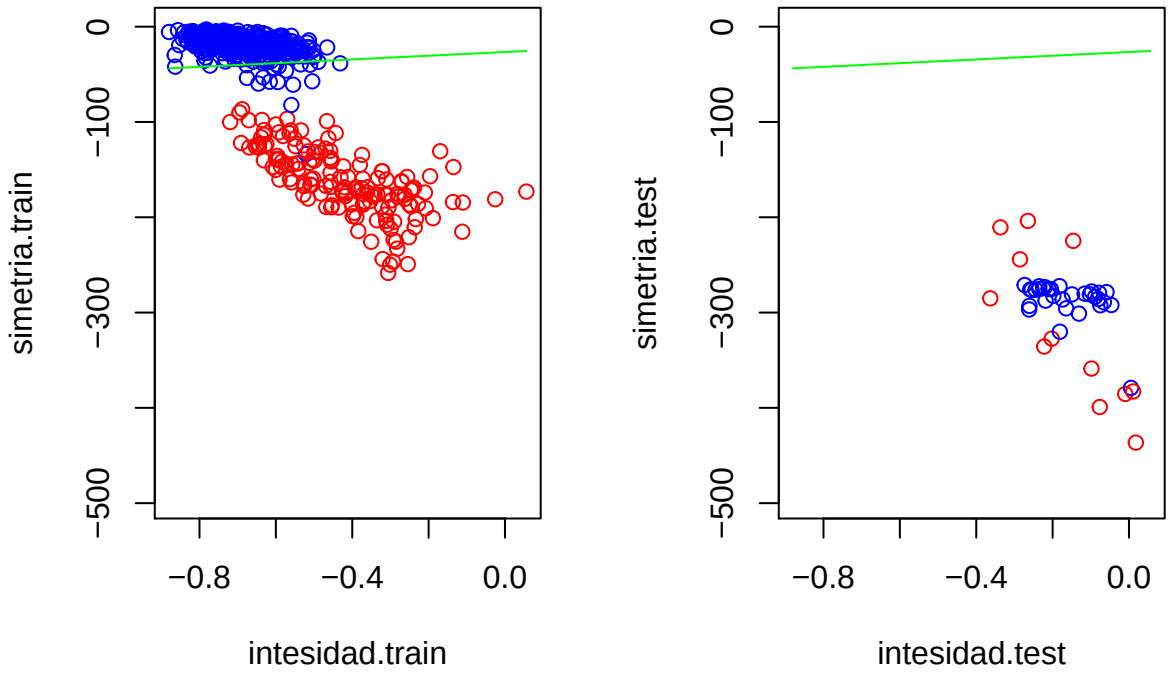



```r
rm(x,y)
```

Ahora vamos a implementar nuestro algoritmo de gradiente descendente estocástico el cual es un gradiente descendente el cual no toma todas las muestras, sino un subconjunto aleatorio de estas que es un porcentaje del total del dataset. Hacemos lo mismo que con la pseudo inversa pero esta vez con el gradiente descendente estocástico, obtenemos unos resultados realmente buenos, con dos fallos unicamente en el train, uno más que con la pseudo inversa pero en el test obtenemos exactamente los mismos resultados en cuanto al error. Podemos apreciar lo que ha ocurrido observando las gráficas.

```r
# Gradiente descendiente estocástico

SGD = function(datos, resultados, wini = c(0, 0, 0), nitr = 100000, nu, tminibatches) {
  numero_de_iteraciones = 0
  sumatoria = c(1,1,1)
  N = as.integer(length(resultados) * tminibatches)

  for(i in 1:nitr) {
    for(i in 1:N) {
      aleatorio = sample(length(resultados),1)
      sumatoria = sumatoria - (as.vector(resultados[aleatorio] * datos[aleatorio,])) * (as.vector((exp(
    }
```

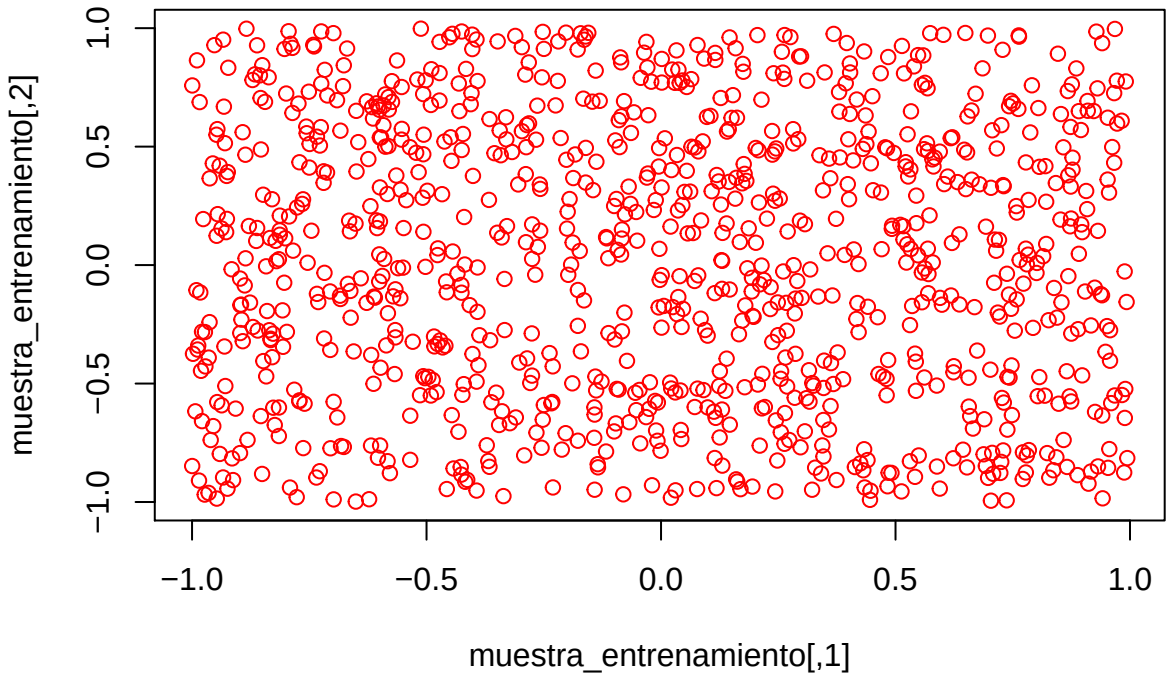
```
    sumatoria = sumatoria / N
    wini = wini - nu * sumatoria
    numero_de_iteraciones = numero_de_iteraciones + 1
  }

  wini
}

# REGRESION CON GRADIENTE DESCENDENTE ESTOCÁSTICO
pesos_2a_sgd = SGD(datosTr, digitos.train, c(0, 0, 0), 100, 0.1, 0.2)  # Calculo de los pesos mediante

prediccion_train_2a_sgd = h(pesos_2a_sgd, datosTr) # Atraves de los pesos hallados con anterioridad dam

fallos_train_2a_sgd = prediccion_train_2a_sgd == digitos.train   # Vemos cuantos aciertos hemos tenido

Ein_2a_sgd = length(fallos_train_2a_sgd[fallos_train_2a_sgd != 1]) / ndigitos.train  # Elegimos solo lo

prediccion_test_2a_sgd = h(pesos_2a_sgd, datosTst) # Volvemos a repetir para el test y así poder obtene

fallos_test_2a_sgd = prediccion_test_2a_sgd == digitos.test

Eout_2a_sgd = length(fallos_test_2a_sgd[fallos_test_2a_sgd != 1]) / ndigitos.test

# Imprimimos por pantalla las dos gráficas y la recta de regresión donde podemos ver que ocurre
old.par = par(mfrow=c(1,2))
plot(intesidad.train, simetria.train, col = digitos.train + 3, type = "p", xlim = c(min(datosTr[,1]),ma
x = c(min(datosTr[,1]),max(datosTr[,1]))
y = c(pasoARecta(pesos_2a_sgd)[1]*x[1] + pasoARecta(pesos_2a_sgd)[2], pasoARecta(pesos_2a_sgd)[1]*x[2]
lines(x, y, type = "l", col = "green")

plot(datosTst, col = digitos.test + 3, type = "p", xlim = c(min(datosTr[,1]),max(datosTr[,1])), ylim =
x = c(min(datosTr[,1]),max(datosTr[,1]))
y = c(pasoARecta(pesos_2a_sgd)[1]*x[1] + pasoARecta(pesos_2a_sgd)[2], pasoARecta(pesos_2a_sgd)[1]*x[2]
lines(x, y, type = "l", col = "green")
```




```
rm(x,y)

# ^^
```

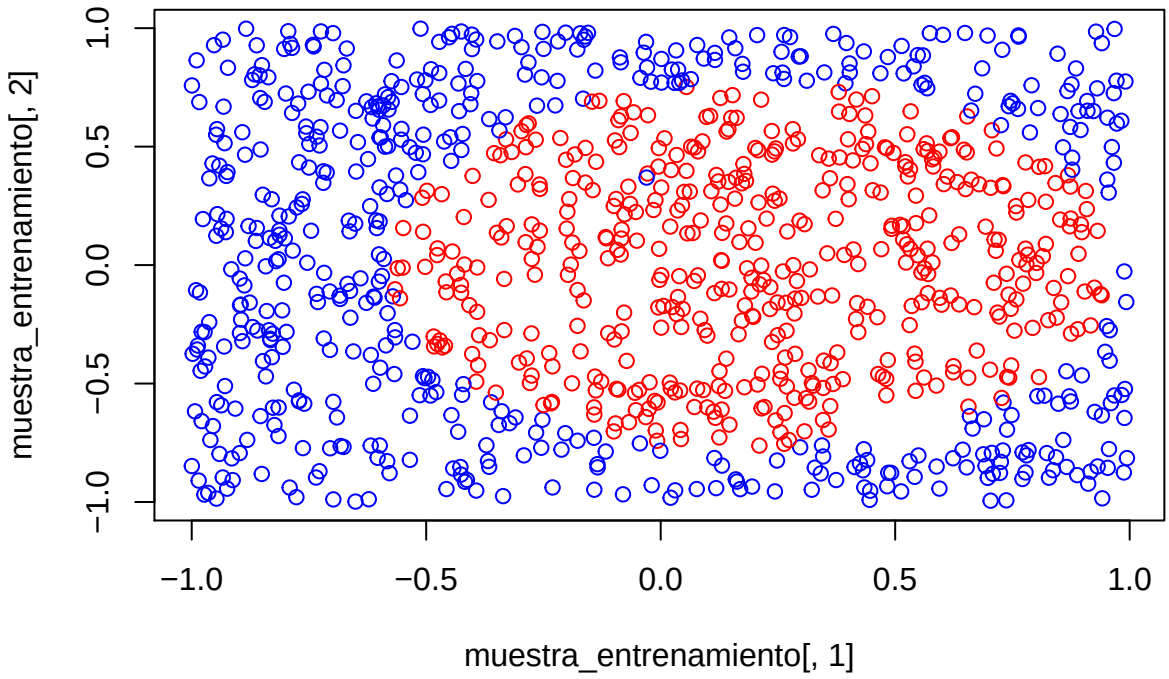
Para este apartado se nos pide generar una muestra de entrenamiento de N = 1000 puntos en el cuadrado X = [-1,1] * [-1,1], para ello hacemos uso de una función que se nos ha pasado en prácticas que lo hace por nosotros. Esta es la gráfica de los puntos generados:

```
# APARTADO 2

# por defecto genera 2 puntos(N) entre [0,1](rango) de 2 dimensiones(dims)

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

# 2.a
old.par = par(mfrow=c(1,1))
muestra_entrenamiento = simula_unif(1000, 2, c(-1,1))
plot(muestra_entrenamiento, col = "red")
```



A continuación insertamos ruido en nuestra muestra con una función que he creado y sustituimos el 10% de la muestra por ese ruido:

```
# 2.b

f = function(x1, x2) {
  sign((x1 - 0.2)^2 + x2^2 - 0.6)
}

genera_etiquetas_y_ruido = function(muestra) {
  devolver = c()
  N = dim(muestra)[1]

  for(i in 1:N) {
    devolver[i] = f(muestra[i,1],muestra[i,2])
  }

  aleatorios_d = sample(1:N) # Vamos a introducir ruido cambiando aleatoriamente el signo de un 10% de

  for(i in 1:N * 0.1) {
    devolver[aleatorios_d[i]] = devolver[aleatorios_d[i]] * -1
  }

  devolver
}

etiquetas = genera_etiquetas_y_ruido(muestra_entrenamiento)

old.par = par(mfrow=c(1,1))
plot(muestra_entrenamiento[,1], muestra_entrenamiento[,2], col = etiquetas + 3)
```



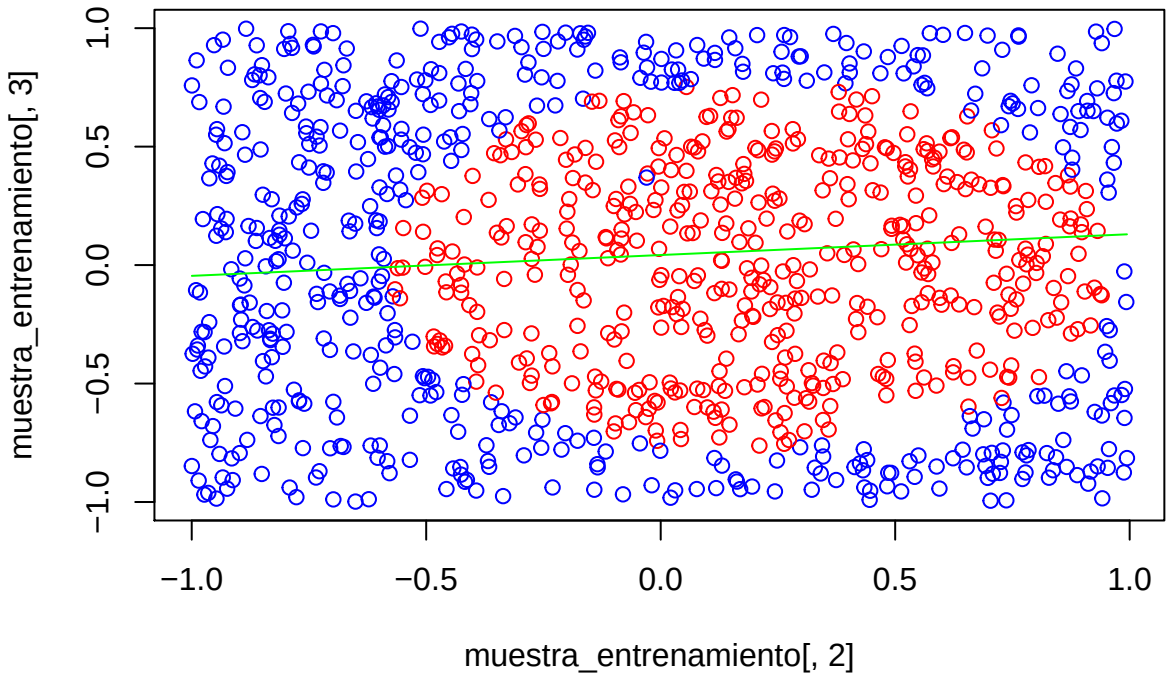

```
rm(old.par)
```

Ahora para este apartado tenemos que introducir el termino independiente el cual según nos indican en el enunciado tiene que estar en la primera columna, una vez insertado, ajustamos nuestro modelo de regresión con los datos generados y vemos el Ein usando el gradiente descendente:

```
# 2.c

indep = vector("numeric", length = length(etiquetas))
indep[indep == 0] = 1
muestra_entrenamiento = cbind(indep, muestra_entrenamiento)

pesos_2c_sgd = SGD(muestra_entrenamiento, etiquetas, c(0, 0, 0), 1000, 0.05, 0.2)  # Calculo de los pes

prediccion_2c_sgd = h(pesos_2c_sgd, muestra_entrenamiento) # Atraves de los pesos hallados con anterior

fallos_2c_sgd = prediccion_2c_sgd == etiquetas   # Vemos cuantos aciertos hemos tenido

Ein_2c_sgd = length(fallos_2c_sgd[fallos_2c_sgd != 1]) / length(etiquetas)  # Elegimos solo los fallos

print(sprintf("Ein = %s", Ein_2c_sgd))
```

```
## [1] "Ein = 0.355"
```

```
# Imprimimos por pantalla las dos gráficas y la recta de regresión donde podemos ver que ocurre
plot(muestra_entrenamiento[,2], muestra_entrenamiento[,3], col = etiquetas + 3, type = "p", xlim = c(mi
x = c(min(muestra_entrenamiento[,2]),max(muestra_entrenamiento[,2]))
y = c(pasoARecta(pesos_2c_sgd)[1]*x[1] + pasoARecta(pesos_2c_sgd)[2], pasoARecta(pesos_2c_sgd)[1]*x[2]
lines(x, y, type = "l", col = "green")
```



Podemos apreciar que el error que hemos obtenido es bastante elevado, la gráfica corroborá este error. Podemos apreciar que estos datos no se podrán ajustar con este modelo a menos que le hagamos algunas trasformaciones a los datos, necesitaríamos elevar al cuadrado en principio aunque habría que estudiarlo.

Por último nos piden que hagamos esto mismo pero para 1000 dataset de 1000 datos cada uno generados como en los apartados anteriores, calculamos los errores Ein e Eout los sumamos y dividimos entre 1000 que es el número total de dataset para así obtener la media de los errores con nuestro modelo de regresión.

```
# 2.d
# Generamos 1000 muestras de 1000 componentes cada una
muestra_entrenamiento_d = list()
indep = vector("numeric", length = length(etiquetas))
indep[indep == 0] = 1
Ein_2d_sgd = 0
Eout_2d_sgd = 0

for(i in 1:1000) {
  # Entrenamiento
  muestra_entrenamiento_d = simula_unif(1000, 2, c(-1,1))
  etiquetas_d = genera_etiquetas_y_ruido(muestra_entrenamiento_d)
  muestra_entrenamiento_d = cbind(indep, muestra_entrenamiento_d)
  pesos_2d_sgd = SGD(muestra_entrenamiento_d, etiquetas_d, c(0, 0, 0), 10, 0.05, 0.2)  # Calculo de los
  prediccion_2d_sgd = h(pesos_2d_sgd, muestra_entrenamiento_d) # Atraves de los pesos hallados con ante
  fallos_2d_sgd = prediccion_2d_sgd == etiquetas_d   # Vemos cuantos aciertos hemos tenido
  Ein_2d_sgd = Ein_2d_sgd + length(fallos_2d_sgd[fallos_2d_sgd != 1]) / length(etiquetas_d)  # Elegimos

  # Test
  muestra_entrenamiento_d = simula_unif(1000, 2, c(-1,1))
  etiquetas_d = genera_etiquetas_y_ruido(muestra_entrenamiento_d)
  muestra_entrenamiento_d = cbind(indep, muestra_entrenamiento_d)
  prediccion_2d_sgd = h(pesos_2d_sgd, muestra_entrenamiento_d) # Atraves de los pesos hallados con ante
  fallos_2d_sgd = prediccion_2d_sgd == etiquetas_d   # Vemos cuantos aciertos hemos tenido
  Eout_2d_sgd = Eout_2d_sgd + length(fallos_2d_sgd[fallos_2d_sgd != 1]) / length(etiquetas_d)  # Elegim
```

```
}

Ein_2d_sgd_media = Ein_2d_sgd / 1000
Eout_2d_sgd_media = Eout_2d_sgd / 1000

print(sprintf("Ein = %s", Ein_2d_sgd_media))
```

```
## [1] "Ein = 0.415349"
```

```
print(sprintf("Eout = %s", Eout_2d_sgd_media))
```

```
## [1] "Eout = 0.419811"
```

```
rm(Ein_2d_sgd, Eout_2d_sgd, i)

# Apartado e, explicar que no es un buen ajuste pues no conseguimos una buena tasa de acierto en el tra
```

Apartado 2.d

Es un ajuste muy malo ya que nos da un error muy alto, esto se debe a la forma que tienen nuestros datos los cuales no permiten que se pueda hacer una división de manera lineal, nuestrbmodelo lineal se ajusta lo mejor que pude para ser lineal, consigue el menor error posible para una linea que divida. Necesitamos hacerle trasformaciones como puede ser elevar al cuadrado ya que si nos fijamos podemos dividir los datos con un circulo.