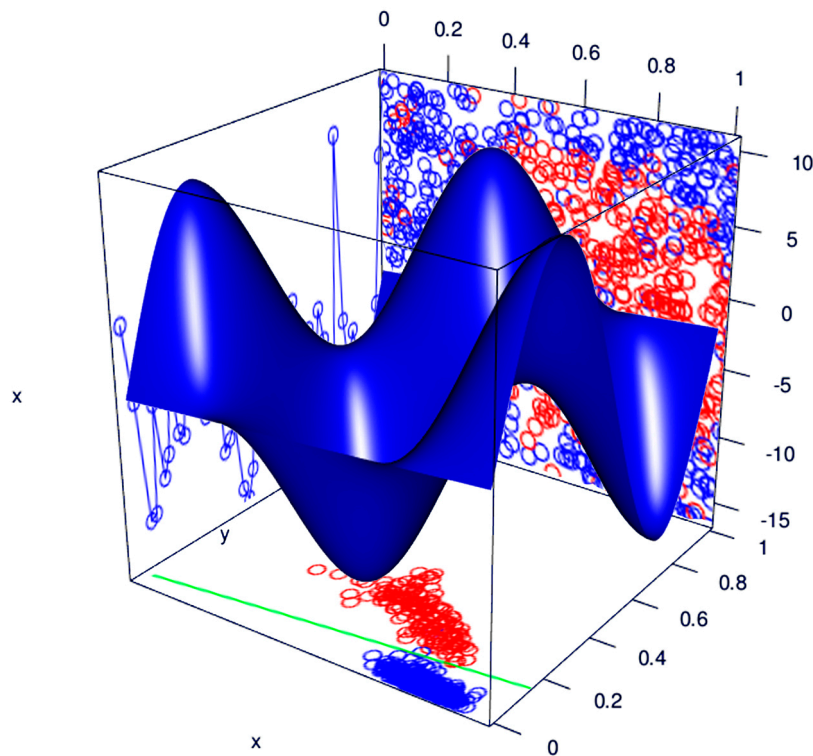


Práctica 1

Aprendizaje Automático

Julio A. Fresneda - juliofresnedag@correo.ugr.es



EJERCICIO SOBRE LA BÚSQUEDA ITERATIVA DE ÓPTIMOS

1. IMPLEMENTACIÓN DEL ALGORITMO DE GRADIENTE DESCENDIENTE.

El algoritmo Gradiente Descendiente es un algoritmo iterativo de estimación de parámetros que sirve para minimizar variables. Matemáticamente se describe así:

$$w_j = w_j - \eta \cdot \frac{\partial f(w)}{\partial w_j}$$

donde η es el learning rate, f la función a usar y w_j es el peso inicial a actualizar. Para cada j de w_0, w_1, \dots, w_k , el peso se actualiza simultáneamente.

Implementado el resultado es el siguiente.

```
#Función del GD para funciones de dos variables
GD <- function(f,dfu,dfv,valor_inicial=c(0,0),num_iteraciones=1000000,nu=0.01,umbral_parada=NaN,umbral_dif=NaN){
  i = 0
  valor_anterior = valor_inicial # Para calcular la diferencia de valores (si es muy pequeña el GD para)

  while( i < num_iteraciones & (f(valor_inicial[1],valor_inicial[2]) > umbral_parada | is.nan(umbral_parada)) &
    |abs(f(valor_anterior[1],valor_anterior[2]) - f(valor_inicial[1],valor_inicial[2]) > umbral_dif | i == 0 | is.nan(umbral_dif)) ){

    valor_anterior[1] = valor_inicial[1]
    valor_anterior[2] = valor_inicial[2]

    valor_inicial[1] = valor_inicial[1] - nu*dfu(valor_inicial[1],valor_inicial[2])
    valor_inicial[2] = valor_inicial[2] - nu*dfv(valor_anterior[1],valor_inicial[2]) # Usa el valor u recién actualizado

    i = i+1

    if( !is.nan(umbral_parada) & f(valor_inicial[1],valor_inicial[2]) < umbral_parada) print("Alcanzado umbral de parada")
    if( !is.nan(umbral_dif) & (f(valor_anterior[1],valor_anterior[2]) - f(valor_inicial[1],valor_inicial[2]) < umbral_dif)) print("Alcanzado umbral de diferencia")
  }

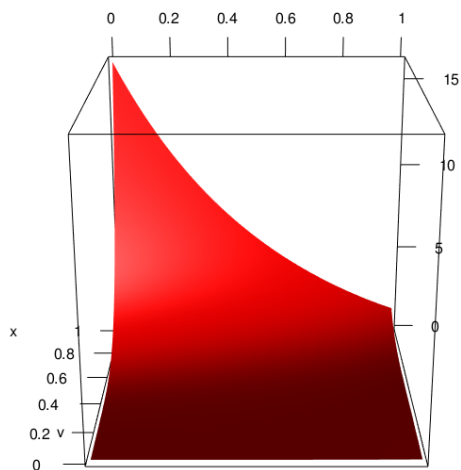
  cat("\nGD ha acabado. Iteraciones completadas:", i)
  cat("\nCoordenadas del minimo: ", valor_inicial[1], valor_inicial[2] )
  cat("\nValor minimo de la funcion: ", f(valor_inicial[1],valor_inicial[2]))
  cat("\n")

  f(valor_inicial[1],valor_inicial[2])
}
```

2. ENCONTRAR EL MÍNIMO DE LA FUNCIÓN E(U,V)

En este apartado usaremos el Gradiente Descendente para encontrar un mínimo a la siguiente función.

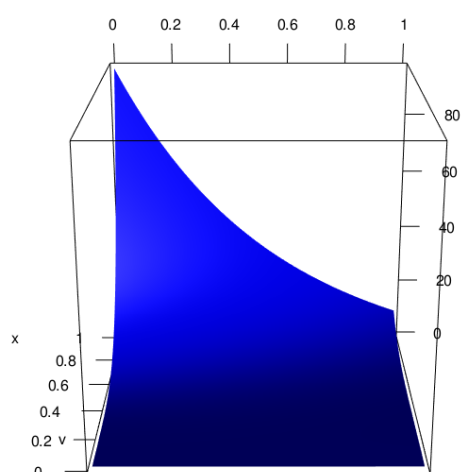
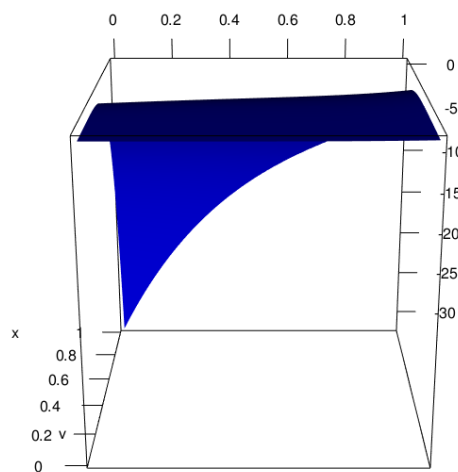
$$E(u, v) = (u^3 e^{(v-2)} - 4v^3 e^{-u})^2$$



Para usar el Gradiente Descendente necesitamos calcular las derivadas de la función, respecto a u y respecto a v:

$$\frac{\partial E(u, v)}{\partial u} = 2 \cdot (u^3 e^{(v-2)} - 4v^3 e^{-u}) \cdot (4v^3 e^{-u} + 3u^2 e^{(v-2)})$$

$$\frac{\partial E(u, v)}{\partial v} = 2 \cdot (u^3 e^{(v-2)} - 12v^2 e^{-u}) \cdot (u^3 e^{(v-2)} - 4v^3 e^{-u})$$



Comenzaremos desde el punto $E(1,1)$, con un número de iteraciones de 30000, un learning rate de 0.05, un umbral de diferencia (diferencia de valor $E(u,v)$ en dos iteraciones continuas) de 10^{-14} y un umbral de parada de 10^{-14}

Vamos a ejecutar el algoritmo:

```
valor_inicial = c(1,1)
umbral_parada = 10^-14
umbral_dif = 10^-14
num_iteraciones = 30000
nu = 0.05

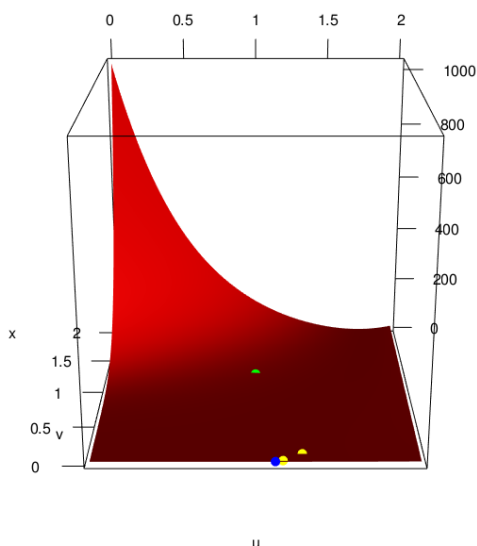
minimo_funcion1 = GD(funcion1,funcion1du,funcion1dv,valor_inicial,num_iteraciones,nu,umbral_parada,umbral_dif)
```

El resultado es el siguiente:

```
Console Terminal x
~/Desktop/Uni/AA/Practicas/P1/ ↗
[1] "Alcanzado umbral de parada"

GD ha acabado. Iteraciones completadas: 38
Coordenadas del mínimo:  1.119544 0.6539881
Valor mínimo de la funcion: 8.795204e-15
> |
```

Vemos que el algoritmo ha tardado 38 iteraciones en llegar al umbral de parada, en las coordenadas 1.119544 y 0.6539881.

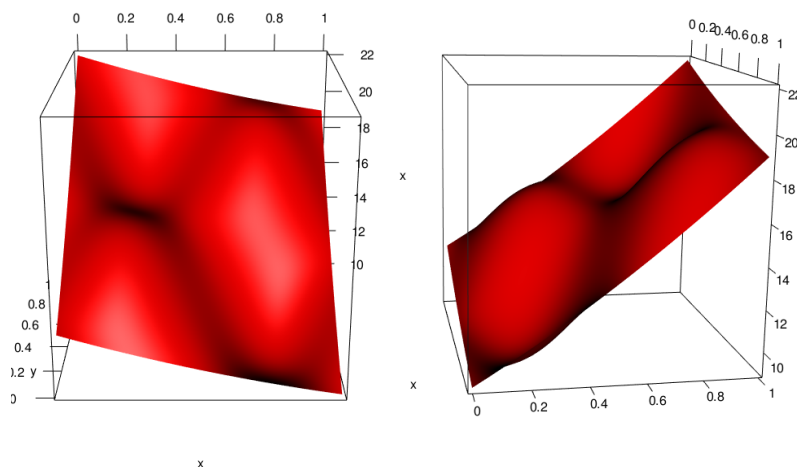


El punto verde representa el punto de inicio. Los puntos amarillos son los mínimos alcanzados en la iteración 1 y 5, mientras que el punto azul representa el mínimo final que el algoritmo ha alcanzado. Como vemos, en la primera iteración ha aproximado muy bien el mínimo óptimo, pero conforme pasan las iteraciones es más difícil llegar.

3. ENCONTRAR EL MÍNIMO DE LA FUNCIÓN F(X,Y)

En este apartado buscaremos con Gradiente Descendiente el mínimo de la siguiente función:

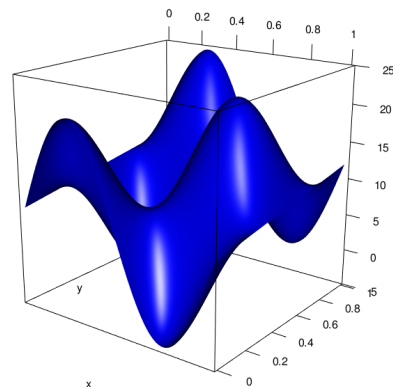
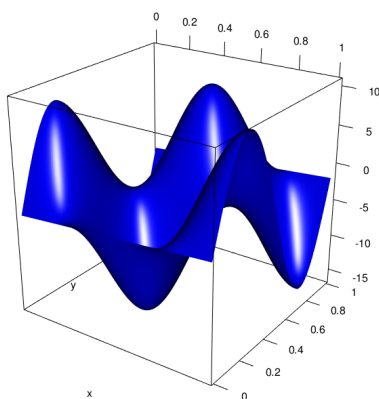
$$f(x, y) = 2\sin(2\pi y)\sin(2\pi x) + (x - 2)^2 + 2(y + 2)^2$$



Para eso vamos a necesitar las derivadas respecto a x y a y:

$$\frac{\partial f(x,y)}{\partial x} = 4\pi\sin(2\pi y)\cos(2\pi x) + 2(x - 2)$$

$$\frac{\partial f(x,y)}{\partial y} = 4\pi\sin(2\pi x)\cos(2\pi y) + 4(y + 2)$$



Vamos a ejecutar el Gradiente Descendiente y sacar el mínimo. Empezaremos por el punto (1,1), usaremos un learning rate de 0.01. Empezaremos con un máximo de 1 iteración, e iremos subiendo hasta llegar a 50. Vamos a repetir el proceso, pero con un learning rate de 0.1

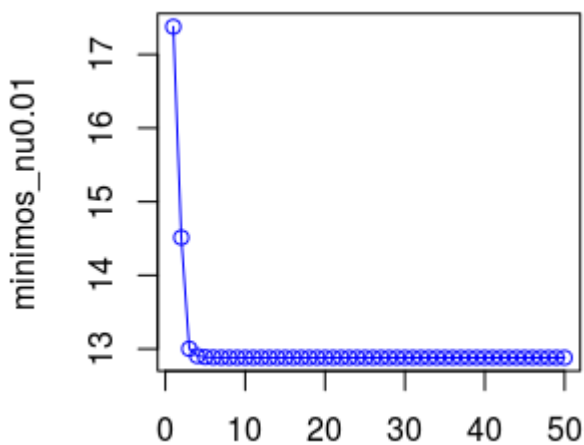
```
punto_inicial_funcion2 = c(1,1)
nu_funcion2 = 0.01
nu_funcion2_0.1 = 0.1
num_iteraciones_funcion2 = 50

minimos_nu0.01 = GD(funcion2,funcion2dx,funcion2dy,punto_inicial_funcion2,1,nu_funcion2)
for(i in 2:50){
  minimos_nu0.01 = c(minimos_nu0.01,GD(funcion2,funcion2dx,funcion2dy,punto_inicial_funcion2,i,nu_funcion2))
}

minimos_nu0.1 = GD(funcion2,funcion2dx,funcion2dy,punto_inicial_funcion2,1,nu_funcion2_0.1)
for(i in 2:50){
  minimos_nu0.1 = c(minimos_nu0.1,GD(funcion2,funcion2dx,funcion2dy,punto_inicial_funcion2,i,nu_funcion2_0.1))
}
```

Con el learning rate de 0.01:

```
Console Terminal x
~/
> minimos_nu0.01
[1] 17.37761 14.51384 13.00319 12.90535 12.88908 12.88442 12.88275 12.88208 12.88180 12.88167 12.88161 12.88159 12.88157
[14] 12.88157 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156
[27] 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156
[40] 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156 12.88156
> |
```

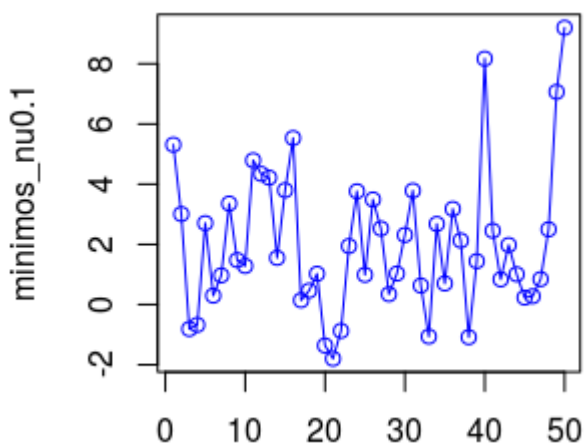


El eje horizontal corresponde al número de iteraciones, mientras que el eje vertical representa el mínimo alcanzado.

Como vemos, hemos encontrado un mínimo local, 12.88156, y debido al learning rate tan pequeño no ha conseguido moverse de ahí.

Con el learning rate de 0.1:

```
Console Terminal x
~/
> minimos_nu0.1
[1] 5.3109830 3.0063434 -0.8126752 -0.6769429 2.7068502 0.2961967 0.9687030 3.3551159 1.4776309 1.2804029
[11] 4.7917227 4.3422213 4.2166028 1.5507864 3.7991849 5.5318754 0.1420142 0.4617775 1.0224323 -1.3619714
[21] -1.7996953 -0.8746918 1.9431593 3.7635422 0.9864017 3.4900245 2.5246248 0.3421628 1.0289648 2.3108424
[31] 3.7853180 0.6303245 -1.0636412 2.6790459 0.6995782 3.1780020 2.1270635 -1.0855535 1.4401239 8.1690039
[41] 2.4530952 0.8354779 1.9775468 1.0046579 0.2311901 0.2750028 0.8433093 2.4983448 7.0661874 9.1973921
> |
```



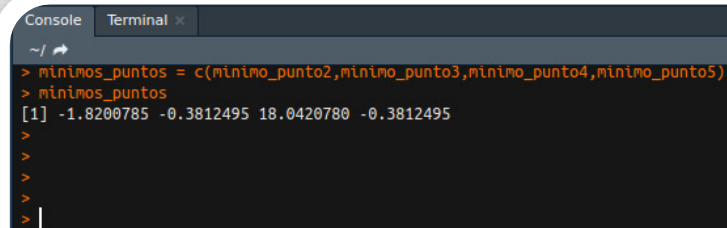
En el segundo experimento ha ocurrido todo lo contrario que en el primero, el mínimo no ha parado de subir y bajar, y no ha conseguido acabar en un mínimo aceptable. Esto es por que la función es particularmente irregular, por lo que el algoritmo de Gradiente Descendiente no funciona demasiado bien.

Vamos ahora a volver a buscar el mínimo, pero empezando desde un punto inicial distinto, para ver las diferencias.

Empezaremos desde el punto (2.1,-2.1), (3,-3), (1.5,1.5), y (1,-1)

```
minimo_punto2 = (GD(funcion2,funcion2dx,funcion2dy,c(2.1,-2.1),num_iteraciones_funcion2,nu_funcion2))
minimo_punto3 = (GD(funcion2,funcion2dx,funcion2dy,c(3,-3),num_iteraciones_funcion2,nu_funcion2))
minimo_punto4 = (GD(funcion2,funcion2dx,funcion2dy,c(1.5,1.5),num_iteraciones_funcion2,nu_funcion2))
minimo_punto5 = (GD(funcion2,funcion2dx,funcion2dy,c(1,-1),num_iteraciones_funcion2,nu_funcion2))

#Puntos minimos desde distintos inicios
minimos_puntos = c(minimo_punto2,minimo_punto3,minimo_punto4,minimo_punto5)
```



```
Console Terminal x
~/
> minimos_puntos = c(minimo_punto2,minimo_punto3,minimo_punto4,minimo_punto5)
> minimos_puntos
[1] -1.8200785 -0.3812495 18.0420780 -0.3812495
>
>
>
>
> |
```

Los puntos de inicio que mayor resultado han dado han sido el (3,-3) y (1,-1), alcanzando un mínimo de -0.3812495.

En conclusion, con el Gradiente Descendiente no siempre se puede encontrar el mínimo global, depende mucho del tipo de función, y de la forma que tenga. Funciona muy bien en funciones sin muchos altibajos, como la primera, pero mal en funciones irregulares como la segunda. En la segunda función en particular, al tener senos y cosenos, su derivada cambia sustancialmente, por lo que el Gradiente Descendiente, cuyo funcionamiento depende de la derivada, no funciona correctamente.

EJERCICIO SOBRE REGRESIÓN LINEAL

1. ESTIMAR UN MODELO DE REGRESIÓN LINEAL A PARTIR DE LOS DATOS PROPORCIONADOS USANDO LA PSEUDOINVERSA Y EL SGD, PARA OBTENER UN CLASIFICADOR DE NÚMEROS 1 Y 5

En este apartado vamos a estimar un modelo de Regresión Lineal a partir de la intensidad promedio y la simetría de los datos proporcionados. Se hará de dos formas, con la pseudoinversa y con el Gradiente Descendiente Estocástico (SGD). Después, se comprobará la bondad de la función estimada con datos de test.

Lo primero será leer los dígitos y las etiquetas tanto del set de entrenamiento como de test

```
# Leer datos y etiquetas
digit.train <- read.table("zip.train", quote="\"", comment.char="", stringsAsFactors=FALSE) # 1. lectura zip del train
digitos15.train = digit.train[digit.train$V1==1 | digit.train$V1==5,] # Se obtienen los números 1 y 5
etiquetasTrain = digitos15.train[,1] # vector de etiquetas del train
netiquetasTrain = nrow(digitos15.train) # numero de muestras del train

digit.test <- read.table("zip.test", quote="\"", comment.char="", stringsAsFactors=TRUE) # 1. lectura zip del test
digitos15.test = digit.test[digit.test$V1==1 | digit.test$V1==5,] # Se obtienen los números 1 y 5
etiquetasTest = digitos15.test[,1] # vector de etiquetas del test
netiquetasTest = nrow(digitos15.test) # numero de muestras del test

setwd("../..")
```

Después, obtendremos los grises, y a partir de éstos, la simetría y la intensidad de cada dígito.

```
# Obtener grises
grisesTrain = array(unlist(subset(digitos15.train,select=-V1)),c(netiquetasTrain,16,16))
grisesTest = array(unlist(subset(digitos15.test,select=-V1)),c(netiquetasTest,16,16))

# Obtener intensidad
intensidadTrain = apply( grisesTrain, MARGIN = 1, FUN = mean )
intensidadTest = apply( grisesTest, MARGIN = 1, FUN = mean )

# Obtener simetria
simetriaTrain = apply( grisesTrain, MARGIN = 1, FUN = fsimetria )
simetriaTest = apply( grisesTest, MARGIN = 1, FUN = fsimetria )
```

Vamos a reetiquetar al número 5 como -1, y vamos a crear una matriz con todos los datos.

```
# Reetiquetar etiquetas
etiquetasTrain[etiquetasTrain==5]=-1
etiquetasTest[etiquetasTest==5]=-1

# Crear matriz de datos
datosTr = as.matrix(cbind(intensidadTrain,simetriaTrain,1))
datosTest = as.matrix(cbind(intensidadTest,simetriaTest,1))
```


Ya tenemos todo listo. Ahora vamos a calcular la función que necesitamos. Lo que buscamos es una función que clasifique los datos en unos y cincos. En este caso, nuestra función debe seguir este modelo:

$$h(x) = \text{sign}(w_0 + w_1x_1 + w_2x_2 + \dots + w_kx_k) = \text{sign}(w^T x), \quad h(x) \in \{-1, 1\}$$

Donde x es el vector de datos de entrada, y w el peso que queremos calcular.

Si $h(x)=+1$, quiere decir que los datos de entrada corresponden (o deberían corresponder) al número 1.

Si $h(x)=-1$, los datos deberían corresponder a un 5.

PSEUDOINVERSA

Primero calcularemos estos pesos mediante la pseudoinversa.

Algebraicamente, sabemos que el error de entrada es el siguiente:

$$E_{in}(w) = \frac{1}{N} \|Xw - y\|^2$$

Nosotros buscamos minimizar esta función (minimizar el error), por lo que derivaremos e igualaremos a 0:

$$\nabla E_{in}(w) = \frac{2}{N} X^T (Xw - y) = 0$$

Calculando, obtenemos que el w mínimo es el siguiente:

$$w = X^\dagger y \text{ donde } X^\dagger = (X^T X)^{-1} X^T \quad \text{Y la pseudoinversa es } X^\dagger$$

En código, la función que calcula los pesos con ayuda de la pseudoinversa y la función que calcula la pseudoinversa están implementadas así

```
#Función de pseudoinversa
pseudoinversa = function(fdatos){
  solve(t(fdatos) %*% fdatos ) %*% t(fdatos) # (Xt*X)^-1 * Xt
}

#Función de regresión lineal usando la pseudoinversa
RegresionLinealPseudoinversa <- function(fdatos, fetiquetas){
  pesos <- pseudoinversa(fdatos) %*% fetiquetas
}
```

Por lo tanto, vamos a llamar a la función para obtener los pesos de $h(x)$, partiendo de $w = (1,1,1)$:

```
w = c(1,1,1)
w = RegresionLinealPseudoinversa(datosTr, etiquetasTrain)
#w = SGD(EinLR,dEinLR,datosTr,etiquetasTrain,w,1000,0.1,0.)
```

Los pesos resultantes son los siguientes:

```
Console Terminal x
~/Desktop/Uni/AA/Practicas/P1/ ➔
> w = RegresionLinealPseudoinversa(datosTr, etiquetasTrain)
> w
      [,1]
intensidadTrain 0.73182914
simetriaTrain   0.01402067
               1.71412717
> |
```

Es decir, nuestra función clasificadora es la siguiente:

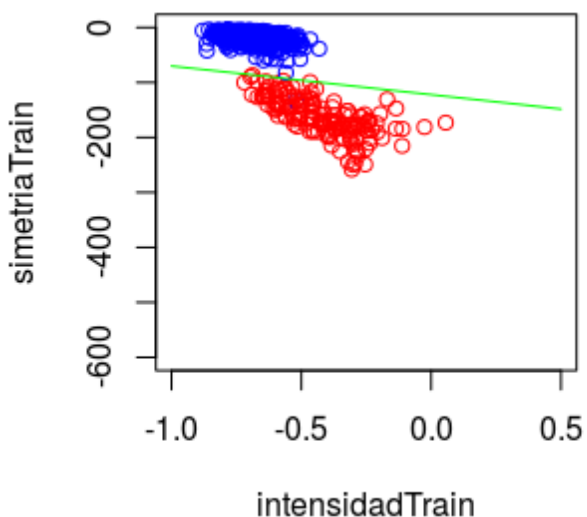
$$h(x) = \text{sign}(0.73182914x_0 + 0.01402067x_1 + 1.71412717)$$

Vamos a ver gráficamente como funciona:

```
# Dibujar función
ayb = pasoARecta(w)

x = c(-1,0.5)
y = c(-1*ayb[1] + ayb[2], 0.5*ayb[1] + ayb[2] )

plot(x=intensidadTrain,y=simetriaTrain,col=etiquetasTrain+3,xlim=c(-1,0.5),ylim=c(-600,0)) # TRAIN
par(new=TRUE)
lines(x,y,col='green')
```



En azul, los números 1. En rojo, los números 5.

Como vemos, la función clasifica casi a la perfección.

Vamos a calcular el error que hemos tenido, E_{in} . Para ello obtenemos las etiquetas con la función que hemos calculado, y contamos las etiquetas que difieran con la etiqueta real. El resultado lo dividimos entre el número total de etiquetas:

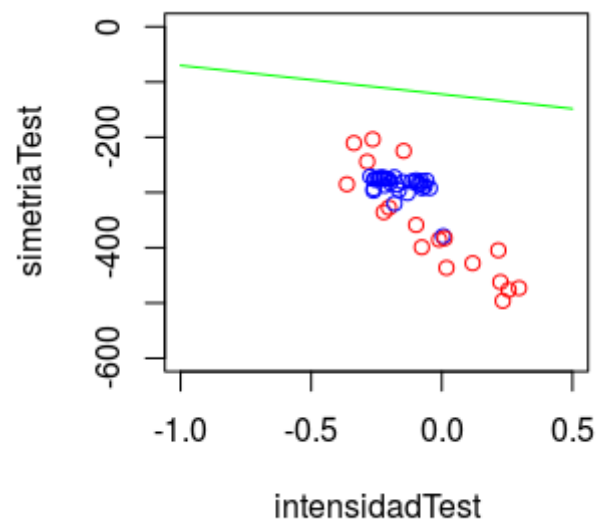
```
# Obtener resultados predichos por h(x)
hx = sign(datosTr %*% w)      # TRAIN
```

```
#Función que devuelve el error
error <- function(etiquetas,predicciones){
  length(etiquetas[etiquetas != as.vector(predicciones)])/length(etiquetas)
}
```

```
Console Terminal x
~/Desktop/Uni/AA/Practicas/P1/ ↗
> Ein = error(etiquetasTrain,hx)
> Ein
[1] 0.001669449
>
>
>
>
> |
```

Como ya intuíamos con el gráfico, el error es muy pequeño (poco mas de un 0.0016%)

Vamos a ver ahora cómo trabaja la función con los datos de test:



No tiene buena pinta. Vamos a calcular el error que hemos tenido, Eout

```
# Obtener resultados predichos por h(x)|
hxout = sign(datosTest %*% w) # TEST
```

```
Console Terminal x
~/Desktop/Uni/AA/Practicas/P1/ ➔
> Eout = error(etiquetasTest,hxout)
> Eout
[1] 0.6326531
>
>
>
>
> |
```

Como vemos, en el test no se nos ha dado tan bien. El error es de un 63%.

GRADIENTE DESCENDIENTE ESTOCÁSTICO (SGD)

Ya hemos obtenido los pesos con la pseudoinversa, ahora vamos a obtenerlo con el Gradiente Descendiente Estocástico. Este algoritmo es como el Gradiente Descendiente, solo que en vez de usar todos los datos para calibrar, usamos un pequeño porcentaje, obteniendo el resultado mucho más rápido y con poca diferencia en cuanto a error.

Matemáticamente se expresa así:

$$w_j = w_j - \eta \cdot \frac{\partial f(w)}{\partial w_j}$$

donde η es el learning rate, f la función a usar y w_j es el peso inicial a actualizar. Para cada j de w_0, w_1, \dots, w_k , el peso se actualiza simultáneamente.

La función a usar será una función obtenida a partir de Regresión Logística, y cuya derivada es la siguiente:

$$\nabla_w E_{in}(w) = \frac{1}{M} \sum_{i=0}^M -y_i x_i \frac{e^{(-y_i w^T x_i)}}{1 + e^{(-y_i w^T x_i)}}$$

En la cual 'M' es el número de datos escogidos al azar, 'y' es el vector de etiquetas, 'w' es el vector de pesos y 'x' es la matriz de datos.

El Gradiente Descendiente Estocástico está implementado de la siguiente manera.

```
## - Función del algoritmo SGD
SGD <- function(f,df,x,y,w_inicial=c(1,1,1),num_iteraciones=1000,porcentaje=0.2,nu=0.01){

  for( i in 1:num_iteraciones ){
    w_inicial = w_inicial - nu*df(w_inicial,x,y,porcentaje)

    cat(w_inicial)
  }

  cat("\nSGD ha acabado. Pesos: ", w_inicial)
  cat("\n")

  w_inicial
}
```

La derivada de la función que usaremos está implementada así:

```
## - Función derivada de Ein para Logistic Regression
dEinLR <- function(w,x,y,porcentaje){
  suma = c(1,1,1)
  N = length(y)*porcentaje
  for( i in 1:N ){
    irand = sample(length(y),1)

    a = y[irand]*x[irand,]
    b = exp(-y[irand]*t(w)%*%x[irand,])
    c = 1+exp(-y[irand]*t(w)%*%x[irand,])

    suma = suma - as.vector(a)*(as.vector(b)/as.vector(c))

  }

  suma = suma/N

  suma
}
```

Vamos a obtener los pesos. El algoritmo hará 1000 iteraciones, en cada una calibrará los pesos con un 20% aleatorio de los puntos totales, y usaremos un learning rate de 0.1.

```
w = c(1,1,1)

#w = RegresionLinealPseudoinversa(datosTr, etiquetasTrain)
w = SGD(EinLR,dEinLR,datosTr,etiquetasTrain,w,1000,0.2,0.1)
```

Los pesos resultantes son los siguientes:

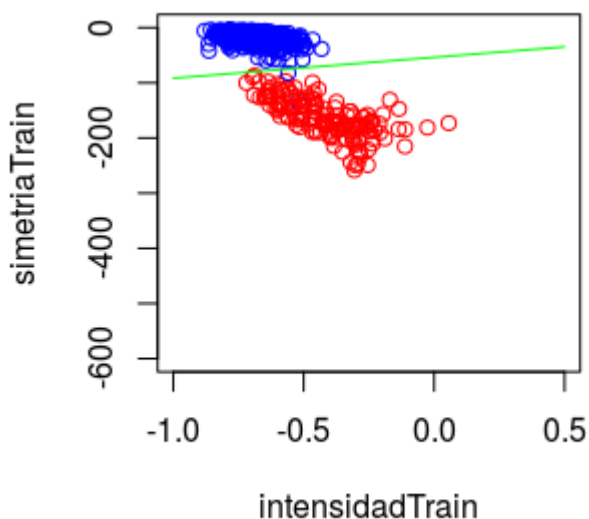
```
Console Terminal
~/Desktop/Uni/AA/Practicas/P1/
> #w = RegresionLinealPseudoinversa(datosTr, etiquetasTrain)
> w = SGD(EinLR,dEinLR,datosTr,etiquetasTrain,w,1000,0.2,0.1)

SGD ha acabado. Pesos: -10.6349 0.2814266 15.0908
>
>
>
> |
```

Es decir, nuestra función clasificadora es la siguiente:

$$h(x) = \text{sign}(-10.6349x_0 + 0.2814266x_1 + 15.0908)$$

Parece muy distinta a la función que obtuvimos con la pseudoinversa. Vamos a ver gráficamente cómo es.



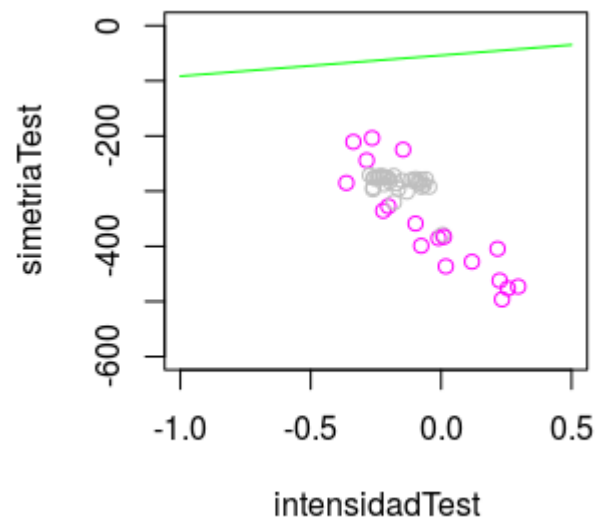
Aunque los pesos sean muy distintos a los obtenidos con la pseudoinversa, vemos que la función que obtenemos es muy parecida, y clasifica aparentemente bien.

Vamos a calcular el error que hemos tenido con el SGD:

```
Console Terminal x
~/Desktop/Uni/AA/Practicas/P1/ ➔
> Eout = error(etiquetasTest,hxout)
> Ein = error(etiquetasTrain,hx)
> Ein
[1] 0.003338898
>
>
>
> |
```

El error es un poquito mayor que el que obtuvimos con la pseudoinversa, es de un 0.0033% aproximadamente.

Vamos a ver como se comporta la función con los datos de test:



Al igual que con la pseudoinversa, el error obtenido es grande. Vamos a medirlo:

```
Console Terminal x
~/Desktop/Uni/AA/Practicas/P1/ ➔
> lines(x,y,col="green")
> Eout = error(etiquetasTest,hxout)
> Eout
[1] 0.6326531
>
>
>
> |
```

Tenemos el mismo error que antes. De un 63% aproximadamente.

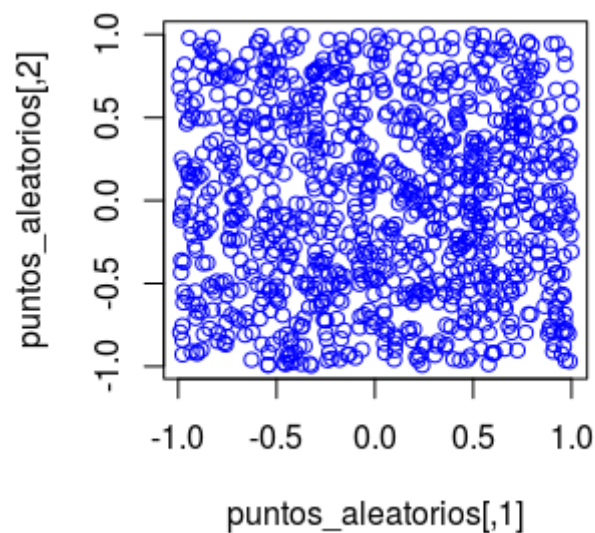
2. ESTIMAR UN MODELO DE REGRESIÓN LINEAL SOBRE PUNTOS ALEATORIOS

En este apartado lo primero que vamos a hacer es generar 1000 puntos aleatorios en un espacio de $[-1,1] \times [-1,1]$.

```
# Creamos 1000 puntos aleatorios
puntos_aleatorios = simula_unif(1000,2,c(-1,1))

#Función para simular puntos en un plano
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
}
```

Gráficamente se ven así:



Vamos a considerar la siguiente función:

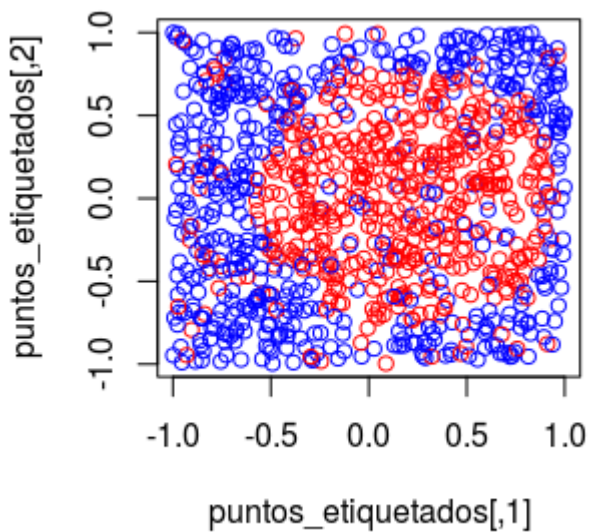
$$f(x_1, x_2) = \text{sign}((x_1 - 0.2)^2 + x_2^2 - 0.6)$$

Esta función se encargará de asignar una etiqueta a cada punto de los que hemos generado.

Vamos a generar un poco de ruido (10%), y para eso usaremos la siguiente función:

```
#Función para añadir ruido
noise <- function(label, p){
  result <- label * sample(c(1, -1), size=length(label), replace=TRUE, prob=c(1 - p, p))
  result
}
```

Gráficamente, los puntos, ya etiquetados y con ruido, son los siguientes:



En azul, los puntos con la etiqueta +1.

En rojo, los puntos con la etiqueta -1.

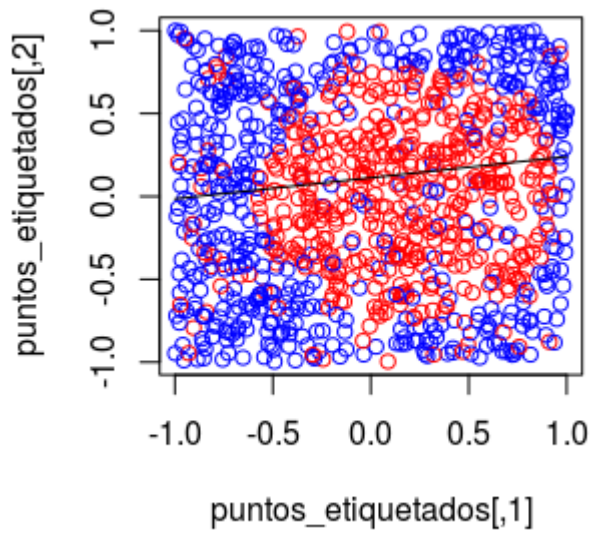
Vamos a ajustar un modelo de regresión lineal, usando el SGD. Para eso calibraremos el vector de características $(1, x_1, x_2)$.

Vamos a usar como características iniciales el vector $(1, 1, 1)$, vamos a realizar 1000 iteraciones, de la cual en cada una se cogerán un 30% de puntos aleatorios para calibrar los pesos. Se ha usado un learning rate de 0.1.

```
Console Terminal x
~/Desktop/Uni/AA/Practicas/P1/ ➔
>
> # Predecimos los pesos con el algoritmo SGD
> w_f3 = SGD(EinLR, dEinLR, puntos_aleatorios, etiquetas_puntos, c(1, 1, 1), 1000, 0.3, 0.1)

SGD ha acabado. Pesos: 0.1033073 -0.8104968 0.09126739
> |
```

Vamos a ver cómo ha quedado la función visualmente:



Evidentemente un ajuste óptimo es imposible, pero el algoritmo ha hecho lo que ha podido.

Vamos a ver el error cometido:

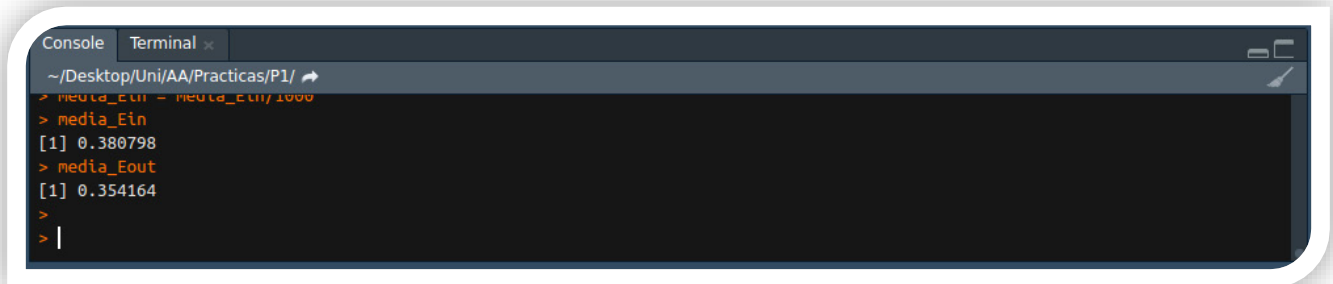
```
Console Terminal x
~/Desktop/Uni/AA/Practicas/P1/ ➔
> Elin = error(etiquetas_puntos,fixf3)
> Eln = error(etiquetas_puntos,hxf3)
> Eln
[1] 0.431
>
>
> |
```

Vemos que hemos obtenido un error del 43.1%

Ahora vamos a repetir este proceso 1000 veces, en el cual en cada iteración el SGD usará 1000 puntos distintos para aprender, y además se someterá a un test de otros 1000 puntos nuevos, distintos también en cada iteración.

Al acabar, obtendremos la media resultante de Ein y Eout.

Esta vez, para hacer más rápido el proceso, el SGD sólo usará un 5% de los puntos totales.



```
~/Desktop/Uni/AA/Practicas/P1/
> media_Ein = media_Ein/1000
> media_Ein
[1] 0.380798
> media_Eout
[1] 0.354164
>
>
> |
```

La media de acierto de Ein es del 38.08%, y la media de acierto de Eout es del 35.41%

En conclusión, para que la regresión lineal sea eficaz y se pueda hacer un clasificador efectivo, los puntos deben seguir un patrón concreto. Con puntos distribuidos al azar, como hemos visto, son preferibles otros métodos.