Trabajo 1. Programación

José Carlos Martínez Velázquez

28 de marzo de 2016

Ejercicios de generación y visualización de datos

1. Construir una función *lista=simula_unif(N,dim,rango)* que calcule una lista de longitud N de vectores de dimensión *dim* conteniendo números aleatorios uniformes en el intervalo del rango.

Para realizar este ejercicio debemos familiarizarnos con la sintaxis de definición de una función: `ejemplo<-function(parametros){...}`. Dado que vamos a trabajar con números aleatorios, es conveniente inicializar la semilla de generación de los mismos mediante un número primo: `set.seed(7)` Una vez hecho, vamos a definir la función que utilizaremos:

```
simula_unif=function(N,dim,rango){
  lapply(seq_len(N),function(i) runif(dim,rango[1],rango[2]));
}
```

Analicemos la línea dentro de la función simula_unif:
`lapply(seq_len(N),function(i) runif(dim,rango[1],rango[2]));`
Básicamente, lapply aplica una función a un vector. Declararemos un vector con el comando `seq_len(N)`, que lo inicializará con N posiciones vacías. La función que se aplica a dicho vector genera un vector de *dim* valores aleatorios en una distribución uniforme entre los valores `rango[1]` y `rango[2]`.

Para llamar a la función que acabamos de crear, ejecutamos la siguiente línea, que nos devolverá una lista con dos vectores, cada uno de ellos con 10 valores aleatorios de una distribución uniforme:

```
simula_unif(2,10,c(0,10))
```

```
## [[1]]
##  [1] 3.7612468 6.8978904 5.3875608 0.2333050 5.8380972 8.1595384 4.4379483
##  [8] 0.6973649 1.8124898 2.8866669
##
## [[2]]
##  [1] 7.460613 2.882946 9.964670 4.558446 9.058052 5.371630 7.474670
##  [8] 3.844618 8.782806 3.720746
```

2. Construir una función *lista=simula_gauss(N,dim,sigma)* que calcule una lista de longitud N de vectores de dimensión *dim* conteniendo números aleatorios gaussianos de media 0 y varianzas dadas por el vector sigma.

Al igual que procedimos en el ejercicio anterior, la función que crearemos inicializará un vector vacío de dimensión N, con la diferencia de que ahora le aplicará la función que genera números aleatorios en una distribución gaussiana.

El código que define la función es el siguiente:

```
simula_gauss<-function(N, dim, sigma){
  lapply(seq_len(N),function(i) rnorm(dim,0,sigma));
}
```

Para llamar a nuestra función, pasar vector de varianzas como parámetro:

```
simula_gauss(3,10,c(1,2,3))
```

Dicha llamada proporciona la siguiente salida:

```
## [[1]]
##  [1] -0.6120989  0.4952069 -3.7389351  1.1365342  0.1794293 -1.0393181
##  [7] -1.4497174  1.3675281  3.2132042  0.7538836
##
## [[2]]
##  [1] -0.4789765 -1.4790937  0.5159174  1.3830644 -1.3812979 -5.6951546
##  [7]  0.3286694 -2.1294419  1.0221230  0.9008776
##
## [[3]]
##  [1] -2.2849962 -1.5607733 -4.6564932 -1.2821664  1.7966716 -3.5164313
##  [7] -1.0011334  1.4126560 -0.3728879 -0.1974702
```

3. Suponer N=50, dim=2 y rango=[-50,+50] en cada dimensión. Dibujar una gráfica de la salida de la función correspondiente.

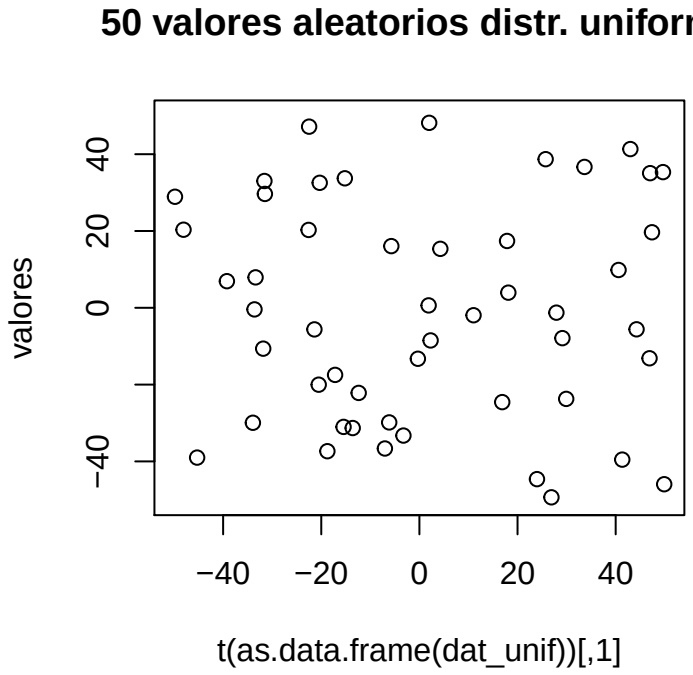
Tenemos que pintar puntos de la forma (x,y). Dada la estructura utilizada, habrá que convertirlo a una tabla o matriz que proporcione los puntos a modo de filas. Yo he utilizado la función `as.data.frame()`. Para saber cómo funciona, es mejor verlo con un ejemplo.

Supongamos la lista ejemplo, que contiene vectores cuyos valores son:

- lista:
  - lista[[1]]: 1,2
  - lista[[2]]: 3,4
  - lista[[3]]: 5,6

Entonces, debemos representar los puntos (1,2), (3,4), (5,6). Si aplicamos la función `as.data.frame(lista)`, entonces lo que tenemos es un *dataframe* que se puede ver análogamente como la siguiente matriz: $\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$. Aún no podemos hacer nada con el resultado, pero no tenemos más que trasponer el *dataframe* resultante con `t(as.data.frame(lista))` y obtendríamos la tabla $\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$, donde cada fila es un punto a representar en la gráfica. Lo único que nos queda por hacer es representar el *dataframe* en una gráfica. Para ello no tenemos más que envolver la anterior llamada con un `plot()` y, opcionalmente cambiar los parámetros. El código que resulta es el siguiente:

```
dat_unif<-simula_unif(50,2,c(-50,50))
plot(t(as.data.frame(dat_unif)),ylab="valores",
     main="50 valores aleatorios distr. uniforme",
     xlim=c(-50,50),ylim=c(-50,50))
```



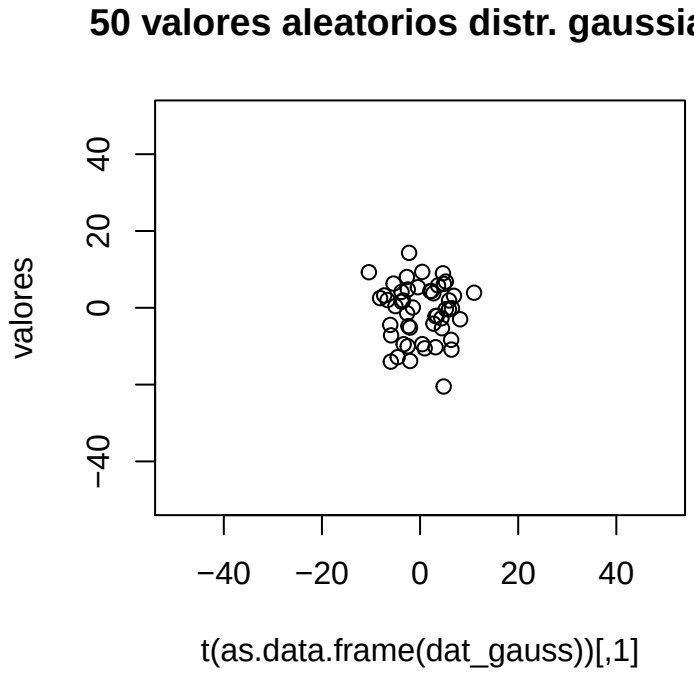

```
#plot(do.call(rbind,dat_unif))

#plot(NA, ylab="valores", main="50 valores aleatorios distr. uniforme",
#     xlim=c(-50,50),ylim=c(-50,50))
#for(i in 1:50){points(dat_unif[[i]][1],dat_unif[[i]][2])}
```

Los comentarios son un modo de hacer exactamente lo mismo, pero con otros comandos.

4. Suponer N=50, dim=2 y sigma=[5,7] dibujar una gráfica de la salida de la función correspondiente.

De la misma manera que en el ejercicio anterior, reinterpretamos la lista y la representamos:

```
dat_gauss<-simula_gauss(50,2,c(5,7))
plot(t(as.data.frame(dat_gauss)),ylab="valores",
     main="50 valores aleatorios distr. gaussiana",
     xlim=c(-50,50),ylim=c(-50,50))
```



5. Construir la función v=simula_recta(intervalo) que calcula los parámetros v=(a,b) de una recta aleatoria, $y = ax + b$, que corte al cuadrado $[-50, 50]$x$[-50, 50]$.

La función que realiza esta tarea es la siguiente:

```
simula_recta<-function(intervalo){
  x=runif(2,intervalo[1],intervalo[2])
  y=runif(2,intervalo[1],intervalo[2])
  a=(y[2]-y[1])/(x[2]-x[1])#Calculo la pendiente
  b=(y[1])-((a)*(x[1]))#Aplico y=ax+b a un punto cualquiera
  c(x,y,a,b)
}
```

Pasemos a explicarla. El objetivo es calcular los coeficientes a (pendiente de la recta) y b (offset) de una ecuación del tipo $y = ax + b$ que pasa por dos puntos conocidos, los generaremos aleatoriamente. En el vector x generaremos dos números enteros aleatorios pertenecientes al intervalo (que lógicamente delimita perfectamente un cuadrado), que corresponderán con las coordenadas x de los dos puntos que delimitarán nuestra recta aleatoria. Del mismo modo, en el vector y guardaremos las coordenadas y de ambos puntos. Así, los puntos que definen la recta aleatoria serán $p1 = (x[1], y[1])$ y $p2 = (x[2], y[2])$.

Dado que la pendiente se calcula mediante el incremento de y entre el incremento de x ($a = \frac{\triangle y}{\triangle x}$) y el offset b se calcula como los valores x e y de un punto conocido aplicados a la ecuación de la recta, podemos devolver ya ambos valores. Adicionalmente devolveré las coordenadas x e y de los puntos, de modo que, si hacemos una llamada `vars<-simula_recta(...)`, la función va a devolver los siguientes valores:

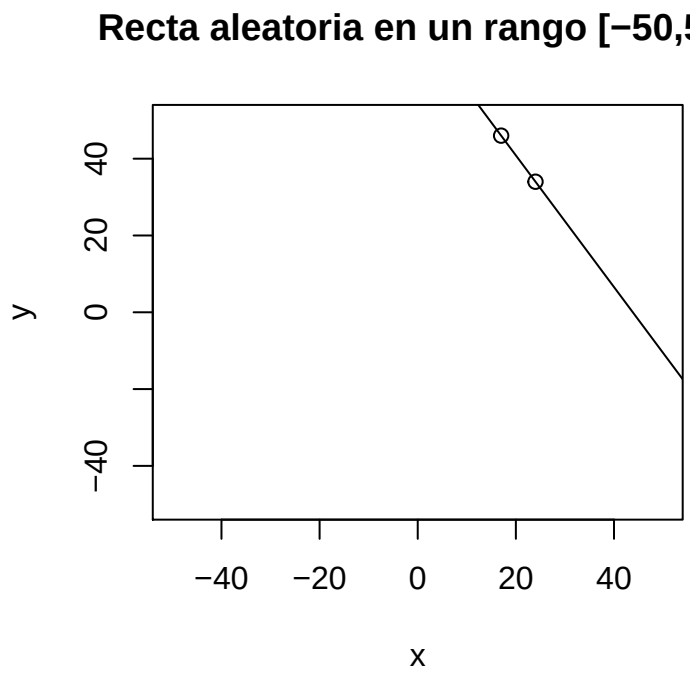
- vars[1] y vars[2] las coordenadas x de los dos puntos
- vars[3] y vars[4] las coordenadas y de los dos puntos
- vars[5] pendiente de la recta (a)
- vars[6] offset (b)

Entonces, una recta aquí será del tipo $y = vars[5] * x + vars[6]$

Para probar la función ejecutamos el siguiente código:

```
vars<-simula_recta(c(-50,50));
#Dibujamos los puntos
plot(c(vars[1],vars[2]),c(vars[3],vars[4]),xlab="x",ylab="y",
main="Recta aleatoria en un rango [-50,50]",ylim=c(-50,50), xlim=c(-50,50))
abline(vars[6],vars[5])#Añadimos la recta a la gráfica.
```

Cabe destacar que la función abline de R tiene los coeficientes cambiados, es decir, el código `abline(a,b)` añadirá una recta de offset a y pendiente b, justo lo contrario de lo que queremos. Entonces, si queremos una recta con pendiente a y offset b, deberemos llamar a la función `abline(b,a)`. La prueba realizada nos proporciona la siguiente salida:



6. Generar una muestra 2D de puntos usando *simula_unif()* y etiquetar la muestra usando el signo de la función $f(x,y) = y - ax - b$ de cada punto a una recta simulada con *simula_recta()*. Mostrar una gráfica con el resultado de la muestra etiquetada junto con la recta usada para ello.

Lo primero que haremos será definir el rango para pasarlo a las funciones *simula_unif* y *simula_recta*. Posteriormente obtendremos la muestra 2d en la variable homónima y obtendremos los coeficientes de la recta en la variable vars. Posteriormente, en la variable signos, vamos a guardar el signo de la función del enunciado para cada par de valores x e y en cada índice de los dos vectores de muestra2d.

```
rango=c(-50,50)
muestra2d<-simula_unif(2,50,c(rango[1],rango[2]))
vars<-simula_recta(c(rango[1],rango[2]))
signos6 <- sign(muestra2d[[2]] - vars[5]*muestra2d[[1]] - vars[6])
```

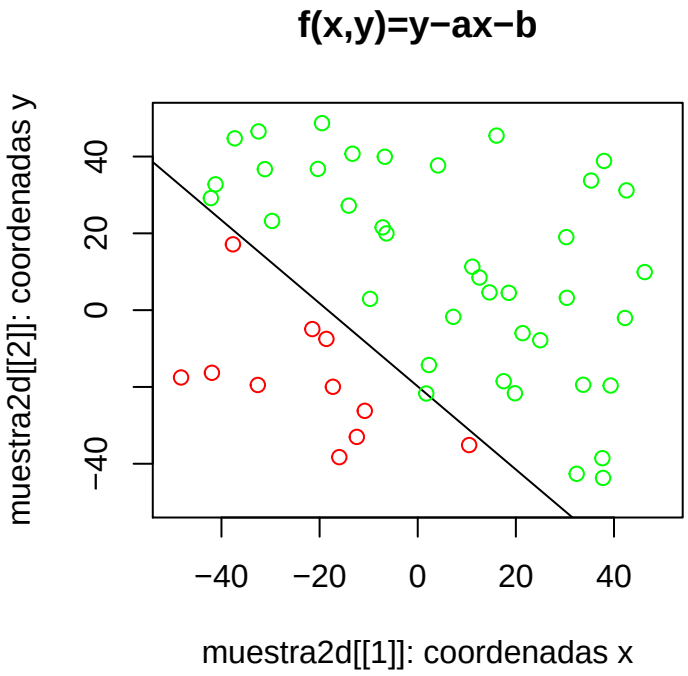
Ahora que ya tenemos todo lo necesario, vamos a dibujar la gráfica. Desgranemos el código que lo hará.

```
plot(
  NA,xlim=c(rango[1],rango[2]),ylim=c(rango[1],rango[2]),
```

```
  xlab='muestra2d[[1]]: coordenadas x',ylab='muestra2d[[2]]: coordenadas y',
  main='f(x,y)=y-ax-b'
)
abline(vars[6],vars[5]);
with(setNames(muestra2d,c('x','y')),{
  points(x[signos6>=0],y[signos6>=0],col='green');
  points(x[signos6<0],y[signos6<0],col='red');
});
```

La primera línea inicializa una gráfica vacía con los límites dados por el rango. La segunda línea va a añadir la recta que divide los puntos en dos partes. A partir de la tercera línea vamos a configurar algunos parámetros de lo que aparece en la gráfica. La función `with(...)` nos ayudará a hacerlo. Cambiamos el nombre a cada vector y les llamamos x e y respectivamente. Posteriormente, los puntos en el índice i cuyo valor de la función es positivo (están por encima de la recta) se pintarán en verde y del mismo modo, los puntos cuyo signo es negativo (están por debajo de la recta) se pintarán en rojo.

Al ejecutar los dos bloques anteriores, obtenemos el siguiente resultado:



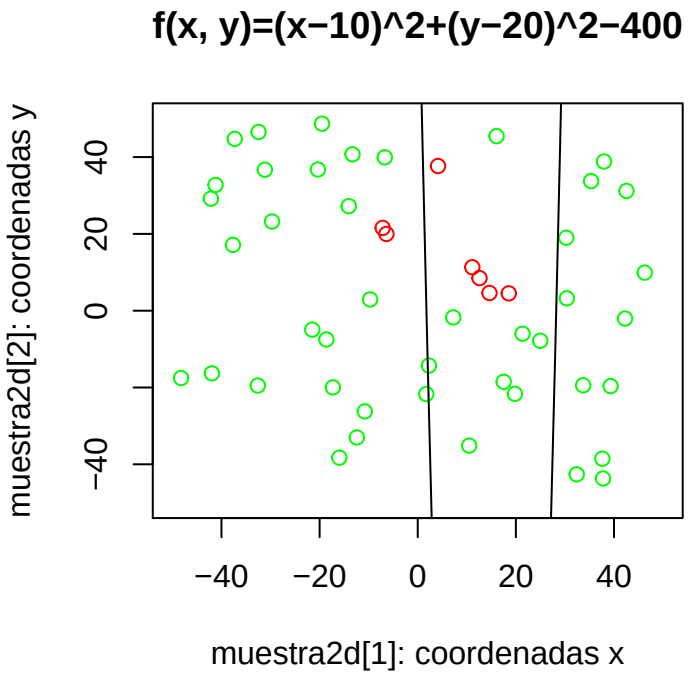
7. Usar la muestra generada en el apartado anterior y etiquetarla con +1,-1 usando el signo de cada una de las siguientes funciones:

Ejecutaremos el mismo código para cada una de las funciones, pero con la salvedad de que evaluamos, evidentemente, el signo de diferentes funciones.

- $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$

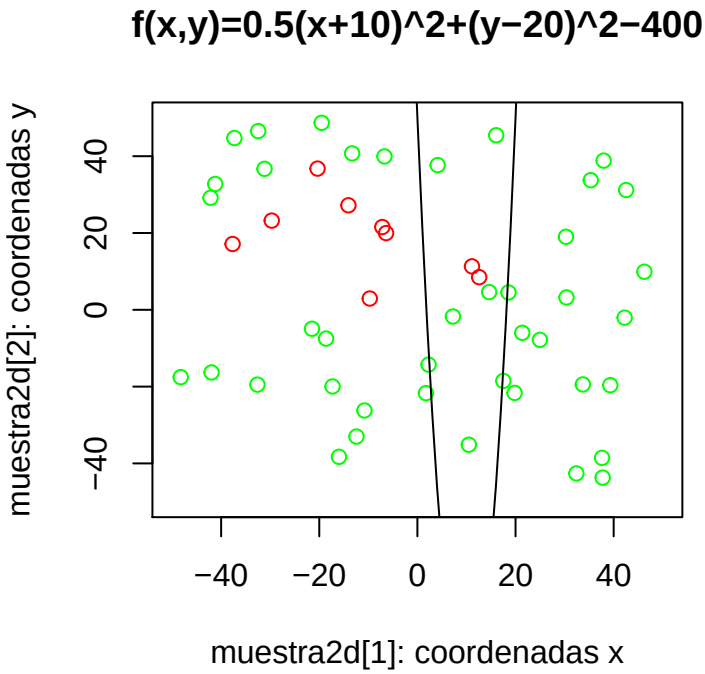
```
#Cambio la funcion de la cual quiero extraer el signo
signos71 <- sign( (muestra2d[[1]]-10)^2 + (muestra2d[[2]]-20)^2 -400 );
#Añado la gráfica vacía
plot(NA,
     xlim=c(rango[1],rango[2]),
```

```
     ylim=c(rango[1],rango[2]),
     xlab='muestra2d[1]: coordenadas x',
     ylab='muestra2d[2]: coordenadas y',
     main='f(x, y)=(x-10)^2+(y-20)^2-400'
)
#Cambio los colores de los puntos en función de su signo: verde (+) o rojo (-)
with(setNames(muestra2d,c('x','y')),{
  points(x[signos71>=0],y[signos71>=0],col='green');
  points(x[signos71<0],y[signos71<0],col='red');
});
lines(rango[1]:rango[2],(rango[1]:rango[2]-10)^2 + (rango[1]:rango[2]-20)^2 -400)
```



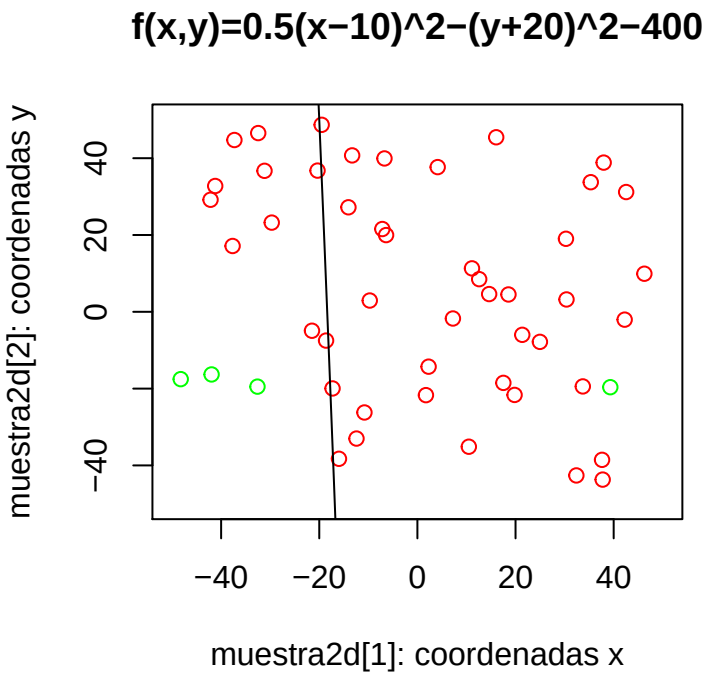
- $f(x, y) = 0.5(x + 10)^2 + (y - 20)^2 - 400$

```
signos72 <- sign( 0.5*(muestra2d[[1]]+10)^2+(muestra2d[[2]]-20)^2-400 );
plot(NA,
     xlim=c(rango[1],rango[2]),
     ylim=c(rango[1],rango[2]),
     xlab='muestra2d[1]: coordenadas x',
     ylab='muestra2d[2]: coordenadas y',
     main='f(x,y)=0.5(x+10)^2+(y-20)^2-400'
);
with(setNames(muestra2d,c('x','y')),{
  points(x[signos72>=0],y[signos72>=0],col='green');
  points(x[signos72<0],y[signos72<0],col='red');
});
lines(rango[1]:rango[2],0.5*(rango[1]:rango[2]+10)^2+(rango[1]:rango[2]-20)^2-400)
```



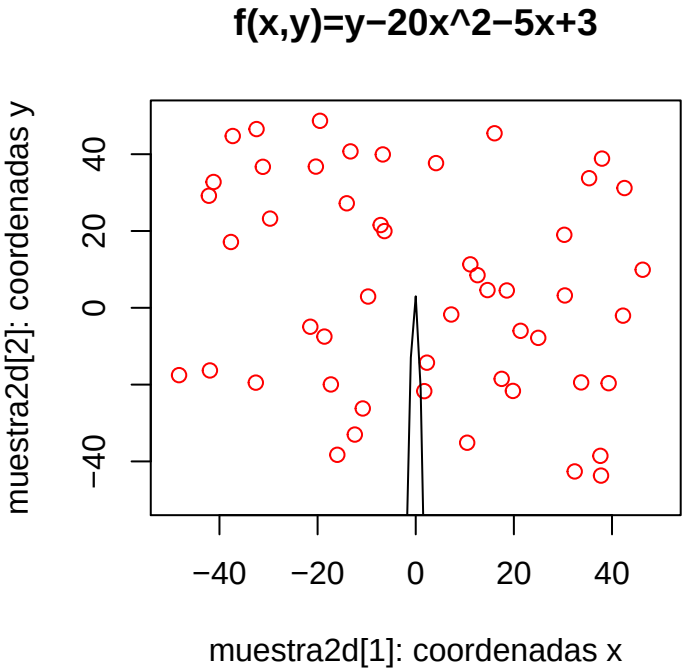
- $f(x, y) = 0.5(x - 10)^2 - (y + 20)^2 - 400$

```
signos73 <- sign( 0.5*(muestra2d[[1]]-10)^2-(muestra2d[[2]]+20)^2-400 );
plot(NA,
     xlim=c(rango[1],rango[2]),
     ylim=c(rango[1],rango[2]),
     xlab='muestra2d[1]: coordenadas x',
     ylab='muestra2d[2]: coordenadas y',
     main='f(x,y)=0.5(x-10)^2-(y+20)^2-400'
);

with(setNames(muestra2d,c('x','y')),{
  points(x[signos73>=0],y[signos73>=0],col='green');
  points(x[signos73<0],y[signos73<0],col='red');
});
lines(rango[1]:rango[2],0.5*(rango[1]:rango[2]-10)^2-(rango[1]:rango[2]+20)^2-400)
```



- $f(x, y) = y - 20x^2 - 5x + 3$

```
signos74 <- sign( muestra2d[[2]]-20*(muestra2d[[1]])^2-5*(muestra2d[[1]])+3 );
plot(NA,
     xlim=c(rango[1],rango[2]),
     ylim=c(rango[1],rango[2]),
     xlab='muestra2d[1]: coordenadas x',
     ylab='muestra2d[2]: coordenadas y',
     main='f(x,y)=y-20x^2-5x+3'
);

with(setNames(muestra2d,c('x','y')),{
  points(x[signos74>=0],y[signos74>=0],col='green');
  points(x[signos74<0],y[signos74<0],col='red');
});
lines(rango[1]:rango[2],rango[1]:rango[2]-20*(rango[1]:rango[2])^2-5*(rango[1]:rango[2])+3)
```



Las conclusiones que podemos sacar de estas gráficas son que los modelos no son lineales y, por tanto, no pueden ser clasificados por una línea recta, como en el caso de la función del ejercicio 6. Serán entonces necesarios otros modelos no lineales para clasificar las muestras obtenidas.

8. Considerar de nuevo la muestra etiquetada en el apartado 6. Modifique las etiquetas de un 10% aleatorio de muestras positivas y otro 10% aleatorio de negativas.

Para cada uno de las funciones recalcularemos el vector de signos con la que toque y variaremos aleatoriamente el 10% de valores positivos y el 10% de valores negativos.

- **Visualice los puntos con las nuevas etiquetas y la recta del apartado 6**

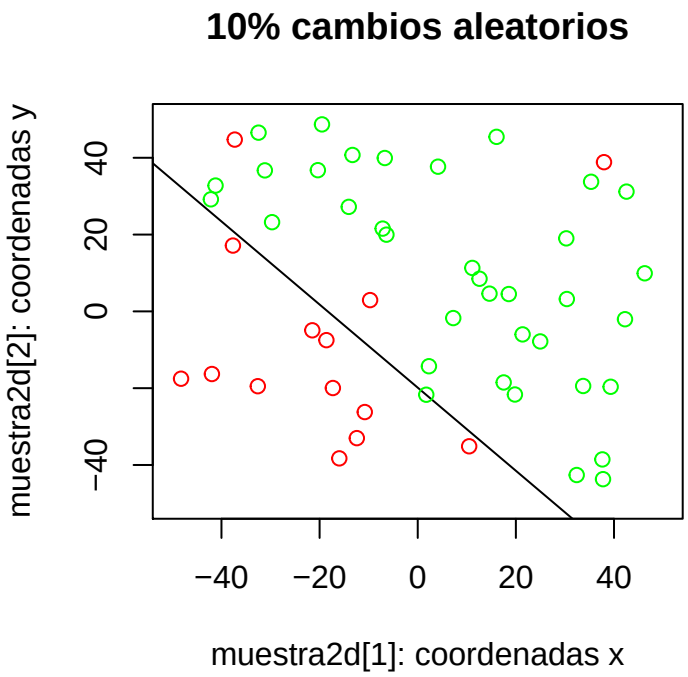
En este apartado vamos a explicar una sola vez lo que haremos para todas las funciones. El código para recalcular el signo y cambiar el 10% de valores para cada signo es el siguiente:

```
signos81 <- sign(muestra2d[[2]] - vars[5]*muestra2d[[1]] - vars[6]);

index_sub<-sample(which(signos81<0),trunc(0.1*length(which(signos81<0))))
signos81[index_sub]<-1

index_sub<-sample(which(signos81>0),trunc(0.1*length(which(signos81>0))))
signos81[index_sub]<-(-1)
```

Pasemos a explicarlo. La primera línea es trivial, nos devuelve el vector de signos en la función para cada punto de muestra2d. La segunda línea devolverá índices aleatorios del vector de signos, ¿cómo? con la función wich. Dicha función nos devuelve, dado un vector, qué posiciones del mismo hacen verdadera una condición. Pues bien, vamos a obtener una muestra del 10% (valor truncado a entero) del número de elementos que cumplen la condición, es decir, vamos a elegir aleatoriamente un 10% de los índices que cumplen la condición, ¿qué condición? que los signos sean mayores o menores que 0 según en la fase que estemos, por eso las cuatro últimas líneas son prácticamente iguales. Las líneas 2 y 3 cambian a positivo el 10% de los valores negativos. Las líneas 4 y 5 hacen justo lo contrario.

Una vez entendido cómo vamos a cambiar los signos y hecho, vamos a dibujar la gráfica correspondiente:

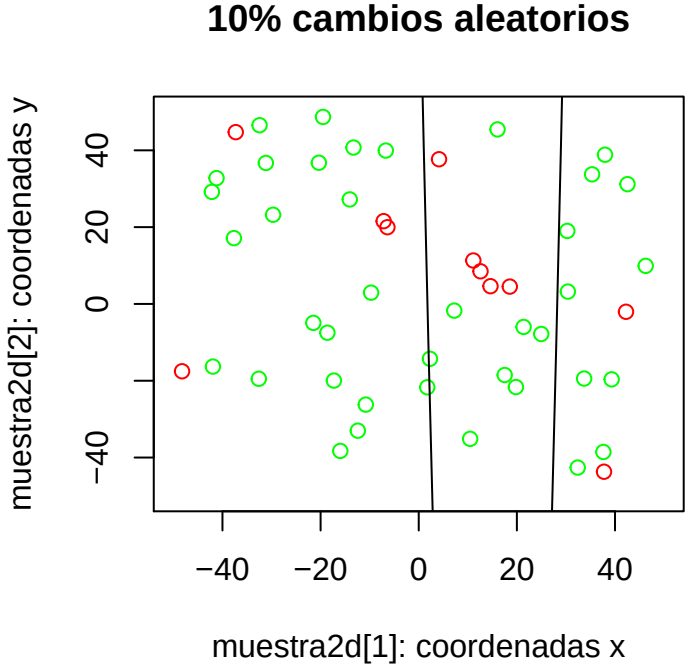
```
plot(
  NA,
  xlim=c(rango[1],rango[2]),
  ylim=c(rango[1],rango[2]),
  xlab='muestra2d[1]: coordenadas x',
  ylab='muestra2d[2]: coordenadas y',
  main='10% cambios aleatorios'
);
abline(vars[6],vars[5]);
with(setNames(muestra2d,c('x','y')),{
  points(x[signos81>=0],y[signos81>=0],col='green');
  points(x[signos81<0],y[signos81<0],col='red');
});
```



- **En una gráfica aparte vistualice de nuevo los mismos putos pero junto con las funciones del apartado 7**
- $f(x, y) = (x - 10)^2 + (y - 20)^2 - 400$

```
signos82 <- sign( (muestra2d[[1]]-10)^2 + (muestra2d[[2]]-20)^2 -400 );

index_sub<-sample(which(signos82<0),trunc(0.1*length(which(signos82<0))))
signos82[index_sub]<-1

index_sub<-sample(which(signos82>0),trunc(0.1*length(which(signos82>0))))
signos82[index_sub]<-(-1)

plot(
  NA,
```

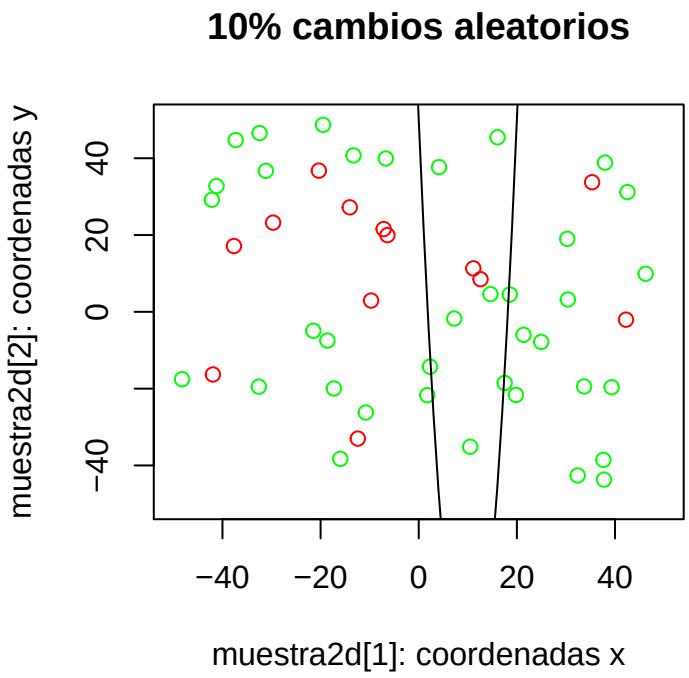
```
  xlim=c(rango[1],rango[2]),
  ylim=c(rango[1],rango[2]),
  xlab='muestra2d[1]: coordenadas x',
  ylab='muestra2d[2]: coordenadas y',
  main='10% cambios aleatorios'
);

with(setNames(muestra2d,c('x','y')),{
  points(x[signos82>=0],y[signos82>=0],col='green');
  points(x[signos82<0],y[signos82<0],col='red');
});
lines(rango[1]:rango[2],(rango[1]:rango[2]-10)^2 + (rango[1]:rango[2]-20)^2 -400)
```



- $f(x, y) = 0.5(x + 10)^2 + (y - 20)^2 - 400$

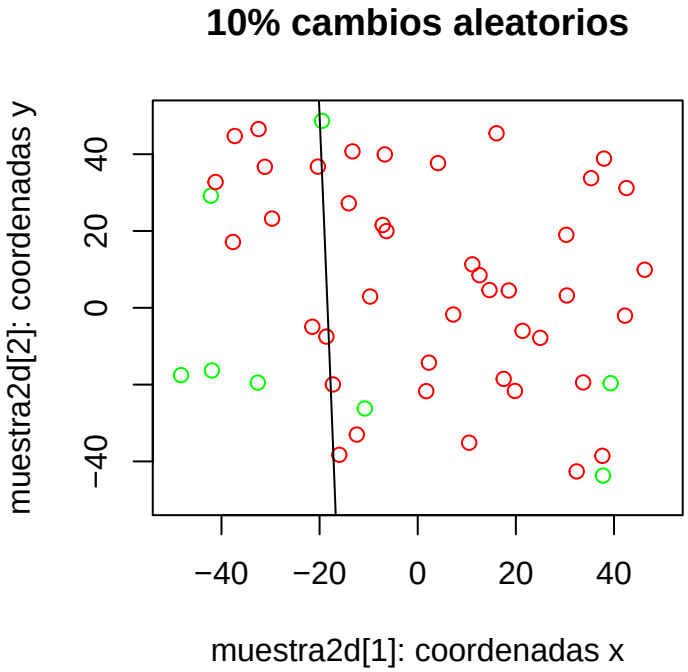
```
signos83 <- sign( 0.5*(muestra2d[[1]]+10)^2+(muestra2d[[2]]-20)^2-400 );

index_sub<-sample(which(signos83<0),trunc(0.1*length(which(signos83<0))))
signos83[index_sub]<-1

index_sub<-sample(which(signos83>0),trunc(0.1*length(which(signos83>0))))
signos83[index_sub]<-(-1)

plot(
  NA,
  xlim=c(rango[1],rango[2]),
  ylim=c(rango[1],rango[2]),
  xlab='muestra2d[1]: coordenadas x',
  ylab='muestra2d[2]: coordenadas y',
  main='10% cambios aleatorios'
);
```

```
with(setNames(muestra2d,c('x','y')),{
  points(x[signos83>=0],y[signos83>=0],col='green');
  points(x[signos83<0],y[signos83<0],col='red');
});
lines(rango[1]:rango[2],0.5*(rango[1]:rango[2]+10)^2+(rango[1]:rango[2]-20)^2-400)
```



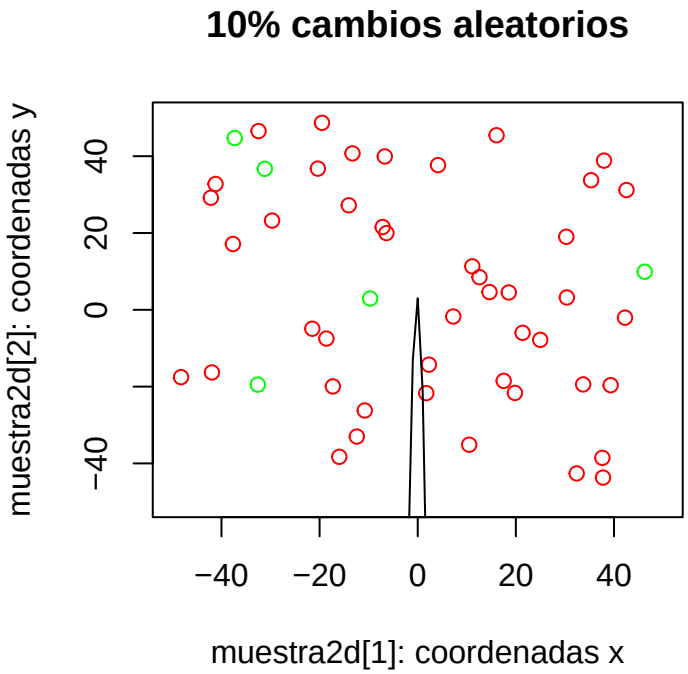
- $f(x, y) = 0.5(x - 10)^2 - (y + 20)^2 - 400$

```
signos84 <- sign( 0.5*(muestra2d[[1]]-10)^2-(muestra2d[[2]]+20)^2-400 );

index_sub<-sample(which(signos84<0),trunc(0.1*length(which(signos84<0))))
signos84[index_sub]<-1

index_sub<-sample(which(signos84>0),trunc(0.1*length(which(signos84>0))))
signos84[index_sub]<-(-1)

plot(
  NA,
  xlim=c(rango[1],rango[2]),
  ylim=c(rango[1],rango[2]),
  xlab='muestra2d[1]: coordenadas x',
  ylab='muestra2d[2]: coordenadas y',
  main='10% cambios aleatorios'
);

with(setNames(muestra2d,c('x','y')),{
  points(x[signos84>=0],y[signos84>=0],col='green');
  points(x[signos84<0],y[signos84<0],col='red');
});
lines(rango[1]:rango[2],0.5*(rango[1]:rango[2]-10)^2-(rango[1]:rango[2]+20)^2-400)
```



- $f(x, y) = y - 20x^2 - 5x + 3$

```
signos85 <- sign( muestra2d[[2]]-20*(muestra2d[[1]])^2-5*(muestra2d[[1]])+3 );

index_sub<-sample(which(signos85<0),trunc(0.1*length(which(signos85<0))))
signos85[index_sub]<-1

index_sub<-sample(which(signos85>0),trunc(0.1*length(which(signos85>0))))
signos85[index_sub]<-(-1)

plot(
  NA,
  xlim=c(rango[1],rango[2]),
  ylim=c(rango[1],rango[2]),
  xlab='muestra2d[1]: coordenadas x',
  ylab='muestra2d[2]: coordenadas y',
  main='10% cambios aleatorios'
);

with(setNames(muestra2d,c('x','y')),{
  points(x[signos85>=0],y[signos85>=0],col='green');
  points(x[signos85<0],y[signos85<0],col='red');
});
lines(rango[1]:rango[2],rango[1]:rango[2]-20*(rango[1]:rango[2])^2-5*(rango[1]:rango[2])+3)
```



Fijémonos en que sólo hemos cambiado el 10% de valores positivos y otro 10% de valores negativos. Como vemos, las gráficas son un caos. Cuando fijamos la muestra a una función, se puede clasificar con un determinado modelo. Si la función cambia, el modelo que utilizábamos deja de ser válido, de hecho, podría ocurrir que, como aquí vemos, la muestra deja de ser clasificable. Observamos pues, que un modelo clasifica una función concreta. Si la muestra cambia, el modelo de clasificación que utilizábamos dejará de ser válido.

En estos casos estamos utilizando en cada caso la propia función como modelo para clasificar la función. Si hacemos cambios (más si son aleatorios), ni siquiera la propia función podrá clasificar la muestra.

Ejercicios de ajuste del algoritmo Perceptrón

1. Implementar la función *sol=ajusta_PLA(datos,label,max_iter,vini)* que calcula el hiperplano solución a un problema de clasificación binaria usando el algoritmo PLA. La entrada *datos* es una matriz donde cada item con su etiqueta está representado por una fila de la matriz, label el vector de etiquetas (cada etiqueta es un valor +1 ó -1), *max_iter* es el número máximo de iteraciones permitidas y *vini* el valor inicial del vector. La salida *sol* devuelve los coeficientes del hiperplano.

El algoritmo perceptrón es capar de obtener una estimación aproximada (incluso a veces encuentra el óptimo) de una solución a un problema de clasificación binaria, es decir, una recta que separa puntos de diferentes categorías (en este caso el signo de una función).

El código que he usado es el siguiente que, como se puede observar, también devuelve el número de iteraciones que se han dado para converger así como el número de errores cometidos. Hay que tener cuidado de que *vini* sea un vector columna de tres filas por una columna (definición matemática de vector):

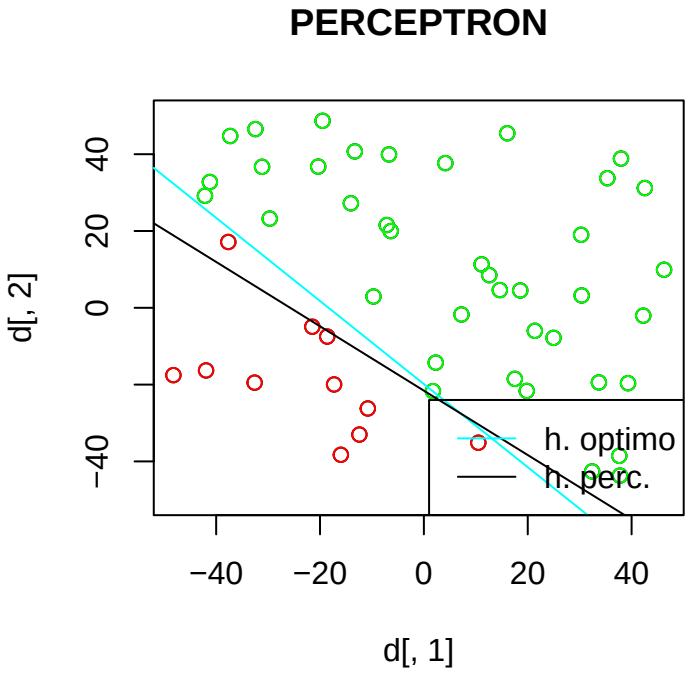
```
ajusta_PLA<-function(datos,label,max_iter=100,vini){
  hubo_cambio<-TRUE
  cont<-0
  niter<-0
  nerrores<-0
  pesos<-vini
```

```
  datos<-matrix(
    c(datos,rep(1,nrow(datos))),
    nrow=nrow(datos),
    ncol=ncol(datos)+1
  )
    while(hubo_cambio & cont<max_iter){
      hubo_cambio<-FALSE
      for (j in 1:nrow(datos)){
        point<-matrix(datos[j,],ncol=1,byrow=FALSE)
        error<-sum(label[j]-(t(pesos)%*%point))

        if(sign(t(pesos)%*%point)!=label[j]){
          bias_ant<-pesos[nrow(pesos),1]
          pesos<-pesos+(label[j]*point)
          pesos[nrow(pesos),1]<-bias_ant+error
          hubo_cambio=TRUE
          nerrores<-nerrores+1
        }
        niter<-niter+1
      }
      cont<-cont+1
    }
  pend <- -(pesos[1,1]/pesos[2,1])
  offs <- -(pesos[3,1]/pesos[2,1])
  c(pend,offs,niter,nerrores)
}
```

Para probar la función con los datos del ejercicio 6 del apartado anterior, hay que adaptar las estructuras de datos. Rellenaré una matriz `d` con dos columnas tales que son los vectores de la lista `muestra2d`. A continuación llamaré al algoritmo para 100 épocas (100 épocas * 50 puntos = 5000 iteraciones como máximo), que pueden no completarse, es decir, ofrecer una solución antes siempre que no haya cambios en una época. Con los resultados dibujaré una gráfica que mostrará los puntos del ejercicio 6 del apartado anterior, junto con la recta óptima (en azul) y la recta que ha calculado el perceptrón.

```
signos <- sign(muestra2d[[2]] - vars[5]*muestra2d[[1]] - vars[6])
d<-matrix(unlist(muestra2d),ncol=2,byrow=FALSE);
#Lo mejor que puedas encontrar con 1000 épocas (1000000 iteraciones)
resultados<-ajusta_PLA(d,signos,100,matrix(c(0,0,0),nrow=3,ncol=1))
plot(d[,1],d[,2],main="PERCEPTRON",ylim=c(-50,80))
points(d[,1][signos>=0],d[,2][signos>=0],col='green');
points(d[,1][signos<0],d[,2][signos<0],col='red');
#Recta óptima
abline(vars[6],vars[5],col='cyan')
#Recta ajustada
abline(resultados[2],resultados[1])
#Añadir leyenda
legend(0,80,
       c("h. optimo","h. perceptron"),
       lty=c(1,1),
       lwd=c(1,1),
       col=c("cyan","black"))
```

El resultado de la ejecución es el siguiente:



2. Ejecutar el algoritmo PLA con los valores simulados en el ejercicio 6 del apartado anterior inicializando el algoritmo con el vector cero y con vectores de números aleatorios en [0,1] (10 veces). Anotar el número medio de iteraciones necesarias en ambos para converger. Valorar el resultado.

Dado que mi codificación del algoritmo perceptrón permite saber el número de iteraciones que se han necesitado para converger, vamos a hacer dos bucles `for` que lo calculen. Para que el tiempo de ejecución no se dispare, podremos 100 épocas (máximo) en cada ejecución. Para ello, ejecutamos el siguiente código:

```
suma_c<-0
for(i in 1:10){
  resultados_c<-ajusta_PLA(d,signos6,100,matrix(c(0,0,0),nrow=3,ncol=1));
  print(resultados_c[3])
  suma_c<-suma_c+resultados_c[3]
}
```

```
## [1] 5000
## [1] 5000
## [1] 5000
## [1] 5000
## [1] 5000
## [1] 5000
## [1] 5000
## [1] 5000
## [1] 5000
## [1] 5000
```

```
print(
  paste(
    "La media de iteraciones para inicializacion 0 es:", suma_c/10
  )
)
```

```
## [1] "La media de iteraciones para inicializacion 0 es: 5000"
```

```
suma_r<-0
for(i in 1:10){
  resultados_r<-ajusta_PLA(d,signos6,100,matrix(runif(3, min=0, max=1),nrow=3,ncol=1));
  print(resultados_r[3])
  suma_r<-suma_r+resultados_r[3]
}
```

```
## [1] 5000
## [1] 5000
## [1] 5000
## [1] 5000
## [1] 5000
## [1] 5000
## [1] 5000
## [1] 5000
## [1] 5000
## [1] 5000
```

```
print(
  paste(
    "La media de iteraciones para inicializacion aleatoria es:", suma_r/10
  )
)
```

```
## [1] "La media de iteraciones para inicializacion aleatoria es: 5000"
```

Lo normal es que, desde una inicialización aleatoria ya hayamos quitado un poco de trabajo al perceptrón, por lo que de forma general, una inicialización aleatoria hará menos iteraciones, aunque en experimentos repetidos fluctuará más. Si inicializamos en 0, todas las iteraciones serán iguales, por lo que la media será igual al numero de iteraciones de un solo experimento. Si desde una inicialización a 0 hay que hacer pocas iteraciones, la media sería igualmente muy buena, pero lo normal es que esto no ocurra y que una inicialización aleatoria sea mejor en términos de número de iteraciones.

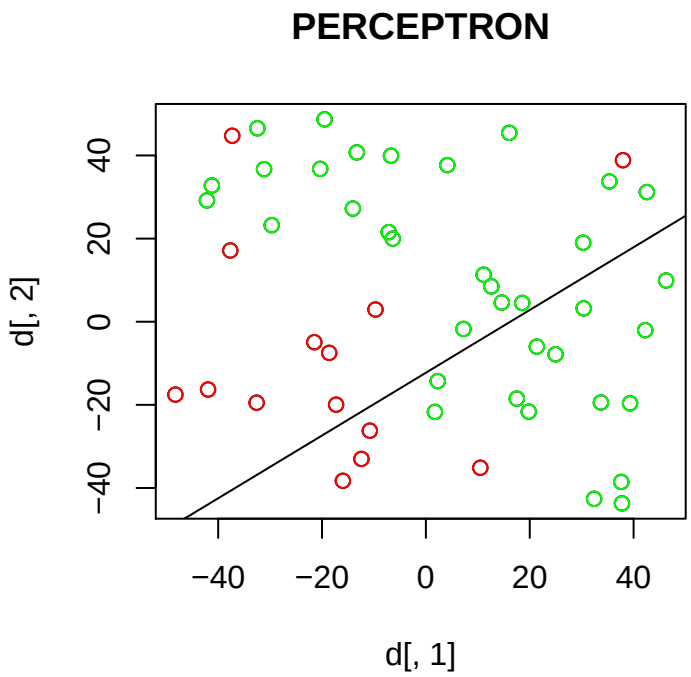
3. Ejecutar el algoritmo PLA con los datos generados en el ejercicio 8 del apartado anterior usando valores de 10, 100 y 1000 para max_iter. Etiquetar los datos de la muestra usando la función solución encontrada y contar el número de errores respecto de las etiquetas originales

Ejecutaremos el siguiente codigo:

```
d<-matrix(unlist(muestra2d),ncol=2,byrow=FALSE);

resultados<-ajusta_PLA(d,signos81,10,matrix(c(0,0,0),nrow=3,ncol=1))
signos_est <- sign(muestra2d[[2]] - resultados[1]*muestra2d[[1]] - resultados[2]);
print(paste("Errores cometidos con 10 iters: ",length(signos81[signos_est!=signos81])));
```

```
## [1] "Errores cometidos con 10 iters:  24"
```

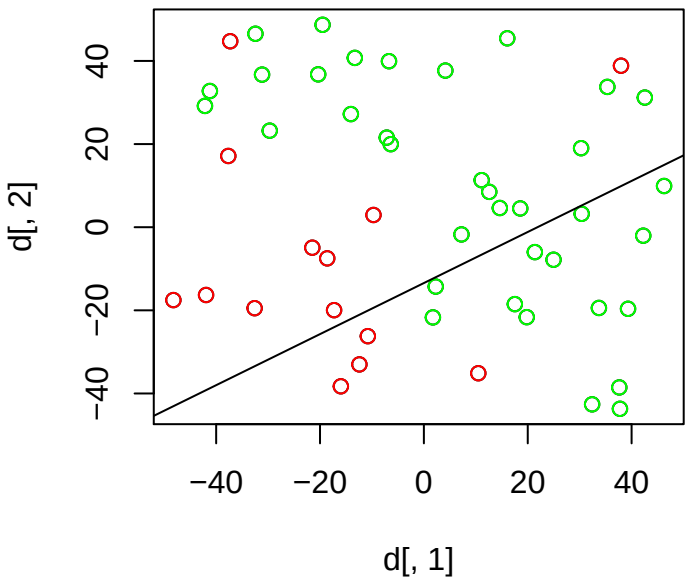
```
plot(d[,1],d[,2],main="PERCEPTRON")
points(d[,1][signos81>=0],d[,2][signos81>=0],col='green');
points(d[,1][signos81<0],d[,2][signos81<0],col='red');
abline(resultados[2],resultados[1])
```




```
resultados<-ajusta_PLA(d,signos81,100,matrix(c(0,0,0),nrow=3,ncol=1))
signos_est <- sign(muestra2d[[2]] - resultados[1]*muestra2d[[1]] - resultados[2]);
print(paste("Errores cometidos con 100 iters: ",length(signos81[signos_est!=signos81])));
```

```
## [1] "Errores cometidos con 100 iters:  24"
```

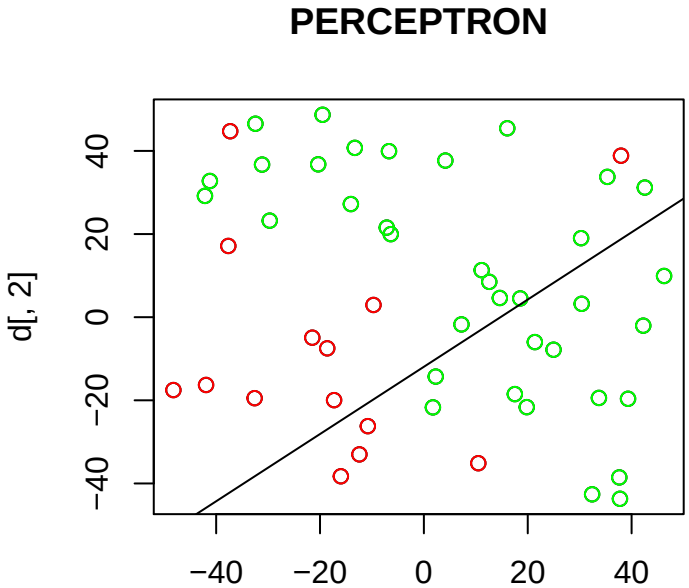
```
plot(d[,1],d[,2],main="PERCEPTRON")
points(d[,1][signos81>=0],d[,2][signos81>=0],col='green');
points(d[,1][signos81<0],d[,2][signos81<0],col='red');
abline(resultados[2],resultados[1])
```




```
resultados<-ajusta_PLA(d,signos81,1000,matrix(c(0,0,0),nrow=3,ncol=1))
signos_est <- sign(muestra2d[[2]] - resultados[1]*muestra2d[[1]] - resultados[2]);
print(paste("Errores cometidos con 1000 iters: ",length(signos81[signos_est!=signos81])));
```

```
## [1] "Errores cometidos con 1000 iters:  24"
```

```
plot(d[,1],d[,2],main="PERCEPTRON")
points(d[,1][signos81>=0],d[,2][signos81>=0],col='green');
points(d[,1][signos81<0],d[,2][signos81<0],col='red');
abline(resultados[2],resultados[1])
```



Como vemos, no puede converger, dado que los datos no son separables.

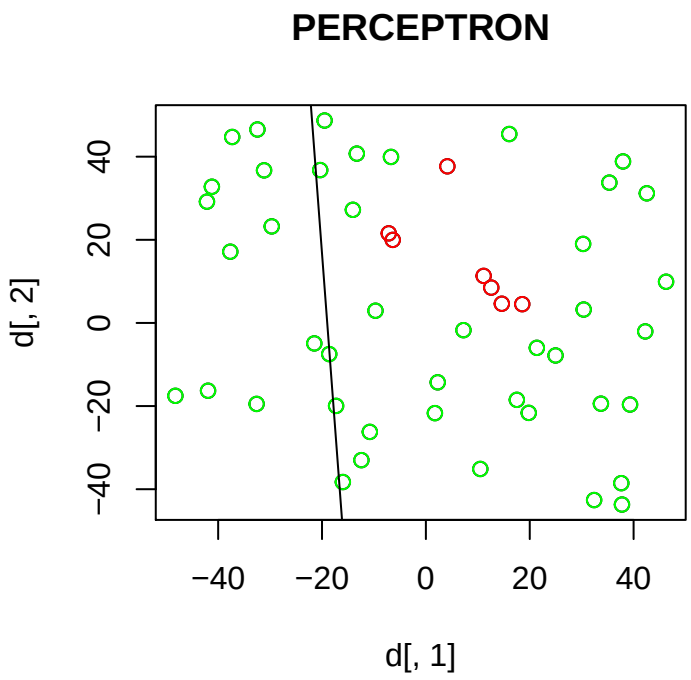
4. Repetir el análisis del punto anterior usando la primera función del ejercicio 7 del apartado 3.2

Sólo cambiaremos los signos de la función a utilizar. Posteriormente ajustaremos el signo de la función al de la recta obtenida con ajusta_PLA y compararemos los resultados.

```
d<-matrix(unlist(muestra2d),ncol=2,byrow=FALSE);

resultados<-ajusta_PLA(d,signos71,10,matrix(c(0,0,0),nrow=3,ncol=1))
signos_est <- sign(muestra2d[[2]] - resultados[1]*muestra2d[[1]] - resultados[2])
print(paste("Errores cometidos con 10 iters: ",length(signos71[signos_est!=signos71])));
```

```
## [1] "Errores cometidos con 10 iters:  19"
```

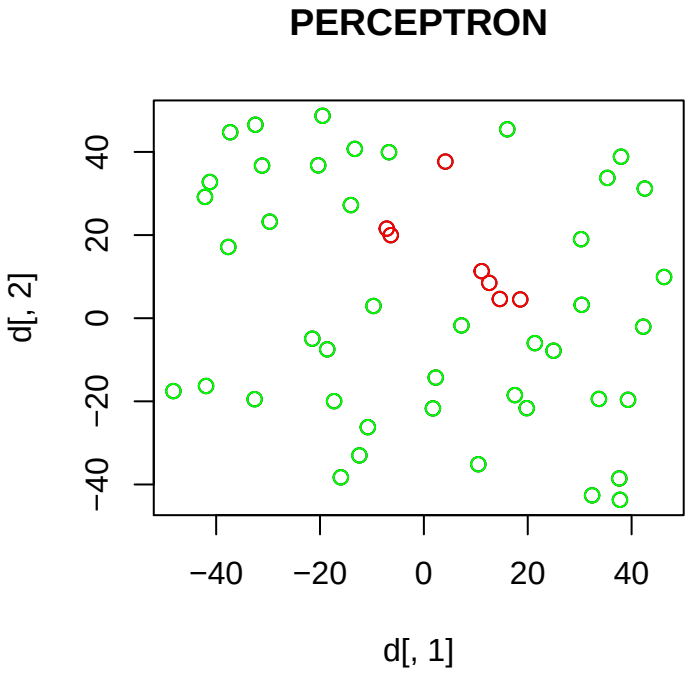
```
plot(d[,1],d[,2],main="PERCEPTRON")
points(d[,1][signos71>=0],d[,2][signos71>=0],col='green');
points(d[,1][signos71<0],d[,2][signos71<0],col='red');
abline(resultados[2],resultados[1])
```




```
resultados<-ajusta_PLA(d,signos71,100,matrix(c(0,0,0),nrow=3,ncol=1))
signos_est <- sign(muestra2d[[2]] - resultados[1]*muestra2d[[1]] - resultados[2])
print(paste("Errores cometidos con 100 iters: ",length(signos71[signos_est!=signos71])));
```

```
## [1] "Errores cometidos con 100 iters:  7"
```

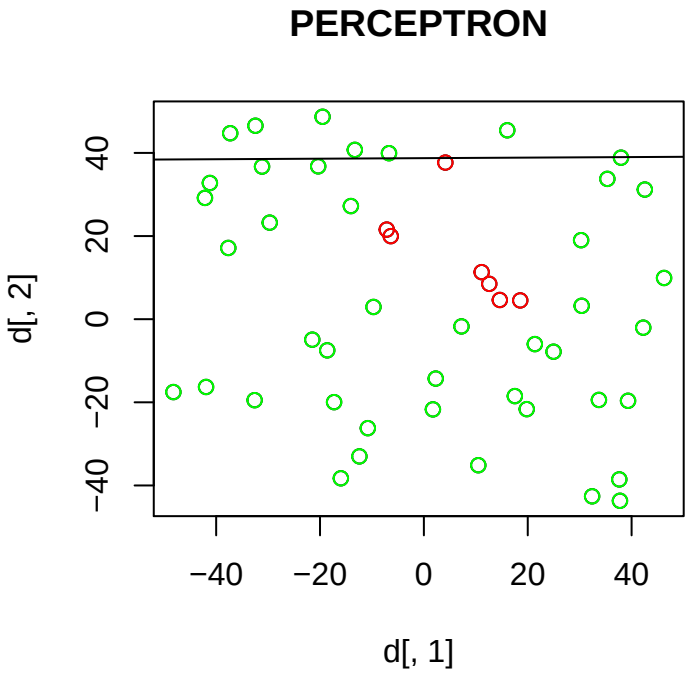
```
plot(d[,1],d[,2],main="PERCEPTRON")
points(d[,1][signos71>=0],d[,2][signos71>=0],col='green');
points(d[,1][signos71<0],d[,2][signos71<0],col='red');
abline(resultados[2],resultados[1])
```




```
resultados<-ajusta_PLA(d,signos71,1000,matrix(c(0,0,0),nrow=3,ncol=1))
signos_est <- sign(muestra2d[[2]] - resultados[1]*muestra2d[[1]] - resultados[2])
print(paste("Errores cometidos con 1000 iters: ",length(signos71[signos_est!=signos71])));
```

```
## [1] "Errores cometidos con 1000 iters:  37"
```

```
plot(d[,1],d[,2],main="PERCEPTRON")
points(d[,1][signos71>=0],d[,2][signos71>=0],col='green');
points(d[,1][signos71<0],d[,2][signos71<0],col='red');
abline(resultados[2],resultados[1])
```



Dado que esta función es cuadrática, no se puede evaluar con una recta y, por ende, nunca va a converger, por muchas iteraciones que le pongamos. Lo más probable es que el perceptrón siempre coloque la recta cerca del mismo sitio, porque no sabe muy bien cómo corregir los errores, por eso, el número de errores es parecido independientemente del número de iteraciones.

5. Modifique la funcion ajusta_PLA para que le permita visualizar los datos y soluciones que va encontrando a lo largo de las iteraciones. Ejecute con la nueva versión el ejercicio 3 del apartado 3.2.

El nuevo algoritmo es el siguiente. He encontrado un método para extraer un gif superponiendo gráficas en R. Añadiendo la biblioteca `animation`, cambiando las opciones de animación (10 gráficas por segundo), y con saveGIF.

```
ajusta_PLA<-function(datos,label,max_iter=100,vini){
  library(animation)

  hubo_cambio<-TRUE
  cont<-0
  niter<-0
  nerrores<-0
  pesos<-vini
  datos<-matrix(
    c(datos,rep(1,nrow(datos))),
    nrow=nrow(datos),
    ncol=ncol(datos)+1
  )

  ani.options(interval=0.1);
  saveGIF({
    while(hubo_cambio & cont<max_iter){
      hubo_cambio<-FALSE
      for (j in 1:nrow(datos)){
        point<-matrix(datos[j,],ncol=1,byrow=FALSE)
        error<-sum(label[j]-(t(pesos)%*%point))

        if(sign(t(pesos)%*%point)!=label[j]){
          bias_ant<-pesos[nrow(pesos),1]
          pesos<-pesos+(label[j]*point)
          pesos[nrow(pesos),1]<-bias_ant+error
          hubo_cambio=TRUE
          nerrores<-nerrores+1
        }
        niter<-niter+1
      }

      plot(datos[,1],datos[,2],main=paste("PERCEPTRON", niter, sep=' '))
      points(datos[,1][label>=0],datos[,2][label>=0],col='green');
      points(datos[,1][label<0],datos[,2][label<0],col='red');
      pend <- -(pesos[1,1]/pesos[2,1])
      offs <- -(pesos[3,1]/pesos[2,1])
      abline(offs,pend)

      cont<-cont+1
    }
```

```
  })
  pend <- -(pesos[1,1]/pesos[2,1])
  offs <- -(pesos[3,1]/pesos[2,1])
  c(pend,offs,niter,nerrores)
}

dat_unif<-simula_unif(2,50,c(-50,50))
recta<- simula_recta(c(-50,50))
signos5 <- sign(dat_unif[[2]] - recta[1]*dat_unif[[1]] - recta[2])
muestraej5<-matrix(unlist(dat_unif),ncol=2,byrow=FALSE);
resultados<-ajusta_PLA(muestraej5,signos5,10,matrix(c(0,0,0),nrow=3,ncol=1))
resultados<-ajusta_PLA(muestraej5,signos5,100,matrix(c(0,0,0),nrow=3,ncol=1))
resultados<-ajusta_PLA(muestraej5,signos5,1000,matrix(c(0,0,0),nrow=3,ncol=1))
```

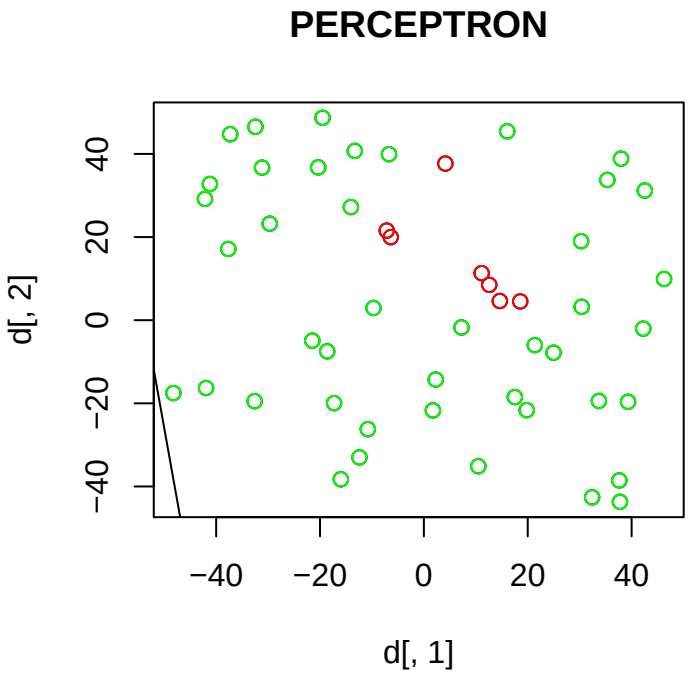
Se adjuntan los ejemplos **max__10__iters.gif**, **max__100__iters.gif** y **max__1000__iters.gif** que, dado que en 10 iteraciones converge, serán gifs exactamente iguales.

6. A la vista de la conducta de las soluciones observada en el apartado anterior, proponga e implemente una modificación de la funcion original ajusta__PLA__MOD() que permita obtener soluciones razonables sobre datos no linealmente separables. Mostrar y valorar el resultado encontrado usando los datos del ejercicio 7 del apartado 3.2

Vamos a proceder a modificar el algoritmo quedándonos con el número mínimo de errores. Compararemos en cada iteración el número de errores que tenemos con un global (`best_nerrors`) y sólo cambiaremos la pendiente y el offset de la recta en caso de que el número de errores sea menor. Finalmente devolveremos las variables `best_pend` y `best_off` como los mejores coeficientes encontrados. Esto es lo que se conoce como la versión "pocket" del algoritmo PLA.

```
ajusta_PLA<-function(datos,label,max_iter=100,vini){
  hubo_cambio<-TRUE
  cont<-0
  niter<-0
  nerrores<-0
  pesos<-vini
  datos<-matrix(
    c(datos,rep(1,nrow(datos))),
    nrow=nrow(datos),
    ncol=ncol(datos)+1
  )
  best_pend<-0
  best_off<-0
  best_nerrors<-nrow(datos)+1
    while(hubo_cambio & cont<max_iter){
      hubo_cambio<-FALSE
      for (j in 1:nrow(datos)){
        point<-matrix(datos[j,],ncol=1,byrow=FALSE)
        error<-sum(label[j]-(t(pesos)%*%point))

        if(sign(t(pesos)%*%point)!=label[j]){
          bias_ant<-pesos[nrow(pesos),1]
          pesos<-pesos+(label[j]*point)
          pesos[nrow(pesos),1]<-bias_ant+error
          hubo_cambio=TRUE
```

```
          nerrores<-nerrores+1
        }
        niter<-niter+1
      }

      ######Aquí está la parte novedosa######
      pend <- -(pesos[1,1]/pesos[2,1])
      offs <- -(pesos[3,1]/pesos[2,1])
      sgn <- sign(datos[,2] - pend*datos[,1] - offs)
      if(length(label[sgn!=label])<best_nerrors){
        best_pend<-pend
        best_off<-offs
        best_nerrors<-length(label[sgn!=label])
      }
      ########################################
      cont<-cont+1
    }
  c(best_pend,best_off,niter,nerrores)
}



d<-matrix(unlist(muestra2d),ncol=2,byrow=FALSE);
resultados<-ajusta_PLA(d,signos71,100,matrix(c(0,0,0),nrow=3,ncol=1))
signos_est <- sign(muestra2d[[2]] - resultados[1]*muestra2d[[1]] - resultados[2])
plot(d[,1],d[,2],main="PERCEPTRON")
points(d[,1][signos71>=0],d[,2][signos71>=0],col='green');
points(d[,1][signos71<0],d[,2][signos71<0],col='red');
abline(resultados[2],resultados[1])
```



Se adjunta el ejemplo **PocketExample.gif** sobre la primera función del ejercicio 7.

Ejercicios sobre regresión lineal

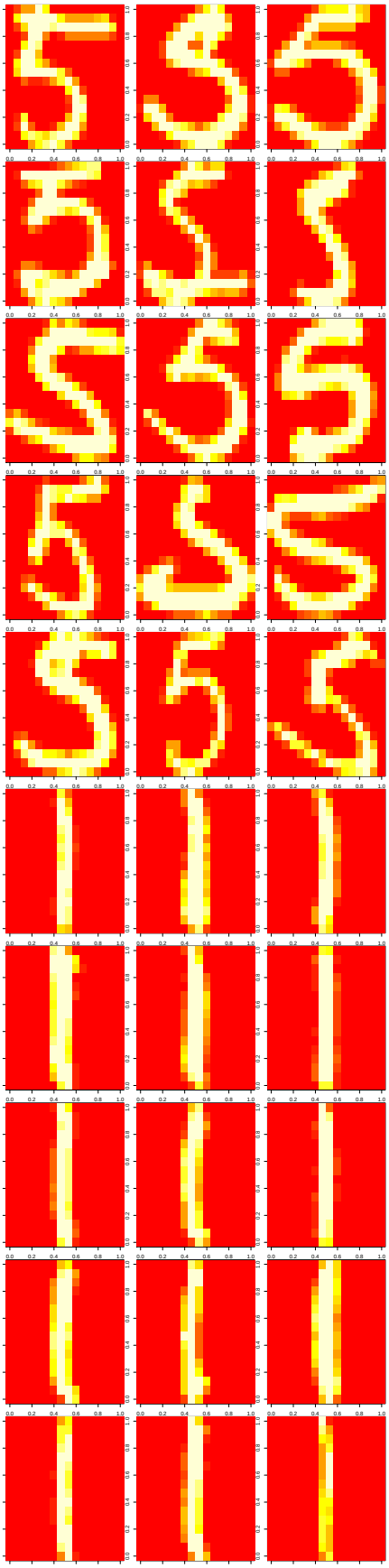
1.Abra el fichero ZipDigits.info disponible en la web del curso y lea la descripción de la representación numérica de la base de datos de números manuscritos que hay en el fichero ZipDigits.train

El archivo explica un poco cómo se representan los datos en el archivo de entrenamiento y test. Cada fila de dichos archivos es un número del que se sabe su ID (clase) y los valores observados de dicho número para los 256 valores de gris. El archivo de información tambien explica cuántos números de cada clase hay en cada uno de los archivos con respecto al total y también nos informa de qué tanto por uno ocupa en cada archivo cada número. En el test, según nos explican sólo se tiene una tasa de error del 2.5%, lo cual es muy bueno.

2. Lea el fihero ZipDigits.train dentro de su código y visualize las imágenes. Seleccione sólo las instancias de los números 1 y 5. Guárdelas como matrices de tamaño 16x16.

El código que expongo a continuación nos permitirá cargar el dataset en una variable (dataframe). Con `rbind` uniremos en un solo dataframe las instancias de 5 y las instancias de 1. Una vez lo tenemos, vamos a visualizarlas con el comando image, que trabaja sobre matrices, entonces, convertiremos cada fila del dataframe en una matriz, quitando la primera columna que es la salida o la clase a la que pertenece el ejemplo. Sacaré 30 gráficas solo, pues pueden considerarse una muestra de todas las muestras que hay.

```
zip<-read.table("../ZipDigits/zip.train")
zip51<-rbind(zip[zip[,1]==5,],zip[zip[,1]==1,])
par(mfrow = c(10,3), mar=c(1,1,1,1))
for(i in 1:30){
  image(matrix(unlist(zip51[i*10,2:257]),nrow=16,ncol=16)[,16:1])
}
```



3. Para cada matriz de números calcularemos su valor medio y su grado de simetría vertical. Para calcular la simetria calcularemos la suma del valor absoluto de las diferencias en cada píxel entre la imagen original y la imagen que obtenemos invirtiendo el orden de las columnas. Finalmente le cambiamos el signo.

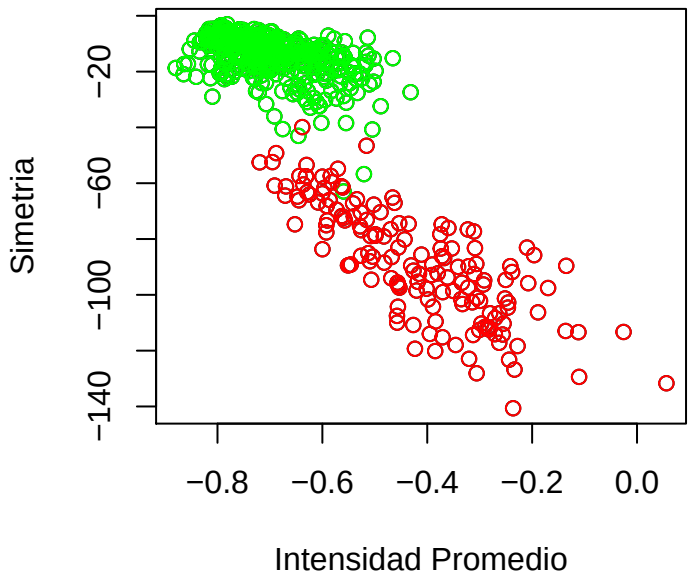
La función que nos permite calcular esos datos es la siguiente. Pasaremos el dataframe `misDatos` con todas las filas (directamente leídas de un fichero). Nos devolverá una matriz de dos columnas, con tantas filas como tenga `misDatos`. La primera columna indica el valor medio de la matriz actual y la segunda, el grado de simetría vertical.

```
valorMedioSimetria<-function(misDatos){
  puntos<-c()
  for(i in 1:nrow(misDatos)){
    filaActual<-matrix(unlist(misDatos[i,2:257]),nrow=16,ncol=16)
    simetriaVertical<-(-sum(abs(filaActual[,1:8]-filaActual[,9:16])))
    valorMedio<-mean(filaActual)
    point<-c(valorMedio,simetriaVertical)
    puntos<-c(puntos,point)
  }
  matrix(puntos,nrow=nrow(misDatos),ncol=2,byrow=TRUE)
}
```

4. Representar en los ejes $\{X = Intensidad\ Promedio, Y = Simetría\}$ las instancias seleccionadas de 1's y 5's.

Para representar los ejes, basta con hacer un plot. Por diferenciar, pintaremos los puntos cuya clase es 1 de color verde y de rojo los puntos cuya salida es 5.

```
puntos<-valorMedioSimetria(zip51)
plot(puntos[,1],puntos[,2],xlab="Intensidad Promedio",ylab="Simetria")
points(puntos[,1][zip51[,1]==1],puntos[,2][zip51[,1]==1],col='green')
points(puntos[,1][zip51[,1]==5],puntos[,2][zip51[,1]==5],col='red')
```



5. Implementar la función *sol=Regress_Lin(datos,label)* que permita ajustar a un modelo de regresión lineal (usar SVD). Los datos de entrada se interpretan igual que en clasificación.

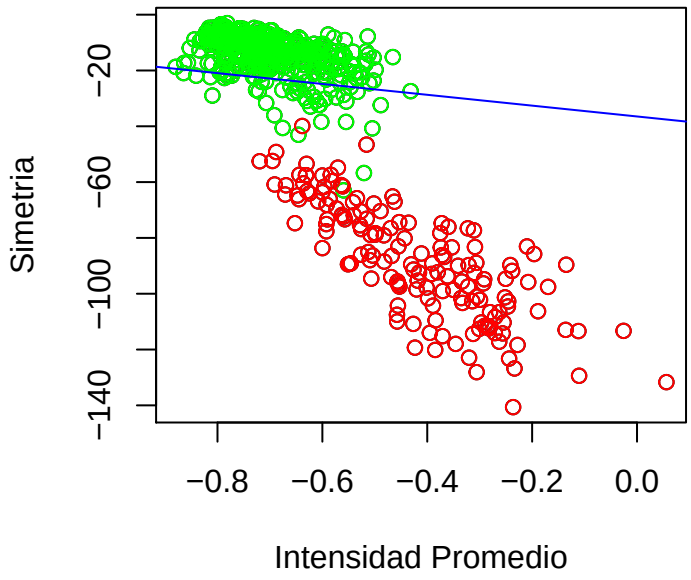
Según la teoría, el algoritmo de regresión lineal consta de tres pasos. El primero consiste en representar los descriptores como una matriz donde los de cada ejemplo sean una fila: `X`. Del mismo modo, las etiquetas de los ejemplos deben estar en una matriz donde la fila i es la clase del ejemplo i en de la matriz `X`, esta es la matriz `Y`. El segundo paso es computar la pseudoinversa de la matriz `X` que se calcula como $X^\dagger = (X^T X)^{-1} X^T$. Dada su descomopsición en valores singulares $X = U\Sigma V^T$, podemos aprovechar sus propiedades y tenemos que: $X^\dagger = V\Sigma^{-1}U$. El tercer y último paso es, calculada ya la pseudoinversa, devolver $w = X^\dagger Y$. Se supone que w contiene los coeficientes que intervienen en la ecuación de la recta, es decir, una recta tiene la ecuación $ax + by + c = 0$, que se puede ver como $w_1 X_1 + w_2 X_2 + w_3 = 0$. Despejando, tenemos que $X_2 = -\frac{w_1}{w_2} X_1 - \frac{w_3}{w_2}$, que corresponde a una ecuación de la recta tipo $y = mx + b$, entonces la pendiente (m) será $-\frac{w_1}{w_2}$ y el offset (b) $-\frac{w_3}{w_2}$.

Según las transparencias de teoría, los errores son calculables, pero no se usan en el algoritmo.

```
Regress_Lin<-function(data,label){
  data<-data[,ncol(data):1]
  data<-cbind(rep(1,nrow(data)),data)
  x<-t(data)%*%data
  udv<-svd(x)
  x.inv <- udv$v %*% diag(1/udv$d) %*% t(udv$u)
  x.pseudo.inv <- x.inv %*% t(data)
  w <- x.pseudo.inv %*% label
  pend <- -(w[1,1]/w[2,1])
  offs <- -(w[3,1]/w[2,1])
  c(pend,offs)
}
```

6. Ajustar un modelo de regresión lineal a los datos de (Itensidad promedio, Simetría) y pintar la solución junto con los datos. Valorar el resultado.

```
plot(puntos[,1],puntos[,2],xlab="Intensidad Promedio",ylab="Simetria")
points(puntos[,1][zip51[,1]==1],puntos[,2][zip51[,1]==1],col='green')
points(puntos[,1][zip51[,1]==5],puntos[,2][zip51[,1]==5],col='red')

datos<-matrix(c(puntos[,1],puntos[,2]),ncol=2,byrow=FALSE)
label<-matrix(zip51[,1],ncol=1)

solucion<-Regress_Lin(datos,label)

abline(solucion[2],solucion[1],col="blue")
```



Vemos cómo regresión lineal nos da justamente la recta que minimiza la suma de errores (distancias) entre cada punto y la recta.

7. En este ejercicio exploramos cómo funciona regresión lineal en problemas de clasificación. Para ello generamos datos usando el mismo procedimiento que en ejercicios anteriores. Suponemos $X = [-10, 10]x[-10, 10]$ y elegimos muestras aleatorias uniformes dentro de X. La función f en cada caso será una recta aleatoria que corta a X y que asigna etiqueta a cada punto con el valor de su signo. En cada apartado generamos una muestra y le asignamos etiqueta con la función f generada. En cada ejecución generamos una nueva función f.

a) Fijar el tamaño de muestra $N = 100$. Usar regresión lineal para encontrar g y evaluar E_{in}, (el porcentaje de puntos incorrectamente clasificados). Repetir el experimento 1000 veces y promediar los resultados ¿Qué valor obtiene para E_{in}?

```
rango=c(-10,10)
promedio_E_in<-0
for(i in 1:1000){
  muestra<-matrix(unlist(simula_unif(2,100,c(rango[1],rango[2]))),ncol=2,byrow=FALSE);
  vars<-simula_recta(c(rango[1],rango[2]))
  signos <- matrix(sign(muestra[,2] - vars[5]*muestra[,1] - vars[6]),ncol=1)


  resultados<-Regress_Lin(muestra,signos)
  signos_est <- sign(muestra[,2] - resultados[1]*muestra[,1] - resultados[2])
  promedio_E_in<-promedio_E_in+length(signos[signos_est!=signos])

}
promedio_E_in<-(promedio_E_in/1000)/length(signos)
print(paste("Promedio E_in: ",promedio_E_in))
```

```
## [1] "Promedio E_in:  0.45039"
```

b) Fijar el tamaño de muestra N=100. Usar regresón lineal para encontrar g y evaluar E_{out}. Para ello generar 1000 puntos nuevos y usarlos para estimar el error fuera de la muestra, E_{out} (porcentaje de puntos mal clasificados). De nuevo ejecutar el experimento 1000 veces y tomar el promedio. ¿Qué valor obtiene de E_{out}? Valore los resultados.

```
rango=c(-10,10)
promedio_E_out<-0
for(i in 1:1000){
  muestra<-matrix(unlist(simula_unif(2,100,c(rango[1],rango[2]))),ncol=2,byrow=FALSE);
  vars<-simula_recta(c(rango[1],rango[2]))
  signos <- matrix(sign(muestra[,2] - vars[5]*muestra[,1] - vars[6]),ncol=1)

  resultados<-Regress_Lin(muestra,signos)

  muestra_test<-matrix(unlist(simula_unif(2,1000,c(rango[1],rango[2]))),ncol=2,byrow=FALSE)
  signos_reales_test <- sign(muestra_test[,2] - vars[5]*muestra_test[,1] - vars[6])
  signos_est_test <- sign(muestra_test[,2] - resultados[1]*muestra_test[,1] - resultados[2])

  promedio_E_out<-promedio_E_out+length(signos_reales_test[signos_reales_test!=signos_est_test])/length
}
promedio_E_out<-promedio_E_out/1000
print(paste("Promedio E_out: ",promedio_E_out))
```

```
## [1] "Promedio E_out:  0.454479"
```

Por lo general (y como es lógico), el error en test aumenta un poco con respecto al error en la muestra aprendida.

c) Ahora fijamos $N = 10$, ajustamos regresión lineal y usamos el vector de pesos encontrado como un vector inicial de pesos para PLA. Ejecutar PLA hasta que converja a un vector de pesos final que separe completamente la muestra de entrenamiento. Anote el número de iteraciones y repita el experimento 1000 veces. ¿Cual es valor promedio de iteraciones que tarda PLA en converger? (En cada iteración de PLA elija un punto aleatorio del conjunto de mal clasificados). Valore los resultados.

Vamos a redefinir PLA_pocket y Regresión lineal para que nos devuelva solo los pesos:

```
Regress_Lin<-function(data,label,voltear){
  if(voltear){
    data<-cbind(data,1)
    data<-data[,ncol(data):1]
  }

  x<-t(data)%*%data
  udv<-svd(x)
  x.inv <- udv$v %*% diag(1/udv$d) %*% t(udv$u)
  x.pseudo.inv <- x.inv %*% t(data)
  w <- x.pseudo.inv %*% label
  w
}
```

```
PLA_pocket<-function(datos,label,max_iter=100,vini){
  hubo_cambio<-TRUE
  cont<-0
  niter<-0
  nerrores<-0
  pesos<-vini
  datos<-cbind(datos,rep(1,nrow(datos)))
  best_nerrors<-nrow(datos)+1
  best_pesos<-pesos
    while(hubo_cambio & cont<max_iter){
      hubo_cambio<-FALSE
      for (j in 1:nrow(datos)){
        point<-matrix(datos[j,],ncol=1,byrow=FALSE)
        error<-sum(label[j]-(t(pesos)%*%point))

        if(sign(t(pesos)%*%point)!=label[j]){
          umbral_ant<-pesos[nrow(pesos),1]
          pesos<-pesos+(label[j]*point)
          pesos[nrow(pesos),1]<-umbral_ant+error
          hubo_cambio=TRUE
          nerrores<-nerrores+1
        }
        niter<-niter+1
      }

      if(nerrores<best_nerrors){
        best_nerrors<-nerrores
        best_pesos<-pesos
      }

      cont<-cont+1
    }
  c(best_pesos,cont)
}
```

Aplicamos lo que se nos pide:

```
rango=c(-10,10)
promedio_iter<-0
cont<-0
for(i in 1:1000){
  muestra<-matrix(unlist(simula_unif(2,10,c(rango[1],rango[2]))),ncol=2,byrow=FALSE)
  vars<-simula_recta(c(rango[1],rango[2]))
  signos <- matrix(sign(muestra[,2] - vars[5]*muestra[,1] - vars[6]),ncol=1)

  resultados<-Regress_Lin(muestra,signos,TRUE)
  while(is.na(resultados[1])){
    muestra<-matrix(unlist(simula_unif(2,10,c(rango[1],rango[2]))),ncol=2,byrow=FALSE)
    vars<-simula_recta(c(rango[1],rango[2]))
    signos <- matrix(sign(muestra[,2] - vars[5]*muestra[,1] - vars[6]),ncol=1)

    resultados<-Regress_Lin(muestra,signos,TRUE)
  }
```

```
  resultados<-PLA_pocket(muestra,signos,1000,matrix(resultados[3:1]))

  promedio_iter<-promedio_iter+resultados[length(resultados)]
  cont<-cont+1
}
promedio_iter<-promedio_iter/1000
print(paste("Promedio iteraciones: ",promedio_iter))
```

```
## [1] "Promedio iteraciones:  14.328"
```

En teoría, si en PLA partimos desde una solución que nos proporciona regresión lineal (no es un proceso iterativo), tardaremos menos iteraciones en converger que si partiéramos de 0. No siempre es así pues podría haber soluciones muy cercanas al vector de pesos 0 que se alcanzaran en poco tiempo partiendo del vector 0.

8. En este ejercicio exploramos el uso de tranformaciones no lineales. Consideremos la función objetivo $f(x_1, x_2) = sign(x_1^2+x_2^2-0,6)$. Generar una muestra de entrenamiento de N=1000 puntos a partir de $X = [-10, 10]x[-10, 10]$ muestrando cada punto $x \in X$ uniformemente. Generar las salidas usando el signo de la función en los puntos muestreados. Generar ruido sobre las etiquetas cambiando el signo de las salidas a un 10% de puntos del conjunto aleatorio generado.

a) Ajustar regresión lineal, para estimar los pesos w. Ejecutar el experimento 1000 veces y calcular el valor promedio del error de entrenamiento E_{in}. Valorar el resultado.

Lo primero es implementar la función.

```
f<-function(x1,x2){
  sign(x1*x1+x2*x2-0.6)
}
```

A continuación generamos los datos en el rango indicado.

```
promedio_E_in<-0
rango<-c(-10,10)
for(i in 1:1000){
  muestra<-matrix(unlist(simula_unif(2,1000,c(rango[1],rango[2]))),ncol=2,byrow=FALSE);
  #vars<-simula_recta(c(rango[1],rango[2]))
  signos <- f(muestra[,1],muestra[,2])
  index_aleat<-sample(1:nrow(muestra),0.1*nrow(muestra))
  signos[index_aleat]<--signos[index_aleat]

  #plot(muestra[,1],muestra[,2])
  #points(muestra[,1][signos>0],muestra[,2][signos>0],col="green")
  #points(muestra[,1][signos<0],muestra[,2][signos<0],col="red")
  #abline(vars[6],vars[5])

  resultados<-Regress_Lin(muestra,signos,TRUE)

  #abline((-resultados[1]/resultados[2]),(-resultados[3]/resultados[2]),col="cyan")

  signos_est<-matrix(sign(muestra[,2] - (-resultados[3]/resultados[2])*muestra[,1] - (-resultados[1]/res
```

```
  promedio_E_in<-promedio_E_in+length(signos[signos!=signos_est])/length(signos)
}

promedio_E_in<-promedio_E_in/1000
print(paste("Promedio de error en la muestra: ",promedio_E_in))
```

```
## [1] "Promedio de error en la muestra:  0.49514"
```

Con regresión lineal nos equivocamos en torno al 50% de los puntos totales. Hay que tener en cuenta que el ruido que hemos añadido es del 10% y que se está equivocando en un 40% más. Se equivoca porque los puntos no son separables, además casi todos tienen etiqueta positiva, por lo que la recta estimada casi siempre se va a quedar por debajo de la muestra para minimizar el error.

b) Ahora, consideremos N=1000 datos de entrenamiento y el siguiente vector de variables: $(1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$. Ajustar de nuevo regresión lineal y calcular el nuevo vector de pesos $\hat{w}$. Mostrar el resultado.

```
muestra<-matrix(unlist(simula_unif(2,1000,c(rango[1],rango[2]))),ncol=2,byrow=FALSE)
muestra<-cbind(1,muestra)
muestra<-cbind(muestra,muestra[,1]*muestra[,2])
muestra<-cbind(muestra,muestra[,1]*muestra[,1])
muestra<-cbind(muestra,muestra[,2]*muestra[,2])


signos <- matrix(f(muestra[,2],muestra[,3]),ncol=1)
index_aleat<-sample(1:nrow(muestra),0.1*nrow(muestra))
signos[index_aleat]<--signos[index_aleat]

resultados<-Regress_Lin(muestra,signos,FALSE)

signos_est<-rep(1,length(signos))

signos_est[
  sign(resultados[1]+resultados[2]*muestra[,2]+resultados[3]*muestra[,3]+
  resultados[4]*muestra[,4]+resultados[5]*muestra[,5]+resultados[6]*muestra[,6])
  <
  f(muestra[,2],muestra[,3])] <- -1


print("Vector de pesos: ")
```

```
## [1] "Vector de pesos: "
```

```
print(matrix(resultados))
```

```
##               [,1]
## [1,]  0.6424368103
## [2,] -0.1432257483
## [3,]  0.0026248177
## [4,]  0.1677225895
## [5,]  0.1727320267
## [6,]  0.0001445455
```

c)Repetir el experimento anterior 1000 veces calculando en cada ocasión el error fuera de la muestra. Para ello generar en cada ejecución 1000 puntos nuevos y valorar sobre ellos la función ajustada. Promediar los valores obtenidos. ¿Qué valor obtiene? Valorar el resultado.

```
promedio_E_out<-0
for(i in 1:1000){
  muestra<-matrix(unlist(simula_unif(2,1000,c(rango[1],rango[2]))),ncol=2,byrow=FALSE)
  muestra<-cbind(1,muestra)
  muestra<-cbind(muestra,muestra[,1]*muestra[,2])
  muestra<-cbind(muestra,muestra[,1]*muestra[,1])
  muestra<-cbind(muestra,muestra[,2]*muestra[,2])

  signos <- matrix(f(muestra[,2],muestra[,3]),ncol=1)
  index_aleat<-sample(1:nrow(muestra),0.1*nrow(muestra))
  signos[index_aleat]<--signos[index_aleat]

  resultados<-Regress_Lin(muestra,signos,FALSE)


  muestra_test<-matrix(unlist(simula_unif(2,1000,c(rango[1],rango[2]))),ncol=2,byrow=FALSE)
  muestra_test<-cbind(1,muestra_test)
  muestra_test<-cbind(muestra_test,muestra_test[,1]*muestra_test[,2])
  muestra_test<-cbind(muestra_test,muestra_test[,1]*muestra_test[,1])
  muestra_test<-cbind(muestra_test,muestra_test[,2]*muestra_test[,2])

  signos_test <- matrix(f(muestra_test[,2],muestra_test[,3]),ncol=1)
  index_aleat_test<-sample(1:nrow(muestra_test),0.1*nrow(muestra_test))
  signos_test[index_aleat_test]<--signos_test[index_aleat_test]


  signos_est<-rep(1,length(signos))

  signos_est[sign(resultados[1]+resultados[2]*muestra_test[,2]+resultados[3]*muestra_test[,3]+
  resultados[4]*muestra_test[,4]+resultados[5]*muestra_test[,5]+resultados[6]*muestra_test[,6])
  <f(muestra_test[,2],muestra_test[,3])]<--1


  promedio_E_out<-promedio_E_out+(length(signos_test[signos_test!=signos_est])/length(signos_test))


}

promedio_E_out<-promedio_E_out/1000
print(paste("Promedio de error fuera de la muestra: ",promedio_E_out))
```

```
## [1] "Promedio de error fuera de la muestra:  0.122327999999999"
```

Se obtiene al rededor de un 13% (130 puntos de equivocación en test), lo cual es un buen resultado. La complejidad del ajuste ha permitido reducir el error fuera de la muestra gracias a las posibles oscilaciones de un modelo no lineal, algo que, evidentemente no permite el modelo lineal.