Análisis y Visualización de Datos con R

Orientado al Aprendizaje Automático

Rafael Nogales

First printing, February 2016

Índice general

1. Práctica 1

1.1 Eliminar valores perdidos

En muchas ocasiones cuanto trabajamos con un vector, una lista o un data.frame nos encontramos con que hay valores NA *not available* o NaN *not a number*.
Para poder trabajar cómodamente lo ideal es eliminar esos valores, vamos a ver dos formas de hacer esto:
La primera es utilizar la función de R **is.na()**

Sintaxis 1 — is.na(). Devuelve un vector de TRUE/FALSE del mismo tamaño que el argumento, la posición *i* es TRUE si x[i] es NA o NaN.

Ejemplo:

```
> x <- c(2, NA, NaN, 123)
> x
[1]   2  NA NaN 123
> x <- x[!is.na(x)]
> x
[1]   2 123
```

La otra forma es utilizando la función **complete.cases()** que hace lo mismo pero puede aplicarse a data.frames
Ejemplo:

```
> x <- c(2, NA, NaN, 123)
> x
[1]   2  NA NaN 123
> x <- x[complete.cases(x)]
> x
[1]   2 123
```

```
> x <- c(2, NA, NaN, 123)
> complete.cases(x)
[1]  TRUE FALSE FALSE  TRUE
> !is.na(x)
[1]  TRUE FALSE FALSE  TRUE
```

1.2 Lanzar dos dados

Ejercicio 1 Crea una función denominada *puntuación* que simule tirar dos dados, y devuelva la suma. Los lanzamientos deben ser independientes.
Nota: Utiliza la función **sample**. Deben de poder obteners valores como 2 o 12. ■

Solución
Código de la función puntuación para un dado:

```
> puntuacion <- function(){
    return(sample(6,1) + sample(6,1));
}
```

Observemos la sintaxis de la función sample:

Sintaxis 2 — sample(x, size, replace = FALSE, prob = NULL). Los parámetros que vamos a considerar importantes por el momento son los tres primeros:

- **x** indica el conjunto de donde se van a extraer las muestras un número *n* indica 1...*n*
- **size** indica cuantas muestras vamos a extrar, en nuestro caso solo queremos una de cada dado
- **replace** indica si se pueden repetir muestras, en nuestro caso da igual porque solo vamos a sacar una muestra.

Aquí vemos unas cuantas ejecuciones de la función puntuación tal y como la llevamos:

```
> puntuacion()
[1] 3
> puntuacion()
[1] 12
> puntuacion()
[1] 7
> puntuacion()
[1] 9
```

Nota: Si escribimos "puntuacion.[en] RStudio podemos ver el código de la función puntuación.
Ahora vamos a redefinir la función para que haga *n* lanzamientos y los guarde en un vector. Luego vamos a ejecutarla unas cuantas veces para ver como funciona:

```
> puntuacion <- function(n){
+     vect_punt<-0;
+     for(i in 1:n){
+         vect_punt[i]<-(sample(6,1) + sample(6,1));
```

```
+      }
+      return (vect_punt);
+ }
> puntuacion(12)
 [1]  9  5 10  4  5  3  8 12  7  6 10  7
> x<-puntuacion(12)
> x
 [1]  8 10  4 10 10 11  3 11  9  7  9  3
> sort(x)
 [1]  3  3  4  7  8  9  9 10 10 10 11 11
```

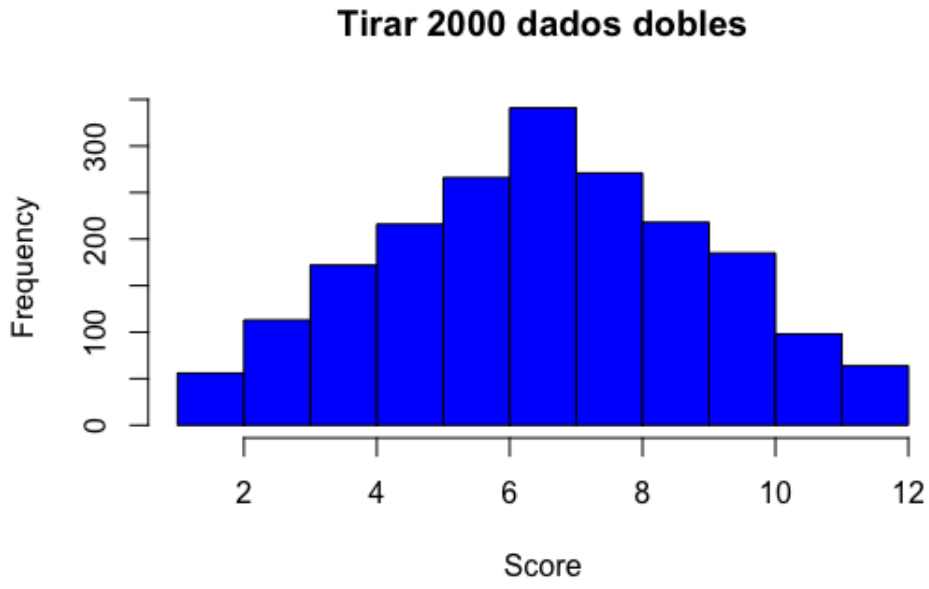
Parece que funciona correctamente, pero nosotros sabemos que en este experimento la media es 7, veamoslo experimentalmente:

```
> mean(puntuacion(10000))
[1] 6.9958
> mean(puntuacion(10000))
[1] 7.0069
```

Podemos además hacer cosas más interesantes como un histograma:

```
> x<-puntuacion(2000)
> hist(x, col="blue",breaks = 1:12,main="Tirar 2000 dados
   dobles", xlab = "Score")
```

Que devuelve la siguiente figura:



1.3 Baraja Española

Ejercicio 2 Vamos a manipular la baraja española, a barajarla y a robar de la baraja. Para ello será necesario crear la baraja: un dataframe y dos funciones *barajar* y *robar*. ■

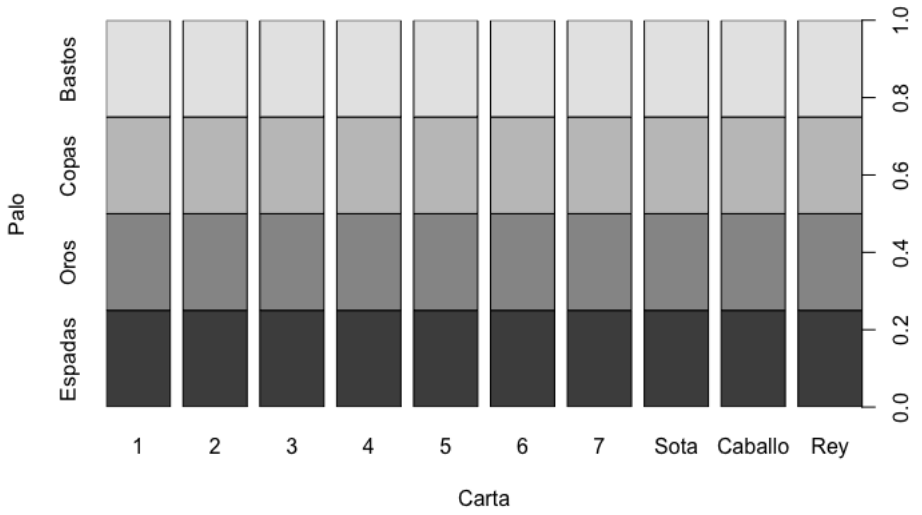
Solución
Para crear la baraja vamos a utilizar la función **expand.grid()** que genera un data.frame con todas las combinaciones posibles de elementos de los vectores que se le meten como parámetros.

```
BarajaESP<-function(){
    figuras <- c(1:7, "Sota", "Caballo", "Rey");
    palos <- c("Espadas", "Oros", "Copas", "Bastos");
    baraja <- expand.grid(Carta = figuras, Palo = palos);
    return(baraja)
}
```

Devuelve el siguiente data.frame:

```
> baraja<-BarajaESP()
> baraja> baraja
     Carta    Palo
1        1 Espadas
2        2 Espadas
3        3 Espadas
4        4 Espadas
5        5 Espadas
6        6 Espadas
7        7 Espadas
8     Sota Espadas
9  Caballo Espadas
10     Rey Espadas
11       1    Oros
12       2    Oros
.................
37       7  Bastos
38    Sota  Bastos
39 Caballo  Bastos
40     Rey  Bastos
}
```

Podemos incluso hacer un plot de baraja y veremos la baraja así:



Para la función de barajar volvemos a utilizar la función **sample**:

```
> barajar <- function(baraja){
    tmp <- baraja;
    orden <- sample(1:40,40,replace = FALSE);
    for(i in orden){
        tmp[i, 1:2] <- baraja[orden[i], 1:2]
    }
    return(tmp)
}
```

Tenemos en cuenta que tmp será un nuevo dataframe que al principio contendrá la baraja en el mismo orden. Pero poco a poco vamos reemplazando las cartas, para ello hacemos una permutación de los primeros 40 naturales y la guardamos en *orden* después aplicamos esa permutación a las cartas.
Es importante darse observar que *baraja[i, 1:2]* es la forma de representar la carta i-esima de la baraja, utilizamos *1:2* para indicar "número y palo".

La función **robar()** es más sencilla y no requiere explicación:

```
> robar <- function(baraja){
    return(baraja[sample(1:40,1, replace = FALSE), 1:2])
}
> robar(baraja)
   Carta Palo
11     1 Oros
> robar(baraja)
   Carta Palo
14     4 Oros
```

Es importante apreciar que esta forma de robar va volviendo a colocar la carta en la baraja, por tanto es posible robar dos veces la misma carta.

2. Práctica 2

2.1 Generación y Visualización de Datos

Ejercicio 3 Construir una función $lista = simula\_unif(N, dim, rango)$ que calcule una lista de longitud N de vectores de dimension dim conteniendo números aleatorios uniformes en el intervalo rango. ■

Parece que lo más natural es crear una función que utilice **runif()** que es una función de R que genera números siguiendo una distribución uniforme y a partir de ello devolver una matriz $N \times dim$

```
simula_unif<-function(N, dim, minimo, maximo){
    unif<-matrix(nrow = dim, ncol = N);
    for(i in 1:N){
        unif[,i] <- runif(dim,minimo, maximo)
    }
    return(unif);
}
```

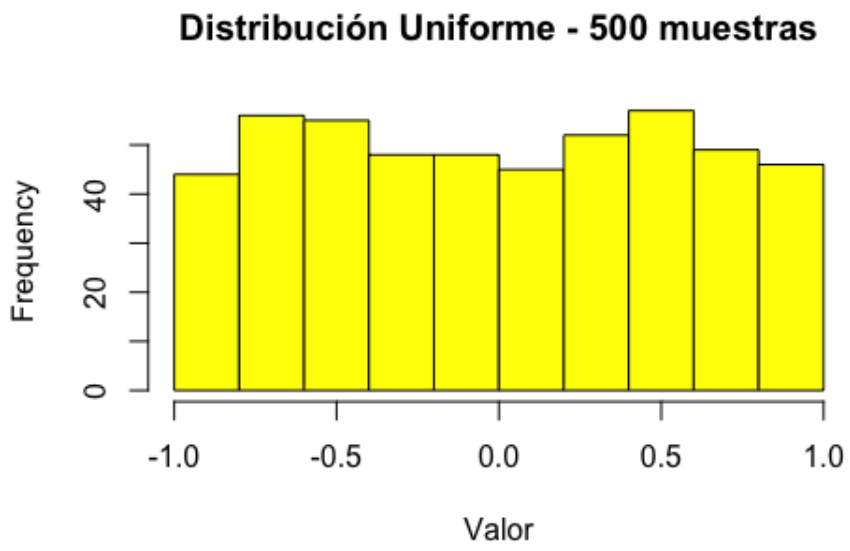
Pero como nos pide que sea una lista hacemos que la variable unif sea una lista (y por tanto accedemos a las componentes con **[[·]]**)

```
> simula_unif<-function(N, dim, minimo, maximo){
    unif<-list();
    for(i in 1:N){
        unif[[i]] <- runif(dim,minimo, maximo)
    }
    return(unif);
}
> lista_uniforme <- simula_unif(N=3,dim=4,-1,1)
> lista_uniforme
```

```
[[1]]
[1]  0.0386862 -0.1860388  0.9824784  0.2079370

[[2]]
[1]  0.888472 -0.454218 -0.932526 -0.543044

[[3]]
[1]  0.79459142  0.60399254 -0.15797747 -0.06230832
```

Podemos ver si funciona correctamente mediante un histograma:

```
> lista_unif<-simula_unif(N=1,dim = 500,-1,1)
> hist(lista_unif[[1]], col = "yellow",main = "Distribucion
   Uniforme - 500 muestras",xlab = "Valor")
```

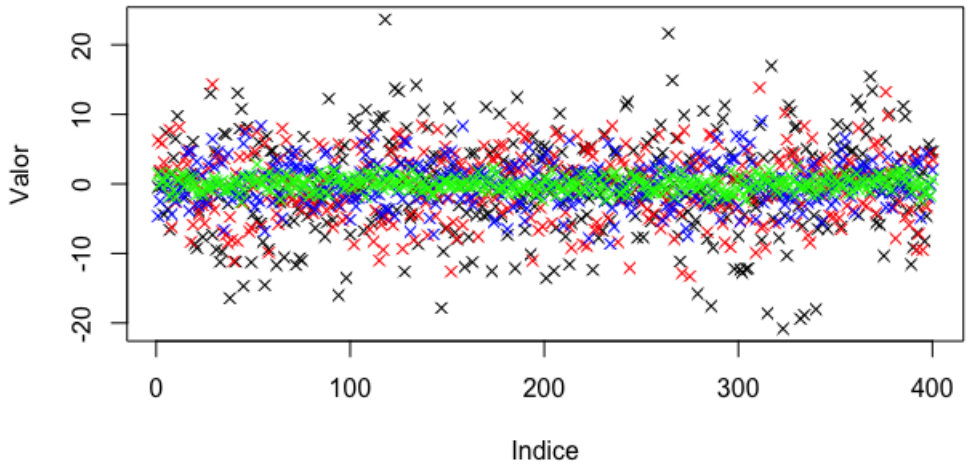


Ejercicio 4 Construir una función $lista = simula\_gaus(N, dim, sigma)$ que calcule una lista de longitud N de vectores de dimension dim conteniendo números aleatorios gaussianos de media 0 y varianzas dadas por el vector *sigma*. ■

```
> simula_gaus <- function(N, dim, sigma){
    normal<-list();
    for(i in 1:N){
        normal[[i]] <- rnorm(n =dim, mean=0,sd = sigma[i]);
    }
    return(normal)
}
> x<-simula_gaus(4,400,c(1,3,5,7))
> plot(x[[4]], col="black", ylab = "Valor", xlab = "Indice",
   pch=4)
```

```
> points(x[[3]], col="red", pch=4)
> points(x[[2]], col="blue", pch=4)
> points(x[[1]], col="green", pch=4)
```

Podemos ver el resultado de las últimas ordenes del cuadro de código anterior en la siguiente figura. Se aprecia claramente como los puntos más dispersos corresponden a las que tienen mayor desviación típica (sigma)

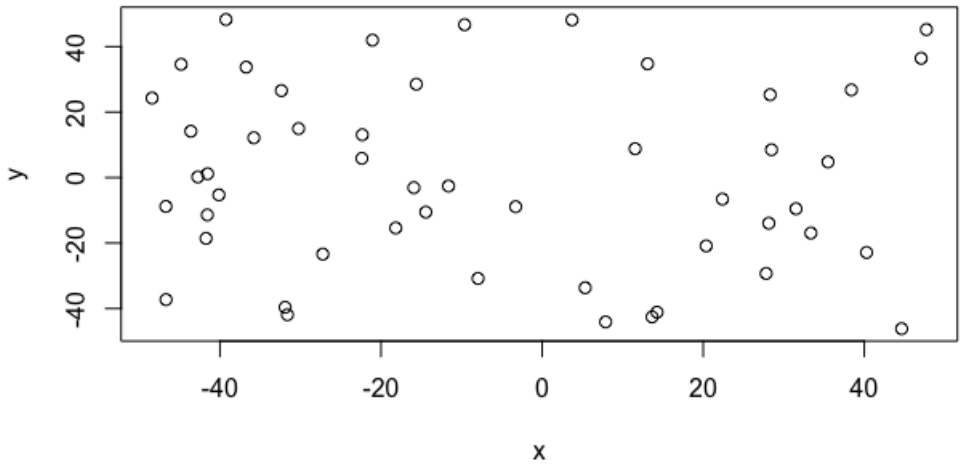


Ejercicio 5 Suponer N = 50, dim=2, rango = [-50, 50] en cada dimensión. Dibujar una gráfica de la salida de la función correspondiente. ■

Creamos una función para que nos haga todo automáticamente:

```
plotUnif<-function(N, minimo, maximo){
    x <- c();
    y <- c();
    puntos <- list();
    puntos <- simula_unif(N, 2, minimo, maximo);
    for(i in 1:N){
        x[i] <- puntos[[i]][[1]];
        y[i] <- puntos[[i]][[2]];
    }
    plot(x, y);
}
```

En la siguiente figura aparece una ejecución de la orden *>plotUnif(50, -50, 50)*:



Observación:
La forma de almacenar las coordenadas x e y de nuestra lista de puntos no es muy apropiada para trabajar en R tal y como lo estamos haciendo. Es decir como una lista con la siguiente estructura: $(x_1, y_1); (x_2, y_2); ...; (x_n, y_n)$
Una forma de almacenar los puntos que simplifica las implementaciones en R es la siguiente:
Lista de puntos = $(x_1, x_2, ..., x_n); (y_1, y_2, ..., y_n)$
En el siguiente ejercicio vamos a reescribir la función *simula_gaus*() para adaptarla a esta estructura y luego vamos a dibujar:

Ejercicio 6 Suponer N = 50, dim=2, sigma = [5, 7] en cada dimensión. Dibujar una gráfica de la salida de la función correspondiente. ■

Función simula_gaus() *(versión 2)*

```
simula_gaus <- function(N, dim, sigma){
    puntos <- list();
    for(i in 1:dim){
        xi <- c();
        xi <- rnorm(N, 0, sigma[i]);
        puntos[[i]]<-xi;
    }
    return(puntos);
}
```

La forma en la que utilizamos la función es la misma que usabamos en la primera versión pero la estructura que devuelve es diferente:

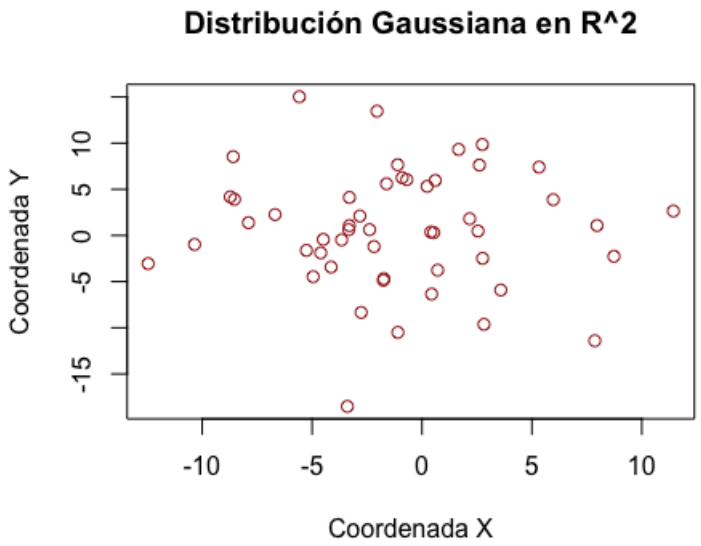
```
> simula_gaus(20, 2, c(5,7))
[[1]]
 [1] -2.4491073 -3.2584142  2.3869703 -9.8788259  0.1670119
   6.2144173  4.5063578  3.1975652
 [9] -2.0464434  1.0344226 -2.0405586 -8.9180882 -1.5210091
   -1.4812887 -3.1102648 -2.3895402
```

```
[17]  -6.4188965   1.5368615   1.7382187  -6.1144230

[[2]]
 [1]   -6.8042244    0.1074266   -3.0641119    4.0550469
   -2.6651593   -1.2383127    1.5766815   -3.0189121
 [9]    5.9075899  -12.1843298    0.9067658   -3.4628036
   -6.5992591   -8.4734840   -3.9468448   -2.1081100
[17]   11.1544907    3.4082678    0.1045350   -5.2854292
```

Vamos a hacer una función para dibujar nubes de puntos en las que la coordenada X sigue una distribución normal y la coordenada Y sigue otra distribución normal diferente:

```
plotGaus2D <- function(N, sigma1, sigma2){
    puntos <- list();
    puntos <- simula_gaus(N, 2, c(sigma1,sigma2))
    plot(puntos[[1]], puntos[[2]],col="brown", pch=1, main = "
   Distribucion Gaussiana en R^2", ylab = "Coordenada Y", xlab
    = "Coordenada X")
}
```

Notamos que esta función solo dibuja en R^2 pero ampliarla a R^n es bastante sencillo.// La gráfica que pide el encunciado puede obtenerse ejecutando: plotGaus2D(50, 5, 7)



Ejercicio 7 Construir la función $v = simula\_recta(intervalo)$ que calcula los parametros $v = (a,b)$ de una recta aleatoria $y = ax+b$, que corte al cuadrado $[-50,50]\times[-50,50]$
(Ayuda: Para calcular la recta simula las coordenadas de dos puntos y usalos para obtener (a,b))
■

```
simula_recta <- function(min, max){
```

```
    x <- c();
    y <- c();
    x <- runif(2, min, max);
    y <- runif(2, min, max);
    a <- (y[2] - y[1]) / (x[2] - x[1]);
    b <- -a*x[1] + y[1];
    v <- c(a, b);
    return(v);
}
```

2.1.1 Modelo lineal

Ejercicio 8 Genera una muestra uniforme usando simula_unif() y etiqueta 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 junto con la recta usada para ello. ■

En primer lugar vamos a modificar la función simula_unif() para que saque una lista con dos elementos (coordenadas x y coordenadas y).

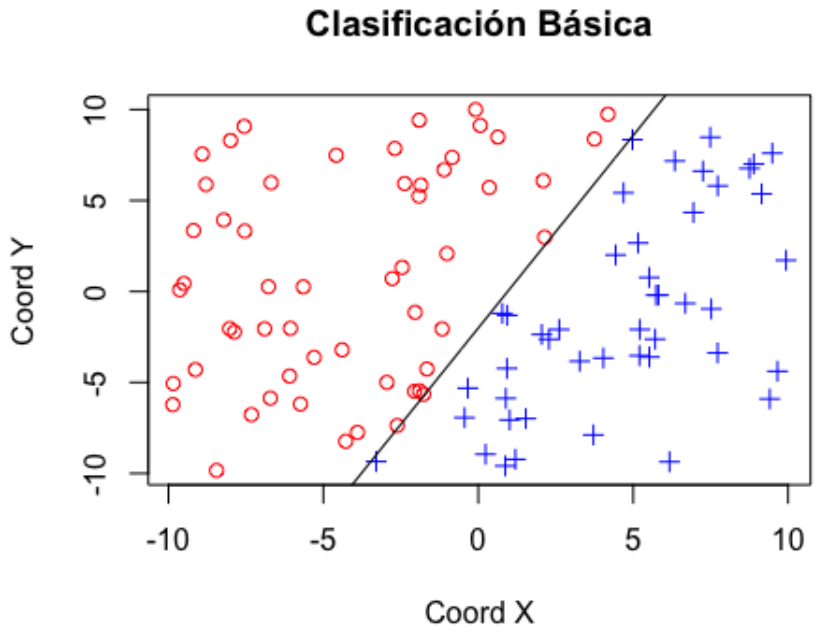
```
> simula_unif <- function(N, dim, minimo, maximo){
    puntos <- list();
    for(i in 1:dim){
        xi <- c();
        xi <- runif(N, minimo, maximo);
        puntos[[i]] <- xi;
    }
    return(puntos)
}
```

Hecho esto ya podemos hacer la función plotEtiquetas() con más facilidad:

```
> plotEtiquetas <- function(N, min, max){
    v <- simula_recta(min, max);
    a <- v[1];
    b <- v[2];
    puntos <- simula_unif(N, dim=2, min, max);
    colores <- c();
    forma <- c();
    for(i in 1:N){
        if(puntos[[2]][[i]] - a*puntos[[1]][[i]] - b > 0){
            colores[i] <- "red";
            forma[i] <- 1;
        }
        else{
            colores[i] <- "blue";
            forma[i] <- 3;
        }
    }
```

```
    plot(puntos[[1]], puntos[[2]], col=colores, pch=forma,
   xlab="Coord X", ylab="Coord Y", main = "Clasificacion
   Basica");
    curve(a*x + b, add=TRUE);
}
```

Podemos ejecutar **plotEtiquetas(100, -10, 10)** con un resultado como este:



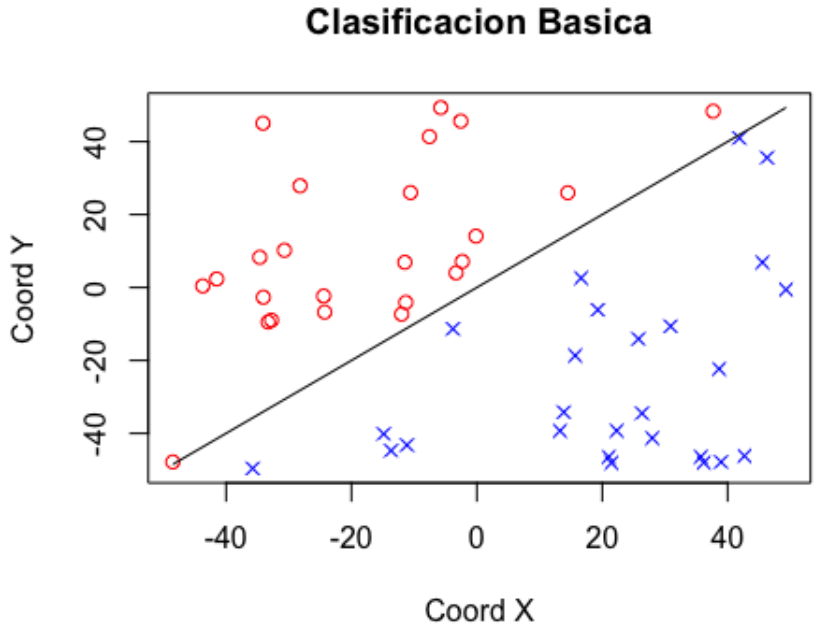
Como ahora queremos clasificar nubes de puntos de diferentes formas vamos a crear una función más general a la cual se le va a pasar una lista de puntos en el formato en el que estamos acostumbrados y los parámetros *a* y *b* de la recta que usaremos para clasificar.
Esto es una solución más general del ejercicio:

```
clasificaPuntosRecta <- function(puntos, a, b){
    xtipo1 <- c();
    ytipo1 <- c();
    xtipo2 <- c();
    ytipo2 <- c();

    N <- length(puntos[[1]]);

    for(i in 1:N){
        if(puntos[[2]][[i]] - a*puntos[[1]][[i]] - b > 0){
            xtipo1 <- c(xtipo1, puntos[[1]][[i]]);
            ytipo1 <- c(ytipo1, puntos[[2]][[i]]);
        }
        else{
            xtipo2 <- c(xtipo2, puntos[[1]][[i]]);
```

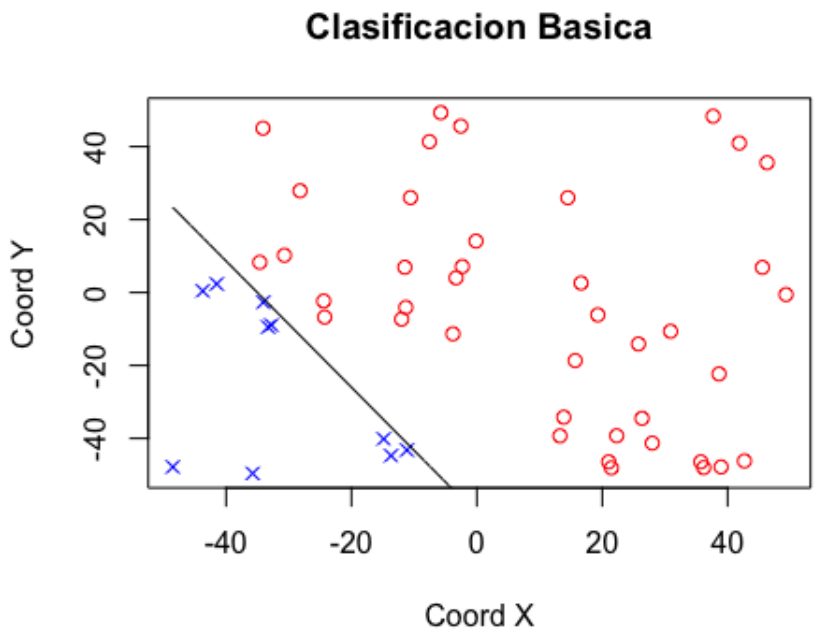
```
            ytipo2 <- c(ytipo2, puntos[[2]][[i]]);
        }
    }

    #Limites de los ejes para el plot:
    miny = min(ytipo1, ytipo2);
    maxy = max(ytipo1, ytipo2);
    minx = min(xtipo1, xtipo2);
    maxx = max(xtipo1, xtipo2);

    plot(xtipo1, ytipo1, col="red", pch=1, xlab="Coord X",
   ylab="Coord Y", main = "Clasificacion Basica", xlim = c(
   minx, maxx), ylim = c(miny, maxy) );
    points(xtipo2, ytipo2, col="blue", pch=4);
    curve(a*x + b, add=TRUE);
    return(list(xtipo1, ytipo1, xtipo2, ytipo2));
}
```

La función **clasificaPuntosRecta(Puntos, a, b)** clasifica los puntos que hay en la lista *Puntos* con dos etiquetas en función del signo de su distancia a la recta $y = ax + b$
Podemos ver un ejemplo con la recta $y = x$ y otro con una recta aleatoria:

```
> misPuntos <- simula_unif(50, 2, -50, 50)
> clasificaPuntosRecta(misPuntos, 1, 0)
```




```
> recta <- simula_recta(-50, 50)
```




```
> clasificaPuntosRecta(misPuntos, recta[1], recta[2])
```

Ejercicio 9 Usar la muestra generada en el apartado anterior y etiquetarla con $+1, -1$ usando el signo de cada una de las siguientes funciones:

- $f_1(x,y) = (x-10)^2 + (y-20)^2 - 400$
- $f_2(x,y) = 0{,}5(x+10)^2 + (y-20)^2 - 400$
- $f_3(x,y) = 0{,}5(x-10)^2 - (y+20)^2 - 400$
- $f_4(x,y) = y - 20x^2 - 5x + 3$

■

2.1.2 Modelo Circular

Observemos en primer lugar que $f_1(x,y) = 0$ representa una circunferencia, luego clasificar los puntos en función del signo de $f_1(x,y)$ es clasificar puntos dependiendo de que estén dentro o fuera de la circunferencia $\{(x,y) \in \mathbb{R}^2 : (x-10)^2 + (y-20)^2 - 400 = 0\}$.
Por eso vamos además a crear una subfuncion en R llamada **draw.circle** para que se vea más claro como está funcionando la clasificación.
El código en R de la clasificación atendiendo al signo de f_1 es el siguiente:

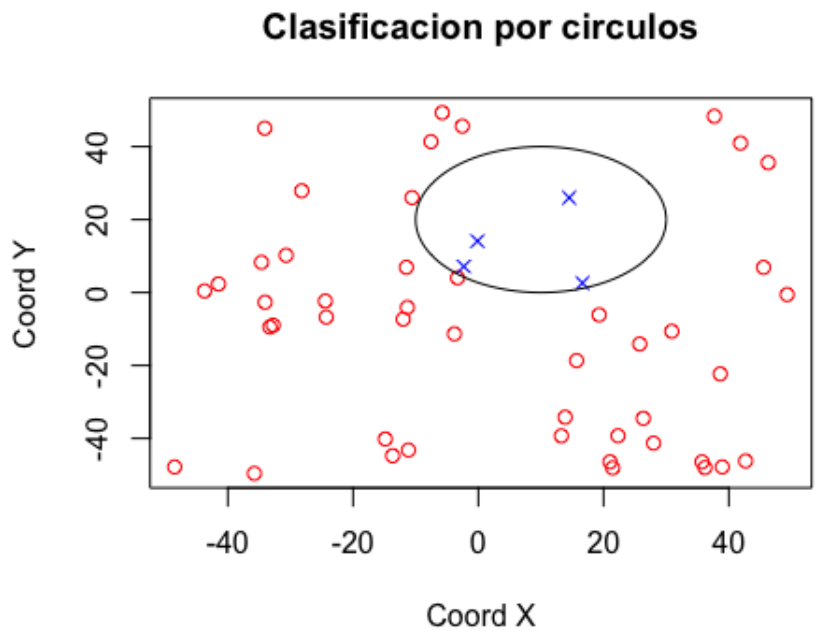
```
clasificaPuntosFuncion1 <- function(puntos){
    xtipo1 <- c();
    ytipo1 <- c();
    xtipo2 <- c();
    ytipo2 <- c();

    N <- length(puntos[[1]]);
```

```
    ### Editar Funcion para cambiar la forma de clasificar
    mifuncion <- function(x,y){
        return((x-10)^2 + (y-20)^2 - 400);
    }

    ### Funcion para dibujar circulos
    draw.circle <- function(x,y, radius){
        theta <- seq(0, 2 * pi, length = 200);
        # draw the circle
        lines(x = radius * cos(theta) + x, y = radius * sin(
   theta) + y);
    }


    for(i in 1:N){
        if(mifuncion(puntos[[1]][[i]],puntos[[2]][[i]]) > 0){
            xtipo1 <- c(xtipo1, puntos[[1]][[i]]);
            ytipo1 <- c(ytipo1, puntos[[2]][[i]]);
        }
        else{
            xtipo2 <- c(xtipo2, puntos[[1]][[i]]);
            ytipo2 <- c(ytipo2, puntos[[2]][[i]]);
        }
    }

    #Limites de los ejes para el plot:
    miny = min(ytipo1, ytipo2);
    maxy = max(ytipo1, ytipo2);
    minx = min(xtipo1, xtipo2);
    maxx = max(xtipo1, xtipo2);

    plot(xtipo1, ytipo1, col="red", pch=1, xlab="Coord X",
   ylab="Coord Y",
         main = "Clasificacion por circulos", xlim = c(minx,
   maxx), ylim = c(miny, maxy) );
    points(xtipo2, ytipo2, col="blue", pch=4);
    draw.circle(10, 20, sqrt(400));
    return(list(xtipo1, ytipo1, xtipo2, ytipo2));
}
```

Para aplicar esta clasificación a los datos que teníamos en el ejercicio anterior ejecutamos:

```
> clasificaPuntosFuncion1(misPuntos)
```

y obtenemos el siguiente gráfico:

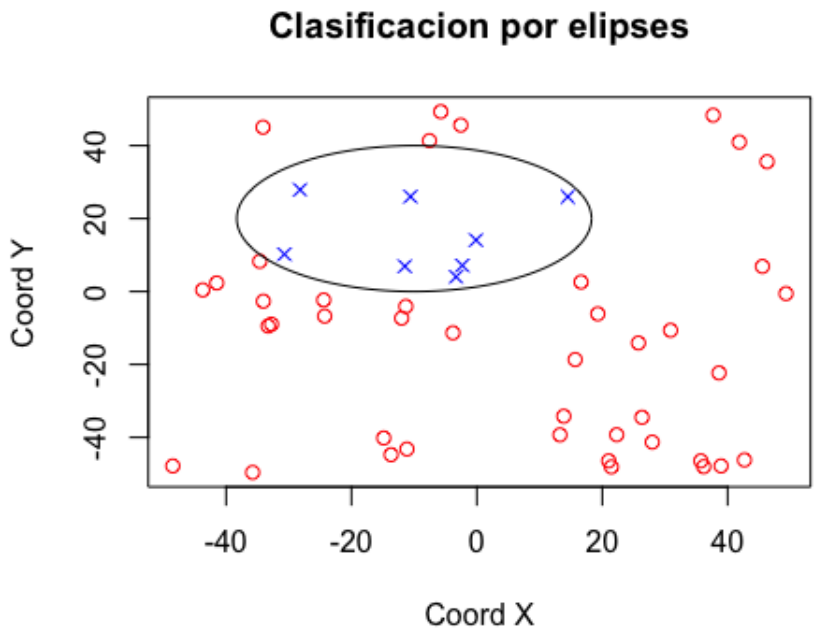


2.1.3 Modelo Elíptico

La clasificación en función de f_2 es bastante parecida a la primera, pero en este caso la figura es una elipse.
Podemos hacer una función draw.elipse(x,y, a,b) donde (x,y) es el centro y (a,b) son los semiejes X e Y de la elipse.

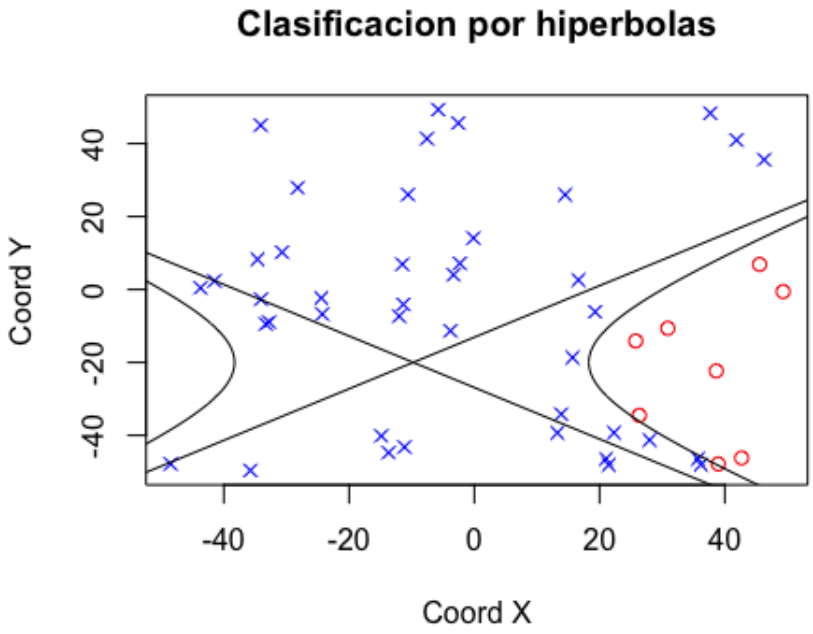
```
> clasificaPuntosFuncion2 <- function(puntos){
    xtipo1 <- c();
    ytipo1 <- c();
    xtipo2 <- c();
    ytipo2 <- c();

    N <- length(puntos[[1]]);

    ### Editar Funcion para cambiar la forma de clasificar
    mifuncion <- function(x,y){
        return(0.5*(x+10)^2 + (y-20)^2 - 400);
    }

    ### Funcion para dibujar elipses
    draw.elipse <- function(x=0,y=0,a=1,b=1){
        theta <- seq(0, 2 * pi, length = 200);
        # draw the circle
        lines(x = a*cos(theta) + x, y = b*sin(theta) + y);
    }


    for(i in 1:N){
        if(mifuncion(puntos[[1]][[i]],puntos[[2]][[i]]) > 0){
```

```
            xtipo1 <- c(xtipo1, puntos[[1]][[i]]);
            ytipo1 <- c(ytipo1, puntos[[2]][[i]]);
        }
        else{
            xtipo2 <- c(xtipo2, puntos[[1]][[i]]);
            ytipo2 <- c(ytipo2, puntos[[2]][[i]]);
        }
    }

    #Limites de los ejes para el plot:
    miny = min(ytipo1, ytipo2);
    maxy = max(ytipo1, ytipo2);
    minx = min(xtipo1, xtipo2);
    maxx = max(xtipo1, xtipo2);

    plot(xtipo1, ytipo1, col="red", pch=1, xlab="Coord X",
   ylab="Coord Y",
         main = "Clasificacion por elipses", xlim = c(minx,
   maxx), ylim = c(miny, maxy) );
    points(xtipo2, ytipo2, col="blue", pch=4);
    draw.elipse(-10, 20, sqrt(400)/sqrt(0.5), sqrt(400));
    return(list(xtipo1, ytipo1, xtipo2, ytipo2));
}
```



2.1.4 Modelo Hiperbólico

```
clasificaPuntosFuncion3 <- function(puntos){
    xtipo1 <- c();
    ytipo1 <- c();
    xtipo2 <- c();
    ytipo2 <- c();

    N <- length(puntos[[1]]);

    ### Editar Funcion para cambiar la forma de clasificar
    mifuncion <- function(x,y){
        return(0.5*(x+10)^2 - (y+20)^2 - 400);
    }

    ### Funcion para dibujar hiperbola
    draw.hiperbola <- function(x=0,y=0,a=1,b=1){
        theta <- seq(0, 2 * pi, length = 200);
        # draw
        lines(x = a*(1/cos(theta)) + x, y =  b*tan(theta) + y)
   ;
    }


    for(i in 1:N){
        if(mifuncion(puntos[[1]][[i]],puntos[[2]][[i]]) > 0){
            xtipo1 <- c(xtipo1, puntos[[1]][[i]]);
            ytipo1 <- c(ytipo1, puntos[[2]][[i]]);
        }
        else{
            xtipo2 <- c(xtipo2, puntos[[1]][[i]]);
            ytipo2 <- c(ytipo2, puntos[[2]][[i]]);
        }
    }

    #Limites de los ejes para el plot:
    miny = min(ytipo1, ytipo2);
    maxy = max(ytipo1, ytipo2);
    minx = min(xtipo1, xtipo2);
    maxx = max(xtipo1, xtipo2);

    plot(xtipo1, ytipo1, col="red", pch=1, xlab="Coord X",
   ylab="Coord Y",
         main = "Clasificacion por hiperbolas", xlim = c(minx,
    maxx), ylim = c(miny, maxy) );
    points(xtipo2, ytipo2, col="blue", pch=4);
    draw.hiperbola(-10, -20, sqrt(400)/sqrt(0.5), sqrt(400));
    return(list(xtipo1, ytipo1, xtipo2, ytipo2));
}
```

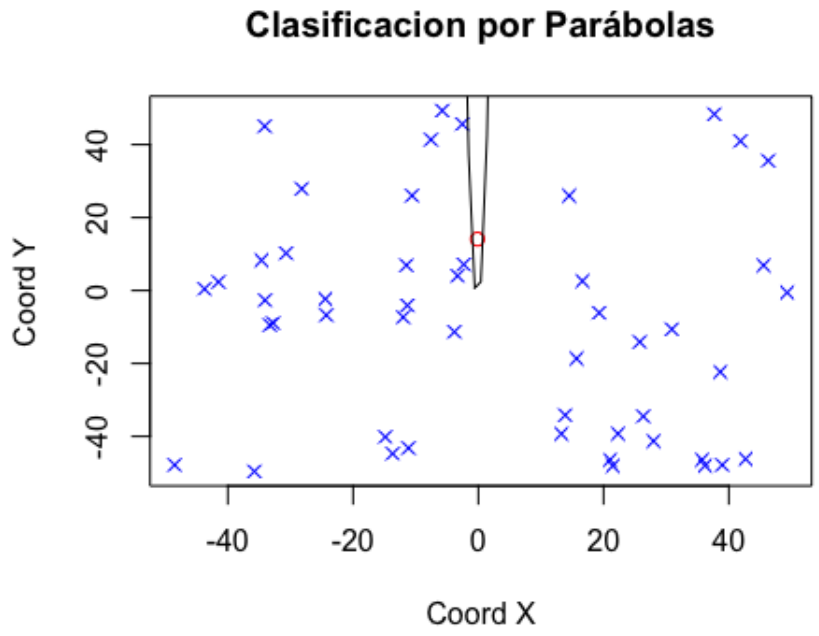


2.1.5 Modelo Parabólico

Por último vamos a hacer una clasificación mediante una parábola:

```
clasificaPuntosFuncion4 <- function(puntos){
    xtipo1 <- c();
    ytipo1 <- c();
    xtipo2 <- c();
    ytipo2 <- c();

    N <- length(puntos[[1]]);

    ### Editar Funcion para cambiar la forma de clasificar
    mifuncion <- function(x,y){
        return(y - 20*x^2 - 5*x+3);
    }

    for(i in 1:N){
        if(mifuncion(puntos[[1]][[i]],puntos[[2]][[i]]) > 0){
            xtipo1 <- c(xtipo1, puntos[[1]][[i]]);
            ytipo1 <- c(ytipo1, puntos[[2]][[i]]);
        }
        else{
            xtipo2 <- c(xtipo2, puntos[[1]][[i]]);
            ytipo2 <- c(ytipo2, puntos[[2]][[i]]);
        }
    }

    #Limites de los ejes para el plot:
    miny = min(ytipo1, ytipo2);
```

```
    maxy = max(ytipo1, ytipo2);
    minx = min(xtipo1, xtipo2);
    maxx = max(xtipo1, xtipo2);

    plot(xtipo1, ytipo1, col="red", pch=1, xlab="Coord X",
   ylab="Coord Y",
         main = "Clasificacion por Parabolas", xlim = c(minx,
   maxx), ylim = c(miny, maxy) );
    points(xtipo2, ytipo2, col="blue", pch=4);
    curve(expr = (y = 20*x^2 + 5*x -3),add = TRUE);
    return(list(xtipo1, ytipo1, xtipo2, ytipo2));
}
```



Interpretación de las formas de las regiones:
La clasificación lineal corresponde a una $f(x,y)$ cuya gráfica es un plano en $\mathbb{R}^3$ y la clasificación que hacemos es rojo"si $(x,y,f(x,y)) \in \mathbb{R}^3$ queda por encima del plano $z=0$ y .ªzul.en caso contrario.
La recta de la frontera de decisión corresponde a la intersección de la gráfica de f y el plano $z=0$.
En general la forma de la frontera de decisión entre las regiones positiva y negativa depende de la función f:
Podemos ver f como un campo de potencial y la frontera de clasificación como la linea de nivel correspondiente al 0. Los puntos que tienen un *"potencial"* positivo se corresponden a la etiqueta roja y los que tienen *"potencial"* negativo son los que corresponden a la etiqueta azul.

2.1.6 Alteración de muestras

Vamos a simular el ruido en una muestra mediante el cambio de algunas etiquetas.
Para ello primero vamos a clasificar una nube de puntos (en este caso lo haremos mediante una recta) y luego vamos a alterar el 10 % de las etiquetas.

Ejercicio 10 Considerar una muestra como la etiquetada en el ejercicio de clasificación mediante un modelo lineal. Modifique las etiquetas de un 10% aleatorio de muestras positivas y otro 10% aleatorio de negativas ■

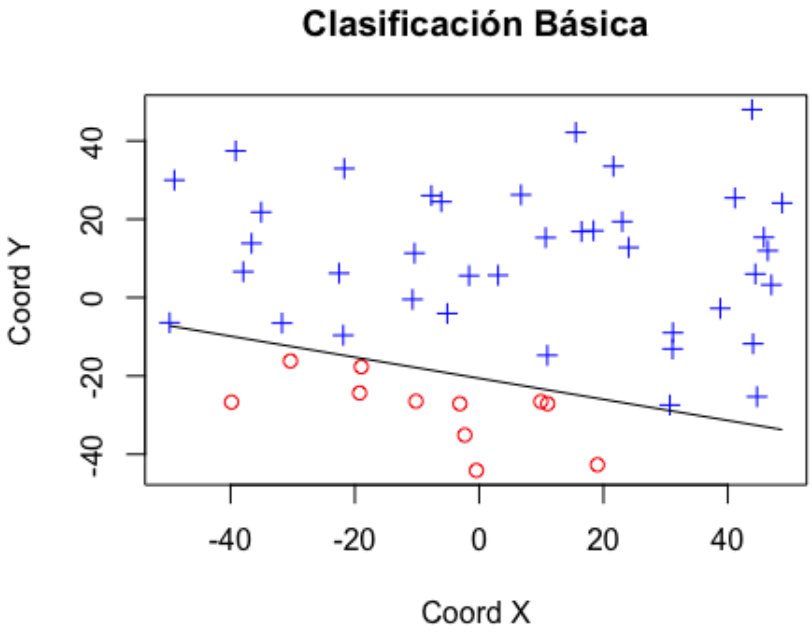
Solución:
Para la alteración de las etiquetas vamos a utilizar la siguiente función a la cual hay que pasarle puntos **ya etiquetados**, como los que devuelven las

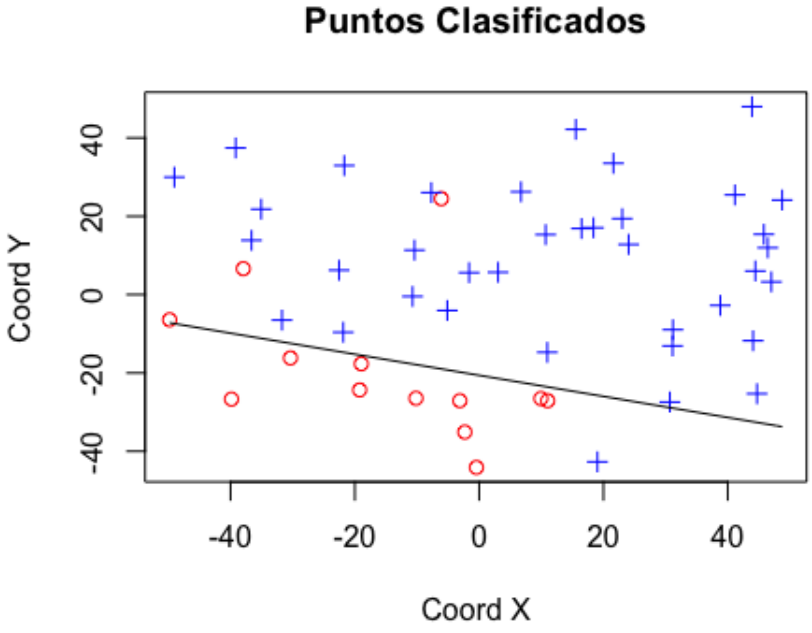
```
> alterarClasificacion <- function(puntosClasificados, prob){
    N <- length(puntosClasificados[[1]]);
    vector_intercambio <- sample(x = c(1,-1),size = N, replace
    = TRUE, prob = c(1-prob, prob))
    puntosClasificados[[3]] = puntosClasificados[[3]]*vector_
   intercambio;
    return(puntosClasificados)
}
```

```
> plotPuntosClasificados <- function(puntos, a, b){
    plot(puntos[[1]], puntos[[2]], col=puntos[[3]]+3, pch=
   puntos[[3]]+2, xlab="Coord X", ylab="Coord Y", main = "
   Clasificacion Basica");
    curve(a*x + b, add=TRUE);
}
```

Ahora vamos a ver la diferencia entre las dos:

```
> misOtrosPuntosClasificadosAlter <- alterarClasificacion(
   misOtrosPuntosClasificados, 0.1)
> plotPuntosClasificados(misOtrosPuntosClasificadosAlter, v
   [1], v[2])
```


Clasificación Básica
Coord Y
40
20
0
-20
-40
-40
-20
0
20
40
Coord X


Puntos Clasificados
Coord Y
40
20
0
-20
-40
-40
-20
0
20
40
Coord X

2.1.7 Aceleración de R

La función que hemos creado utiliza un *bucle for* por dentro, esto podría ser una buena solución en lenguajes como C++ en el que los bucles son muy eficientes, pero en el caso de R el bucle for es una estructura muy lenta.
De hecho es trabajar con bucles for es unas 50 veces más lento que trabajar con vectores aprovechando su estructura vectorial:

```
> sumaF <- function(f, a,k){
    suma=0;
    for(i in a:k){
    suma = suma+f(i);
  }
  return(suma);
}
# Version vectorizada
> sumaV <- function(f, a,k){
  i = a:k;
    sum(f(i))
}
> # install.packages("rbenchmark") # descomentar para instalar
    el paquete "rbenchmark"
> library(rbenchmark)
> k = 10000000 #k = 10 millones
> f = function(n) 1/n^2
> benchmark(sumaF(f, 1, k),sumaV(f, 1, k),replications = 5)[,c
   (1,3,4)]
           test elapsed relative
1 sumaF(f, 1, k)  48.434   53.224
2 sumaV(f, 1, k)   0.910    1.000
```

Por esto mismo sería conveniente modificar la estructura interna de nuestras funciones en caso de trabajar con conjuntos de puntos muy grandes.
Una posible modificación para el clasificador mediante una recta es la siguiente:

```
> clasificaPuntosRectav2 <- function(puntos, a, b){
        # tipo[i] = 1 o -1 ya que TRUE es 1 y FALSE es 0
        # La funcion f(x) = 2x-1 lleva el 0 al -1 y el 1 al 1
        # Por tanto tipo es un vector de -1 y 1
        tipo <- 2*(puntos[[2]] - a*puntos[[1]] - b > 0) - 1;

        #Limites de los ejes para el plot:
        miny = min(puntos[[2]]);
        maxy = max(puntos[[2]]);
        minx = min(puntos[[1]]);
        maxx = max(puntos[[1]]);

        plot(puntos[[1]], puntos[[2]], col=tipo+3, pch=1, xlab
   ="Coord X", ylab="Coord Y", main = "Clasificacion Basica",
   xlim = c(minx, maxx), ylim = c(miny, maxy) );
```

```
        curve(a*x + b, add=TRUE);
        return(list(puntos[[1]], puntos[[2]], tipo));
    }
```

Como vemos en el siguiente fragmento de código el rendimiento mejora sustanciosamente: Para una muestra de 100.000 puntos la primera versión (usando un bucle for) tarda 147 segundos en hacer la clasificación. Por el contrario la segunda versión (usando la estructura vectorial del problema) tarda tan solo 8.4 segundos en hacer el mismo trabajo. **17 veces más rápido!!**

```
> misPuntos <- simula_unif(100000, 2, -100,100)
> v <- simula_recta(-100, 100);
> benchmark(clasificaPuntosRecta(misPuntos,v[1],v[2]),
   clasificaPuntosRectav2(misPuntos,v[1],v[2]),replications =
   3)
                                              test replications
   elapsed relative user.self sys.self
1   clasificaPuntosRecta(misPuntos, v[1], v[2])            3
   147.163    17.36   103.816   42.611
2 clasificaPuntosRectav2(misPuntos, v[1], v[2])            3
    8.477     1.00     8.377    0.065
```

Una solución mejorada para los otros modelos es la siguiente:
Modelo Circular:

```
clasificaPuntosFuncion1v2 <- function(puntos){
    ### Editar Funcion para cambiar la forma de clasificar
    mifuncion <- function(x,y){
        return((x-10)^2 + (y-20)^2 - 400);
    }
    ### Funcion auxiliar para dibujar circulos
    draw.circle <- function(x,y, radius){
        theta <- seq(0, 2 * pi, length = 200);
        # draw the circle
        lines(x = radius * cos(theta) + x, y = radius * sin(
   theta) + y);
    }

    #Vector de clasificacion:
    tipo <- 2*(mifuncion(puntos[[1]], puntos[[2]]) > 0) - 1

    #Limites de los ejes para el plot:
    miny = min(puntos[[2]]);
    maxy = max(puntos[[2]]);
    minx = min(puntos[[1]]);
    maxx = max(puntos[[1]]);

    plot(puntos[[1]], puntos[[2]], col=tipo+3, pch=1, xlab="
   Coord X", ylab="Coord Y", main = "Clasificacion por
   Circulos", xlim = c(minx, maxx), ylim = c(miny, maxy) );
```

```
    draw.circle(10, 20, sqrt(400));
    return(list(puntos[[1]], puntos[[2]], tipo));
}
```

Modelo Eliptico:

```
clasificaPuntosFuncion2v2 <- function(puntos){
    ### Editar Funcion para cambiar la forma de clasificar
    mifuncion <- function(x,y){
        return(0.5*(x+10)^2 + (y-20)^2 - 400);
    }

    ### Funcion para dibujar elipses
    draw.elipse <- function(x=0,y=0,a=1,b=1){
        theta <- seq(0, 2 * pi, length = 200);
        # draw the circle
        lines(x = a*cos(theta) + x, y = b*sin(theta) + y);
    }

    #Vector de clasificacion:
    tipo <- 2*(mifuncion(puntos[[1]], puntos[[2]]) > 0) - 1

    #Limites de los ejes para el plot:
    miny = min(puntos[[2]]);
    maxy = max(puntos[[2]]);
    minx = min(puntos[[1]]);
    maxx = max(puntos[[1]]);

    plot(puntos[[1]], puntos[[2]], col=tipo+3, pch=tipo+4,
   xlab="Coord X", ylab="Coord Y", main = "Clasificacion por
   Elipses", xlim = c(minx, maxx), ylim = c(miny, maxy) );
    draw.elipse(-10, 20, sqrt(400)/sqrt(0.5), sqrt(400));
    return(list(puntos[[1]], puntos[[2]], tipo));
}
```

Modelo Hiperbólico:

```
> clasificaPuntosFuncion4v2 <- function(puntos){
    ### Editar Funcion para cambiar la forma de clasificar
    mifuncion <- function(x,y){
        return(0.5*(x+10)^2 - (y+20)^2 - 400);
    }

    ### Funcion para dibujar hiperbola
    draw.hiperbola <- function(x=0,y=0,a=1,b=1){
        theta <- seq(0, 2 * pi, length = 200);
        # draw
        lines(x = a*(1/cos(theta)) + x, y =  b*tan(theta) + y)
   ;
```

```
    }

    #Vector de clasificacion:
    tipo <- 2*(mifuncion(puntos[[1]], puntos[[2]]) > 0) - 1
    print(tipo);
    #Limites de los ejes para el plot:
    miny = min(puntos[[2]]);
    maxy = max(puntos[[2]]);
    minx = min(puntos[[1]]);
    maxx = max(puntos[[1]]);

    plot(puntos[[1]], puntos[[2]], col=tipo+3, pch=tipo+4,
   xlab="Coord X", ylab="Coord Y", main = "Clasificacion por
   Hiperbolas", xlim = c(minx, maxx), ylim = c(miny, maxy) );
    draw.hiperbola(-10, -20, sqrt(400)/sqrt(0.5), sqrt(400));
    return(list(puntos[[1]], puntos[[2]], tipo));
}
```

Modelo Parabólico:

```
> clasificaPuntosFuncion3v2 <- function(puntos){
      ### Editar Funcion para cambiar la forma de clasificar
      mifuncion <- function(x,y){
          return(y - 20*x^2 - 5*x+3);
      }

      #Vector de clasificacion:
      tipo <- 2*(mifuncion(puntos[[1]], puntos[[2]]) > 0) - 1

      #Limites de los ejes para el plot:
      miny = min(puntos[[2]]);
      maxy = max(puntos[[2]]);
      minx = min(puntos[[1]]);
      maxx = max(puntos[[1]]);

    plot(puntos[[1]], puntos[[2]], col=tipo+3, pch=tipo+4,
   xlab="Coord X", ylab="Coord Y", main = "Clasificacion por
   Parabolas", xlim = c(minx, maxx), ylim = c(miny, maxy) );
    curve(expr = (y = 20*x^2 + 5*x -3),add = TRUE);
    return(list(puntos[[1]], puntos[[2]], tipo));
}
```

Si en el problema no se puede aprovechar la vectorización hay otra alternativa: El paquete **Rcpp** (Rc++) permite meter código en C++ dentro de funciones R, con lo que podemos optimizar los bucles internos de nuestros programas para hacerlos capaces de procesar grandes cantidades de datos en un tiempo razonable.

2.2 Introducción a las Redes Neuronales - El Perceptrón Simple

El perceptrón simple según muchos autores es el tipo de *Red Neuronal* más simple que exisite. Ya que es una red neuronal compuesta por una sola neurona. Para entender el perceptrón no es necesario tener ningún conocimiento previo:
Nuestro objetivo es distinguir entre dos tipos de elementos, el primer tipo de elementos lo vamos a etiquetar con 1 y el segundo tipo con -1 pero podríamos verlos como rojo y azul o cualquier otra característica.
A nosotros se nos da una muestra de elementos etiquetados, nosotros le damos esos datos al perceptrón y él los clasifica mediante un hiperplano. Es como lo que hemos hecho en las secciones previas al clasificar mediante una recta.
Hemos de observar que el perceptrón sólo funcionará cuando nuestra muestra de elementos pueda **separarse linealmente**, es decir, cuando exista un hiperplano que deje las muestras etiquetadas con 1 a un lado y al otro lado deje las etiquetadas con -1.

2.2.1 El algoritmo PLA (*Perceptron Learning Algorithm*)

Ejercicio 11 Implementar la función **sol = ajusta_PLA(datos, label, max_iter, v_ini)** 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 vectorde etiquetas (cada etiqueta es un valor $+1$ o -1), *max_iter* es el número máximo de iteraciones permitidas y *v_ini* el valor inicial del vector de pesos.
La salida *sol* devuelve los coeficientes del hiperplano. ■

Solución:

```
> ajusta_PLA <- function(datos, label, max_iter, v_ini){
    w <- v_ini;
    w[ncol(datos)]=0;
    N <- nrow(datos);
    dim <- ncol(datos)-1;
    salir = FALSE;
    for(h in 1:max_iter){
        salir <- TRUE;
        for(i in 1:N){
            x <- c(datos[i,1:dim],1);
            if( sign(w %*% x) != label[i]){
                w <- w + label[i]*x
                salir <- FALSE;
            }
        }
        if(salir == TRUE){
            break
        }
    }
    coefs <- c(-w[1]/w[2], -w[3]/w[2])
    return(coefs)
}
```

Vamos a probarlo con la muestra que tenemos ya clasificada y probemos a cambiar *v_ini* para ver cuantas rondas necesita para converger.

Ejercicio 12 Ejecutar el algoritmo PLA con los valores simulados en el apartado de clasificación por rectas 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 para converger. Valorar el resultado. ■

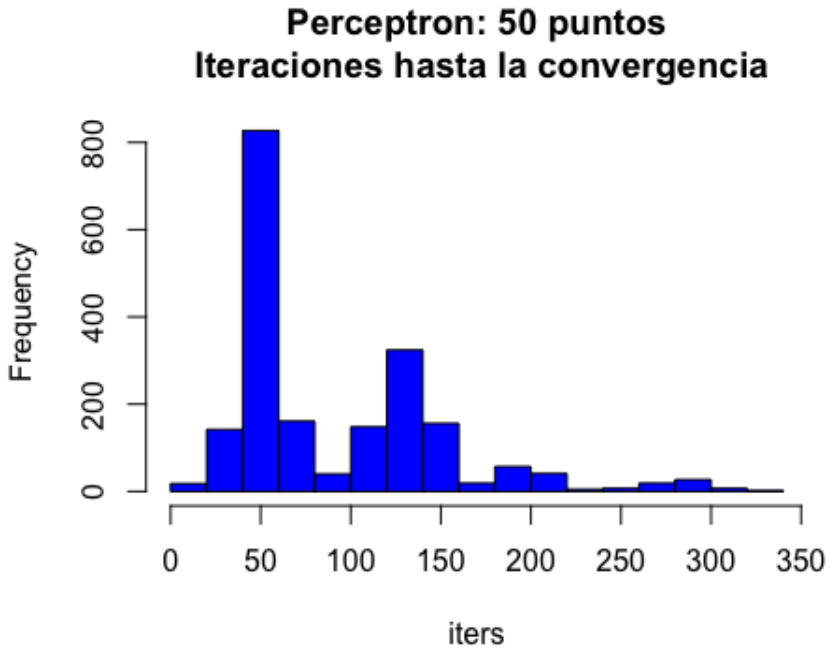
Solucion: Vamos a hacer en primer lugar una función que haga analisis del número de iteraciones:

```
> analisis_PLA <- function(datos, label, num_pruebas){


    iteraciones <- c();
    N <- nrow(datos);
    dim <- ncol(datos)-1;
    max_iter <- 20000

    for(its in 1:num_pruebas){
        if(its == 1){
            w <- c(0,0,0)
        }
        else{
            w <- runif(n = 3, min = 0, max = 1);
        }
        salir = FALSE;
        for(h in 1:max_iter){
            salir <- TRUE;
            for(i in 1:N){
                x <- c(datos[i,1:dim],1);
                if( sign(w %*% x) != label[i]){
                    w <- w + label[i]*x
                    salir <- FALSE;
                }
            }
            if(salir == TRUE){
                iteraciones[its] <- h;
                print(iteraciones[its])
                break
            }
        }
    }
    print(c("Media",mean(iteraciones)));
    coefs <- c(-w[1]/w[2], -w[3]/w[2])
    return(iteraciones)
}
```

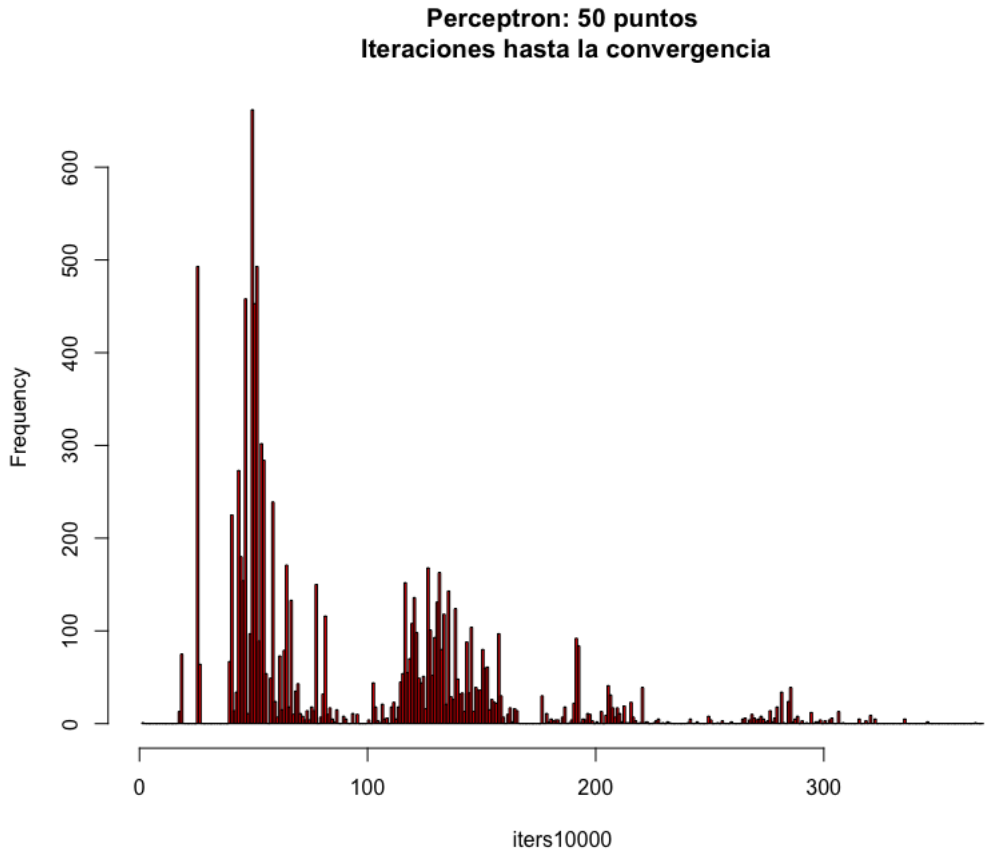
Ponemos un tope de 2000 iteraciones como máximo, aunque en la práctica este limite no se va a tocar.
La función ejecuta el PLA con distintos valores iniciales de w, veamos lo que ocurre con 10 pruebas (lo que nos pide el enunciado):

```
> iters <- analisis_PLA(misDatos, misEtiquetas, 10)
[1] 46
[1] 62
[1] 139
[1] 118
[1] 46
[1] 103
[1] 117
[1] 117
[1] 127
[1] 50
[1] "Media" "92.5"
```

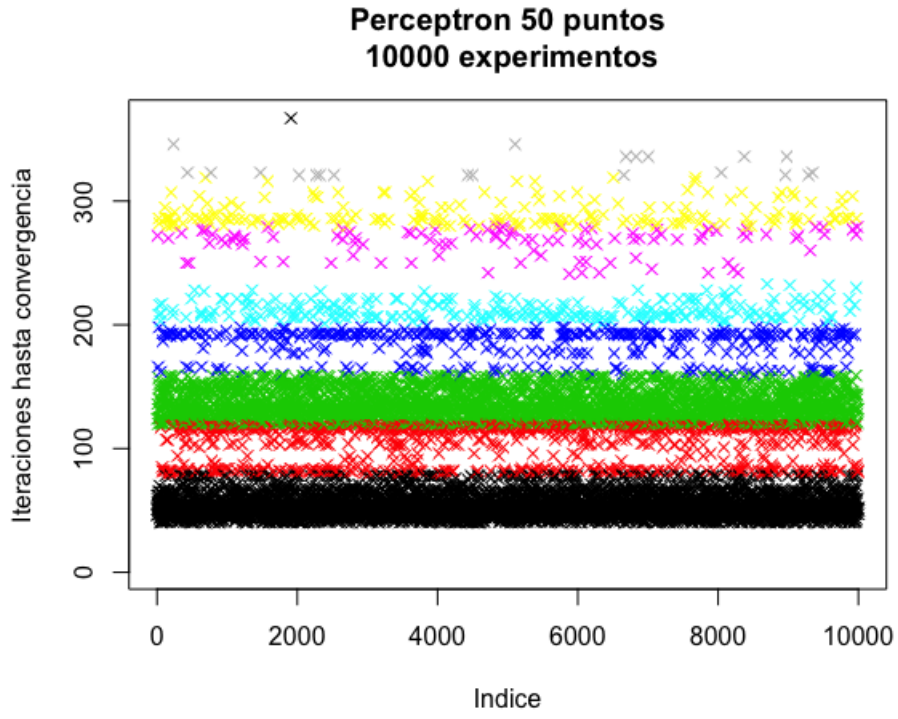
Vamos ahora a probar con 2000 pruebas, ya que 10 dan poca información sobre lo que ocurre en media. Veamos además de la media otros datos como la dispersión mediante un histograma y una gráfica.



Vemos que la distribución tiene puntos de acumulación. Para verlo mejor vamos a hacerlo con 10.000 pruebas:

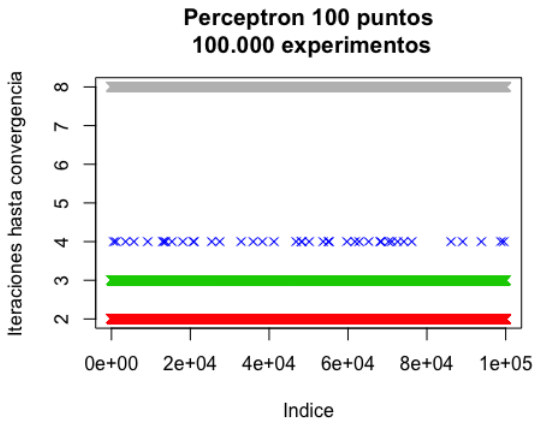


Veamos ahora una nube de puntos donde cada punto es una prueba y su altura indica el número de iteraciones que se han necesitado:
Para resaltar las franjas vacías vamos a resaltar con colores:

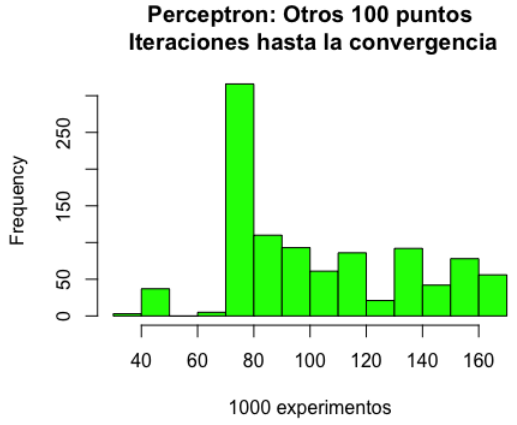


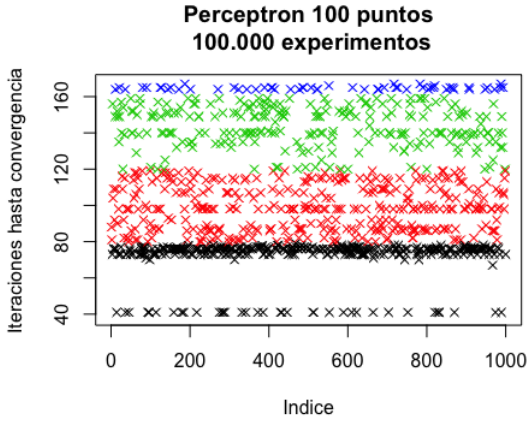
La media de iteraciones es $\bar{x} = 93{,}8279$ para 10.000 experimentos, pero como vemos, en este caso la media no es nada significativa, pues las zonas que más probabilidad concentra es: $[20,80] \cup [100,160]$.
La mediana, en este caso es una medida más significativa en este caso (la mediana es 65).

Probando con otros ejemplos vemos que tambien muestran bandas, en el siguiente caso se muestra la velocidad de convergencia del PLA para 100 puntos uniformemente distribuidos en el rectangulo $[-50,50] \times [-50,50]$ clasificados mediante la recta $y = x$
Vemos que tambien aparecen tímidamente algunos casos en los que el perceptrón necesita 4 iteraciones para converger:



¿Por qué ocurre esto? Parece que hay una especie de *valores propios* asociados a cada recta clasificadora. Ya que si probamos con 100 datos en el mismo rectangulo pero esta vez clasificados mediante la recta $y = 0{,}3x + 5$





Ahora vamos a analizar como funciona el perceptrón cuando los datos **no son linealmente separables**.
Para ello vamos a usar la muestra de puntos alterados que obtuvimos en la seccion *Alteración de muestras*.
Además vamos a modificar el código del algoritmo PLA para que nos permita dibujar lo que se ha ajustado:

```
ajusta_PLA <- function(datos, label, max_iter, v_ini, dibujar
   = FALSE){
    w <- v_ini;
    w[ncol(datos)]=0;
    N <- nrow(datos);
    dim <- ncol(datos)-1;
    salir = FALSE;
    for(h in 1:max_iter){
        salir <- TRUE;
        for(i in 1:N){
            x <- c(datos[i,1:dim],1);
            if( sign(w %*% x) != label[i]){
                w <- w + label[i]*x
                salir <- FALSE;
            }
        }
        if(salir == TRUE){
            break
        }
    }
    coefs <- c(coef1 = -w[1]/w[2], coef2 = -w[3]/w[2])

    if (dibujar == TRUE){
        #Limites de los ejes para el plot:
        miny = min(datos[,2]);
        maxy = max(datos[,2]);
        minx = min(datos[,1]);
        maxx = max(datos[,1]);
        plot(x = datos[,1], y= datos[,2], col = label+3, xlim
   = c(minx, maxx), ylim = c(miny, maxy), main = c("Perceptron
   ", h, "iteraciones"), xlab = "coord X", ylab = "coord Y" )
        curve(coefs[1]*x + coefs[2], add = TRUE)
    }
    return(coefs)
}
```

Ejercicio 13 Ejecuta el algoritmo PLA con los datos generados en la seccion de *Alteración de muestras* 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. Valorar el resultado. ■

Solución:

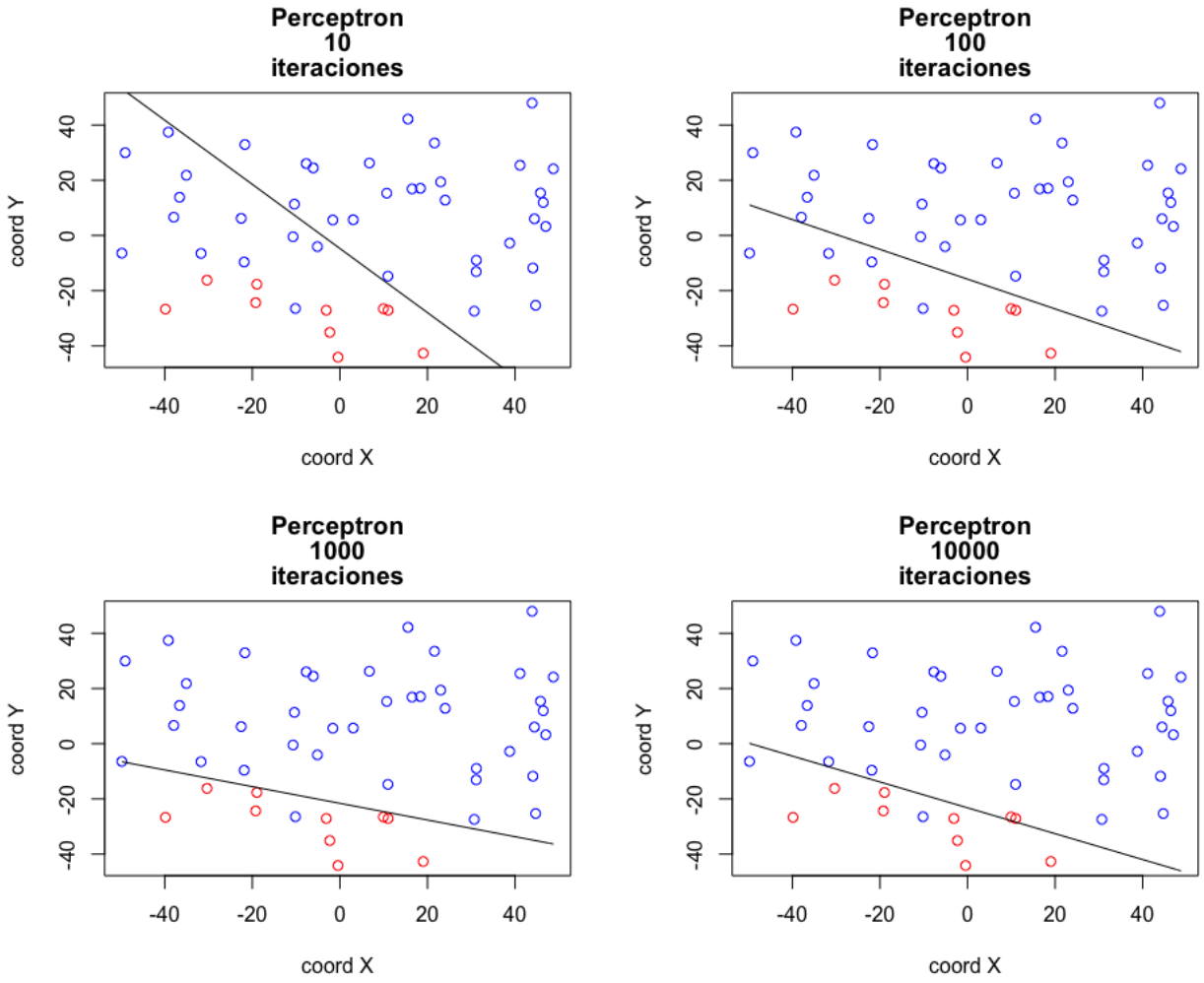
Vamos a crear una función que se base en *ajusta_PLA* y calcule el número de errores que se cometen al clasificar puntos (no necesariamente linealmente separables).
Para simplificar las cosas vamos a usar las funciones auxiliares que hemos ido creando a lo largo de todo el capítulo:

```
ajusta_PLA_puntos <- function(puntosClasificados, max_iter){
    datos <- puntosClasificadosToMatrix(puntosClasificados)
    etiquetas <- datos[,3]
    coefs <- ajusta_PLA(datos, etiquetas, max_iter, c(0,0,0),
   dibujar = T)
    #cuenta errores
    etiquetaPLA <- 2*(puntosClasificados[[2]] - coefs[1]*
   puntosClasificados[[1]] - coefs[2] > 0) - 1;
    num_errores <- length(etiquetaPLA[etiquetaPLA!=etiquetas])
   ;
    return(c(coeficientes=coefs, Errores=num_errores))
}
```

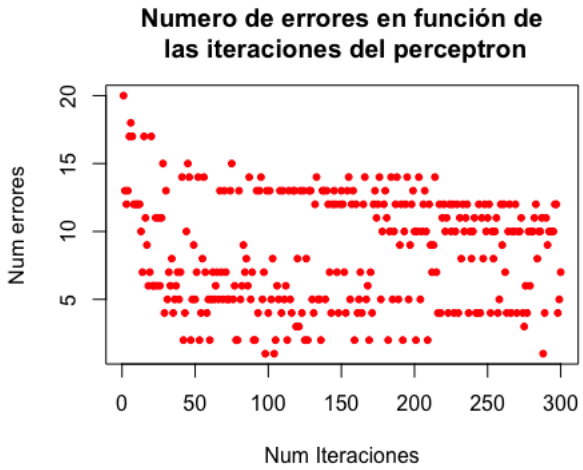
Los resultados que obtenemos son los siguientes:

```
> ajusta_PLA_puntos(misOtrosPuntosClasificadosAlter, 10)
     coef1            coef2                        Errores
     -1.161950      -4.724129                        12
> ajusta_PLA_puntos(misOtrosPuntosClasificadosAlter, 100)
     coef1            coef2                        Errores
     -0.5392779     -15.8210893                        4
> ajusta_PLA_puntos(misOtrosPuntosClasificadosAlter, 1000)
     coef1            coef2                        Errores
     -0.3022515     -21.5970765             1
> ajusta_PLA_puntos(misOtrosPuntosClasificadosAlter, 10000)
     coef1            coef2                        Errores
     -0.4685635     -23.2239785                        4
```

Y obtenemos estos hiperplanos:



Como vemos, el número de errores no siempre se reduce al aumentar el número de iteraciones, pero si parece razonable pensar que cuando hay pocos puntos conflictivos el perceptrón va a comportarse bien.
Veamos lo que ocurre en 300 experimentos:

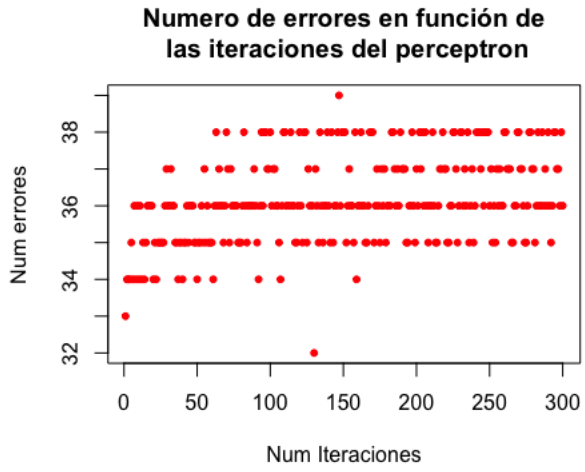


Iteraciones	Error
10	12
100	4
1000	1
10000	4

Ejercicio 14 Repetir el análisis del ejercicio anterior usando puntos clasificados mediante puntos ■

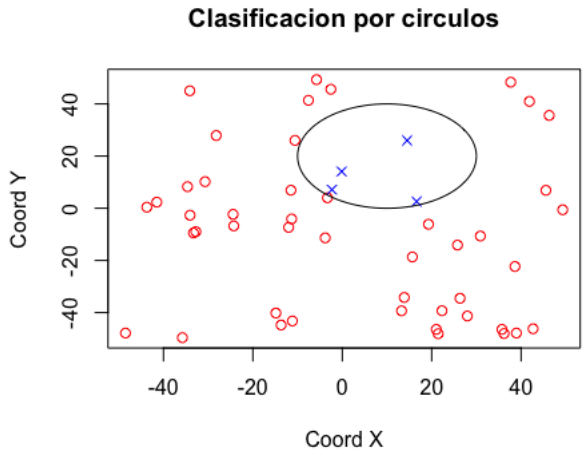
Solución:

```
> misOtrosPuntosClasificadosCirc <- clasificaPuntosFuncion1v2(
   misOtrosPuntos)
> numero_errores_PLA(misOtrosPuntosClasificadosCirc, 10)
[1] 34
> numero_errores_PLA(misOtrosPuntosClasificadosCirc, 100)
[1] 38
> numero_errores_PLA(misOtrosPuntosClasificadosCirc, 1000)
[1] 38
> for(i in 1:300){
+     v_err[i] <- numero_errores_PLA(
   misOtrosPuntosClasificadosCirc, i)
+ }
> plot(1:300, v_err, col="red", pch = 20,  xlab = "Num
   Iteraciones", ylab = "Num errores", main = "Numero de
   errores en funcion de\n las iteraciones del perceptron")
>
```



Iteraciones	Error
10	34
100	38
1000	38

Como podemos observar el número de errores del perceptron depende fundamentamente de lo *cerca o lejos* que esten los datos de ser linealmente separables. En el caso de los datos linealmente separables obteníamos 0 errores (el perceptrón acababa), en el caso de cambiar unas cuantas etiquetas de un modelo que era linealmente separable el perceptron fallaba poco. En este último caso caso los datos se clasificaban mediante un circulo, como vemos en la siguiente figura, esto hace que el perceptrón se inutil para clasificar en este caso.



Ejercicio 15 Modificar la función *ajusta_PLA* para que permita visualizar los datos y soluciones que va encontrando a lo largo de las iteraciones y utilizala en un caso práctico. ■

Vamos a modificarlo para que genere una animación en la que se muestre la evolución del perceptrón. Para ello necesitamos el paquete *animation*.
Con RStudio podemos instalar el paquete y luego activarlo (marcando la casilla correspondiente en la pestaña *Packages* de la ventana derecha).

```
> install.packages("animation")
> library("animation", lib.loc="/Library/Frameworks/R.
   framework/Versions/3.2/Resources/library")
```

Tambien vamos a necesitar el programa **ImageMagick:**
Puede descargarse en `http://www.imagemagick.org/script/binary-releases.php`
Una vez instalado podremos usar la orden *im.convert* en R, que es la que va a crear el archivo -gif con la animación

```
> ajusta_PLA <- function(datos, label, max_iter, v_ini,
   dibujar = FALSE, animation = FALSE){
    w <- v_ini;

    #Limites de los ejes para el plot:
    miny = min(datos[,2]);
    maxy = max(datos[,2]);
    minx = min(datos[,1]);
    maxx = max(datos[,1]);

    #Plot inicial
    plot(x = datos[,1], y= datos[,2], col = label+3, xlim = c(
   minx, maxx), ylim = c(miny, maxy), main = c("Perceptron", h
   , "iteraciones"), xlab = "coord X", ylab = "coord Y" )

    w[ncol(datos)]=0;
    N <- nrow(datos);
    dim <- ncol(datos)-1;
```

```
 salir = FALSE;
 cambio_color_circulo = FALSE; #Variable auxiliar para la animacion
 for(h in 1:max_iter){
     salir <- TRUE;
     for(i in 1:N){
         cambio_color_circulo <- FALSE
         x <- c(datos[i,1:dim],1);
         if( sign(w %*% x) != label[i]){
             w <- w + label[i]*x
             salir <- FALSE;
             cambio_color_circulo <- TRUE
         }
         if(animation == TRUE){
                 coefs <- c(coef1 = -w[1]/w[2], coef2 = -w[3]/w[2])
                 frame = 10000000 + (h-1)*N + i; #El 10000000 es para que la visualizacion se haga en orden
                 filename <- paste("perceptron", frame, ".png", sep="")
                 png(file=filename, width=550, height=550)
                 plot(x = datos[,1], y= datos[,2], col = label+3, xlim = c(minx, maxx), ylim = c(miny, maxy), main = paste("Perceptron iteracion: ", frame - 10000000,"\nRonda", h,  sep=""), xlab = "coord X", ylab = "coord Y" )
                 if(cambio_color_circulo){
                      color_circulo <- "indianred2"
                 }
                 else{
                     color_circulo <- "steelblue2"
                 }

                 symbols(x = datos[i,1], y = datos[i,2], circles=5, inches=1/3, add=T, ann=F, bg=color_circulo, fg=NULL)
                 points(x = datos[i,1], y = datos[i,2], col = datos[i,3]+3, add=T)
                 curve(coefs[1]*x + coefs[2], add = TRUE)
                 dev.off()
             }
     }

     if(salir == TRUE){
         break;
     }
 }
```

```
    coefs <- c(coef1 = -w[1]/w[2], coef2 = -w[3]/w[2])
56
    if (dibujar == TRUE){
58      plot(x = datos[,1], y= datos[,2], col = label+3, xlim
   = c(minx, maxx), ylim = c(miny, maxy), main = c("Perceptron
   ", h, "iteraciones"), xlab = "coord X", ylab = "coord Y" )
        curve(coefs[1]*x + coefs[2], add = TRUE)
60  }

62  if (animation == TRUE){
        im.convert("perceptron*.png", output = "perceptron.gif
   ")
64  }

66  return(coefs)
}
```

Para que veamos un ejemplo:

2.2.2 Versión mejorada del PLA: *PLA - Pocket*

Ejercicio 16 A la vista de la conducta de las soluciones observada en el apartado anterior, proponga e implemente una modificación de la función original *sol = ajusta_PLA_MOD(...)* que permita obtener soluciones razonables sobre datos no linealmente separables. Mostrar y valorar el resultado encontrado usando datos no linealmente separables (los que utilizamos en el modelo circular). ■

El problema era que el número de errores que se cometen no decrecía al aumentar el número de

iteraciones. Por eso la solución más natural es tener una variable que recuerde los pesos asociados a la mejor solución y devolver esos pesos en lugar de los últimos pesos calculados.
La implementación es la siguiente:

```
> ajusta_PLA_pocket <- function(datos, label, max_iter, v_ini,
    dibujar = FALSE, animation = FALSE){
    w <- v_ini;

    #Limites de los ejes para el plot:
    miny = min(datos[,2]);
    maxy = max(datos[,2]);
    minx = min(datos[,1]);
    maxx = max(datos[,1]);

    #Plot inicial
    plot(x = datos[,1], y= datos[,2], col = label+3, xlim = c(
  minx, maxx), ylim = c(miny, maxy), main = c("Perceptron", h
  , "iteraciones"), xlab = "coord X", ylab = "coord Y" )

    w[ncol(datos)]=0;
    N <- nrow(datos);
    dim <- ncol(datos)-1;
    min_num_errores <- N;

    salir = FALSE;
    cambio_color_circulo = FALSE; #Variable auxiliar para la
  animacion
    for(h in 1:max_iter){
        salir <- TRUE;
        recorrido <- sample(1:N, N, replace = F)
        for(i in recorrido){
            cambio_color_circulo <- FALSE
            x <- c(datos[i,1:dim],1);
            if( sign(w %*% x) != label[i]){
                w <- w + label[i]*x
                salir <- FALSE;
                cambio_color_circulo <- TRUE
            }
            if(animation == TRUE){
                coefs <- c(coef1 = -w[1]/w[2], coef2 = -w[3]/w
  [2])
                frame = 10000000 + (h-1)*N + i; #El 10000000
  es para que la visualizacion se haga en orden
                filename <- paste("perceptron", frame, ".png",
    sep="")
                png(file=filename, width=550, height=550)
                plot(x = datos[,1], y= datos[,2], col = label
  +3, xlim = c(minx, maxx), ylim = c(miny, maxy), main =
  paste("Perceptron iteracion: ", frame - 10000000,"\nRonda "
```

```
    , h,  sep=""), xlab = "coord X", ylab = "coord Y" )
37                  if(cambio_color_circulo){
                        color_circulo <- "indianred2"
39                  }
                    else{
41                      color_circulo <- "steelblue2"
                    }
43
                    symbols(x = datos[i,1], y = datos[i,2],
    circles=5, inches=1/3, add=T, ann=F, bg=color_circulo, fg=
    NULL)
45                  points(x = datos[i,1], y = datos[i,2], col =
    datos[i,3]+3, add=T)
                    curve(coefs[1]*x + coefs[2], add = TRUE)
47                  dev.off()
                }
49          }

51          coefs <- c(coef1 = -w[1]/w[2], coef2 = -w[3]/w[2])

53          #cuenta errores
            etiquetaPLA <- 2*(datos[,2] - coefs[1]*datos[,1] -
    coefs[2] > 0) - 1;
55          num_errores_actual <- length(etiquetaPLA[etiquetaPLA!=
    label]);

57          if(num_errores_actual < min_num_errores){
                min_num_errores <- num_errores_actual;
59              mejores_coefs <- coefs;
            }
61          if(salir == TRUE){
                break;
63          }
     }
65
     if (dibujar == TRUE){
67          plot(x = datos[,1], y= datos[,2], col = label+3, xlim
    = c(minx, maxx), ylim = c(miny, maxy), main = c("Perceptron
    ", h, "iteraciones"), xlab = "coord X", ylab = "coord Y" )
            curve(coefs[1]*x + coefs[2], add = TRUE)
69   }

71   if (animation == TRUE){
            im.convert("perceptron*.png", output = "perceptron.gif
    ")
73   }

75   return(mejores_coefs)
}
```

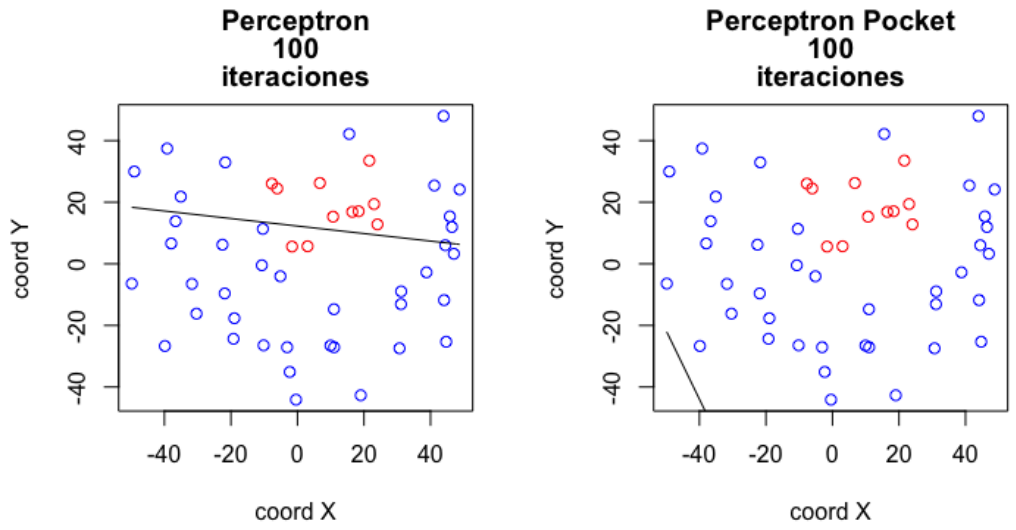
Para facilitar el uso del algoritmo vamos a hacer una nueva interfaz para hacer el PLA directamente desde puntos clasificados.

```
ajusta_PLA_puntos_pocket <- function(puntosClasificados, max_iter){
  datos <- puntosClasificadosToMatrix(puntosClasificados)
  etiquetas <- datos[,3]
  coefs <- ajusta_PLA_pocket(datos, etiquetas, max_iter, c(0,0,0), dibujar = T)
  #cuenta errores
  etiquetaPLA <- 2*(puntosClasificados[[2]] - coefs[1]*puntosClasificados[[1]] - coefs[2] > 0) - 1;
  num_errores <- length(etiquetaPLA[etiquetaPLA!=etiquetas]);
  return(c(coeficientes=coefs, Errores=num_errores))

}
```

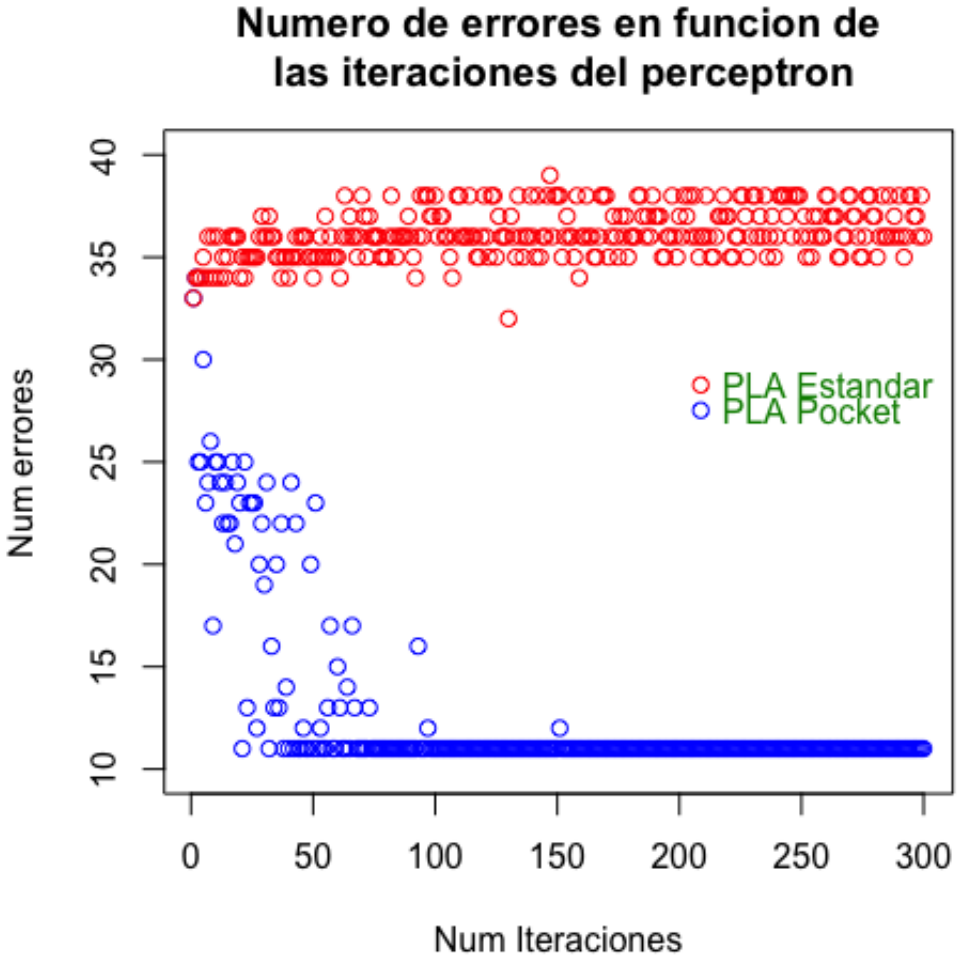
Podemos ver la diferencia entre los dos algoritmos ejecutando para el mismo conjunto de datos y el mismo número de iteraciones los dos algoritmos y comparar el número de puntos mal clasificados:

```
> ajusta_PLA_puntos_pocket(misOtrosPuntosClasificadosCirc,100)
   coef1          coef2                          Errores
   -0.7990891          -72.5481299                    11
> ajusta_PLA_puntos(misOtrosPuntosClasificadosCirc,100)
   coef1          coef2                          Errores
   -0.1221047           12.2697756                     38
```

Como podemos ver el número de errores del pocket es mucho menor, en las gráficas podemos ver como son esas soluciones:

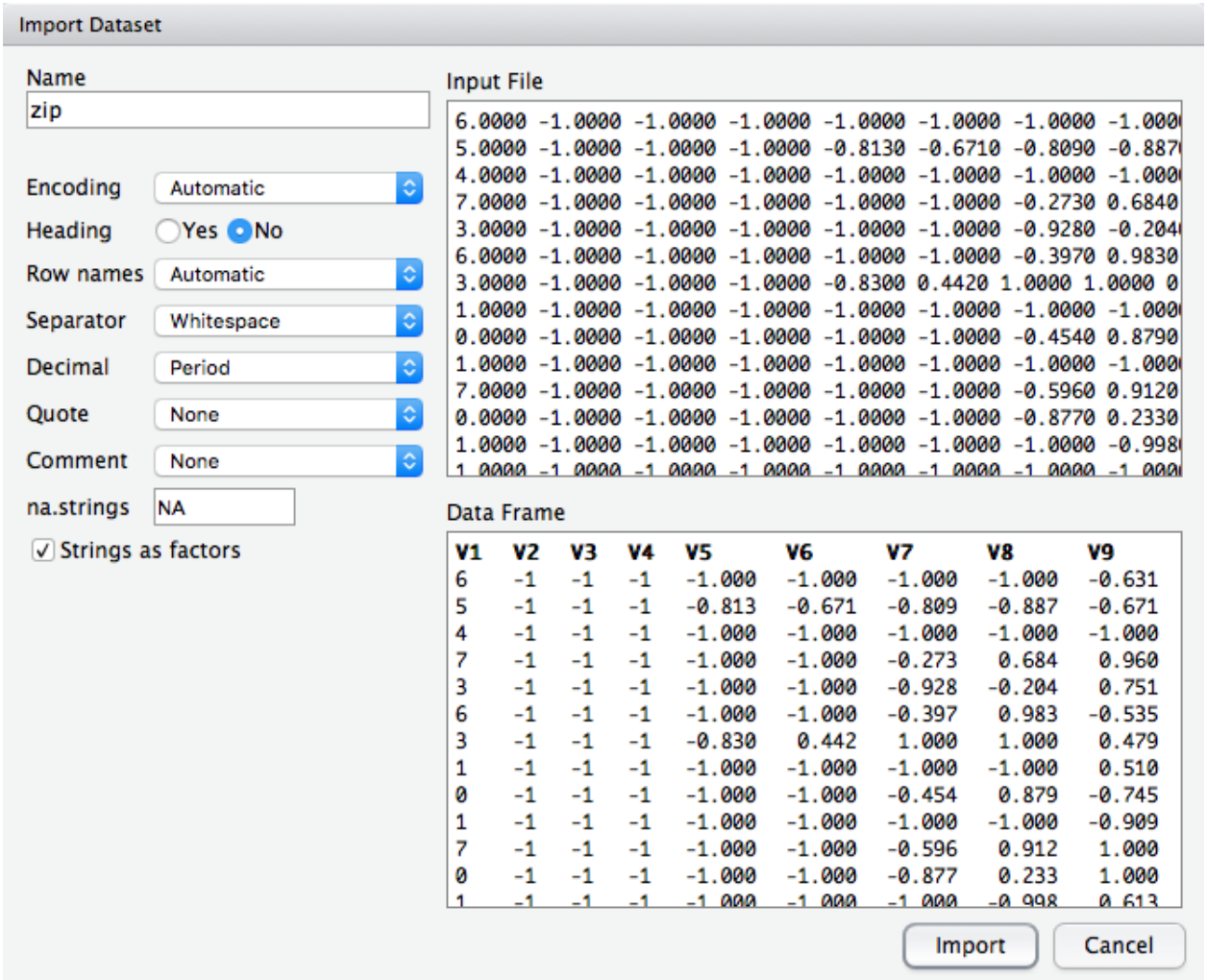


Tambien podemos ver como el número de iteraciones en el pocket decrece cuando aumentamos el *max_iter*, cosa que no ocurría con la version más sencilla: Para el mismo conjunto de datos tenemos:



2.3 Regresión Lineal

A lo largo de esta sección vamos a trabajar con el dataset "*Handwritten Digits (MNIST)*", uno de los más famosos junto con el de las flores "*iris*".
En este caso vamos a trabajar únicamente con los çincosz los ünos"para hacer la tarea mucho más sencilla.
Para importar el dataset vamos al menú de arriba de RStudio: **Tools <Import Dataset <From Local File.**
Y seleccionamos *zip.train*:



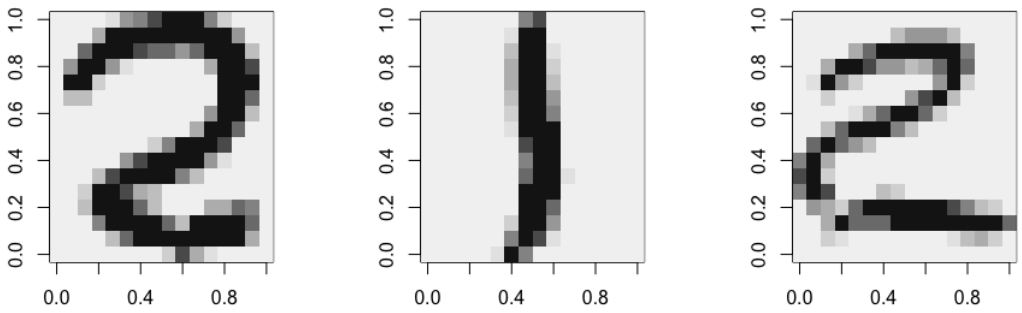
Este proceso creará un data.frame llamado zip con muestras de todos los digitos Una vez tengamos el data.frame vamos a quedarnos solo con los 1 y los 5:

```
> indicesUnosYCincos <- which(zip$V1 == 1 | zip$V1 == 5)
> misNumeros <- zip[indicesUnosYCincos,]
```

Ahora vamos a guardarla en formato de matrices de números 16x16
Para ello vamos a crear una funcion auxiliar que se encargue de pasar los datos a un formato más comodo para R:

```
importarNumeros <- function(data, dibujar = FALSE){
    num_datos <- length(data$V1)
    listaDigitos <- list();
    for(i in 1:num_datos){
        miMatriz_actual <- misNumeros[i,]
        listaDigitos[[i]] <- matrix(as.numeric(miMatriz_actual
   [2:257]),nrow = 16,ncol = 16)
        if(dibujar == TRUE){
            image(z = listaDigitos[[i]], col = rev(grey.colors
   (start = 0.1, n = 10, end = 0.95)))
        }
    }
    return(listaDigitos)
}
#Ejecutamos la funcion y nos quedamos con la lista de numeros:
> miListaNumeros <- importarNumeros(misNumeros)
```

Al ejecutar la función tendremos una lista de matrices, donde cada matriz es de 16x16 y tiene valores numericos en el intervalo $[-1,1]$ donde -1 es blanco y 1 es negro (aunque el color realmente se puede interpretar de muchas formas, en mi caso *col = rev(grey.colors(start = 0.1, n = 10, end = 0.95))* indica -1 es un gris casi blanco y 1 es un gris muy oscuro casi negro.
Tal y como vemos en las imagenes siguientes:



Es importante observar que los cincos están girados y parecen doses.

Ahora vamos a crear una función que muestre como de simétrico es un número respecto de un eje vertical:

```
> calcularGradoSimetria <- function(matriz){
    #Calcula el grado de simetria vertical
    matrizSimetrica <- calcularMatrizSimetricaVertical(matriz)
    gradoSimetria <- sum(abs(matriz - matriz[,16:1]))
    return(gradoSimetria)
}
```

Los números con un grado de simetria alto serán cincos (ya que son muy asimetricos) y aquellos

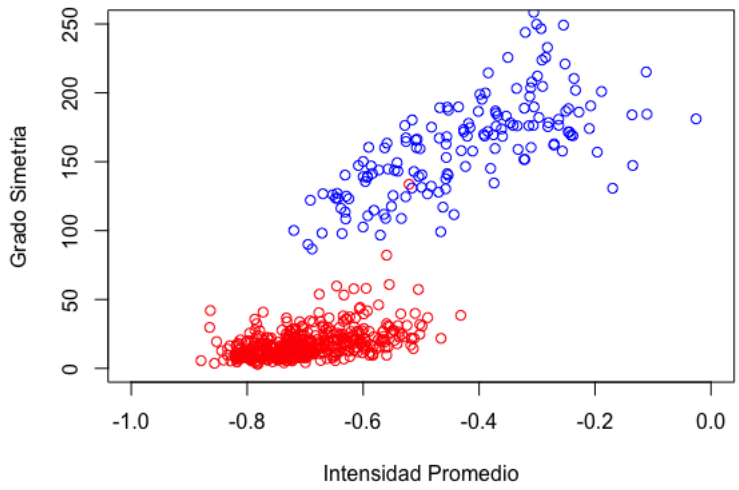
que tengan un grado de asimetria menor serán unos.
La otra función que vamos a utilizar para clasificar va a ser el *valor medio* entendiendolo como la media de todos los valores de la matriz:

```
> gradoIntensidadMedia <- function(matriz){
    return(mean(matriz))
}
```

Ejercicio 17 Representar en los ejes X = Intensidad Promedio, Y = Simetría las instancias seleccionadas de 1's y 5's. ■

En primer lugar vamos a crear dos vectores en los que vamos a colocar en la componente i-esima el valor del grado de simetría y el grado de intensidad del digito i-esimo de nuestro dataset de 1's y 5's.

```
> gradosSimetria <- c()
> for(i in 1:length(miListaNumeros)){
    gradosSimetria[i] <- calcularGradoSimetria(miListaNumeros
  [[i]])
 }
> gradosIntensidad <- c()
> for(i in 1:length(miListaNumeros)){
    gradosIntensidad[i] <- gradoIntensidadMedia(
  miListaNumeros[[i]])
 }
#Podemos ver como quedan el grafico si aniadimos un color a
  cada punto en funcion de su etiqueta:
> plot(x=gradosIntensidad, y=gradosSimetria,col=(
  miVectorEtiquetas-1)/2 + 2, xlab="Intensidad Promedio",
  ylab="Grado Simetria")
```



2.3.1 Conceptos de Algebra Lineal - SVD

Para hacer un clasificiador lineal hay que calcular la inversa de una matriz que en general será bastante grande, por eso un metodo como el de Cramer va a ser muy ineficiente.
En su lugar vamos a utilizar la descomposición en valores singulares de una matriz y despues calcular la inversa aprovechando dicha descomposición.
Dada una matriz:

$$M = XX^t$$

Se puede ver que M es simetrica y definida positiva:
Una matriz simetrica y definida positiva puede descomponerse como:

$$M = USV^t$$

Donde:

- U es una matriz cuyas columnas son los vectores propios de MM^t
- S es una matriz diagonal cuyos elementos son los valores singulares de M
- V es una matriz cuyas columnas son los vectores propios de M^tM

Por tanto la inversa queda así:

$$M^{-1} = (USV^t)^{-1} = (V^t)^{-1}S^{-1}U^{-1} = VD^{-1}U^t$$

Con lo que tan solo debemos invertir una matriz diagonal.
Para ello vamos a implementar la siguiente función en R:

```
> invertirMatrizSVD <- function(matriz){
    svdDesc <- svd(matriz)
    S_inversa_coef <- 1/svdDesc$d
    S_inversa <- diag(x = S_inversa_coef, nrow = nrow(matriz),
   ncol = ncol(matriz))
    V <- matrix(svdDesc$v, nrow = nrow(matriz))
    U <- matrix(svdDesc$u, nrow = nrow(matriz))
    matriz_inversa <- V %*% S_inversa %*% t(U)
    return(matriz_inversa)
}
```

2.3.2 Algoritmo de Regresion lineal

Ejercicio 18 Implementar la funcion *regress_lin(datos, label)* que permita ajustar un modelo de regresion lineal (usar SVD). Los datos de entrada se interpretan igual que en clasificación. ■

En primer lugar vamos a crear el vector de etiquetas (*label*):

```
#Recordamos que las etiquetas eran 5's y 1's y queremos -1's y
    1's:
> head(misNumeros$V1)
[1] 5 1 1 1 1 1
> miVectorEtiquetas <- misNumeros$V1
> miVectorEtiquetas <- (miVectorEtiquetas-3)/2
> head(miVectorEtiquetas)
[1]  1 -1 -1 -1 -1 -1
```

Ahora que ya tenemos *miVectorEtiquetas* vamos a preparar los datos:
Recordamos que la estructura que queremos es $(1|x_1|x_2)$ donde x_1 es el grado de intensidad media de cada dígito y x_2 es el grado de simetría vertical:

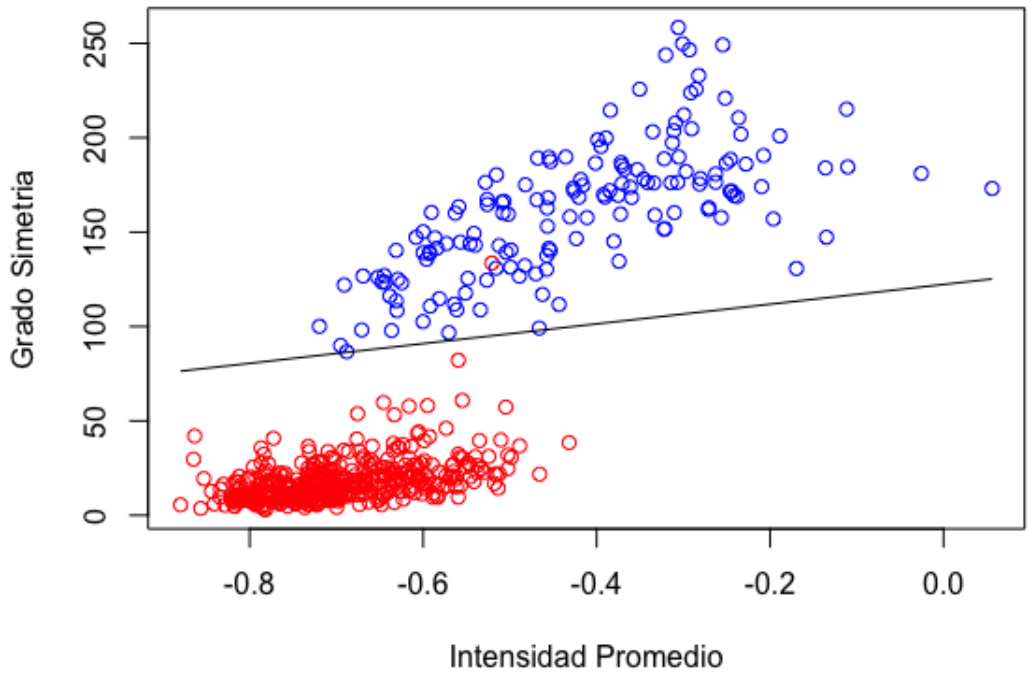
```
> datos <- matrix(c(rep(1, length(gradosIntensidad)),
   gradosIntensidad, gradosSimetria), ncol=3 )
> head(datos)
     [,1]       [,2]    [,3]
[1,]    1 -0.1117383 215.162
[2,]    1 -0.7539141  15.240
[3,]    1 -0.7722813  18.060
[4,]    1 -0.7692578   9.472
[5,]    1 -0.7954375  11.212
[6,]    1 -0.7159141  27.492
>
```

Ahora vamos a implementar la funcion *regress_lin* que es la que se va a encargar de darnos los coeficientes de la recta que separa nuestra nube de puntos de 1's y 5's:

```
> regress_lin <- function(datos, label){
    X <- datos; #datos es una matriz [N x 3] ---->>> (1, x0,
   x1)
    H <- (invertirMatrizSVD(t(X) %*% X) %*% t(X))
    w <- H %*% t(matrix(label, nrow = 1))

  # hiperplano: w1 + w2*x + w3*y = 0  ==>
    coefs <- c(-w[2]/w[3], -w[1]/w[3])
    return(coefs)
}
```

Utilizamos la función que acabamos de implementar y obtenemos unos coeficientes que colocamos en la recta y vemos que la nube se queda bien separada:

```
> regress_lin(datos, miVectorEtiquetas)
[1]  52.19644 122.25713
> plot(x=gradosIntensidad, y=gradosSimetria,col=
   miVectorEtiquetas +3, xlab="Intensidad Promedio", ylab="
   Grado Simetria")
> curve(52.19644*x + 122.25713, add = T)
```



Ejercicio 19 En este ejercicio exploramos como funciona regresión lineal en problemas de clasificación. Para ello generamos datos usando el mismo procedimiento que en los ejercicios anteriores. Suponemos $X = [-10, 10] \times [-10, 10]$ 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.

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

■

En primer lugar vamos a hacer una única prueba:

```
> misPuntos <- simula_unif(N = 100, dim = 2, min = -10, max =
    10)
> misPuntosClasificados <- clasificaPuntosRectav2(misPuntos,
    rectaF[1], rectaF[2])
> matrizDatos <- puntosClasificadosToMatrix(
    misPuntosClasificados)
> datos <- matrix( c(rep(1, length(matrizDatos[,1])),
    matrizDatos[,1], matrizDatos[,2]), ncol=3)
```

```
> etiquetas <- matrizDatos[,3]
> recta_regresion <- regress_lin(datos, etiquetas)
> curve(recta_regresion[1]*x + recta_regresion[2], add = T,
   col="blue", lwd=3, lty= 5)
```

Creamos una función para ver el número de puntos que quedan mal clasificados que se utilizará para ver el error relativo:

```
> numero_errores_regress <- function(puntosClasificados, recta
   ){
    etiquetasReales <- puntosClasificadosToMatrix(
   puntosClasificados)[,3]
    etiquetaRegresion <- 2*(puntosClasificados[[2]] - recta[1]
   *puntosClasificados[[1]] - recta[2] > 0) - 1;
    num_errores <- length(etiquetaRegresion[etiquetaRegresion!
   =etiquetasReales]);
    return(num_errores)
}

> errorRelativo <- function(puntosClasificados, recta){
    num_errores <- numero_errores_regress(puntosClasificados,
   recta)
    errorRelativo <- num_errores / length(puntosClasificados
   [[1]])
    return(errorRelativo)
}
```

Hacemos una prueba y vemos que el error fuera de esa muestra es 2'8

```
> errorRelativo(misPuntosClasificados, recta_regresion)
[1] 0.03
#Veamos lo que pasa fuera de la muestra:
> misPuntosFuera <- simula_unif(1000, 2, -10, 10)
> misPuntosFueraClasificados <- clasificaPuntosRectav2(
   misPuntosFuera, rectaF[1], rectaF[2])
> errorRelativo(misPuntosFueraClasificados, recta_regresion)
[1] 0.028
```

Vamos a ver como se comporta la regresión dentro de la muestra haciendo 1000 experimentos:

```
# E_in:
> v_errores <- c()
> for(i in 1:1000){
    misPuntos <- simula_unif(100, 2, -10, 10)
    recta <- simula_recta(-10, 10)
    misPuntosClasificados <- clasificaPuntosRectav2(misPuntos,
    recta[1], recta[2])
```

```
    matrizDatos <- puntosClasificadosToMatrix(
   misPuntosClasificados)
    datos <- matrix(c(rep(1,length(matrizDatos[,1])),
   matrizDatos[,1], matrizDatos[,2]),ncol=3)
    etiquetas <- matrizDatos[,3]
    recta_regresion <- regress_lin(datos, etiquetas)
    v_errores[i] <- errorRelativo(misPuntosClasificados, recta
   _regresion)
}

#En media nos equivocamos en el 5'66% de los puntos
> mean(v_errores)
[1] 0.05774
```

Ahora vamos a estimar el error fuera de la muestra:

```
#E_out
> v_errores_fuera <- c()
> for(i in 1:1000){
    misPuntos <- simula_unif(100, 2, -10, 10)
    misPuntosFuera <- simula_unif(1000, 2, -10, 10)
    recta <- simula_recta(-10, 10)
    misPuntosClasificados <- clasificaPuntosRectav2(misPuntos,
    recta[1], recta[2])
    misPuntosClasificadosFuera <- clasificaPuntosRectav2(
   misPuntosFuera, recta[1], recta[2])
    matrizDatos <- puntosClasificadosToMatrix(
   misPuntosClasificados)
    datos <- matrix(c(rep(1,length(matrizDatos[,1])),
   matrizDatos[,1], matrizDatos[,2]),ncol=3)
    etiquetas <- matrizDatos[,3]
    recta_regresion <- regress_lin(datos, etiquetas)

    v_errores_fuera[i] <- errorRelativo(
   misPuntosClasificadosFuera, recta_regresion)
}
> mean(v_errores_fuera)
[1] 0.075112
```

2.3.3 Regresion y PLA

Los pesos que se obtienen a partir del algoritmo de regresión pueden usarse como pesos iniciales para el PLA ya que son unos pesos que ya de por sí aproximan la recta objetivo del PLA.
El algorimo de Regresión es computacionalmente muy eficiente y puede hacer que el PLA converja más rápido.

No voy a programarlo ya que la estructura de matrices de datos que he usado para el PLA es diferente del que he usado en el modelo de regresión.

Para ajustar los pesos habría que hacer una función como esta:

```
> pesos_PLA_regress_lin <- function(datos, label){
    X <- datos; #datos es una matriz [N x 3] ---->>> (1, x0,
   x1)
    H <- (invertirMatrizSVD(t(X) %*% X) %*% t(X))
    w <- H %*% t(matrix(label, nrow = 1))
    #Devolvemos el vector de pesos "girado" porque en el PLA
   datos es (x0, x1, 1)
    w_PLA <- c()
    w_PLA[1] <- w[2]
    w_PLA[2] <- w[3]
    w_PLA[3] <- w[1]
    return(w_PLA)
}
```