Práctica 2. Aprendizaje Automático

José Carlos Martínez Velázquez

2 de Mayo de 2016

Modelos lineales

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

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

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

El gradiente de E(u,v) no es más que su derivada con respecto de cada una de sus variables. Lo primero que haremos será desarrollar la expresión para dejar E(u,v) como sumandos:

$$E(u,v) = (ue^v - 2ve^{-u})^2 = u^2e^{2v} + 4e^{-2u}v^2 - 4uve^{v-u}$$

A partir de aquí la estrategia es clara. La derivada de una suma es la suma de las derivadas de los sumandos. Iremos calculando ambas derivadas de forma paralela.

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

Del mismo modo tenemos que:

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

Ya sólo nos queda ir calculando la derivada de cada sumando con respecto a cada variable:

$$\frac{\partial(u^2e^{2v})}{\partial u} = 2e^{2v}u$$
$$\frac{\partial(u^2e^{2v})}{\partial v} = u^2e^{2v}2$$

$$\frac{\partial(4e^{-2u}v^2)}{\partial u} = -8e^{-2u}v^2$$
$$\frac{\partial(4e^{-2u}v^2)}{\partial v} = 8e^{-2u}v$$

$$\frac{\partial(4uve^{v-u})}{\partial u} = -4e^{v-u}(u-1)v$$
$$\frac{\partial(4uve^{v-u})}{\partial v} = 4e^{v-u}(v+1)u$$

Así, nos queda que:

$$\frac{\partial E(u,v)}{\partial u} = 2e^{2v}u \ + \ (-8e^{-2u}v^2) \ - \ (-4e^{v-u}(u-1)v)$$
$$\frac{\partial E(u,v)}{\partial v} = 2u^2e^{2v} \ + \ 8e^{-2u}v \ - \ 4e^{v-u}(v+1)u$$

Teniendo esto, implementamos las funciones directamente:

```
f<-function(u,v){
  u*u*exp(2*v)+4*exp(-2*u)*v*v-4*u*v*exp(v-u)
}


dfdu<-function(u,v){
  2*exp(2*v)*u-8*exp(-2*u)*v*v+4*exp(v-u)*(u-1)*v

}

dfdv<-function(u,v){
  2*u*u*exp(2*v)+8*exp(-2*u)*v-4*exp(v-u)*(v+1)*u
}
```

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

Teniendo las derivadas ya sólo nos queda entender el algoritmo de GD. Imaginemos que nos acabamos de levantar y nuestras sábanas están muy arrugadas, lo más parecido a una función 3D con máximos y mínimos. Escogemos un punto cualquiera y tiramos una bolita (no muy pesada para que no se deformen las sábanas) ¿dónde se va a parar? En un mínimo local. ¿Siempre se va a parar en el mismo lugar? No, evidentemente donde se pare la bolita va a depender de dónde tiremos la bolita (punto de inicio).

Con esta idea básica podemos entender el algoritmo de gradiente descendente, pero ¿qué es eso de tasa de aprendizaje? (η). Eso es más dificil de ver en el ejemplo de la bolita, pero cuando esa bolita se está moviendo, ese movimiento tiene un vector director con un determinado módulo. El módulo de este vector nos va a decir cuánto va a avazar la bolita hasta que se tenga que volver a preguntar ¿hacia donde voy ahora? La tasa de aprendizaje nos es una medida que nos reduce el módulo del vector de movimiento en un porcentaje determinado. De este modo, podemos hacer que la bolita se aproxime a su objetivo despacio (podría no avanzar) o tan rápido que podríamos perder la bolita por fuertes oscilaciones (divergencia). La obtención de una buena tasa de aprendizaje no es una tarea mecánica ni mucho menos sencilla. Hay ríos de tinta escritos sobre esto y sólo se ha llegado a conjeturas, pero la forma en que nosotros las obtendremos será a ensayo y error.

La matemática subyacente no tiene más que entender el concepto de derivada, recta tangente a un punto de una curva. En esta recta se encuentra nuestro vector director. El vector final no es más que restarle al vector actual un trocito, que viene dado por la tasa de aprendizaje.

Creo que con el ejemplo se entiende, así que vamos a implementar el algoritmo de gradiente descendente:

```
gradiente_descendente<-function(punto_comienzo,f,dfdu,dfdv,umbral,max_iter,tasa_aprend){
  cont_iter<-0
  w<-punto_comienzo
  valor_f<-f(w[1,1],w[2,1])
  du<-0
  dv<-0
  points_mat=matrix(punto_comienzo,nrow=2)
```

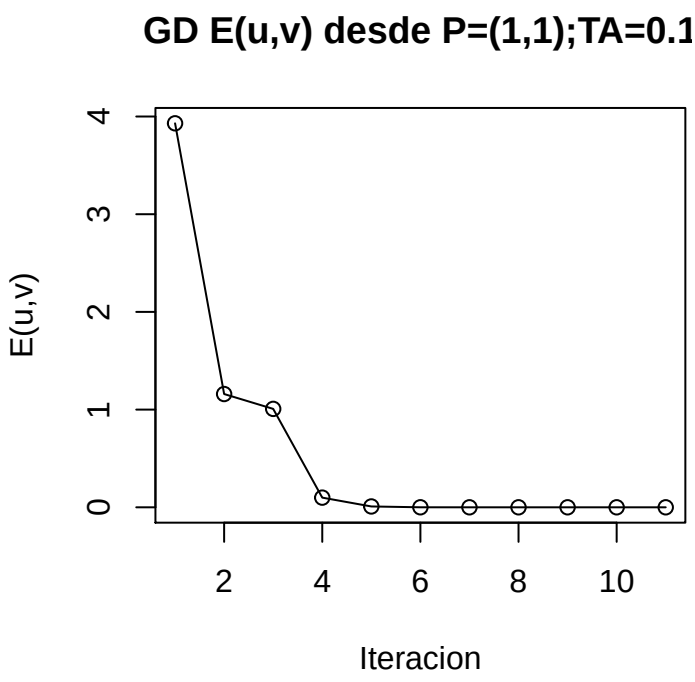
```
  while(cont_iter<max_iter && valor_f>umbral){
    du<-dfdu(w[1,1],w[2,1])
    dv<-dfdv(w[1,1],w[2,1])
    w[1,1]<-w[1,1] - tasa_aprend*du
    w[2,1]<-w[2,1] - tasa_aprend*dv
    valor_f<-f(w[1,1],w[2,1])
    cont_iter<-cont_iter+1
    #print(paste("Iteracion ",cont_iter,": ",w[1,1],",",w[2,1],",",valor_f))
    points_mat<-cbind(points_mat,w)
  }
  points_mat
}
```

Vamos a ver en funcionamiento el algoritmo. Pararemos cuando el valor de la funcion sea menor o igual a 10^{-14} o bien se hayan completado 100 iteraciones.

```
gradiente_descendente(
  matrix(c(1,1),nrow=2),f,dfdu,dfdv,as.double(1e-14),100,as.double(0.1)
)->pt
```

En la siguiente gráfica vamos a ver cómo se comporta el algoritmo GD, si alcanza el mínimo y si diverge:

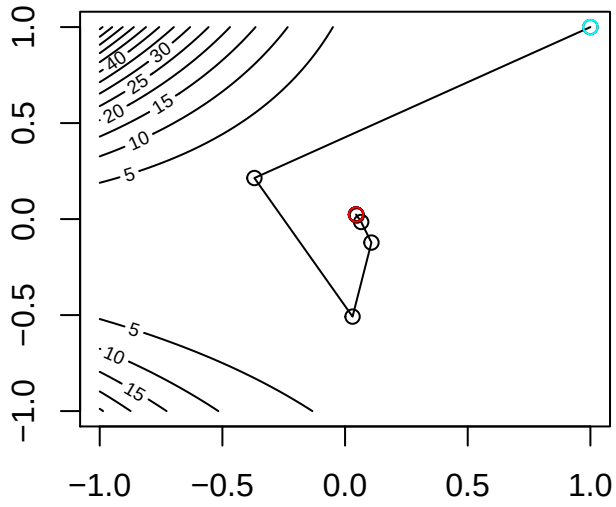
```
plot(f(pt[1,],pt[2,]),type="o",xlab="Iteracion",ylab="E(u,v)",
     main="GD E(u,v) desde P=(1,1);TA=0.1")
```



Vemos cómo, en tan solo 10 iteraciones (quitamos el punto de inicio), el algoritmo ha alcanzado el mínimo local más cercano.

Como extra, en la siguiente gráfica podemos ver el contorno de la gráfica y qué mínimo ha alcanzado.

```
pintar_contorno<-function(x_min_range,
                          x_max_range,
                          y_min_range,
                          y_max_range,
                          precision,func,pt,col_ini="cyan",col_fin="red"){
  xpt<-seq(x_min_range,x_max_range,length.out = precision)
  ypt<-seq(y_min_range,y_max_range,length.out = precision)
  zpt<-outer(xpt,ypt,func)
  contour(xpt,ypt,zpt)
  points(t(pt))
  lines(t(pt))
  points(t(pt)[1,1],t(pt)[1,2],col="cyan")
  points(t(pt)[nrow(t(pt)),1],t(pt)[nrow(t(pt)),2],col="red")
}

pintar_contorno(-1,1,-1,1,100,f,pt)
```



Donde el punto color cyan es el punto de inicio y el punto color rojo es el mínimo alcanzado.

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

Tal y cómo está diseñado el algoritmo, para responder a esta pregunta sólo tenemos que ver el último punto que nos devuelve GD y evaluarlo en E(u,v):

```
print(paste("Ultimo punto de GD: (",
            t(pt)[nrow(t(pt)),1],
            ",",
            t(pt)[nrow(t(pt)),2],
            ") -> ",
            f(t(pt)[nrow(t(pt)),1],t(pt)[nrow(t(pt)),2])))
```

```
## [1] "Ultimo punto de GD: ( 0.0447362903977819 , 0.0239587140991417 ) ->  1.20910226275583e-15"
```

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

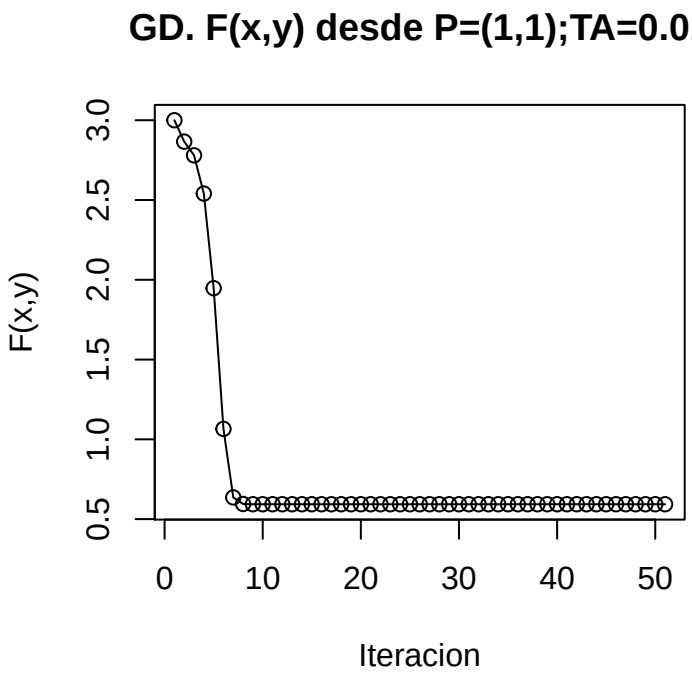
Para ahorrar tiempo, calcularemos las derivadas con algún tipo de software matemático. Una vez obtenidas, las implementamos:

```
f_2<-function(x,y){
  x*x+2*(y*y)+2*sin(2*pi*x)*sin(2*pi*y)
}


dfdx_2<-function(x,y){
  2*(x+2*pi*sin(2*pi*y)*cos(2*pi*x))
}

dfdy_2<-function(x,y){
  4*(y+pi*sin(2*pi*x)*cos(2*pi*y))
}
```

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

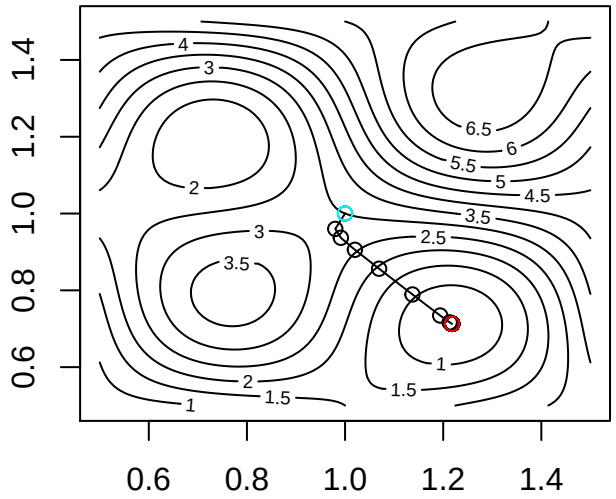
Del mismo modo que antes, ejecutamos gradiente descendiente y visualizamos las mismas gráficas:

```
gradiente_descendente(
  matrix(c(1,1),nrow=2),f_2,dfdx_2,dfdy_2,as.double(1e-14),50,as.double(0.01)
)->pt_2

plot(f_2(pt_2[1,],pt_2[2,]),type="o",xlab="Iteracion",ylab="F(x,y)",
     main="GD. F(x,y) desde P=(1,1);TA=0.01")
```



Vamos a visualizar tambien la gráfica de contorno:

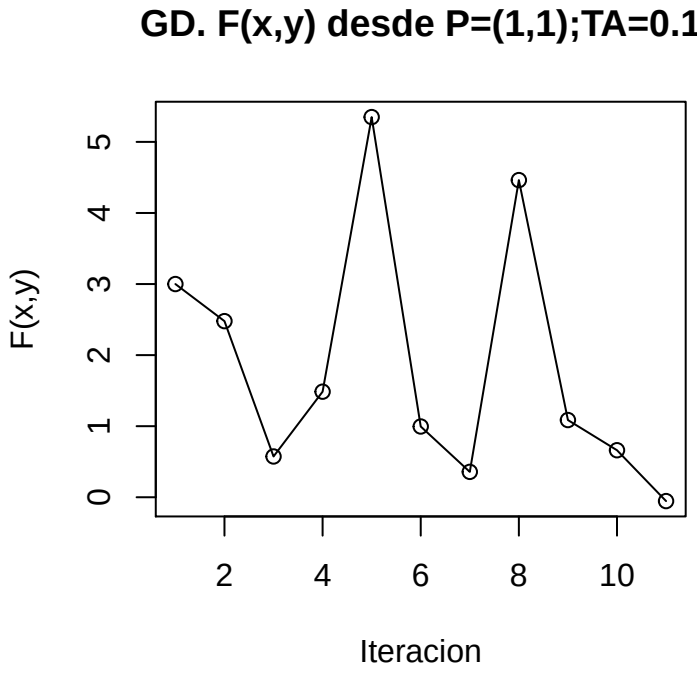
```
pintar_contorno(0.5,1.5,0.5,1.5,100,f_2,pt_2)
```



Con una tasa de aprendizaje de 0.01 vemos cómo alcalza el mínimo local pero no para en 50 iteraciones pues el valor de la función no es menor que el umbral en ningún momento.

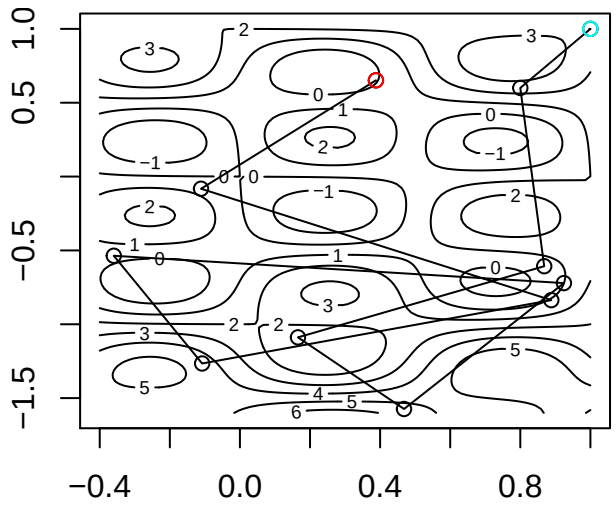
Vamos a repetir el experimento para una tasa de aprendizaje $\eta = 0.1$:

```
gradiente_descendente(
  matrix(c(1,1),nrow=2),f_2,dfdx_2,dfdy_2,as.double(1e-14),50,as.double(0.1)
)->pt_2

plot(f_2(pt_2[1,],pt_2[2,]),type="o",xlab="Iteracion",ylab="F(x,y)",
     main="GD. F(x,y) desde P=(1,1);TA=0.1")
```



Vamos a visualizar tambien la gráfica de contorno en este caso:

```
pintar_contorno(-0.4,1,-1.6,1.0,100,f_2,pt_2)
```



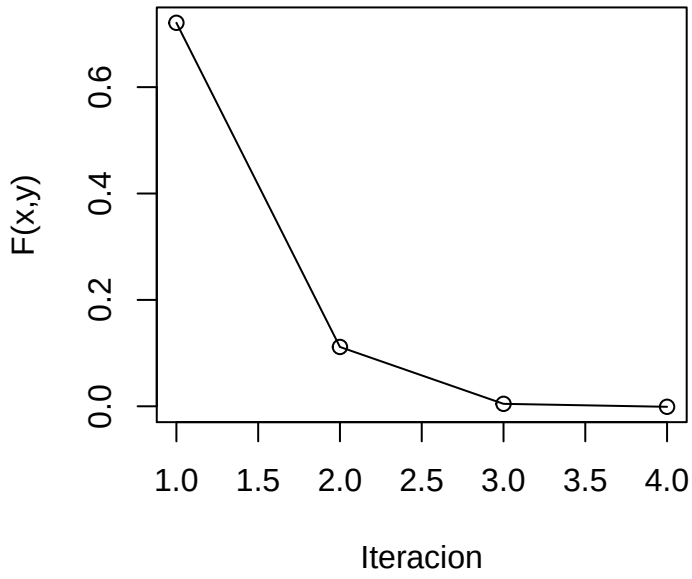
Con una tasa de aprendizaje $\eta = 0.01$ vemos cómo alcalza el mínimo local más cercano pero no para en 50 iteraciones pues el valor de la función no es menor que el umbral en ningún momento. En el caso de una tasa de aprendizaje $\eta = 0.1$, el algoritmo diverge, pero alcanza un mínimo en 10 iteraciones.

2) Obtener el valor mínimo y los valores de las variables que lo alcanzan cuando el punto de inicio se fija: (0,1, 0,1), (1, 1),(-0,5, -0,5),(-1, -1). Generar una tabla con los valores obtenidos ¿Cuál sería su conclusión sobre la verdadera dificultad de encontrar el mínimo global de una función arbitraria?

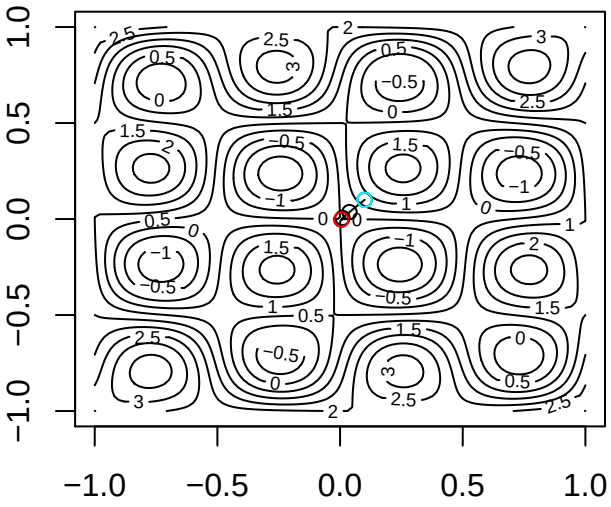
Como el ejercicio no especifica nada, vamos a considerar la tasa de aprendizaje que se nos imponía en el ejericio anterior, es decir, 0.01.

Para punto P=(0.1,0.1)

```
gradiente_descendente(
  matrix(c(0.1,0.1),nrow=2),f_2,dfdx_2,dfdy_2,as.double(1e-14),50,as.double(0.01)
)->pt_2

plot(f_2(pt_2[1,],pt_2[2,]),type="o",xlab="Iteracion",ylab="F(x,y)",
     main="GD. F(x,y) desde P=(0.1,0.1);TA=0.01")
```




```
pintar_contorno(-1,1,-1,1,100,f_2,pt_2)
```



El algoritmo converge en pocas iteraciones, se queda en un valor de f aproximadamente igual a 0 porque es un punto de silla.

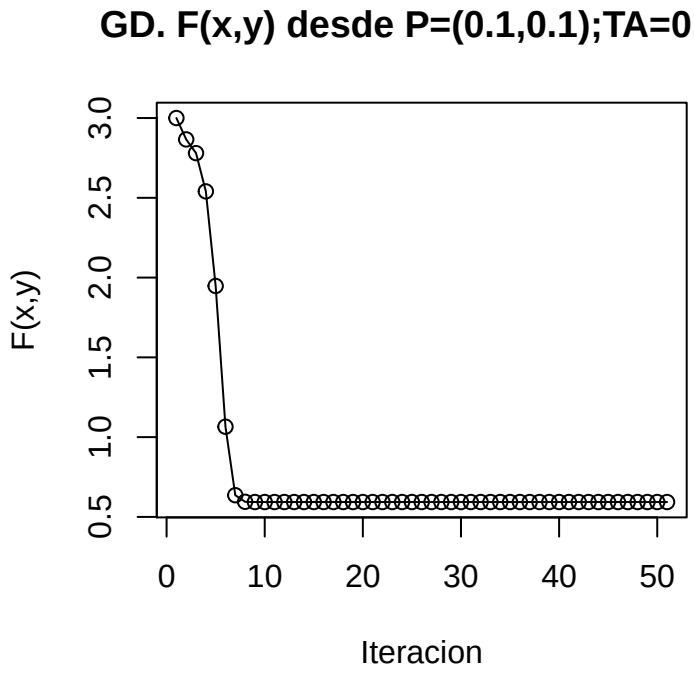
El punto mínimo ha sido:

```
print(paste("Inicio (0.1,0.1) ->",f_2(0.1,0.1)," (",
            t(pt_2)[nrow(t(pt_2)),1],
            ",",
            t(pt_2)[nrow(t(pt_2)),2],
            ")->",
            f_2(t(pt_2)[nrow(t(pt_2)),1],t(pt_2)[nrow(t(pt_2)),2])))
```

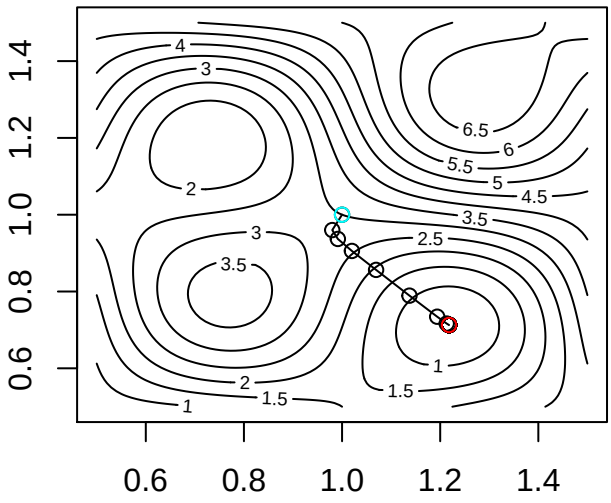
```
## [1] "Inicio (0.1,0.1) -> 0.720983005625053  ( 0.00526609481014761 , -0.00239206934812207 )-> -0.0009
```

Para punto P=(1,1)

```
gradiente_descendente(
  matrix(c(1,1),nrow=2),f_2,dfdx_2,dfdy_2,as.double(1e-14),50,as.double(0.01)
)->pt_2

plot(f_2(pt_2[1,],pt_2[2,]),type="o",xlab="Iteracion",ylab="F(x,y)",
     main="GD. F(x,y) desde P=(0.1,0.1);TA=0.01")
```




```
pintar_contorno(0.5,1.5,0.5,1.5,100,f_2,pt_2)
```



Este experimento lo habíamos realizado anteriormente y veíamos que convergía pero no paraba antes de 50 iteraciones porque el valor de f no era inferior al umbral.

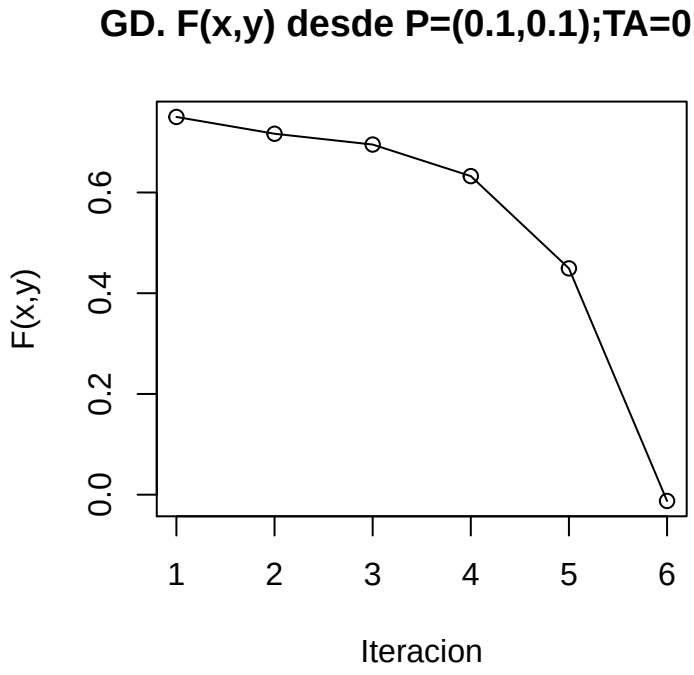
El punto mínimo ha sido:

```r
print(paste("Inicio (1,1) ->",f_2(1,1)," (",
            t(pt_2)[nrow(t(pt_2)),1],
            ",",
            t(pt_2)[nrow(t(pt_2)),2],
            ")->",
            f_2(t(pt_2)[nrow(t(pt_2)),1],t(pt_2)[nrow(t(pt_2)),2])))
```

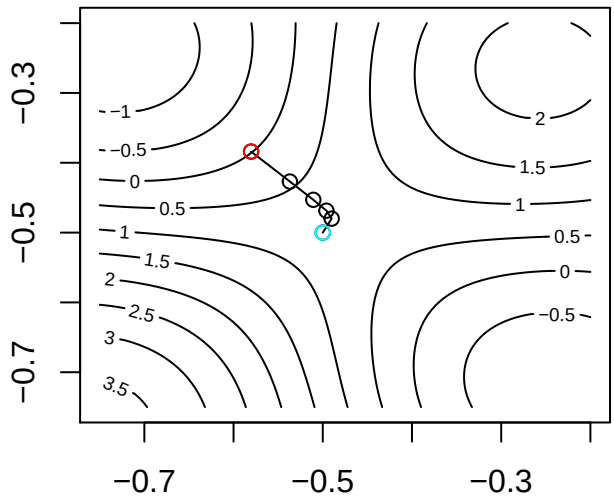
```
## [1] "Inicio (1,1) -> 3  ( 1.21807030131108 , 0.712811950601778 )-> 0.593269374325836"
```

Para punto P=(-0.5,-0.5)

```r
gradiente_descendente(
  matrix(c(-0.5,-0.5),nrow=2),f_2,dfdx_2,dfdy_2,as.double(1e-14),50,as.double(0.01)
)->pt_2

plot(f_2(pt_2[1,],pt_2[2,]),type="o",xlab="Iteracion",ylab="F(x,y)",
     main="GD. F(x,y) desde P=(0.1,0.1);TA=0.01")
```




```r
pintar_contorno(-0.75,-0.2,-0.75,-0.2,100,f_2,pt_2)
```



El algoritmo converge en pocas iteraciones pero no a un mínimo local, ya que, el primer valor alcanzado antes del mínimo local supera el umbral. Para que convergiera al mínimo local habría que bajar el umbral a -1.

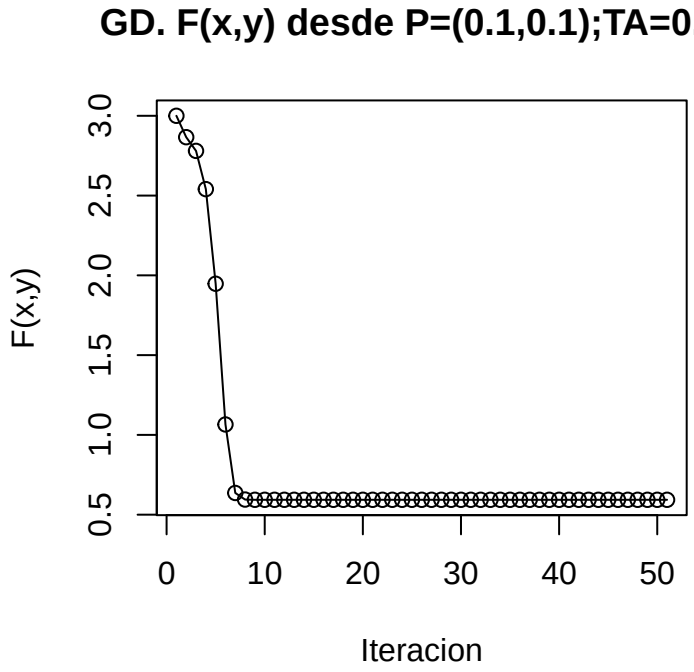
El punto mínimo ha sido:

```
print(paste("Inicio (-0.5,-0.5) ->",f_2(-0.5,-0.5)," (",
            t(pt_2)[nrow(t(pt_2)),1],
            ",",
            t(pt_2)[nrow(t(pt_2)),2],
            ")->",
            f_2(t(pt_2)[nrow(t(pt_2)),1],t(pt_2)[nrow(t(pt_2)),2])))
```

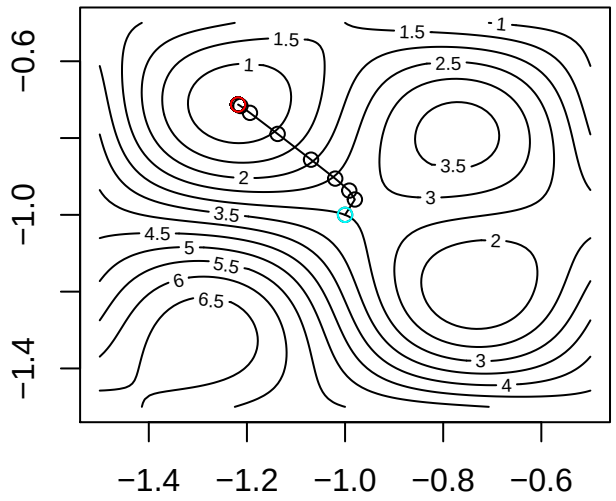
```
## [1] "Inicio (-0.5,-0.5) -> 0.75  ( -0.580323406173378 , -0.383991863552131 )-> -0.012440020728642"
```

Para punto P=(-1,-1)

```
gradiente_descendente(
  matrix(c(-1,-1),nrow=2),f_2,dfdx_2,dfdy_2,as.double(1e-14),50,as.double(0.01)
)->pt_2

plot(f_2(pt_2[1,],pt_2[2,]),type="o",xlab="Iteracion",ylab="F(x,y)",
     main="GD. F(x,y) desde P=(0.1,0.1);TA=0.01")
```




```
pintar_contorno(-1.5,-0.5,-1.5,-0.5,100,f_2,pt_2)
```



Al igual que el punto (1,1), el algoritmo no para antes de 50 iteraciones porque el valor de f no es inferior al umbral.

El punto mínimo ha sido:

```
print(paste("Inicio (-1,-1) ->",f_2(-1,-1)," (",
            t(pt_2)[nrow(t(pt_2)),1],
            ",",
            t(pt_2)[nrow(t(pt_2)),2],
            ")->",
            f_2(t(pt_2)[nrow(t(pt_2)),1],t(pt_2)[nrow(t(pt_2)),2])))
```

```
## [1] "Inicio (-1,-1) -> 3  ( -1.21807030131108 , -0.712811950601778 )-> 0.593269374325836"
```

x_0	y_0	x_f	y_f	$F(x_0, y_0)$	$F(x_f, y_f)$
0.1	0.1	0.0053	-0.0024	0.721	0.001
1	1	1.218	0.713	3	0.593
-0.5	-0.5	-0.58	-0.384	0.75	0.012
-1	-1	-1.218	-0.713	3	0.593

A la vista de los resultados, vemos que encontrar un mínimo local ya es dificil, pues depende del punto de inicio y de la tasa de aprendizaje. Con una tasa de aprendizaje muy grande podemos diverger y con una muy pequeña no llegar a converger porque el módulo del vector gradiente apenas nos permita avanzar. Encontrar el mínimo global requerirá un estudio previo y, en caso de querer asegurar el mínimo local, habrá que aplicar gradiente descendente asegurándonos de empezar en el entorno del mismo y con un umbral menor o igual que el mínimo. Podríamos probar técnicas multiarranque, pero nada nos asegura llegar.

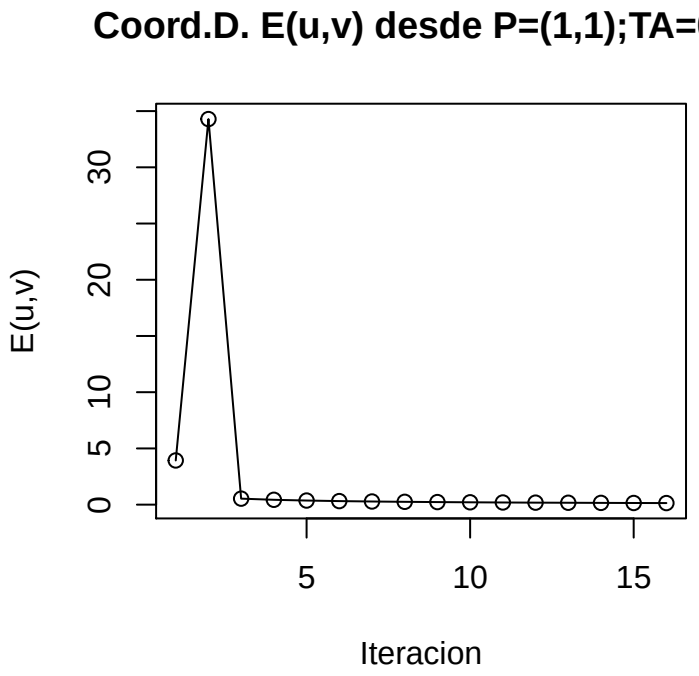
2. Coordenada descendente. En este ejercicio comparamos la eficiencia de la técnica de optimización de "coordenada descendente" usando la misma función del ejercicio 1.1a. En cada iteración, tenemos dos pasos a lo largo de dos coordenadas. En el Paso-1 nos movemos a lo largo de la coordenada u para reducir el error (suponer que se verifica una aproximación de primer orden como en gradiente descendente), y el Paso-2 es para reevaluar y movernos a lo largo de la coordenada v para reducir el error (hacer la misma hipótesis que en el paso-1). Usar una tasa de aprendizaje $\eta = 0,1$.

Se puede hacer el algoritmo de coordenada descendente a partir del algoritmo de gradiente desdencente, la única diferencia es que se actualizar primero la variable u (o x) y posteriormente se actualiza v (o y) con el nuevo valor de u (o x). Veamos cómo se implementa:

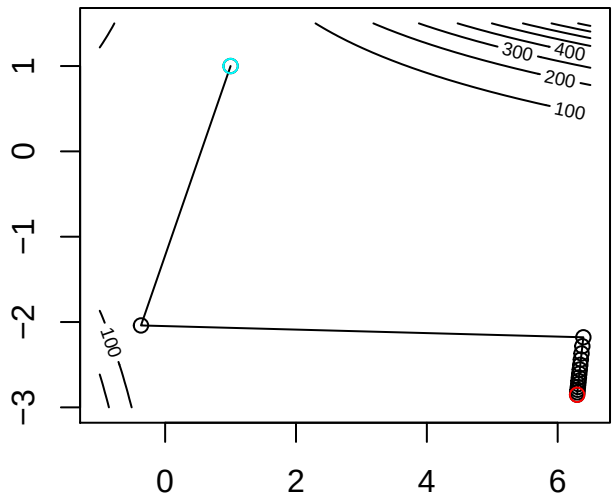
```
coordenada_descendente<-function(punto_comienzo,f,dfdu,dfdv,umbral,max_iter,tasa_aprend){
  cont_iter<-0
  w<-punto_comienzo
  valor_f<-f(w[1,1],w[2,1])
  du<-0
  dv<-0
  points_mat=matrix(punto_comienzo,nrow=2)
  while(cont_iter<max_iter && valor_f>umbral){
    du<-dfdu(w[1,1],w[2,1])
    w[1,1]<-w[1,1] - tasa_aprend*du

    dv<-dfdv(w[1,1],w[2,1])
    w[2,1]<-w[2,1] - tasa_aprend*dv

    valor_f<-f(w[1,1],w[2,1])

    cont_iter<-cont_iter+1
    #print(paste("Iteracion ",cont_iter,": ",w[1,1],",",w[2,1],",",valor_f))
    points_mat<-cbind(points_mat,w)
  }
  points_mat
}
```

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

```
coordenada_descendente(
  matrix(c(1,1),nrow=2),f,dfdu,dfdv,as.double(1e-14),15,as.double(0.1)
)->pt

plot(f(pt[1,],pt[2,]),type="o",xlab="Iteracion",ylab="E(u,v)",
     main="Coord.D. E(u,v) desde P=(1,1);TA=0.1")
```




```
pintar_contorno(-1,6.5,-3,1.5,100,f,pt)
```




```
print(paste("El valor de E(u,v) obtenido es: ",
            f(t(pt)[nrow(t(pt)),1],t(pt)[nrow(t(pt)),2])))
```

```
## [1] "El valor de E(u,v) obtenido es:  0.139813791996153"
```

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

Aunque vemos que la trayectoria seguida por este algoritmo tiene sentido, va peor que gradiente descendente, pues avanza muy lento y le costaría converger tan bien como gradiente descendente.

Analíticamente, GD se centra en minimizar las dos variables a la vez mientras que CD no, es decir, se centra en minimizar primero una variable y luego la otra, por lo que en casos concretos podría llegar a ser mejor que GD, pero no así en términos generales.

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

El método de Newton introduce el uso de la matriz Hessiana, que no es más que la matriz definida por cuatro elementos:

$$H = \begin{pmatrix} \frac{\partial^2 f}{\partial x x} & \frac{\partial^2 f}{\partial x y} \\ \frac{\partial^2 f}{\partial y x} & \frac{\partial^2 f}{\partial y y} \end{pmatrix}$$

Hablamos de las segundas derivadas de la función que tratamos con respecto a todas las combinaciones de variables.

El Método de Newton consiste en aplicar de forma iterativa la expresión:

$$w(t+1) = -H^{-1} * w(t) * \eta$$

Hasta que el valor de f alcance un umbral o bien alcancemos un máximo de iteraciones.

Observando que los elementos H_{12} y H_{21} deberían ser iguales, sólo debemos calcular tres derivadas. Para ahorrar tiempo utilizamos algún software matemático e implementamos las derivadas:

```
ddfdxx_2<-function(x,y){
  2-8*(pi*pi)*sin(2*pi*y)*sin(2*pi*x)
}

ddfdxy_2<-function(x,y){
  8*(pi*pi)*cos(2*pi*x)*cos(2*pi*y)
}

ddfdyx_2<-function(x,y){
  8*(pi*pi)*cos(2*pi*x)*cos(2*pi*y)
}

ddfdyy_2<-function(x,y){
  4-8*(pi*pi)*sin(2*pi*x)*sin(2*pi*y)
}
```

Vamos también a codificar un método que nos compone la matriz Hessiana y que utilizaremos dentro del algoritmo del método de Newton:

```
m_hessiana<-function(punto,ddxx,ddxy,ddyx,ddyy){
  matrix(
    c(
      ddxx(punto[1,1],punto[2,1]),
      ddxy(punto[1,1],punto[2,1]),
      ddyx(punto[1,1],punto[2,1]),
```

```
      ddyy(punto[1,1],punto[2,1])
    ),nrow=2,ncol=2,byrow=TRUE)
}
```

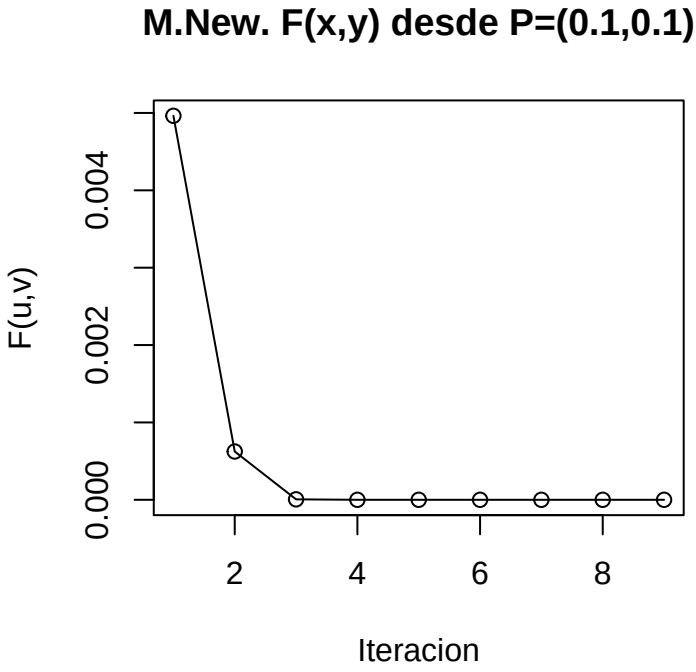
Por último, implementamos el método de Newton:

```
metodo_newton<-function(punto_comienzo,f,dx,dy,ddxx,ddxy,ddyx,ddyy,umbral,max_iter,tasa_apr){
  cont_iter<-0
  w<-punto_comienzo
  valor_f<-f(w[1,1],w[2,1])
  points_mat=matrix(punto_comienzo,nrow=2)
  while(cont_iter<max_iter && valor_f>umbral){
    Hinv<-solve(m_hessiana(w,ddxx,ddxy,ddyx,ddyy))
    der_x<-dx(w[1,1],w[2,1])
    der_y<-dy(w[1,1],w[2,1])
    w<- -Hinv%*%matrix( c(der_x,der_y),nrow=2 )*tasa_apr
    valor_f<-f(w[1,1],w[2,1])
    cont_iter<-cont_iter+1
    #print(paste("Iteracion ",cont_iter,": ",w[1,1],",",w[2,1],",",valor_f))
    points_mat<-cbind(points_mat,w)
  }
  points_mat
}
```

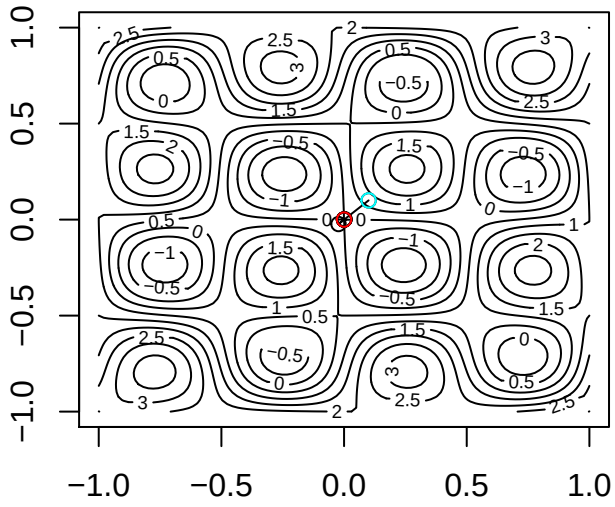
a) Generar un gráfico de como desciende el valor de la función con las iteraciones.

Para punto P=(0.1,0.1)

```
metodo_newton(
  matrix(c(0.1,0.1)),f_2,dfdx_2,dfdy_2,ddfdxx_2,ddfdxy_2,ddfdyx_2,ddfdyy_2,as.double(1e-14),1000,0.1
)->mn_1

plot(f(mn_1[1,],mn_1[2,]),type="o",xlab="Iteracion",ylab="F(u,v)",
     main="M.New. F(x,y) desde P=(0.1,0.1)")
```

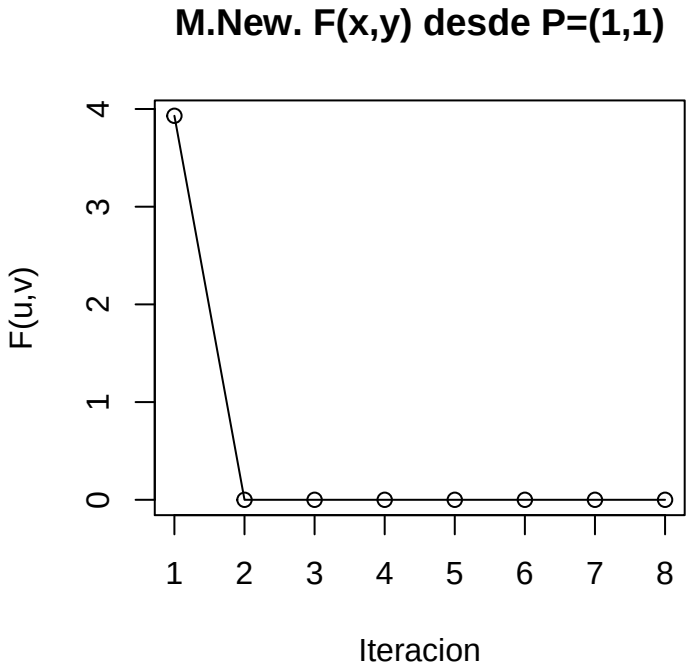



```
pintar_contorno(-1,1,-1,1,100,f_2,mn_1)
```

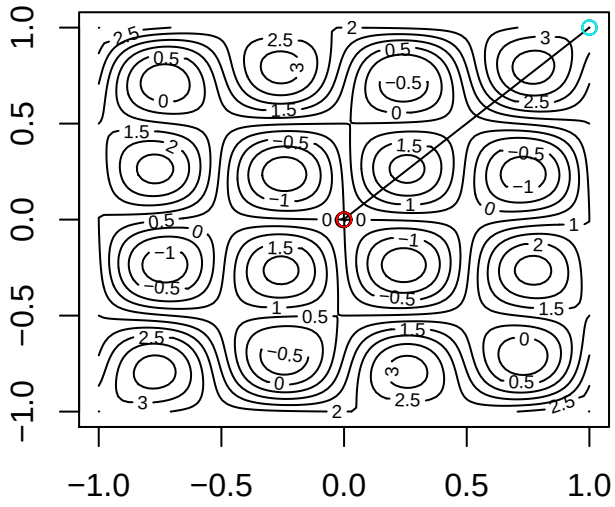


Para punto P=(1,1)

```
#Con (1, 1), el Metodo de Newton alcanza el minimo en 57 iteraciones..
metodo_newton(
  matrix(c(1,1)),f_2,dfdx_2,dfdy_2,ddfdxx_2,ddfdxy_2,ddfdyx_2,ddfdyy_2,as.double(1e-14),100,0.1
)->mn_2

plot(f(mn_2[1,],mn_2[2,]),type="o",xlab="Iteracion",ylab="F(u,v)",
     main="M.New. F(x,y) desde P=(1,1)")
```

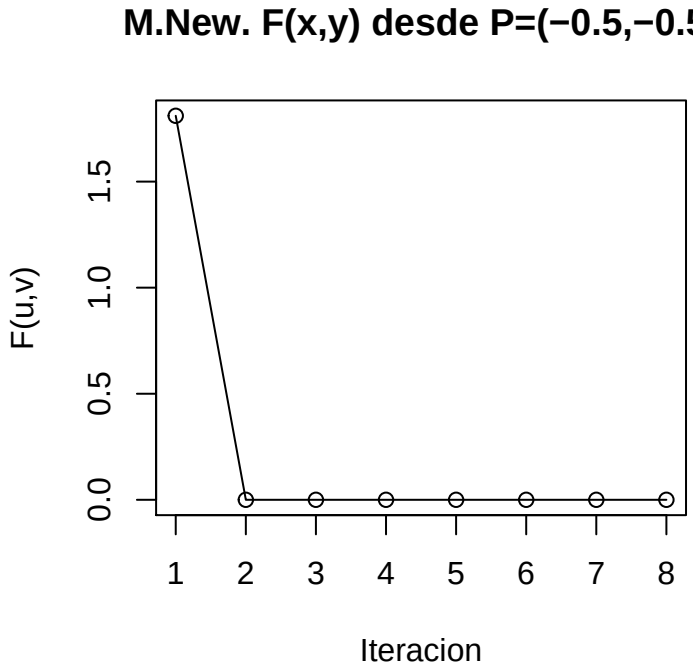



```
pintar_contorno(-1,1,-1,1,100,f_2,mn_2)
```

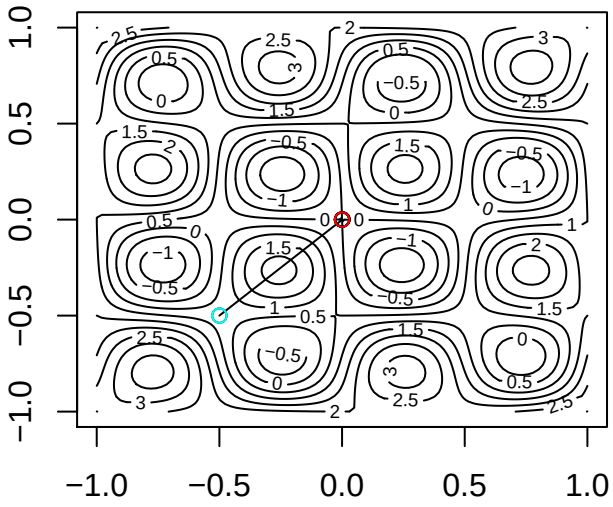


Para punto P=(-0.5,-0.5)

```
#Con (-0,5, -0,5), el Metodo de Newton alcanza el minimo en 76 iteraciones.
metodo_newton(
  matrix(c(-0.5,-0.5)),f_2,dfdx_2,dfdy_2,ddfdxx_2,ddfdxy_2,ddfdyx_2,ddfdyy_2,as.double(1e-14),100,0.1
)->mn_3

plot(f(mn_3[1,],mn_3[2,]),type="o",xlab="Iteracion",ylab="F(u,v)",
     main="M.New. F(x,y) desde P=(-0.5,-0.5)")
```

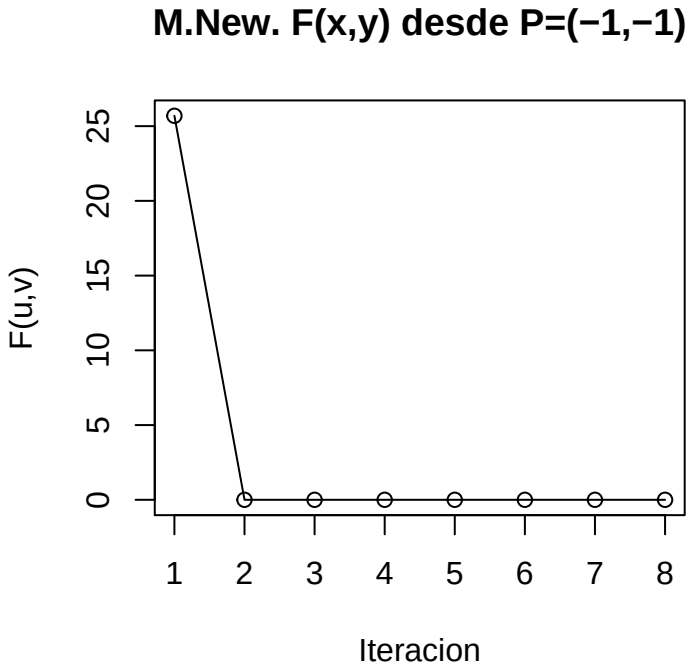



```
pintar_contorno(-1,1,-1,1,100,f_2,mn_3)
```

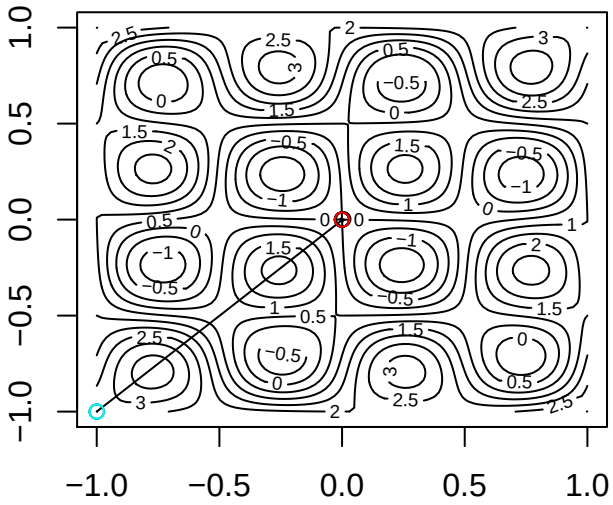


Para punto P=(-1,-1)

```
#Con (-1, -1), el Metodo de Newton alcanza el minimo en 57 iteraciones.
metodo_newton(
  matrix(c(-1,-1)),f_2,dfdx_2,dfdy_2,ddfdxx_2,ddfdxy_2,ddfdyx_2,ddfdyy_2,as.double(1e-14),100,0.1
)->mn_4

plot(f(mn_4[1,],mn_4[2,]),type="o",xlab="Iteracion",ylab="F(u,v)",
     main="M.New. F(x,y) desde P=(-1,-1)")
```




```
pintar_contorno(-1,1,-1,1,100,f_2,mn_4)
```



Como vemos, el Método de Newton siempre llega a un punto de silla, además de forma directa. No puede moverse de ahí ya que, los dos puntos que bajan tiran de "la bolita" con la misma fuerza así que se queda ahí, podríamos entonces considerar que un punto de silla es un mínimo local.

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

El método de Newton busca el punto donde la función es cero, no dónde la función se hace mínima. En estas condiciones, si el mínimo de una función es 0, el Método de Newton es el algoritmo indicado. Cuando las funciones tienen mínimos menores que 0, el Método de Newton no funciona, pues el mínimo no hace cero la función.

El algoritmo de gradiente descendente es independiente de que la función sea 0 o no, por lo que alcanzará mínimos locales siempre que se fije un umbral adecuado.

En términos generales, es más adecuado el algoritmo de gradiente descendente ya que buscamos mínimos de funciones independientemente del valor de la función en dicho mínimo.

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

Consideremos d = 2 para que los datos sean visualizables, y sea $X = [-1, 1]x[-1, 1]$ con probabilidad uniforme de elegir cada $x \in X$. Elegir una línea en el plano como la frontera entre f (x) = 1 (donde y toma valores +1) y f (x) = 0 (donde y toma valores -1), para ello seleccionar dos puntos aleatorios del plano y calcular la línea que pasa por ambos.

Seleccionar $N = 100$ puntos aleatorios $\{x_n\}$ de X y evaluar las respuestas de todos ellos $\{y_n\}$ respecto de la frontera elegida.

Para implementar y probar el algoritmo de Regresión logística vamos a necesitar algunas funciones de la práctica 1. Vamos a necesitar la función de generar puntos en un rango con probabilidad uniforme (simula_unif), simular una recta que corta aleatoriamente un rango (simula_recta)

```
set.seed(7)
simula_unif=function(N,dim,rango){
  lapply(seq_len(N),function(i) runif(dim,rango[1],rango[2]));
}


simula_recta<-function(intervalo){
  x=runif(2,intervalo[1],intervalo[2])
  y=runif(2,intervalo[1],intervalo[2])
  #plot(x,y) Para asegurarse de que la recrta pasa por los puntos generados
  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)
}
```

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

Inicializar el vector de pesos con valores 0.

Parar el algoritmo cuando ||w (t-1) - w (t) || < 0,01, donde w (t) denota el vector de pesos al final de la época t. Una época es un pase completo a través de los N datos.

Aplicar una permutación aleatoria de 1, 2, . . . , N a los datos antes de usarlos en cada época del algoritmo.

Usar una tasa de aprendizaje de $\eta = 0,01$

Lo novedoso será la función sigmoide (escalón suave) que utilizará el algoritmo de Regresión logísitica y el propio algoritmo. Es la siguiente:

```
sigmoide<-function(x){
  1/(1+exp(-x))
}
```

Vamos a revisar todas las condiciones que se nos imponen para implementar Regresión Logísitca. La más extraña quizás es utilizar Gradiente Descendente Estocástico. Utilizar SGD (o GDE) no es más que corregir el error considerando solo una muestra de la muestra, esto es, si consideramos clasificar una muestra (global) de puntos, vamos a calcular el error con respecto de la función objetivo con sólo una muestra aleatoria elegida de dentro de la muestra global, esto es representado mediante las variables `datos_estocast` y `label_estocast`.

Regresión logística consiste en aplicar iterativamente la siguiente expresión:

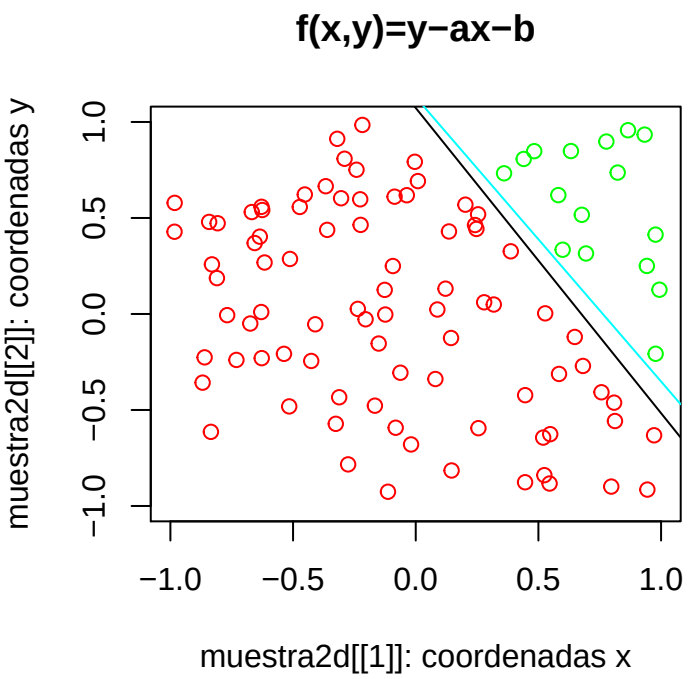
$$w(t+1) = w(t) - \eta(\frac{1}{N}\Sigma_{i=1}^{N}(-y_i x_i \sigma(-y_i w^T x_i)))$$

Hasta que alcancemos un máximo de iteraciones o bien la norma vectorial de la diferencia entre w(t) y w(t+1) sea menor que un determinado valor, en este caso, 0.01, es decir: $||w(t) - w(t+1)|| < 0.01$.

```
regresion_logistica<-function(datos,label,max_iter,vini,tasa_apr,umbral,perc){
  cont_iter<-0
  w<-vini
  datos<-cbind(datos,rep(1,nrow(datos)))
  dif<-1
  while(cont_iter<=max_iter && dif>umbral){
    i_estocast<-sample(1:nrow(datos),trunc(perc*nrow(datos)))
    datos_estocast<-matrix(datos[i_estocast,],nrow=length(i_estocast))
    label_estocast<-label[i_estocast]
    #gw <- colSums(-label_estocast *
    #                as.matrix(datos_estocast) *
    #              c(sigmoide( -label_estocast * (t(as.matrix(datos_estocast))%*%w))))
    gw<-0
    for(i in 1:nrow(datos_estocast)){
      gw<- gw+(-label_estocast[i] * datos_estocast[i,] *
        #Debe ser negativo para anular el - de la sigmoide
        sigmoide( -label_estocast[i] * (t(w)%*%datos_estocast[i,])))
    }

    #gw <- (- gw / nrow(as.matrix(datos_estocast)))
    gw <- (gw / nrow(as.matrix(datos_estocast)))

    dif<-norm(w-(w-gw*tasa_apr),type="2")
    w=w - gw*tasa_apr
    cont_iter<-cont_iter+1
  }
  c(w,cont_iter)
}
```

Vamos ahora a ponerlo en funcionamiento. Generamos una muestra, la etiquetamos y aplicamos regresión logística con una tasa de aprendizaje de 0.01. Daremos al algoritmo un máximo de 10000 iteraciones.

```
rango=c(-1,1)

muestra2d<-matrix(unlist(simula_unif(2,100,c(rango[1],rango[2]))),ncol=2,byrow=FALSE);

vars<-simula_recta(c(rango[1],rango[2]))
signos <- sign(muestra2d[,2] - vars[5]*muestra2d[,1] - vars[6]);

plot(NA,xlim=c(rango[1],rango[2]),ylim=c(rango[1],rango[2]),xlab='muestra2d[[1]]: coordenadas x',ylab='

abline(vars[6],vars[5]);

points(muestra2d[,1][signos>=0],muestra2d[,2][signos>=0],col='green');
points(muestra2d[,1][signos<0],muestra2d[,2][signos<0],col='red');

sol<-regresion_logistica(muestra2d,signos,10000,matrix(c(0,0,0),nrow=3),0.1,0.00001,0.1)

abline(-(sol[3]/sol[2]),-(sol[1]/sol[2]),col="cyan")
```



Como vemos, no es una buena aproximación. Se demuestra empíricamente que con una tasa de aprendizaje neutra ($\eta = 1$), sí se consiguen buenos resultados.

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

Vamos a definir la funcion de error dentro de la muestra, que depende de los datos, de los pesos calculados por el algoritmo de regresión logística y de las etiquetas. Es calculado con la siguiente expresión:

$$E_{in}(w) = \frac{1}{N}\Sigma_{n=1}^{N} \log(1 + e^{-y_n w^T x_n})$$

```
Ein_Reg_Log<-function(datos,w,label){
  datos<-cbind(datos,rep(1,nrow(datos)))
  sum<-0
  for(i in 1:nrow(datos)){
    sum<-sum+ log(1+exp(sign(-label[i] * t(w) %*% datos[i,])))
  }
  sum<-sum/nrow(datos)
  sum
}
```

Generamos una nueva muestra de test y calculamos los errores con respecto a la muestra y con respecto al conjunto de test:

```
rango=c(-1,1)

test<-matrix(unlist(simula_unif(2,1000,c(rango[1],rango[2]))),ncol=2,byrow=FALSE);
signostest <- sign(test[,2] - vars[5]*test[,1] - vars[6]);

E_in<-Ein_Reg_Log(muestra2d,matrix(c(sol[1],sol[2],sol[3]),ncol=1),signos)
E_test<-Ein_Reg_Log(test,matrix(c(sol[1],sol[2],sol[3]),ncol=1),signostest)

print(paste("Error en la muestra: ",E_in))
```

```
## [1] "Error en la muestra:  0.313261687518224"
```

```
print(paste("Error en test: ",E_test))
```

```
## [1] "Error en test:  0.337261687518223"
```

Vemos como el error es elevadísimo, pues el algoritmo de regresión logística no es para nada bueno con esa tasa de aprendizaje.

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

Para satisfacer todo lo que se nos pide en este ejercicio, utilizaremos este bloque de código, que no es más que hacer 100 veces el experimento anterior.

```
promedio_errorIN<-0
promedio_errorOUT<-0
promedio_epocas<-0
for(i in 1:100){
  #Generar una muestra de 100 pts
  rango=c(-1,1)
  vars<-simula_recta(c(rango[1],rango[2]))

  muestra2d<-matrix(unlist(simula_unif(2,100,c(rango[1],rango[2]))),ncol=2,byrow=FALSE);
  signos <- sign(muestra2d[,2] - vars[5]*muestra2d[,1] - vars[6]);
  sol<-regresion_logistica(muestra2d,signos,1000,matrix(c(0,0,0),nrow=3),0.1,0.00001,0.1)

  #Generar un test
```

```
  test<-matrix(unlist(simula_unif(2,100,c(rango[1],rango[2]))),ncol=2,byrow=FALSE);
  signostest <- sign(test[,2] - vars[5]*test[,1] - vars[6]);

  promedio_errorIN<-promedio_errorIN+Ein_Reg_Log(muestra2d,
                                                 matrix(c(sol[1],sol[2],sol[3]),nrow=3),
                                                 signos)
  promedio_errorOUT<-promedio_errorOUT+Ein_Reg_Log(test,
                                                   matrix(c(sol[1],sol[2],sol[3]),nrow=3),
                                                   signostest)
  promedio_epocas<-promedio_epocas+sol[length(sol)]
}
promedio_errorIN<-promedio_errorIN/100
promedio_errorOUT<-promedio_errorOUT/100
promedio_epocas<-promedio_epocas/100
```

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

```
print(paste("El promedio de error en la muestra es: ",promedio_errorIN))
```

```
## [1] "El promedio de error en la muestra es:  0.332461687518223"
```

```
print(paste("El promedio de error fuera de la muestra es: ",promedio_errorOUT))
```

```
## [1] "El promedio de error fuera de la muestra es:  0.341761687518223"
```

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

```
print(paste("El promedio de epocas que se tarda en converger: ",promedio_epocas,"."))
```

```
## [1] "El promedio de epocas que se tarda en converger:  1001 ."
```

Para demostrar empíricamente que la tasa de aprendizaje $\eta = 0.01$ no es buena, repetiré voluntariamente el experimento para $\eta = 1$, obtenemos los siguientes resultados:

```
## [1] "El promedio de error en la muestra es:  0.322861687518224"
```

```
## [1] "El promedio de error fuera de la muestra es:  0.330761687518223"
```

```
## [1] "El promedio de epocas que se tarda en converger:  1000.99 ."
```

5. Clasificación de Dígitos. Considerar el conjunto de datos de los dígitos manuscritos y seleccionar las muestras de los dígitos 1 y 5. Usar los ficheros de entrenamiento (training) y test que se proporcionan. Extraer las características de intensidad promedio y simetría en la manera que se indicó en el ejercicio 3 del trabajo 1. Plantear un problema de clasificación binaria que considere el conjunto de entrenamiento como datos de entrada para aprender la función g. Usando el modelo de Regresión Lineal para clasificación seguido por PLA-Pocket como mejora. Responder a las siguientes cuestiones.

Vamos a realizar la lectura del conjunto de entrenamiento y calcularemos sus valores de intensidad y simetria.

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

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

```
digitos15.train = digit.train[digit.train$V1==1 | digit.train$V1==5,]
digitos.train = digitos15.train[,1]
ndigitos.train = nrow(digitos15.train)


grises.train = array(unlist(subset(digitos15.train,select=-V1)),c(ndigitos.train,16,16))
rm(digit.train)
rm(digitos15.train)

fsimetria1 <- function(A){
  A <- abs(A-A[,nrow(A):1])
  -sum(A)
}

intensidad.train = apply(grises.train[1:ndigitos.train,,],1, mean)
simetria1.train = apply(grises.train[1:ndigitos.train,,],1,fsimetria1)
datos1.train = as.matrix(cbind(intensidad.train,simetria1.train))
```

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

A partir de aquí estarían leídos los datos. Ahora vamos a rescatar de la práctica 1 el algoritmo de regresión lineal:

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

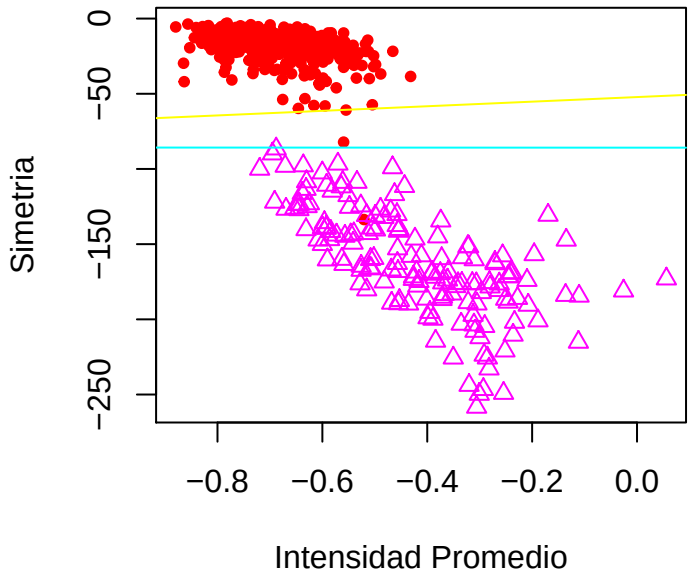
Vamos a rescatar también el algoritmo PLA_POCKET de la práctica anterior:

```
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)
}
```

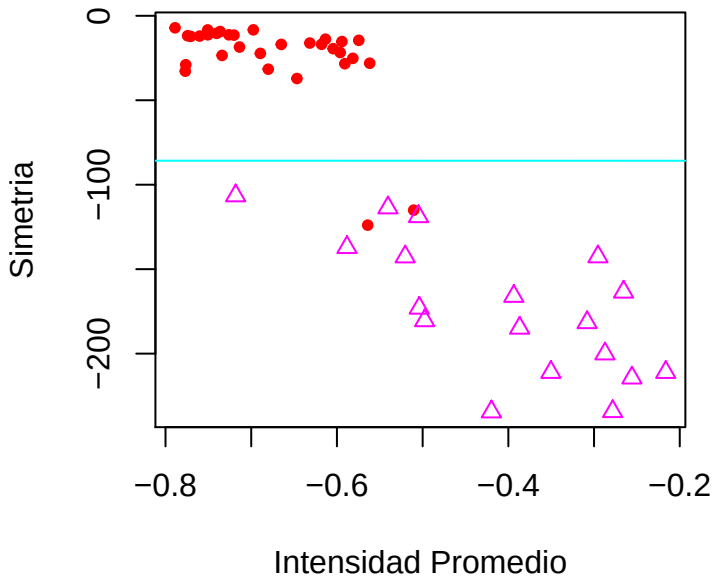
Lo primero que haremos será aplicar regresión lineal, que nos hará una primera aproximación a la separación de los datos. Aunque no es demasiado bueno (amarillo). A partir de lo devuelto por regresión lineal, daremos una vueltecita de tuerca más con el algoritmo PLA_Pocket (cyan), que ya nos dará una separación mejor. Este es el resultado de ambos algoritmos.

```
plot(datos1.train,xlab="Intensidad Promedio",ylab="Simetria",col=digitos.train+1,pch=digitos.train+19)
pesos<-Regress_Lin(datos1.train,digitos.train)
abline(-pesos[3]/pesos[2],pesos[1]/pesos[2],col="yellow")
digitos.train[digitos.train==5]<--1
pesos<-PLA_pocket(datos1.train,digitos.train,10,pesos)
abline(-pesos[3]/pesos[2],pesos[1]/pesos[2],col="cyan")
```



Ahora, con la función estimada por la combinación de RL + PLA pocket evaluaremos el conjunto de test:

```
digit.test<-read.csv("ZipDigits/zip.test", sep=' ', header=FALSE, stringsAsFactors = FALSE)
digitos15.test<-as.data.frame(data.matrix( digit.test[digit.test[,1]==1 | digit.test[,1]==5, 1:257]))

digitos.test = digitos15.test[,1]
ndigitos.test = nrow(digitos15.test)

grises.test = array(unlist(subset(digitos15.test,select=-V1)),c(ndigitos.test,16,16))
rm(digit.test)
rm(digitos15.test)

intensidad.test = apply(grises.test[1:ndigitos.test,,],1, mean)
simetria1.test = apply(grises.test[1:ndigitos.test,,],1,fsimetria1)
datos1.test = as.matrix(cbind(intensidad.test,simetria1.test))
plot(datos1.test,xlab="Intensidad Promedio",ylab="Simetria",col=digitos.test+1,pch=digitos.test+19)


abline(-pesos[3]/pesos[2],pesos[1]/pesos[2],col="cyan")
```



Vemos cómo la función estimada por RL + PLA pocket es un buen estimador para el conjunto de test.

b) Calcular E in y E test (error sobre los datos de test).

El error lo vamos a calcular como el porcentaje de puntos mal clasificados en cada conjunto: entrenamiento y test. Lo primero que hacemos es crear la función que nos lo calcule, al que le pasaremos la expresión (usando los pesos calculados) y los signos originales para comparar:

```
Error_clasificacion<-function(func_evaluada,conj_y,signos_originales){
  signos_estimados<-as.integer(func_evaluada>=conj_y)
  signos_estimados[signos_estimados==0]<--1
  length(which(signos_originales!=signos_estimados))/length(signos_originales)
}
```

Procedemos entonces a calcular los errores en entrenamiento y test:

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

E_in<-Error_clasificacion(pesos[1]*datos1.train[,1]+pesos[2]*datos1.train[,2]+pesos[3],datos1.train[,2]
                          digitos.train)

E_test<-Error_clasificacion(pesos[1]*datos1.test[,1]+pesos[2]*datos1.test[,2]+pesos[3],datos1.test[,2],
                           digitos.test)

print(paste("Error en la muestra: ",E_in))
```

```
## [1] "Error en la muestra:  0.00166944908180301"
```

```
print(paste("Error en test: ",E_test))
```

```
## [1] "Error en test:  0.0408163265306122"
```

Lógicamente, el error fuera de la muestra es mayor que dentro. Si la diferencia fuese abismal, la razón, probablemente sea que hemos sobreajustado la muestra.

c) Obtener cotas sobre el verdadero valor de E_{out} . Pueden calcularse dos cotas una basada en E_{in} y otra basada en E_{test} . Usar una tolerancia $\delta = 0,05$. ¿Que cota es mejor?

Ya sabemos cómo se calculan las cotas de generalización teóricas, vamos a hacer una función para que sea más cómodo. La dimensión VC de los datos son el tamaño de la entrada (2: x e y) más uno, luego la dimensión VC para estos datos es igual a 3.

```
generalization_bound<-function(delta,dvc,N,E_in){
  E_in + sqrt(dvc*log(N)/N)
}

cotaE_in<-generalization_bound(0.05,ncol(datos1.train)+1,nrow(datos1.train),E_in)
cotaE_test<-generalization_bound(0.05,ncol(datos1.test)+1,nrow(datos1.test),E_test)

print(paste("Cota de generalizacion para E_in: ",cotaE_in))
```

```
## [1] "Cota de generalizacion para E_in:  0.180637856373681"
```

```
print(paste("Cota de generalizacion para E_test: ",cotaE_test))
```

```
## [1] "Cota de generalizacion para E_test:  0.528950234520236"
```

Hemos calculado un error para un conjunto de test del que no hemos visto nada, ese valor es una estimación de un error fuera de la muestra. Aunque no deja de ser una muestra, no está tan manipulada como la muestra que hemos utilizado entrenamiento, pues no la hemos aprendido, por lo que la cota que obtengamos con el conjunto de test será más realista que la que obtengamos con el conjunto de entrenamiento.

d) OPCIONAL Repetir los puntos anteriores pero usando una transformación polinómica de tercer orden ($\phi_3(x)$ en las transparencias de teoría).

Vamos a transformar los conjuntos de entrenamiento y test con ϕ_3 y calcularemos los pesos asociados (ajustar el modelo).

```
puntos.train<-datos1.train

puntos.train<-cbind(puntos.train,puntos.train[,1]*puntos.train[,1])
puntos.train<-cbind(puntos.train,puntos.train[,1]*puntos.train[,2])
puntos.train<-cbind(puntos.train,puntos.train[,2]*puntos.train[,2])

puntos.train<-cbind(puntos.train,puntos.train[,1]*puntos.train[,1]*puntos.train[,1])
puntos.train<-cbind(puntos.train,puntos.train[,1]*puntos.train[,1]*puntos.train[,2])
puntos.train<-cbind(puntos.train,puntos.train[,1]*puntos.train[,2]*puntos.train[,2])
puntos.train<-cbind(puntos.train,puntos.train[,2]*puntos.train[,2]*puntos.train[,2])


puntos.test<-datos1.test

puntos.test<-cbind(puntos.test,puntos.test[,1]*puntos.test[,1])
puntos.test<-cbind(puntos.test,puntos.test[,1]*puntos.test[,2])
puntos.test<-cbind(puntos.test,puntos.test[,2]*puntos.test[,2])

puntos.test<-cbind(puntos.test,puntos.test[,1]*puntos.test[,1]*puntos.test[,1])
puntos.test<-cbind(puntos.test,puntos.test[,1]*puntos.test[,1]*puntos.test[,2])
```

```
puntos.test<-cbind(puntos.test,puntos.test[,1]*puntos.test[,2]*puntos.test[,2])
puntos.test<-cbind(puntos.test,puntos.test[,2]*puntos.test[,2]*puntos.test[,2])

labels.test<-digitos.test
labels.test[labels.test==5]=-1

labels.train<-digitos.train
labels.train[labels.train==-1]=5
sol_rl<-Regress_Lin(puntos.train,labels.train)
labels.train[labels.train==5]=-1
sol_pp<-PLA_pocket(puntos.train,labels.train,10,sol_rl)
```

Ahora vamos a comparar los errores dentro de la muestra, en test y las cotas de generalización con los pesos obtenidos.

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

E_in_t_polin <- Error_clasificacion(
  sol_pp[1]*puntos.train[,1]+sol_pp[2]*puntos.train[,2]+sol_pp[3]*puntos.train[,3]+
  sol_pp[4]*puntos.train[,4]+sol_pp[5]*puntos.train[,5]+sol_pp[6]*puntos.train[,6]+
  sol_pp[7]*puntos.train[,7]+sol_pp[8]*puntos.train[,8]+sol_pp[9]*+sol_pp[10],
  datos1.train[,2],labels.train
  )

E_test_t_polin <- Error_clasificacion(
  sol_pp[1]*puntos.test[,1]+sol_pp[2]*puntos.test[,2]+sol_pp[3]*puntos.test[,3]+
  sol_pp[4]*puntos.test[,4]+sol_pp[5]*puntos.test[,5]+sol_pp[6]*puntos.test[,6]+
  sol_pp[7]*puntos.test[,7]+sol_pp[8]*puntos.test[,8]+sol_pp[9]*+sol_pp[10],
  datos1.test[,2],labels.test
  )

E_in_lineal<-Error_clasificacion(pesos[1]*datos1.train[,1]+pesos[2]*datos1.train[,2]+pesos[3],
                                 datos1.train[,2],
                                 digitos.train)


E_test_lineal<-Error_clasificacion(pesos[1]*datos1.test[,1]+pesos[2]*datos1.test[,2]+pesos[3],
                                  datos1.test[,2],
                                  digitos.test)


cotaE_in_t_polin<-generalization_bound(0.05,ncol(puntos.train)+1,nrow(puntos.train),E_in_t_polin)
cotaE_test_t_polin<-generalization_bound(0.05,ncol(puntos.test)+1,nrow(puntos.test),E_test_t_polin)

cotaE_in_lineal<-generalization_bound(0.05,3,nrow(datos1.train),E_in_lineal)
cotaE_test_lineal<-generalization_bound(0.05,3,nrow(datos1.test),E_test_lineal)

print("Errores dentro de muestra: ")
```

```
## [1] "Errores dentro de muestra: "
```

```r
print(paste("Modelo lineal: ",E_in_lineal))
```

```
## [1] "Modelo lineal:  0.00166944908180301"
```

```r
print(paste("Transformacion polinomica: ",E_in_t_polin))
```

```
## [1] "Transformacion polinomica:  0.262103505843072"
```

```r
print("Errores en test: ")
```

```
## [1] "Errores en test: "
```

```r
print(paste("Modelo lineal: ",E_test_lineal))
```

```
## [1] "Modelo lineal:  0.0408163265306122"
```

```r
print(paste("Transformacion polinomica: ",E_test_t_polin))
```

```
## [1] "Transformacion polinomica:  0.36734693877551"
```

```r
print("Cotas de generalizacion: ")
```

```
## [1] "Cotas de generalizacion: "
```

```r
print(paste("Modelo lineal (en training): ",cotaE_in_lineal))
```

```
## [1] "Modelo lineal (en training):  0.180637856373681"
```

```r
print(paste("Transformacion polinomica (en training): ",cotaE_in_t_polin))
```

```
## [1] "Transformacion polinomica (en training):  0.588853618358184"
```

```r
print(paste("Modelo lineal (en test): ",cotaE_test_lineal))
```

```
## [1] "Modelo lineal (en test):  0.528950234520236"
```

```r
print(paste("Transformacion polinomica (en test): ",cotaE_test_t_polin))
```

```
## [1] "Transformacion polinomica (en test):  1.2585534470724"
```

Para empezar, los errores dentro de la muestra son mayores en el caso del modelo polinómico, pero eso no nos dice nada, pues lo que buscamos es minimizar E_{out}. En conjunto de test vemos cómo el modelo lineal gana por goleada a la transformación polinómica, a pesar de su simplicidad, esto ya si es un indicativo que nos haría decantarnos por el modelo lineal. Por último las cotas de generalización nos están diciendo que si elegimos el modelo lineal, el error fuera de la muestra está acotado (menor al 100%), mientras que si elegimos el modelo de transformación polinómica, el error fuera de la muestra se dispara a mayor que 1, por lo que no se nos está acotando nada. Vemos entonces cómo un modelo lineal es más robusto y estable que un modelo de transformación polinómica.

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

Como podemos observar en el apartado anterior, al comparar el modelo lineal con el polinómico, procedente de la transformación $\phi_3(x) = (x_1, x_2, x_1^2, x_1x_2, x_2^2, x_1^3, x_1^2x_2, x_1x_2^2, x_2^3)$, el polinomico no es rival para el lineal. Es evidente que la simplicidad y la robustez del modelo lineal es mayor que la del polinomio, además, por lo general, el porcentaje de acierto también.

Sin ninguna duda, la transformación no merece la pena y está totalmente justificada la teoría de la navaja de Occam en este caso, pues vemos cómo un modelo simple, en este caso, supera a un modelo más complejo. Aunque no siempre es así, siempre debemos probar un modelo lineal antes que un modelo más complejo.

Por supuesto, en este caso, si tuviera que dar los resultados a un potencial cliente no usaría la transformación polinómica, pues la ganancia en resultados (si es que la hubiera) es prácticamente nula con respecto al modelo lineal, sin embargo, la sobrecarga en cálculo y complejidad si crece.

Sobreajuste

1. Vamos a construir un entorno que nos permita experimentar con los problemas de sobreajuste. Consideremos el espacio de entrada $X = [-1, 1]$ con una densidad de probabilidad uniorme, $P(x) = \frac{1}{2}$. Considereamos dos modelos H_1 y H_{10} representando el conjunto de todos los polinomios de grado 2 y grado 10 respectivamente. La función objetivo es un polinomio de grado Q_f que escribimos como $f(x) = \Sigma_{q=0}^{Q_f} a_q L_q(x)$, donde $L_q(x)$ son los polinomios de Legendre. El conjunto de datos es $D = \{(x_1, y_1), ..., (x_N, y_N)\}$ donde $y_n = f(x_n) + \sigma\varepsilon_n$ ylas $\{\varepsilon_n\}$ son variables aleatorias i.i.d $N(0, 1)$ y σ^2 la varianza del ruido.

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

Generamos los coeficientes a_q a partir de muestras de una distribución $N(0, 1)$ y escalamos dichos coeficientes de manera que $E[f^2] = 1$.

Como sabemos, para H_2 y para H_{10} habrá que generar 3 coeficientes (del 0 al 2) y 11 coeficientes (del 0 al 10). Para ello usamos `rnorm` que por defecto genera valores en una distribución normal de media (μ) 0 y desviación estándar (σ) 1. Para que la esperanza matemática de f^2 sea igual a 1, lo único que tenemos que hacer es dividir cada coeficiente por la expresión $\sqrt{\Sigma_{q=0}^{Q_f} \frac{1}{2q+1}}$

```
generate_Lg_coef<-function(qf){
  coef<-rnorm(n=qf+1,mean=0,sd=1)
  grados<-seq(0,qf,length.out = qf+1)
  div<-sqrt(sum(1/(2*grados+1)))
  coef<-coef/div
  coef
}

coefs_objetivo<-generate_Lg_coef(6)
```

Generamos un conjunto de datos $x_1, ..., x_N$, muestreando de forma independiente $P(x)$ y los valores $y_n = f(x_n) + \sigma_{\varepsilon_n}$.

Vamos a generar el conjunto de datos dentro del espacio de entrada $X = [-1, 1]$ con una distribución uniforme.

```
conj_datos_x<-runif(10,min=-1,max=1)
```

Vamos a codificar nuestra función objetivo de forma genérica. Dependiendo del tamaño de los coeficientes (grados del 0 a N del polinomio de Legendre), generaremos N-1 polinomios de Legendre, que iremos multiplicando por los coeficientes generados. Dado que nos interesa no solo el resultado de la evaluación de la función objetivo, devolveremos la expresión, que podremos evaluar gracias a `as.function` de `R`.

Para generar los polinomios de Legendre será necesario instalar los paquetes `polinom` y `orthopolinom`, esto se hace con las líneas: `install.packages("polynom")` y `install.packages("orthopolynom")` respectivamente.

```
funcion_objetivo_Legendre<-function(coefs){
  polinomios<-orthopolynom::legendre.polynomials(length(coefs)-1, normalized = FALSE)
  sum<-0
  expr<-0
  for(i in 1:length(coefs)){
    polinomios[[i]]<-coefs[i]*polinomios[[i]]
    expr<-expr+polinomios[[i]]
  }
  expr
}
```

Con la siguiente función asignaremos ruido a cada punto.

```
generar_ruido<-function(min,max,y){
  ruido<-c(0)
  for(i in 1:length(y)){
    #ruido<-c(ruido,runif(1,min=(min-y[i]),max=(max-y[i])))
    ruido<-c(ruido,runif(1,min=min,max=max))
  }
  ruido[2:length(ruido)]
}
```

Con estas dos funciones, ya podemos calcular la coordenada Y de los puntos X generados anteriormente.

```
conj_datos_y<-as.function(funcion_objetivo_Legendre(coefs_objetivo))(conj_datos_x)
```

```
## Loading required package: polynom
```

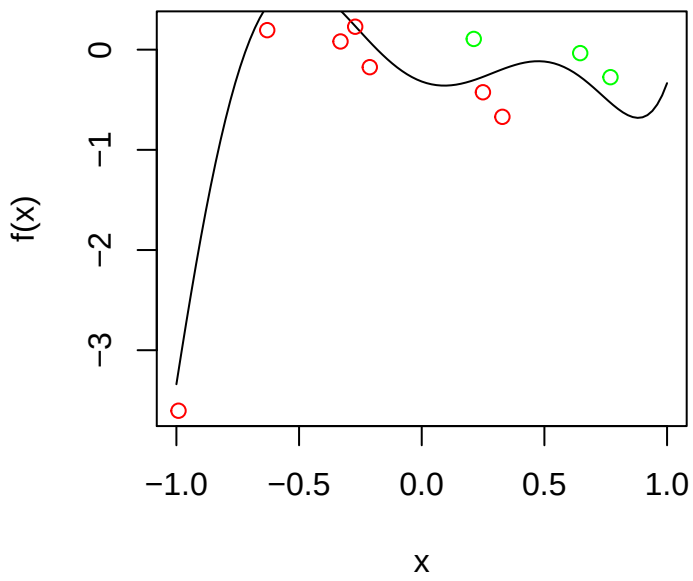
```
conj_datos_y<-conj_datos_y+generar_ruido(-0.5,0.5,conj_datos_y)
puntos<-cbind(conj_datos_x,conj_datos_y)
```

Sean g_2 y g_10 los mejores ajustes a los datos usando H_2 y H_{10} respectivamente, y sean $E_{out}(g_2)$ y $E_{out}(g_{10})$ sus respectivos errores fuera de la muestra.

a)Calcular g_2 y g_{10}

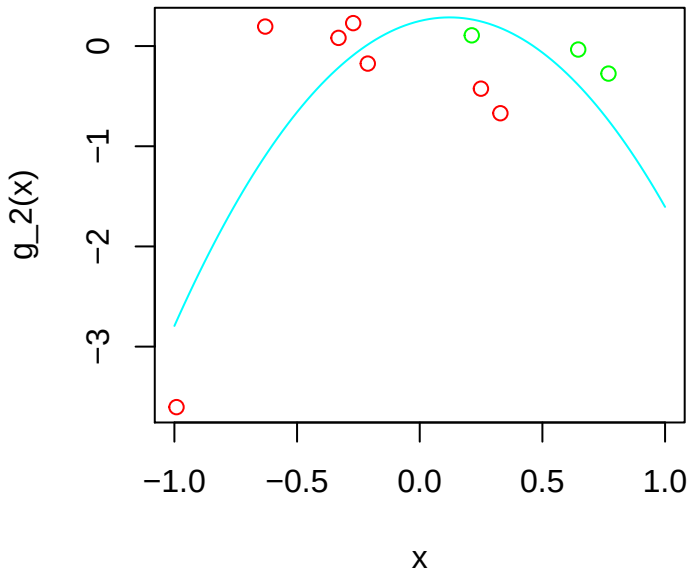
A partir de aquí vamos a ver cómo ajustar las funciones g_2 y g_{10} a la función objetivo. Lo primero que hacemos es mostrar la función objetivo junto con los puntos, que contienen un poco de ruido.

```
#Representacion de la funcion objetivo y los datos clasificados
funcion_objetivo<-function(x){ as.function(funcion_objetivo_Legendre(coefs_objetivo))(x)}
curve(funcion_objetivo,xlim=c(-1,1),ylim=c(min(conj_datos_y),max(conj_datos_y)),ylab="f(x)")

signos<-rep(1,nrow(puntos))
signos[which(puntos[,2]<funcion_objetivo(puntos[,1]))]<-(-1)

points(puntos[,1][signos>=0],puntos[,2][signos>=0],col="green")
points(puntos[,1][signos<0],puntos[,2][signos<0],col="red")
```



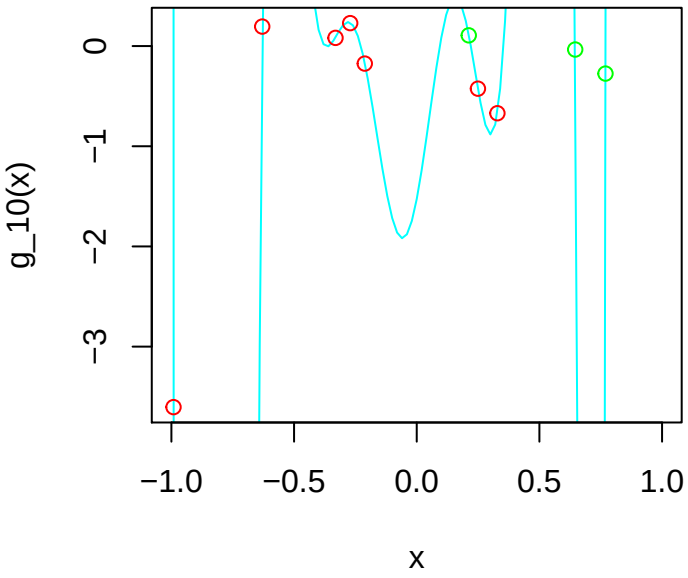
Ahora vamos a ajustar g_2 todo lo que se pueda.

```
datos<-cbind(conj_datos_x*conj_datos_x,conj_datos_x)
labels<-conj_datos_y

coefs_H2<-Regress_Lin(datos,labels)

#Estimo la funcion y la pinto
g_2<-function(x){coefs_H2[1]+coefs_H2[2]*x+coefs_H2[3]*x*x}
curve(g_2,xlim=c(-1,1),ylim=c(min(conj_datos_y),max(conj_datos_y)),col="cyan")
points(puntos[,1][signos>=0],puntos[,2][signos>=0],col="green")
points(puntos[,1][signos<0],puntos[,2][signos<0],col="red")
```



Ajustamos ahora la función g_{10}

```
#Representación de la funcion de H10 con una transformacion polinomica
datos<-cbind(conj_datos_x**10,conj_datos_x**9,conj_datos_x**8,conj_datos_x**7,conj_datos_x**6,
             conj_datos_x**5,conj_datos_x**4,conj_datos_x**3,conj_datos_x*conj_datos_x,conj_datos_x)

labels<-conj_datos_y

coefs_H10<-Regress_Lin(datos,labels)


#Estimo la funcion y la pinto
g_10<-function(x){coefs_H10[1]+coefs_H10[2]*x+coefs_H10[3]*x*x+coefs_H10[4]*x**3+
                  coefs_H10[5]*x**4+coefs_H10[6]*x**5+coefs_H10[7]*x**6+coefs_H10[8]*x**7+
                  coefs_H10[9]*x**8+coefs_H10[10]*x**9+coefs_H10[11]*x**10}

curve(g_10,xlim=c(-1,1),ylim=c(min(conj_datos_y),max(conj_datos_y)),col="cyan")
points(puntos[,1][signos>=0],puntos[,2][signos>=0],col="green")
points(puntos[,1][signos<0],puntos[,2][signos<0],col="red")
```

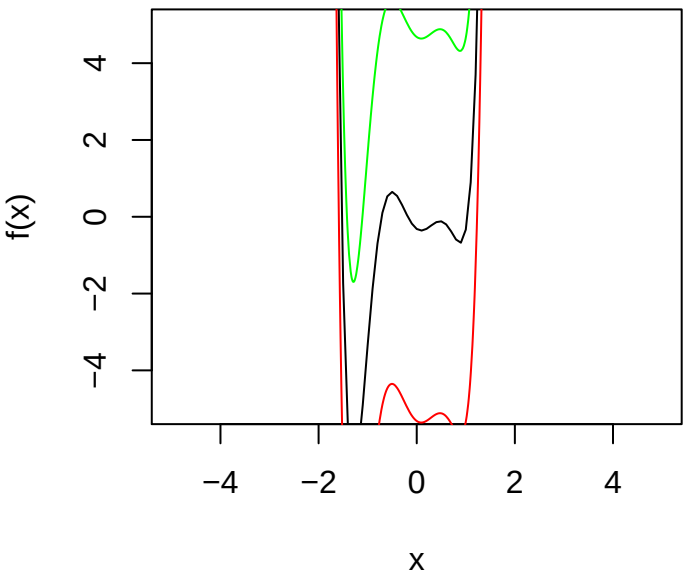


b) ¿Por qué normalizamos f?

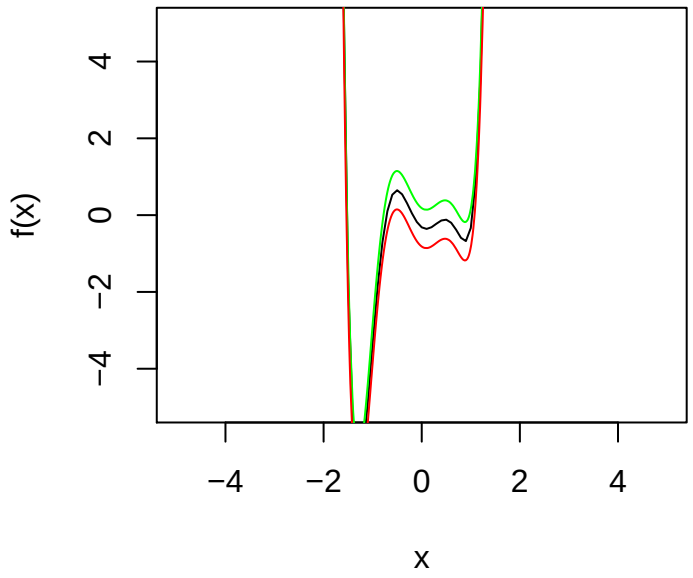
Al normalizar f, los pesos bajan a una escala entre 0 y 1. Dadas las propiedades de los polinomios de Legendre, sabemos que cualquier polinomio puede representarse con los polinomios de Legendre ponderados, es decir, multiplicados por un número entre 0 y 1.

Al hacer esto, acotamos también el ruido, por lo que si los puntos tienen ruido, normalizar f, nos permite acotar también la varianza del ruido. Con el siguiente ejemplo lo vamos a entender mejor.

En el primer gráfico vamos a ver una función, una linea verde y otra roja. El espacio entre las líneas verde y roja es el rango que puede adquirir el ruido si f no ha sido normalizada.



Sin embargo, si normalizamos f, dicho rango se reduce así:



Además, las propiedades de los polinomios de Legendre permiten que cualquier polinomio se pueda representar como una suma de polinomios de Legendre multiplicados por unos coeficientes ponderados (entre 0 y 1), por eso se normalizan los coeficientes, para poder representar cualquier polinomio.

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

$$E_{out}(H_2) = promedio\ sobre\ experimentos(E_{out}(g_2))$$

$$E_{out}(H_{10}) = promedio\ sobre\ experimentos(E_{out}(g_{10}))$$

```
nexperimentos<-100;
promedio_error_g2<-0
promedio_error_g10<-0

promedio_eout_g2<-0
promedio_eout_g10<-0

for(i in 1:nexperimentos){
  ein_g2<-0
  ein_g10<-0
  #Generacion de una funcion objetivo
  coefs_objetivo<-generate_Lg_coef(20)

  #Generar los puntos
  conj_datos_x<-runif(50,min=-1,max=1)
  conj_datos_y<-as.function(funcion_objetivo_Legendre(coefs_objetivo))(conj_datos_x)
```

```
  conj_datos_y<-conj_datos_y+generar_ruido(-0.5,0.5,conj_datos_y)
  puntos<-cbind(conj_datos_x,conj_datos_y)

  #Calculo de los signos
  signos<-rep(1,nrow(puntos))
  signos[which(puntos[,2]<funcion_objetivo(puntos[,1]))]<-(-1)

  #Calcular g_2 y su error
  datos<-cbind(conj_datos_x*conj_datos_x,conj_datos_x)
  coefs_H2<-Regress_Lin(datos,conj_datos_y)
  g_2<-function(x){coefs_H2[1]+coefs_H2[2]*x+coefs_H2[3]*x*x}
  signos_estimados<-rep(1,length(signos))
  signos_estimados[which(conj_datos_y<g_2(conj_datos_x))]<-(-1)
  ein_g2<-length(which(signos!=signos_estimados))/length(signos)
  promedio_error_g2<-promedio_error_g2+ein_g2
  promedio_eout_g2<-promedio_eout_g2+generalization_bound(0.05,ncol(datos)+1,nrow(datos),ein_g2)

  #Calcular g_10 y su error
  datos<-cbind(conj_datos_x**10,conj_datos_x**9,conj_datos_x**8,conj_datos_x**7,conj_datos_x**6,
               conj_datos_x**5,conj_datos_x**4,conj_datos_x**3,conj_datos_x*conj_datos_x,conj_datos_x)

  coefs_H10<-Regress_Lin(datos,conj_datos_y)
  g_10<-function(x){coefs_H10[1]+coefs_H10[2]*x+coefs_H10[3]*x*x+coefs_H10[4]*x**3+
                    coefs_H10[5]*x**4+coefs_H10[6]*x**5+coefs_H10[7]*x**6+coefs_H10[8]*x**7+
                    coefs_H10[9]*x**8+coefs_H10[10]*x**9+coefs_H10[11]*x**10}

  #Calculo el porcentaje de error
  signos_estimados<-rep(1,length(signos))
  signos_estimados[which(conj_datos_y<g_10(conj_datos_x))]<-(-1)
  ein_g10<-length(which(signos!=signos_estimados))/length(signos)
  promedio_error_g10<-promedio_error_g10+ein_g10
  promedio_eout_g10<-promedio_eout_g10+generalization_bound(0.05,ncol(datos)+1,nrow(datos),ein_g10)
}

promedio_error_g2<-promedio_error_g2/nexperimentos
promedio_error_g10<-promedio_error_g10/nexperimentos

promedio_eout_g2<-promedio_eout_g2/nexperimentos
promedio_eout_g10<-promedio_eout_g10/nexperimentos

print(paste("Promedio ein g2: ",promedio_error_g2))
```

```
## [1] "Promedio ein g2:  0.3342"
```

```
print(paste("Promedio ein g10: ",promedio_error_g10))
```

```
## [1] "Promedio ein g10:  0.2648"
```

```
print(paste("Promedio eout g2: ",promedio_eout_g2))
```

```
## [1] "Promedio eout g2:  0.818680526260539"
```

```r
print(paste("Promedio eout g10: ",promedio_eout_g10))
```

```
## [1] "Promedio eout g10:  1.19250957804379"
```

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

Cuando tenemos un porcentaje de acierto razonable pero la cota de error fuera de la muestra se nos dispara, es un síntoma claro de sobreajuste.

Vemos que cuanto mayor es la complejidad de la función que ajustamos, mayor es el sobreajuste, siendo esto así, es lógico que la capacidad de sobreajuste de la función g_2 no sea muy alta y por ello, comete más error en la muestra (E_{in}) aunque por contra, el error fuera de la muestra está acotado ($< 100\%$). El error en promedio de la función g_{10}, es menor, pero vemos cómo la cota de error fuera de la muestra (E_{out}) se dispara por encima de 1, es decir, el error fuera de la muestra no está acotado.

Con este experimento deducimos que el sobreajuste es un problema real y que nos va a disparar el error fuera de la muestra. Nuestra obsesión es conseguir el máximo poder de generalización (reducir E_{out} todo lo posible), por lo que vemos que aprender muy bien una muestra no es, para nada, algo beneficioso.

En este caso concreto, el error dentro de la muestra será todavía elevado, pues estamos aproximando una función de grado 20 con una función de grado 10 (en el caso en el que podríamos tener más sobreajuste), aun así vemos cómo aún así, la cota de error fuera de la muestra deja de estar acotada.

Regularización y Selección de Modelos

1. Para d=3 (dimensión) generar un conjunto de N datos aleatorios $\{x_n, y_n\}$ de la siguiente forma. Para cada punto x_n generamos sus coordenadas muestreando de forma independiente una $N(0,1)$. De forma similar generamos un vector de pesos de (d+1) dimensiones w_f, y el conjunto de valores $y_n = w_f^t x_n + \sigma_{\varepsilon_n}$, donde ε_n es un ruido que sigue también una $N(0,1)$ y σ^2 es la varianza del ruido; fijar $\sigma = 0,5$.

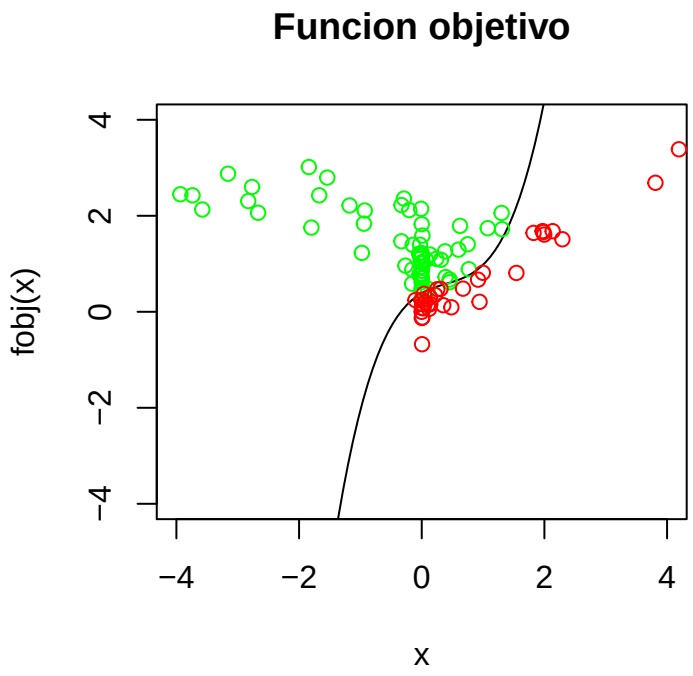
Usar regresión lineal con regularización "weight decay" para estimar w_f con w_{reg}. Fijar el parámetro de regularización a $0,05/N$.

Vamos a implementar el algoritmo de regresion lineal con weight decay, que no es mas que añadir un parámetro lambda que multiplica a la identidad. Si antes regresión lineal se calculaba como $(X^TX)^{-1}X^T$, añadir weight decay no es más que calcular regresión lineal como $(X^TX + \lambda I)^{-1}X^T$, donde λ es el parámetro de regularización.

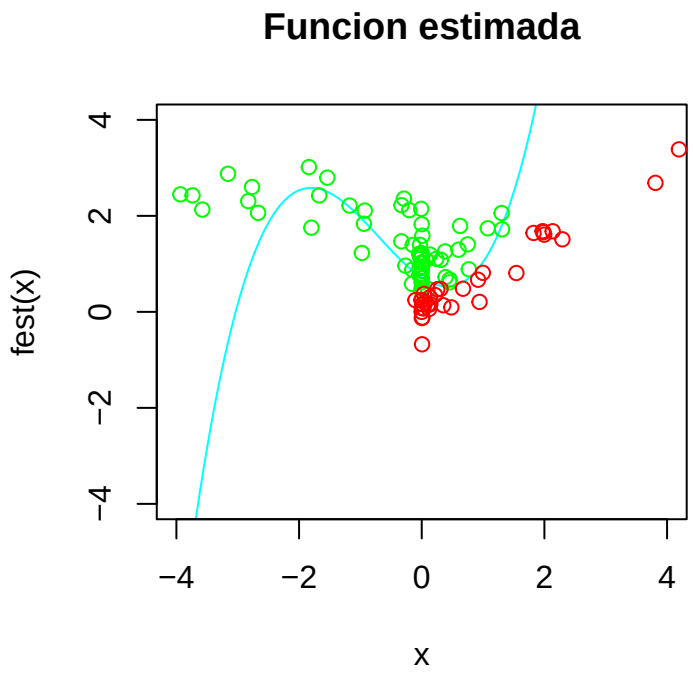
```r
WD_Regress_Lin<-function(data,label,lambda){
  data<-data[,ncol(data):1]
  #data<-cbind(1,data)

  x<-t(data)%*%data
  x<-x+(lambda*diag(nrow(x)))
  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
}
```

Vamos a generar algunos datos que utilizaremos posteriormente. Tambien veamos la función objetivo y estimemos una, para realizar la prueba del enunciado:

```r
#Generación de los datos
dimension<-3
N<-100
x_1<-matrix(rnorm(N,0,1),ncol=1)
dset<-cbind(x_1*x_1*x_1,x_1*x_1,x_1,1)
w_f<-matrix(rnorm(dimension+1,0,1),ncol=1)
y<-numeric(N)
for(i in 1:N){
  y[i]<-t(w_f)%*%dset[i,]
}

ruido<-rnorm(N,0,0.5)
y<-y+ruido

#Estimacion w_f (w_reg)
pr<-0.05/N
w_reg<-WD_Regress_Lin(dset,y,pr)

#Creamos las funciones con los pesos obtenidos
fobj<-function(x){w_f[4]*x**3+w_f[3]*x**2+w_f[2]*x+w_f[1]}
fest<-function(x){w_reg[4]*x**3+w_reg[3]*x**2+w_reg[2]*x+w_reg[1]}

#Calculamos los signos de ambas funciones
signos_fobj<-rep(1,nrow(dset))
signos_fobj[which(y<fobj(dset[,1]))]<-(-1)

signos_fest<-rep(1,nrow(dset))
signos_fest[which(y<fest(dset[,1]))]<-(-1)

#Pinto la función objetivo
curve(fobj,xlim=c(-4,4),ylim=c(-4,4),main="Funcion objetivo")
points(dset[,1][signos_fobj>=0],y[signos_fobj>=0],col="green")
points(dset[,1][signos_fobj<0],y[signos_fobj<0],col="red")
```




```
#Pinto la función estimada
curve(fest,xlim=c(-4,4),ylim=c(-4,4),col="cyan",main="Funcion estimada")
points(dset[,1][signos_fobj>=0],y[signos_fobj>=0],col="green")
points(dset[,1][signos_fobj<0],y[signos_fobj<0],col="red")
```




```
#Calculo el error cometido
print(paste("Error e_in",length(which(signos_fobj!=signos_fest))/length(signos_fobj)))
```

```
## [1] "Error e_in 0.13"
```

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

Definimos un error e_i como el error cometido por la funcion aprendida con cada conjunto de N puntos. Definimos también $E_{cv} = \frac{1}{N}\Sigma_{i=1}^{N} e_i$, es decir, la media de todos los errores cometidos por cada función aprendida para un conjunto de N puntos. Vamos a implementarlo:

```
offset<-15
errors<-c(0)
cont<-0

while(offset<=115){
  #Aumento el numero de puntos
  N<-dimension+offset
  #Genero nuevos puntos
  x_1<-matrix(rnorm(N,0,1),ncol=1)
  dset<-cbind(x_1*x_1*x_1,x_1*x_1,x_1,1)
  #Los ajusto con los pesos de la funcion objetivo
  y<-numeric(N)
  for(i in 1:N){
    y[i]<-t(w_f)%*%dset[i,]
  }
  #Añado ruido a los puntos
  ruido<-rnorm(N,0,0.5)
  y<-y+ruido
  #Estimo w_f (w_reg)
  pr<-0.05/N
  w_reg<-WD_Regress_Lin(dset,y,pr)
  #Calculo los errores
  signos_reales<-rep(1,length(y))
  signos_reales[which(y-ruido<y)]<-(-1)

  y_estimado<-numeric(N)
  for(i in 1:N){
    y_estimado[i]<-t(w_reg)%*%dset[i,]
  }
  signos_est<-rep(1,length(y))
  signos_est[which(y_estimado<y)]<-(-1)

  ei<-length(which(signos_reales!=signos_est))/length(signos_reales)
  errors<-c(errors,ei)

  cont<-cont+1
  offset<-offset+10
  print(paste("Error e_",cont,"=",ei))
}
```

```
## [1] "Error e_ 1 = 0.277777777777778"
## [1] "Error e_ 2 = 0.357142857142857"
## [1] "Error e_ 3 = 0.236842105263158"
## [1] "Error e_ 4 = 0.375"
## [1] "Error e_ 5 = 0.224137931034483"
## [1] "Error e_ 6 = 0.294117647058824"
## [1] "Error e_ 7 = 0.346153846153846"
## [1] "Error e_ 8 = 0.238636363636364"
```

```
## [1] "Error e_ 9 = 0.163265306122449"
## [1] "Error e_ 10 = 0.212962962962963"
## [1] "Error e_ 11 = 0.271186440677966"
```

```
errors<-errors[2:length(errors)]
E_cv<-mean(errors)
print(paste("Error de validacion cruzada =",E_cv))
```

```
## [1] "Error de validacion cruzada = 0.27247483980279"
```

b) Repetir el experimento 10^3 veces, anotando el promedio y la varianza de e_1, e_2 y E_{cv}

El código para este apartado es muy sencillo. Basta con envolver el código del apartado anterior del siguiente modo:

```
nexp<-10
nIncrementos<-(115-15)/10+1
e_i<-matrix(nrow=nexp,ncol=nIncrementos)
E_cvs<-numeric(nexp)
for(j in 1:nexp){
  #Codigo del apartado anterior
}
varianzas_e<-numeric(nIncrementos)
for(i in 1:nIncrementos){
  varianzas_e[i]<-var(e_i[,i])
}
var_E_cv<-var(E_cvs)
```

Donde `varianzas_e` y `var_E_cv` contienen las varianzas de los errores e_i y la varianza de E_{cv} respectivamente. El resultado es el siguiente:

```
## [1] "El promedio de e_1 es: 0.326"
```

```
## [1] "La varianza de e_1 es: 0.0140343306269232"
```

```
## [1] "El promedio de e_2 es: 0.313428571428571"
```

```
## [1] "La varianza de e_2 es: 0.00848019448019448"
```

```
## [1] "El promedio de E_cv es: 0.305908270829303"
```

```
## [1] "La varianza de E_cv es: 0.000462001766221185"
```

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

Hemos hecho 1000 experimentos y hemos definido E_{cv}, por resumirlo de algún modo, como la media de las medias de los errores en cada experimento. En un experimento i con un conjunto de N puntos, obtenemos un error $e_{i,j}$. Si hacemos 1000 experimentos, lo que tenemos es una matriz, donde cada fila i es un experimento y cada columna j contiene los resultados de todos los experimentos para cada conjunto de un número de

puntos variables. Entonces, un elemento de la matriz $M_{i,j}$ será el error cometido en el experimento i para un número N_j de puntos. Dicho de otro modo, si hacemos las medias de cada columna, estaremos considerando un vector $errors = (\bar{e_1}, ..., \bar{e_n})$. Si hacemos las medias de dicho vector, estaremos entonces calculando E_{cv}.

Siendo esto así, la relación que guardan $\bar{e_1}$ y $\bar{e_2}$ con E_{cv} es la misma, son sumandos de una media. a la vista de los experimentos, y teniendo tambien muy claro el concepto de varianza (cuadrado de la desviación típica), está claro que podemos acotar la media de un error e_i será: $\bar{e_i} \leq E_{cv} \pm \sqrt{var(E_{cv})}$.

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

Como en cualquier cálculo de una varianza lo que influye es un dato y la media de la población a la que pertenece el dato, es decir, la varianza del error e_j cometido con un conjunto de N_j puntos se define como: $var(e_j) = \Sigma_{i=1}^{N_{experimentos}}(e_{j,i} - \bar{e_j})^2$

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

Si los errores de validación fueran realmente independientes, la varianza de e_1 sería un sumando de la media de la varianza de E_{cv}, dicho de otro modo, si los errores fueran independientes, tendríamos que:

$$var(E_{vc}) = \frac{1}{nIncremenos}\Sigma_{j=1}^{nIncrementos} var(e_i)$$

Como podemos comprobar, no es así:

```
print(paste("Varianza E_cv:",var_E_cv))
```

```
## [1] "Varianza E_cv: 0.000462001766221185"
```

```
print(paste("Media de las varianzas de e_i:", mean(varianzas_e)))
```

```
## [1] "Media de las varianzas de e_i: 0.00493314946088305"
```

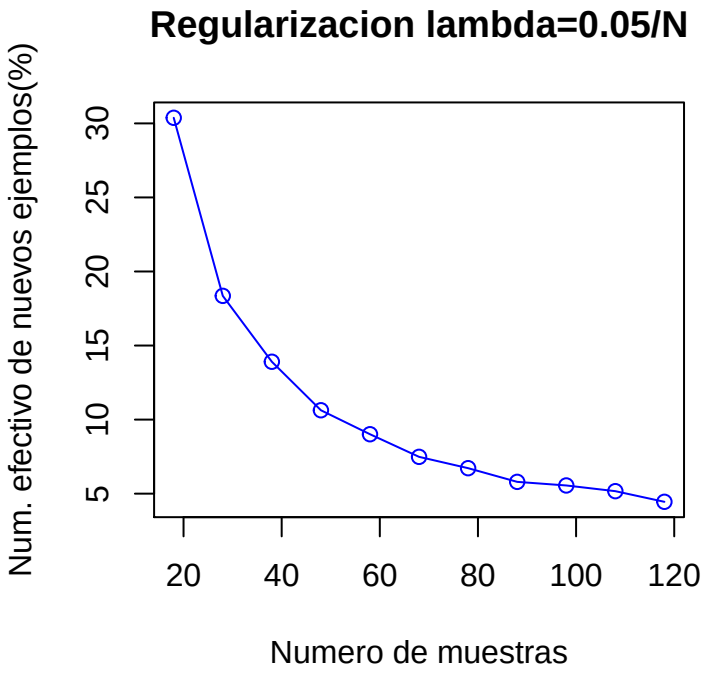
f) Una medida del número efectivo de muestras nuevas usadas en el cálculo de E_{cv} es el cociente entre la varianza de e_1 y la varianza de E_{cv}. Explicar por qué, y dibujar, respecto de N, el número efectivo de nuevos ejemplos (N_{eff}) como un porcentaje de N.

Con ese cociente estamos hallando la relación entre cuánto varían los experimentos entre cuánto varian las diferentes hipótesis en un mismo experimento. En realidad, para encontrar N_{eff} habría que utilizar la máxima varianza de los e_i, pues es la medida que más se va a acercar a N. Por ejemplo, para $N = dimension + 15$, el número efectivo de muestras es:

```
print(paste("Numero efectivo de muestras",varianzas_e[1]/var_E_cv))
```

```
## [1] "Numero efectivo de muestras 30.3772228875078"
```

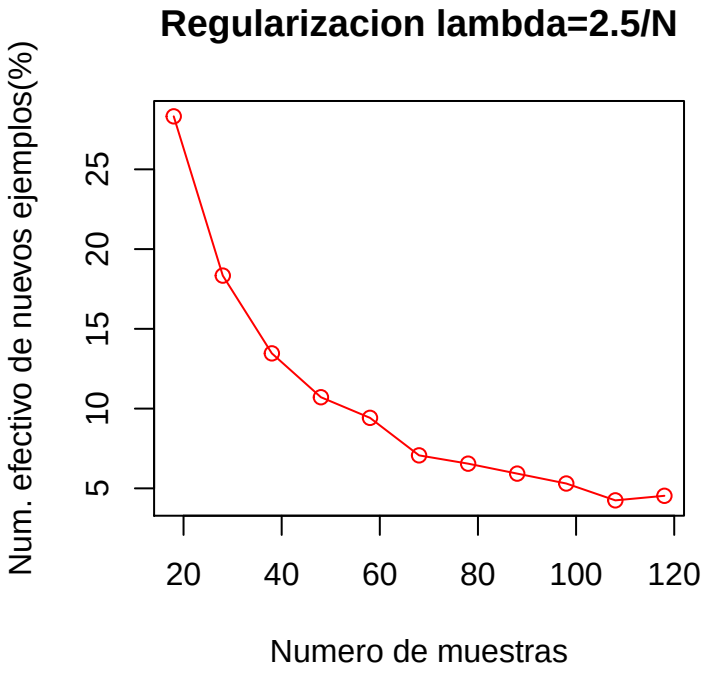
Veamos a ver el gráfico de N_{eff} para todos los tamaños de N pedidos:



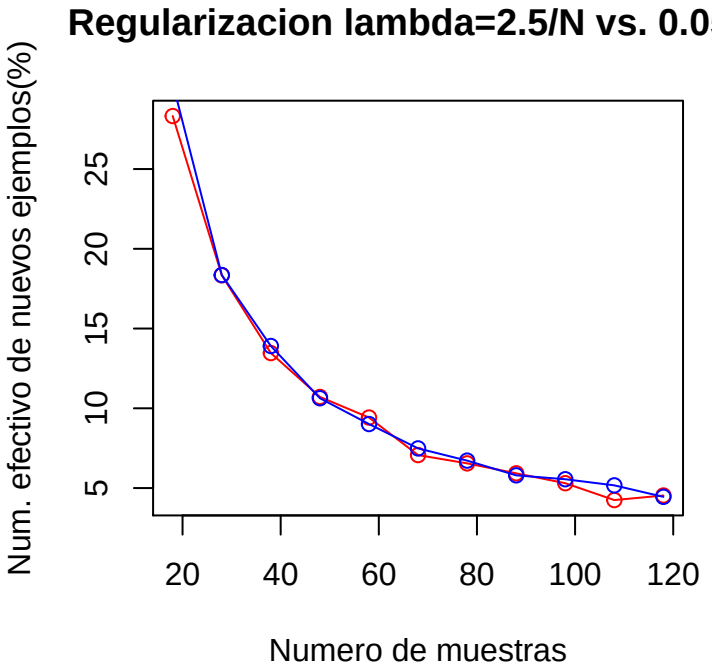
Como es lógico, cuanto mayor es la muestra menos ejemplos nuevos se necesitarán para explicar el fenómeno.

g) Si se incrementa la cantidad de regularización ¿Debería N_{eff} subir o bajar? Argumentar la respuesta. Ejecutar el mismo experimento con $\lambda = 2,5/N$ y comparar los resultados del punto anterior para verificar la conjetura.

Teóricamente debería bajar. Cuando elegimos un lambda alto nos estamos acercando a un modelo lineal, así que el número de puntos efectivos necesarios para fijarlo será menor que con un lambda muy cercano a 0 (modelo sobreajustado).



Comparemos ambas regularizaciones. En rojo, la regularicación con $\lambda = 2.5/N$, en azul, la regularización $\lambda = 0.05/N$:



Vemos como, efectivamente, la regularización $\lambda = 2.5/N$ requiere normalmente menos número efectivo de nuevos ejemplos N_{eff}