P3

JuanE y Adri

5 May 2018

1. AJUSTE DE MODELOS LINEALES

PROBLEMA DE CLASIFICACIÓN: base de datos "Optical Recognition of Handwritten Digits"

1. Problema a resolver

El problema en si trata de ajustar un modelo lineal sobre dicha base de datos para decidir si un digito es un número entre {0,1,..,9} escritos a mano, es decir, estamos ante un problema de clasificación multi clase (10 clases). Los conjuntos se generaron dividiendo cada digito en una matriz de 8x8 y calculando un valor de escala de grises de 0 a 16 para cada celda de la matriz promediando sus pixeles. Los datos entonces tienen 64 columnas para cada uno de los valores de escala de grises y una ultima columna (65) con el digito real (será nuestro label).

Una manera de resolver este problema podria ser crear 10 clasificadores donde cado uno clasifica entre un digito y los demás y asi con todas las clases (One vs All). Otra manera podría ser utilizando Softmax, es decir, buscamos poder clasificar usando un modelo lineal un dataset multiclase. Para hacer esto tenemos a parte de la opción One vs All, la opción softmax o función exponencial normalizada, esta es una generalización de la función logística. Su función es la de "comprimir" un vector K-dimensional, z, de valores arbitrarios (clasificaciones) en un vector K-dimensional, $\sigma(z)$, de valores reales en el rango [0, 1]. La función esta dada por:

$\sigma : \mathbb{R}^{\mathbb{K}} \to [0,1]^K \ \sigma(z)_j = \frac{e^{zj}}{\sum_{k=1}^{K} e^{zk}}$ for $j = 1, ..., K$

La salida de la función softmax puede ser utilizada para representar una distribución categórica (distribución de probabilidad sobre K diferentes posibles salidas). Por tanto, sabiendo esto podemos determinar que es idónea para nuestro problema de clasificación.

Los pasos a seguir para afrontar este problema serían en general estos:

1º Cabe señalar que lo primero que habría que hacer es asegurarnos de que todos los disgitos son igual de grandes (comparandolos mediante el número de píxeles), asumimos que esto es así pues en la web de descarga del dataset no se indica ninguna anomalía de este estilo. En cualquier caso si esto sucediese nuestra función cv.glmnet en algún momento se quejaría y mostraría el error correspondiente.

2º Normalizar los datos, esto nos sirve para no tener un cuenta unas características por encima de otras en nuestros datos. Es decir, es importante pues nos garantiza que cada parámetro de entrada (píxel, en nuestro caso) tiene una distribución de datos similar. Esto hará que la convergencia sea mucho más rápida al entrenar nuestro modelo de regresión logística. Para normalizar sacaremos el máximo de cada conjunto de pixeles (mismo pixel en cada número) y el mínimo también de cada conjunto de píxeles y una vez hecho esto a cada pixel le restamos el mínimo y lo dividimos entre el maximo menos el mínimo, de esta manera obtendremos una distribución de datos entre 0 y 1, tal cual como lo queremos y además obtendremos columnas a 0 lo cual quiere decir que el mínimo y el máximo es igual y por tanto ya podemos adelantar que estas columnas son ruido para nuestro modelo.

3º Tenemos los datos en un solo canal como es la escala de grises por lo que no necesitamos reducir la dimensionalidad de estos pues ya están correctos.

Depues de esto procederiamos con la regularización como vamos a ver más adelante.

2. Preprocesamiento de los datos.

Se deben normalizar los datos ya que no es igual considerar los mínimos y máximos globales (dataset completo) que considerar los de cada conjunto (mismo pixel en distintos números), es decir, pudiera ser que se "envenenaran" los conjuntos. Esta función nos normalizará los datos para así conseguir comprimir los datos en un rango [0, 1].

```
normalized<-function(data) {
  x<-data[!is.na(data)]
  max=max(x)
  min=min(x)
  if(max==min){
      x=0.0
    }
  else
    x<-(x - min(x)) / (max(x) - min(x))
  data[!is.na(data)]<-x
  return(data)
}
```

3. Selección de clases de funciones a usar.

Utilizaremos Regresión Logística ya que dicho modelo es muy usado cuando la respuesta es categórica (exactamente nuestro problema). Como nuestro modelo tiene 10 posibles salidas, nosotros usaremos el tipo de modelo Multinomial que es una generalizacion del metodo de regresion logistica para problemas multiclase (función softmax explicada anteriormente).

El modelo predice las probabilidades de los diferentes resultados posibles de una distribucion categorica como variable independiente, dado un conjunto de variables independientes.

Para este tipo de modelo la variable de salida K tiene 10 clases: G={0,1,...,9}.

Una vez ajustado el modelo de regresion logística, lo compararemos con otro modelo de regresión logística usando regularización:

4. Conjuntos de training, validacion y test usados.

Como los datos ya nos vienen correctamente divididos en train y test, procedemos a su lectura y a la descomposición para tener correctamente identificadas y separadas las características y las etiquetas de cada conjunto de datos (train y test). El conjunto de validación lo explicamos en la siguiente sección.

```
train = read.csv("datos/optdigits_tra.csv", header = FALSE)
names(train)[ncol(train)] = "digit"
test = read.csv("datos/optdigits_tes.csv", header = FALSE)
names(test)[ncol(train)] = "digit"

features_train = data.matrix(subset(train, select = -digit))
labels_train   = data.matrix(subset(train, select = digit))
features_test  = data.matrix(subset(test,  select = -digit))
labels_test    = data.matrix(subset(test,  select = digit))
```

Ahora con la función definida antes, normalizamos las características:

```
features_train = apply(features_train,2,normalized)
features_test = apply(features_test,2,normalized)
```

```
train[,-ncol(train)]=apply(train[,-ncol(train)],2,normalized)
```

5. Regularización, modelo a usar e hyperparámetros.

En concreto, usaremos (ajustaremos) un modelo de regresión LASSO (R-LASSO). Este modelo de regresión lineal selecciona las variables con coeficiente mayor de un umbral prefijado. La ventaja de este modelo es que es una técnica de **regresión lineal ya regularizada**, es decir, como los datos de entrada son pixeles, es decir, pueden llegar a ser muy redundantes, dicho modelo ya penaliza variables que no dicen nada acerca de la salida, disminuyendo el correspondiente sobreajuste que esto pudiera ocasionar y además reduciendo el error fuera de la muestra.

El hiperparámetro lambda: vamos a estimar el mejor lambda mediante validacion cruzada. El mejor lambda es aquél que penaliza minimizando el error de validacion cruzada, es decir, cvm.

Como se ha mencionado antes, este modelo se intuye que será el principal, pero para tener otro con el que comparar e ir viendo la necesidad de regularización, primero ajustaremos el modelo de regresión logísitca sin regualrizar.

El paquete **glmnet** de R nos permite aplicar todo lo antes mencionado.

El paquete **glmnet** a parte de ajustar el modelo también hace un procesamiento de los datos el cual es muy necesario para el correcto funcionamiento posterior del modelo de regresión logística. Los pasos que sigue esta libreria para preprocesar son:

glmnet es un set de procedimientos extremadamente eficientes para ajustar todo el procedimiento de regularización lasso (también puede hacer elastic-net pero usaremos solo lasso) para regresión lineal, modelos de regresión logística multinomial (nuestro caso), regresión de Poisson y el modelo de Cox. Los algoritmos que contiene usan lo que se conoce como descenso cíclico coordinado, que optimiza sucesivamente la función objetivo sobre cada parámetro con otros fijos y cicla hasta que converge.

Ahora hablemos del **conjunto de validación**: usaremos la funcin cv.glmnet() para ajustar el modelo usando cross-validation. Por defecto, el metodo divide el conjunto de entrenamiento antes creado en 10 trozos no superpuestos de aproximadamente el mismo tamaño, utilizando el primero de ellos como **validacion** y el resto se usa para el ajuste.

```
#Ajuste del modelo. Regresión logistica normal
rln=multinom(digit ~ ., data = train)
```

```
## # weights:  660 (585 variable)
## initial  value 8802.782811
## iter  10 value 1446.527616
## iter  20 value 587.767761
## iter  30 value 337.450780
## iter  40 value 207.169230
## iter  50 value 100.984459
## iter  60 value 21.914484
## iter  70 value 0.483734
## iter  80 value 0.002841
## final  value 0.000083
## converged
```

```
#summary(rln)
```

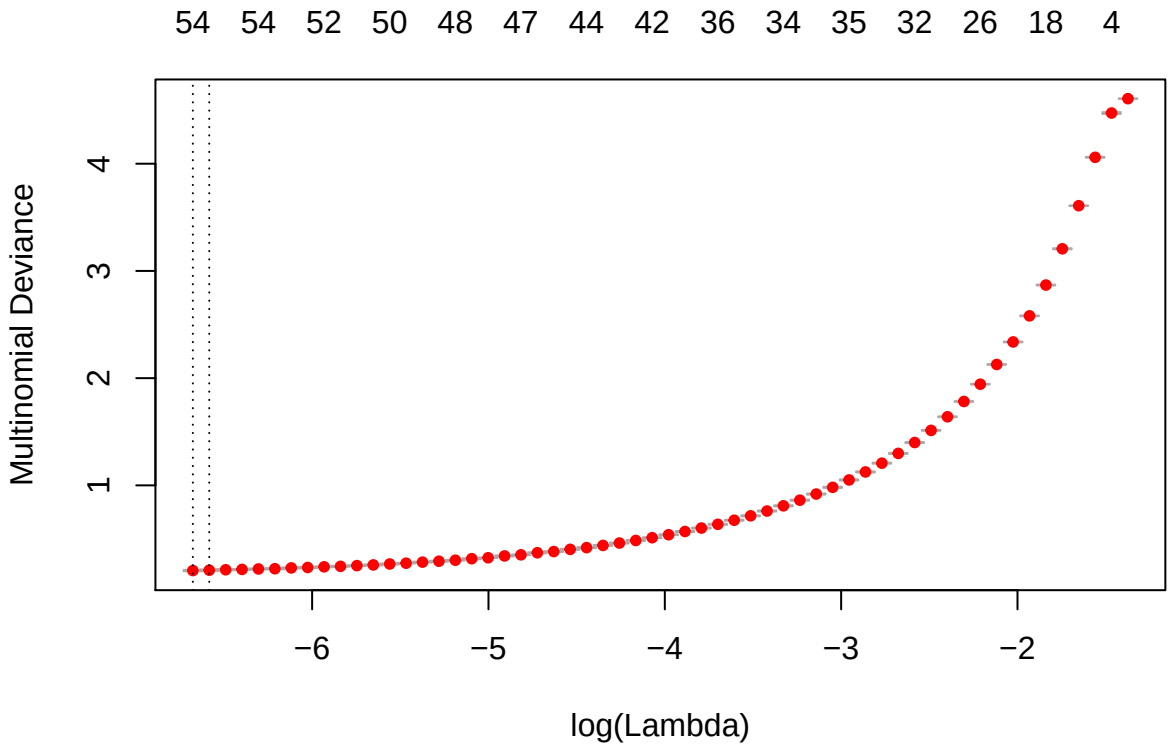
```
#Para parelelizar (demasiado tiempo secuencialmente)
library(doParallel)
```

```
## Warning: package 'doParallel' was built under R version 3.4.4
```

```
## Loading required package: iterators
## Warning: package 'iterators' was built under R version 3.4.4
## Loading required package: parallel
registerDoParallel(cores=8)

# Elegimos el mejor lambda por CV de, por ejemplo, 10 particiones (por defecto). Si utilizaramos tantas

#Ajuste del modelo (R-lasso)

cvfit_ls=cv.glmnet(features_train, labels_train, family="multinomial", type.multinomial = "grouped", pa

best_lambda_ls=cvfit_ls$lambda.min

#Ajuste del modelo (R-ridge)

cvfit_rd=cv.glmnet(features_train, labels_train, family="multinomial", type.multinomial = "grouped", pa

best_lambda_rd=cvfit_rd$lambda.min
```

Como vemos, el valor de lambda para Ridge es mayor, por lo que aplciará mas regularización (tendrá mas en cuenta la penalizacion a las características).

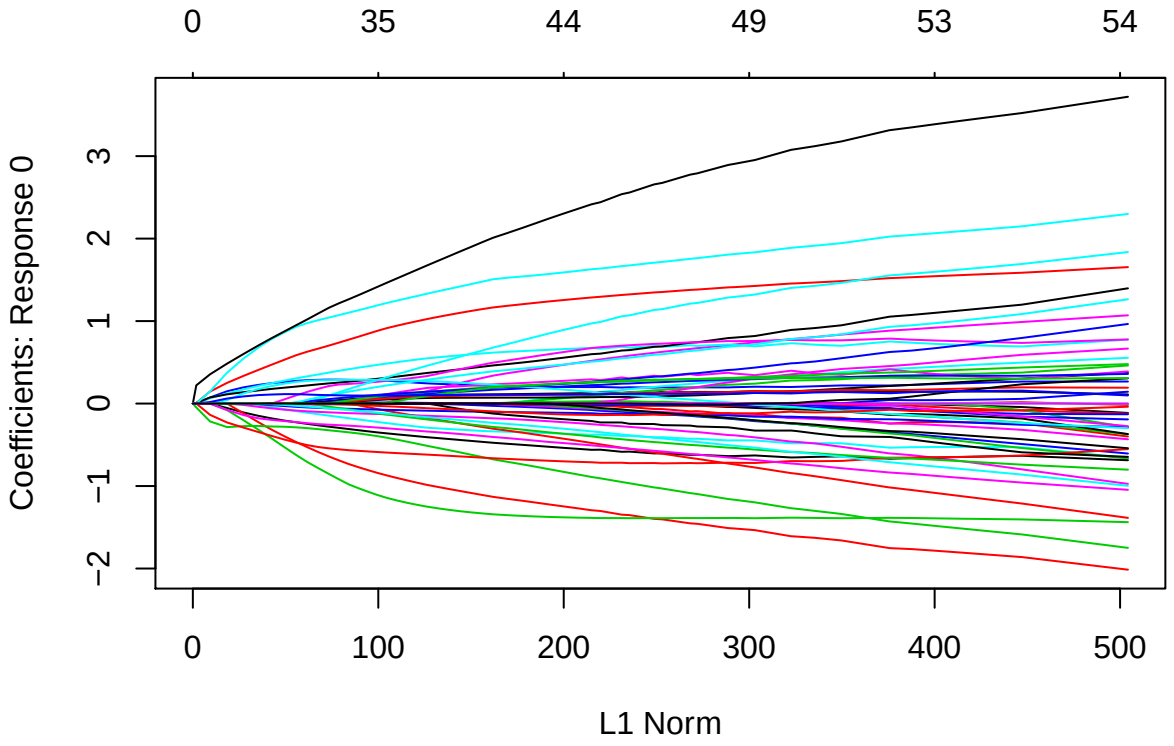
```
#Evolucion de los grados de libertad del modelo y porcentaje de desviacion segun los distintos valores
plot(cvfit_ls)
```

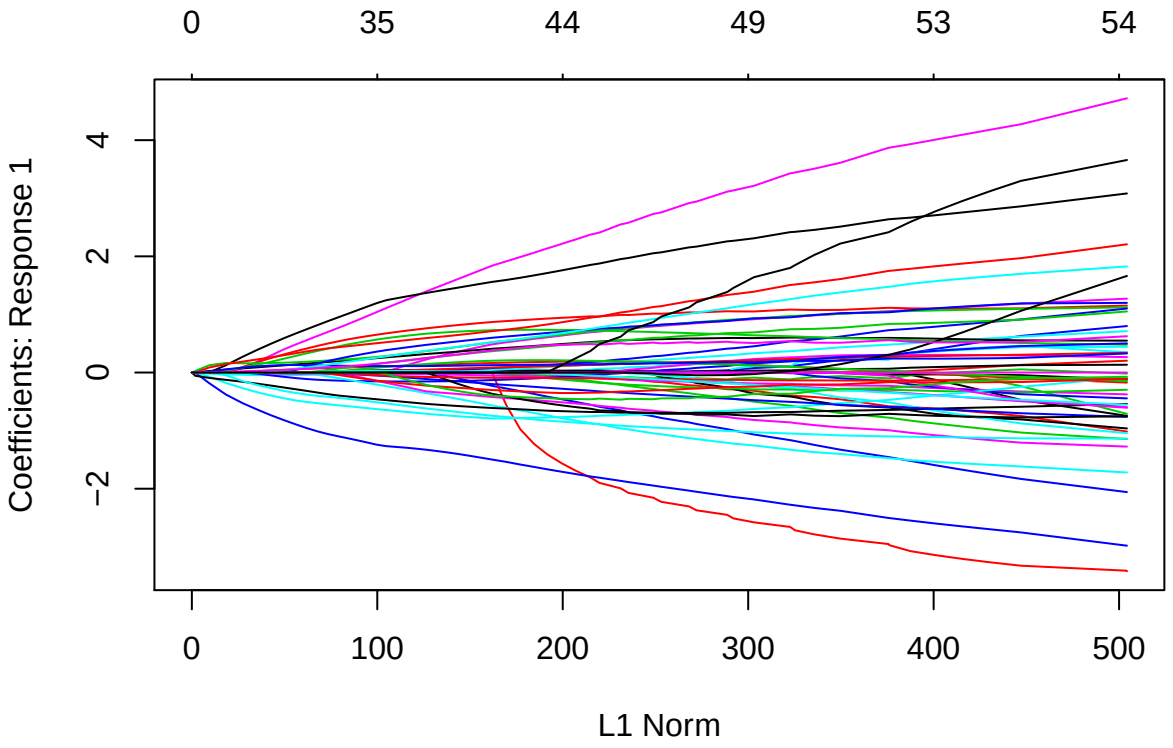


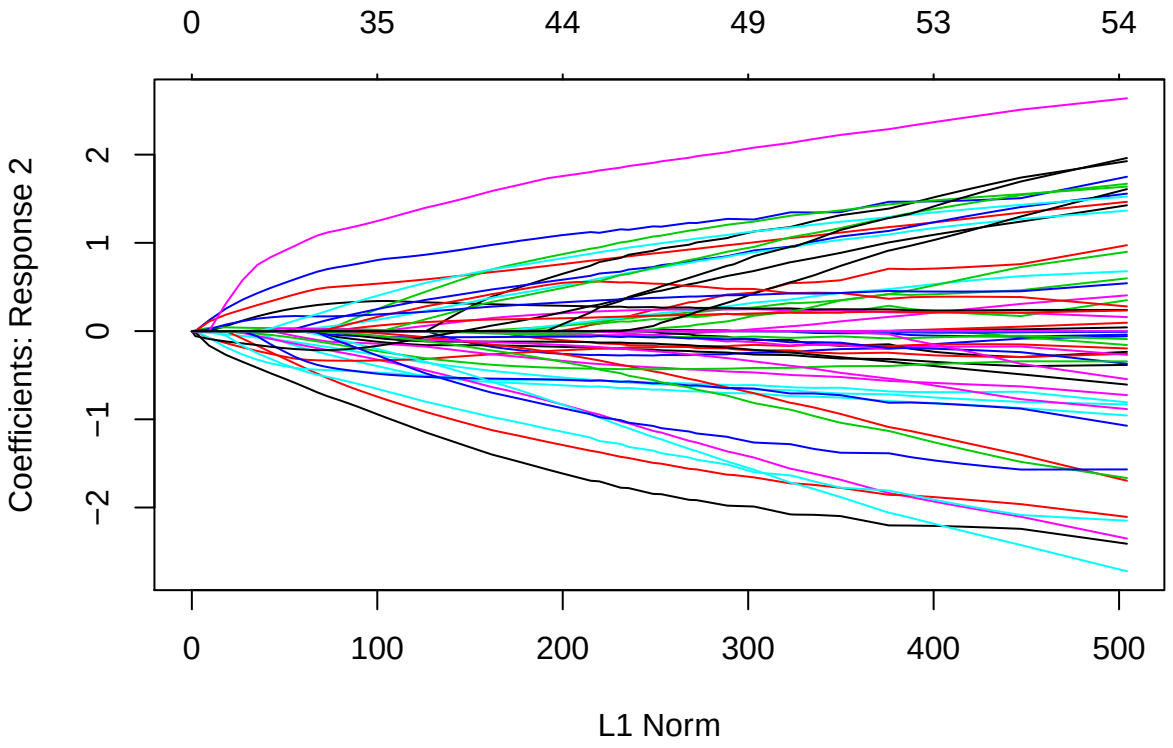
```
par(c(2,2))
```

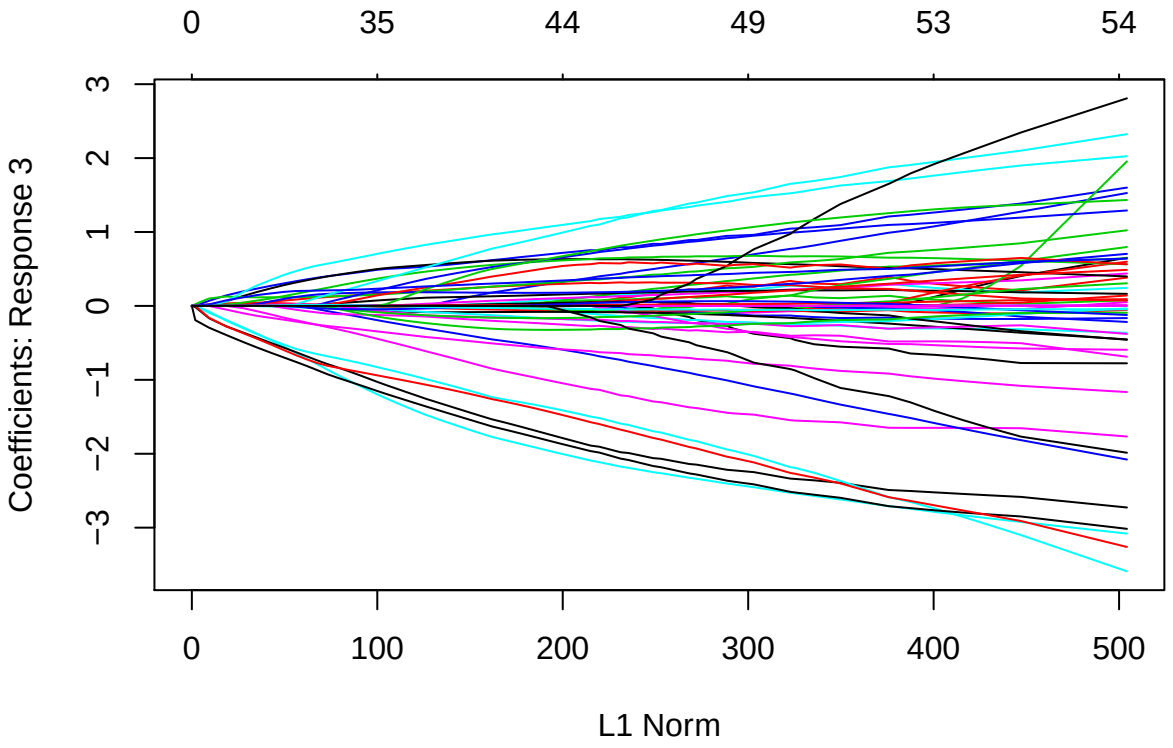
```
## NULL
```

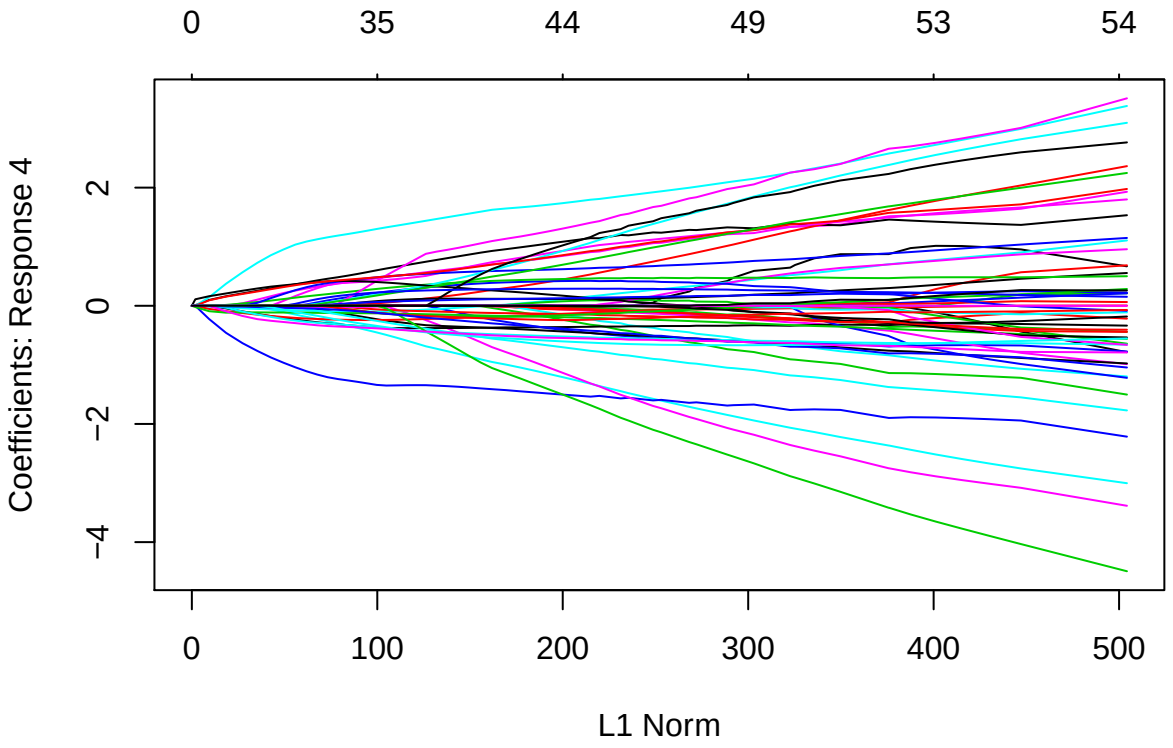
```
plot(cvfit_ls$glmnet.fit, "norm")
```

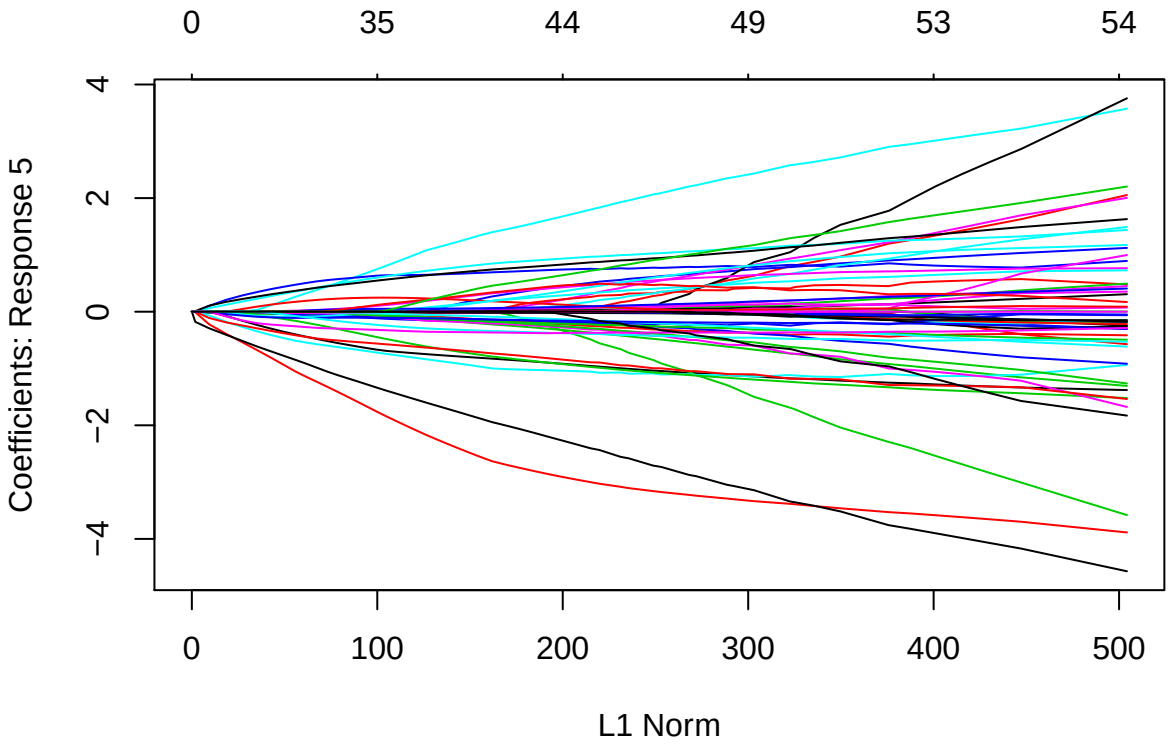


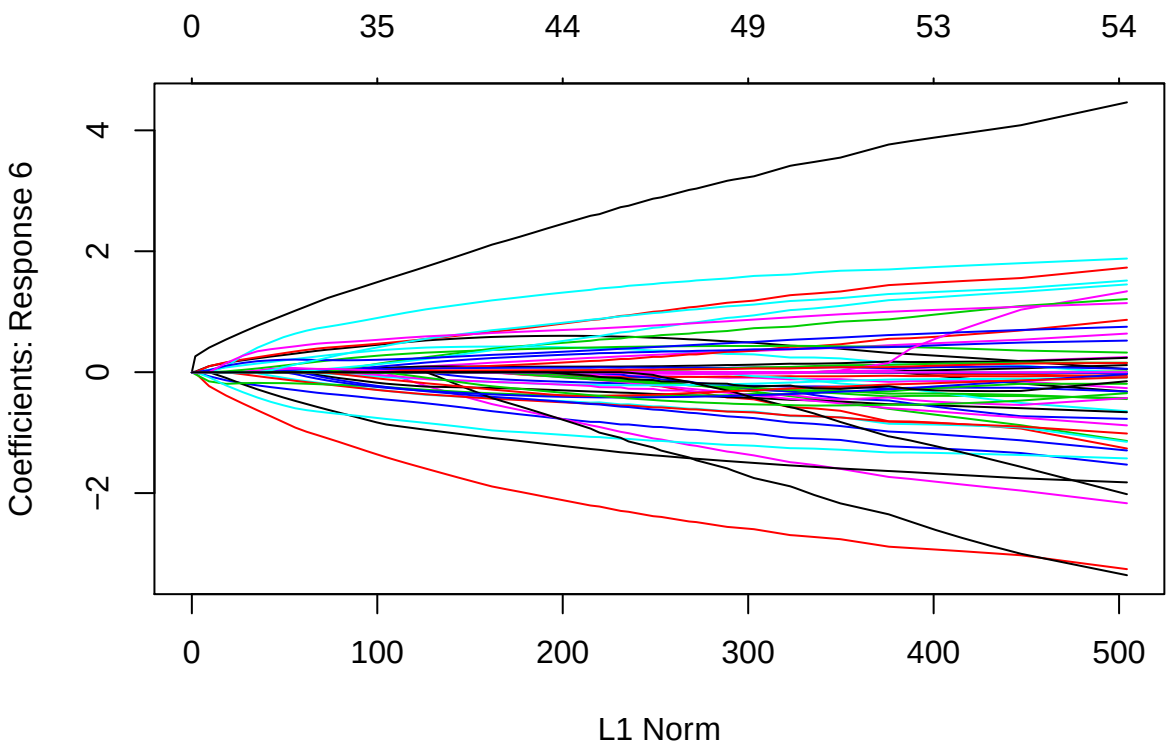

0
35
44
49
53
54
Coefficients: Response 1
4
2
0
−2
0
100
200
300
400
500
L1 Norm

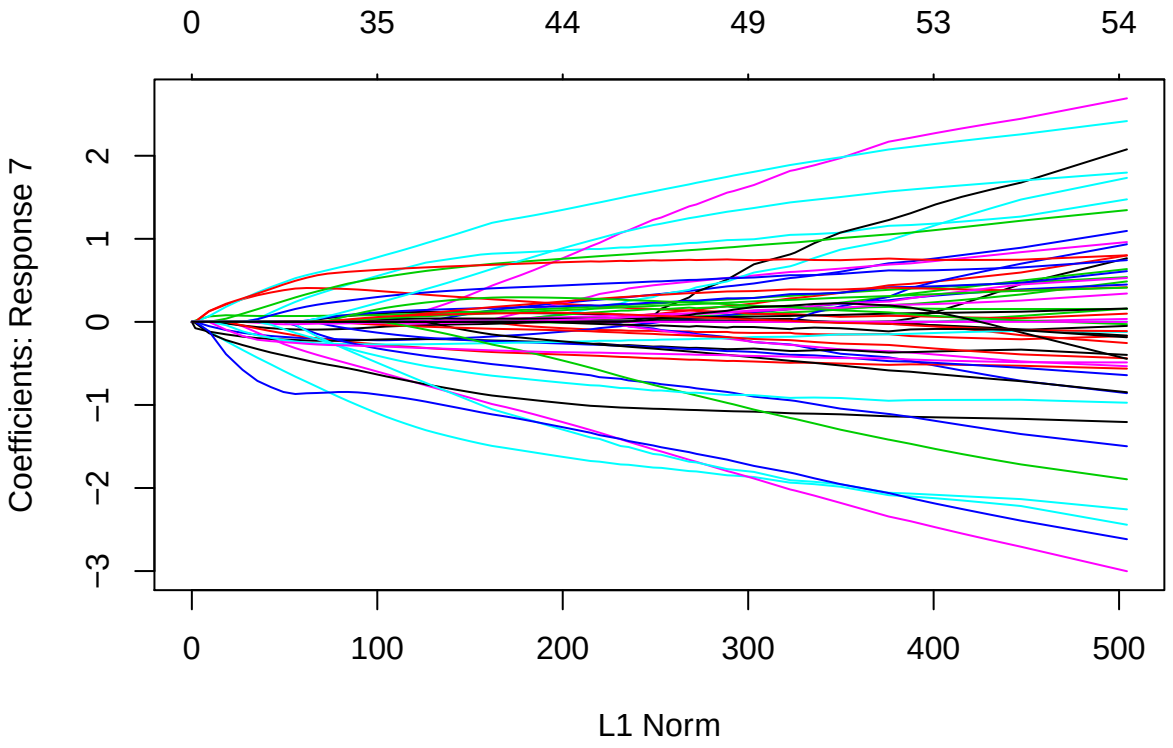

0
35
44
49
53
54
Coefficients: Response 2
2
1
0
−1
−2
0
100
200
300
400
500
L1 Norm

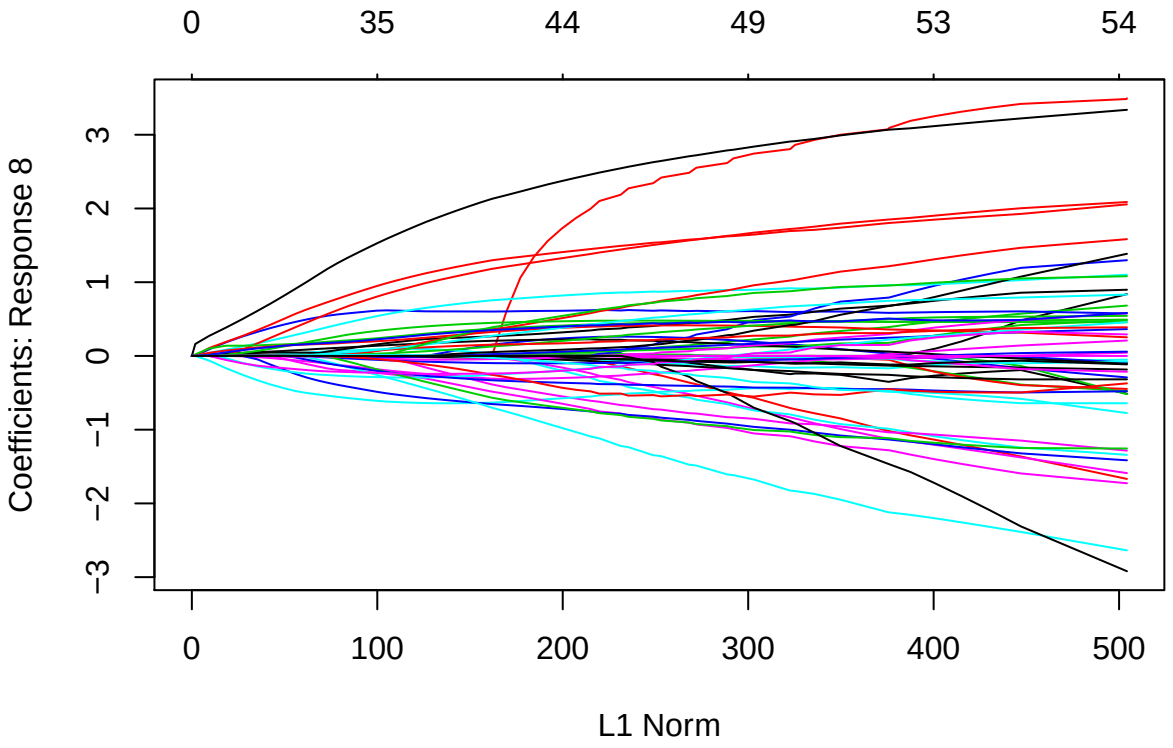

0
35
44
49
53
54
Coefficients: Response 3
3
2
1
0
−1
−2
−3
0
100
200
300
400
500
L1 Norm


0
35
44
49
53
54
Coefficients: Response 4
2
0
−2
−4
0
100
200
300
400
500
L1 Norm

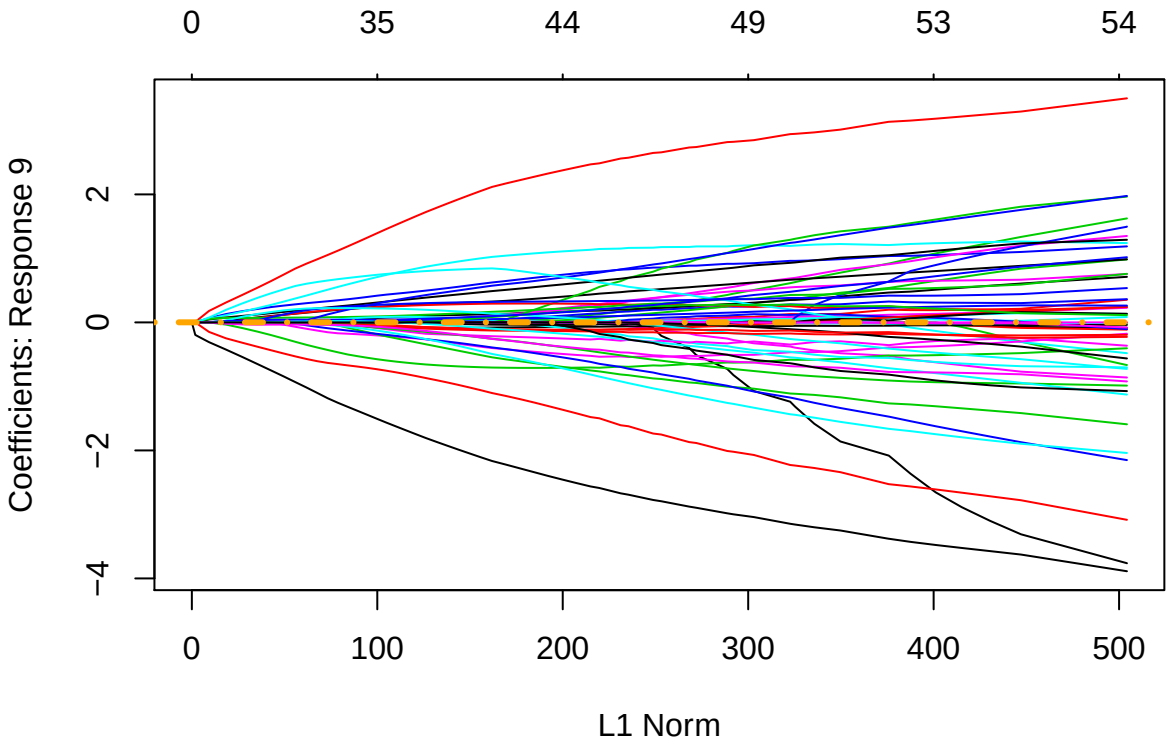

0
35
44
49
53
54
Coefficients: Response 5
4
2
0
−2
−4
0
100
200
300
400
500
L1 Norm


0
35
44
49
53
54
Coefficients: Response 6
4
2
0
−2
0
100
200
300
400
500
L1 Norm


0
35
44
49
53
54
Coefficients: Response 7
2
1
0
−1
−2
−3
0
100
200
300
400
500
L1 Norm



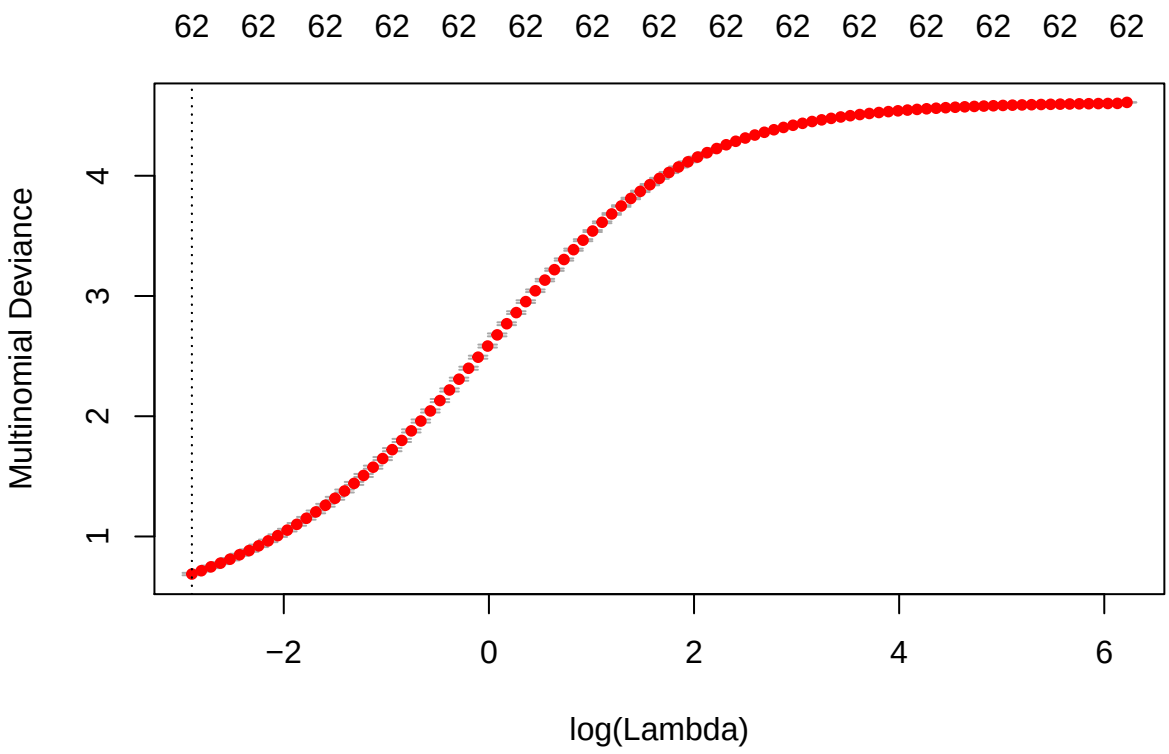

```
abline(h=best_lambda_ls, col= "orange", lty=4, lwd=3)
```



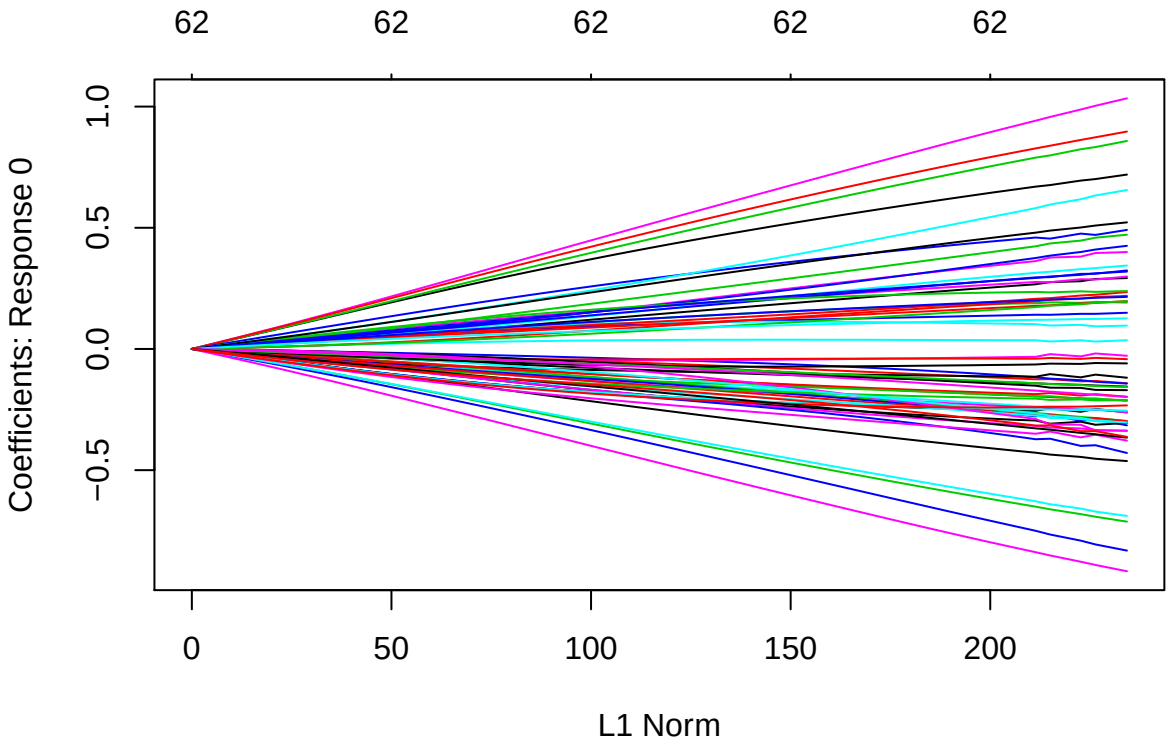

```
paste ("El mejor lambda es:", best_lambda_ls)
```

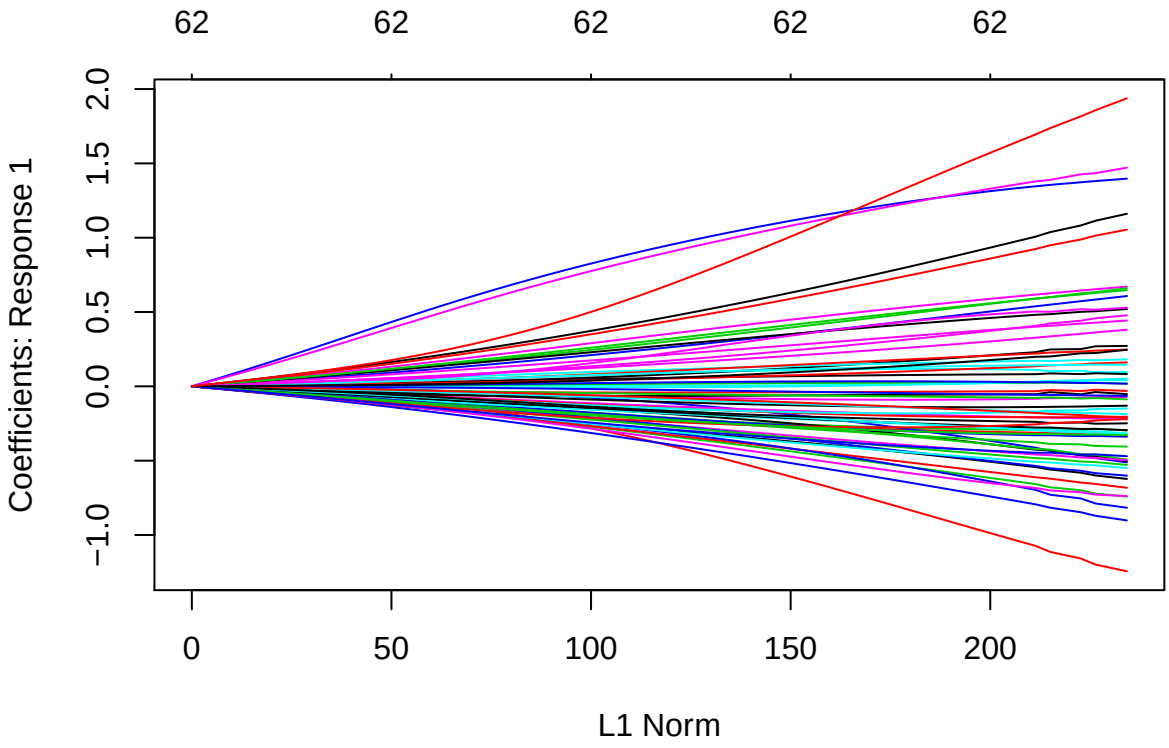
```
## [1] "El mejor lambda es: 0.00126015569533332"
```

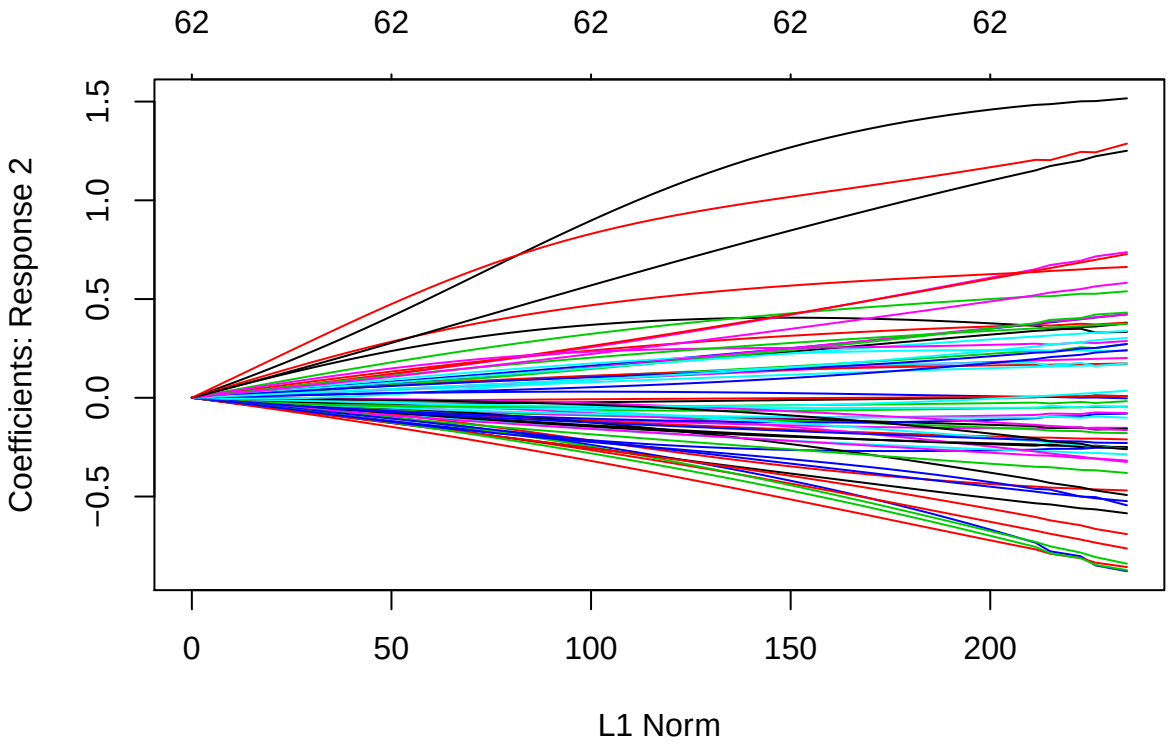
```
#ridge
plot(cvfit_rd)
```

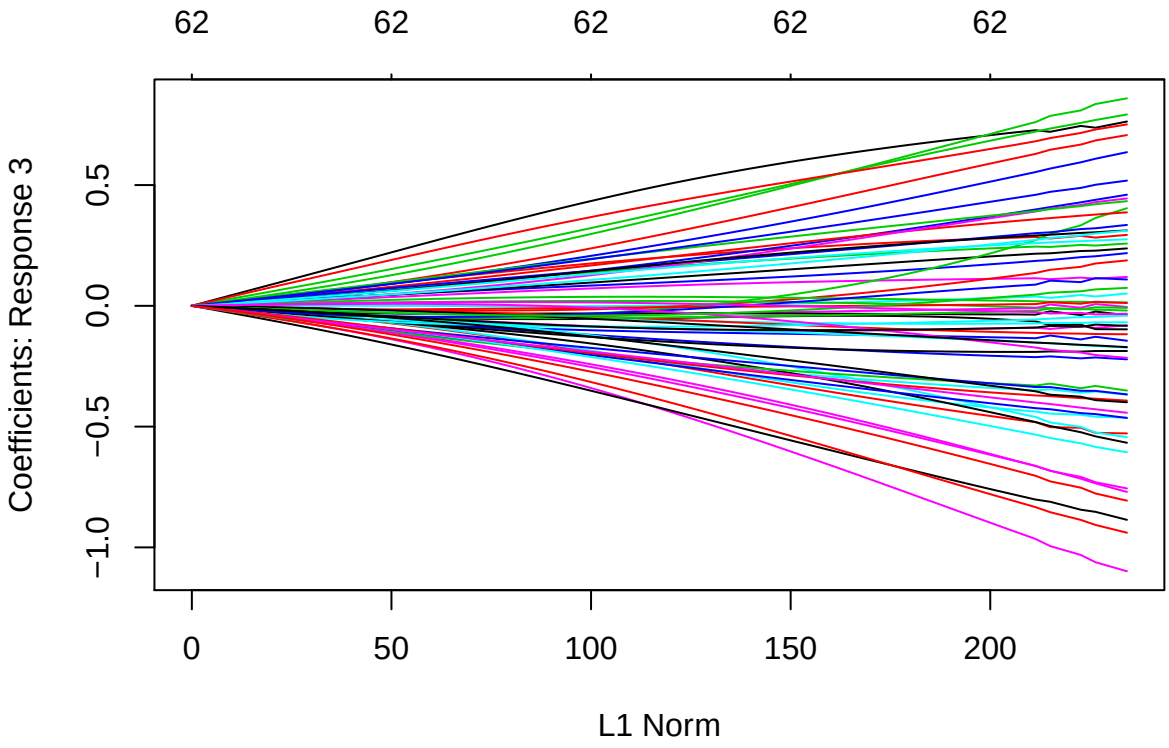


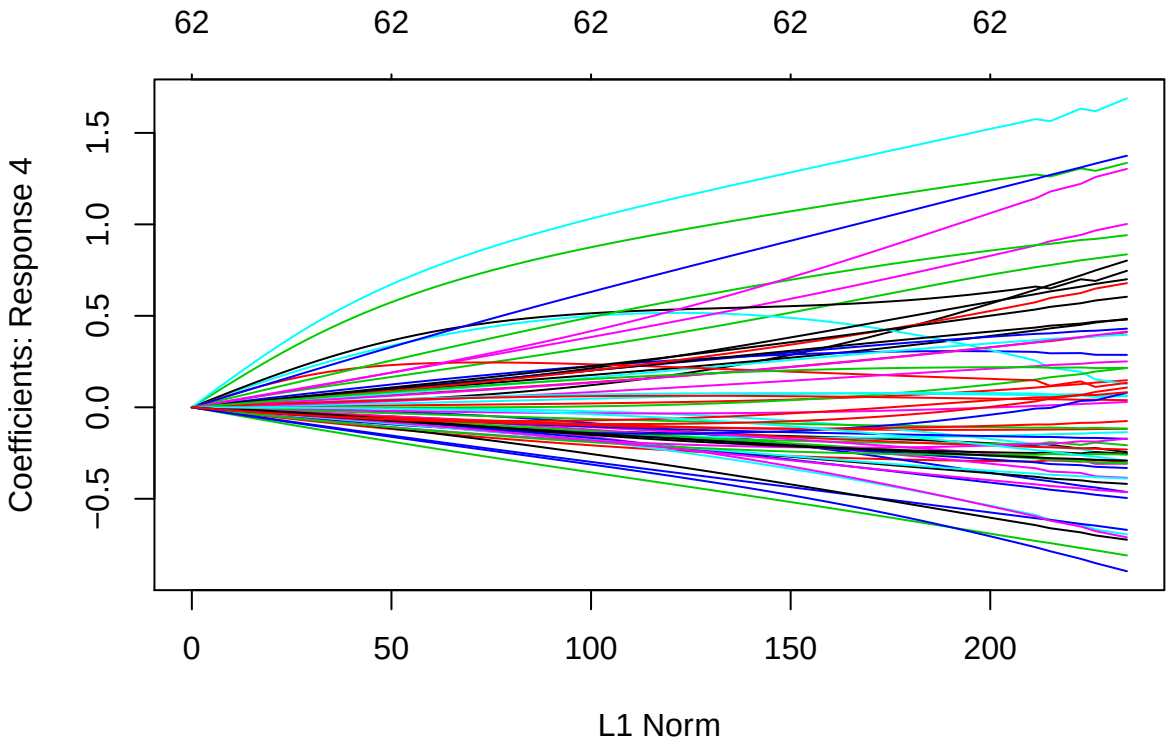

```
plot(cvfit_rd$glmnet.fit, "norm")
```

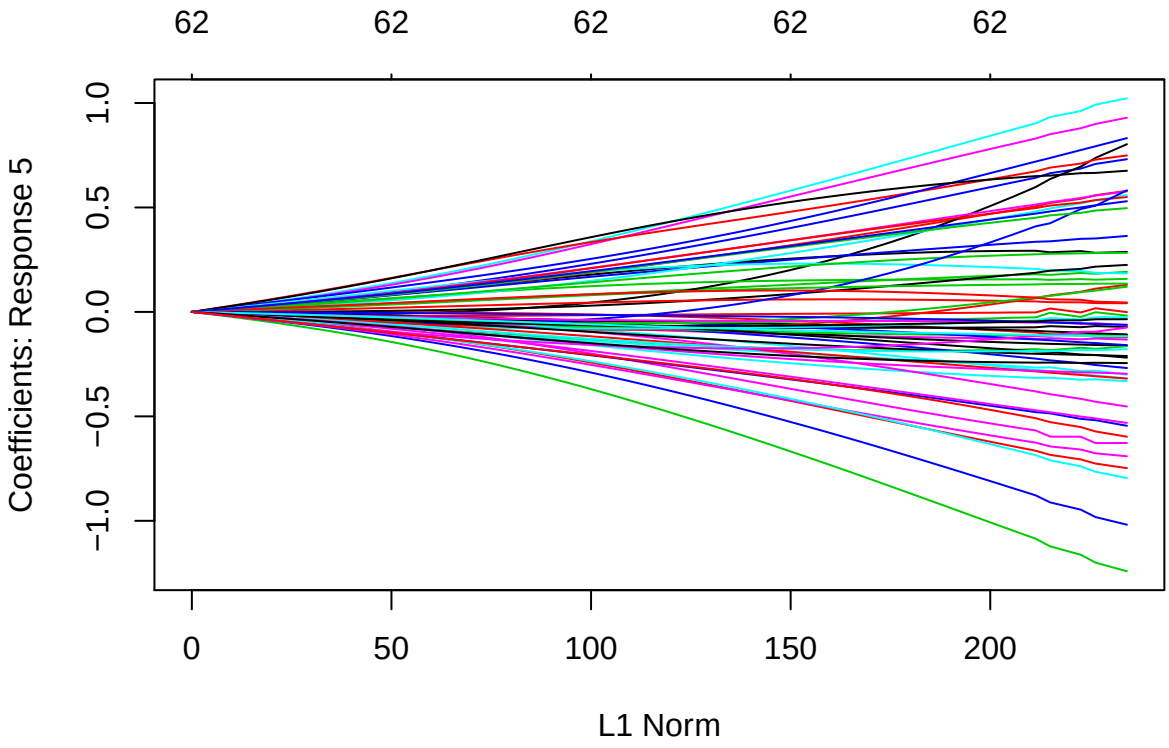

62
62
62
62
62
Coefficients: Response 0
1.0
0.5
0.0
−0.5
0
50
100
150
200
L1 Norm

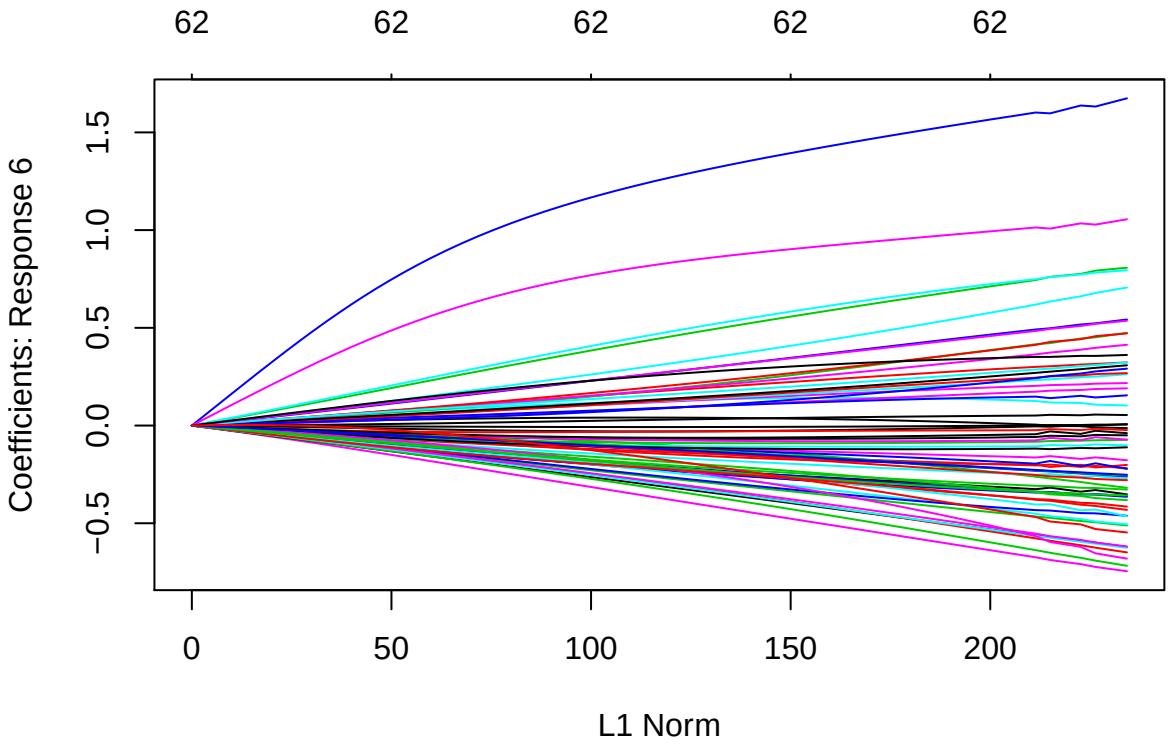

62
62
62
62
62
Coefficients: Response 1
2.0
1.5
1.0
0.5
0.0
−1.0
0
50
100
150
200
L1 Norm

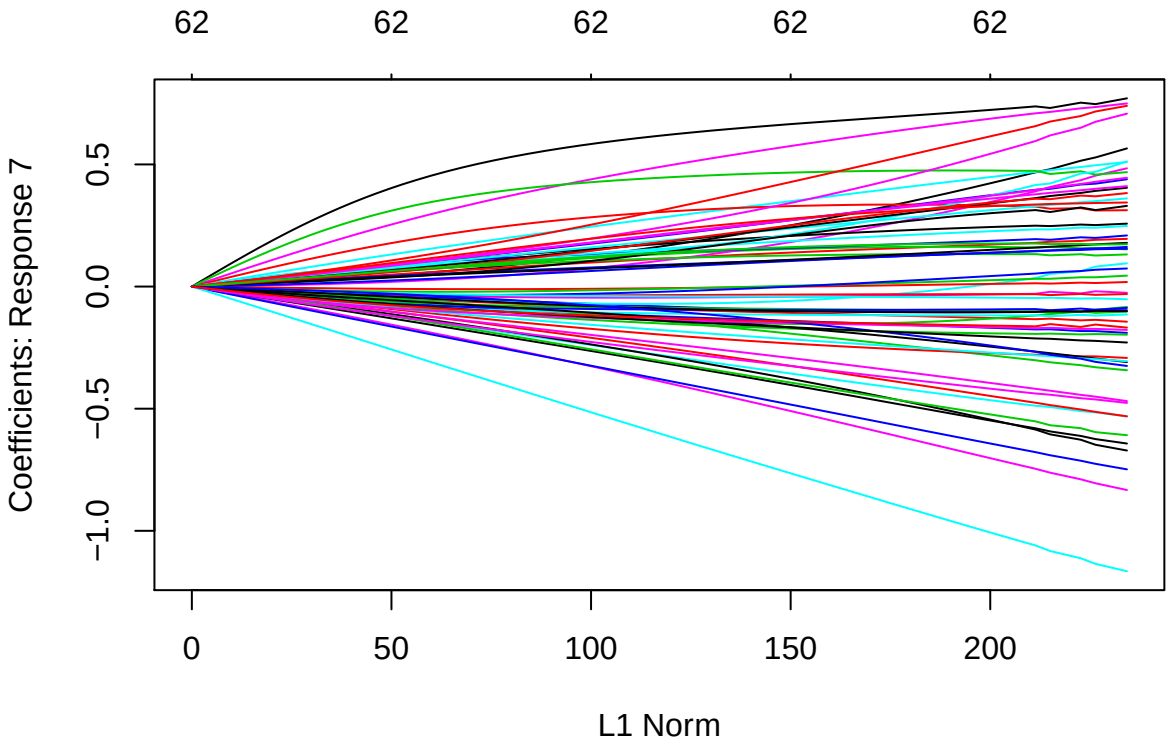

62
62
62
62
62
Coefficients: Response 2
1.5
1.0
0.5
0.0
−0.5
0
50
100
150
200
L1 Norm

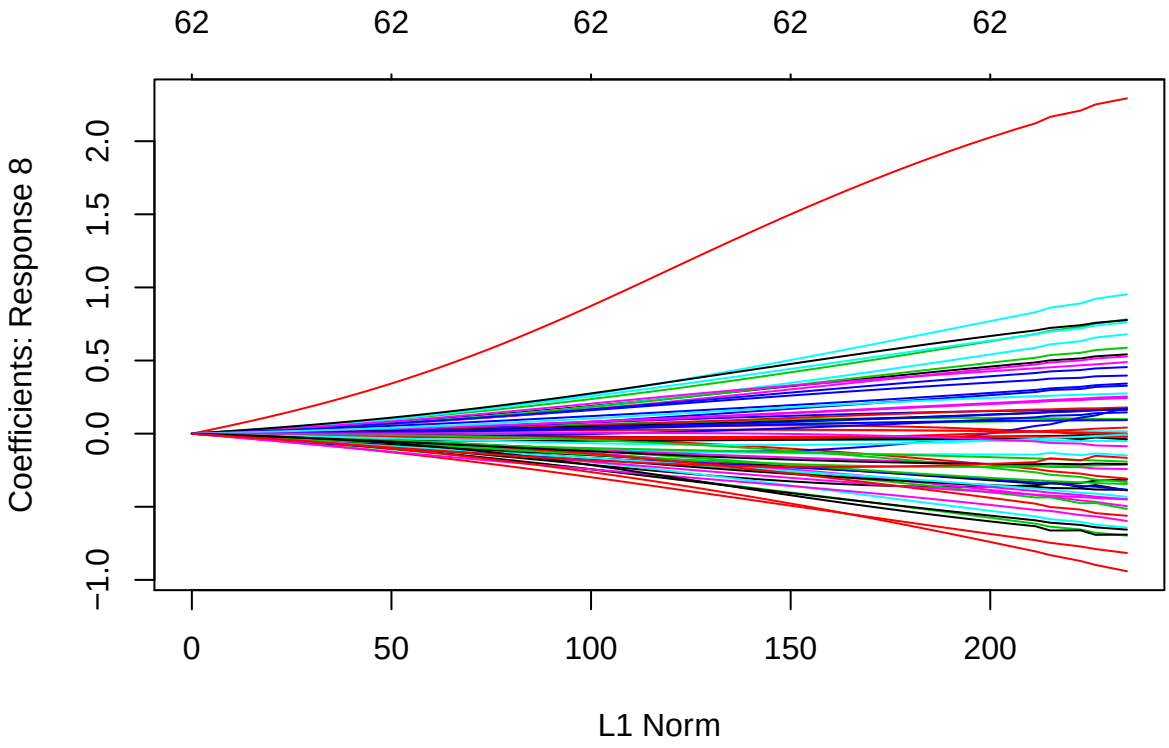

62
62
62
62
62
Coefficients: Response 3
0.5
0.0
−0.5
−1.0
0
50
100
150
200
L1 Norm


62
62
62
62
62
Coefficients: Response 4
−0.5
0.0
0.5
1.0
1.5
0
50
100
150
200
L1 Norm

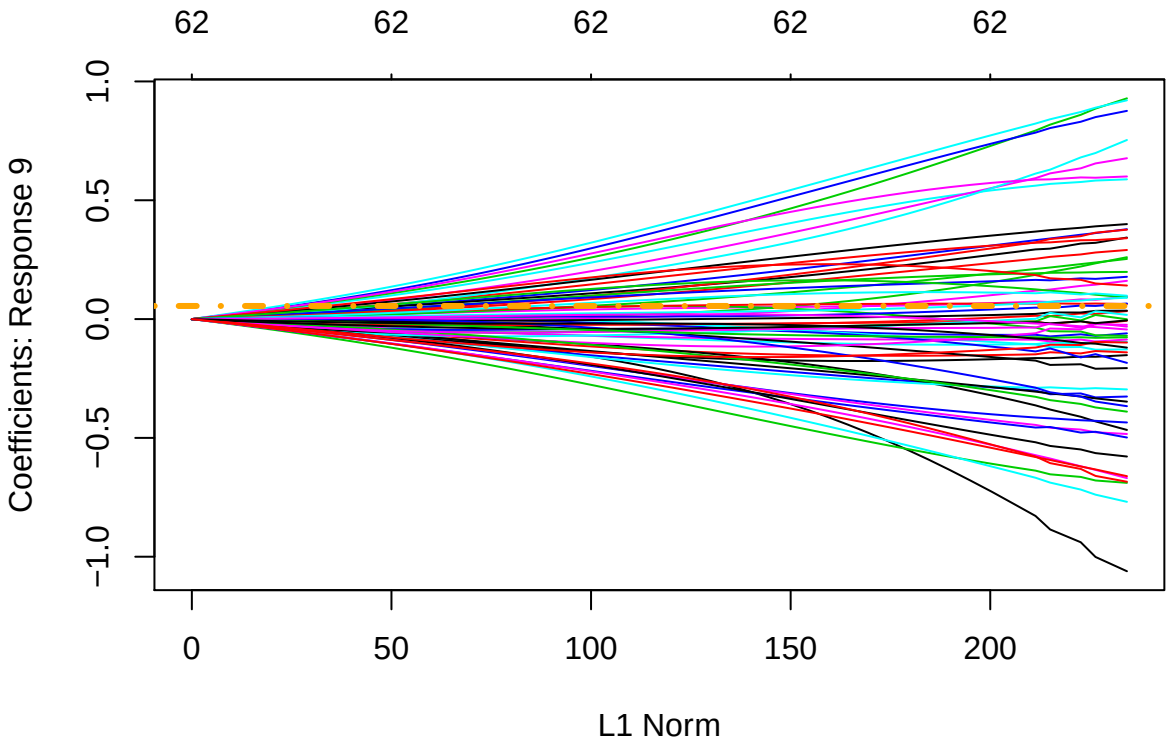

62
62
62
62
62
Coefficients: Response 5
1.0
0.5
0.0
−0.5
−1.0
0
50
100
150
200
L1 Norm


62
62
62
62
62
Coefficients: Response 6
−0.5
0.0
0.5
1.0
1.5
0
50
100
150
200
L1 Norm


62
62
62
62
62
Coefficients: Response 7
0.5
0.0
−0.5
−1.0
0
50
100
150
200
L1 Norm




```
abline(h=best_lambda_rd, col= "orange", lty=4, lwd=3)
```




```
paste ("El mejor lambda es:", best_lambda_rd)
```

```
## [1] "El mejor lambda es: 0.0552244371003232"
```

Los anteriores gráficos (uno para cada clase) muestran los valores que toma lambda por cada variable predictora. A nosotros solo nos interesan aquellas que esten por encima de un umbral definido por nosotros. La linea naranja es dicho umbral, el mejor lambda. Las lineas que superan dicho umbral son aquellas variables que son mas significativas para ajustar el modelo. Cuando la norma L1 (ejeX) es baja, todas las variables nos dicen lo mismo (estimador nulo o aleatorio, es decir, el modelo no sabe nada realmente).

Explicación de los parámetros de la función cv.glmnet:

lambda -> Son los valores de lambda usados para ajustar

cvm -> La media de los errores de validación cruzada (cross validation measure), es un vector de tamaño length(lambda)

cvsd -> Cross validation standard error de cvm, es la estimación del error estandar de cvm

cvup -> Curva superior que básicamente es cvm+csvd

cvlo -> Curva inferior que es básicamente cvm-csvd

nzero-> Cantidad de coeficientes que nos son cero con cada lambda

glmnet.fit -> Un objeto ajustado para todos los datos dados

lambda.min -> Valor del lambda que hace mínimo cvm

lambda.1se -> Mayor valor de lambda tal que el error estandarizado esta dentro de 1

8. Métrica

Una vez ajustado el modelo, es momento de medir nuestro modelo. Como métrica hemos decidido usar **matriz de confusión** ya que nos permite mostrar de forma explicita cuando una clase es confundida con otra (falsos positivos/negativos), es decir, permite trabajar de forma separada con distintos tipos de error.

```
# Predicciones

#Predicciones modelo con regularización
pred_lasso_train=predict(cvfit_ls, newx = features_train, type= "class", s="lambda.min")

confusionMatrix(data = as.factor(pred_lasso_train), reference = as.factor(labels_train))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1   2   3   4   5   6   7   8   9
##          0 374   0   0   0   0   0   0   0   0   1
##          1   0 379   0   0   0   1   2   0   4   4
##          2   0   1 378   0   0   0   0   0   0   0
##          3   0   0   1 382   0   1   0   1   0   1
##          4   1   1   0   0 386   0   1   0   2   2
##          5   0   0   0   4   0 371   0   0   1   1
##          6   1   1   0   0   1   0 374   0   0   0
##          7   0   1   0   1   0   0   0 386   0   1
##          8   0   4   0   0   0   0   0   0 373   2
##          9   0   2   1   2   0   3   0   0   0 370
##
## Overall Statistics
##
##                Accuracy : 0.9869
##                  95% CI : (0.9828, 0.9903)
##     No Information Rate : 0.1018
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9855
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity           0.99468  0.97429  0.99474  0.98201   0.9974  0.98670
## Specificity           0.99971  0.99680  0.99971  0.99884   0.9980  0.99826
## Pos Pred Value        0.99733  0.97179  0.99736  0.98964   0.9822  0.98408
## Neg Pred Value        0.99942  0.99709  0.99942  0.99796   0.9997  0.99855
## Prevalence            0.09835  0.10175  0.09940  0.10175   0.1012  0.09835
## Detection Rate        0.09783  0.09914  0.09888  0.09992   0.1010  0.09704
## Detection Prevalence  0.09809  0.10201  0.09914  0.10097   0.1028  0.09861
## Balanced Accuracy     0.99720  0.98554  0.99722  0.99042   0.9977  0.99248
##                      Class: 6 Class: 7 Class: 8 Class: 9
## Sensitivity           0.99204   0.9974  0.98158  0.96859
## Specificity           0.99913   0.9991  0.99826  0.99768
## Pos Pred Value        0.99204   0.9923  0.98417  0.97884
## Neg Pred Value        0.99913   0.9997  0.99797  0.99652
## Prevalence            0.09861   0.1012  0.09940  0.09992
```

```
## Detection Rate        0.09783   0.1010  0.09757  0.09678
## Detection Prevalence  0.09861   0.1018  0.09914  0.09888
## Balanced Accuracy     0.99559   0.9983  0.98992  0.98313
```

```
pred_lasso_train=predict(cvfit_rd, newx = features_train, type= "class", s="lambda.min")

confusionMatrix(data = as.factor(pred_lasso_train), reference = as.factor(labels_train))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1   2   3   4   5   6   7   8   9
##          0 374   0   0   0   0   1   0   0   0   1
##          1   0 360   1   0   1   1   4   1  15   8
##          2   0   5 369   1   0   1   0   1   0   0
##          3   0   0   0 374   0   1   0   2   1   5
##          4   1   0   0   0 370   0   2   0   3   6
##          5   0   0   1   4   0 355   0   0   1   1
##          6   1   2   0   0   3   0 370   0   3   0
##          7   0   2   1   1   1   0   0 383   0   8
##          8   0   8   7   2   4   0   1   0 356   4
##          9   0  12   1   7   8  17   0   0   1 349
##
## Overall Statistics
##
##                Accuracy : 0.9574
##                  95% CI : (0.9505, 0.9635)
##     No Information Rate : 0.1018
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9526
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity           0.99468  0.92545  0.97105  0.96144  0.95607  0.94415
## Specificity           0.99942  0.99097  0.99768  0.99738  0.99651  0.99797
## Pos Pred Value        0.99468  0.92072  0.97878  0.97650  0.96859  0.98066
## Neg Pred Value        0.99942  0.99155  0.99681  0.99564  0.99506  0.99393
## Prevalence            0.09835  0.10175  0.09940  0.10175  0.10123  0.09835
## Detection Rate        0.09783  0.09417  0.09652  0.09783  0.09678  0.09286
## Detection Prevalence  0.09835  0.10228  0.09861  0.10018  0.09992  0.09469
## Balanced Accuracy     0.99705  0.95821  0.98436  0.97941  0.97629  0.97106
##                      Class: 6 Class: 7 Class: 8 Class: 9
## Sensitivity           0.98143   0.9897  0.93684  0.91361
## Specificity           0.99739   0.9962  0.99245  0.98663
## Pos Pred Value        0.97625   0.9672  0.93194  0.88354
## Neg Pred Value        0.99797   0.9988  0.99303  0.99037
## Prevalence            0.09861   0.1012  0.09940  0.09992
## Detection Rate        0.09678   0.1002  0.09312  0.09129
## Detection Prevalence  0.09914   0.1036  0.09992  0.10332
## Balanced Accuracy     0.98941   0.9929  0.96465  0.95012
```

La lectura de la matriz nos muestra como el modelo ha predicho muy bien las distintas muestras de train,

clasificando erroneamente muy pocos digitos respecto al tamaño del conjunto.

Como vemos, tenemos varias métricas obtenidas a través de la matriz de confusión que nos seran muy utiles para medir nuestro modelo:

- precision: numero de predicciones correctas entre el numero total de predicciones.
- Sensibilidad y especificidad: valores que indican la capacidad de nuestro estimador para discriminar los casos positivos de los negativos.

La sensibilidad se puede decir que es la tasa de verdaderos positivos. La especificidad como la tasa de verdaderos negativos.

9. Estimacion del error Eout.

Veamos como se comporta nuestro modelo:

```
#Predicciones
#Predicciones modelo sin regularización
predict_train=predict(rln, newdata = features_train)
aciertos_train=predict_train==labels_train
sprintf("El error dentro de la muestra (Ein) es %s", (length(aciertos_train[aciertos_train==F]) / lengt
```

```
## [1] "El error dentro de la muestra (Ein) es 0"
```

```
predict_test=predict(rln, newdata = features_test)
aciertos_test=predict_test==labels_test
sprintf("El error fuera de la muestra (Eout) es %s", (length(aciertos_test[aciertos_test==F]) / length(
```

```
## [1] "El error fuera de la muestra (Eout) es 9.34891485809683"
```

```
# Predicciones lasso
pred_lasso_train_ls=predict(cvfit_ls, newx = features_train, type= "class", s="lambda.min")

pred_lasso_test_ls=predict(cvfit_ls, newx = features_test, type= "class", s="lambda.min")

#Variables mas significativas
#cvfit$`1`>best_lambda

aciertos_train=pred_lasso_train_ls==labels_train
sprintf("LASSO -> El error dentro de la muestra (Ein) es %s", (length(aciertos_train[aciertos_train==F]
```

```
## [1] "LASSO -> El error dentro de la muestra (Ein) es 1.30787339785509"
```

```
aciertos_test=pred_lasso_test_ls==labels_test
sprintf("LASSO -> El error fuera de la muestra (Eout) es %s", (length(aciertos_test[aciertos_test==F]) /
```

```
## [1] "LASSO -> El error fuera de la muestra (Eout) es 4.61880912632165"
```

```
# Predicciones ridge
pred_ridge_train=predict(cvfit_rd, newx = features_train, type= "class", s="lambda.min")

pred_ridge_test=predict(cvfit_rd, newx = features_test, type= "class", s="lambda.min")

#Variables mas significativas
#cvfit$`1`>best_lambda

aciertos_train=pred_ridge_train==labels_train
sprintf("RIDGE -> El error dentro de la muestra (Ein) es %s", (length(aciertos_train[aciertos_train==F]
```

```
## [1] "RIDGE -> El error dentro de la muestra (Ein) es 4.26366727700759"
```

```
aciertos_test=pred_ridge_test==labels_test
sprintf("RIDGE -> El error fuera de la muestra (Eout) es %s", (length(aciertos_test[aciertos_test==F]) /
```

```
## [1] "RIDGE -> El error fuera de la muestra (Eout) es 6.62214802448525"
```

Como vemos, con el modelo normal sin regularizar, tenemos sobreajuste (error dentro de la muestra 0 y al tener ruido por no regularizar), por lo que hace falta regularización. Los errores nos muestran que nuestro modelo regularizado con ridge se comporta peor que regularizado con lasso, es porque lasso directamente elimina variables (características) y ridge tiene algo mas de ruido, es decir, nos quedamos con el **modelo de regresión logística con regularización Lasso**.

10. Calidad del modelo

Se puede afirmar que el modelo es de una calidad excepcional, ya que se utilizan librerias muy probadas y testeadas que utilizan algoritmos excelentes para llevar a cabo la tarea de clasificacion. En este caso se comporta tan bien ya que el conjunto a clasificar, aunque no es perfectamente linear-separable, haciendo breves transformaciones (internamente la libreria) consigue dejar el conjunto prácticamente linear-separable , de ahi que prediga tan bien. Otra cosa que ayuda a que el modelo se comporte tan bien es que el conjunto de datos usado no tenga prácticamente ruido, es decir, los digitos usados son de bastante calidad, alomejor si testeamos el modelo con digitos con ruido o poco visibles, el modelo no se comporta tan bien.

PROBLEMA DE REGRESIÓN: base de datos "Airfoil Self Noise"

1. Problema a resolver

El problema, planteado incicialmente por la NASA, trata de, dados unos ejemplos medidos realmente en un tunerl de viento, predecir el ruido (dB) producido por la interacción de un ala de avión y las turbulencias a su alrededor en torno a su perfil aerodinamico.

Las caracterísitcas medidas son las siguientes: frecuencia (Hz), ángulo de ataque (grados), profundidad del ala o chord lenght (metros), velocidad maxima libre de turbulencias (m/s) y desplazamiento lateral debido a la succion y grosor del ala (metros).

Usaremos en principio un modelo de regresión lineal normal sin usar regularización y lo compararemos con un modelo de regresion lineal usando regularizacion.

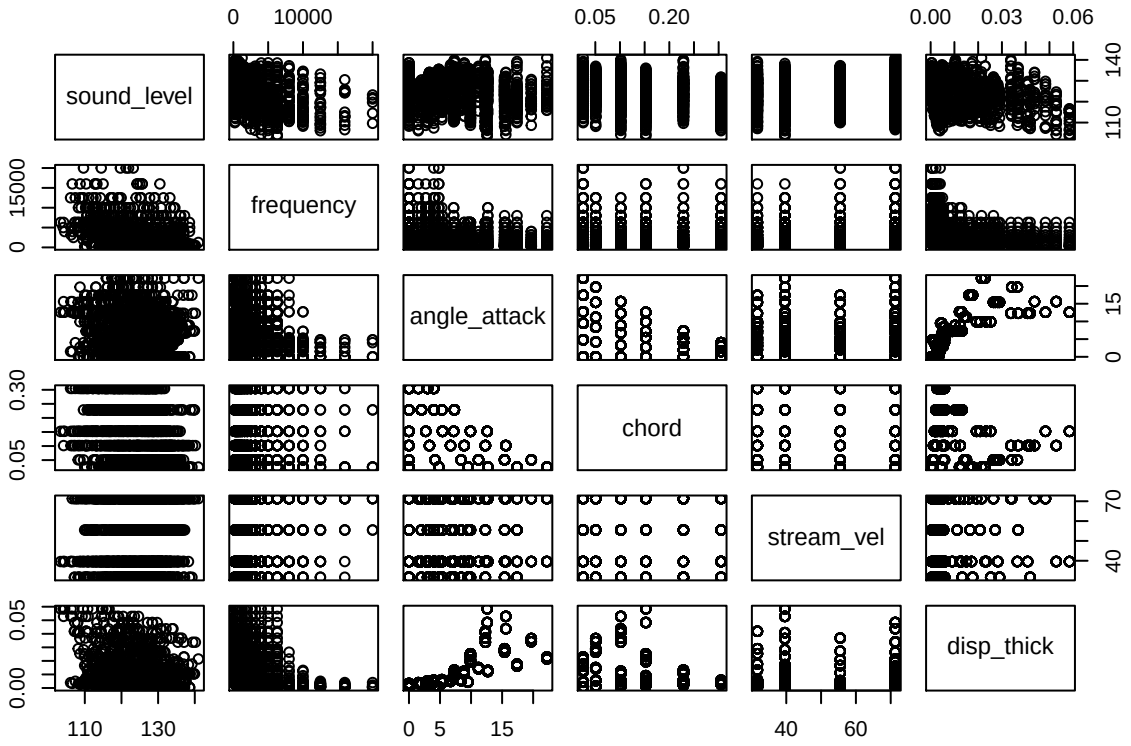
2. Preprocesamiento de los datos.

En este caso, no hace falta normalizar (explicar por qué, creo que estaria bien decirle que los datos estan medidos en sus respectivos rangos y todas las variables son cuantitativas, por lo tanto, no es necesario normalizar).

En probabilidad y estadística, la correlación indica la fuerza y la dirección de una relación lineal y proporcionalidad entre dos variables estadísticas. Se considera que dos variables cuantitativas están correlacionadas cuando los valores de una de ellas varían sistemáticamente con respecto a los valores homónimos de la otra: si tenemos dos variables (A y B) existe correlación entre ellas si al disminuir los valores de A lo hacen también los de B y viceversa. La correlación entre dos variables no implica, por sí misma, ninguna relación de causalidad.

Procedemos a ver la correlacion de las caracteristicas para quitar alguna de estas en caso de que algunas si lo esten, es decir, de que sean dependientes y puedan empeorar la calidad del modelo ajustado.

```r
# Leemos datos

datos = read.csv("datos/airfoil_self_noise.csv", header = FALSE)
names(datos)[1] = "frequency"
names(datos)[2] = "angle_attack"
names(datos)[3] = "chord"
names(datos)[4] = "stream_vel"
names(datos)[5] = "disp_thick"
names(datos)[6] = "sound_level"

pairs(sound_level~., data=datos)
```




```r
descrCor = cor(datos) # -> Con esto vemos la correlación de las variables con las otras y ella misma (d

summary(descrCor)
```

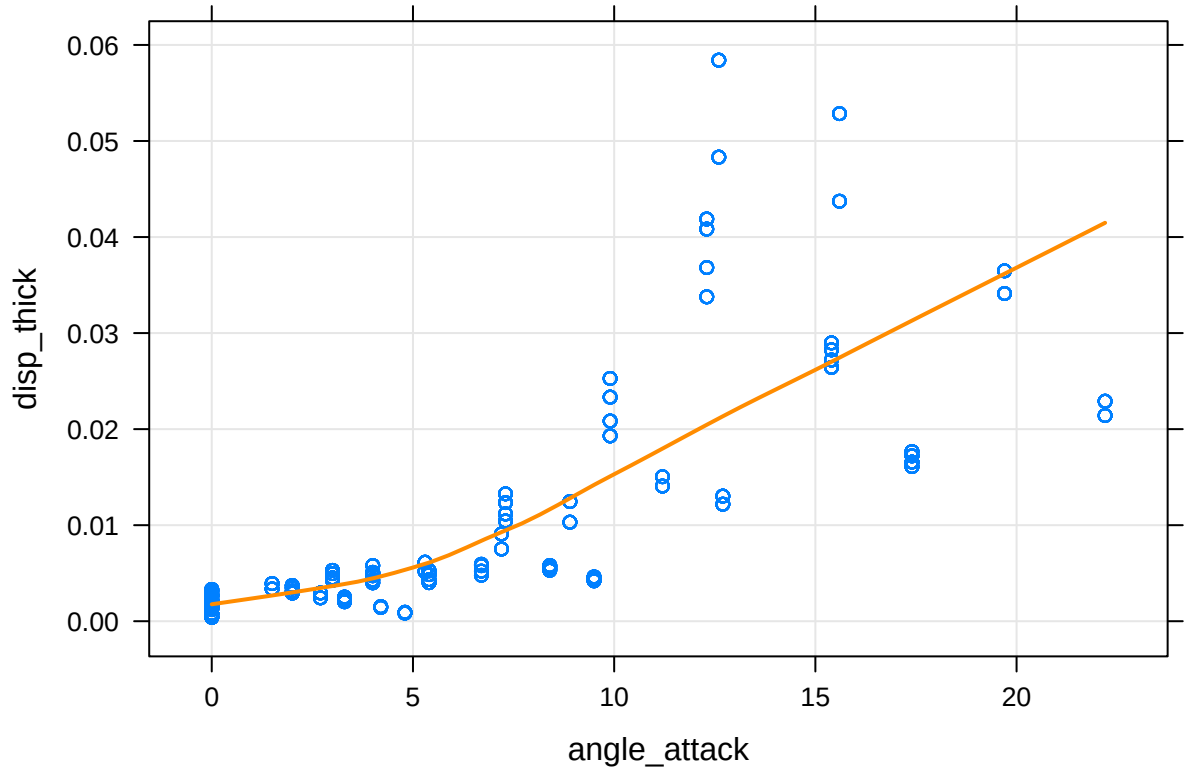
```
##    frequency         angle_attack          chord
##  Min.   :-0.39071   Min.   :-0.50487   Min.   :-0.504868
##  1st Qu.:-0.26210   1st Qu.:-0.24360   1st Qu.:-0.232332
##  Median :-0.11688   Median :-0.04867   Median :-0.112252
##  Mean   : 0.03940   Mean   : 0.14640   Mean   : 0.006376
##  3rd Qu.: 0.09933   3rd Qu.: 0.57974   3rd Qu.: 0.001925
##  Max.   : 1.00000   Max.   : 1.00000   Max.   : 1.000000
##    stream_vel          disp_thick        sound_level
##  Min.   :-0.003974   Min.   :-0.3127   Min.   :-0.390711
##  1st Qu.: 0.017530   1st Qu.:-0.2278   1st Qu.:-0.293542
```

```
##  Median : 0.091931   Median :-0.1124   Median :-0.196134
##  Mean   : 0.219556   Mean   : 0.1643   Mean   : 0.004909
##  3rd Qu.: 0.131524   3rd Qu.: 0.5641   3rd Qu.: 0.054800
##  Max.   : 1.000000   Max.   : 1.0000   Max.   : 1.000000
```

```
#Si establecemos un punto de corte en el que decidamos cuando dos variables son dependientes (correlaci
highlyCorDescr <- findCorrelation(descrCor, cutoff = .75)
sum(highlyCorDescr)
```

```
## [1] 2
```

```
xyplot(disp_thick~angle_attack,datos,grid=T,type = c("p", "smooth"),, col.line = "darkorange",lwd = 2)
```




```
datos=datos[,-highlyCorDescr]
```

Se utilzia pairs(), forma mas visual de verlo y cor(), forma estadística para ver la correlación entre variables.

Como se ve en la grafica, las variables “disp_thick” y “angle attack” son dependientes unas de otra, por tanto, hay que quitar una de ellas. Decidimos eliminar “angle of attack”.

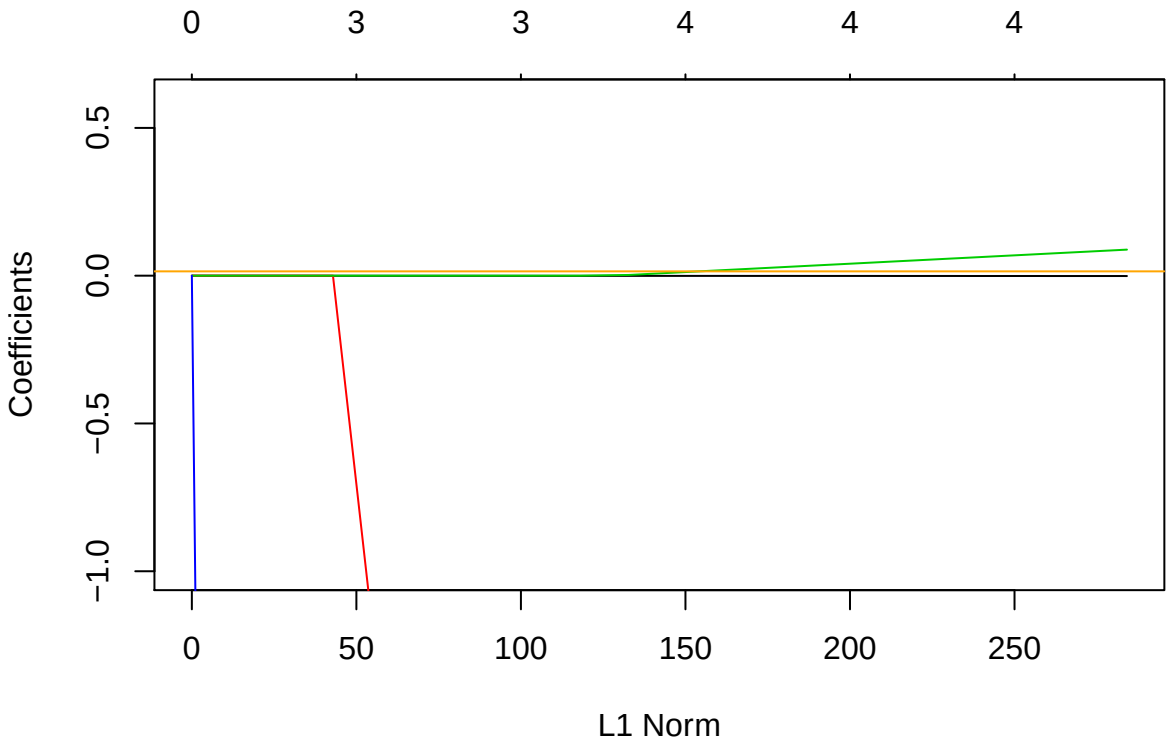
3. Selección de clases de funciones a usar.

Como hemos comentado anteriormente, usaremos regresión lineal normal en un princpio y regresión lineal regularizada (con lasso) para compararlos y decidir con qué modelo quedarnos.

4. Conjuntos de training, validacion y test usados.

Separaremos los datos, manteniendo la proporcion de valores igual en cada conjunto, en una proporcion 70/30.

```
#separamos los datos en conjuntos de train y test

# Dividimos los datos en train y test ya que no tenemos un dataset de train:
set.seed(1000)
train.index = createDataPartition(datos$`sound_level`, p = 0.7, list = FALSE)   # Igual que StratifiedK
train = as.matrix(datos[train.index,])
test = as.matrix(datos[-train.index,])

features_train=train[,-ncol(train)]
features_test=test[,-ncol(test)]
labels_train=train[,ncol(train)]
labels_test=test[,ncol(test)]

# Cross validation:

nfolds = 10  # Para elegir el mejor lambda

cv = cv.glmnet(x = train[,-ncol(train)], y = train[,ncol(train)], alpha = 1, nfolds = nfolds)

best_lambda = cv$lambda.min

plot(cv$glmnet.fit, "norm", ylim = c(-1,0.6), col = c(1:ncol(train)))
abline(h = best_lambda, col = "orange")
```




```
sprintf("Mejor lambda en las %s particiones es = %s", nfolds, best_lambda)
```

```
## [1] "Mejor lambda en las 10 particiones es = 0.0146844224312334"
```

La gráfica nos muestra que quita radicalmente dos de variables mas, cosa que puede perjudicar a nuestro modelo seriamente.

5. Regularización, modelo a usar e hyperparámetros.

En un principio, la regularización en este caso no parece necesaria ya que tenemos pocas características. Aplicaremos regularización para comparar un modelo con el otro. Como ya sabemos la función de regularización depende de un término lambda, dicho término (hiperparámetro) lo estimaremos usando validación cruzada.

Procedemos a ajustar el modelo usando regresión normal sin regularización, con la variable "angle of attack" ya eliminada.

```
#Ajuste del modelo de regresion lineal sin regularizar
ans_reg = train(sound_level ~ ., data=train, method="lm")

#summary(ans_reg)
```

8. Métrica

Root Mean Square Error: es la desviación estándar de los residuos (errores de predicción). Los residuos son una medida de cuán lejos están los puntos de datos de la línea de regresión; RMSE es una medida de la dispersión de estos residuos. En otras palabras, le dice qué tan concentrado está la información en la línea de

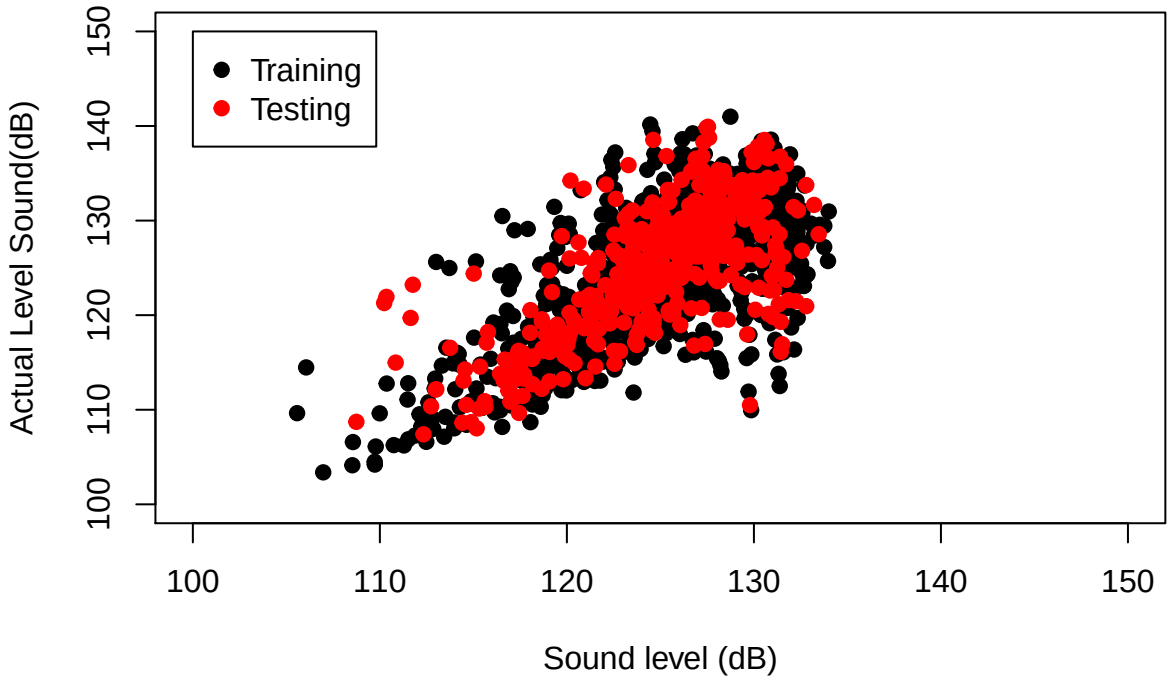
mejor ajuste. El error cuadrático medio se usa comúnmente en la climatología, la predicción y el análisis de regresión para verificar los resultados experimentales.

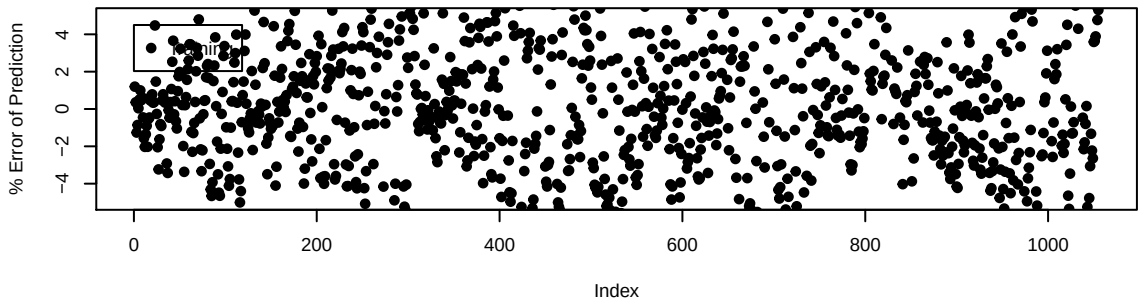
Los valores de error que obtenemos nos dicen que, de la recta de regresión de nuestro modelo, las predicciones se desvian de la recta en una media del error obtenido (SRM).

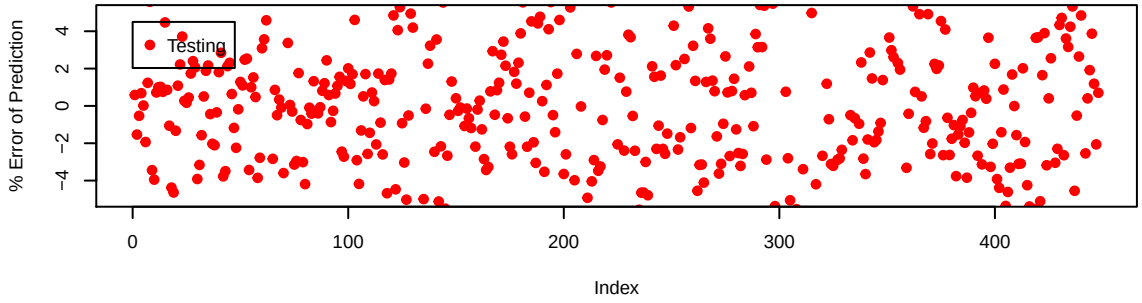
9. Estimacion del error Eout y calidad del modelo.

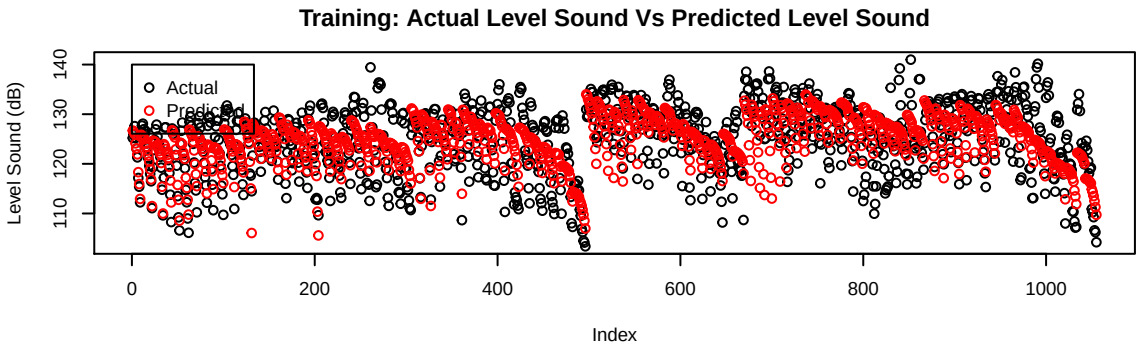
```
# Funcion para evaluar nuestro modelo ya ajustado (errores)


eval_model <- function(model) {
    pred_train <- predict(model,newdata = features_train)
    pred_test <- predict(model,newdata = features_test)


    # Graficas de dispersion para training y test
    plot(pred_train,labels_train,xlim=c(100,150),ylim=c(100,150),col=1,
         pch=19,xlab = "Sound level (dB)",ylab = "Actual Level Sound(dB)")
    points(pred_test,labels_test,col=2,pch=19)
    leg <- c("Training","Testing")
    legend(100, 150, leg, col = c(1, 2),pch=c(19,19))

    # Scatter plots of % error on predictions on Training and Testing sets
    par(mfrow = c(2, 1))
    par(cex = 0.6)
    par(mar = c(5, 5, 3, 0), oma = c(2, 2, 2, 2))
    plot((pred_train - labels_train)* 100 /labels_train,
         ylab = "% Error of Prediction", xlab = "Index",
         ylim = c(-5,5),col=1,pch=19)
    legend(0, 4.5, "Training", col = 1,pch=19)
    plot((pred_test-labels_test)* 100 /labels_test,
         ylab = "% Error of Prediction",  xlab = "Index",
         ylim = c(-5,5),col=2,pch=19)
    legend(0, 4.5, "Testing", col = 2,pch=19)

    # Actual data Vs Predictions superimposed for Training and Testing Data
    plot(1:length(labels_train),labels_train,pch=21,col=1,
         main = "Training: Actual Level Sound Vs Predicted Level Sound",
         xlab = "Index",ylab = "Level Sound (dB)")
    points(1:length(labels_train),pred_train,pch=21,col=2)
    #leg <- c("Training","Predicted Training")
    legend(0, 140, c("Actual","Predicted"), col = c(1, 2),pch=c(21,21))
    plot(1:length(labels_test),labels_test,pch=21,col=1,
         main = "Testing: Actual Level Sound Vs Predicted Level Sound",
         xlab = "Index",ylab = "Level Sound (dB)")
    points(1:length(labels_test),pred_test,pch=21,col="red")
    legend(0, 140, c("Actual","Predicted"), col = c(1, 2),pch=c(21,21))

    ## Line graph of errors
    plot(pred_train-labels_train,type='l',ylim=c(-5,+5),
         xlab = "Index",ylab = "Actual - Predicted",main="Training")
    plot(pred_test-labels_test,type='l',ylim=c(-5,+5),
```

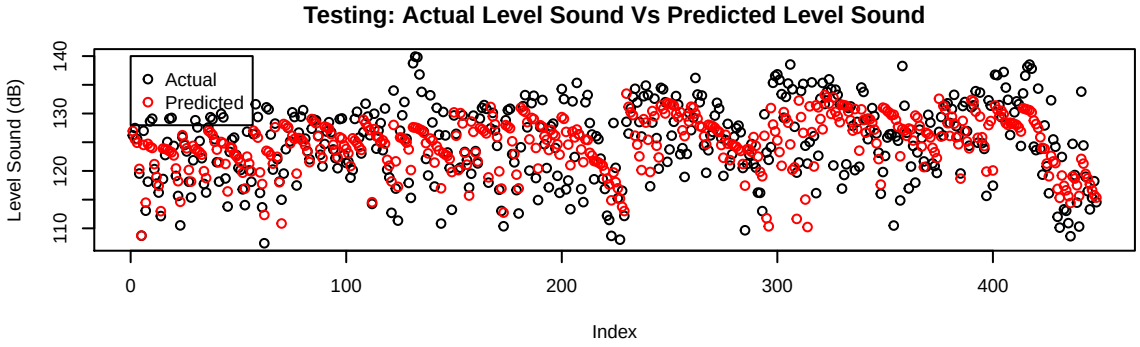
```
         xlab = "Index",ylab = "Actual - Predicted",main="Testing")

    ISRME<- sqrt(mean((pred_train-labels_train)^2))
    OSRME<- sqrt(mean((pred_test-labels_test)^2))

    return(c(ISRME,OSRME))
}

# Calidad del modelo en una única division.

#Errores fuera y dentro de la muestra

sprintf("Error dentro del train = %s", eval_model(ans_reg)[1])
```

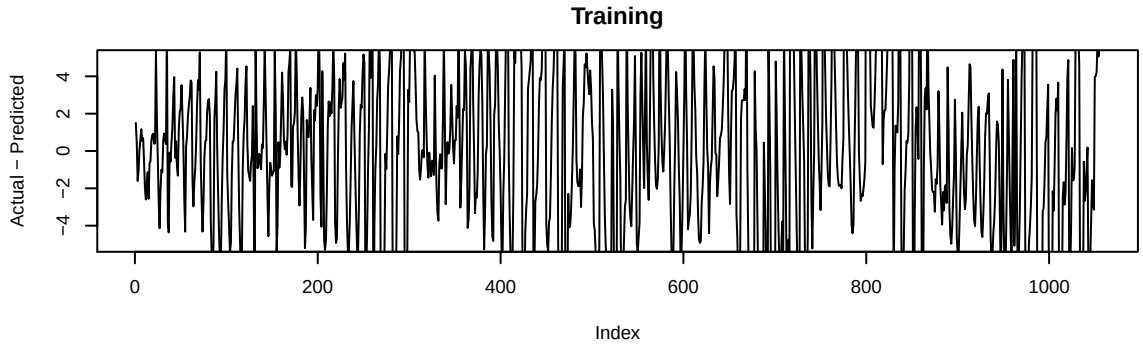


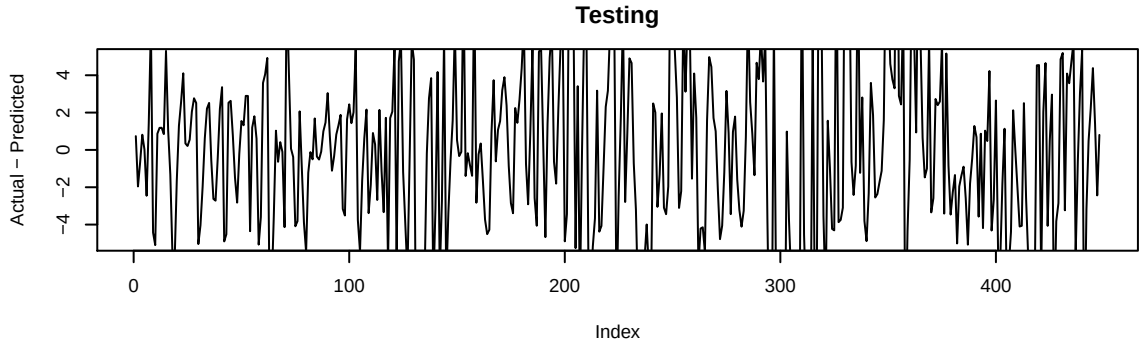

% Error of Prediction
Training
4
2
0
−2
−4
0
200
400
600
800
1000
Index


% Error of Prediction
Testing
4
2
0
−2
−4
0
100
200
300
400
Index


Training: Actual Level Sound Vs Predicted Level Sound
Actual
Predicted
Level Sound (dB)
110
120
130
140
0
200
400
600
800
1000
Index

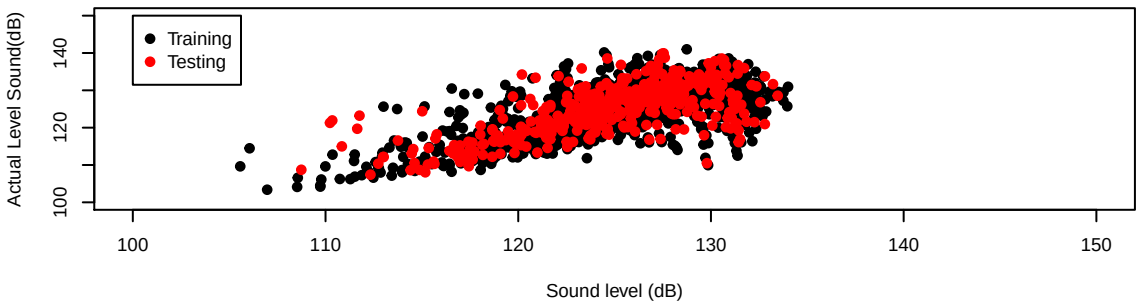

Testing: Actual Level Sound Vs Predicted Level Sound
Actual
Predicted
Level Sound (dB)
110
120
130
140
0
100
200
300
400
Index

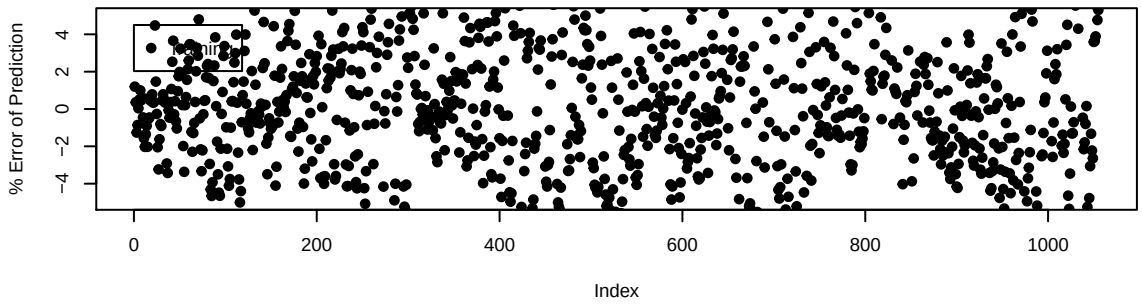


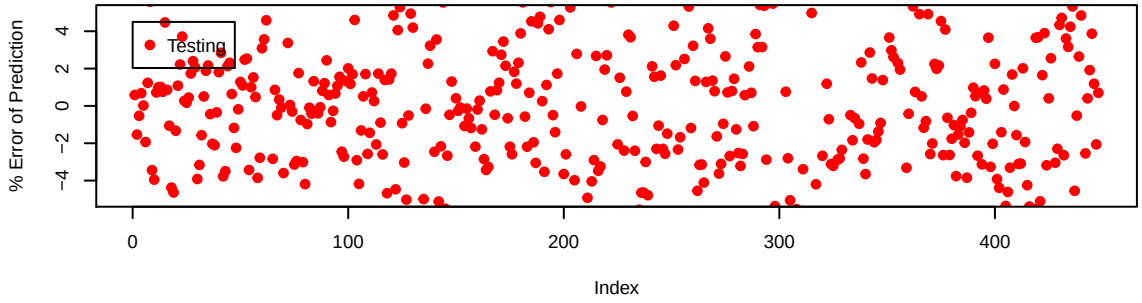


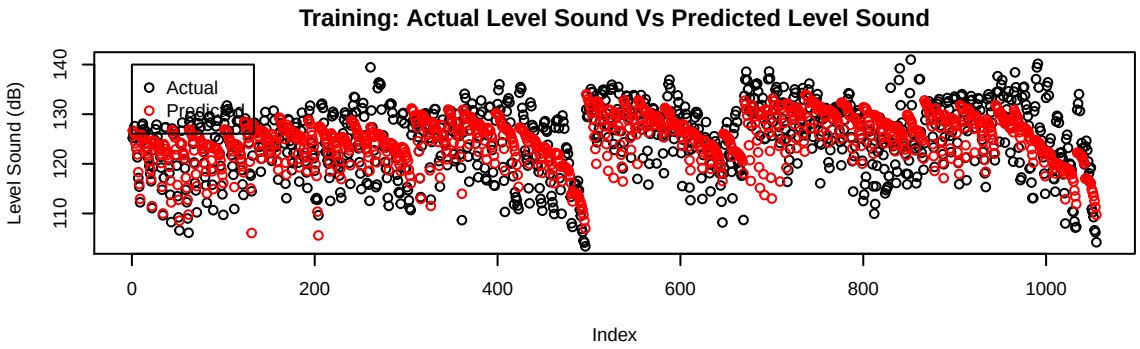

```
## [1] "Error dentro del train = 5.00403417495364"
```

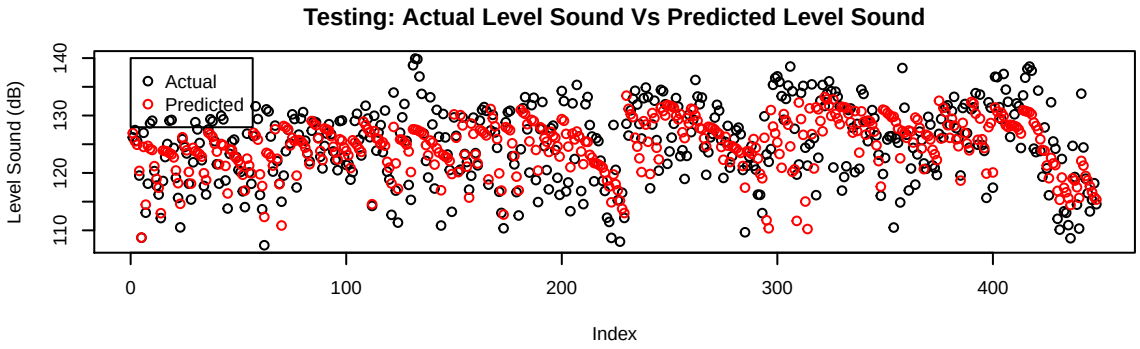
```
sprintf("Error dentro del test = %s", eval_model(ans_reg)[2])
```

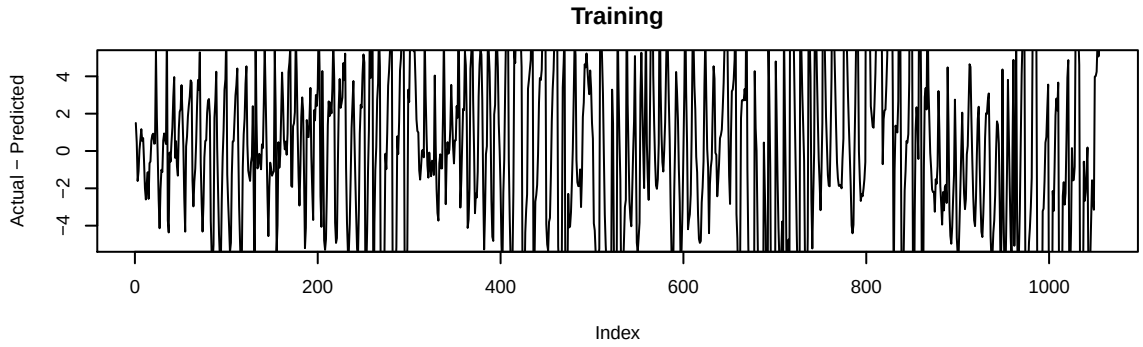

Training
Testing
Actual Level Sound(dB)
100
120
140
100
110
120
130
140
150
Sound level (dB)

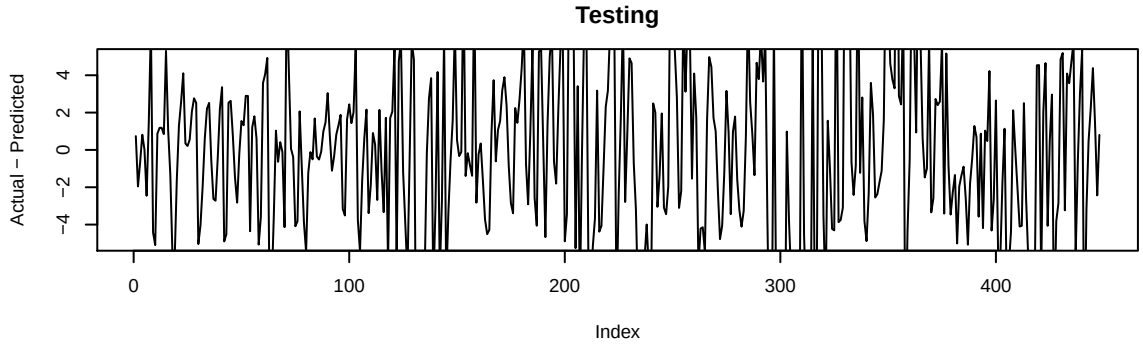

% Error of Prediction
Training
4
2
0
−2
−4
0
200
400
600
800
1000
Index


% Error of Prediction
Testing
4
2
0
−2
−4
0
100
200
300
400
Index


Training: Actual Level Sound Vs Predicted Level Sound
Actual
Predicted
Level Sound (dB)
110
120
130
140
0
200
400
600
800
1000
Index


Testing: Actual Level Sound Vs Predicted Level Sound
Actual
Predicted
Level Sound (dB)
110
120
130
140
0
100
200
300
400
Index






```
## [1] "Error dentro del test = 4.94747002025112"
```

```
# Calidad del modelo usando Cross-Validation
nparticiones = 5

folds = cut(seq(1, nrow(train)), breaks=nparticiones, labels=F)

#barajamos
folds=sample(folds)

# Comprobación errores cross validation

ein = eout = 0

for (i in 1:nparticiones) {
    testIndexes=which(folds==i, arr.ind = T)
    trainIndexes=which(folds!=i, arr.ind = T)

    train_cv=train[trainIndexes,]
    test_cv=train[testIndexes,]

    cv_reg = train(sound_level ~ ., data=train_cv, method="lm")

    pred_train <- predict(cv_reg,newdata = train_cv)
    pred_test <- predict(cv_reg,newdata = test_cv)

    ISRME<- sqrt(mean((pred_train-train_cv[,5])^2))
```

```
    OSRME<- sqrt(mean((pred_test-test_cv[,5])^2))

    ein = ein + ISRME
    eout = eout + OSRME

    sprintf("Error dentro del train, particion %s es %s", i, ISRME)
    sprintf("Error dentro del test, particion %s es %s", i, OSRME)
}

sprintf("Ein medio con %s particiones = %s", nparticiones, ein/nparticiones)
```

```
## [1] "Ein medio con 5 particiones = 4.99651436166632"
```

```
sprintf("Eout medio con %s particiones = %s", nparticiones, eout/nparticiones)
```

```
## [1] "Eout medio con 5 particiones = 5.05011072278605"
```

Como podemos ver, el error tomando un conjunto de train y test del dataset proporcionado es mucho menor que el error medido con Cross-Validation, ya que este obtiene el error medio tomando conjuntos de test-train cinco veces en este caso, por lo que podemos ver que nuestro modelo se comporta medianamente bien ya que el ruido predicho **se puede desviar en unos 14-15 dB de media**, del ruido real que pudiera generarse.