

Practica 3

Jose Antonio Ruiz Millan

15 de mayo de 2018

1. Ajuste de modelos lineales.

Este ejercicio se centra en el ajuste de un modelo lineal a conjuntos de datos dadas con el objetivo de obtener el mejor predictor posible. En todos los casos los pasos a desarrollar serán aquellos que nos conduzcan al ajuste y selección del mejor modelo y a la estimación del error Eout del modelo final. Cómo mínimo se habrán de analizar y comentar los siguientes pasos sobre un problema de clasificación y otro de regresión:

Clasificación

1. Comprender el problema a resolver.

En este caso, tenemos un conjunto de datos que corresponden a imágenes de números (del 0 al 9), donde para cada elemento, tenemos 65 variables de las cuales 64 son información del elemento y la última la etiqueta del mismo. Estos elementos representan una imagen de 32x32 que a su vez está dividida en bloques de 4x4, lo que hace que tengamos almacenado para cada elemento una matriz 8x8, donde cada casilla tiene un valor que representa el número de píxeles “pintados” en cada uno de los bloques (0...16).

Tenemos que ser capaces de a través de un modelo lineal, separar estos números y hacer una buena clasificación de los mismos.

2. Preprocesado los datos: por ejemplo categorización, normalización, etc.

Los datos que tenemos son un poco abstractos, ya que al tener una porción de 8x8 en la que cada casilla es una de 4x4, tenemos un valor que define a la matriz completa 4x4 pero no sabemos qué posiciones son las que están dibujadas. Por ello, realizaremos un preprocesado de datos para facilitar la clasificación al predictor.

Para ello, utilizaremos una función específica que se utiliza para este tipo de ocasiones llamada `preProcess()`. Esta función pertenece al paquete *caret* y nos permite realizar el prepoceso de los datos con distintas funcionalidades. En mi caso, las funcionalidades a utilizar serán las siguientes:

- **nzv:** Nos permite eliminar los atributos que tengan varianza 0 o muy cercana a 0. Esto quiere decir que los atributos en los que no haya diferencia para determinar de qué clase podría ser un nuevo elemento basándonos en esa propia columna será eliminada.

- (**YeoJohnson:**) Realiza unos cambios en los datos muy similares al BoxCox (que a su vez, es equivalente a la familia de transformaciones de la potencia), pero permite que existan valores negativos y el cero. Para ello, se basa en la siguiente función.

$$\begin{aligned} & - ((y + 1)^\lambda - 1) / \lambda \text{ si } \lambda \neq 0, y \geq 0 \\ & - \log(y + 1) \text{ si } \lambda = 0, y \geq 0 \\ & - [(-y + 1)^{2-\lambda} - 1] / (2 - \lambda) \text{ si } \lambda \neq 2, y < 0 \\ & - -\log(-y + 1) \text{ si } \lambda = 2, y < 0 \end{aligned}$$

- **center:** Resta la media de los datos a los propios datos, para tener la información un poco más concentrada pero sin perder el propio valor del dato.

- **scale:** Divide los datos por la desviación estándar. Consigue centrar más aún los datos con el objetivo de tener una visión de los mismos más acorde con la realidad.

- **pca**: Nos permite reducir el espacio de dimensiones, ésto se lo hace cojiendo las variables más importantes mientras consigan explicar un 95

En primer lugar, cargaremos los datos del train para posteriormente realizarle el preprocesamiento.

```
#Cargamos los datos
clas.train <- read.csv("optdigits_tra.csv",header = FALSE,sep = ",")

#Creamos los cambios que se van a realizar.
clas.cambios <- preProcess(clas.train[,-ncol(clas.train)],method = c("nzv","YeoJohnson","center","scale"))
print(clas.cambios)

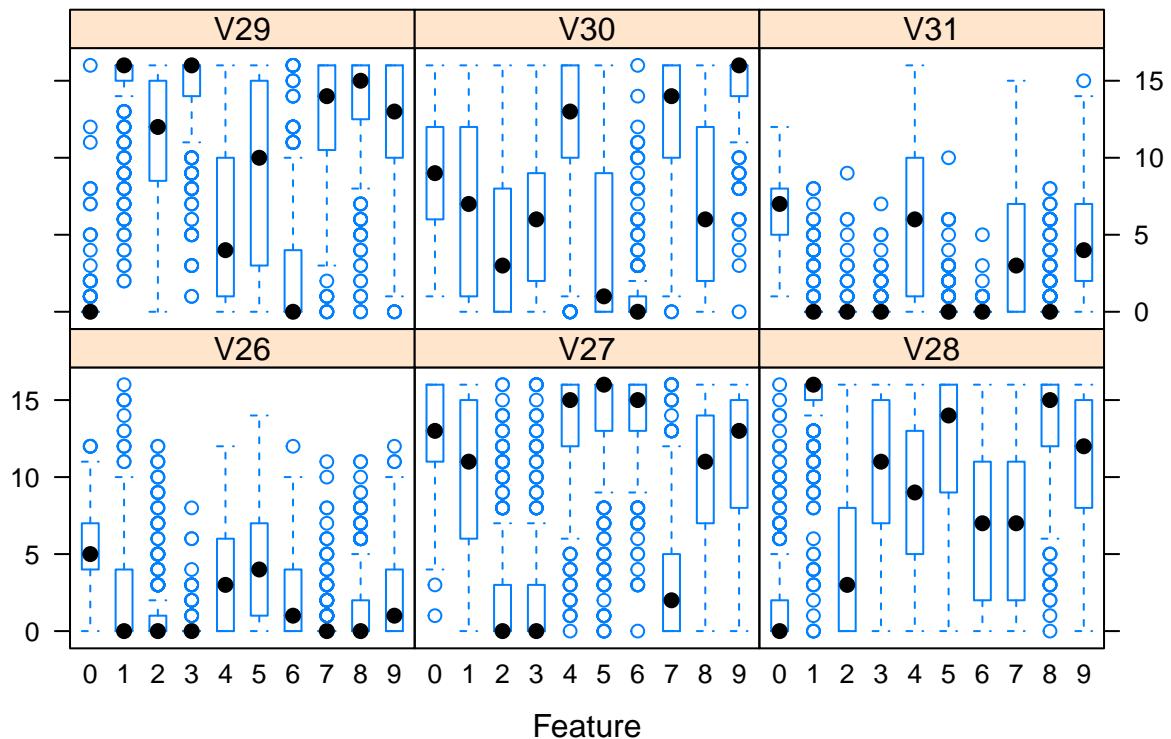
## Created from 3823 samples and 64 variables
##
## Pre-processing:
##   - centered (48)
##   - ignored (0)
##   - principal component signal extraction (48)
##   - removed (16)
##   - scaled (48)
##   - Yeo-Johnson transformation (46)
##
## Lambda estimates for Yeo-Johnson transformation:
##      Min. 1st Qu. Median 3rd Qu. Max.
## -2.6960 -0.3414  0.4648  0.2634  0.8722  1.9120
##
## PCA needed 30 components to capture 95 percent of the variance
# Aplicamos los cambios (Combinamos con cbind los cambios de los datos con las etiquetas de los mismos)
clas.trainPre <- as.data.frame(cbind(predict(clas.cambios,as.matrix(clas.train[,-ncol(clas.train)])),clas.train[,ncol(clas.train)]))
```

Para comentar un poco los datos y las transformaciones, vamos a realizar unos gráficos de los datos antes de ser transformados y después. No obstante, no vamos a mostrar todas las variables ya que tienen una dimensión muy elevada y sería imposible dibujarlas en un mismo gráfico.

1. Diagrama de caja:

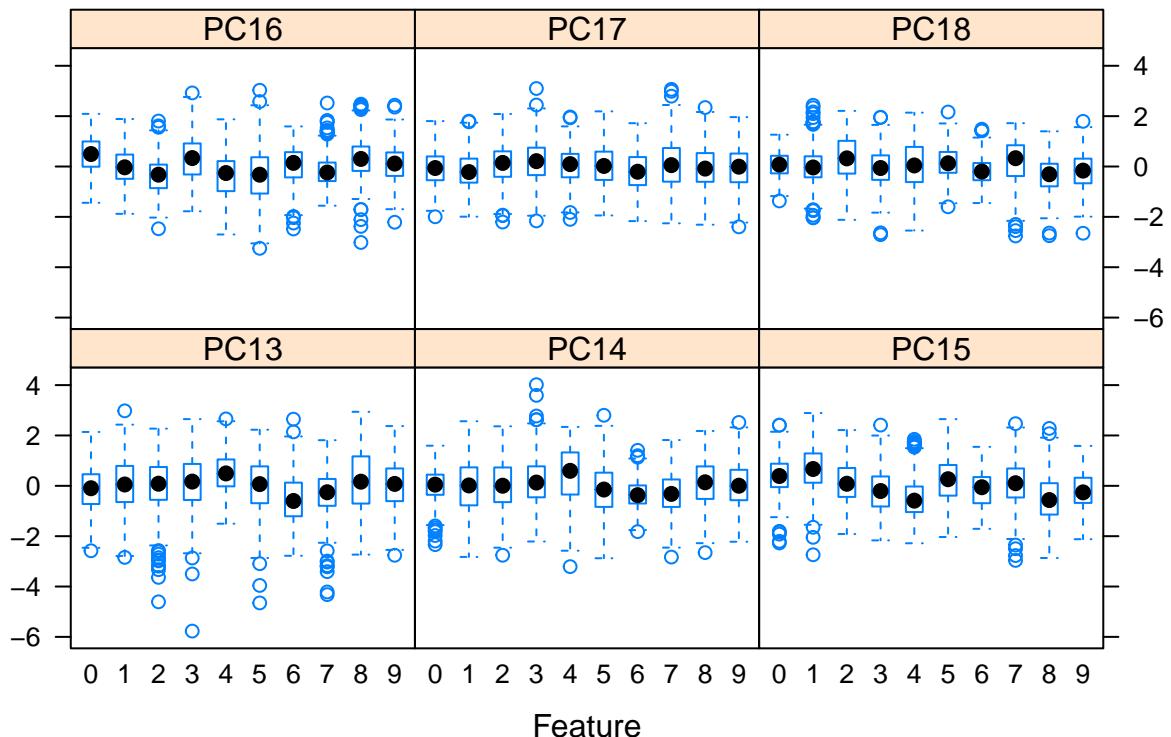
```
#Train sin preprocessado.
featurePlot(x=clas.train[,26:31],y=as.factor(clas.train[,ncol(clas.train)]),plot="box",main = "Diagrama de caja")
```

Diagrama de caja (TRAIN)



```
#Train con preprocesado  
featurePlot(x=clas.trainPre[,13:18],y=as.factor(clas.trainPre[,ncol(clas.trainPre)]),plot="box",main =
```

Diagrama de caja (TRAIN–PREPROC)



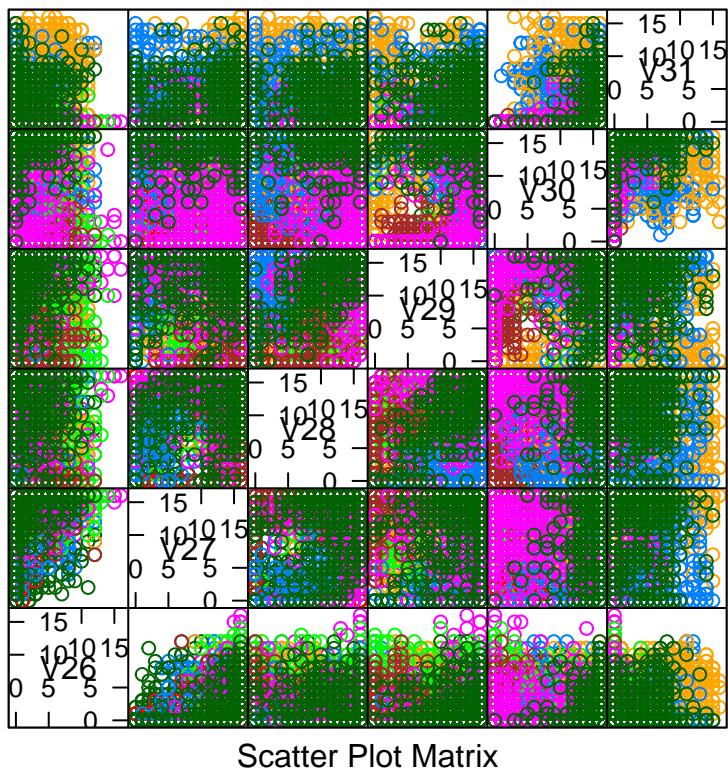
Claramente podemos ver las grandes diferencias entre los gráficos, donde para el train sin preprocesar tenemos unos valores para las variables mucho más dispersos y con mucha dispersión mientras que en el train preprocesado tenemos unos valorse mucho más medianos y concentrados.

2. Diagrama de dispersión.

```
#Train sin preprocesado.
```

```
featurePlot(x=clas.train[,26:31],y=as.factor(clas.train[,ncol(clas.train)]),plot="pairs",main = "Diagrama de dispersión")
```

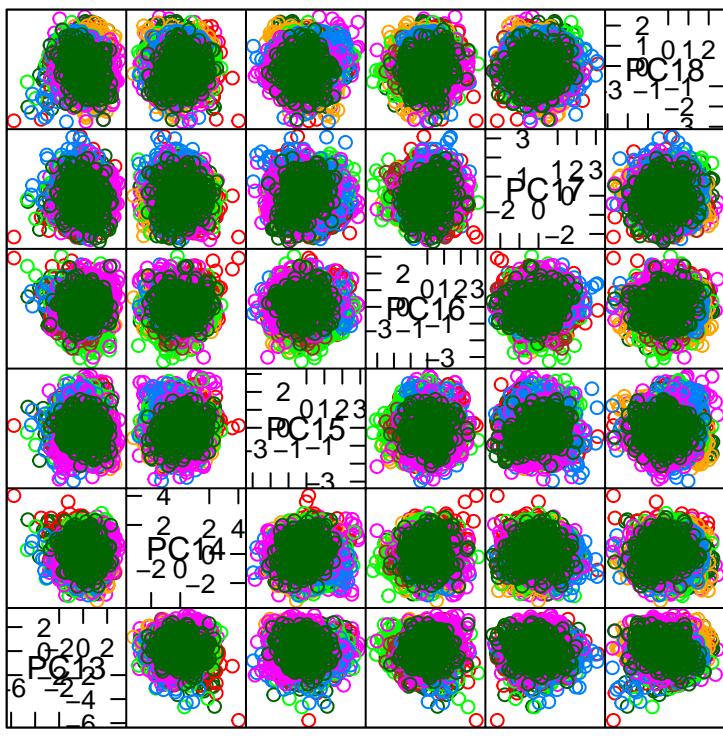
Diagrama de dispersión (TRAIN)



#Train con preprocesado

```
featurePlot(x=clas.trainPre[,13:18],y=as.factor(clas.trainPre[,ncol(clas.trainPre)]),plot="pairs",main =
```

Diagrama de dispersión (TRAIN-PREPROC)



Scatter Plot Matrix

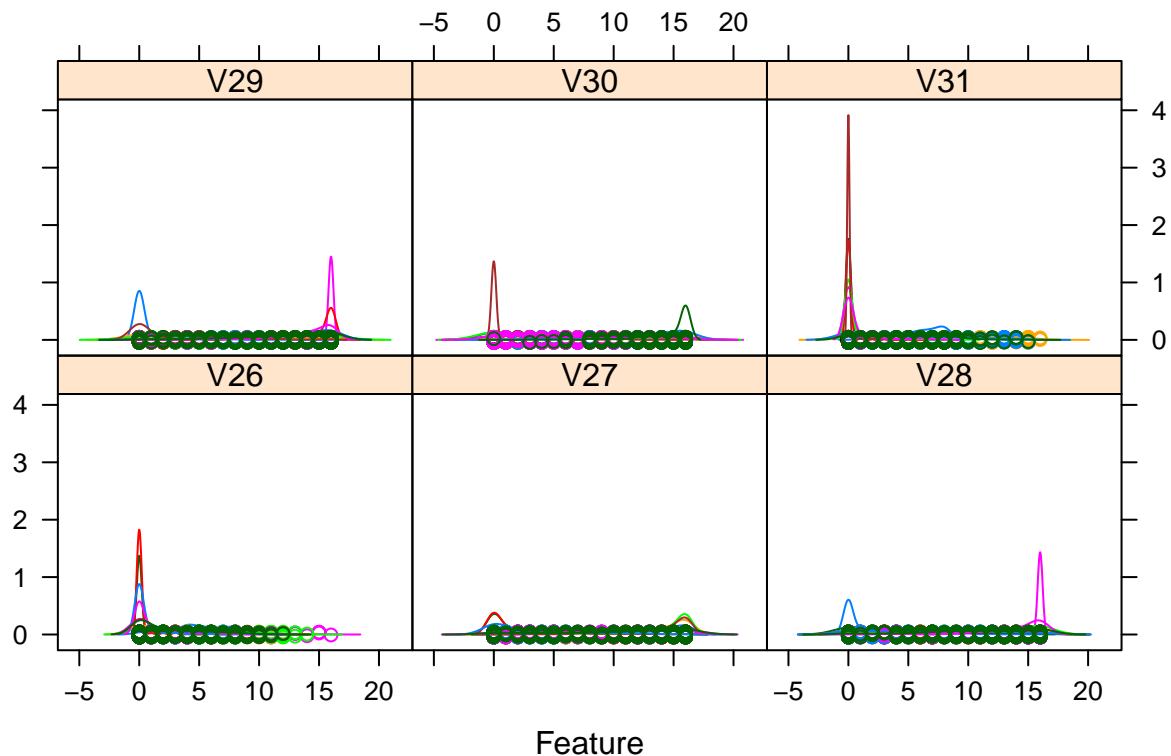
Como en el caso anterior, tenemos la misma conclusión respecto a los datos.

3. Por último, diagrama de densidad.

#Train sin preprocesado.

```
featurePlot(x=clas.train[,26:31],y=as.factor(clas.train[,ncol(clas.train)]),plot="density",main = "Diag
```

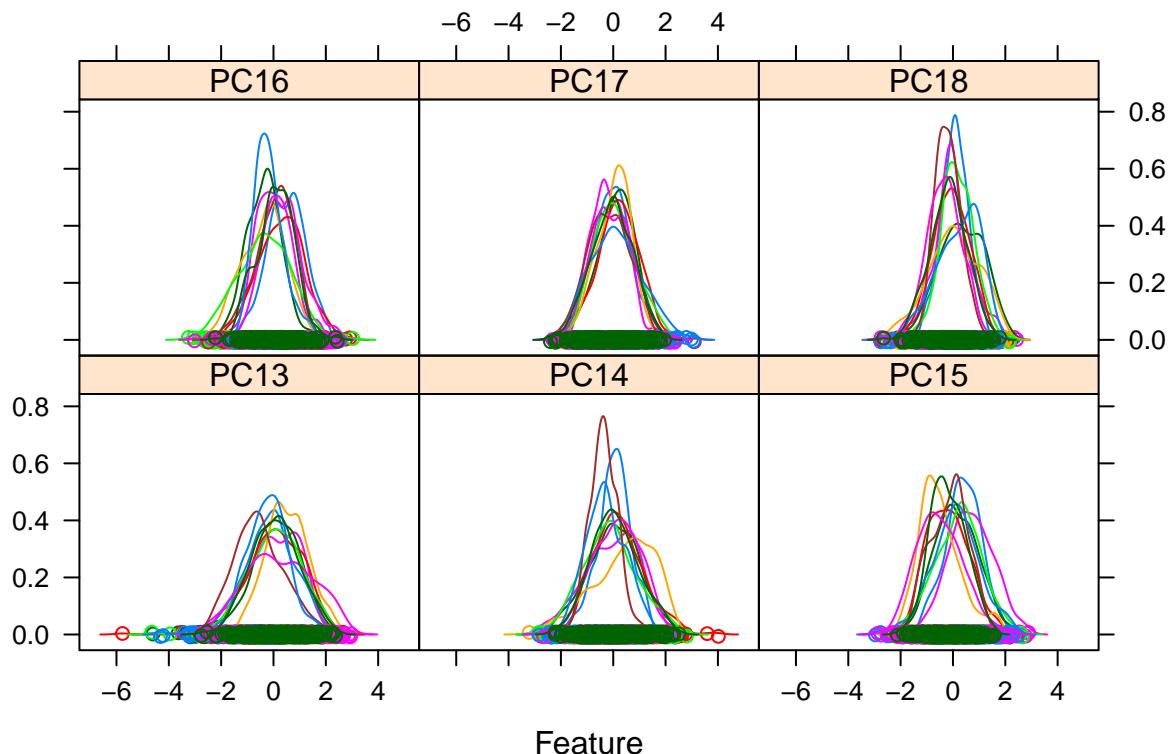
Diagrama de densidad (TRAIN)



```
#Train con preprocesado
```

```
featurePlot(x=clas.trainPre[,13:18],y=as.factor(clas.trainPre[,ncol(clas.trainPre)]),plot="density",main=
```

Diagrama de densidad (TRAIN–PREPROC)



Podemos concluir finalmente, que hemos conseguido representar los datos de una forma más expresiva y teniendo en cuenta que **hemos pasado de tener 64 variables ha tener 30**, con lo que hemos conseguido reducir la dimensión en un **53%**

3. Selección de clases de funciones a usar.

Como se ha comentado en clase, utilizaremos las clases de funciones lineales para clasificar los datos que tenemos. Con estas funciones intentaremos separar los datos que tenemos para poder realizar una clasificación decente, teniendo en cuenta la dimensionalidad del problema, que en mi caso es de 30 gracias al preprocesado realizado.

4. Definición de los conjuntos de training, validación y test usados en su caso.

El conjunto de datos que tenemos está repartido en training y test de forma predeterminada, para el cual hay 3823 observaciones en el train y 1797 en el test. En el train los tenemos con la siguiente distribución:

0: 376

1: 389

2: 380

3: 389

4: 387

5: 376

6: 377

7: 387

8: 380

9: 382

En el test tenemos la siguiente distribucion:

0: 178

1: 182

2: 177

3: 183

4: 181

5: 182

6: 181

7: 179

8: 174

9: 180

Por esta razón, tanto el training como el test los tenemos bien definidos. Por otra parte, el conjunto de validación será creado y seleccionado a la hora de hacer cross-validation ya que las propias funciones que lo realizan se encargan de hacer las particiones y seleccionarlo.

5. Discutir la necesidad de regularización y en su caso la función usada para ello.

La regularización es un campo que debemos tener en cuenta, esto es porque nos permite ajustar el modelo de tal forma que no sobreajustemos a la muestra que tenemos de entrenamiento y así no perjudicar a las posibles predicciones en el futuro sobre muestras de test una vez se ponga en uso el modelo. En mi caso, voy a realizar dos modelos distintos, uno basado en L1 regularization (Regularización Lasso) y otro basado en L2 regularization (Regularización Ridge).

- L1 regularization: Añade una penalización igual a la suma del valor absoluto de los coeficientes.

$$Error_{L1} = Error + \lambda \sum_{i=0}^N |\beta_i|$$

- L2 regularization: Añade una penalización igual a la suma de los coeficientes al cuadrado.

$$Error_{L2} = Error + \lambda \sum_{i=0}^N |\beta_i|^2$$

Por lo que tendremos dos modelos distintos, que finalmente elegiremos uno de ellos comparando el E_{in} que obtengamos.

6. Definir los modelos a usar y estimar sus parámetros e hyperparámetros.

En mi caso, el modelo que he decidido utilizar es regresión logística, ya que de los modelos lineales es el que más se ajusta a este problema, ya que utilizando un modelo multinomial, conseguimos obtener una probabilidad de pertenecer a cada una de las clases y finalmente elegir la que más probabilidad tenga. Como he indicado anteriormente, utilizaré dos modelos distintos de la regresión logística multinomial, uno que utiliza L1 regularization y otro que utiliza L2 regularization. Para ello, utilizaré la función (`cv.glmnet`) donde dependiendo del parámetro α que le pasemos realizará una regularización u otra, siendo $\alpha = 1$ L1 y $\alpha = 0$ L2.

```

#Entrenamos con cross-validation
clas.mod.L1 <- cv.glmnet(x = as.matrix(clas.trainPre[,-ncol(clas.trainPre)]),y = as.factor(clas.trainPre))
summary(clas.mod.L1)

##          Length Class   Mode
## lambda     99    -none- numeric
## cvm        99    -none- numeric
## cvsd       99    -none- numeric
## cvup       99    -none- numeric
## cvlo       99    -none- numeric
## nzero      99    -none- numeric
## name        1    -none- character
## glmnet.fit 15   multnet list
## lambda.min  1    -none- numeric
## lambda.1se  1    -none- numeric

clas.mod.L2 <- cv.glmnet(x = as.matrix(clas.trainPre[,-ncol(clas.trainPre)]),y = as.factor(clas.trainPre))
summary(clas.mod.L2)

##          Length Class   Mode
## lambda     99    -none- numeric
## cvm        99    -none- numeric
## cvsd       99    -none- numeric
## cvup       99    -none- numeric
## cvlo       99    -none- numeric
## nzero      99    -none- numeric
## name        1    -none- character
## glmnet.fit 15   multnet list
## lambda.min  1    -none- numeric
## lambda.1se  1    -none- numeric

```

7. Selección y ajuste modelo final.

Ya tenemos los dos modelos creados, ahora vamos a calcular el E_{in} con los dos modelos distintos que hemos creado para finalmente seleccionar unos de ellos y terminar de entrenarlo.

```

#Funcion para calcular el error
clas.E_in <- function(datos,predicciones){
  err <- 0

  for(i in 1:length(predicciones)){
    if(datos[i,ncol(datos)] != predicciones[i]) err <- err+1
  }
  err <- err/length(predicciones)
  err
}

#Predecimos las etiquetas del train utilizando el modelo creado anteriormente
clas.L1.pred <- predict.cv.glmnet(clas.mod.L1,as.matrix(clas.trainPre[,-ncol(clas.trainPre)]),type="class")
#Calculamos el error.
clas.L1.E_in <- clas.E_in(clas.trainPre,clas.L1.pred)
print(paste("Error de clasificación en train con L1 regularization: ",clas.L1.E_in))

## [1] "Error de clasificación en train con L1 regularization:  0.0256343185979597"
#Predecimos las etiquetas del train utilizando el modelo creado anteriormente
clas.L2.pred <- predict.cv.glmnet(clas.mod.L2,as.matrix(clas.trainPre[,-ncol(clas.trainPre)]),type="class")

```

```

#Calculamos el error.
clas.L2.E_in <- clas.E_in(clas.trainPre,clas.L2.pred)
print(paste("Error de clasificación en train con L2 regularization: ",clas.L2.E_in))

## [1] "Error de clasificación en train con L2 regularization: 0.0457755689249281"

En mi caso, tengo un error menor utilizando el modelo con regularización L1. Por lo que ahora voy a entrenar con todos los datos y éste será mi modelo final. Guardaremos también el valor de lambda para las futuras predicciones, utilizando el que nos devuelve el mejor modelo obtenido anteriormente.

#Entrenamos con todos los datos (NO cross-validation)
clas.mod <- glmnet(x = as.matrix(clas.trainPre[,-ncol(clas.trainPre)]),y = c(clas.trainPre[,ncol(clas.trainPre)]))

##          Length Class  Mode
## a0           1000  -none- numeric
## beta         10    -none- list
## dfmat        1000  -none- numeric
## df           100  -none- numeric
## dim           2    -none- numeric
## lambda       100  -none- numeric
## dev.ratio    100  -none- numeric
## nulldev      1    -none- numeric
## npasses       1    -none- numeric
## jerr          1    -none- numeric
## offset        1    -none- logical
## classnames   10   -none- character
## grouped       1    -none- logical
## call          5    -none- call
## nobs          1    -none- numeric

#Guardamos el lambda del mejor modelo anterior para utilizarlo en las predicciones futuras.
clas.lambda <- as.double(clas.mod.L1[9])
clas.lambda

## [1] 0.0003348937

#Comprobamos el E_in finalmente obtenido.
clas.mod.pred <- predict(clas.mod,as.matrix(clas.trainPre[,-ncol(clas.trainPre)]),type="class",s=clas.lambda)

#Calculamos el error.
clas.mod.E_in <- clas.E_in(clas.trainPre,clas.mod.pred)
print(paste("Error de clasificación en train con modelo final: ",clas.mod.E_in))

## [1] "Error de clasificación en train con modelo final: 0.0211875490452524"

8. Discutir la idoneidad de la métrica usada en el ajuste

Para ello, vamos a utilizar la matriz de confusión que nos permite ver perfectamente como el predictor está realizando su trabajo y tambien podemos ver dónde específicamente se está equivocando el clasificador.

confusionMatrix(data=as.factor(clas.trainPre[,ncol(clas.trainPre)]), reference = as.factor(clas.mod.pred))

## Confusion Matrix and Statistics
##
##          Reference
## Prediction  0   1   2   3   4   5   6   7   8   9
##          0 374   0   0   0   1   0   1   0   0   0
##          1   0 375   0   1   2   0   1   1   6   3

```

```

##      2   0   1 376   1   0   0   0   0   1   1
##      3   0   0   1 380   0   4   0   1   2   1
##      4   0   0   0   0 378   0   3   0   1   5
##      5   0   0   2   2   0 366   1   0   2   3
##      6   0   2   0   0   1   0 374   0   0   0
##      7   0   0   0   1   0   0   0 386   0   0
##      8   0   7   0   0   3   2   1   0 367   0
##      9   0   4   1   1   5   0   0   1   4 366
##
## Overall Statistics
##
##          Accuracy : 0.9788
## 95% CI : (0.9737, 0.9831)
## No Information Rate : 0.102
## P-Value [Acc > NIR] : < 2.2e-16
##
##          Kappa : 0.9765
## McNemar's Test P-Value : NA
##
## Statistics by Class:
##
##          Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity      1.00000  0.96401  0.98947  0.9845  0.96923  0.98387
## Specificity       0.99942  0.99592  0.99884  0.9974  0.99738  0.99710
## Pos Pred Value    0.99468  0.96401  0.98947  0.9769  0.97674  0.97340
## Neg Pred Value    1.00000  0.99592  0.99884  0.9983  0.99651  0.99826
## Prevalence        0.09783  0.10175  0.09940  0.1010  0.10201  0.09731
## Detection Rate    0.09783  0.09809  0.09835  0.0994  0.09888  0.09574
## Detection Prevalence 0.09835  0.10175  0.09940  0.1018  0.10123  0.09835
## Balanced Accuracy  0.99971  0.97997  0.99416  0.9909  0.98330  0.99049
##
##          Class: 6 Class: 7 Class: 8 Class: 9
## Sensitivity      0.98163  0.9923  0.9582  0.96570
## Specificity       0.99913  0.9997  0.9962  0.99535
## Pos Pred Value    0.99204  0.9974  0.9658  0.95812
## Neg Pred Value    0.99797  0.9991  0.9954  0.99622
## Prevalence        0.09966  0.1018  0.1002  0.09914
## Detection Rate    0.09783  0.1010  0.0960  0.09574
## Detection Prevalence 0.09861  0.1012  0.0994  0.09992
## Balanced Accuracy  0.99038  0.9960  0.9772  0.98053

```

Podemos ver por los resultados, que hemos obtenido un buen modelo, ya que vemos en la matriz que para cada una de las clases acierta prácticamente todos los números aunque vemos que el número 9 es el que más problemas está teniendo aunque sean mínimos. Por la información que nos proporciona la función, vemos que tenemos un acierto de 98% y pudiendo afirmar con un 95% de confianza el el rango del porcentaje de acierto estará en [97.46, 98.38], que es un valor excelente. También nos ofrece más datos estadísticos que podemos comprobar que nos informan básicamente de que hemos realizado un buen ajuste para los datos y teniendo en cuenta que en pasos anteriores, conseguimos reducir el espacio de dimensiones un 53%. También podemos sacar de esta información la sensibilidad y la especificidad que es lo que nos indica la curva ROC, para cada una de las clases, y vemos en este caso que la proporción de falsos negativos y de falsos positivos es mínima, cosa que indica de nuevo que tenemos un buen ajuste.

9. Estimación del error E_{out} del modelo lo más ajustada posible.

Para ello, vamos a leer los datos del test, realizarle las transformaciones de los datos que creamos con el train, calcularemos las predicciones con el modelo creado anteriormente y por último calcularemos el E_{out} con la

función definida en apartados anteriores que nos devuelve el porcentaje de error en clasificación de los datos totales.

```
#Cargamos los datos
clas.test <- read.csv("optdigits_tes.csv",header = FALSE,sep = ",")

#Realizamos las transformaciones de los datos creadas con el train
clas.testPre <- as.data.frame(cbind(predict(clas.cambios,as.matrix(clas.test[,-ncol(clas.test)])),clas.))

#Predecimos las nuevas etiquetas
clas.test.predic <- predict(clas.mod,as.matrix(clas.testPre[, -ncol(clas.testPre)]),type="class",s=clas.s)
```

Ya tenemos las etiquetas predichas, por lo que ahora vamos a calcular el E_{out} y también crearemos la matriz de confusión únicamente para tener más visible que ha realizado.

```
#Calculamos el error
clas.test.Err <- clas.E_in(clas.testPre,clas.test.predic)
print(paste("Error de clasificación en test: ",clas.test.Err))

## [1] "Error de clasificación en test: 0.0623260990539789"

#Mostramos la matriz de confusión
table(clas.testPre[,ncol(clas.testPre)],clas.test.predic)

##    clas.test.predic
##    0   1   2   3   4   5   6   7   8   9
##  0 176   0   0   0   1   0   0   0   0   1
##  1   0 173   0   0   1   0   0   0   3   5
##  2   0   8 163   5   0   0   0   0   1   0
##  3   1   0   1 173   0   2   0   2   2   2
##  4   0   1   0   0 175   0   0   1   2   2
##  5   0   0   0   0   0 178   1   0   1   2
##  6   1   1   0   0   1   1 174   0   3   0
##  7   0   0   0   0   2   6   0 159   2   10
##  8   0   9   0   0   0   1   0   0 155   9
##  9   1   2   0   1   5   4   0   1   7 159
```

Podemos ver que el error que obtenemos es de aprox. 6% mientras que en el train teníamos aprox. 2%. Esto es normal ya que en el train tenemos los datos con los que hemos entrenado y aquí tenemos datos desconocidos. No obstante, hemos conseguido un 94% de acierto que es un valor bastante bueno de clasificación. Podemos ver también en la matriz de confusión que como ya nos indicaba el train, el mayor problema lo tenemos en las predicciones del número 9 que tiene más confusiones con el resto de etiquetas.

Por lo que finalmente conseguimos un $E_{out} = 0.06177$

10. Discutir y justificar la calidad del modelo encontrado y las razones por las que considera que dicho modelo es un buen ajuste que representa adecuadamente los datos muestrales.

Para esta conclusión nos basamos en los resultados anteriores. Aunque ya he comentado en cada una de ellas las razones, aquí haré un resumen. Este modelo nos ha permitido tener un E_{out} del valor de un 6% de la muestra, que es un valor bastante pequeño. Teniendo en cuenta el preprocesado de datos realizado, hemos conseguido reducir los datos un 53%, por lo que hemos obtenido un 94% de acierto teniendo en cuenta que estamos utilizando menos de la mitad de los datos de entrada para clasificar los distintos números. Esto claramente nos dice que el modelo ha realizado un buen ajuste a los datos, evitando sobre ajuste de los mismos y con reducción y modelado de éstos. Concluimos finalmente que el modelo escogido con todos sus modificaciones, se ajusta adecuadamente a los datos muestrales y permite obtener resultados de alta calidad.

Regresión

1. Comprender el problema a resolver.

En este caso, tenemos un conjunto de datos de la NASA. Nos dan una serie de características sobre aerodinámicas para poder predecir con éstas características una presión final para las alas de un avión. En conjunto de datos está compuesto por 1503 datos de los cuales tenemos 5 características y su respectiva clase. Estas características son la frecuencia, el angulo de ataque, longitud de cuerda, velocidad de flujo y el espesor de desplazamiento. La salida sería el nivel de presión de sonido. Todos los datos son variables enteras o reales.

2. Preprocesado los datos: por ejemplo categorización, normalización, etc.

Como en el apartado anterior, realizaremos el preprocesado de datos para obtener unos datos más representativos e intentar disminuir el número de dimensiones.

- **nzv**: Nos permite eliminar los atributos que tengan varianza 0 o muy cercana a 0. Esto quiere decir que los atributos en los que no haya diferencia para determinar de qué clase podría ser un nuevo elemento basándonos en esa propia columna será eliminada.
- (YeoJohnson): Realiza unos cambios en los datos muy similares al BoxCox (que a su vez, es equivalente a la familia de transformaciones de la potencia), pero permite que existan valores negativos y el cero. Para ello, se basa en la siguiente función.

$$\begin{aligned} & - ((y + 1)^\lambda - 1)/\lambda \text{ si } \lambda \neq 0, y \geq 0 \\ & - \log(y + 1) \text{ si } \lambda = 0, y \geq 0 \\ & - -[(-y + 1)^{2-\lambda} - 1]/(2 - \lambda) \text{ si } \lambda \neq 2, y < 0 \\ & - -\log(-y + 1) \text{ si } \lambda = 2, y < 0 \end{aligned}$$

- **center**: Resta la media de los datos a los propios datos, para tener la información un poco más concentrada pero sin perder el propio valor del dato.
- **scale**: Divide los datos por la desviación estándar. Consigue centrar más aún los datos con el objetivo de tener una visión de los mismos más acorde con la realidad.
- **pca**: Nos permite reducir el espacio de dimensiones, ésto se lo hace cojiendo las variables más importantes mientras consigan explicar un 95%

En primer lugar, cargaremos los datos del train para posteriormente realizarle el preprocesamiento.

```
#Cargamos los datos
reg.datos <- read.csv("airfoil_self_noise.csv", header = FALSE, sep = ",")
names(reg.datos) <- c("frecuencia", "angulo de ataque", "longitud de cuerda", "velocidad de flujo", "espacio de desplazamiento")

#Creamos los cambios que se van a realizar.
reg.cambios <- preProcess(reg.datos[, -ncol(reg.datos)], method = c("nzv", "YeoJohnson", "center", "scale", "pca"))
print(reg.cambios)

## Created from 1503 samples and 5 variables
##
## Pre-processing:
##   - centered (5)
##   - ignored (0)
##   - principal component signal extraction (5)
##   - scaled (5)
##   - Yeo-Johnson transformation (3)
##
## Lambda estimates for Yeo-Johnson transformation:
## 0.01, 0.34, -0.12
```

```
## PCA needed 4 components to capture 95 percent of the variance
```

```
# Aplicamos los cambios (Combinamos con cbind los cambios de los datos con las etiquetas de los mismos)
reg.datosPre <- as.data.frame(cbind(predict(reg.cambios,as.matrix(reg.datos[, -ncol(reg.datos)]))), reg.datos[, -ncol(reg.datos)])
```

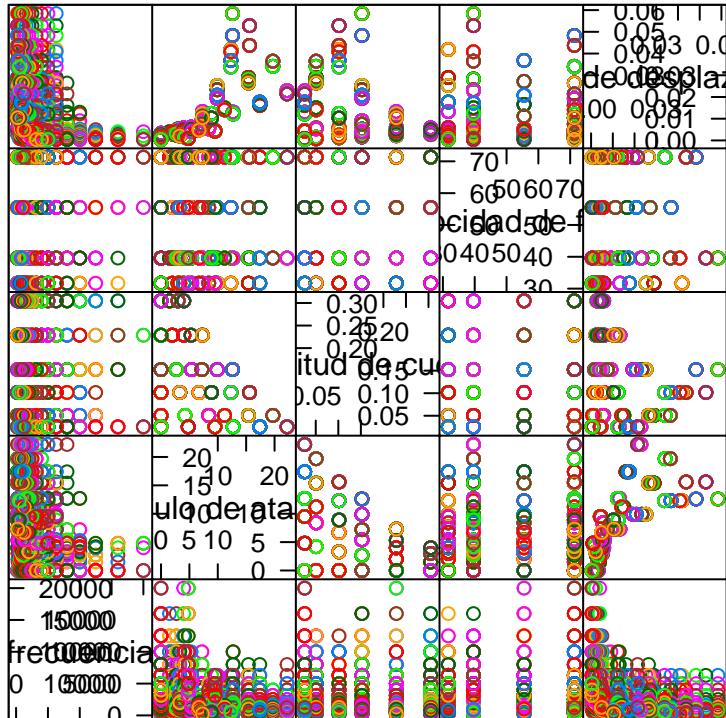
Para comentar un poco los datos y las transformaciones, vamos a realizar unos gráficos de los datos antes de ser transformados y después. No obstante, no vamos a mostrar todas las variables ya que tienen una dimensión muy elevada y sería imposible dibujarlas en un mismo gráfico.

1. Diagrama de dispersión.

```
#Datos sin preprocessado.
```

```
featurePlot(x=reg.datos[, -ncol(reg.datos)], y=as.factor(reg.datos[, ncol(reg.datos)]), plot="pairs", main =
```

Diagrama de dispersión (DATOS)

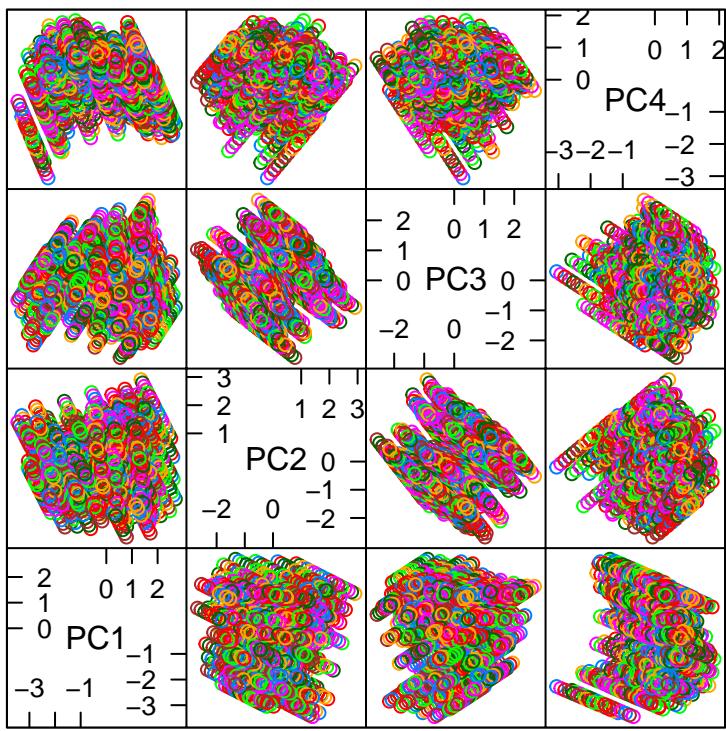


Scatter Plot Matrix

```
#Datos con preprocessado
```

```
featurePlot(x=reg.datosPre[, -ncol(reg.datosPre)], y=as.factor(reg.datosPre[, ncol(reg.datosPre)]), plot="pairs", main =
```

Diagrama de dispersión (DATOS-PREPROC)

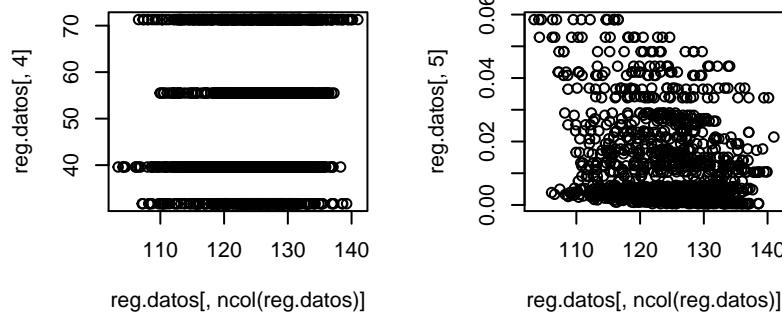
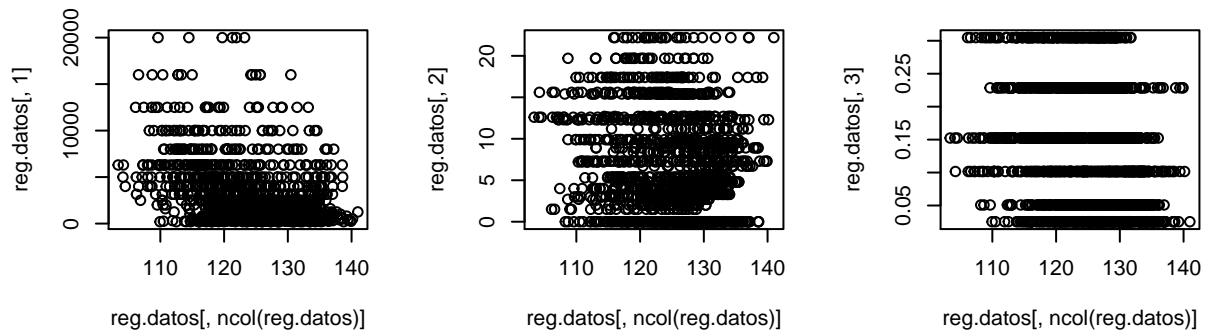


variable dependiente.

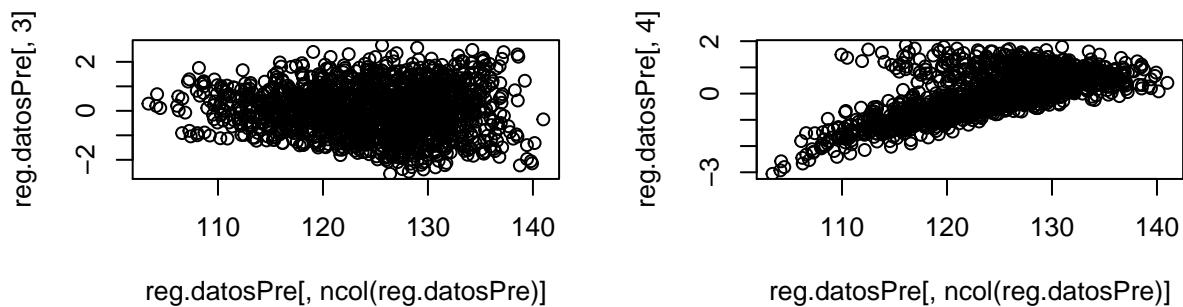
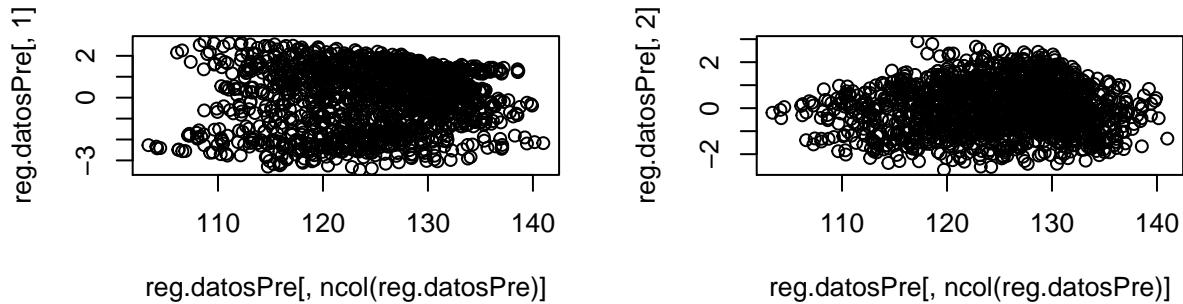
```
par(mfrow=c(2,3))
plot(reg.datos[,ncol(reg.datos)],reg.datos[,1])
plot(reg.datos[,ncol(reg.datos)],reg.datos[,2])
plot(reg.datos[,ncol(reg.datos)],reg.datos[,3])
plot(reg.datos[,ncol(reg.datos)],reg.datos[,4])
plot(reg.datos[,ncol(reg.datos)],reg.datos[,5])

par(mfrow=c(2,2))
```

2. Relacion de las variables con la vari-



```
plot(reg.datosPre[, ncol(reg.datosPre)], reg.datosPre[, 1])
plot(reg.datosPre[, ncol(reg.datosPre)], reg.datosPre[, 2])
plot(reg.datosPre[, ncol(reg.datosPre)], reg.datosPre[, 3])
plot(reg.datosPre[, ncol(reg.datosPre)], reg.datosPre[, 4])
```



Para este caso, vemos como es capaz de concentrar los datos y sacar una relación más lineal entre los propios

datos con la variable dependiente. Como podemos ver en los plots, vemos como las transformaciones de los mismos han conseguido centrar los datos y tenerlos más concetrados y con más relación entre ellos y no tan dispersos como al principio.

3. Selección de clases de funciones a usar.

Como en el apartado anterior, utilizaremos las clases de funciones lineales para predecir. Con estas funciones intentaremos hacer una predicción lo mas correcta posible. Probaremos con un modelo de regresion lineal simple y otro de regresion lineal múltiple ya que viendo los datos intentaré que la variable dependiente dependa sólo de la última variable que vemos en el gráfico anterior.

4. Definición de los conjuntos de training, validación y test usados en su caso.

En este caso solo tenemos un conjunto de datos, por lo que vamos a tener que divirlo en train y en test para poder tener unos datos en el futuro que “no conocemos”. Por lo que vamos a dividir los datos utilizando la función `createDataPartition` de la librería caret. Utilizaré un 80% para el training y un 20% para el test.

```
#Obtenemos los indices de la particion del 80% para el training
trainIndex <- createDataPartition(reg.datosPre[,ncol(reg.datosPre)], p = .8, list = FALSE, times = 1)

#Creamos el train y el test con los indices anteriores
reg.train <- reg.datosPre[ trainIndex,]
reg.test  <- reg.datosPre[-trainIndex,]
```

5. Discutir la necesidad de regularización y en su caso la función usada para ello.

Como he indicado anteriormente, la regularización nos permite no sobreajustar el aprendizaje y así obtener un mejor predictor generalmente para las muestras futuras y desconocidas. En este caso, utilizaré regularización L1 y L2 que están explicadas en los apartados anteriores por lo que no las pondré de nuevo. Aparte de estos dos modelos, probaré una regresion lineal básica con `lm`. Asíque tendremos 3 modelos, el básico con `lm` y los dos con regularización con `glmnet` al igual que en el apartado anterior, con la diferencia de que aquí tenemos regresion y no una clasificación multinomial.

6. Definir los modelos a usar y estimar sus parámetros e hyperparámetros.

Como he indicado anteriormente, utilizaré dos modelos distintos respecto a la regularización, uno que utiliza L1 regularization y otro que utiliza L2 regularization. Para ello, utilizaré la función (`cv.glmnet`) donde dependiendo del parámetro α que le pasemos realizará una regularización u otra, siendo $\alpha = 1$ L1 y $\alpha = 0$ L2. Por otra parte utilizaré `lm` para realizar una regresión simple.

```
#Entrenamos con cross-validation los modelos con regularización
reg.mod.L1 <- cv.glmnet(x = as.matrix(reg.train[,-ncol(reg.train)]),y = reg.train[,ncol(reg.train)],nfold=10)
summary(reg.mod.L1)

##          Length Class  Mode
## lambda      61   -none- numeric
## cvm         61   -none- numeric
## cvsd        61   -none- numeric
## cvup        61   -none- numeric
## cvlo        61   -none- numeric
## nzero       61   -none- numeric
## name         1    -none- character
## glmnet.fit  12   elnet  list
## lambda.min   1    -none- numeric
## lambda.1se   1    -none- numeric

reg.mod.L2 <- cv.glmnet(x = as.matrix(reg.train[,-ncol(reg.train)]),y = reg.train[,ncol(reg.train)],nfold=10)
summary(reg.mod.L2)

##          Length Class  Mode
```

```

## lambda    99    -none- numeric
## cvm       99    -none- numeric
## cvsd      99    -none- numeric
## cvup      99    -none- numeric
## cvlo      99    -none- numeric
## nzero     99    -none- numeric
## name       1    -none- character
## glmnet.fit 12    elnet  list
## lambda.min  1    -none- numeric
## lambda.1se   1    -none- numeric

#Entrenamos el modelo simple.
ctrl <- trainControl(method="cv", number=10)
reg.mod.lm <- train(x=as.matrix(reg.train[,-ncol(reg.train)]), y=reg.train[,ncol(reg.train)], method =
summary(reg.mod.lm)

##
## Call:
## lm(formula = .outcome ~ ., data = dat)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -23.0541  -2.9216   0.4354   3.4799  14.9879
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 124.8152    0.1471 848.232 <2e-16 ***
## PC1         0.2436    0.1018   2.393  0.0169 *
## PC2         0.2656    0.1419   1.871  0.0616 .
## PC3        -0.3811    0.1529  -2.493  0.0128 *
## PC4         5.7179    0.1791  31.922 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.102 on 1198 degrees of freedom
## Multiple R-squared:  0.4633, Adjusted R-squared:  0.4615
## F-statistic: 258.6 on 4 and 1198 DF,  p-value: < 2.2e-16

```

7. Selección y ajuste modelo final.

Ya tenemos los dos modelos creados, ahora vamos a calcular el $E_{in} = \frac{1}{N} \sum_{i=1}^N (\hat{y}_i - y_i)^2$ con los dos modelos distintos que hemos creado para finalmente seleccionar unos de ellos y terminar de entrenarlo.

```

#Funcion de error
reg_Err <- function(datos,predicciones){
  suma <- 0.0
  for(i in 1:length(predicciones)){
    suma <- suma+((predicciones[i]-datos[i,ncol(datos)])^2)
  }
  suma/length(predicciones)
}

#Predecimos las etiquetas del train utilizando el modelo creado anteriormente
reg.L1.pred <- predict.cv.glmnet(reg.mod.L1,as.matrix(reg.train[,-ncol(reg.train)]))
reg.L1.err <- reg_Err(reg.train,reg.L1.pred)
print(paste("Error de regresión con regularización L1 en train: ",reg.L1.err))

```

```

## [1] "Error de regresión con regularización L1 en train: 27.6178610031598"
#Predecimos las etiquetas del train utilizando el modelo creado anteriormente
reg.L2.pred <- predict.cv.glmnet(reg.mod.L2,as.matrix(reg.train[, -ncol(reg.train)]))
reg.L2.err <- reg_err(reg.train,reg.L2.pred)
print(paste("Error de regresión con regularización L2 en train: ",reg.L2.err))

## [1] "Error de regresión con regularización L2 en train: 26.9554569590647"
#Predecimos las etiquetas del train utilizando el modelo creado anteriormente
reg.lm.pred <- predict(reg.mod.lm,as.matrix(reg.train[, -ncol(reg.train)]))
reg.lm.err <- reg_err(reg.train,reg.lm.pred)
print(paste("Error de regresión en train: ",reg.lm.err))

## [1] "Error de regresión en train: 25.9259194062971"

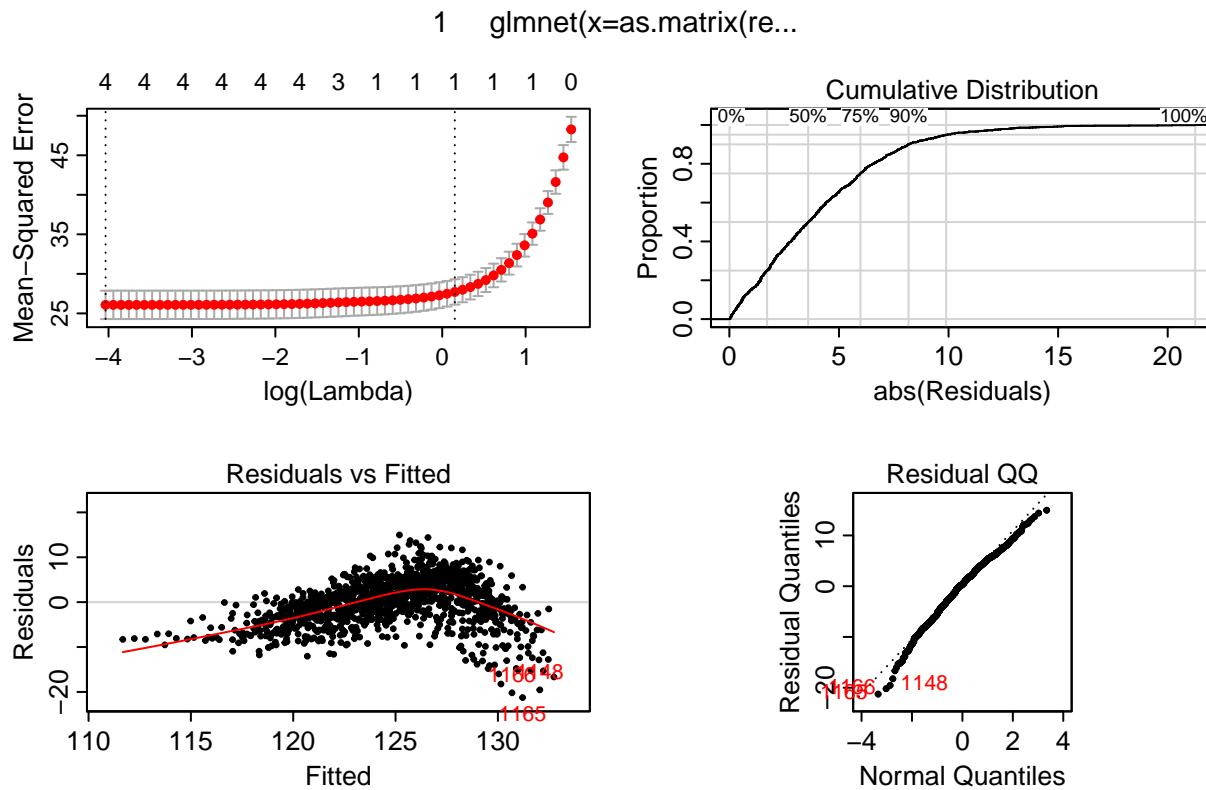
```

Podemos observar que obtenemos un error más bajo en la regresión lineal simple (lm) que con regularización, esto tiene sentido ya que con regularización se aumenta un poco el error para evitar el sobreajuste a los datos de entrenamiento. Para hacer una comparación un poco más exaustiva voy a dibujar la dispersión de los residuos. Ésta debe estar distribuida respecto al cero para todo X ya que existe una relación lineal entre ellas.

```

#Residuos L1
plotres(reg.mod.L1)

```

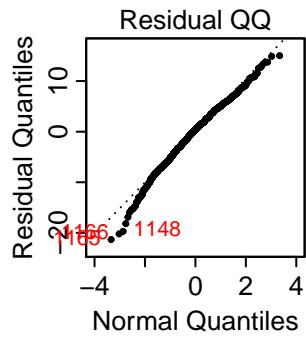
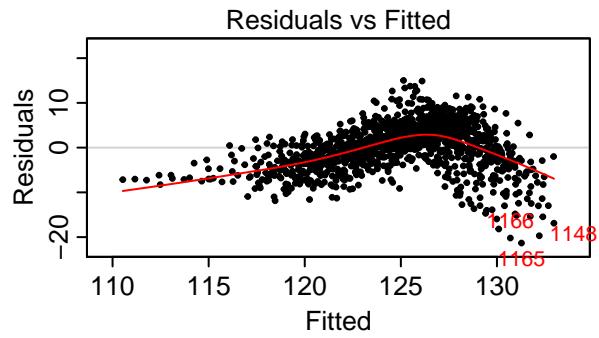
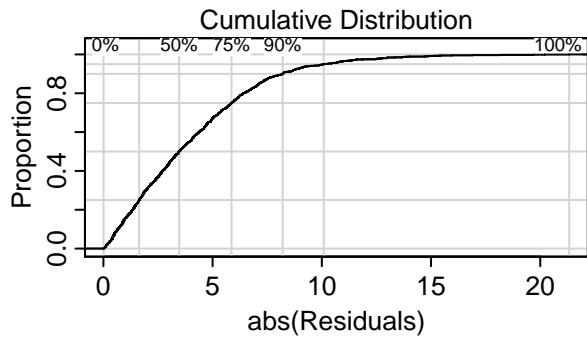
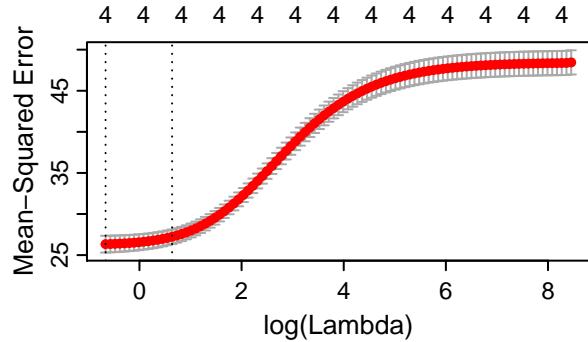


```

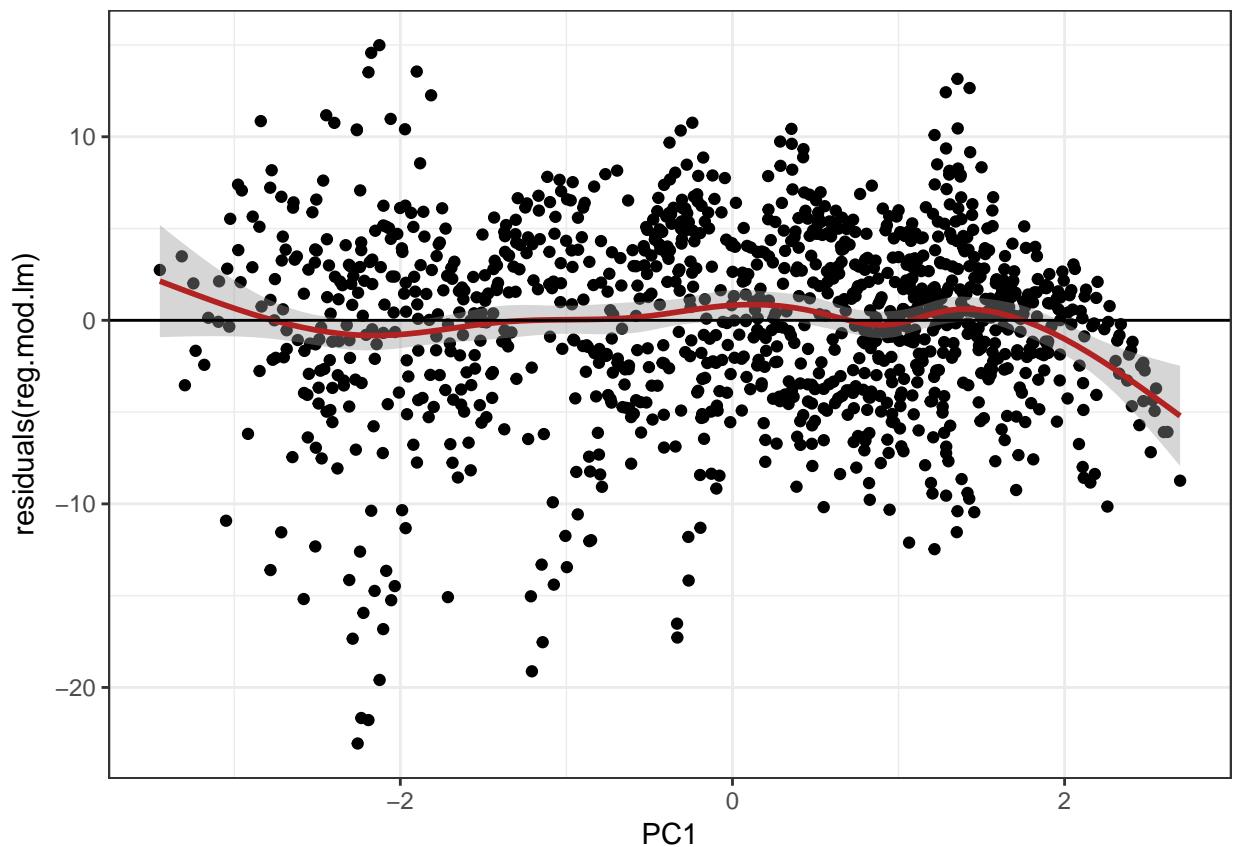
#Residuos L1
plotres(reg.mod.L2)

```

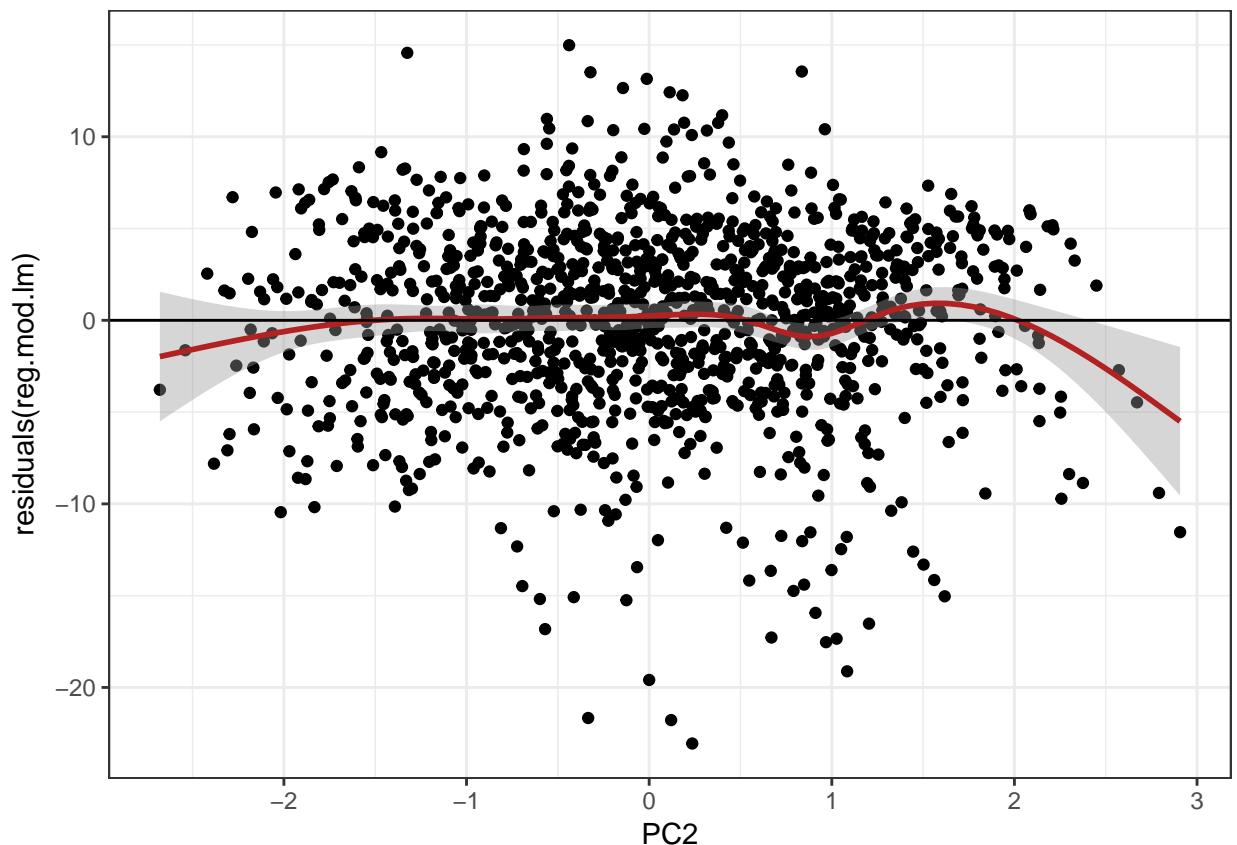
```
1     glmnet(x=as.matrix(re...
```



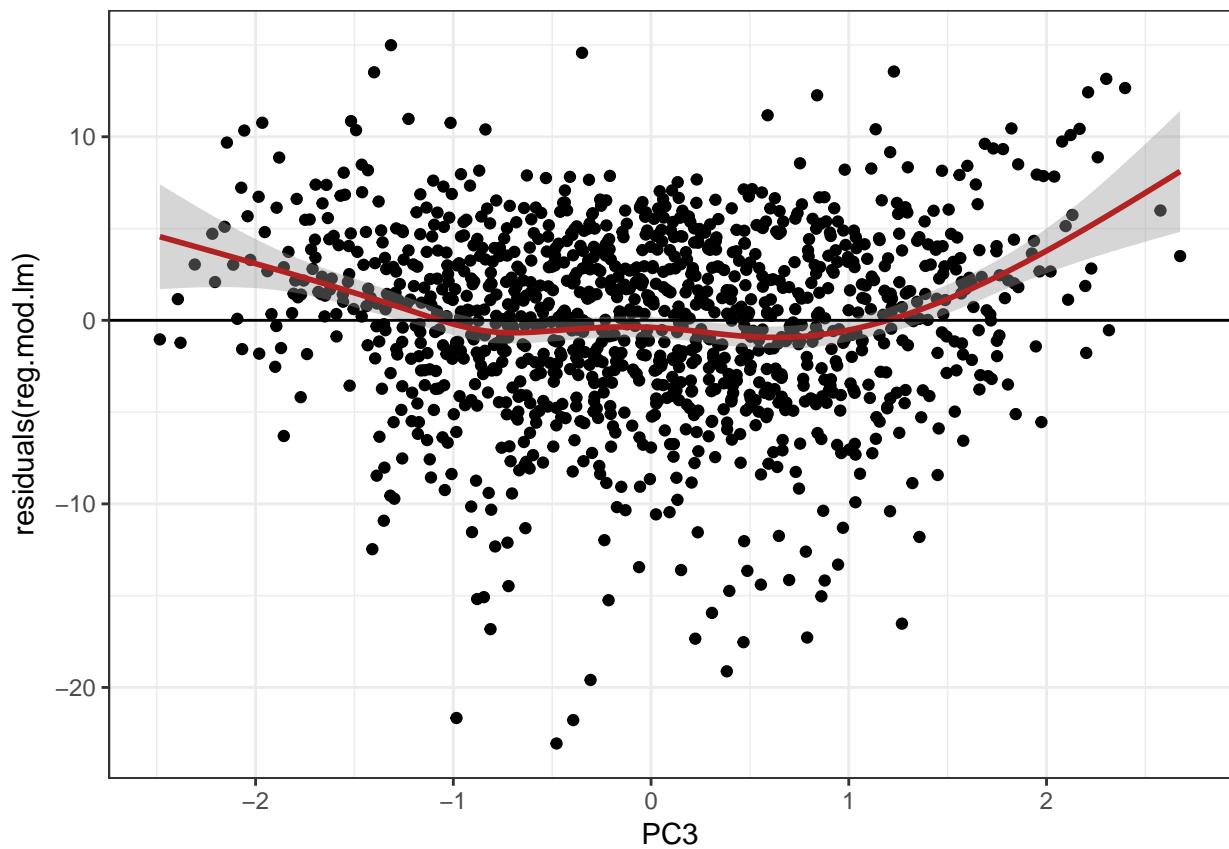
```
#Residuos lm
ggplot(data = reg.train, aes(PC1, residuals(reg.mod.lm))) + geom_point() + geom_smooth(color = "firebrick")
## `geom_smooth()` using method = 'gam'
```



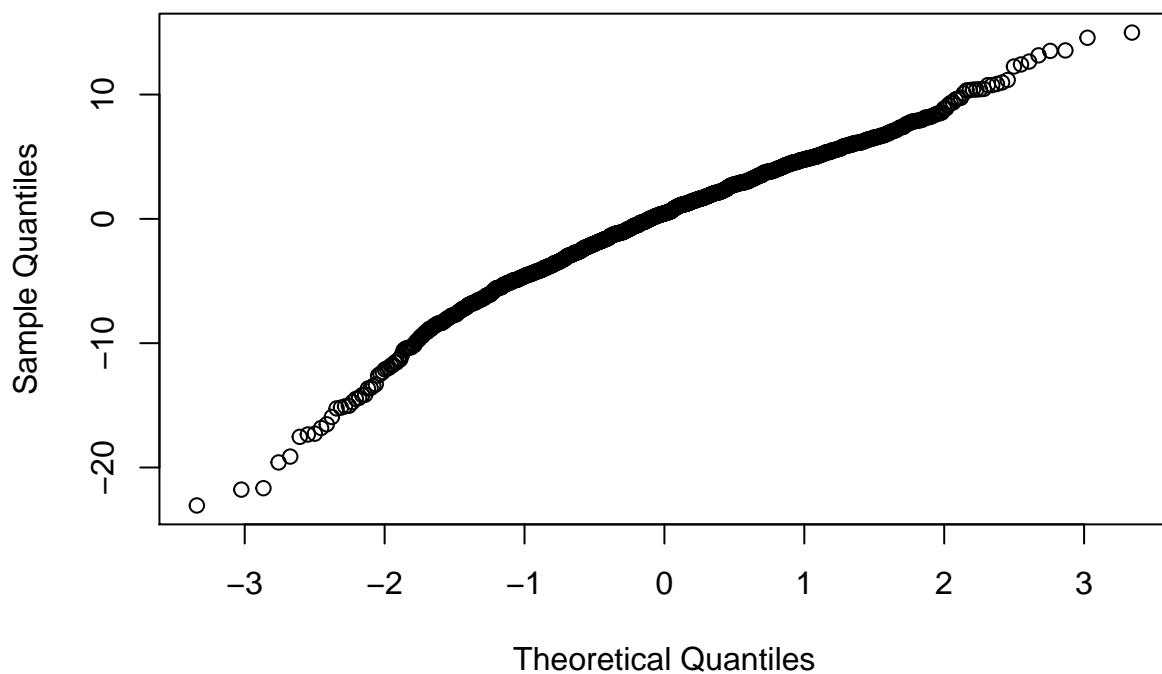
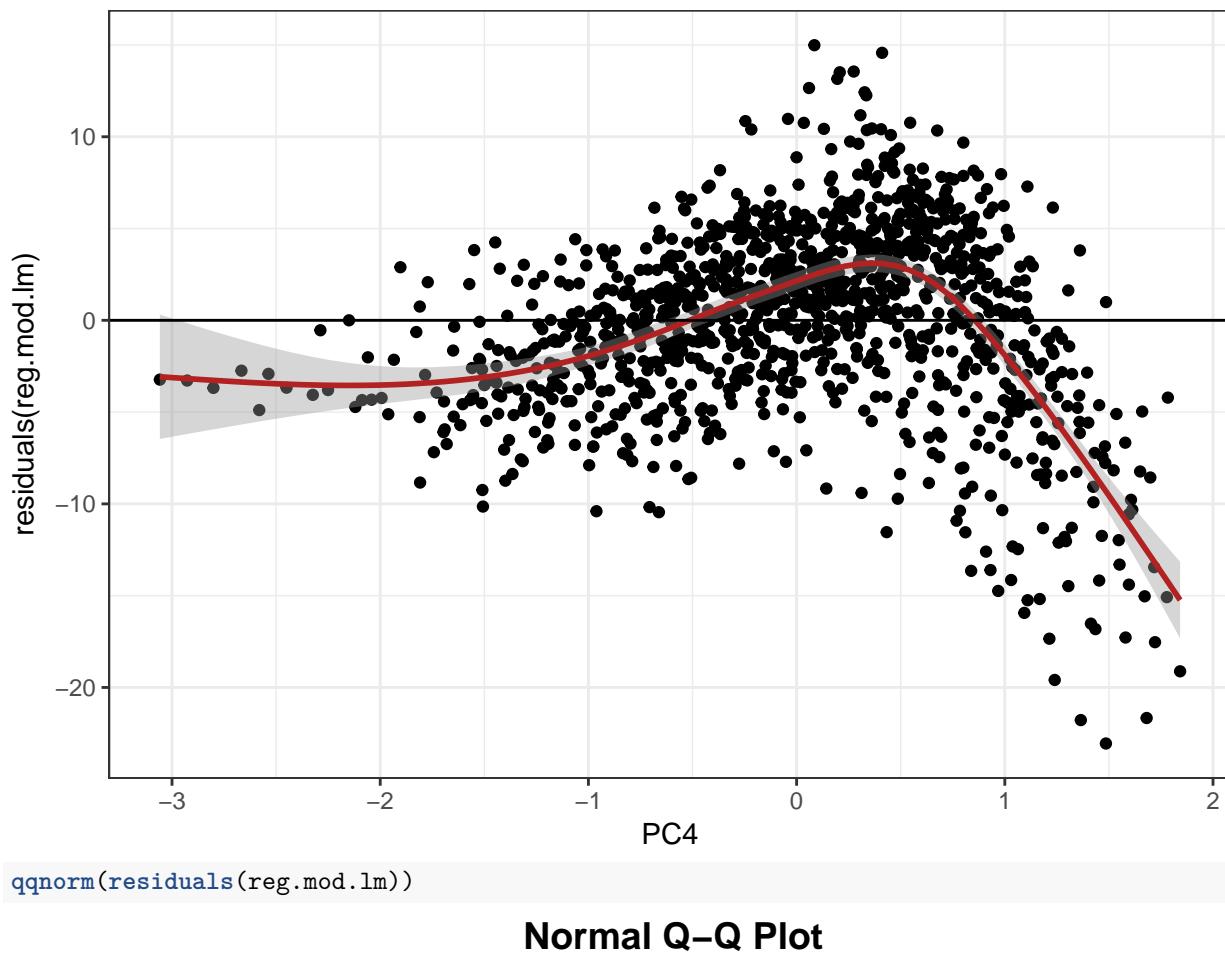
```
ggplot(data = reg.train, aes(PC2, residuals(reg.mod.lm))) + geom_point() + geom_smooth(color = "firebrick")  
## `geom_smooth()` using method = 'gam'
```



```
ggplot(data = reg.train, aes(PC3, residuals(reg.mod.lm))) + geom_point() + geom_smooth(color = "firebrick")  
## `geom_smooth()` using method = 'gam'
```



```
ggplot(data = reg.train, aes(PC4, residuals(reg.mod.lm))) + geom_point() + geom_smooth(color = "firebrick")  
## `geom_smooth()` using method = 'gam'
```



Podemos ver que todas siguen una distribución correcta, por lo que finalmente para seleccionar el modelo

final vamos a calcular R^2 para ver cuánto son capaces de explicar cada uno de los modelos.

```
#R ~2 L1
reg.mod.L1$glmnet.fit$dev.ratio[which(reg.mod.L1$glmnet.fit$lambda == reg.mod.L1$lambda.min)]
## [1] 0.4633031

#R ~2 L2
reg.mod.L2$glmnet.fit$dev.ratio[which(reg.mod.L2$glmnet.fit$lambda == reg.mod.L2$lambda.min)]
## [1] 0.4611161

#R ~2 lm
summary(reg.mod.lm)

##
## Call:
## lm(formula = .outcome ~ ., data = dat)
##
## Residuals:
##       Min     1Q   Median     3Q    Max 
## -23.0541 -2.9216  0.4354  3.4799 14.9879 
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 124.8152    0.1471  848.232 <2e-16 ***
## PC1         0.2436    0.1018   2.393   0.0169 *  
## PC2         0.2656    0.1419   1.871   0.0616 .  
## PC3        -0.3811    0.1529  -2.493   0.0128 *  
## PC4         5.7179    0.1791  31.922 <2e-16 *** 
## ---      
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.102 on 1198 degrees of freedom
## Multiple R-squared:  0.4633, Adjusted R-squared:  0.4615 
## F-statistic: 258.6 on 4 and 1198 DF, p-value: < 2.2e-16
```

Finalmente he decidido utilizar glmnet con L2 para este caso, mis conclusiones son porque aunque lm tiene menor error que éste, puede ser debido a la regularización y las dos son capaces de explicar lo mismo. Así que como L2 tiene menos error que L1 y respecto a lm son prácticamente iguales y éste nos ofrece regularización, elijo L2 con glmnet finalmente.

```
#Entrenamos con todos los datos (NO cross-validation)
reg.mod <- glmnet(x = as.matrix(reg.train[, -ncol(reg.train)]), y = c(reg.train[, ncol(reg.train)]), family = "gaussian")
```

```
##          Length Class      Mode
## a0            100  -none-   numeric
## beta          400   dgCMatrix S4
## df             100  -none-   numeric
## dim            2   -none-   numeric
## lambda         100  -none-   numeric
## dev.ratio     100  -none-   numeric
## nulldev         1  -none-   numeric
## npasses         1  -none-   numeric
## jerr            1  -none-   numeric
## offset           1  -none- logical
```

```

## call      5    -none-    call
## nobs     1    -none-    numeric
#Guardamos el lambda del mejor modelo anterior para utilizarlo en las predicciones futuras.
reg.lambda <- as.double(reg.mod.L2[9])
reg.lambda

```

```
## [1] 0.5152809
```

8. Discutir la idoneidad de la métrica usada en el ajuste

Para ello, voy a utilizar el error mínimo cuadrático que lo tenemos calculado en el apartado anterior, también utilizaré la media de diferencias entre la etiquetas reales y la predichas. Con estos métodos podemos comprobar como el modelo ajusta los datos y guiarnos sobre el ajuste.

```

reg_Err_dif <- function(datos,predicciones){
  suma <- 0.0
  for(i in 1:length(predicciones)){
    suma <- suma+abs(predicciones[i]-datos[i,ncol(datos)]))
  }
  suma/length(predicciones)
}

reg.mod.predic <- predict(reg.mod,as.matrix(reg.train[,-ncol(reg.train)]),s=reg.lambda)

#Cuadrático
reg_Err(reg.train,reg.mod.predic)

## [1] 26.03281

#Diferencia
reg_Err_dif(reg.train,reg.mod.predic)

## [1] 3.968106

```

Podemos ver que el error cuadrático es un buen valor comparandolo con los modelos que hemos utilizado antes. Podemos comprobar por la diferencia entre las etiquetas media, que tenemos un valor aprox. de 4, este valor es una buen valor que teniendo en cuenta que tenemos 1203 observaciones, que en media tenga 4 valores de diferencia, lo que nos dice que este modelo ha sido capaz de mostrarnos cómo se ha ajustado a los datos.

9. Estimacion del error Eout del modelo lo más ajustada posible.

Para ello, utilizaré los test que particionamos en apartados anteriores, con estos, calcuraré el error mínimo cuadrático y la diferencia de error media. En estos resultados nos basaremos para estimar la calidad del modelo respescto a estos datos.

```

reg.mod.predic <- predict(reg.mod,as.matrix(reg.test[,-ncol(reg.test)]),s=reg.lambda)

#Cuadrático
reg_Err(reg.test,reg.mod.predic)

## [1] 22.86534

#Diferencia
reg_Err_dif(reg.test,reg.mod.predic)

## [1] 3.819938

```

Viendo los resultados, hemos obtenido unos errores menores que en el propio train, esto puede ser debido al particionado de los datos, no obstante, nos demuestra que el ajuste que he realizado es bastante bueno

ya que la diferencia media de errores es bastante pequeña y puedo comprobar la información de los datos correctamente.

10. Discutir y justificar la calidad del modelo encontrado y las razones por las que considera que dicho modelo es un buen ajuste que representa adecuadamente los datos muestrales.

Para este razonamiento simplemente me baso en los resultados, he obtenido un modelo que veímos que obtenía un error mínimo cuadrático de 25, un valor bajo para el número de muestras que tenemos, he comprobado que la distribución de los residuos es correcta y que obtenemos un error medio de 3.8, que es una diferencia bastante baja. Por lo que finalmente puedo concluir que este modelo ha sido capaz de conseguir un buen resultado y teniendo en cuenta que he conseguido reducir el espacio de dimensiones en un 25%.