Practica3

Adrián Sánchez Cerrillo, Miguel Ángel López Robles

14 de mayo de 2018

Problema a resolver

En este apartado vamos a resolver un problema de regresión, centrándonos en el ajuste de un modelo lineal con la finalidad de obtener el mejor predictor posible para dicho problema. En este caso contamos con la base de datos *Airfoil Self-Noise*, la cual contiene un conjunto de datos proporcionado por la NASA sobre distintos test acústicos y aerodinámicos de alerones en túneles de viento. Podemos encontrar los datos que usaremos y su descripción en la página https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise

Disponemos de séis atributos, siendo el sexto la salida proporcionada, y nuestro objetivo será predecir el nivel de presión del sonido, medido en decibelios a partir del resto de características, como podrían ser la frecuencia, ángulo, longitud, velocidad y desplazamiento.

Carga de ficheros CSV

Primero vamos a cargar las librerias necesarias para la ejecución de nuestra práctica y, posteriormente, se cargará en memoria el fichero correspondiente a la base de datos a analizar, mediante la orden *read.csv( )*.

```
## Loading required package: lattice
```

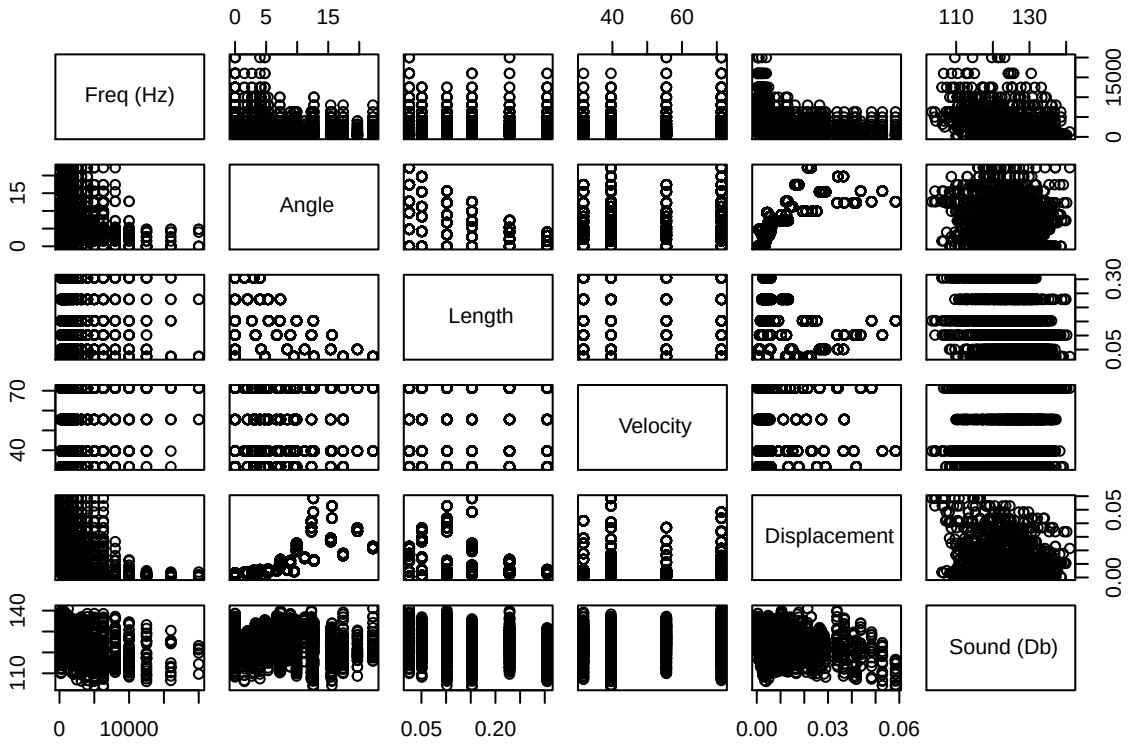
```
## Loading required package: ggplot2
```

```
## Loading required package: Matrix
```

```
## Loading required package: foreach
```

```
## Loaded glmnet 2.0-16
```

```
air <- read.csv("datos/airfoil_self_noise.csv", header = FALSE, sep = ",")
names(air) <- c("Freq (Hz)", " Angle", " Length", " Velocity", " Displacement", " Sound (Db)")
plot(air)
```



Como podemos observar, se han cargado correctamente los datos. Ahora procederemos a preparar y preprocesar los datos.

Preprocesado de los datos

Para realizar la validación y entrenamiento de nuestro modelo, puesto que solo disponemos de un archivo con datos para hacer test, se realizará de la siguiente manera:

*** Train :** 70% Correspondiente al conjunto de datos servirá para realizar entrenamiento de nuestro modelo, particionando dicho subconjunto y validando el entrenamiento mediante la técnica de validación cruzada.

*** Test :** 30% Restante para validar nuestro modelo ya entrenado, obteniendo así un conjunto de datos para realizar test.

Separamos las etiquetas del conjunto de datos:

```
air.Data <- air[,1:(ncol(air)-1)]
air.Etiquetas <- air[,ncol(air)]
```

Primeramente vamos a preprocesar nuestros datos, vamos a comprobar si existe algúna columna de atributos cuya varianza sea cercana a cero, dichas variables serán irrelevantes para el estudio del problema.

```
zero <- nearZeroVar(air.Data)
zero
```

```
## integer(0)
```

Como podemos observar, no existe ningún atributo cuya varianza sea cero.

Ahora vamos a dar un paso más en el estudio de la varianza y la muestra, usaremos la función **preProcess()**, aplicando diferentes métodos para ajustar los valores de los atributos, como puede ser la normalización mediante la transformación de *Yeo-Johnson*, restando la media y dividiendo por la desviación estándar con *center* y *scale*.
Para establecer una comparación, usaremos el preprocesamiento con y sin la aplicación de *PCA*, "Principal

Component Analysis", el cual reduce la dimensión de los datos eliminando, si procede, aquellos atributos que no sean relevantes para el desarrollo del modelo de aprendizaje.

```
ProcesamientoPCA <- preProcess(air.Data, method = c("YeoJohnson","center", "scale", "pca"),thres=0.95)
ProcesamientoPCA
```

```
## 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
```

El preprocesamiento con PCA nos ha reducido el modelo a 4 variables, por lo tanto estaría descartando uno de los atributos conjunto de datos.
Posteriormente analizaremos los resultados, pero por el momento, y teniendo en cuenta que disponemos de únicamente 5 atributos como máximo, no es completamente necesario reducir el número de atributos que disponemos. Los componentes en el *"Principal Component Analysis"* se crean como como combinación lineal de varias variables y es generalmente usado cuando el número de atributos es suficientemente grande, así como el número de datos. Puesto que únicamente disponemos 5 atributos en nuestro modelo para predecir la salida, optaremos por no aplicar PCA en el preprocesado, y por tanto no reduciremos el número de atributos a utilizar.

```
ProcesamientoSinPCA <- preProcess(air.Data, method = c("YeoJohnson","center", "scale"), thres = 0.95)
ProcesamientoSinPCA
```

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

Definir conjuntos

Obtenemos el nuevo conjunto de datos tras haber aplicado el preprocesamiento anterior con la función **predict()**. Actualizamos el fichero de datos:

```
air.Data  <- predict(ProcesamientoSinPCA, air.Data)

air <- cbind(air.Data, air.Etiquetas)
```

Por último, creamos particiones de los datos manteniendo la idea principal del 70/30%, para cada uno de los conjuntos de datos obtenidos anteriormente. Extraemos los índices mediante el uso de **sample()** y obtenemos el conjunto de train y test.

```r
set.seed(7)

index  <- sample (nrow(air.Data),  round(nrow(air.Data) * 0.7))

air.Tr  <- air.Data[index,]
air.Tst <- air.Data[-index,]

Etiquetas.Tr  <- air.Etiquetas[index]
Etiquetas.Tst <- air.Etiquetas[-index]
```

Entrenamiento del modelo

Para realizar el entrenamiento de nuestro modelo de regresión seguiremos el procedimiento SRM o *"Structural Risk Minimization"*, empezando por la clase de funciones lineal menos compleja, analizando los resultados obtenidos e, incrementando la clase de funciones, si fuera necesario.

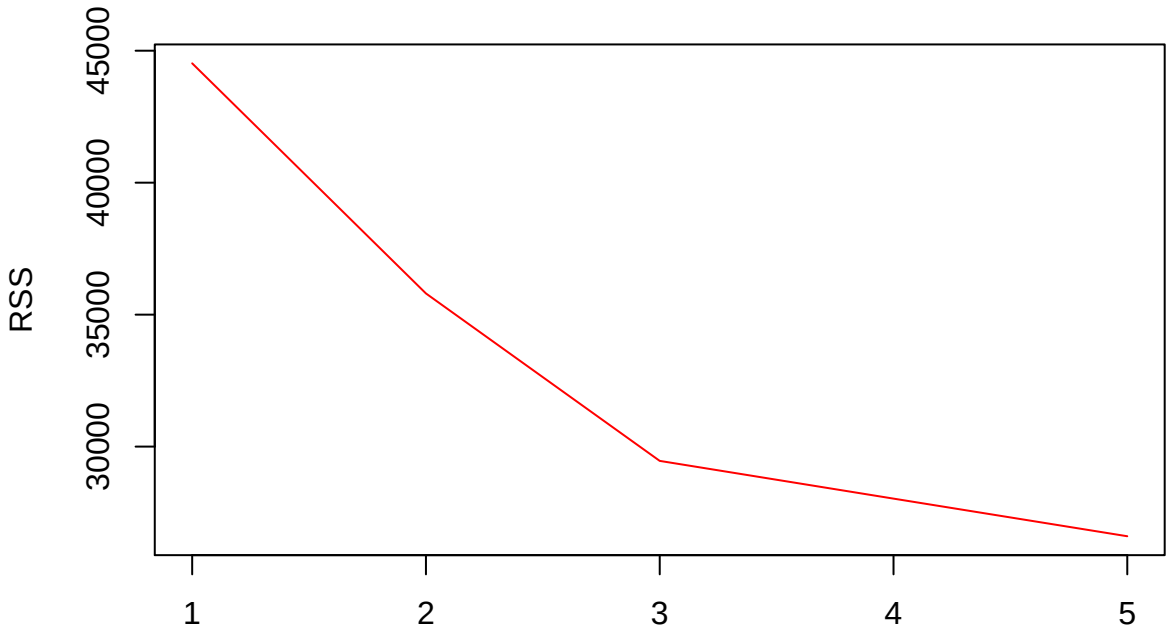
Uno de los pasos más importantes a la hora de abordar un problema de Aprendizaje Automático es el de escoger un subconjunto de los datos que resulte representativo de las etiquetas, por tanto dicho subconjunto debe ser un buen predictor de los datos a predecir.
Tal y como se ha mencionado anteriormente, el número de atributos que disponemos es reducido, aunque se analizarán los resultados obtenidos por **regsubset()** para comprobar si es buena opción mantener todos los atributos.

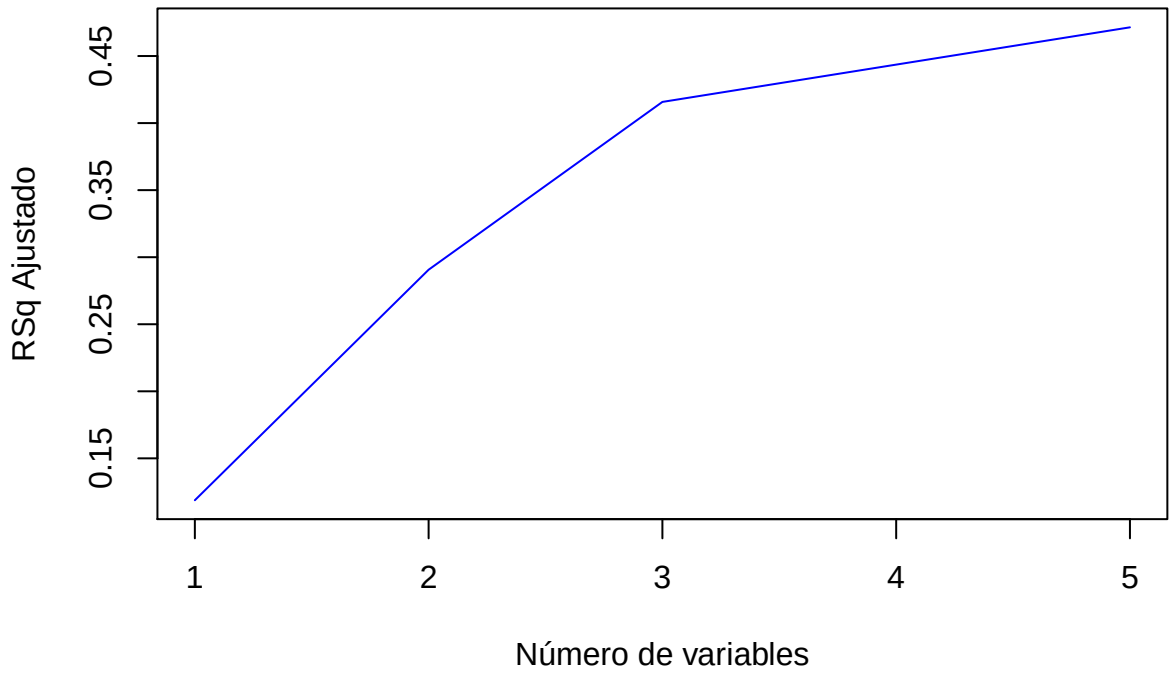
```r
reg.subset  <- regsubsets(x = air.Tr, y = Etiquetas.Tr, nvmax = 5, method = "forward")
reg.summary <- summary(reg.subset)
```

Mediante el uso de la función anterior Se puede observar que se incluyen las cinco variables en el modelo. Ahora vamos a analizar más profundamente los resultados devueltos por el resumen del subconjunto, para ello, analizaremos las métricas devueltas R^2, C_P y BIC.

```r
reg.summary$rsq
```

```
## [1] 0.1196111 0.2920068 0.4174578 0.4457204 0.4739121
```





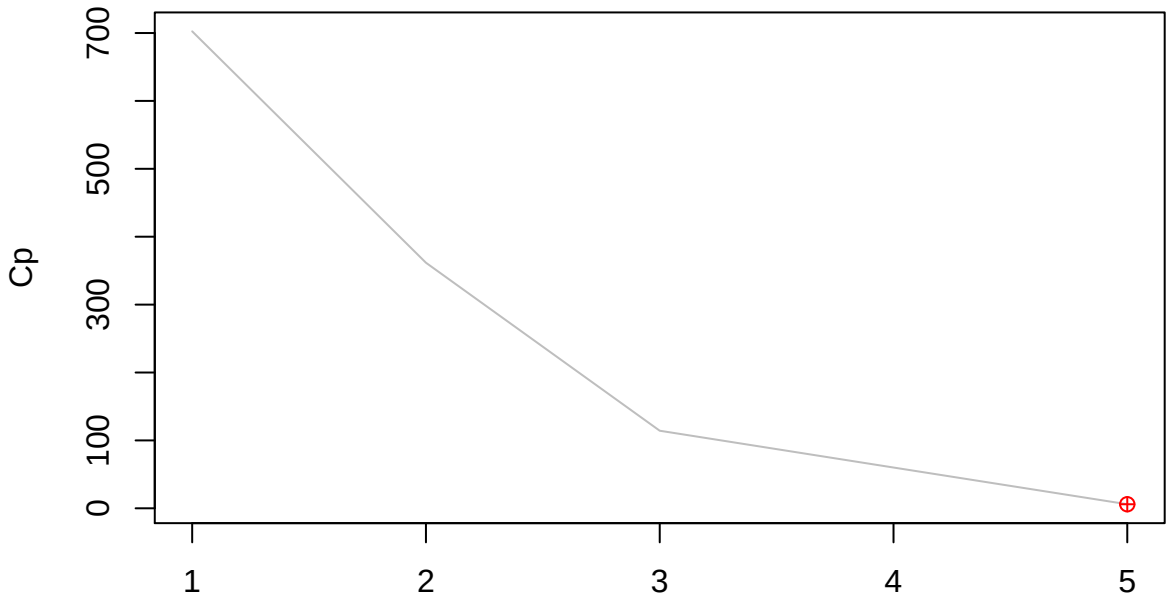
Podemos concluir, analizando la métrica ajustada *RSq*, que dicha métrica va aumentando conforme se utiliza un mayor número de atributos de partida como predictores de nuestro modelo, por lo tanto es capaz de explicar mejor la varianza existente en la muestra cuando usamos un número mayor de variables.
De la misma manera, el error residual para la métrica *RSS* decrece con el uso de todas las variables disponibles.

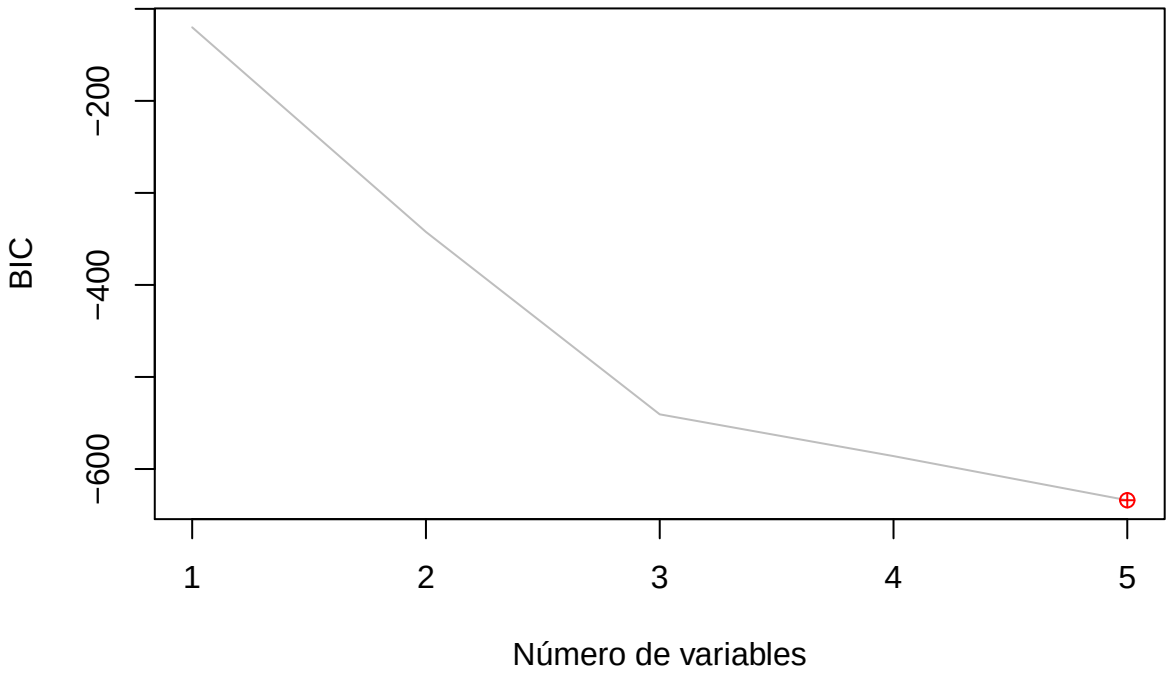
```
reg.summary$cp
```

```
## [1] 702.44293 361.67539 114.24589  60.05264   6.00000
```

```
reg.summary$bic
```

```
## [1] -120.0990 -342.4021 -540.6183 -585.9782 -633.9353
```





Tal y como habíamos expuesto anteriormente, tanto la métrica *BIC* como *Cp* confirman lo mencionado con anterioridad, por lo tanto, podemos continuar con el entrenamiento de nuestro modelo usando para ello todos los atributos del mismo.
Por último antes de empezar a definir los modelos, vamos a definir la métrica R^2 como una función para facilitar los cálculos en modelos posteriores.

```
rsquare = function(yhat, y){
  e = y - yhat
  SSR = sum((e)^2)
  SST = sum ( (y - mean(y))^2)
  1-(SSR/SST)
```

```
}
```

Simple Lineal-Regression

Establecemos el método de control para validar el modelo con la función **trainControl()**, indicando la aplicación de validación cruzada con cinco particiones como método de validación.

```
control = trainControl(method = "cv", number = 5)
```

Ahora usamos la función **train()** para entrenar nuestro modelo mediante un método lineal simple, usando el método de control anterior.

```
Lin <- train(x = air.Tr, y = Etiquetas.Tr, method = "lm", trControl = control, metric = "Rsquared")
Lin
```

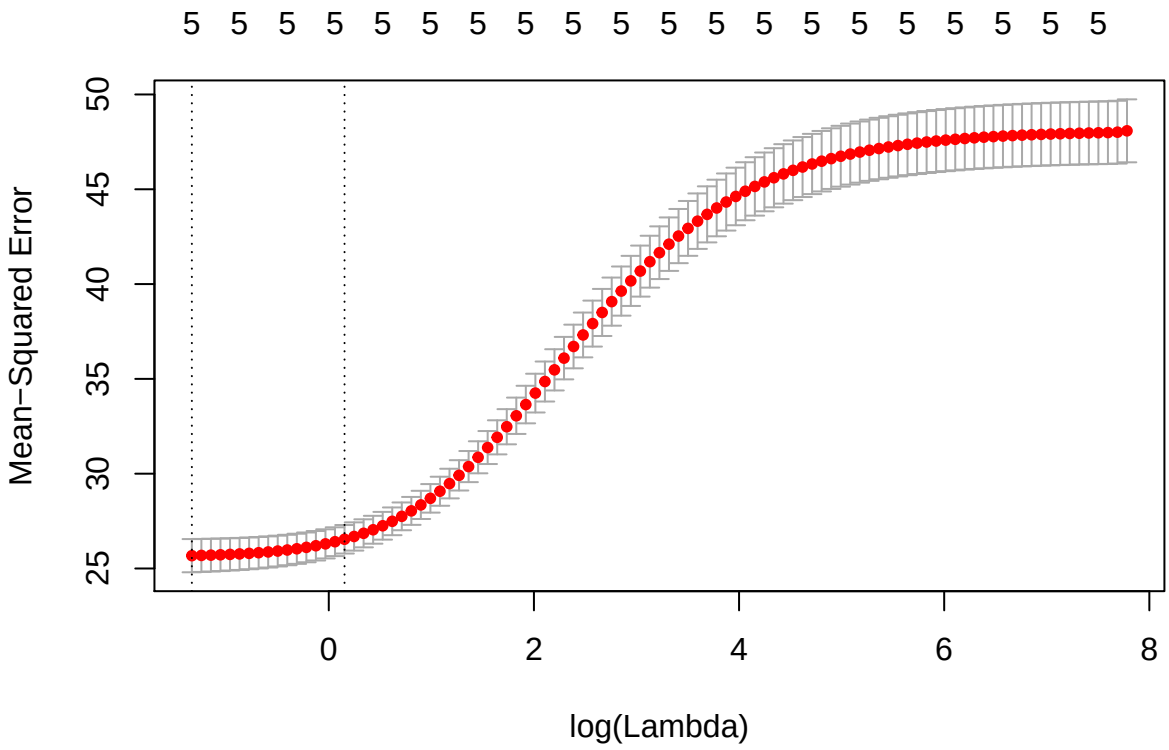
```
## Linear Regression
##
## 1052 samples
##    5 predictors
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 841, 841, 842, 842, 842
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   5.076573  0.4687891  3.953864
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

Como se puede observar, una vez hemos aplicado métodos lineales para intentar ajustar nuestro modelo, el error en el conjunto de entrenamiento $E_{in} = 5.070389$, teniendo en cuenta que estamos aplicando mínimos cuadrados en regresión. La métrica de error $R^2 = 0.4672993$, indica el porcentaje de varianza que nuestro modelo puede explicar sobre las etiquetas, por lo tanto podemos concluir que el ajuste es malo y también lo será el error fuera de la muestra. Aún no estamos cerca de una solución valida.

Ridge Regression

En este apartado vamos a aplicar la regularización sobre nuestro conjunto de datos. Primeramente vamos a ejecutar validación cruzada para obtener nuestro parámetro λ de regularización; para ello usaremos la función **cv.glmnet()**. Es importante establecer el parámetro *alpha* de la función a cero, así indicaremos que queremos realizar la regresión *Ridge*.

```
cv.ridge <- cv.glmnet(x = as.matrix(air.Tr), y = Etiquetas.Tr, alpha = 0)
plot(cv.ridge)
```




```
cv.ridge$lambda.min
```

```
## [1] 0.2631638
```

Hemos obtenido el coeficiente de λ que nos proporciona un menor error en la muestra. Ahora vamos a entrenar nuestro modelo usando el resultado obtenido como parámetro de regularización, para ello, volveremos a usar el método **train()**, junto con el método de control definido en el apartado de regresión lineal anterior.

```
fitRidge <- train(x = air.Tr, y = Etiquetas.Tr,
                  method = "glmnet",
                  family = "gaussian",
                  trControl = control,
                  tuneGrid = expand.grid(.alpha = 0, .lambda = cv.ridge$lambda.min),
                  metric = "Rsquared"
            )
```

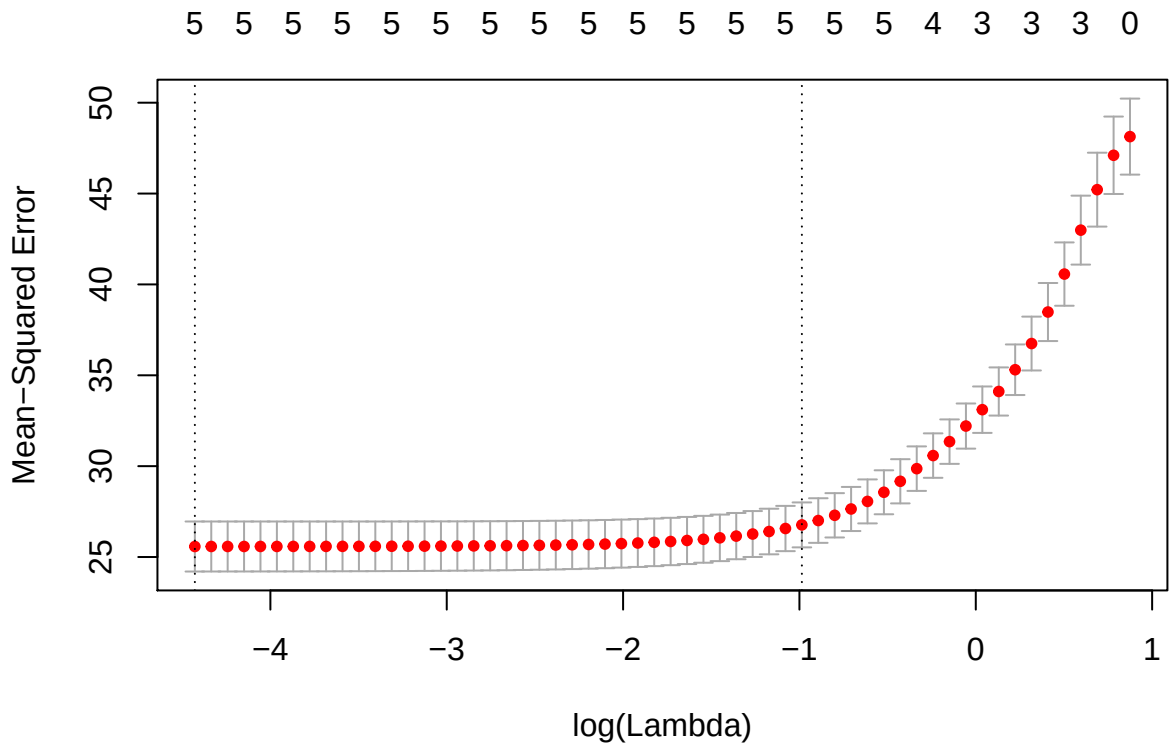
$MSE = 5.0359879$
$R^2 = 0.4724097$

Como podemos observar, hemos obtenido una mejora muy leve e insignificativa en nuestro modelo a pesar de regularizarlo, pueden existir varias causas por las que esto ha ocurrido, pero dado que el procedimiento seguido es el correcto, todo indica a que la clase de funciones que estamos usando no es capaz de explicar los datos y por tanto, necesitaríamos aumentar la clase de funciones a una más compleja. En posteriores apartados realizaremos dicho aumento de la clase de funciones para comprobarlo, a continuación seguiremos intentando regularizar con el método *Lasso*.

The Lasso

Ahora vamos a aplicar la regularización con Lasso. Usaremos las mismas funciones del apartado anterior, con la única diferencia de que el parámetro *alpha* será 1 en este caso para indicar que queremos usar este método. Primeramente vamos a obener el coeficiente λ al igual que en el apartado anterior.

```
cv.lasso <- cv.glmnet(x = as.matrix(air.Tr), y = Etiquetas.Tr, alpha = 1)
plot(cv.lasso)
```




```
cv.lasso$lambda.min
```

```
## [1] 0.01193416
```

Entrenamos de nuevo nuestro modelo usando el método **train()** y el parámetro de regularización obtenido.

```
fitRidge <- train(x = air.Tr, y = Etiquetas.Tr,
                  method = "glmnet",
                  family = "gaussian",
                  trControl = control,
                  tuneGrid = expand.grid(.alpha = 1, .lambda = cv.lasso$lambda.min),
                  metric = "Rsquared"
            )
```

$MSE = 5.0289337$
$R^2 = 0.4738867$

Una vez obtenidos estos resultados, se corroboran las hipótesis realizadas en el apartado anterior, debemos aumentar la clase de funciones puesto que la clase actual es demasiado simple como para explicar y aprender del conjunto de datos de entrenamiento que disponemos.

Aumento en la clase de funciones

Hemos comprobado que disponemos de un conjunto de datos que resulta ser complejo para ser ajustado con métodos lineales simples, incluso usando métodos con regularización; la única solución aparente es incrementar la clase de funciones. Hemos propuesto realizar mediante un bucle, un aumento de la clase de funciones de forma iterativa, e iremos calculando el error que se produce en cada una de las iteraciones mediante el uso de la regularización *Ridge* y *Lasso*.
Como métrica para comprobar las diferentes clases utilizaremos R^2, la cual, como se ha mencionado en otros

apartados, es una métrica que nos indica el porcentaje de la varianza que nuestro modelo es capaz de explicar, por tanto, cuanto mayor sea esta métrica, obtendremos un modelo que es capaz de explicar los datos de mejor forma. Hay que tener en cuenta que podemos realizar un sobreajuste u *Overfitting*, en el caso de que nuestro modelo esté ajustándose demasiado a los datos, y esta métrica nos puede ayudar a identificar este hecho.

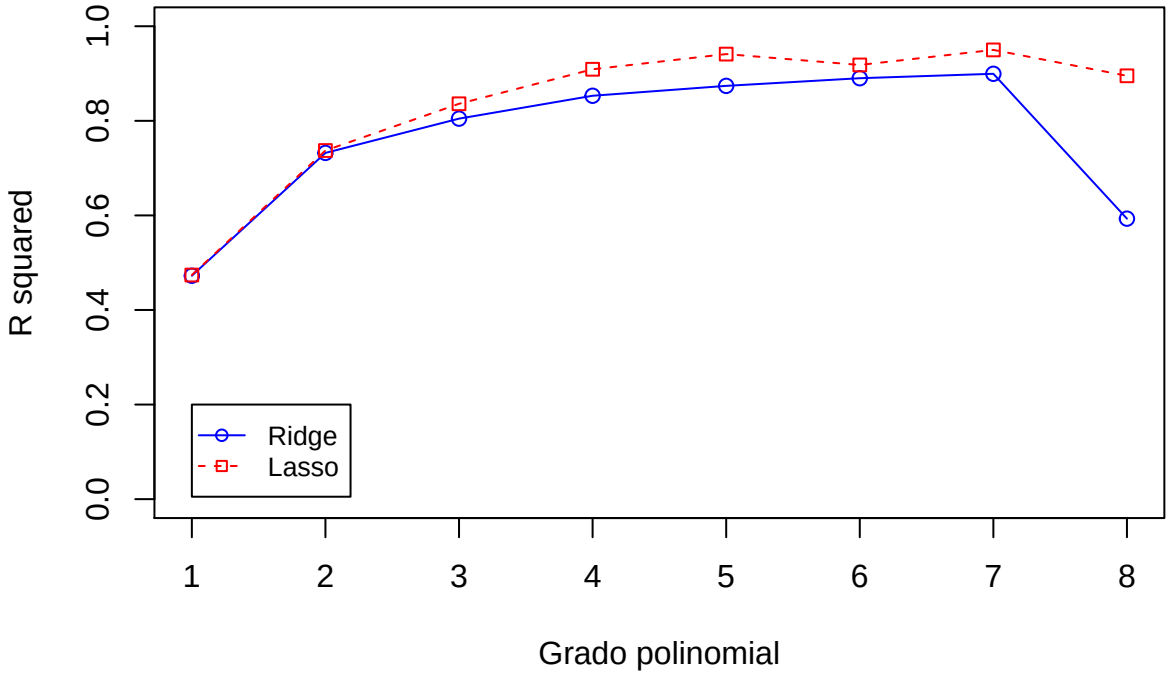
Haremos uso de la función **poly()** para ir incrementando el grado de nuestro conjunto de datos, y con ello, aumentando su complejidad.

```
train = air.Tr

maxIt <- 8
errores <- matrix(, nrow = 2, ncol = maxIt)

for (i in 1:maxIt){
  air.Tr  <- poly(as.matrix(train), degree = i, raw = T)

  cv.ridge <- cv.glmnet(x = as.matrix(air.Tr), y = Etiquetas.Tr, alpha = 0)
  cv.lasso <- cv.glmnet(x = as.matrix(air.Tr), y = Etiquetas.Tr, alpha = 1)

  minRidge = cv.ridge$lambda.min
  minLasso = cv.lasso$lambda.min

  fitRidge <- train(x = air.Tr, y = Etiquetas.Tr,
                    method = "glmnet",
                    family="gaussian",
                    trControl = control,
                    tuneGrid = expand.grid(.alpha = 0,.lambda = minRidge),
                    metric = "Rsquared"
  )

  fitLasso <- train(x = air.Tr, y = Etiquetas.Tr,
                    method = "glmnet",
                    family="gaussian",
                    trControl = control,
                    tuneGrid = expand.grid(.alpha = 1,.lambda = minLasso),
                    metric = "Rsquared"
  )
  errores[1,i] <- rsquare(predict(fitRidge), Etiquetas.Tr)
  errores[2,i] <- rsquare(predict(fitLasso), Etiquetas.Tr)

  cat("Polinomio Grado ", i, "\n")
  cat("----------------", "\n")
  cat("Ridge", "\n")
  cat("Ein = ", errores[1,i] , "\n")

  cat("----------------", "\n")
  cat("Lasso", "\n")
  cat("Ein = ", errores[2,i] , "\n")
  cat("================", "\n")
}
```

```
## Polinomio Grado  1
## ----------------
## Ridge
## Ein =  0.4724097
```

```
## ----------------
## Lasso
## Ein =  0.4738818
## ================
## Polinomio Grado  2
## ----------------
## Ridge
## Ein =  0.7321822
## ----------------
## Lasso
## Ein =  0.7372127
## ================
## Polinomio Grado  3
## ----------------
## Ridge
## Ein =  0.8046102
## ----------------
## Lasso
## Ein =  0.8360428
## ================
## Polinomio Grado  4
## ----------------
## Ridge
## Ein =  0.8530146
## ----------------
## Lasso
## Ein =  0.9090264
## ================
## Polinomio Grado  5
## ----------------
## Ridge
## Ein =  0.8739389
## ----------------
## Lasso
## Ein =  0.9411223
## ================
## Polinomio Grado  6
## ----------------
## Ridge
## Ein =  0.8901047
## ----------------
## Lasso
## Ein =  0.9181297
## ================
## Polinomio Grado  7
## ----------------
## Ridge
## Ein =  0.899459
## ----------------
## Lasso
## Ein =  0.9499739
## ================
## Polinomio Grado  8
## ----------------
```

```
## Ridge
## Ein =  0.5931301
## ----------------
## Lasso
## Ein =  0.8952076
## ================
```



Con el gráfico anterior podemos comparar visualmente la mejora significativa que produce aumentar la clase de funciones en nuestro modelo, por lo tanto podemos comprobar que las hipótesis dispuestas en los apartados anteriores sobre la causa de que nuestro modelo sea malo, eran ciertas. El error calculado ha sido el producido en el conjunto de entrenamiento, por lo tanto no conocemos a priori si hemos realizado sobreajuste y hemos seguido los datos de la muestra en lugar de crear un modelo genérico que sea capaz de explicar y predecir nuevos datos.

Podemos ver como el método *Ridge* ha empeorado cuando se ha aumentado la clase de funciones a un polinomio de octavo grado. La curva en la función es visualmente representativa y *Lasso* funciona mejor a lo largo de las iteraciones, el crecimiento de la función se ralentiza cuando alcanzamos un polinomio de grado 4 aproximadamente, por lo que estableceremos nuestro objetivo en analizar la clase de funciones cúbica y cuártica, y nuestro método definitivo Lasso.

```
test <- air.Tst

for (i in 3:4){
  air.Tst  <- poly(as.matrix(test), degree = i, raw = T)

  cv.lasso <- cv.glmnet(x = as.matrix(air.Tst), y = Etiquetas.Tst, alpha = 1)

  minLasso = cv.lasso$lambda.min

  fitLasso <- train(x = air.Tst, y = Etiquetas.Tst,
                    method = "glmnet",
                    family="gaussian",
                    trControl = control,
                    tuneGrid = expand.grid(.alpha = 1,.lambda = minLasso),
```

```
                    metric = "Rsquared"
  )

  cat("Polinomio grado ", i, "\n")
  cat("----------------", "\n")
  cat("Lasso", "\n")
  cat("Eout = ", rsquare(predict(fitLasso), Etiquetas.Tst) , "\n")
  cat("================", "\n")
}
```

```
## Polinomio grado  3
## ----------------
## Lasso
## Eout =  0.8702884
## ================
## Polinomio grado  4
## ----------------
## Lasso
## Eout =  0.9316758
## ================
```

Hemos obtenido muy buenos resultados a la hora de predecir en el conjunto de datos de validación, una mejora significativa a la hora de incrementar la clase de funciones. De la misma manera, podríamos haber usado una clase de funciones más compleja aunque sin la garantía de que vayamos a obtener un error fuera de la muestra mejor que el actual, por lo tanto consideramos que nuestro modelo final será mediante el uso del método Lasso, forzando la estimación de los coeficientes de los predictores para que éstos tiendan a cero, y el uso de una clase de funciones cuártica, ya que nos ha aportado buenos resultados, sin producir un sobreajuste del modelo. Si hubiesemos aumentado nuestra clase hasta grado 5 podríamos haber llegado a un sobreajuste ya que llegaríamos a un mejor resultado dentro de la muestra pero comenzaríamos a ver como va empeorando fuera de ella.

Por tanto en conclusión final nuestro modelo propuesto será el modelo con regularización Lasso y que usa el polinomio de grado 4 de características.

Clasificación

Ahora se nos presenta un problema de clasificación que podemos considerar una extensión del usado en las practicas anteriores. En este caso, no solo queremos clasificar entre dos dígitos, ahora tenemos imágenes con los dígitos de 0-9.

El objetivo por tanto va a ser reconecer cada imagen y determinar de que dígito se trata con la mayor exactitud posible. Para ello disponemos dos archivos cvs, un train y un test. En estos ya encontramos una parte de trabajo realizado ya que tenemos múltiples características extraidas de las imágenes. Vamos a cargar estos dataset.

```
digitosTr = read.csv("datos/optdigits_tra.csv")
digitosTs = read.csv("datos/optdigits_tes.csv")
names(digitosTs) = names(digitosTr)
```

Hemos hecho un pequeño ajuste para que conincidan los nombres del data.frame de ambos conjuntos y no tener problemas más adelante a la hora de aplicar las mismas trasformaciones de train para test.
Estos archivos podemos encontrarlos en decsai y en la dirección https://archive.ics.uci.edu/ml/datasets/optical+recognition+of+handwritten+digits .
En la web podemos encontrar una descripción más detallada de los datos que vamos a usar, en concreto nos interesa ver la distribución que tienen estos datos:

9. Class Distribution Class: No of examples in training set 0: 376 1: 389 2: 380 3: 389 4: 387 5: 376 6: 377 7: 387 8: 380 9: 382

Estos datos nos dicen que tenemos una distribución equilabrada y no tenemos el problema de tener pocos ejemplos de un determinado dígito. Ahora vamos a ver el número de caracteristicas de las que disponemos para nuestra clasificación

```
    cat("Dimension del conjunto train:", dim(digitosTr), "\n")
```

```
## Dimension del conjunto train: 3822 65
```

```
    cat("Dimension del conjunto train:", dim(digitosTs), "\n")
```

```
## Dimension del conjunto train: 1796 65
```

Como podemos ver contamos con un buen número de ejemplos para después realizar el test por lo que usaremos el train por completo para entrenar. Por otro lado, vemos que tenemos bastantes características para intentar realizar un buen modelo.

Preprocesamiento

Ahora se nos plantea una de las primeras decisiones a tomar para resolver nuestro problema, qué preprocesamiento aplicar a nuestros datos. En primer lugar, vamos a decidir usar *YeoJohnson, center, scale*. Usaremos en nuestro caso YeoJohnson en lugar de BoxCox ya que son trasformaciones similares pero con este método se permiten valores iguales a 0 y negativos, mientras que en Box-Cox se fuerza a que sean estrictamente positivos. Por otro lado, usaremos center y scale para normalizar los datos, estos métodos le restan la media y dividen por la desciación típica, lo cual vimos en clase que era una forma buena de normalización. Por último cabe decir que hemos desechado usar PCA. Este método se usa para reducir la dimensionalidad pero como hemos explicado anteriormente se usa en casos que la dimensionalidad es muy alta. Otros métodos para seleccionar caracteristicas que podríamos usar sería *regsubsets* pero en nuestro caso vamos a usar regularización con Lasso, como explicaremos más adelante que tambien nos ayuda a quitar las carecterísticas que no aportan información.

Lo que si vamos a usar en este caso es la función *nearZeroVar* para eliminar caracteristicas cuya varianza es baja y no nos aportan mucho. Debemos tener una cosa clara antes de comenzar a aplicar transformaciones a nuestros datos y es que debemos aplicarla de igual forma a los datos test. El preprocesado lo vamos a realizar más adelante indicandoselo a la función *train* que nos ofrece la facilidad de indicarselo como parámetro. Ahora vamos a usar *nearZeroVar*

```
columnascero = nearZeroVar(digitosTr)
datosTr = digitosTr[,-columnascero]
datosTs = digitosTs[,-columnascero]
```

Definir conjuntos

El siguiente paso va a ser dividir los datos de las etiquetas para poder usarlas por separado y de una manera más clara. Como conjuntos de train y test usaremos los que nos han proporcionado en los archivos. En este caso no tenemos que hacer divisiones y, aunque como método de validación vamos a usar *validación cruzada*, también lo haremos de forma automática con la función train ya que se lo indicaremos con el control. Realizaremos una validación cruzada con 5 conjuntos.

```
datosTr = data.frame(datosTr[,-ncol(datosTr)])
datosTs = data.frame(datosTs[,-ncol(datosTs)])
etiquetasTr = as.factor(make.names(digitosTr$X0.26))
etiquetasTs = as.factor(make.names(digitosTs$X0.26))
```

Las etiquetas tenemos que almacenarlas como factor, para que los métodos usados no tengan problema y actuen como clasificadores. Si mantenemos dichas etiquetas como números normales, se interpretarían como regresión y pueden darnos errores más adelante. Además, para generar estos factores usaremos *make.name* para que sean nombres válidos para R.

Elección de modelos y regularización

El siguiente paso va a ser seleccionar los modelos que vamos a usar. En primer lugar seleccionamos la clase de funciones, que en el caso de esta práctica solo podíamos usar las lineales.

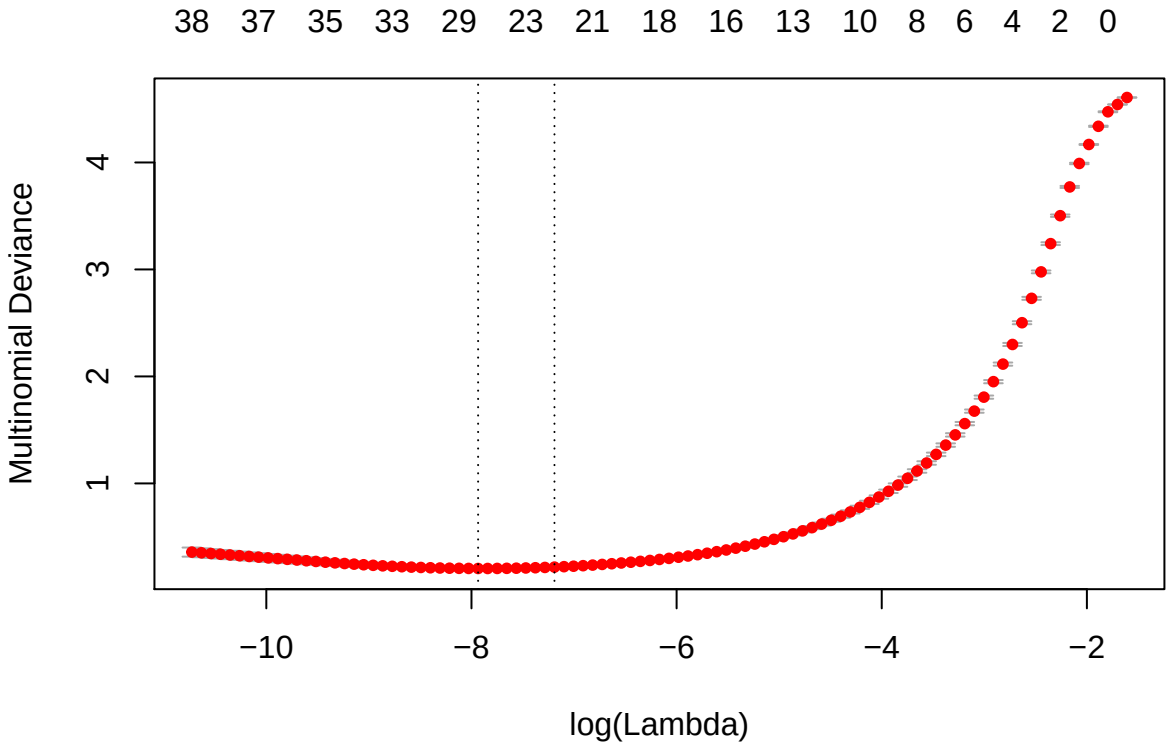
Por otro lado también tenemos que discutir el uso de regularización. Como vimos en clase de teoría podemos decir que es casi con seguridad un acierto usar la regularización para intentar evitar el overfitting. Si no aplicamos regularización podriamos estar intentando ajustar demasido ampliando la clase de funciones y cayendo en el sobreajuste. Por tanto vamos a usar los dos modelos de regularización *Lasso y Ridge*. Para ello vamos a usar la función *glmnet* con la cual podemos usar ambos casos de regularización y con la que ádemas podemos hacer clasificación multietiqueta.

En un principio, la idea inicial que se planteaba para resolver el problema era la de convertir dicho problema en un problema de clasificación binaria, para ello, lo que haríamos sería crear un modelo para cada dígito y el cual nos decía si ese caso se correspondía con ese digito o no, es decir *1 vs all*. Esto conllevaría un proceso complejo que realmente es lo que tenemos internamente en la función glmnet con multinomial, transforma el problema en binomial. Por tanto, ese va a ser nuestra función y vamos a usar tanto Lasso que descarta características como Ridge.

Seleccionar hyperparámetros

Antes de poder aplicar el metodo tenemos que ajustar los hyperparámetros. Para ello necesitamos *alpha*, que nos va a servir para determinar si usar *Lasso* o *Ridge* (1,0). Si no ponemos este parámetro, el método usado sería la elasticnet la cual no esta permitida en esta práctica. El segundo hyperparametro será el λ que usar para la regularización; para calcularlo usaremos validación cruzada. También vamos a definir con *trainControl* el control para la función train el cual será usar validación cruzada con 5 conjuntos.

```
ctrol = trainControl(method="cv", number=5)
hipL=cv.glmnet(as.matrix(datosTr),etiquetasTr,family="multinomial",alpha=1)

plot(hipL)
```



Como podemos ver esta función va buscando los mejores valores para lambda que vamos a usar para calcular nuestro modelo. Usaremos como métrica el accuracy ya que consideramos la más acertada para usar teniendo multietiqueta. En este caso usar las curva ROC sería algo mas complejo ya que tendríamos una curva para cada modelo como el que describiamos antes de 1 vs all.

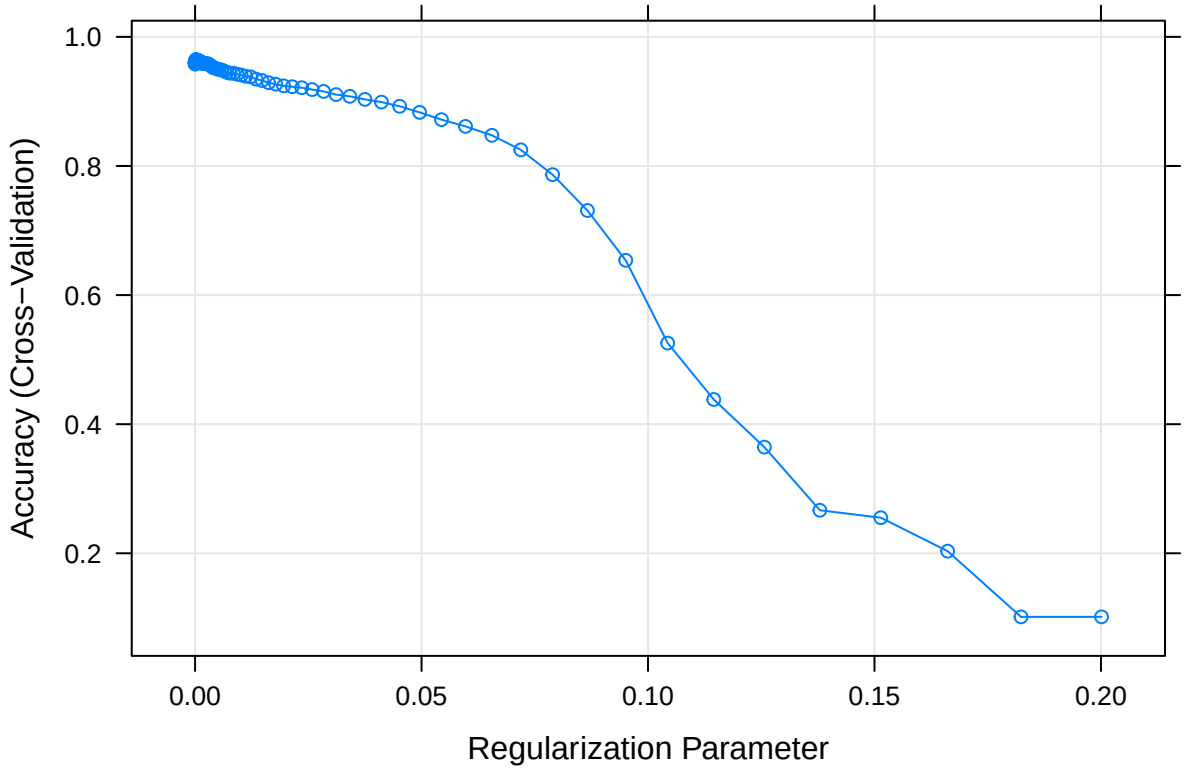
Ajuste de modelos

Comenzamos con Lasso:

```
fitLasso <- train(x=datosTr, y=etiquetasTr,
                  method = "glmnet",
                  family="multinomial",
                   metric="Accuracy",
                  preProcess = c( "center", "scale", "YeoJohnson"),
                  trControl = ctrol,
                  tuneGrid = expand.grid(.alpha = 1,.lambda = hipL$lambda)
)
```

Mencionar que usando estas funciones, en *fitLasso* no solo tenemos el modelo si no que almacena los datos de training, la metrica usada, el tiempo usado para el cálculo y muchos datos que pueden ser relevantes.

```
plot(fitLasso)
```




```
fitLasso$bestTune
```

```
##    alpha       lambda
## 25     1 0.0002048394
```

Puede ser realmente útil toda esta información ya que como podemos ver el accuracy obtenido con los distintos valores de lambda y cual a sido la mejor configuración de los hyperparametros.

```
confusion = table(etiquetasTr, predict(fitLasso))
print(confusion)
```

```
##
## etiquetasTr  X0  X1  X2  X3  X4  X5  X6  X7  X8  X9
##          X0 375   0   0   0   0   0   0   0   0   0
##          X1   0 381   1   1   0   0   0   1   3   2
##          X2   0   0 380   0   0   0   0   0   0   0
##          X3   0   1   0 386   0   1   0   0   0   1
##          X4   0   0   0   0 386   0   0   0   0   1
##          X5   0   1   2   1   0 371   0   0   0   1
##          X6   0   2   0   0   1   0 374   0   0   0
##          X7   0   0   0   0   0   0   0 387   0   0
##          X8   0   4   0   0   2   1   0   0 373   0
##          X9   0   3   0   1   3   0   0   0   1 374
```

Vamos a ver como quedaría la matriz de confusión para ver como está funcionando nuestro modelo dentro de la muestra. Como podemos ver, la mayoría de las clases predichas coinciden con las etiquetas reales que tenía el problema. Se equivoca en algunas pero podemos decir que estamos consiguiendo un modelo bastante bueno ya que este error podría darse por el ruido. Vamos a calcular para tener un porcentaje de acierto de nuestro modelo. Este sería el E_{in} ya que estamos dentro de la muestra. No debemos olvidar que no hemos usado el training, si no que lo hemos usado con validación cruzada por lo tanto ya hemos realizado validación y tenemos el mejor modelo que se ha calculado con los datos.

```
  print(sum(diag(confusion)*100 / sum(confusion)))
```

```
## [1] 99.08425
```

El resultado en este caso ha sido bastante bueno y hemos obtenido un porcentaje de acierto bastante alto, pero vamos a probar ahora con *Ridge* para comprobar si este podría ser mejor.

```
hipR=cv.glmnet(as.matrix(datosTr),etiquetasTr,family="multinomial",alpha=0)
fitRidge <- train(x=datosTr, y=etiquetasTr,
                  method = "glmnet",
                  family="multinomial",
                   metric="Accuracy",
                  preProcess = c( "center", "scale", "YeoJohnson"),
                  trControl = ctrol,
                  tuneGrid = expand.grid(.alpha = 0,.lambda = hipR$lambda)
)
confusionR = table(etiquetasTr, predict(fitRidge))
print(confusionR)
```

```
##
## etiquetasTr  X0  X1  X2  X3  X4  X5  X6  X7  X8  X9
##          X0 373   0   0   0   1   0   1   0   0   0
##          X1   0 364   4   0   0   0   2   2   8   9
##          X2   0   2 369   1   0   1   0   1   3   3
##          X3   0   1   1 376   0   3   0   2   3   3
##          X4   1   2   0   0 366   0   4   1   4   9
##          X5   0   1   2   1   0 363   2   0   0   7
##          X6   0   2   0   0   1   0 374   0   0   0
##          X7   0   1   0   1   0   0   0 385   0   0
##          X8   0  11   0   1   5   1   3   0 359   0
##          X9   1   8   0   4   7   0   0   2   6 354
```

```
print(sum(diag(confusionR)*100 / sum(confusionR)))
```

```
## [1] 96.36316
```

Ambos errores son bastante buenos y como podemos ver son dos ajustes que, a prioiri, no podemos descartar. Por tanto nos quedaremos con ambos y los usaremos para probarlos con el test. La métrica usada en este caso ha sido el Accuracy, es decir, el numero total de aciertos dividido entre el número total de casos $ aciertos/casos$.

Elección de métricas

En este caso tanto las curvas ROC como el Fit-Score no serían las mejores ya que estamos ante un problema de multietiquetado. El fit-score es más acertado usarlo cuando tenemos una muestra desbalanceada. Hemos usado el Accuracy y lo usaremos para comprobar finalmente con el test. Un problema que podría presentar nuestra métrica es si tuvieramos una muestra de un tamaño muy grande. Pensemos que tenemos un millón de ejemplos y que conseguimos un acierto del 99%, es decir nos equivocamos en 1%. Podríamos decir que tenemos un buen modelo pero si lo pensamos realmente estamos equivocandonos en 100000 ejemplos. Cierto es que tenemos un 99% de acierto pero con una muestra tan grande quizás deberiamos intentar bajar ese error aunque también es verdad que puede deberse a ruido. Nosotros no tenemos ese caso y vamos a comprobar en el test si hemos encontrado un buen modelo que no se haya sobreajustado y que de buenos resultados.

Estimar el Eout

Vamos a realizar la prueba con el test de ambos modelos y finalmente elegiremos uno.

```
testLassotable = table(etiquetasTs, predict(fitLasso,datosTs))
print(testLassotable)
```

```
##
## etiquetasTs  X0  X1  X2  X3  X4  X5  X6  X7  X8  X9
##          X0 176   0   0   0   1   0   0   0   0   0
##          X1   0 174   0   0   2   0   0   0   4   2
##          X2   0   4 169   3   0   0   1   0   0   0
##          X3   0   0   2 171   0   4   0   1   1   4
##          X4   0   1   0   0 175   0   0   1   1   3
##          X5   0   0   1   2   0 175   1   0   1   2
##          X6   0   1   0   0   2   0 177   0   1   0
##          X7   0   0   0   0   2   7   0 161   2   7
##          X8   0   9   0   1   0   1   0   0 155   8
##          X9   1   2   0   0   3   4   0   0   4 166
```

```
print(sum(diag(testLassotable)*100 / sum(testLassotable)))
```

```
## [1] 94.59911
```

```
testRidgetable = table(etiquetasTs, predict(fitRidge,datosTs))
print(testRidgetable)
```

```
##
## etiquetasTs  X0  X1  X2  X3  X4  X5  X6  X7  X8  X9
##          X0 176   0   0   0   1   0   0   0   0   0
##          X1   0 160   4   0   1   1   1   0   3  12
##          X2   0   4 166   2   0   0   0   2   2   1
##          X3   2   0   1 168   0   3   0   3   2   4
##          X4   0   2   0   0 175   0   0   1   2   1
##          X5   0   1   0   0   0 178   1   0   0   2
##          X6   0   4   0   0   2   0 174   0   1   0
##          X7   0   0   0   0   2   5   0 164   2   6
##          X8   0  17   0   1   0   2   1   1 145   7
##          X9   0   3   0   1   5   3   0   0   3 165
```

```
print(sum(diag(testRidgetable)*100 / sum(testRidgetable)))
```

```
## [1] 93.04009
```

La matriz de confusión nos ayuda a detectar posibles errores entre clases y en nuestros modelos se destaca que existe una pequeña confusión entre los digitos 1 y 8 ya que es donde más errores se cometen. Esto es difícil de de detectar por que se debe pero lo más probable sea que un predictor el cual es muy bueno para diferenciar otros digitos en este caso es muy parecido y nos hace cometer un error mayor. Aunque pensemos que diferencia un 1 de un 8 debe ser fácil tenemos que tener en cuenta que estamos usando muchas características y en algunas pueden parecerse.

A pesar de este pequeño error, tanto dentro como fuera de la muestra tenemos un resultado bastante bueno aunque queda por encima el modelo que se había definido usando la regularización Lasso. Además, este modelo tiene otra ventaja y es que nos ayuda a reducir características y por tanto simplificar nuestro modelo y en un futuro nos ahorrará tiempo de computación y a la hora de realizar extracción de características. Por tanto, como conclusión final, podemos ver que nuestras decisiones han sido buenas ya que hemos conseguido un buen error fuera de la muestra y por tanto nuestro modelo estará preparado para nuevos datos que se desconocen, dándonos ciertas garantías como veíamos con la Desigualdad de Hoeffding.

El preprocesado parece el correcto ya que, a priori, no conocíamos los datos y podíamos encontrar datos muy dispersos, con esa normalización hemos llegado a una buena solución.
El método usado para generar el modelo final sería el Lasso el cual nos ayudaría a eliminar caractericticas que realmente no han sido útiles y, como podemos ver, tenemos un clasificador bastante bueno que aunque posee error puede que se deba al posible ruido existente. Además, como veíamos en clase, el momento en el cual dejar de ajustar dentro de la muestra era una de las grandes incógnitas, ya que podemos caer en sobreajuste con la muestra o quedarnos lejos de un buen resultado. En nuestro caso creemos que es un modelo bastante bueno y que puede servirnos para resolver el problema de clasificación que se nos presentaba con una buena calidad.