Proyecto de Regresión para Housting

Julio A. Fresneda - 49215154F

June 4, 2018

Introducción

La base de datos "Housting", es una base de datos que contiene datos sobre 506 casos sobre el valor medio de viviendas de Boston ocupadas por sus propietarios, así como algunos atributos de esas viviendas.

En concreto, los atributos que tenemos en nuestra base de datos son los siguientes.

1. CRIM Tasa de criminalidad per cápita por ciudad
2. ZN Proporción de tierra residencial dividida en zonas para lotes de más de 25,000 pies cuadrados.
3. INDUS Proporción de acres (medida de superficie) de negocios no minoristas por ciudad
4. CHAS Variable ficticia de Charles River (= 1 si el tramo limita el río; 0 de lo contrario)
5. NOX Concentración de óxidos nítricos (partes por 10 millones)
6. RM Número promedio de habitaciones por vivienda
7. AGE Proporción de unidades ocupadas por sus propietarios construidas antes de 1940
8. DIS Distancias ponderadas a cinco centros de empleo de Boston
9. RAD Indice de accesibilidad a las autopistas radiales
10. TAX Tasa de impuesto a la propiedad a valor completo por $10,000
11. PTRATIO Relación alumnos-profesor por ciudad
12. B 1000 (Bk - 0.63) ^ 2 donde Bk es la proporción de personas por color por ciudad
13. LSTAT % menor status de la población
14. MEDV Valor medio de las viviendas ocupadas por sus propietarios, en $1000

Lo que nosotros intentaremos conseguir en este proyecto es obtener un modelo basado en regresión que consiga predecir la MEDV a partir del resto de atributos, con un margen de error lo más pequeño posible.

Para ellos, compararemos dos modelos: Uno obtenido a través de Regresión Lineal Regularizada, y otro obtenido a través de Random Forests.

Preprocesamiento de datos

Para este proyecto usaremos la semilla 11.

```
set.seed(11)
library(glmnet)
```

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

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

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

```
library(randomForest)
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

Lo promero que vamos a hacer es cargar los datos.

```
data <- read.csv("datos/housing-data.csv", header=T, sep=" ") # 1. Lectura de datos
```

Vamos a separar estos datos en dos conjuntos: Uno de train, para entrenar nuestros modelos, y otro de test, para ponerlos a prueba. Aproximadamente el 80% de los datos irán al conjunto de train, mientras que el 20% restante irán al conjunto de test. Esta división la vamos a hacer de forma aleatoria. También separaremos estos conjuntos en características y etiquetas (MEDV).

```
rowNumber <- sample(1:nrow(data), 0.8*nrow(data))
train = data[rowNumber,]
test = data[-rowNumber,]

# Datos de entrenamiento
xtrain = data.matrix(train[,1:13])
ytrain = data.matrix(train[,14])

# Datos de test
xtest = data.matrix(test[,1:13])
ytest = data.matrix(test[,14])
```

Ya tenemos nuestros datos listos para ser usados. Ahora entrenaremos usando dos métodos distintos: Regresión Lineal Regularizada y Random Forests.

Regresión Lineal Regularizada

Metodología a usar y regularización

Para estimar un modelo vamos a usar el paquete "glmnet".

Glmnet es un paquete que ajusta modelos lineales generalizados a través de la máxima probabilidad regularizada. Se puede regularizar usando LASSO, Ridge o un híbrido (Elastic Net). Glmnet puede ajustar a modelos de regresión lineal, logística y multinomial, poisson y Cox, entre otros. Nosotros usaremos el paquete para ajustar un modelo de regresión lineal.

En resumen, glmnet sirve para minimizar la función de error para regresión:

$$min_{\beta_0,\beta} \frac{1}{N} \sum_{i=1}^{N} w_i l(y_i, \beta_0 + \beta^T x_i) + \lambda[(1-\alpha)||\beta||_2^2/2 + \alpha||\beta||_1]$$

Donde β son los pesos, λ es un hiperparámetro que se elige manualmente, α es un valor entre 0 y 1 que elegiremos para priorizar un tipo de regularización u otra, y $l(y, \eta)$ es $\frac{1}{2}(y-\eta)^2$.

Se pueden usar tres tipos de regularización, dependendo de qué valor de α escojamos:

Ridge

La regularización con Ridge crea un modelo de regresión lineal que se penaliza con la norma L2, que es la suma de los coeficientes al cuadrado, $\frac{\lambda}{2}||\beta||_2^2$. Esto tiene el efecto de reducir los valores de los coeficientes (y la complejidad del modelo), lo que permite que algunos coeficientes con una contribución menor a la respuesta se acerquen a cero (sin llegar a 0).

Para regularizar con Ridge, usaremos $\alpha = 0$.

LASSO

Least Absolute Shrinkage and Selection Operator (LASSO) crea un modelo de regresión que se penaliza con la norma L1, que es la suma de los coeficientes absolutos, $\lambda||\beta||_1$. Esto tiene el efecto de reducir los valores

de los coeficientes hasta el punto de poder eliminarlos (dejarlos a 0), con lo que se reduciría la complejidad del modelo.

Para regularizar con LASSO, usaremos $\alpha = 1$.

Elastic Net

Elastic Net crea un modelo de regresión que se penaliza tanto con la norma L1 como con la norma L2. Esto tiene el efecto de reducir efectivamente los coeficientes (como en Ridge) y establecer algunos coeficientes a cero (como en LASSO).

Para regularizar con Elastic Net, debemos usar $\alpha \in (0, 1)$. Nosotros usaremos $\alpha = 0.5$.

Entrenamiento

Vamos a estimar tres modelos lineales, cada uno con un tipo de regularización diferente. Nuestra función estimada debería tener la siguiente forma:

$$h_\theta(x) = \theta_1 + \theta_2 x_2 + ... + \theta_{14} x_{14}$$

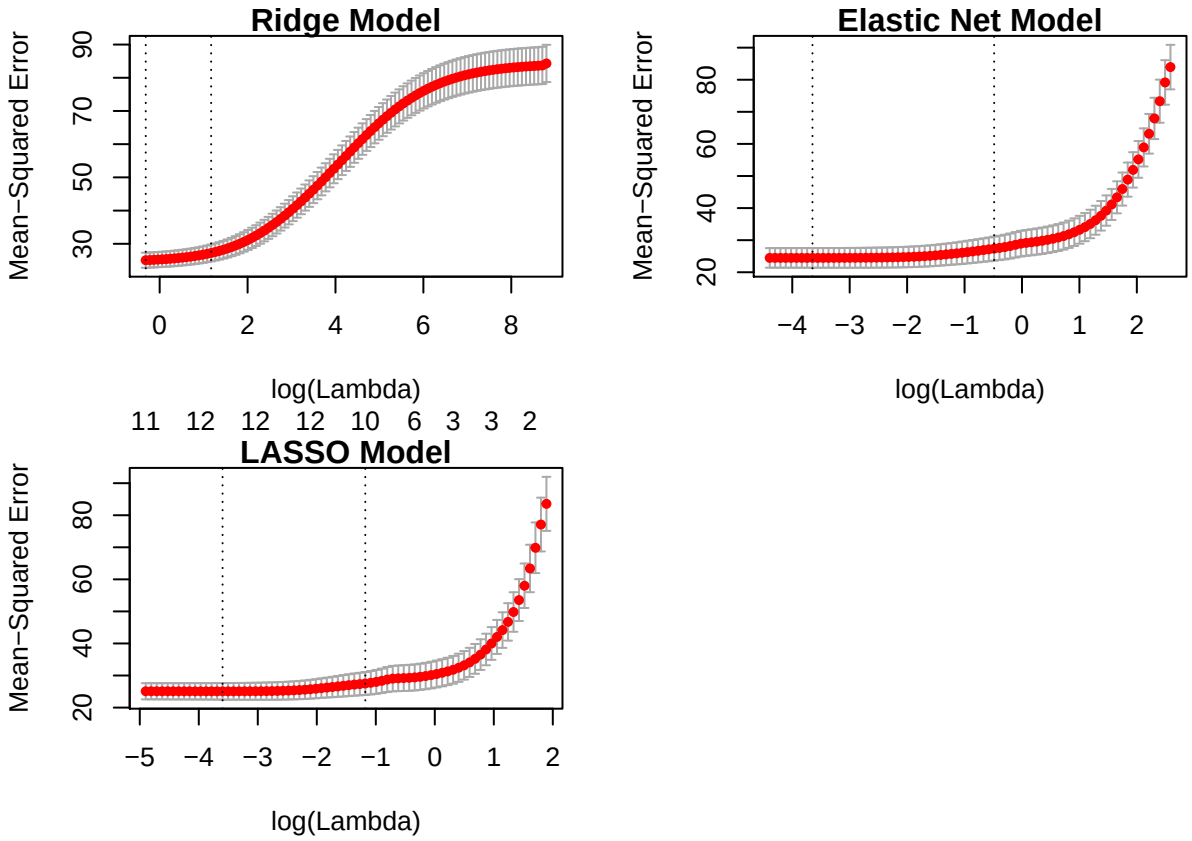
Hay dos hiperparámetros a ajustar: Lambda y alpha. Para alpha, es fácil, usaremos 0 (Ridge), 1 (LASSO), y 0.5 (Elastic Net). Para lambda, elegiremos el mejor valor usando Cross-Validation. Recordemos que Cross-Validation es un sistema que sirve para elegir hiperparámetros (o modelos) y consiste en entrenar con cada hiperparámetro k veces con k-1 partes del train, hacer la media de error del test y elegir el modelo cuya media sea mas baja. Esto no tenemos que hacerlo manualmente, el paquete glmnet incluye una función propia que lo hace automáticamente.

No vamos a necesitar normalizar, ya que la clase glmnet nos normaliza automáticamente.

El entrenamiento es el siguiente:

```
# Entrenamos con cross-validation, donde se probará con 100 distintas lambdas.
model.ridge = cv.glmnet(xtrain,ytrain,alpha=0,nlambda=100)
model.enet = cv.glmnet(xtrain,ytrain,alpha=0.5,nlambda=100)
model.lasso = cv.glmnet(xtrain,ytrain,alpha=1,nlambda=100)

# Obtenemos el MSE (Mean Squared Error) mínimo, y la lambda usada para obtenerlo.
model.ridge.mse <- model.ridge$cvm[model.ridge$lambda == model.ridge$lambda.min]
model.ridge.lambda = model.ridge$lambda.min

model.enet.mse <- model.enet$cvm[model.enet$lambda == model.enet$lambda.min]
model.enet.lambda = model.enet$lambda.min

model.lasso.mse <- model.lasso$cvm[model.lasso$lambda == model.lasso$lambda.min]
model.lasso.lambda = model.lasso$lambda.min

# Vamos a ver gráficamente los MSE para cada lambda
par(mfrow=c(2,2),mar=(c(5,5,1,1)))

plot(model.ridge)
title("Ridge Model")

plot(model.enet)
title("Elastic Net Model")
```

```
plot(model.lasso)
title("LASSO Model")
```



Podemos ver de forma muy intuitiva que en los tres casos el MSE va descendiendo conforme lambda se hace más pequeña, hasta que el MSE se acaba estabilizando. También vemos que con Ridge le ha costado un poco más llegar a unos valores de MSE comparables a los de Elastic Net y LASSO, pero al final se ha equiparado.

Veamos los MSE que obtenemos con el mejor lambda en cada caso:

```
## Ridge:
## Lambda =  0.7280293
## MSE =  25.04968
## Elastic Net:
## Lambda =  0.02604378
## MSE =  24.47388
## LASSO:
## Lambda =  0.02740984
## MSE =  25.06489
```

Como vemos, como meejores valores obtenemos es regulando con LASSO. Sin embargo, en cualquier caso tenemos un MSE demasiado alto. Debemos buscar una alternativa.

Vamos a intentar estimar un modelo un poco más complejo. Nuestra función a estimar tendrá la siguiente forma:

$$h_\theta(x) = \theta_1 + \theta_2 x_2 + ... + \theta_{14} x_{14} + \theta_{15} x_2^2 + \theta_{16} x_{16}^2 + ... + \theta_{27} x_{27}^2$$

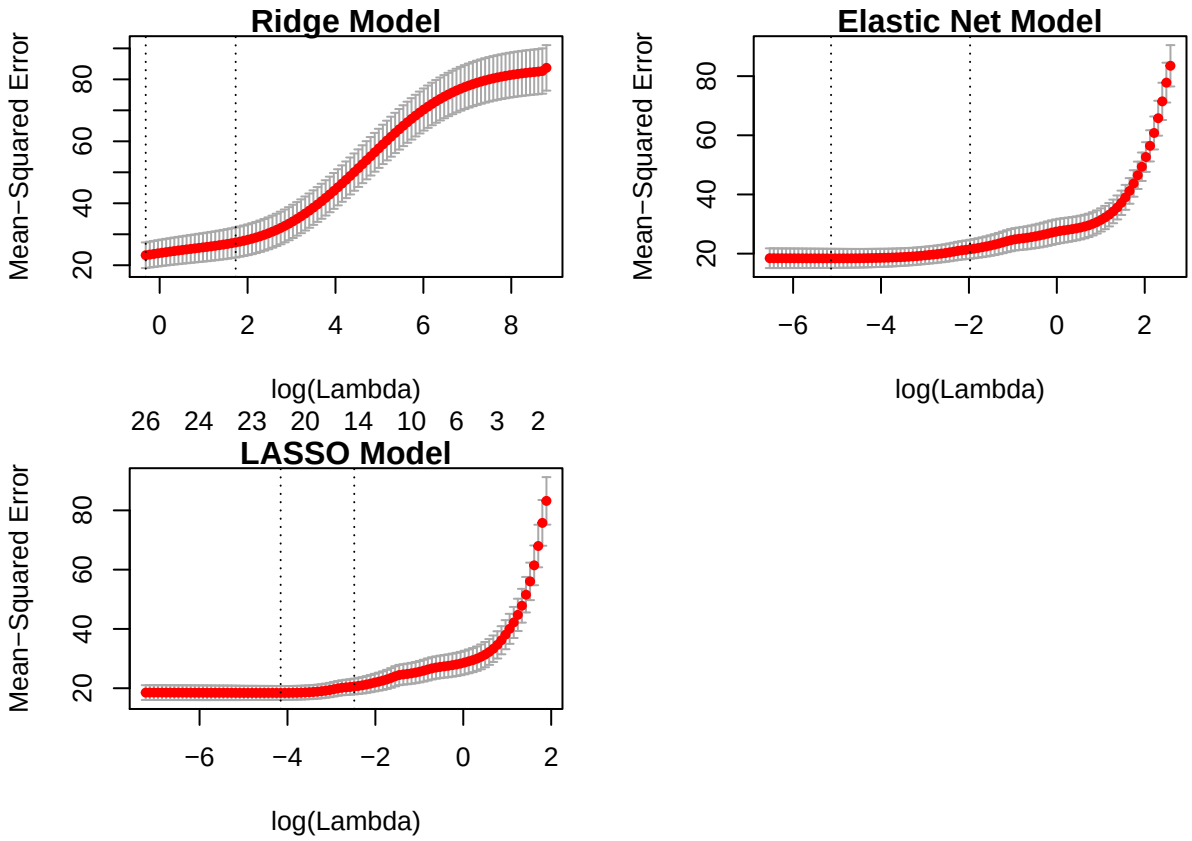
Como hemos visto que nuestro modelo anterior estaba un poco limitado para ajustar correctamente, le hemos añadido los cuadrados de cada característica, para que se pueda encontrar una solución más ajustada.

Ahora nuestro train tendrá 26 columnas de atributos en vez de 13: Las 13 primeras columnas serán como las del train anterior, y las 13 siguiente sus cuadrados. Vamos a modificar los datos:

```
squaredxtrain = xtrain
for( i in 1:nrow(xtrain) )
{
  for( j in 1:ncol(xtrain) )
  {
    if( j != 4 ) squaredxtrain[i,j] = squaredxtrain[i,j]*squaredxtrain[i,j]
  }
}

xtrain = cbind(xtrain,squaredxtrain)
```

Ya podemos entrenar nuestros nuevos modelos. Los pasos son exactamente iguales que los anteriores.

```
# Entrenamos con cross-validation, donde se probará con 100 distintas lambdas.
model2.ridge = cv.glmnet(xtrain,ytrain,alpha=0,nlambda=100)
model2.enet = cv.glmnet(xtrain,ytrain,alpha=0.5,nlambda=100)
model2.lasso = cv.glmnet(xtrain,ytrain,alpha=1,nlambda=100)

# Obtenemos el MSE (Mean Squared Error) mínimo, y la lambda usada para obtenerlo.
model2.ridge.mse <- model2.ridge$cvm[model2.ridge$lambda == model2.ridge$lambda.min]
model2.ridge.lambda = model2.ridge$lambda.min

model2.enet.mse <- model2.enet$cvm[model2.enet$lambda == model2.enet$lambda.min]
model2.enet.lambda = model2.enet$lambda.min

model2.lasso.mse <- model2.lasso$cvm[model2.lasso$lambda == model2.lasso$lambda.min]
model2.lasso.lambda = model2.lasso$lambda.min

# Vamos a ver gráficamente los MSE para cada lambda
par(mfrow=c(2,2),mar=(c(5,5,1,1)))

plot(model2.ridge)
title("Ridge Model")

plot(model2.enet)
title("Elastic Net Model")

plot(model2.lasso)
title("LASSO Model")
```



Los patrones en las gráficas son muy parecidos a los del primer modelo.

```
## Ridge:
## Lambda =  0.7280293
## MSE =  23.23302
## Elastic Net:
## Lambda =  0.005878134
## MSE =  18.39472
## LASSO:
## Lambda =  0.01568492
## MSE =  18.41237
```

Como vemos, hemos conseguido bajar el MSE en este modelo respecto al anterior modelo, especialmente en Elastic Net y LASSO. Que en LASSO y en Elastic Net tengamos menos MSE que en Ridge nos lleva a pensar que hay atributos que se podrían eliminar. En el primer modelo la diferencia no era tan marcada, por lo que podemos pensar que los atributos que sobran pertenecen a los cuadrados de los atributos originales.

Puesto que en ambos modelos como mejores resultados hemos obtenido ha sido con LASSO, lo usaremos para estimar el modelo final.

Por tanto, ya tenemos los hiperparámetros necesarios para estimar un modelo con todo el train. Estos hiperparámetros son $\lambda = 0.008975485,\ \alpha = 1$.

Sin embargo, aun habiendo encontrado el mejor lambda, no lo vamos a usar. Esto es por que la función glmnet busca automáticamente el mejor lambda, por lo que le pasaremos como argumentoun número de lambdas para que la propia función eliga la que mejor vaya.

Vamos a ello.

```
model.final = glmnet(x = xtrain,y = ytrain,alpha = 1,nlambda = 150)
```

Ya tenemos nuestro modelo estimado. Vamos a ver algunas características del modelo:

```
bestLambda = model.final$lambda[which.max(model.final$dev.ratio)]
ytrain.predicted = predict(model.final,newx = xtrain,s = bestLambda)

# MSE y RMSE
mse = sum((ytrain.predicted - c(ytrain))^2)/length(ytrain)
rmse = sqrt(mse)

# R squared
rsq <- 1 - sum((ytrain.predicted - c(ytrain))^2) / sum(ytrain^2)

cat("MSE: ",mse)
```

```
## MSE:  15.38673
```

```
cat("RMSE: ",rmse)
```

```
## RMSE:  3.922592
```

```
cat("RSQ: " ,rsq)
```

```
## RSQ:  0.9743338
```

Vamos a interpretar estos datos.

Recordemos que el MSE es una variable que nos dice cómo de bien hemos ajustado el modelo. qué tan cerca está una línea ajustada de los puntos de datos. Para cada punto de datos, toma la distancia vertical desde el punto hasta el correspondiente valor y en el ajuste de la curva (el error) y cuadra el valor. Cuanto menor es el error cuadrático medio, más se acerca el ajuste a la función real.

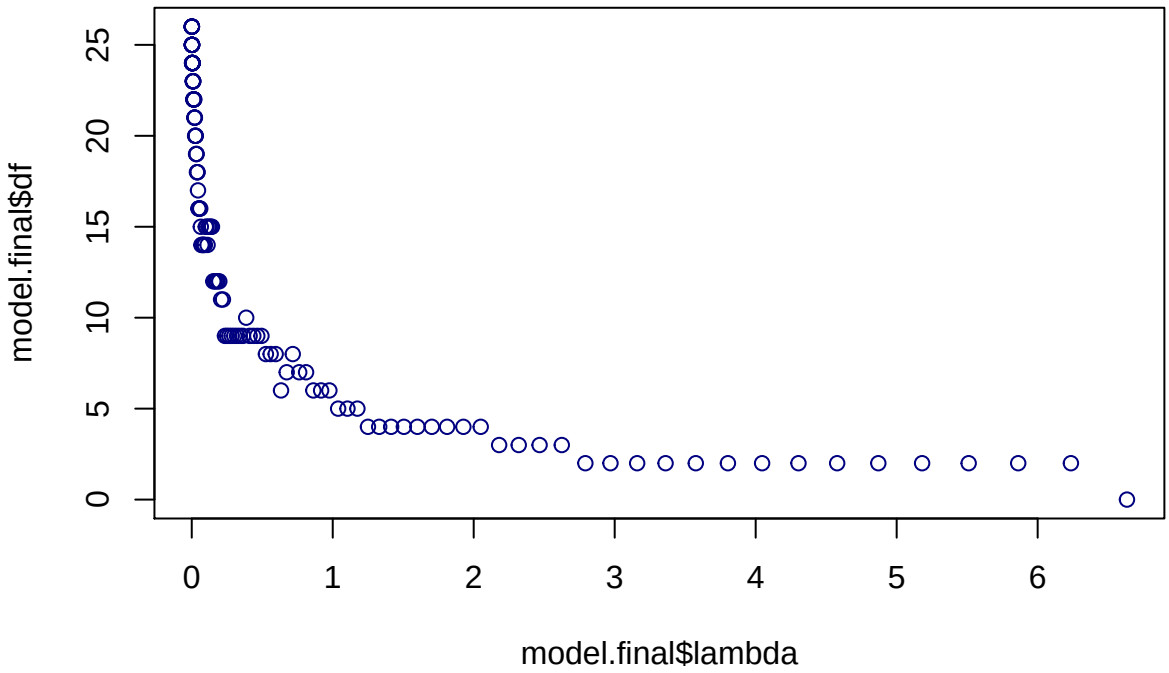
El RMSE es simplemente la raíz cuadrada del MSE. Lo pongo también por que es muy intuitivo y fácil de interpretar.

La definición del RSQ o R-cuadrado es bastante directa; es el porcentaje de la variación de la variable de respuesta que se explica por un modelo lineal. Un 0% indica que el modelo no explica la variabilidad de los datos de respuesta en torno a su media, mientras que el 100% indica que el modelo explica toda la variabilidad de los datos de respuesta en torno a su media.

En general, cuanto mayor sea el R-cuadrado, mejor será el modelo que se ajuste a sus datos.

Vamos a ver cómo se van eliminando atributos conforme lambda aumenta.

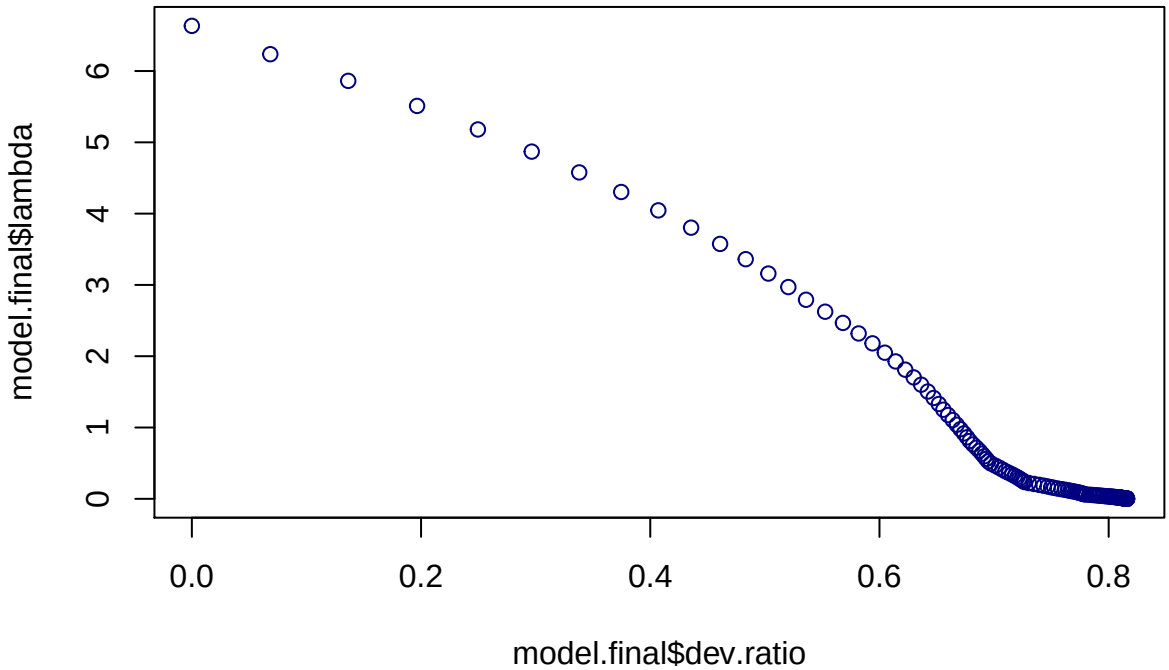
```
plot(x=model.final$lambda,y=model.final$df,col="navy blue")
```



En el gráfico podemos ver cómo se van aproximando a 0 los coeficientes (pesos) conforme lambda se hace más grande. Esto no es ninguna sorpresa, LASSO funciona de esta manera.

Vamos a ver ahora la correlación entre el valor de lambda y la desviación obtenida. Recordemos que la desviación es una variable estadística que refleja la bondad de ajuste de un modelo. Cuanto más cercano a 0, peor ajusta el modelo, y cuanto más cercano a 1, mejor ajusta.

```
plot(y = model.final$lambda,x=model.final$dev.ratio,col="navy blue")
```



En este gráfico se ve claramente que a menor lambda, mejores desviaciones obtenemos. También vemos que cuanto más pequeña es lambda menos diferencia hay entre desviaciones.

Test

Ya tenemos nuestro modelo, y parece que da resultados decentes. Vamos a ponerlo a prueba en el test.

Primero debemos añadirle a los datos de test sus cuadrados.

```
squaredxtest = xtest
for( i in 1:nrow(xtest) )
{
  for( j in 1:ncol(xtest) )
  {
    if( j != 4 ) squaredxtest[i,j] = squaredxtest[i,j]*squaredxtest[i,j]
  }
}



xtest = cbind(xtest,squaredxtest)
```

Ahora vamos a predecir y a obtener el error.

```
ytest.predicted = predict(model.final,newx = xtest,s = bestLambda)

# MSE y RMSE
mse = sum((ytest.predicted - c(ytest))^2)/length(ytest)
rmse = sqrt(mse)

# R squared
rsq <- 1 - sum((ytest.predicted - c(ytest))^2) / sum(ytest^2)

cat("MSE: ",mse)
```

```
## MSE:  10.66029
```

```
cat("RMSE: ",rmse)
```

```
## RMSE:  3.265009
```

```
cat("RSQ: " ,rsq)
```

```
## RSQ:  0.9810667
```

Como vemos, hemos obtenido incluso mejores resultados que en el train, por lo que podemos concluir que hemos conseguido un modelo que predice con muy buenos resultados.

Random Forests

Introducción

Un Random Forest consiste en una colección o conjunto de Decission Trees simples, cada uno capaz de producir una respuesta cuando se presenta con un conjunto de valores predictivos. Para los problemas de regresión, la respuesta del árbol es una estimación de la variable dependiente dados los predictores. He elegido Random Forest para este problema por que es un algoritmo muy fácil de implementar y que suele obtener muy buenos resultados, tanto en clasificación como en regresión.

Una de las ventajas de los Random Forests es que que la mayoría de hiperparámetros ya están preasignados por convenio, por lo que sólo nos tenemos que preocupar por el número de árboles a usar.

Existe el hiperparámetro m (número de variables para las ramificaciones), y aunque se puede asignar manualmente, ya está preasignada por defecto: Raíz de p para clasificación y p/3 para regresión. Están así asignadas por que empíricamente es como mejores resultados han dado, por lo tanto lo mejor es que dejemos ese hiperparámetro por defecto.

No vamos a obtener sobreajuste por usar un número alto de árboles, pero hay que tener en cuenta que cuanto mayor número de árboles, más costoso computacionalmente va a ser entrenar y predecir. Por lo tanto, vamos a buscar como hiperparámetro el número de árboles a usar para entrenar, de forma que haya un equilibrio entre buenos resultados y poco coste computacional.

Entrenamiento

Para encontrar el mejor valor para el hiperparámetro de número de árboles, vamos a entrenar con distintos valores. Vamos a entrenar con entre 10 y 300 árboles, avanzando de 10 en 10.

No nos debemos preocupar por hacer Cross-Validation, el paquete randomForest tiene una función específica que nos ahorra hacerlo manualmente.

Como los Random Forest dependen en cierta medida de la aleatoriedad, para cada valor del hiperparámetro del número de árboles, entrenaremos 3 veces y obtendremos la media de su MSE.

Primero vamos a obtener los datos de train y test.

```
# Datos de entrenamiento
xtrain = data.matrix(train[,1:13])
ytrain = data.matrix(train[,14])

# Datos de test
xtest = data.matrix(test[,1:13])
ytest = data.matrix(test[,14])
```

Ahora, vamos a entrenar para buscar el mejor valor para el número de árboles.

```
i = 10
while( i <= 300 ) # Número de árboles
{

  for( j in 1:5) # Media de 5 entrenamientos
  {

    # Entrenamos nuestro random forest con Cross-Validation
    rf = rfcv(xtrain,ytrain,ntree=i)

    # Guardamos los MSE
    if( j == 1 ) rf.mse = rf$error.cv
    else rf.mse = rbind(rf.mse,rf$error.cv)
  }

  # Calculamos la media de los 3 entrenamientos para cada i, y la guardamos
  if( i == 10 ){
    media.rf.mse = c(mean(rf.mse[,1]),mean(rf.mse[,2]),mean(rf.mse[,3]),mean(rf.mse[,4]))
  }
  else media.rf.mse = rbind(media.rf.mse,c(mean(rf.mse[,1]),mean(rf.mse[,2]),mean(rf.mse[,3]),mean(rf.ms

  i = i + 10
```

```
}
```

Vamos a obtener el número de árboles ideal:

```
ntrees = which.min(media.rf.mse)
while( ntrees > 30 ) ntrees = ntrees-30
ntrees = ntrees*10
ntrees
```

```
## [1] 170
```

Ya tenemos el hiperparámetro listo, solo queda entrenar el modelo final.

```
randomForest = randomForest(x = xtrain,y = c(ytrain),ntree = ntrees)
randomForest
```

```
##
## Call:
##  randomForest(x = xtrain, y = c(ytrain), ntree = ntrees)
##                Type of random forest: regression
##                      Number of trees: 170
## No. of variables tried at each split: 4
##
##           Mean of squared residuals: 11.01391
##                     % Var explained: 86.83
```

Como podemos observar, hemos obtenido un MSE de bastante bueno. Vamos a probar nuestro modelo en el test.

Test

```
ytest.predicted = predict(randomForest,newdata = xtest)

# MSE y RMSE
mse = sum((ytest.predicted - c(ytest))^2)/length(ytest)
rmse = sqrt(mse)

# R squared
rsq <- 1 - sum((ytest.predicted - c(ytest))^2) / sum(ytest^2)

cat("MSE: ",mse)
```

```
## MSE:  6.985196
```

```
cat("RMSE: ",rmse)
```

```
## RMSE:  2.642952
```

```
cat("RSQ: " ,rsq)
```

```
## RSQ:  0.9875939
```

Podemos observar que hemos obtenido unos resultados bastante decentes, mejores que con el modelo de Regresión Lineal que hicimos antes, en muchas menos líneas de código y sólo con un hiperparámetro que ajustar.