Proyecto Aprendizaje Automático: Airfoil Self-Noise and Prediction

José Carlos Martínez Velázquez
Benjamín Vega Herrera

20 de Junio de 2016

1.- Definición del problema a resolver y enfoque elegido

El problema seleccionado es Airfoil Self-Noise and Prediction. Fue planteado originalmente en la NASA, en el centro de investigación Langley, y por Thomas F. Brooks, D. Stuart Pope y Michael A. Marcolini en el año 1989. Lo que se pretende resolver es, dados unos ejemplos medidos realmente en el tunel de viento, predecir el ruido (en decibelios) producido por la interacción de un ala de avión y las turbulencias a su alrededor en función a su perfil aerodinámico, descrito por las siguientes características:

- Frecuencia, en Hertzios
- Ángulo de ataque, en grados
- Profundidad del ala (chord lenght), en metros
- Velocidad máxima libre de turbulencias (free-stream velocity), en metros por segundo
- Desplazamiento lateral debido a la succión y grosor del ala, en metros.

En cuanto al enfoque elegido, el problema fue enunciado originalmente para regresión. Trataremos regresión en el primer apartado, el ajuste de un modelo lineal. En el segundo apartado estableceremos una comparativa entre el modelo lineal ajustado en el primer apartado y un k-nn para regresión.

2.- Codificación de los datos de entrada para hacerlos útiles a los algoritmos.

Los datos están medidos en sus respectivos rangos y todas las variables son cuantitativas (numéricas). No es necesario ningún trato especial en principio para que los algoritmos puedan trabajar con los datos.

3.- Valoración del interés de la variables para el problema y selección de un subconjunto (en su caso).

Dado que el número de variables predictoras que se tratan es reducido, se tendrán todas en cuenta. En las transformaciones polinómicas no ha quedado mas remedio que emplear la técnica de ensayo y error.

4.- Normalización de las variables (en su caso)

En general no es necesario la normalización de las variables, excepto en el caso de k-NN, donde normalizaremos las variables predictoras para calcular las distancias entre los ejemplos.

5.- Selección de las técnicas (paramétrica) y valoración de la idoneidad de la misma frente a otras alternativas

Para empezar hemos elegido un modelo lineal, en concreto regresión lineal con weight decay (regularizada). Nos parece idónea para empezar por la simplicidad que aporta un modelo lineal, aplicando la filosofía de la navaja de Ockam. Optimizaremos, dentro de nuestras posibilidades, para llevar el modelo lineal al error más pequeño fuera de la muestra.

Ampliaremos el estudio con otras técnicas más potentes en su versión de regresión: Support Vector Machine y Bagging, que nos mostrarán que, en ocasiones, la complejidad que conlleva ajustar un modelo de este tipo está totalmente justificada en base al error de test.

6.- Aplicación de la técnica especificando claramente que algoritmos se usan en la estimación de los parámetros, los hiperparámetros y el error de generalización.

En la mayoría de los casos hemos utilizado validación cruzada de 5 o 10 particiones para obtener los mejores valores de los parámetros e hiperparámetros, así como para estimar el error fuera de la muestra. Teóricamente, los valores correctos de las particiones están entre 5 y 10. Hay resultados empíricos que nos dicen que dado un valor de error en validación cruzada (con una parte de los datos actuando como test en cada iteración), el modelo entrenado con la totalidad de los datos tendrá una cota de error inferior a dicho valor. En definitiva, si entrenamos con todos los datos disponibles, el error en test de este nuevo modelo será inferior que la media de los errores de test cometidos en cada iteración de validación cruzada.

Hemos intentado emplear, en regresión lineal, la técnica Leave One Out (LOO), pero debido a su complejidad computacional sólo lo hemos realizado en una ocasión, obteniendo los mismos resultados que con validación cruzada de 5 particiones.

7.- Argumentar sobre la idoneidad de la función regularización usada (en su caso)

Hemos decidido usar regresión lineal regularizada. Esta técnica incluye una función de regularización que depende de un término λ (lambda), el coeficiente de regularización y un α (alpha), el término de penalización. En el estudio veremos cómo funciona internamente.

Veamos como funciona la técnica. Según la teoría, el algoritmo de regresión lineal consta de tres pasos. El primero consiste en representar los descriptores como una matriz donde los de cada ejemplo sean una fila: `X`. Del mismo modo, los valores de salida de los ejemplos deben estar en una matriz donde la fila i es el valor de la variable de salida del ejemplo i de la matriz `X`, esta es la matriz `Y`. El segundo paso es computar la pseudoinversa de la matriz `X` que se calcula como $X^\dagger = (X^T X)^{-1} X^T$. Dada su descomopsición en valores singulares $X = U\Sigma V^T$, podemos aprovechar sus propiedades y tenemos que: $X^\dagger = V\Sigma^{-1}U$. El tercer y último paso es, calculada ya la pseudoinversa, devolver los coeficientes obtenidos $coefs = X^\dagger Y$. Teóricamente, la función definida por dichos coeficientes es la que minimiza los errores, es decir, la suma de la desviación de cada ejemplo con respecto al valor de la función, o lo que es lo mismo, el valor predicho.

Para evitar sobreajustar, añadimos un término lambda, que va a añadir diversas restricciones a los pesos. Dependiendo del valor de λ, estas restricciones serán más fuertes o menos. Si antes los coeficientes de la función predictora de regresión lineal se calculaba como $(X^T X)^{-1} X^T Y$, añadir weight decay no es más que calcular dicha función como $(X^T X + \lambda I)^{-1} X^T Y$, donde λ es el parámetro de regularización. Evidentemente, cuanto más pequeño es λ las restricciones serán más laxas. El parámetro λ ha sido calculado con validación cruzada.

Sabiendo ya cómo funciona regresión lineal con regularización, vamos a poner en juego el término de penalización. Dicho término permite quitar peso a las variables que hablan poco acerca de la salida. Aunque todas las variables tengan relación con ella, es razonable pensar que unas variables predictoras nos dirán más que otras en ese sentido. En regresión lineal regularizada tenemos tres formas de penalizar variables:

- R-LASSO ($\alpha = 1$)
- R-RIDGE ($\alpha = 0$)
- Penalización elástica ($0 \leq \alpha \leq 1$)

Pero ¿qué hace α realmente?

Las expresiones de ambos tipos de regresión se pueden deducir de forma sencilla. Partimos de la expresión común a la regresión.

$$\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda P_\alpha$$

Donde $P_\alpha = \sum_{j=1}^{p} \left((1-\alpha)\beta_j^2 + \alpha\,|\beta_j|\right)$ es el término de penalización. Cuando $\alpha = 0$, estamos ante R-RIDGE, y cuando $\alpha = 1$ estamos ante R-LASSO. Entonces, la expresión de R-LASSO queda así:

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda\sum_{j=1}^{p}|\beta_j|$$

La parte multiplicada por lambda en R-LASSO se llama norma ℓ_1, que es su forma de penalizar.

Lo que se pretende es:

$$minimizar_\beta\left\{\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2\right\} \quad sujeto\ a\ \sum_{j=1}^{p}|\beta_j| \leq s$$

Donde s es una constante que indica lo restrictivos que somos para obtener los coeficientes. Si s es muy grande, apenas seremos restrictivos, por contra, si s es pequeña, la permisividad será mínima.

Del mismo modo, podemos ver que R-RIDGE ($\alpha = 0$), quedaría como

$$minimizar_\beta\left\{\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2\right\} \quad sujeto\ a\ \sum_{j=1}^{p}\beta_j^2 \leq s$$

La forma de penalizar de R-RIDGE es denominada norma ℓ_2.

Estudiaremos de forma más pormenorizada como elegir entre los distintos tipos de penalización.

8.- Valoración de los resultados (gráficas, métricas de error, análisis de residuos, etc)

Todos los errores calculados están en la métrica de error cuadrático, $EC = (y - \hat{y})^2$. Adicionalmente, se ha calculado un error en porcentaje calculado como $sqrtEC/(max_{rango} - min_{rango})$, donde rango es los valores entre los que oscila la variable de salida.

- Regresión lineal regularizada.

El mejor modelo obtenido fue una transformación polinómica de grado 8, con una cota de error de mínimos cuadrados fuera de la muestra 21.05, obtenida por validación cruzada. El valor de los parámetros, también obtenidos por validación cruzada son: $\alpha = 0.1$ y $\lambda = 0.002988585$, lo que representa una regresión lineal de tipo elástica, más próxima a RIDGE. Dado que el rango (según los datos) de la variable de salida oscila entre 103.380 y 140.987, podemos calcular el porcentaje de error cometido como:

$$100 * \frac{\sqrt{cota_{E_{out}}}}{max_{rango} - min_{rango}} = 100 * \frac{\sqrt{21.05}}{140.987 - 103.380} = 12.2\%$$

Los modelos con transformación cuadrática probados, con sus cotas de error son los siguientes:

Modelo	Error cuadrático cv	Error en porcentaje cv (%)
Sin transformaciones	22.97	12.74
$\phi(x_1^2)$	21.73	12.4
$\phi(x_2^2)$	23.02	12.75
$\phi(x_3^2)$	22.82	12.7

Modelo	Error cuadrático cv	Error en porcentaje cv (%)
$\phi(x_4^2)$	23.01	12.75
$\phi(x_5^2)$	22.76	12.68
$\phi(x_1^2, x_2^2)$	21.79	12.41
$\phi(x_1^2, x_3^2)$	21.59	12.35
$\phi(x_1^2, x_4^2)$	21.79	12.41
$\phi(x_1^2, x_5^2)$	21.6	12.35
$\phi(x_2^2, x_3^2)$	22.87	12.71
$\phi(x_2^2, x_4^2)$	23.05	12.76
$\phi(x_2^2, x_5^2)$	22.85	12.71
$\phi(x_3^2, x_4^2)$	22.86	12.71
$\phi(x_3^2, x_5^2)$	22.49	12.61
$\phi(x_4^2, x_5^2)$	28.23	14.13
$\phi(x_1^2, x_2^2, x_3^2)$	21.64	12.36
$\phi(x_1^2, x_2^2, x_4^2)$	21.83	12.7
$\phi(x_1^2, x_2^2, x_5^2)$	21.6	12.35
$\phi(x_1^2, x_3^2, x_4^2)$	21.65	12.37
$\phi(x_1^2, x_3^2, x_5^2)$	21.56	12.34
$\phi(x_1^2, x_4^2, x_5^2)$	21.6	12.35
$\phi(x_2^2, x_3^2, x_4^2)$	22.9	12.72
$\phi(x_2^2, x_4^2, x_5^2)$	22.58	12.63
$\phi(x_3^2, x_4^2, x_5^2)$	22.54	12.62
$\phi(x_1^2, x_2^2, x_3^2, x_4^2)$	21.69	12.38
$\phi(x_1^2, x_2^2, x_3^2, x_5^2)$	21.59	12.35
$\phi(x_1^2, x_2^2, x_4^2, x_5^2)$	21.64	12.37
$\phi(x_1^2, x_3^2, x_4^2, x_5^2)$	21.38	12.29
$\phi(x_2^2, x_3^2, x_4^2, x_5^2)$	22.64	12.65
$\phi(x_1^2, x_2^2, x_3^2, x_4^2, x_5^2)$	21.6	12.35

Nos quedamos con el mínimo error de validación cruzada, la transformación $\phi(x_1^2 x_3^2 x_5^2)$. A partir de ahí comenzamos a probar transformaciones de grado 3, pero nos dimos cuenta que no había correlación entre las transformaciónes de grado n y las de $n+1$. No podremos todas las transformaciones probadas por la explosión combinatoria que supondría, pero a base de ensayo y error, el mejor modelo que conseguimos fue la transformación polinómica:

$$\phi(x_1^2, x_2^2, x_3^2, x_4^2, x_5^2, x_1^3, x_2^3, x_3^3, x_4^3, x_5^3, x_1^4, x_2^4, x_3^4, x_4^4, x_5^4, x_1^5, x_2^5, x_3^5, x_1^6, x_2^6, x_3^6, x_1^7, x_2^7, x_3^7, x_1^8, x_2^8, x_3^8)$$

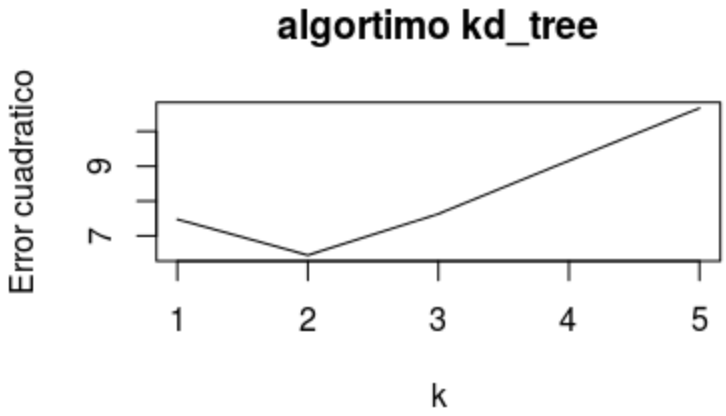
- K-NN:

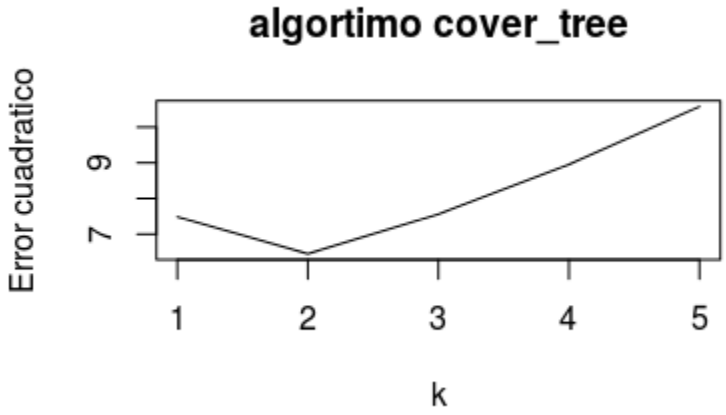
El método k-nn (K nearest neighbors Fix y Hodges, 1951) es un método de clasificación supervisada (Aprendizaje, estimación basada en un conjunto de entrenamiento y prototipos) que sirve para estimar la función de densidad $F(x/C_j)$ de las predictoras x por cada clase C_j.

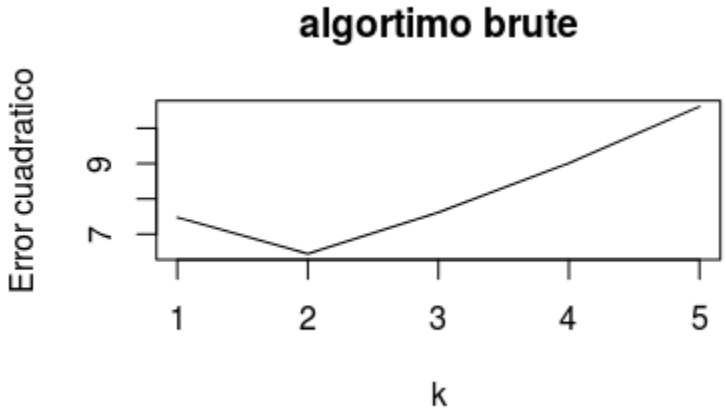
Pero nosotros como tenemos un problema de regresión vamos a utilizar un k-nn en regresión. Que no hace más que una media de los k vacinos más cercanos:

$$g(x) = \frac{1}{k} \sum_{i=1}^{k} y_{[i]}(x)$$

El mejor k lo obtenemos mediante validación cruzada y para nuestro problema tenemos que lo mejor son 2 vecinos (usando los tres algortimos de búsqueda que nos ofrece la libreria FNN, "kd_tree", "cover_tree" y "brute"), como podemos ver:







Como vemos, el error cuadrático más pequeño que se obtiene es 6.44, con k=2, lo que en porcentaje equivale a 6.75%.

También podemos indicarle al modelo knn el algortimo de búsqueda y tras validación cruzada obtenenmos que todos los algoritmos dan resultados parecidos en error cuadrático, 5.455 de "kd_tree" y "brute" frente a 6.448 de "cover_tree", es decir 6.75% frente a 6.751% de error en porcentaje.

kd_tree = KD-árboles son estructuras de datos que se utilizan para almacenar los puntos en el espacio k-dimensional. En las hojas del árbol se almacena los puntos del conjunto de datos (uno o varios puntos en cada hoja). Cada punto se almacena en una y sólo una hoja. Los nodos del árbol corresponden a las divisiones del espacio. Cada división divide el espacio y el conjunto de datos en dos partes bien diferenciadas. Se hacen escisiones de subsecuencias desde el nodo raíz a una de las hojas y se eliminan partes del conjunto de datos (y espacio) hasta que sólo queda una pequeña parte del conjunto de datos (y espacio).

cover_tree = Un árbol de cubierta T en un conjunto de datos S es un árbol explanada donde cada nivel es una "cubierta" para el nivel por debajo de él. Cada nivel está indexada por una escala entero i, que disminuye a medida que se desciende del árbol.

Cover tree es $O(n)$ en espacio de estructura de datos que nos permite buscar en timepo $O(log(n))$ como kd_tree dado con una dimensionalidad intrinseca fija.

brute= busca en forma lineal.

Modelos ajustados adicionalmente

- Support Vector Machine

SVM se puede aplicar no sólo a problemas de clasificación , sino también para el caso de regresión, manteniendo todas las características principales que caracterizan a este algoritmo: La idea básica consiste en realizar un mapeo de los datos de entrenamiento xeX a un espacio de mayor dimensión F a través de un mapeo no lineal $\phi : X ß F$, donde podemos realizar una regresión lineal. Está basado en la definición de la función de pérdida que ignora los errores que están situados dentro de la distancia determinada del valor real. Este tipo de función se llama a menudo - función de pérdida - epsilon intensivo.

Mediante validación cruzada y con kernel lineal, ya que este es paramétrico obtenemos un error cuadrático de: 23.99.

Como podemos ver es más alto que el de Regresión Lineal ajustado todo lo que hemos podido: 23.99 a 21.05 de Regresión Lineal, con lo que el mejor modelo obtenido ha sido Regresiónn Lineal Regularizada.

Además como aporte adicional obtuvimos el mejor modelo para SVM mediante validación cruzada, que es con kernel radial y nos el error cuadrático obtenido es: 10.66.

Podemos observar que los errores obtenidos por los modelos lineales son mucho mayor que el de knn y SVM con kernel radial, lo que podemos empezar a sospechar que este problema no se podría ser resulto por un modelo lineal.

- Bagging

Bagging es un caso particular del algoritmo random forests. La técnica consiste básicamente el realizar un consenso de n árboles de regresión. Evidentemente, depende del número de árboles que se eligen, del número de variables predictoras que se eligen en cada árbol y de los ejemplos que se eligen para entrenar el árbol de decisión. Es importante saber que los ejemplos con los que se entrena cada árbol se eligen con resposición, lo que hace más que probable (si no seguro) que un ejemplo esté repetido más de una vez en el conjunto de entrenamiento de un determinado árbol. Lo bueno de Bagging es que los árboles no están podados, por tanto, cada árbol tiene una alta varianza pero bajo sesgo. El promedio de los árboles reduce la varianza y finalmente tendremos bajo sesgo y varianza que es lo que deseamos altamente cuando ajustamos un modelo.

La diferencia entre Bagging y random forests es el número de variables predictoras que se usan en cada árbol. Sea m el número de variables predictoras y p el número de variables predictoras que elige random forests, un random forest en regresión elige $p = m/3$, mientras que en clasificación lo mejor empíricamente es $p = sqrtm$.

Si hacemos que $p = m$, es decir, si cogemos todas las variables predictoras estamos ante Bagging. El único parámetro, entonces, a calcular como mejor es el número de árboles con el que ajustamos el modelo. Para estimar este parámetro hemos usado validación cruzada.

El mejor modelo conseguido ha sido un modelo con 500 árboles (de un máximo de 2000). A partir de 500, la mejora con más árboles es menor a 0.05. Este modelo explica un 93% del fenómeno. Por validación cruzada de 5 particiones hemos estimado la cota de error fuera de la muestra, que, en comparación con el resto de modelos es de un error cuadrático de 3.59. Calculamos el error en porcentaje:

$$100 * \frac{\sqrt{cota_{E_{out}}}}{max_{rango} - min_{rango}} = 100 * \frac{\sqrt{3.59}}{140.987 - 103.380} = 5.038236\%$$

9.- Justificar que se ha obtenido la mejor de las posibles soluciones con la técnica elegida y la muestra dada. Argumentar en términos de la dimensión VC del modelo, el error de generalización y las curvas de aprendizaje.

- Regresión lineal regularizada

Dado que regresión lineal es un modelo lineal, la dimensión de Vapnik & Chervonenkis es d+1, es decir, 7. Podemos acotar el error fuera de la muestra con la siguiente expresión:

$$E_{out}(h) \leq E_{in}(h) + \sqrt{\frac{8}{N} log\left(\frac{4((2N)^{d_{vc}} + 1)}{\delta}\right)}$$

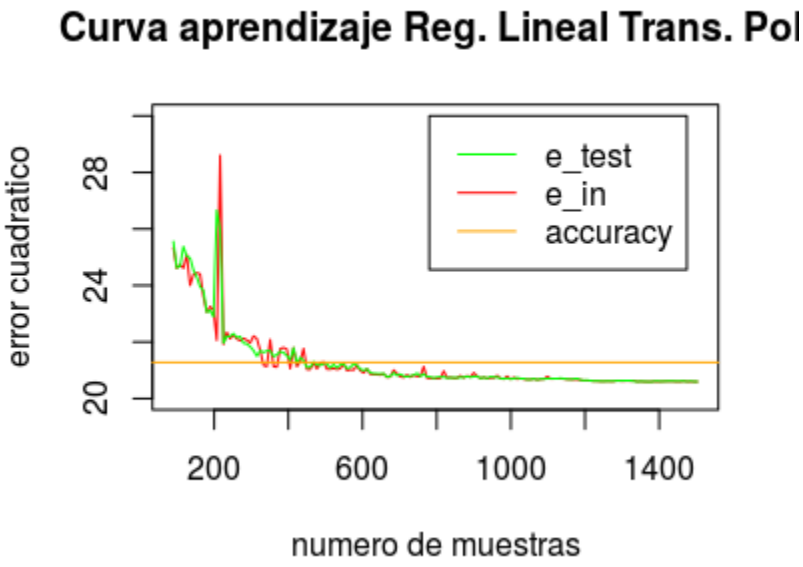
Previamente hemos medido $E_{in} = 20.59$. Para una confianza del 95% ($\delta = 0.05$) tenemos:

$$E_{out}(h) \leq 20.59 + \sqrt{\frac{8}{N} log\left(\frac{4((2 * (1503))^7 + 1)}{0.05}\right)} = 21.16$$

Al realizar las pruebas en test, la cota es razonable, pues el mejor modelo que conseguimos ajustar estaba por debajo de la cota.

Debemos saber que esta cota es muy laxa, es decir, apenas es restrictiva, por lo que, el hecho de que aunque esté por debajo de la máxima cota de error, esté tan cerca nos hizo pensar que podríamos estar ante un error muy elevado y comenzamos a sospechar que este problema podría no ser resuelto mediante un modelo lineal, razón por la cual ajustaremos modelos adicionales.

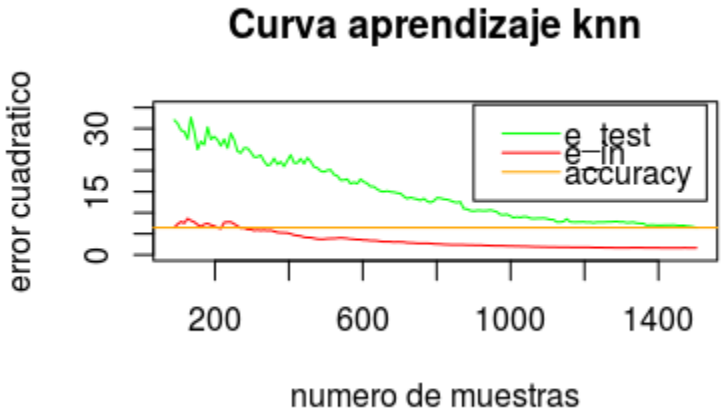
La curva de aprendizaje del mejor modelo es la siguiente:



- K-NN:

Solo sabemos que la dimensión de VC para knn si $k = 1$ es infinito. Nosotros vamos a usar el error obtenido por validación cruzada que es de 6.44, esta sería nuestra cota.

Veamos la curva de aprendizaje:



Como obtenemos que el mejor k es k=2 vecinos, confirmamos nuestras sospechas de que el problema no iba a ser resuelto mediante un modelo lineal pues que mejor k=2 significa que las particiones del espacio son muchas y, por supuesto, no podrían ser modeladas mediante una sola línea recta.

- Support Vector Machine: La dimesión VC para conjuntos orientados a hiperplanos en R^n es n+1. Entonces obtenemos una cota de generalización mediante la siguiente fórmula:

$$E_{out}(h) \leq E_{in}(h) + \sqrt{\frac{8}{N} log\left(\frac{4((2N)^{d_{vc}}+1)}{\delta}\right)}$$

Previamente hemos medido $E_{in} = 20.78$. Para una confianza del 95% ($\delta = 0.05$) tenemos:

$$E_{out}(h) \leq 23.78 + \sqrt{\frac{8}{1503} log\left(\frac{4((2*(1503))^7+1)}{0.05}\right)} = 24.15$$

- Bagging: Dado que bagging se basa en árboles y dado que sabemos que la dimension VC de un árbol es infinita, también sabemos que la dimensión VC de bagging es infinita. Dado que la dimensión de VC aparece en un exponente, la cota de error va a tender a infinito y será imposible acotar el error fuera de la muestra.

10. Estudios y código. ¿Como hemos hecho ajustado los modelos?

APARTADO 1: Solo es necesario aplicar un modelo de tipo parámetrico.

Antes de conocer los datos debemos fijar la clase de funciones y el algoritmo de aprendizaje que queremos utilizar. De este modo, los datos no van a limitar nuestro comportamiento a la hora de actuar. Vamos a ajustar un modelo lineal, para empezar por su simplicidad, y se irá optimizando para conseguir exprimir sus cualidades en la medida de lo posible. El modelo a ajustar será una regresión lineal 'weight decay' o regularizada. Como ya sabemos, deberemos estimar los coeficientes de regularización (λ) y de penalización (α).

Sabiendo ya el modelo que ajustaremos, carguemos los datos. El dataset que tratamos, por fortuna, estaba limpio y sin valores perdidos, y, como se trata de un problema de regresión, no es necesario plantearse el balanceo de carga, es decir, que haya más ejemplos de una clase que de otra. Entonces, leemos los datos una vez nos hemos descargado el fichero del repositorio de Machine Learning. Una vez leidos, barajamos aleatoriamente el los datos, por si viniera ordenado de alguna forma. (En todos los procesos en los que interviene la aleatoriedad se ha fijado una semilla para poder repetir los experimentos).

```
#LECTURA
datos <- read.delim("./airfoil_self_noise.dat",
                    header=FALSE)
colnames(datos)<-c("freq","aattack","clength","fsvel","ssdisp","sound")

#BARAJAR DATOS
set.seed(199)
orden<-runif(nrow(datos))
datos<-datos[order(orden),]
```

Una vez los datos en nuestro poder para ser tratados, vamos a realizar un primer acercamiento a regresión lineal. Entrenaremos un modelo simple y calcularemos una estimación del error fuera de la muestra mediante validación cruzada.

```
set.seed(199)
nparticiones<-5
parts <- cut(seq(1,nrow(datos)),breaks=nparticiones,labels=FALSE)

errors_cv_rl<-numeric(nparticiones)
for (i in 1:nparticiones){
  testIndexes <- which(parts==i,arr.ind=TRUE)
  trainIndexes<- setdiff(as.numeric(rownames(datos)),testIndexes)

  conj_test_cv<-datos[testIndexes,]
  conj_training_cv<-datos[trainIndexes,]


  mod_lineal <- lm(conj_training_cv$sound~.,data=conj_training_cv)
  pred_lin<-predict(mod_lineal,newdata=conj_test_cv[,-ncol(conj_test_cv)])
  error_lin<-mean((conj_test_cv[,ncol(conj_test_cv)]-pred_lin)*
                    (conj_test_cv[,ncol(conj_test_cv)]-pred_lin))

  errors_cv_rl[i]<-error_lin
}
print(paste("Error en test validacion cruzada 5cv:",mean(errors_cv_rl)))
```

```
## [1] "Error en test validacion cruzada 5cv: 23.2301737515847"
```

Este primer acercamiento nos deja un error cuadrático de 23.23, lo que supone un 12.81% de error.

Habiendo hecho regresión lineal simple, vamos a intentar rebajar el error mediante una regularización más seria. Vamos a ajustar un modelo de regresión de tipo LASSO ($alpha = 1$) y a calcular su error de validación cruzada. Para ello, en R, utilizaremos el paquete `glmnet`.

```
library(glmnet)
```

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

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

```
## Loaded glmnet 2.0-5
```

La técnica en este momento es elegir el mejor término de regularización (λ) por validación cruzada particiones y aplicarlo a cada iteración.

```
set.seed(199)
x <- as.matrix(datos[,-ncol(datos)])
y <- as.matrix(datos[,ncol(datos)])

#Elegimos el mejor lambda por validación cruzada de 10 particiones
cv<-cv.glmnet(x,y,alpha=1,nfolds=10)
best_lambda<-cv$lambda.min
print(paste("El mejor lambda para regresion LASSO es: ",best_lambda))
```

```
## [1] "El mejor lambda para regresion LASSO es:  0.0122191714151877"
```

```
nparticiones<-5
parts <- cut(seq(1,nrow(datos)),breaks=nparticiones,labels=FALSE)
errors_cv_rl<-numeric(nparticiones)

for (i in 1:nparticiones){
  testIndexes <- which(parts==i,arr.ind=TRUE)
  trainIndexes<- setdiff(as.numeric(rownames(datos)),testIndexes)

  conj_test_cv<-datos[testIndexes,]
  conj_training_cv<-datos[trainIndexes,]

  x <- as.matrix(conj_training_cv[,-ncol(conj_training_cv)])
  y <- as.matrix(conj_training_cv[,ncol(conj_training_cv)])

  mdl_lasso <- glmnet(x,y,lambda=best_lambda , family="gaussian",alpha=1)
  pred_lasso<-predict(mdl_lasso,
                      as.matrix(conj_test_cv[,-ncol(conj_test_cv)]),s=best_lambda)
  error_lasso<-mean(
    (pred_lasso - conj_test_cv[,ncol(conj_test_cv)])*
    (pred_lasso - conj_test_cv[,ncol(conj_test_cv)]) )

  errors_cv_rl[i]<-error_lasso
}
print(paste("Error en test validacion cruzada 5cv LASSO:",mean(errors_cv_rl)))
```

```
## [1] "Error en test validacion cruzada 5cv LASSO: 23.2314472599444"
```

El error cometido para un modelo de regresión lineal con penalización LASSO es de 23.23, lo que corresponde (calculado igual que en las ocasiones anteriores) a un 12.81% de error.

Vamos ahora, del mismo modo, a ajustar un modelo de regresión lineal con penalización RIDGE. En este caso, el valor del coeficiente de penalización será $\alpha = 0$ y el valor del coeficiente de regularización λ será calculado igualmente mediante validación cruzada.

```
set.seed(199)
x <- as.matrix(datos[,-ncol(datos)])
y <- as.matrix(datos[,ncol(datos)])

#Elegimos el mejor lambda por validación cruzada de 10 particiones
cv<-cv.glmnet(x,y,alpha=0,nfolds=10)
best_lambda<-cv$lambda.min
print(paste("El mejor lambda para regresion RIDGE es: ",best_lambda))
```

```
## [1] "El mejor lambda para regresion RIDGE es:  0.295719621142634"
```

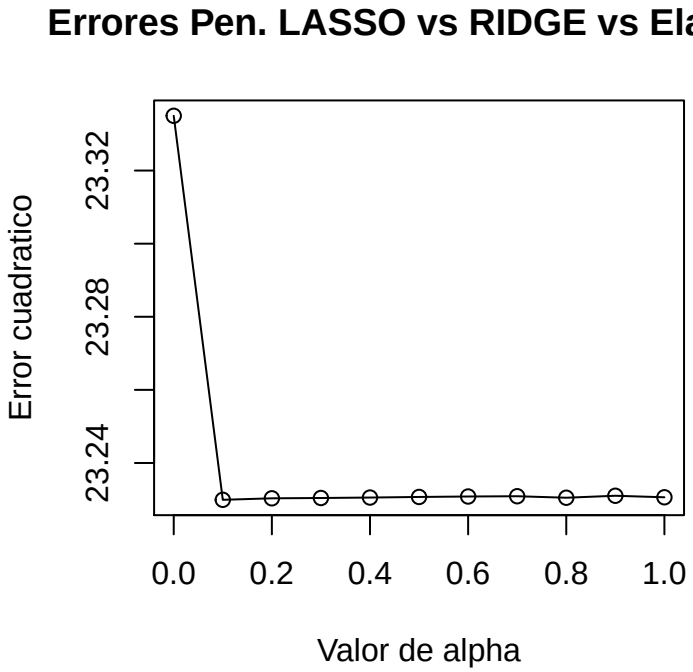
```
nparticiones<-5
parts <- cut(seq(1,nrow(datos)),breaks=nparticiones,labels=FALSE)
errors_cv_rl<-numeric(nparticiones)

for (i in 1:nparticiones){
  testIndexes <- which(parts==i,arr.ind=TRUE)
  trainIndexes<- setdiff(as.numeric(rownames(datos)),testIndexes)

  conj_test_cv<-datos[testIndexes,]
  conj_training_cv<-datos[trainIndexes,]

  x <- as.matrix(conj_training_cv[,-ncol(conj_training_cv)])
  y <- as.matrix(conj_training_cv[,ncol(conj_training_cv)])

  mdl_ridge <- glmnet(x,y,lambda=best_lambda , family="gaussian",alpha=0)
  pred_ridge<-predict(mdl_ridge,
                      as.matrix(conj_test_cv[,-ncol(conj_test_cv)]),s=best_lambda)
  error_ridge<-mean(
    (pred_ridge - conj_test_cv[,ncol(conj_test_cv)])*
    (pred_ridge - conj_test_cv[,ncol(conj_test_cv)]) )

  errors_cv_rl[i]<-error_ridge
}
print(paste("Error en test validacion cruzada 5cv RIDGE:",mean(errors_cv_rl)))
```

```
## [1] "Error en test validacion cruzada 5cv RIDGE: 23.3373648756265"
```

El error de validación cruzada obtenido con el modelo de regresión lineal con penalización RIDGE es de 23.33, lo que corresponde (usando los cálculos anteriores) a un 12.84%.

Parece pues, que la regresión LASSO da mejor resultado que la regresión RIDGE. No obstannte, con ninguno de estos dos modelos hemos conseguido rebajar el error de validación cruzada de regresión lineal simple. La diferencia es que el modelo de regresión lineal regularizada todavía cuenta con un margen de mejora: penalización elástica (valor α entre 0 y 1). Vamos a verlo gráficamente, repitiendo estos experimentos y variando el valor de α desde 0 a 1 aumentando en 0.1.

```r
set.seed(199)

error_lasso_vector<-numeric(11)
for(j in 0:10){

  nparticiones<-5
  parts <- cut(seq(1,nrow(datos)),breaks=nparticiones,labels=FALSE)
  errors_cv_rl<-numeric(nparticiones)

  for (i in 1:nparticiones){
    testIndexes <- which(parts==i,arr.ind=TRUE)
    trainIndexes<- setdiff(as.numeric(rownames(datos)),testIndexes)

    conj_test_cv<-datos[testIndexes,]
    conj_training_cv<-datos[trainIndexes,]

    x <- as.matrix(conj_training_cv[,-ncol(conj_training_cv)])
    y <- as.matrix(conj_training_cv[,ncol(conj_training_cv)])

    cv<-cv.glmnet(x,y,alpha=j/10,nfolds=10)
    best_lambda<-cv$lambda.min

    mdl_lasso <- glmnet(x,y,lambda=best_lambda , family="gaussian",alpha=j/10)
    pred_lasso<-predict(mdl_lasso,as.matrix(conj_test_cv[,-ncol(conj_test_cv)]),
                        s=best_lambda)
    error_lasso<-mean(
      (pred_lasso - conj_test_cv[,ncol(conj_test_cv)])*
      (pred_lasso - conj_test_cv[,ncol(conj_test_cv)]) )

    errors_cv_rl[i]<-error_lasso
  }
  error_lasso_vector[j+1]<-mean(errors_cv_rl)
}
best_alpha_abs<-(which(error_lasso_vector==min(error_lasso_vector))-1)/10

xInd=seq(0,1,0.1)
plot(xInd,
     error_lasso_vector,
     xlab="Valor de alpha",
     ylab="Error cuadratico",
     main="Errores Pen. LASSO vs RIDGE vs Elastica",
     type="o")
```



Con este gráfico vemos que el mejor valor de α no es ni 0 ni 1, sino un modelo elástico que, ciertamente está más cercano a un modelo de regresión lineal con penalización RIDGE, ($\alpha = 0.1$). Conocidas λ y α, vamos a calcular el error de validación cruzada de un modelo de regresión con penalización elástica.

```
set.seed(199)
nparticiones<-5
parts <- cut(seq(1,nrow(datos)),breaks=nparticiones,labels=FALSE)
errors_cv_rl<-numeric(nparticiones)

for (i in 1:nparticiones){
  testIndexes <- which(parts==i,arr.ind=TRUE)
  trainIndexes<- setdiff(as.numeric(rownames(datos)),testIndexes)

  conj_test_cv<-datos[testIndexes,]
  conj_training_cv<-datos[trainIndexes,]

  x <- as.matrix(conj_training_cv[,-ncol(conj_training_cv)])
  y <- as.matrix(conj_training_cv[,ncol(conj_training_cv)])


  cv<-cv.glmnet(x,y,alpha=best_alpha_abs,nfolds=10)
  best_lambda<-cv$lambda.min
  mdl_lasso <- glmnet(x,y,lambda=best_lambda , family="gaussian",alpha=best_alpha_abs)
  pred_lasso<-predict(mdl_lasso,
                      as.matrix(conj_test_cv[,-ncol(conj_test_cv)]),s=best_lambda)
  error_lasso<-mean(
    (pred_lasso - conj_test_cv[,ncol(conj_test_cv)])*
    (pred_lasso - conj_test_cv[,ncol(conj_test_cv)]) )

  errors_cv_rl[i]<-error_lasso
}
print(paste("Error en test validacion cruzada 5cv Elastico:",mean(errors_cv_rl)))
```

```
## [1] "Error en test validacion cruzada 5cv Elastico: 23.2296113039682"
```

Hasta aquí es el mejor modelo que podemos obtener con las variables predictoras disponibles. Debemos plantearnos realizar transformaciones polinómicas que nos permitan llevar más allá el modelo de regresión. Una transformación polinómica consiste en realizar operaciones a columnas del dataset y añadirlas además de las variables predictoras. Debido a la explosión combinatoria que supone todas las pruebas realizadas, no es posible plasmar aquí todos los resultados, algunos de ellos (los de transformación cuadrática) se pueden ver en la tabla del apartado 8. Después de pruebas y pruebas, el mejor modelo que conseguimos es la transformación polinómica siguiente:

$$\phi(x_1^2, x_2^2, x_3^2, x_4^2, x_5^2, x_1^3, x_2^3, x_3^3, x_4^3, x_5^3, x_1^4, x_2^4, x_3^4, x_4^4, x_5^4, x_1^5, x_2^5, x_3^5, x_1^6, x_2^6, x_3^6, x_1^7, x_2^7, x_3^7, x_1^8, x_2^8, x_3^8)$$

Vamos a ver qué error obtenemos con ella.

```
set.seed(199)
nparticiones<-5
parts <- cut(seq(1,nrow(datos)),breaks=nparticiones,labels=FALSE)

errors_cv_rl<-numeric(nparticiones)

for (i in 1:nparticiones){
  testIndexes <- which(parts==i,arr.ind=TRUE)
  trainIndexes<- setdiff(as.numeric(rownames(datos)),testIndexes)

  conj_test_cv<-datos[testIndexes,]
  conj_training_cv<-datos[trainIndexes,]

  x <- cbind(
    as.matrix(conj_training_cv[,-ncol(conj_training_cv)]),
    as.matrix(conj_training_cv[,c(1,2,3,4,5)])**2,
    as.matrix(conj_training_cv[,c(1,2,3,4,5)])**3,
    as.matrix(conj_training_cv[,c(1,2,3,4,5)])**4,
    as.matrix(conj_training_cv[,c(1,3,5)])**5,
    as.matrix(conj_training_cv[,c(1,3,5)])**6,
    as.matrix(conj_training_cv[,c(1,3,5)])**7,
    as.matrix(conj_training_cv[,c(1,3,5)])**8
  )
  y <- as.matrix(conj_training_cv[,ncol(conj_training_cv)])


  cv<-cv.glmnet(x,y,alpha=best_alpha_abs,nfolds=10)
  best_lambda<-cv$lambda.min

  mdl_elastic <- glmnet(x,y,lambda=best_lambda , family="gaussian",alpha=best_alpha_abs)
  pred_elastic<-predict(mdl_elastic,
                        cbind(
                          as.matrix(conj_test_cv[,-ncol(conj_test_cv)]),
                          as.matrix(conj_test_cv[,c(1,2,3,4,5)])**2,
                          as.matrix(conj_test_cv[,c(1,2,3,4,5)])**3,
                          as.matrix(conj_test_cv[,c(1,2,3,4,5)])**4,
                          as.matrix(conj_test_cv[,c(1,3,5)])**5,
                          as.matrix(conj_test_cv[,c(1,3,5)])**6,
                          as.matrix(conj_test_cv[,c(1,3,5)])**7,
```

```
                        as.matrix(conj_test_cv[,c(1,3,5)])**8
                      )
                      ,s=best_lambda)
  error_elastic<-mean(
    (pred_elastic - conj_test_cv[,ncol(conj_test_cv)])*
    (pred_elastic - conj_test_cv[,ncol(conj_test_cv)]) )

  errors_cv_rl[i]<-error_elastic
}
best_error<-mean(errors_cv_rl)
print(paste("Error en test validacion cruzada 5cv Trans. Pol.:",best_error))
```

```
## [1] "Error en test validacion cruzada 5cv Trans. Pol.: 21.0468022062801"
```

El error obtenido con la transformación polinómica baja en unos dos puntos al error cometido sólo con las variables predictoras. El nuevo error es de 21.05, lo que corresponde a un 12.2% de error.

Vamos a ver si esto es razonable. Calculemos con este modelo el error dentro de la muestra. Con este valor, calcularemos una cota superior del error de generalización.

```
set.seed(199)
  x <- cbind(
    as.matrix(datos[,-ncol(datos)]),
    as.matrix(datos[,c(1,2,3,4,5)])**2,
    as.matrix(datos[,c(1,2,3,4,5)])**3,
    as.matrix(datos[,c(1,2,3,4,5)])**4,
    as.matrix(datos[,c(1,3,5)])**5,
    as.matrix(datos[,c(1,3,5)])**6,
    as.matrix(datos[,c(1,3,5)])**7,
    as.matrix(datos[,c(1,3,5)])**8
  )
  y <- as.matrix(datos[,ncol(datos)])


  cv<-cv.glmnet(x,y,alpha=best_alpha_abs,nfolds=10)
  best_lambda<-cv$lambda.min

  mdl_elastic <- glmnet(x,y,lambda=best_lambda , family="gaussian",alpha=best_alpha_abs)
  pred_elastic<-predict(mdl_elastic,
                      cbind(
                        as.matrix(datos[,-ncol(datos)]),
                        as.matrix(datos[,c(1,2,3,4,5)])**2,
                        as.matrix(datos[,c(1,2,3,4,5)])**3,
                        as.matrix(datos[,c(1,2,3,4,5)])**4,
                        as.matrix(datos[,c(1,3,5)])**5,
                        as.matrix(datos[,c(1,3,5)])**6,
                        as.matrix(datos[,c(1,3,5)])**7,
                        as.matrix(datos[,c(1,3,5)])**8
                      )
                      ,s=best_lambda)
  error_in_elastic<-mean(
    (pred_elastic - datos[,ncol(datos)])*
    (pred_elastic - datos[,ncol(datos)]) )
```

```r
print(paste("Error dentro de la muestra RL Elastica Trans.Pol.: ",error_in_elastic))
```

```
## [1] "Error dentro de la muestra RL Elastica Trans.Pol.:  20.5948301638544"
```

Con el el error dentro de la muestra, vamos a calcular una cota superior del error fuera de la muestra con la expresión:

$$E_{out}(h) \leq E_{in}(h) + \sqrt{\frac{8}{N} log\left(\frac{4((2N)^{d_{vc}}+1)}{\delta}\right)} \Rightarrow$$

En un modelo lineal, la dimensión de Vapnik & Chervonenkis es del número de columnas más 1. En este caso, 7. El número de muestras es de 1503 y queremos una confianza del 95%, es decir ($\delta = 0.05$). Entonces, tenemos:

$$\Rightarrow \; E_{out}(h) \leq 20.59 + \sqrt{\frac{8}{N} log\left(\frac{4((2*(1503))^7+1)}{0.05}\right)} = 21.16$$

Nuestra error cuadrático de validación cruzada es de 21.05. Esto nos garantiza empíricamente (que no teóricamente) que si entrenamos con todos los datos, el modelo tendrá un error menor o igual a dicho valor. Pero cuidado, la cota que hemos calculado con el error en la muestra es apenas restrictiva, por lo que nos parece preocupante que esté tan cerca. Comenzamos a sospechar en base a esto que el problema de regresión podría no llegar a tener errores razonables con un modelo lineal. Es por ello que más adelante probaríamos los modelos Support Vector Machines y Bagging.

APARTADO 2: Implementar, valorar y comparar al menos dos modelos uno parmétrico y otra a elegir entre (k-NN, "Función de Base Radial").

Como en el apartado anterior hemos ajustado un modelo paramétrico lo que vamos hacer es utilizar el mejor modelo obtenido y compararlo con k-NN. Para ello vamos a ajustar un k-NN:

Necesitamos saber cual es el mejor k para nuestro modelo, además del mejor algoritmo de búsqueda, por lo que utilizaremos para ello validación cruzada:

Vamos a utilizar la libreria FNN para knn de regresión:

```r
library(FNN)
```

Tras esto vamos a normalizar los datos utilizando una función propia:

```r
normalizar = function(datosNormalizar) {
  normalizado = datosNormalizar
  for (i in seq(from=1,to=ncol(datosNormalizar)-1)) {
    minimo=min(datosNormalizar[,i])
    maximo=max(datosNormalizar[,i])
    for (j in seq(from=1,to=nrow(datosNormalizar))) {
      variable=datosNormalizar[j,i]
      normalizado[j,i] <- (variable-minimo)/(maximo-minimo)
    }
  }
  return (normalizado)
}

datos_normalizados = normalizar(datos)
```

Una vez tenemos los datos normalizados procedemos a calcular el mejor k y el algoritmo de búsqueda mejor mediante validación cruzada:

- Vamos a obtener el mejor k para el algoritmo de búsqueda kd_tree:

```
set.seed(199)
nparticiones<-5
parts <- cut(seq(1,nrow(datos)),breaks=nparticiones,labels=FALSE)

errors_knn<-numeric(5)
errors_knn_iter<-numeric(nparticiones)

for(i in 1:5){

  for (j in 1:nparticiones) {

    testIndexes <- which(parts==j,arr.ind=TRUE)
    trainIndexes<- setdiff(as.numeric(rownames(datos_normalizados)),testIndexes)

    conj_test_cv<-datos[testIndexes,]
    conj_training_cv<-datos[trainIndexes,]

    conj_test_cv=normalizar(conj_test_cv)
    conj_training_cv=normalizar(conj_training_cv)

    m.knn.reg<-knn.reg(train=conj_training_cv[,-ncol(conj_training_cv)],
                       test=conj_test_cv[,-ncol(conj_test_cv)],
                       y=conj_training_cv[,ncol(conj_training_cv)],
                       k=i,
                       algorithm="kd_tree")

    errors_knn_iter[j]<-mean((conj_test_cv[,ncol(conj_test_cv)]- m.knn.reg$pred)*(conj_test_cv[,ncol(co

  }
  errors_knn[i]=mean(errors_knn_iter)
}

best_k_kd_tree<-min(which(errors_knn==min(errors_knn)))
errors_knn_kd_tree=errors_knn[best_k_kd_tree]


raiz=sqrt(errors_knn[best_k_kd_tree])
rango=range(datos[,ncol(datos)])
porcentaje_kd_tree=raiz/(rango[2]-rango[1])

errores_kd_tree_plot = errors_knn
```

- Vamos a obtener el mejor k para el algoritmo de búsqueda cover_tree:

```
set.seed(199)
nparticiones<-5
parts <- cut(seq(1,nrow(datos)),breaks=nparticiones,labels=FALSE)
```

```
errors_knn<-numeric(5)
errors_knn_iter<-numeric(nparticiones)

for(i in 1:5){

  for (j in 1:nparticiones) {

    testIndexes <- which(parts==j,arr.ind=TRUE)
    trainIndexes<- setdiff(as.numeric(rownames(datos_normalizados)),testIndexes)

    conj_test_cv<-datos[testIndexes,]
    conj_training_cv<-datos[trainIndexes,]

    conj_test_cv=normalizar(conj_test_cv)
    conj_training_cv=normalizar(conj_training_cv)

    m.knn.reg<-knn.reg(train=conj_training_cv[,-ncol(conj_training_cv)],
                       test=conj_test_cv[,-ncol(conj_test_cv)],
                       y=conj_training_cv[,ncol(conj_training_cv)],
                       k=i,
                       algorithm="cover_tree")

    errors_knn_iter[j]<-mean((conj_test_cv[,ncol(conj_test_cv)]- m.knn.reg$pred)*(conj_test_cv[,ncol(co

  }
  errors_knn[i]=mean(errors_knn_iter)
}

best_k_cover_tree<-min(which(errors_knn==min(errors_knn)))
errors_knn_cover_tree=errors_knn[best_k_cover_tree]



raiz=sqrt(errors_knn[best_k_cover_tree])
rango=range(datos[,ncol(datos)])
porcentaje_cover_tree=raiz/(rango[2]-rango[1])

errores_cover_tree_plot = errors_knn
```

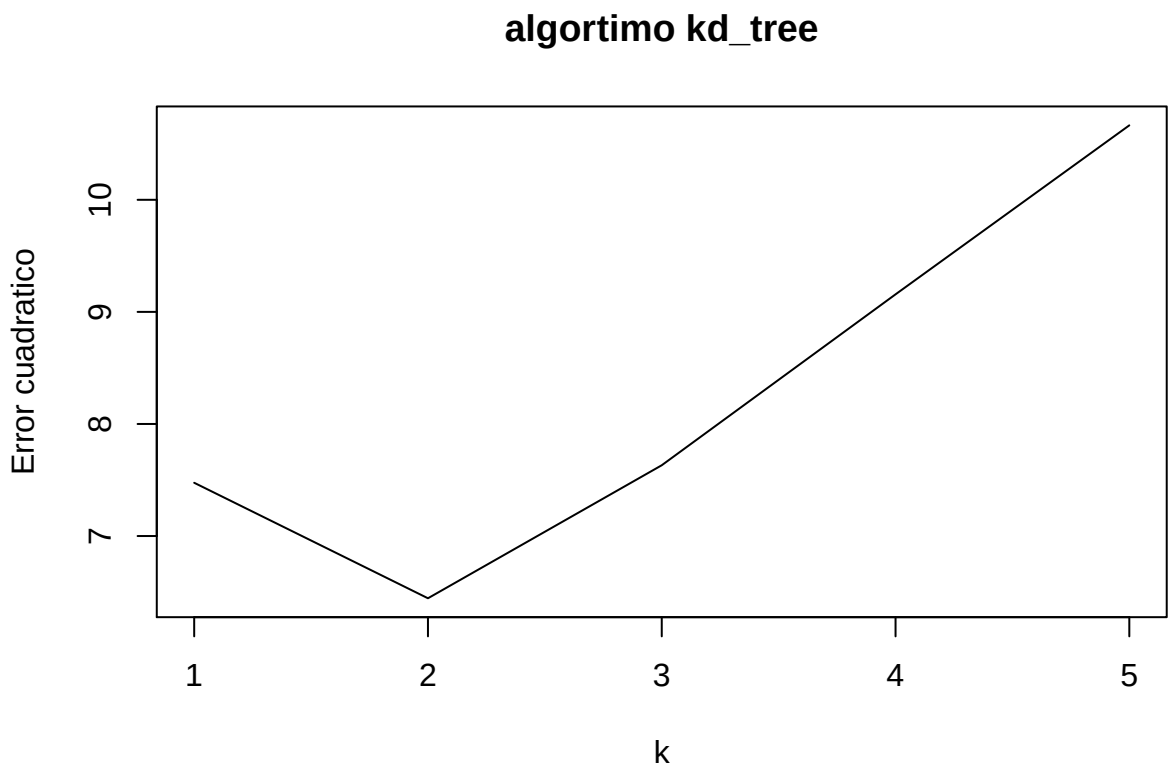
- Vamos a obtener el mejor k para el algoritmo de búsqueda brute:

```
set.seed(199)
nparticiones<-5
parts <- cut(seq(1,nrow(datos)),breaks=nparticiones,labels=FALSE)

errors_knn<-numeric(5)
errors_knn_iter<-numeric(nparticiones)

for(i in 1:5){

  for (j in 1:nparticiones) {

    testIndexes <- which(parts==j,arr.ind=TRUE)
```

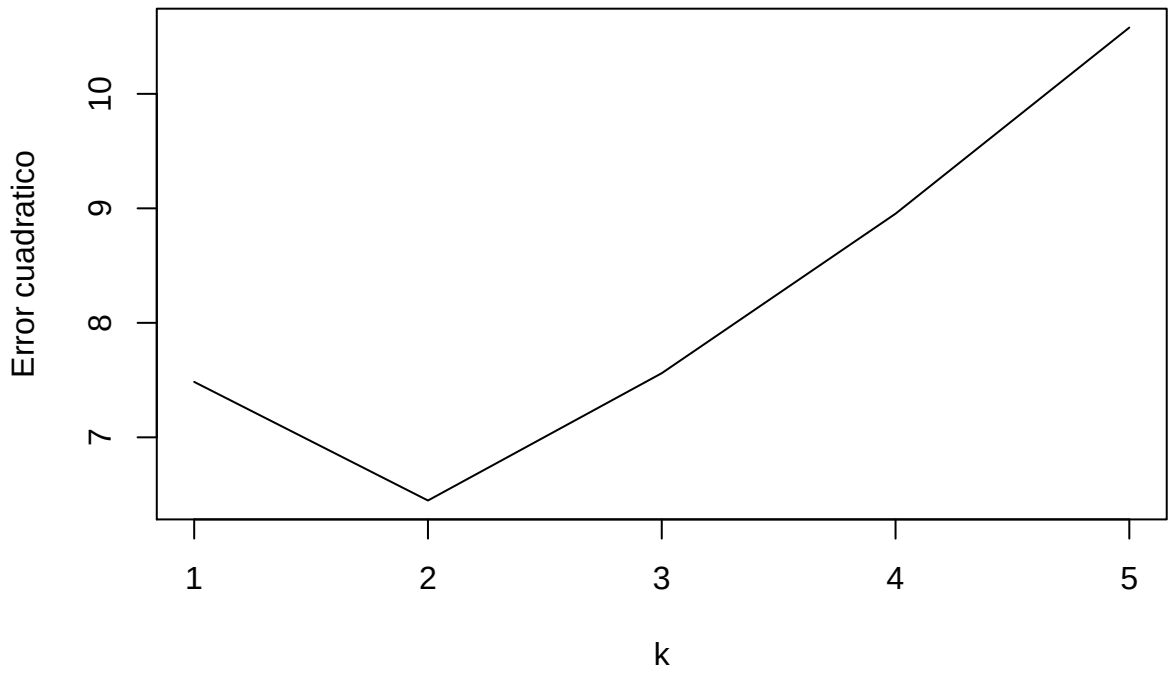
```
    trainIndexes<- setdiff(as.numeric(rownames(datos)),testIndexes)

    conj_test_cv<-datos[testIndexes,]
    conj_training_cv<-datos[trainIndexes,]
    conj_test_cv=normalizar(conj_test_cv)
    conj_training_cv=normalizar(conj_training_cv)

    m.knn.reg<-knn.reg(train=conj_training_cv[,-ncol(conj_training_cv)],
                       test=conj_test_cv[,-ncol(conj_test_cv)],
                       y=conj_training_cv[,ncol(conj_training_cv)],
                       k=i,
                       algorithm="brute")

    errors_knn_iter[j]<-mean((conj_test_cv[,ncol(conj_test_cv)]- m.knn.reg$pred)*(conj_test_cv[,ncol(co

  }
  errors_knn[i]=mean(errors_knn_iter)
}

best_k_brute<-min(which(errors_knn==min(errors_knn)))
errors_knn_brute=errors_knn[best_k_brute]


raiz=sqrt(errors_knn[best_k_brute])
rango=range(datos[,ncol(datos)])
porcentaje_brute=raiz/(rango[2]-rango[1])

errores_brute_plot = errors_knn
```

Vemos los resultados:

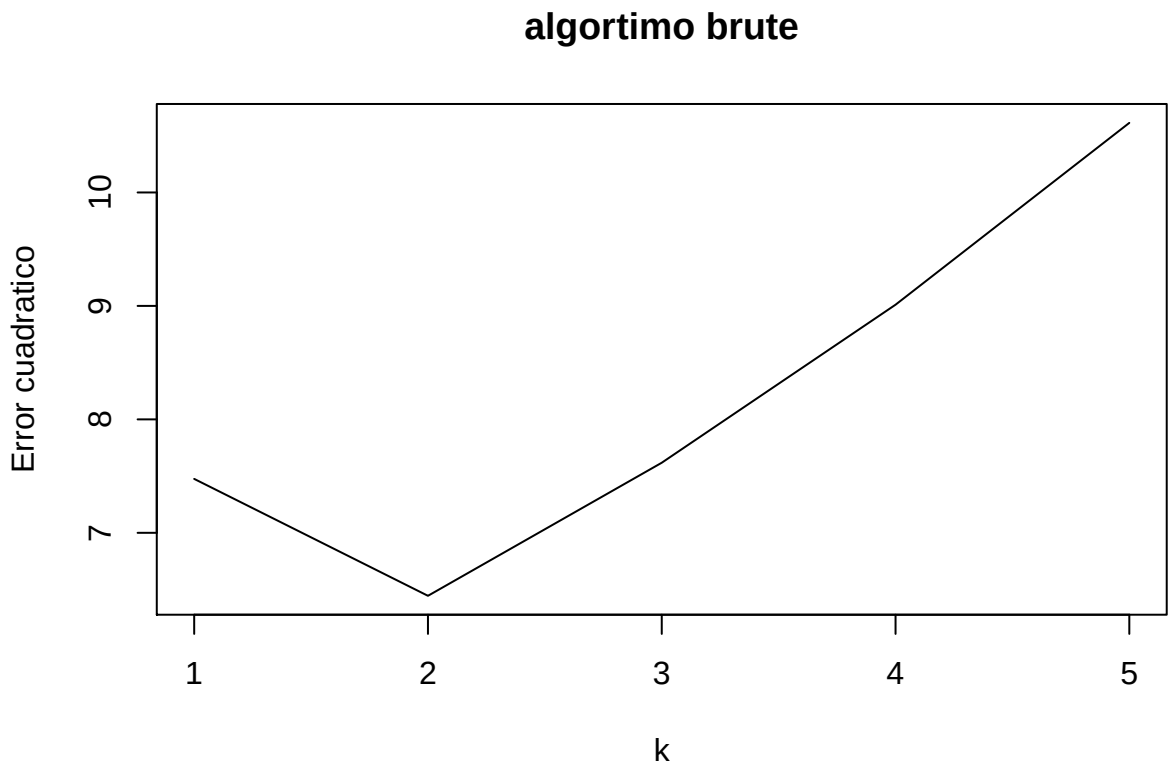
```
plot(errores_kd_tree_plot,xlab = "k",ylab = "Error cuadratico"
     ,type="l",main="algortimo kd_tree")
```




```
plot(errores_cover_tree_plot,xlab = "k",ylab = "Error cuadratico",
     type="l",main="algortimo cover_tree")
```



```
plot(errores_brute_plot,xlab = "k",ylab = "Error cuadratico" ,
     type="l",main="algortimo brute")
```




```
print(paste("Error en test validacion cruzada 5cv para kd_tree:",errors_knn_kd_tree,
            "y mejor k es",best_k_kd_tree ))
```

```
## [1] "Error en test validacion cruzada 5cv para kd_tree: 6.44485005931894 y mejor k es 2"
```

```
print(paste("Porcentaje de error en kd_tree= ",porcentaje_kd_tree*100,"%"))
```

```
## [1] "Porcentaje de error en kd_tree=  6.75052762462937 %"
```

```
print(paste("Error en test validacion cruzada 5cv para cover_tree:",
            errors_knn_cover_tree, "y mejor k es",best_k_cover_tree ))
```

```
## [1] "Error en test validacion cruzada 5cv para cover_tree: 6.44836110583056 y mejor k es 2"
```

```
print(paste("Porcentaje de error en cover_tree= ",porcentaje_cover_tree*100,"%"))
```

```
## [1] "Porcentaje de error en cover_tree=  6.75236616153055 %"
```

```
print(paste("Error en test validacion cruzada 5cv para brute:",errors_knn_brute,
            "y mejor k es",best_k_brute ))
```

```
## [1] "Error en test validacion cruzada 5cv para brute: 6.44485005931894 y mejor k es 2"
```

```r
print(paste("Porcentaje de error en brute= ",porcentaje_brute*100,"%"))
```

```
## [1] "Porcentaje de error en brute=  6.75052762462937 %"
```

Como podemos observar el error en los tres muy parecido.

Como obtenemos que el mejor k es k=2 vecinos, confirmamos nuestras sospechas de que el problema no iba a ser resuelto mediante un modelo lineal pues que mejor k=2 significa que las particiones del espacio son muchas y, por supuesto, no podrían ser modeladas mediante una sola línea recta. Entonces como trabajo adicional vamos a probar otros modelos no paramétricos como SVM en sus distintos kernels, Bagging y RandomForest.

Comparando Regresión Linear con Regularización y K-NN en regresión vemos que el error de RL-regularizada es 21.05 y para k-nn es de 6.44, y viendo los resultados claramente nos quedamos con K-NN en regresión.

10.1 Estudios adicionales.

Aunque quizás suponga una extralimitación, no nos convencían los errores obtenidos y decidimos probar con otros modelos, en vista que el problema no podría ser resuelto con un modelo lineal. El primero es Support Vector Machines en su versión de regresión, el segundo es Bagging, un caso particular de la técnica random forests.

- Support Vector Machines.

Para empezar vamos a probar un SVM con kernel lineal para ver si mejoramos Regresión Lineal Regularizada, lo haremos mediante validación cruzada:

```r
############SUPPORT VECTOR MACHINE EN REGRESION############
library(e1071)
set.seed(199)

nparticiones<-5
parts <- cut(seq(1,nrow(datos)),breaks=nparticiones,labels=FALSE)

#KERNEL LINEAL
error_svm=numeric(nparticiones)
for (i in 1:nparticiones) {

    testIndexes <- which(parts==i,arr.ind=TRUE)
    trainIndexes<- setdiff(as.numeric(rownames(datos)),testIndexes)

    conj_test_cv<-datos[testIndexes,]
    conj_training_cv<-datos[trainIndexes,]

    mod_svm <- svm(conj_training_cv$sound~.,
                   data=conj_training_cv,kernel="linear",type="eps-regression")
    pred_svm <- predict(mod_svm, conj_test_cv[,-ncol(conj_test_cv)])
    error_svm[i] <- mean((conj_test_cv[,ncol(conj_test_cv)]-pred_svm)*
                           (conj_test_cv[,ncol(conj_test_cv)]-pred_svm))
}

error_svm_final =mean(error_svm)
print(paste("Error de test svm kernel lineal: ", error_svm_final))
```

```
## [1] "Error de test svm kernel lineal:  23.9920604655249"
```

Como hemos visto que a este problema le van mejor los modelos no paramétricos vamos a probar que kernel nos va mejor, para ello vamos a utilizar tune para obtener los mejores parámetros y el mejor kernel para utilizarlo sobre nuestros datos.

Primero utilizamos la libreria e1071 y fijamos semilla:

```
library(e1071)
set.seed(199)

nparticiones<-5
parts <- cut(seq(1,nrow(datos)),breaks=nparticiones,labels=FALSE)
```

Obtenemos los mejores parámetros:

```
tuneResult <- tune(svm,
                   datos[,-ncol(datos)],
                   datos[,ncol(datos)])
tuneResult$best.model
```

```
##
## Call:
## best.tune(method = svm, train.x = datos[, -ncol(datos)], train.y = datos[,
##     ncol(datos)])
##
##
## Parameters:
##    SVM-Type:  eps-regression
##  SVM-Kernel:  radial
##        cost:  1
##       gamma:  0.2
##     epsilon:  0.1
##
##
## Number of Support Vectors:  1156
```

Como vemos nos dice que el mejor kernel es radial, que usemos e-regresión, cost 1, gamma 0.2 y epsilon 0.1.

cost = constante del término de regularización en la formulación de Lagrange.

epsilon = épsilon en la función de pérdida-sensible.

e-regresion = fija el ancho del margen.

Y ahora obtenemos mediante validación cruzada el error:

```
set.seed(199)

nparticiones<-5
parts <- cut(seq(1,nrow(datos)),breaks=nparticiones,labels=FALSE)


error_svm_mejor=numeric(nparticiones)
for (i in 1:nparticiones) {
```

```
  testIndexes <- which(parts==i,arr.ind=TRUE)
  trainIndexes<- setdiff(as.numeric(rownames(datos)),testIndexes)

  conj_test_cv<-datos[testIndexes,]
  conj_training_cv<-datos[trainIndexes,]

  mod_svm <- svm(conj_training_cv$sound~.,
                 data=conj_training_cv,
                 kernel="radial",
                 type="eps-regression",
                 cost=tuneResult$best.model$cost,
                 gamma=tuneResult$best.model$gamma,
                 epsilon=tuneResult$best.model$epsilon)

  pred_svm <- predict(mod_svm, conj_test_cv[,-ncol(conj_test_cv)])
  error_svm_mejor[i] <- mean((conj_test_cv[,ncol(conj_test_cv)]-pred_svm)*
                               (conj_test_cv[,ncol(conj_test_cv)]-pred_svm))

}
error_svm_final_mejor =mean(error_svm_mejor)
print(paste("Error de test svm kernel lineal: ", error_svm_final_mejor))
```

```
## [1] "Error de test svm kernel lineal:  10.65581502571"
```

- Bagging.

La forma de proceder de Bagging es muy sencilla. Se van a crear n árboles de regresión, escogiendo en cada uno todas las variables predictoras disponibles ($m = p$). Cada árbol elegirá ejemplos del dataset con reemplazo, es decir, pueden repetirse ejemplos en un mismo árbol. Una predicción de Bagging no es más que un promedio del valor que devuelve cada árbol.

Los parámetros requeridos por Bagging son el número de variables predictoras que elige cada árbol y el número de árboles de que consta el modelo. Dado que Bagging es un caso particular de random forest, donde vamos a elegir la totalidad de las variables predictoras, sólo nos queda elegir el número de árboles. Esto lo estimaremos mediante validación cruzada. Comencemos.

En primer lugar, vamos a estimar el número de árboles necesarios. En R se hace mediante validación cruzada con la función `tuneRF`, del paquete `randomForest`. En este caso, vamos a decirle que pruebe entre 1 (mínimo) y 2000 árboles (máximo) y que pare cuando la mejora sea menor que 0.05 en una determinada iteración.

```
library(randomForest)
```

```
## randomForest 4.6-12
```

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

```
x <- as.matrix(datos[,-ncol(datos)])
y <- as.matrix(datos[,ncol(datos)])

tuneRF(x,y,5,ntreeTry = 2000,stepFactor = 1,improve=0.05,plot=F,doBest=T)
```

```
## mtry = 5  OOB error = 2.988058
## Searching left ...
## Searching right ...

##
## Call:
##  randomForest(x = x, y = y, mtry = res[which.min(res[, 2]), 1])
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 5
##
##           Mean of squared residuals: 2.996802
##                     % Var explained: 93.7
```

El optimizador nos sugiere que ajustemos el modelo con 500 árboles. Calcularemos el valor de error de validación cruzada para este número de árboles.

```
set.seed(199)

nparticiones<-5
parts <- cut(seq(1,nrow(datos)),breaks=nparticiones,labels=FALSE)

errors_cv_rf<-numeric(nparticiones)
for (i in 1:nparticiones){
  testIndexes <- which(parts==i,arr.ind=TRUE)
  trainIndexes<- setdiff(as.numeric(rownames(datos)),testIndexes)

  conj_test_cv<-datos[testIndexes,]
  conj_training_cv<-datos[trainIndexes,]


  #MODELO Y CALCULO ERROR
  my_rf<-randomForest(conj_training_cv[,-ncol(conj_training_cv)],
                      conj_training_cv[,ncol(conj_training_cv)],
                      mtry = 5,
                      ntree = 500)

  pred_myrf<-predict(my_rf,conj_test_cv[,-nrow(conj_test_cv)])

  errors_cv_rf[i]<-mean(
    (pred_myrf-conj_test_cv[,ncol(conj_test_cv)])*
    (pred_myrf-conj_test_cv[,ncol(conj_test_cv)]))
}

print(paste("Error en test validacion cruzada 5cv:",mean(errors_cv_rf)))
```

```
## [1] "Error en test validacion cruzada 5cv: 3.59452007885731"
```

Hemos obtenido un error cuadrático de validación cruzada de 3.59, lo que en porcentaje se reduce a un 5.03% de error.

11. Estudio comparativo.

Vamos a comprarar los mejores modelos que hemos obtenido con cada técnica

- Regresión lineal: Hemos obtenido un error cuadrático mínimo de 21.04
- K-NN: Hemos obtenido un error cuadrático mínimo de 6.44
- SVM kernel radial: Hemos obtenido un error cuadrático mínimo de 10.66
- Bagging: Hemos obtenido un error cuadrático mínimo de 3.59

Claramente, el error más bajo que obtenemos es 3.59, con lo que, el ganador absoluto es Bagging. Si comparamos Regresión lineal con K-NN, el ganador claro es K-NN con un error de 6.44. Teóricamente, sabíamos que K-NN iba a ir muy bien, pues teníamos menos de 10 variables predictoras, cifra en la que K-NN empieza a deteriorar mucho su rendimiento, y efectivamente, así ha sido.

Referencias

- Regresión en espacios de alta dimensión - P.Saavedra
  (http://www.dma.ulpgc.es/webdma/index.php?option=com_docman&task=doc_download&gid=54&Itemid=197.)
- Regularization (Mathematics) - Wikipedia
  (https://en.wikipedia.org/wiki/Regularization_%28mathematics%29)
- Minería de datos (Regresión) - Cristina Tîrnăucă (Universidad de Cantabria)
  (http://personales.unican.es/tirnaucac/Slides/121016%20Laboratorio.pdf)
- VC Dimension - SVM
  (http://www.svms.org/vc-dimension/)
- A Tutorial on Support Vector Machines for Pattern Recognition - CHRISTOPHER J.C. BURGES https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/svmtutorial.pdf
- Paquetes R - https://cran.r-project.org/web/packages/available_packages_by_name.html