Práctica 3 Aprendizaje Automático: Ajuste de Modelos Lineales

Germán González Almagro, Antonio Manuel Milán Jiménez

27 de mayo de 2017

Índice

1. Modelos Lineales: Clasificación

1.1. Descripción del problema.

En esta primera sección trataremos de hallar el mejor clasificador para los datos de la base de datos Email spam. Se trata de un problema de clasificación en el que se tendrán en cuenta características propias de los correos electrónicos para clasificar éstos como spam o como email de contenido relevante. Algunas de las características que tendremos en cuenta son: número de apariciones de una palabra, número de caracteres, media de palabras en mayúscula, etc. Cabe destacar que en esta ocasión todas las características son de tipo cuantitativo, ya sean reales o continuas.

1.2. Conjuntos de datos de entrenamiento y test.

En este caso, los creadores de la base de datos proporcionan la división del conjunto de datos en datos de entrenamiento y datos de test, de forma que serán esas particiones las que utilizaremos para el aprendizaje en esta ocasión.

```
#Cargamos la base de datos
Spam <- read.table("EMailSpam/spam.data")
#Cargamos los indicadores del conjunto de datos
TrainTest <- read.table("EMailSpam/spam.traintest")
#Obtenemos las etiquetas de la base de datos, que se encuentran en la última columna.
Spam_Labels = Spam[ ,ncol(Spam)]
#Retiramos las etiquetas del conjunto de datos para el preprocesamiento
Spam = Spam[, -ncol(Spam)]
#Obtenemos los índices correspondientes al conjunto de entrenamiento
train_set = c(1:nrow(Spam))
train_set = train_set[TrainTest == 0]
```

1.3. Preprocesado de los datos.

Antes de obtener un clasificador, se hace necesario aplicar transformaciones en los datos para obtener de estos la mayor cantidad de información útil posible, dado que, en caso contrario, el clasificador que obtendremos podría estar considerando irregularidades en los datos que hagan que éste no se comporte de la manera esperada. En esta ocasión aplicaremos a los datos las siguientes transformaciones:

- **Transformación de Yeo-Johnson:** de manera muy similar a la transformación de Box-Cox, esta transformación actúa sobre la asimetría de los datos de forma que la disminuye para permitir un trato homogéneo de todos los atributos, con la salvedad de que la transformación de Yeo-Johnson es capaz de considerar atributos con media 0 o inferior, al contrario que la transformación de Box-Cox. En esta ocasión, aunque no hay atributos que presenten media negativa, sí hay atributos cuya media es 0 o muy cercana a 0.
- **Centrado:** consiste en calcular la media de los valores de cada atributo y restarla a cada uno de ellos en particular, de forma que la nueva media obtenida tras la transformación será 0. El objetivo de esta transformación es reducir la distancia entre los diferentes atributos.
- **Escalado:** consiste en dividir los valores de los atributos por la desviación típica de los mismos, de forma que se uniformiza el rango de los atributos para evitar favorecer los atributos con mayor rango en los modelos que tienen esta característica en cuenta. De esta forma trataremos los atributos independientemente de la escala en la que estos fueron medidos.
- **Análisis de componentes principales:** consiste en seleccionar, de entre todos los atributos, aquellos que representan la mayor varianza de los datos, es decir, aquellos que proporcionan más información

sobre el conjunto de datos. Deberemos tener en cuenta que este proceso es de tipo no supervisado, es decir, no tiene en cuenta la variable de respuesta, y por tanto, es posible que perdamos información relevante para la predicción de la misma.

Cabe destacar que, tras aplicar a los datos las transformaciones descritas, perdemos cualquier tipo de interpretación que pudiéramos hacer sobre los mismos.

Para aplicar estas transformaciones haremos uso del paquete `caret`, concretamente de la función `preProcess(...)`, que permite aplicar a los datos una serie de transformaciones dadas como argumento. Como hemos comentado, el análisis de componentes principales `PCA` es susceptible de eliminar atributos significativos para el aprendizaje del modelo, por ello, obtendremos dos conjuntos de datos, de forma que, sólo a uno se le ha aplicado `PCA`.

```
#Preprocesamiento de los datos: Método de Yeo Johnson, centrado, escalado y análisis de
#componentes principales.
PreProccesPCA = preProcess(Spam[train_set, ], thres=0.9,
                           method = c("YeoJohnson", "center", "scale", "pca"))
#Preprocesamiento de los datos: Método de Yeo Johnson, centrado, y escalado.
PreProccesNoPCA = preProcess(Spam[train_set, ], thres=0.9,
                             method = c("YeoJohnson", "center", "scale"))

#Aplicamos la fórmula obtenida a los datos
Spam_PreProcPCA = predict(PreProccesPCA, Spam)
Spam_PreProcNoPCA = predict(PreProccesNoPCA, Spam)

#Restauramos las etiquetas en los datos para poder especificarlas como
#variable de respuesta en la regresion lineal
Spam_PreProcPCA = cbind(Spam_PreProcPCA, Spam_Labels)
Spam_PreProcNoPCA = cbind(Spam_PreProcNoPCA, Spam_Labels)
```

1.4. Selección de la clase de funciones a utilizar.

Dado que no conocemos el mejor modelo que ajusta los datos, es lógico comenzar el estudio empleando los modelos más simples, es decir, los modelos lineales. Así pues, trataremos de obtener un clasificador mediante modelos lineales, utilizando para ello la clase de funciones lineales.

1.5. Discutir la necesidad de regularización y, en su caso, la función empleada para ello.

La regularización es la principal herramienta para combatir el sobreajuste a grandes rasgos, y siguiendo el principio de la navaja de Ockham, consiste en disminuir la complejidad del modelo obtenido, a costa de obtener un mayor error en al ajuste de los datos de entrenamiento. Con esto, y sabiendo que la clase de funciones elegida para la obtención del clasificador corresponde a la clase de polinomios más simple, a saber, los lineales, no parece necesario el uso de regularización. En cualquier caso, comprobaremos experimentalmente que aplicando regularización no obtenemos mejora en el modelo.

1.6. Definición de los modelos a utilizar y estimación de parámetros.

Para obtener un clasificador de los datos emplearemos técnicas de obtención de modelos lineales, en concreto emplearemos regresión lineal y regresión logística haciendo uso de la función `glm(...)` del paquete `stats`. Esta función recibe como parámetro una fórmula en la que debemos especificar la variable de respuesta y las variables que se tendrán en cuenta para obtener una clasificador para la misma. Además, deberemos especificar la familia de funciones que se empleará para obtener el error a minimizar, de forma que especificando como familia las gaussianas estaremos calculando regresión lineal y especificando la familia de binomiales estaremos calculando regresión logística.

Para la selección del subconjunto de variables que consideraremos en el aprendizaje emplearemos la función `regsubsets(...)` del paquete `leaps`, que nos proporciona información sobre los mejores subconjuntos de caraterísticas a utilizar dependiendo del número de ellas que consideremos. Esta función necesita como argumento, entre otros, un método de exploración del conjunto de atributos para proporcionar información sobre ellos; en este caso utilizaremos el método `forward` para los conjuntos de datos a los que no se les ha aplicado `PCA`, dado que el número de caracteristicas consideradas no permite realizar una búsqueda exhaustiva sobre los atributos en tiempo razonable. Al contrario sucede con los conjuntos de datos preprocesados con `PCA`, sobre los que aplicaremos búsqueda exhaustiva para la selección de subconjuntos de características, para ello especificaremos el método `exhaustive`.

Además, puesto que sabemos que al aumentar el número de variables empleadas para la obtención del clasificador, el error proporcionado por el mismo tiende a 0, y sabemos que no siempre escoger el mayor número de variables es la mejor opción, emplearemos un modelo que permita seleccionar el conjunto óptimo de variables teniendo en cuenta los dos factores anteriormente descritos. Para ello emplearemos el método *Akaike information criterion*, o C_p, que introduce una penalización en el error según el número de características empleadas para la obtención del modelo como sigue:

$$C_p = \frac{1}{n}(RSS + 2d\widehat{\sigma}^2)$$

Donde n es el número de variables asociadas a cada predictor, $\widehat{\sigma}$ es un estimador del error asociado a las variables de respuesta, d es el número total de predictores y RSS es la suma residual de errores cuadrados, que es una medida similar a la del error cuadrático, con la salvedad de que no se calcula la media del error total: $RSS = (e_1^2 + e_2^2 + +e_n^2)$. De esta forma tendremos en cuenta el número de características empleadas para la obtención del predictor a la hora de elegir el subconjunto de las mismas.

Para automatizar el proceso que elige el mejor conjunto de características con las que hallar el predictor, así como la obtención del mismo, implementamos las funciones `GetBestModel(...)` y `ProcessClassifModel(...)`, que procesan conjuntos de características y fórmulas para obtener un modelo de clasificación respectivamente:

```
ProcessClassifModel <- function(form, Data, Train_index, RVM_Index,
                                Family, f_valley, CalcErr){

  #Obtenemos el modelo en base a la fórmula dada como argumento
  model = glm(as.formula(form) , data = Data, subset = Train_index, family = Family)
  #Obtenemos las predicciones para los conjuntos de train y test
  model_predict_test = predict(model, Data[-Train_index, -RVM_Index], type = "response")
  model_predict_train = predict(model, Data[Train_index, -RVM_Index], type = "response")
  #Calculamos las etiquetas en base a la predicción
  calculated_labels_test = rep(0, length(model_predict_test))
  calculated_labels_test[model_predict_test >= 0.5] = 1
  calculated_labels_train = rep(0, length(model_predict_train))
  calculated_labels_train[model_predict_train >= 0.5] = 1
  #Calculamos el error mediante la matriz de confusión
  error_test = CalcErr(calculated_labels_test, Data[-Train_index, RVM_Index])
```

```
  error_train = CalcErr(calculated_labels_train, Data[Train_index, RVM_Index])

  #Obtenemos la curva de ROC
  pred = prediction(model_predict_test, Data[-Train_index, RVM_Index])
  perfArea = performance(pred, "auc")
  perfCurva = performance(pred, "tpr", "fpr")
  #Devolvemos los errores y los parámetros utilizados para obtenerlos,
  #además de la curva de ROC y el valor del área bajo ella
  list(E_train = error_train, E_test = error_test, formula = form, modelo = model,
                ROC = perfCurva, AreaROC = perfArea@y.values, FValley = f_valley)

} #Fin de ProcessClassifModel

GetBestModel <- function(ResponseVarName, Data, Train_index, Method,
                Nvmax, ProcessModel, Family, CalcErr = calcularErrorCuadratico){

  if(class(ProcessModel) != "function"){
    print("Error: ProcessModel debe ser una funcion que procese un modelo")
    return(-1)
  }

  #Obtenemos la fórmula que incuye la variable de respuesta
  # y las demás variables como predictores.
  RVM_Formula = paste(ResponseVarName, "~.")
  #Obtenemos la columna en la que se encuentra la variable de respuesta
  RVM_Index = which(colnames(Data) == ResponseVarName)
  #Obtenemos lo subconjuntos de mejores características
  #dentro del conjunto de datos de entrada
  subsets = regsubsets(as.formula(RVM_Formula), data = Data[Train_index, ],
                       method = Method, nvmax = Nvmax, really.big = T)

  summ = summary(subsets)
  matrix_summ = summary(subsets)$which[ , -1]
  #Obtenemos el primer valle de la lista proporcionada por C_p
  valleys = which(diff(sign(diff(summ$cp)), na.pad = FALSE) > 0)
  first_valley = ifelse(length(valleys) == 0, length(summ$cp), valleys[1] + 1)
  var_set = matrix_summ[first_valley, ]
  #Obtenemos los resultados utilizando como subconjunto de variables
  #el indicado por el primer valle de la lista proporcionada por C_p
  no_format_set = which(var_set)
  names_list = names(no_format_set)
  form = paste(names_list, collapse = '+')
  form = paste(ResponseVarName, "~",form, collapse = '')
  #Obtenemos y procesamos el modelo mediante la función dada como argumento
  ProcessModel(form, Data, Train_index, RVM_Index, Family, first_valley, CalcErr)
}
```

1.7. Selección y ajuste del modelo final

Tal y como hemos visto, PCA es un proceso no supervisado, por tanto, existe la posibilidad de que desechemos características relevantes para nuestro problema, consideraremos 4 modelos a comparar, dos obtenidos con regresión logística y dos obtenidos con regresión lineal, incluyendo cada pareja un modelo a cuyos datos se les aplicó `PCA` previamente y otro al que no se le aplicó.

Para calcular el error asociado a nuestros modelos utilizamos la matriz de confusión, de cuyo análisis podemos obtener, entre otros, el error que queremos estudiar:

```
ConfMatrixError <- function(calculated_labels, original_labels){
  #Obtenemos la matriz de confusión
  matrix_c = table(calculated_labels, original_labels)
  #Calculamos los aciertos
  p_aciertos = (matrix_c[1,1] + matrix_c[2,2])/sum(matrix_c)
  #Devolvemos los fallos
  (1 - p_aciertos)*100
}
```

Emplearemos la función `GetBestModel()`, detallada en el apartado anterior, para obtener los cuatro modelos que consideraremos:

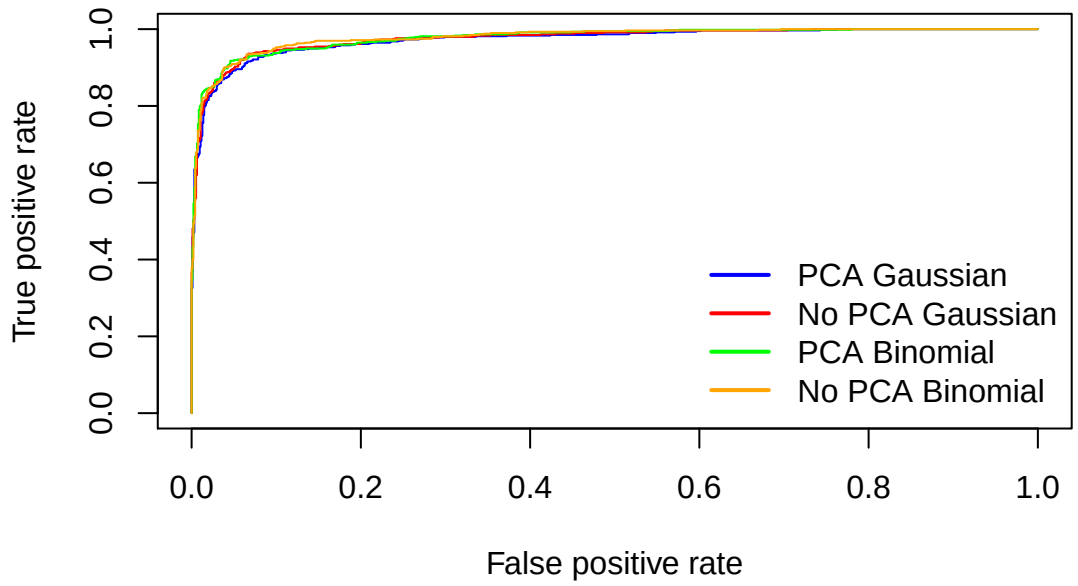
```
glmPCAGaussian = GetBestModel("Spam_Labels", Spam_PreProcPCA, train_set,
                              "exhaustive", 0.6*ncol(Spam_PreProcPCA),
                              ProcessModel = ProcessClassifModel, Family = "gaussian",
                              CalcErr = ConfMatrixError)

glmNoPCAGaussian = GetBestModel("Spam_Labels", Spam_PreProcNoPCA, train_set,
                                "forward", as.integer(0.6*ncol(Spam_PreProcNoPCA)),
                                ProcessModel = ProcessClassifModel, Family = "gaussian",
                                CalcErr = ConfMatrixError)

glmPCABinomial = GetBestModel("Spam_Labels", Spam_PreProcPCA, train_set,
                              "exhaustive", 0.6*ncol(Spam_PreProcPCA),
                              ProcessModel = ProcessClassifModel, Family = "binomial",
                              CalcErr = ConfMatrixError)

glmNoPCABinomial = GetBestModel("Spam_Labels", Spam_PreProcNoPCA, train_set,
                                "forward", as.integer(0.6*ncol(Spam_PreProcNoPCA)),
                                ProcessModel = ProcessClassifModel, Family = "binomial",
                                CalcErr = ConfMatrixError)
```

En la siguente tabla quedan reflejados los resultados obtenidos con cada uno de los 4 modelos considerados:

	PCA Gaussian	No PCA Gaussian	PCA Binomial	No PCA Binomial
E_{test}	7.2916667	6.9010417	5.9895833	6.7708333
E_{train}	6.6557912	5.9053834	5.8727569	5.1549755
Área ROC	0.9725743	0.9755874	0.9772681	0.9784683
Número Predictores	24	34	24	34

Podemos también representar las curvas de ROC obtenidas con cada uno de los modelos para visualizar las diferencias y similitudes entre ellas:

```
plot(glmPCAGaussian$ROC, col = "blue", lwd = 1)
plot(glmNoPCAGaussian$ROC, col = "red", lwd = 1, add = T)
plot(glmPCABinomial$ROC, col = "green", lwd = 1, add = T)
plot(glmNoPCABinomial$ROC, col = "orange", lwd = 1, add = T)
legend("bottomright",col = c("blue", "red", "green", "orange"), lwd = 2, bty = "n",
       c("PCA Gaussian","No PCA Gaussian", "PCA Binomial", "No PCA Binomial"))
```



La similitud de las curvas de ROC indica que todos los modelos son aproximadamente similares, y dado además que el área bajo ellas es en todos los casos cercana a 1, podemos decir que todos los modelos ajustan bien los datos. En esta ocasión no podremos usar esta información para decidir qué modelo es el mejor, dada la similitud de resultados.

Con todo lo anterior, y a la vista de los resultados, podemos decir que el mejor modelo es el obtenido mediante regresión logística y aplicando `PCA` a los datos antes de ser estos considerados para el aprendizaje, puesto que es el que menor E_{in} y E_{test} proporciona.

1.8. Experimentación con regularización

Podemos obtener nuevos modelos aplicando técnicas de regularización para estudiar el comportamiento de la misma cuando consideramos modelos lineales. Aplicaremos las técnicas de "weight decay" y "lasso", dos técnicas que aplican regularización para la obtención de modelos pero de formas distintas:

- **Weight Decay**: consiste en aplicar una factor multiplicativo λ al modelo de forma que acercamos los pesos a 0 y reducimos la complejidad del mismo para combatir el sobreajuste. Este modelo tendrá en cuenta todas las variables proporcionadas como predictores de la variable de respuesta.
- **Lasso**: similar a "weight decay", con la diferencia de que proporciona resultados de más sencilla interpretación. De igual forma que en "weight decay", aplicamos un factor multiplicativo λ al modelo obtenido, con la diferencia de que en esta ocasión cabe la posibilidad de que el peso asociado a algunas de las características predictoras quede a 0, por tanto, esta variable no será significativa en la predicción de nuevas etiquetas. Es decir, "lasso" selecciona el conjunto de variables más significativas, por lo que el modelo proporcionado será de más sencilla interpretación que el proporcionado con "weight decay".

En las dos técnicas descritas anteriormente la estimación del parámetro λ es de decisiva influencia en la obtención de un buen modelo, por tanto, no bastará con asignar λ de forma arbitraria; será necesario emplear técnicas que permitan estimar el mejor valor para este parámetro, como puede ser la validación cruzada.

Para la aplicación de las técnicas descritas emplearemos las funciones `cv.glmnet(...)`, que estima el mejor valor para λ aplicando validación cruzada, y `glmnet(...)`, que obtiene modelos aplicando técnicas de regularización.

1.8.1. Obtención del modelo mediante Weight Decay.

Emplearemos como características predictoras las que resultan del preprocesado de los datos sin `PCA`, teniendo en cuenta que "weight decay" no realizará distinción entre ellas. Comenzamos por obtener la mejor estimación del parámetro λ mediante la función `cv.glmnet(...)`.

```
#Obtenemos la matriz y las etiquetas asociadas al conjunto de datos
WD_DataMatrix = model.matrix(Spam_Labels ~., Spam_PreProcNoPCA)
labels = Spam_PreProcNoPCA[ ,ncol(Spam_PreProcNoPCA)]

#Obtenemos la estimación de lambda (especificamos alpha = 0 para aplicar Weight Decay)
WDCrossValLamb = cv.glmnet(WD_DataMatrix[train_set, ], labels[train_set], alpha = 0)
WD_best_lambda = WDCrossValLamb$lambda.min
```

Una vez obtenido λ lo damos como argumento a la función `glmnet(...)` para la obtención de un modelo que lo utilice como parámetro de regularización.

```
#Obtenemos el modelo (especificamos alpha = 0 para aplicar Weight Decay)
WeightDecayModel = glmnet(WD_DataMatrix[train_set, ], labels[train_set],
                          alpha = 0, lambda = WD_best_lambda, standardize = F)
#Pedecimos las etiquetas del conjunto de test
test_prediction = predict(WeightDecayModel, WD_DataMatrix[-(train_set), ])
calculated_labels_test = rep(0, length(test_prediction))
calculated_labels_test[test_prediction >= 0.5] = 1

#Pedecimos las etiquetas del conjunto de train
train_prediction = predict(WeightDecayModel, WD_DataMatrix[train_set, ])
calculated_labels_train = rep(0, length(train_prediction))
calculated_labels_train[train_prediction >= 0.5] = 1

WD_Error_test = ConfMatrixError(calculated_labels_test, labels[-(train_set)])
WD_Error_train = ConfMatrixError(calculated_labels_train, labels[train_set])

print(c("Error_test obtenido con Weight Decay: ", WD_Error_test))
```

```
## [1] "Error_test obtenido con Weight Decay: "
## [2] "6.640625"
```

```
print(c("Error_train obtenido Weight Decay: ", WD_Error_train))
```

```
## [1] "Error_train obtenido Weight Decay: "
## [2] "6.00326264274061"
```

1.8.2. Obtención del modelo mediante Lasso

De nuevo, emplearemos como características predictoras las que resultan del preprocesado de los datos sin PCA, teniendo en cuenta que, al contrario de lo que sucedía en el caso anterior, "lasso" desestimará algunas de las variables predictoras. Comenzamos por la obtención de la estimación de λ:

```
#Obtenemos la matriz y las etiquetas asociadas al conjunto de datos
LSS_DataMatrix = model.matrix(Spam_Labels ~., Spam_PreProcNoPCA)
labels = Spam_PreProcNoPCA[ ,ncol(Spam_PreProcNoPCA)]

#Obtenemos la estimación de lambda (especificamos alpha = 1 para aplicar Lasso)
LassoCrossValLamb = cv.glmnet(LSS_DataMatrix[train_set, ], labels[train_set], alpha = 1)
LSS_best_lambda = LassoCrossValLamb$lambda.min
```

Una vez obtenida la estimación de λ obtenemos el modelo:

```
#Obtenemos el modelo (especificamos alpha = 0 para aplicar Weight Decay)
LassoModel = glmnet(LSS_DataMatrix[train_set, ], labels[train_set],
                          alpha = 1, lambda = LSS_best_lambda, standardize = F)
#Pedecimos las etiquetas del conjunto de test
test_prediction = predict(LassoModel, LSS_DataMatrix[-(train_set), ])
calculated_labels_test = rep(0, length(test_prediction))
calculated_labels_test[test_prediction >= 0.5] = 1

#Pedecimos las etiquetas del conjunto de train
train_prediction = predict(LassoModel, LSS_DataMatrix[train_set, ])
calculated_labels_train = rep(0, length(train_prediction))
calculated_labels_train[train_prediction >= 0.5] = 1

LSS_Error_test = ConfMatrixError(calculated_labels_test, labels[-(train_set)])
LSS_Error_train = ConfMatrixError(calculated_labels_train, labels[train_set])

print(c("Error_test obtenido con Lasso: ", LSS_Error_test))
```

```
## [1] "Error_test obtenido con Lasso: " "6.8359375"
```

```
print(c("Error_train obtenido con Lasso: ", LSS_Error_train))
```

```
## [1] "Error_train obtenido con Lasso: " "6.06851549755302"
```

Podemos comprobar que efectivamente el modelo obtenido con "lasso" asigna peso 0 a algunas de las características predictoras, así como que el modelo obtenido con "weight decay" las considera todas:

```
weight.coeff = predict(WeightDecayModel, type = "coefficients")[1:ncol(WD_DataMatrix), ]
lasso.coeff = predict(LassoModel, type = "coefficients")[1:ncol(LSS_DataMatrix), ]

print("Características no consideradas por lasso:")
```

```
## [1] "Características no consideradas por lasso:"
```

```
print(lasso.coeff[lasso.coeff == 0])
```

```
## (Intercept)         V29         V56
##           0           0           0
```

```
print("Características no consideradas por weigth decay:")
```

```
## [1] "Características no consideradas por weigth decay:"
```

```
print(weight.coeff[weight.coeff == 0])
```

```
## (Intercept)
##           0
```

1.8.3. Conclusiones sobre regularización.

En conclusión, podemos decir que los modelos obtenidos con técnicas de regularización no mejoran a los obtenidos sin aplicar las mismas. Esto se debe a que el objetivo de la regularización es disminuir la complejidad del modelo obtenido para evitar el sobreajuste y, en este caso, al ser la familia de funciones considerada la de las funciones lineales, no será posible disminuir la complejidad del modelo obtenido, y por tanto, la única mejora que las técnicas de regularización pueden aplicar al modelo son desplazamientos espaciales o modificación de pendientes, modificaciones que en este caso no resultan en un mejor ajuste.

1.9. Estimación de E_{out} basada en el mejor modelo obtenido.

Una vez obtenido el mejor modelo, en este caso obtenido mediante regresión logística, podemos predecir su comportamiento fuera de la muestra calculando cotas para E_{out} basadas en E_in y en E_test.

Para calcular la cota basada en E_{in} aplicamos:

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

Calculamos la cota basada en E_{test} aplicando:

$$P(|E_{test}(g) - E_{out}(g)| > \epsilon) \leq 2e^{-2N\epsilon^2}$$

Sabemos que la probabilidad de $P(|E_{test}(g) - E_{out}(g)| > \epsilon) = \delta = 005$, por tanto, debemos encontrar el valor de ϵ para obtener una cota. Si se cumple la anterior desigualdad entonces se cumple que $\delta \leq 2e^{-2N\epsilon^2}$, basta con despejar ϵ para poder obtener un valor para la cota:

$$\delta \leq 2e^{-2N\epsilon^2} \rightarrow ln\left(\frac{\delta}{2}\right) \leq -2N\epsilon^2 \rightarrow \sqrt{\frac{ln\left(\frac{\delta}{2}\right)}{-2N}} \geq \epsilon \rightarrow \sqrt{\frac{ln(2) - ln(\delta)}{2N}} \geq \epsilon$$

Tomaremos ϵ como el primer valor que hace cierta la desigualdad.

```
#Calculamos los parámetros
N_test = length(train_set)
N_train = nrow(Spam_PreProcPCA) - length(train_set)
d_vc = ncol(Spam_PreProcPCA)+1; delta = 0.05
#Calculamos la cota basada en E_in
BasadaE_in = glmPCABinomial$E_train + sqrt(8/N_test*log((4*(2*N_test)^(d_vc) + 1)/delta))
print(c("Cota basada en E_in: ", BasadaE_in))
```

```
## [1] "Cota basada en E_in: " "6.85635744068915"
```

```
#Calculamos la cota basada en E_test
epsilon = sqrt((log(2) - log(delta))/(N_train))
BasadaE_test = glmPCABinomial$E_test + epsilon
print(c("Cota basada en E_test: ", BasadaE_test))
```

```
## [1] "Cota basada en E_test: " "6.03858960056457"
```

Vemos que la cota obtenida para E_{out} basada en E_{in} vale 6.85 %, mientras que, con un 95 % de confianza, la cota basada en E_{test} vale 6.03 %.

Dado que el conjunto de datos que utilizamos para aprender la función que clasifica los datos es E_{in}, parece lógico pensar que, para obtener un valor lo más cercano al real posible para E_{out}, debemos emplear un conjunto diferente al utilizado para aprender la función, esto es, E_{test}; por ello, deberá ser la cota basada en E_{test} la usada para estimar E_{out} y no la basada en E_{in}.

2. Modelos Lineales: Regresión

2.1. Descripción del problema.

Para el problema de regresión escogemos la base de datos LA Ozone. Con esta base de datos, a partir de datos como la velocidad del viento o la humedad, tendremos que hallar un modelo de regresión que prediga la concentración de ozono en un área concreta de Los Angeles. En esta base de datos, todas las características son cuantitativas.

2.2. Conjuntos de datos de entrenamiento y test.

Al contrario que en el anterior problema, en esta ocasión la base de datos proporcionada no está dividida en conjuntos de entrenamiento y test, de forma que decidimos obtener ambos conjuntos nosotros mismos. Para ello, escogeremos aleatoriamente un 70 % de los datos como conjunto de entrenamiento, siendo el 30 % restante el conjunto de test.

```
Ozone <- read.table("LAOzone/LAozone.data", sep = ',', header = T)

train_set = sample(1:nrow(Ozone), nrow(Ozone)*0.7, replace = F)
#Obtenemos las etiquetas del conjunto de datos
Ozone_Labels = Ozone[ ,1]
#Obtenemos las etiquetas transformadas según el logaritmo
Ozone_Labels_Log = apply(as.array(Ozone_Labels), MARGIN = 1, FUN =  log)
#Obtenemos las etiquetas transformadas según la raiz cuadrada
Ozone_Labels_Sqrt = apply(as.array(Ozone_Labels), MARGIN = 1, FUN =  sqrt)

#Eliminamos las etiquetas del conjunto de datos para el preprocesamiento
Ozone = Ozone[, -1]
```

2.3. Preprocesado de los datos.

Al igual que hicimos en el anterior problema, necesitamos aplicar una serie de transformaciones sobre nuestros datos para evitar que irregularidades en ellos, hagan que se obtenga un mal clasificador. Las transformaciones que usaremos son las mismas que en el anterior problema, es decir, la transformación de Yeo-Johnson, el centrado, el escalado y el análisis de componentes principales, todas ellas explicadas anteriormente.

Dado que el `PCA` es susceptible de eliminar atributos relevantes para el aprendizaje del modelo, decidimos trabajar con dos conjuntos de datos, habiéndole aplicado `PCA` sólo a uno de ellos. Además, obtendremos conjuntos de datos a los que se les han aplicado transformaciones no lineales, el motivo de ello será detallado más adelante.

```
#Preparacion de los datos: Método de Yeo Johnson, centrado, escalado y análisis de
#componentes principales
PreProccesPCA = preProcess(Ozone[train_set, ], thres=0.9,
                           method = c("YeoJohnson", "center", "scale", "pca"))

PreProccesNoPCA = preProcess(Ozone[train_set, ], thres=0.9,
                           method = c("YeoJohnson", "center", "scale"))

#Aplicamos la fórmula obtenida a los datos
Ozone_PreProcPCA = predict(PreProccesPCA, Ozone)
Ozone_PreProcNoPCA = predict(PreProccesNoPCA, Ozone)
```

```
#Aplicamos una transformación polinómica de grado 2 a los datos
Ozone_PreProcPCAPoly2 = data.matrix(Ozone_PreProcPCA)
Ozone_PreProcPCAPoly2 = poly(Ozone_PreProcPCAPoly2, degree = 2)

#Restauramos las etiquetas en los datos para poder especificarlas como
#variable de respuesta en la regresion lineal
Ozone_PreProcPCALog = cbind(Ozone_PreProcPCA, Ozone_Labels_Log)
Ozone_PreProcPCASqrt = cbind(Ozone_PreProcPCA, Ozone_Labels_Sqrt)
Ozone_PreProcPCA = cbind(Ozone_PreProcPCA, Ozone_Labels)
Ozone_PreProcNoPCA = cbind(Ozone_PreProcNoPCA, Ozone_Labels)
Ozone_PreProcPCAPoly2Log = data.frame(cbind(Ozone_PreProcPCAPoly2, Ozone_Labels_Log))
Ozone_PreProcPCAPoly2Sqrt = data.frame(cbind(Ozone_PreProcPCAPoly2, Ozone_Labels_Sqrt))
Ozone_PreProcPCAPoly2 = data.frame(cbind(Ozone_PreProcPCAPoly2, Ozone_Labels))
```

2.4. Selección de la clase de funciones a utilizar.

Dado que no conocemos el mejor modelo que ajusta los datos, es lógico comenzar el estudio empleando los modelos más simples, es decir, los modelos lineales. Así pues, trataremos de obtener un clasificador mediante modelos lineales, utilizando para ello la clase de funciones lineales.

Además, en esta ocasión, aplicaremos transformaciones no lineales a los datos; los motivos de esta transformación serán detallados más adelante.

2.5. Discutir la necesidad de regularización y, en su caso, la función empleada para ello.

Tal y como se comentó en el anterior problema, estamos utilizando la clase de funciones más simples, las lineales, por lo que no parece lógico utilizar regularización, pues no es posible reducir la complejidad del modelo y únicamente nos limitaremos a modificar el modelo lineal obtenido sin la aplicación de regularización.

2.6. Definición de los modelos a utilizar y estimación de parámetros.

Utilizaremos la misma función que empleamos en el problema anterior, a saber, la función `glm(...)` del paquete `stats`. En esta ocasión, sólo utilizaremos la familia de funciones gaussianas, es decir, las correspondientes a regresión lineal, puesto que al ser un problema de regresión no podremos realizar regresión logística.

Vamos a utilizar también la función `regsubsets(...)` para obtener los mejores subconjuntos de características a emplear en función del número de características que se utilicen. De igual forma que en el anterior problema, tendremos que apoyarnos en el método *Akaike information criterion*, o C_p, para obtener el conjunto óptimo de características con el que obtendremos el mejor modelo.

Para obtener el modelo emplearemos la función `GetBestModel()`, que ya detallamos en el problema anterior, y `ProcessRegModel(...)`, que procesa una fórmula para obtener un modelo de regresión:

```
ProcessRegModel <- function(form, Data, Train_index, RVM_Index, Family,
                            f_valley, CalcErr = calcularErrorCuadratico){

  model = glm(as.formula(form) , data = Data, subset = Train_index, family = Family)

  calculated_labels_test = predict(model, Data[-Train_index, -RVM_Index],
                                   type = "response")
```

```
  calculated_labels_train = predict(model, Data[Train_index, -RVM_Index],
                                    type = "response")

  error_test = CalcErr(calculated_labels_test, Data[-Train_index, RVM_Index])
  error_train = CalcErr(calculated_labels_train, Data[Train_index, RVM_Index])
  results = list(E_train = error_train, E_test = error_test,
                 formula = form, modelo = model, FValley = f_valley)

}
```

2.7. Selección y ajuste del modelo final

En esta ocasión emplearemos el error cuadrático como medida del error obtenido con un clasificador, puesto que no es posible emplear la matriz de confusión en un problema de regresión:

```
calcularErrorCuadratico <- function(calculadas, originales){

  diferencia = calculadas-originales
  diferencia = diferencia^2
  (sum(diferencia)/length(diferencia))

}
```

Para obtener los modelos utilizamos la función `GetBestModel()`. Obtendremos en esta ocasión tres modelos, uno a cuyos datos de entrada hemos aplicado `PCA`, otro al que no le aplicamos dicha transformación, y finalmente uno a cuyos datos, además de aplicárseles `PCA`, se les aplica también una transformación polinómica de grado 2.

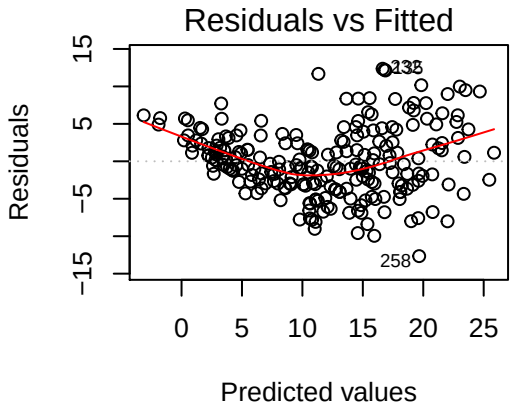
```
glmPCAGaussian = GetBestModel("Ozone_Labels", Ozone_PreProcPCA, train_set,
                              "exhaustive", 0.6*ncol(Ozone_PreProcPCA),
                              ProcessModel = ProcessRegModel, Family = "gaussian")

glmNoPCAGaussian = GetBestModel("Ozone_Labels", Ozone_PreProcNoPCA, train_set,
                                "exhaustive", 0.6*ncol(Ozone_PreProcNoPCA),
                                ProcessModel = ProcessRegModel, Family = "gaussian")

glmPCAGaussianPoly2 = GetBestModel("Ozone_Labels", Ozone_PreProcPCAPoly2, train_set,
                                   "forward", as.numeric(ncol(Ozone_PreProcPCAPoly2)),
                                   ProcessModel = ProcessRegModel, Family = "gaussian")
```

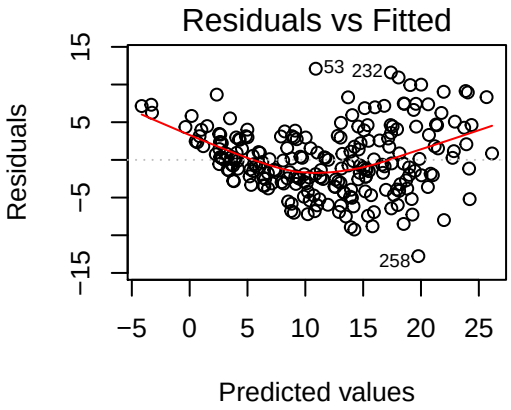
En la siguente tabla quedan reflejados los resultados obtenidos con cada uno de los 2 modelos considerados:

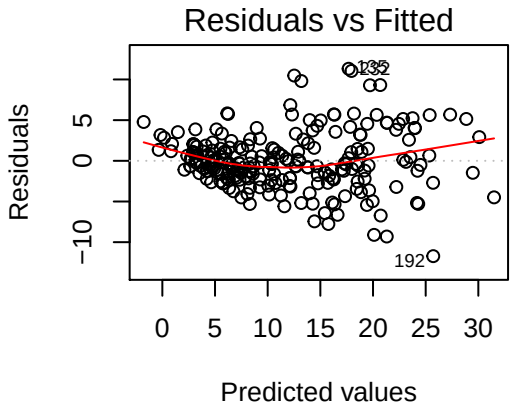
	PCA	No PCA	Poly
E_{test}	20.2229218	21.598072	20.4617342
E_{train}	20.7469341	19.1610992	12.3760735
Predictores	4	6	21

A la vista de los resultados podemos concluir que los errores obtenidos mediante estos modelos están por encima de lo que consideramos como aceptable, esto puede ser indicativo de la presencia de no linealidad en los datos, para cerciorarnos de ello podemos representar los errores residuales obtenidos con el modelo:

```
par(mfrow = c(2,2))
plot(glmPCAGaussian$modelo, which = c(1))
plot(glmNoPCAGaussian$modelo, which = c(1))
plot(glmPCAGaussianPoly2$modelo, which = c(1))
```







Como esperábamos, observamos un patrón de distribución que no corresponde a la distribución que se espera, en concreto, corresponde a la forma de cono que asociamos a la varianza no constante de los errores respecto al hiperplano que los clasifica o heterocedasticidad. Para corregir esto debemos aplicar transformaciones no lineales a los datos, como pueden ser el logaritmo ($log(x)$) o la raíz cuadrada ($\sqrt{x}$) y obtener nuevos modelos considerando estas transformaciones. Para comparar los modelos asociados a estas transformaciones con los dos anteriores es necesario deshacer dichas transformaciones en el cálculo del error, para ello implementamos las funciones `calcularErrorCuadraticoLog(...)` y `calcularErrorCuadraticoSqrt(...)`:

```
#Deshace la transformación logarítmica en el cálculo del error
calcularErrorCuadraticoLog <- function(calculadas, originales){
  originales = exp(originales); calculadas = exp(calculadas)
  diferencia = calculadas-originales
  diferencia = diferencia^2
  (sum(diferencia)/length(diferencia))

}
```

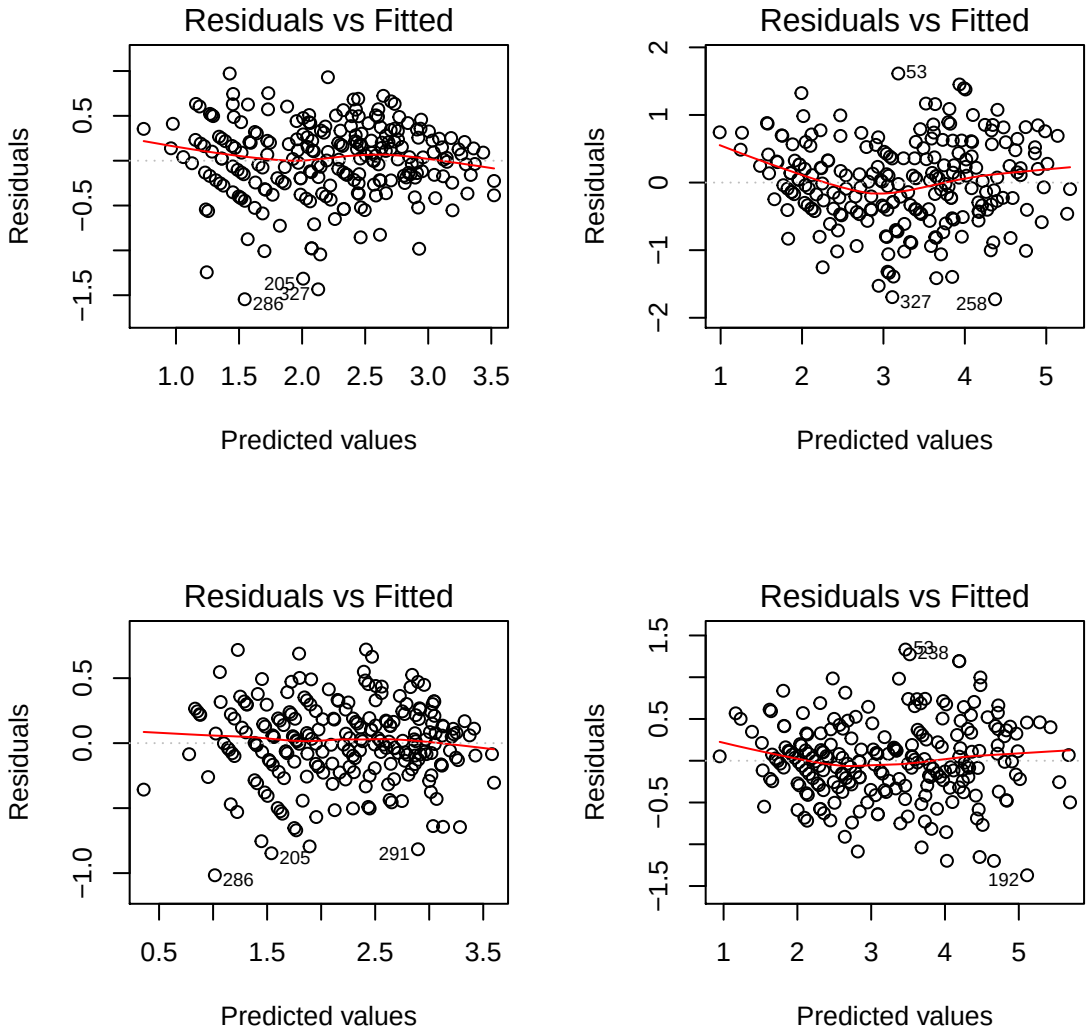
```
#Deshace la transformación raiz cuadrada en el cálculo del error
calcularErrorCuadraticoSqrt <- function(calculadas, originales){
  originales = originales^2; calculadas = calculadas^2
  diferencia = calculadas-originales
  diferencia = diferencia^2
  (sum(diferencia)/length(diferencia))

}
```

Empleamos la función `GetBestModel(...)` para obtener los modelos, dando en este caso como argumento las funciones necesarias para calcular el error en cada caso. Emplearemos `calcularErrorCuadraticoLog(...)` para calcular el error en el caso de las transformaciones logarítmicas, y `calcularErrorCuadraticoSqrt(...)` para calcular el error cuando la transformación aplicada es la raíz cuadrada.

```
glmPCAGaussianLog = GetBestModel("Ozone_Labels_Log", Ozone_PreProcPCALog, train_set,
                                  "exhaustive", ncol(Ozone_PreProcPCALog),
                                  ProcessModel = ProcessRegModel, Family = "gaussian",
                                  CalcErr = calcularErrorCuadraticoLog)

glmPCAGaussianSqrt = GetBestModel("Ozone_Labels_Sqrt", Ozone_PreProcPCASqrt, train_set,
                                 "exhaustive", ncol(Ozone_PreProcPCASqrt),
                                 ProcessModel = ProcessRegModel, Family = "gaussian",
                                 CalcErr = calcularErrorCuadraticoSqrt)

glmPCAGaussianPolyLog = GetBestModel("Ozone_Labels_Log", Ozone_PreProcPCAPoly2Log,
                                 train_set, "forward", ncol(Ozone_PreProcPCAPoly2Log),
                                 ProcessModel = ProcessRegModel, Family = "gaussian",
                                 CalcErr = calcularErrorCuadraticoLog)

glmPCAGaussianPolySqrt = GetBestModel("Ozone_Labels_Sqrt", Ozone_PreProcPCAPoly2Sqrt,
                                 train_set, "forward", ncol(Ozone_PreProcPCAPoly2Sqrt),
                                 ProcessModel = ProcessRegModel, Family = "gaussian",
                                 CalcErr = calcularErrorCuadraticoSqrt)
```

Una vez obtenidos los modelos representamos los errores residuales asociados a los mismos:

```
par(mfrow = c(2,2))
plot(glmPCAGaussianLog$modelo, which = c(1))
plot(glmPCAGaussianSqrt$modelo, which = c(1))
plot(glmPCAGaussianPolyLog$modelo, which = c(1))
plot(glmPCAGaussianPolySqrt$modelo, which = c(1))
```



En este caso los errores se distribuyen de forma que no es posible identificar un patrón en ellos que haga sospechar que sea necesaria otra transformación. Recogemos en la siguiente tabla los resultados obtenidos con los 7 modelos:

	PCA	NoPCA	Poly	Log	Sqrt	PolyLog	PolySqrt
E_{test}	20.2229218	21.598072	20.4617342	16.4123852	16.9263414	15.3344853	14.3629445
E_{train}	20.7469341	19.1610992	12.3760735	19.8086555	18.8204222	13.1411205	11.8499454
Predictores	4	6	21	4	4	19	19

2.8. Conclusiones sobre los modelos obtenidos.

Comparando los cuatro modelos obtenidos podemos decir que aquel que proporciona un mejor ajuste es el obtenido aplicando PCA a los datos, así como una transformación polinómica de orden 2 y transformando las etiquetas según la raíz cuadrada para corregir la heterocedasticidad de los datos, ya que este modelo predice con un error medio de 3.79, frente al 3.93 de mínimo y 4.53 de máximo que proporcionan los demás modelos. Es por ello que será este modelo el que consideraremos para la estimación de E_{out}.

2.9. Estimación de E_{out} basada en el mejor modelo obtenido.

Una vez obtenido el mejor modelo, en este caso obtenido aplicando a los datos análisis de componentes principales en el preprocesado, así como una transformación cuadrática de orden 2 en los predictores y transformando las etiquetas según la raíz cuadrada, podemos predecir su comportamiento fuera de la muestra calculando cotas para E_{out} basadas en E_{in} y en E_{test}.

```
#Calculamos los parámetros
N_test = length(train_set)
N_train = nrow(Ozone_PreProcPCAPoly2Sqrt) - length(train_set)
d_vc = ncol(Ozone_PreProcPCAPoly2Sqrt)+1; delta = 0.05
#Calculamos la cota basada en E_in
BasadaE_in = glmPCAGaussianPolySqrt$E_train +
             sqrt(8/N_test*log((4*(2*N_test)^(d_vc) + 1)/delta))

print(c("Cota basada en E_in: ", BasadaE_in))
```

```
## [1] "Cota basada en E_in: " "14.3672773005061"
```

```
#Calculamos la cota basada en E_test
epsilon = sqrt((log(2) - log(delta))/(N_train))
BasadaE_test = glmPCAGaussianPolySqrt$E_test + epsilon
print(c("Cota basada en E_test: ", BasadaE_test))
```

```
## [1] "Cota basada en E_test: " "14.5550090604202"
```

Vemos que la cota obtenida para E_{out} basada en E_{in} vale 14.36, mientras que, con un 95 % de confianza, la cota basada en E_{test} vale 14.55.

Dado que el conjunto de datos que utilizamos para aprender la función que clasifica los datos es E_{in}, parece lógico pensar que, para obtener un valor lo más cercano al real posible para E_{out}, debemos emplear un conjunto diferente al utilizado para aprender la función, esto es, E_{test}; por ello, deberá ser la cota basada e E_{test} la usada para estimar E_{out} y no la basada en E_{in}.

3. Referencias

- Yaser S. Abu-Mostafa, Malik Magdon-Ismail, Hsuan-Tien Lin. Learning From Data (2012).
- Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani. An Introduction to Statistical Learning with Applications in R (2014).