Práctica 3

Laura Calle Caraballo
Javier León Palomares

21 de mayo de 2017

Índice

1. Clasificación.

Usaremos la base de datos *SPAM E-mail Database*, que contiene 58 atributos distribuidos de la siguiente forma:

- 48 variables reales continuas en el intervalo $[0, 100]$ que corresponden a la proporción en la que aparecen 48 palabras concretas en un correo.
- 6 variables reales continuas en el intervalo $[0, 100]$ que corresponden a la proporción en la que aparecen los caracteres ; · (· [· ! · $ · #.
- Una variable real continua que representa la longitud media de las secuencias de caracteres en mayúsculas.
- Una variable entera continua que contiene la longitud de la mayor cadena de caracteres encontrada en mayúsculas.
- Una variable entera que corresponde al total de mayúsculas encontradas.
- La variable de respuesta (1 si el correo se consideró spam y 0 si no se consideró spam).

Esta base de datos no contiene valores perdidos y su proporción de etiquetas es de $39,4\,\%$ de positivos y $60,6\,\%$ de negativos.

Las particiones de entrenamiento y test se encuentran ya definidas de tal forma que usaremos 3065 de los 4601 datos para ajustar los modelos y los 1536 restantes para realizar pruebas.

1.1. Preprocesamiento de datos.

Para realizar el preprocesamiento de datos hemos empleado la función `preProcess` del paquete `caret`; esta función permite aplicar distintos métodos de transformación para adecuar las características iniciales a los modelos que vamos a emplear.

A continuación vamos a explicar los métodos utilizados:

- Transformación de *Yeo-Johnson*: es muy similar a la transformación de *Box-Cox* pero, a diferencia de ésta, permite la existencia de valores negativos ó 0. En nuestro caso, no existen valores negativos pero sí es bastante frecuente encontrar valores iguales a 0.
- Centrado: se realiza la media de los valores de cada característica y se le resta a cada valor particular, siendo la nueva media igual a 0. Esto reduce las distancias entre las distintas características.
- Escalado: este método divide los valores por su desviación típica. El objetivo de esta transformación es tener características con rangos uniformes para evitar problemas con modelos que consideran distancias entre valores (lo cual favorecería las características con valores más separados entre sí); además, la independencia respecto a la escala en la que se realizaron las medidas también es importante.
- Análisis de componentes principales (*PCA*): se buscan nuevos ejes de coordenadas de forma que la varianza de algunas características sea lo suficientemente pequeña como para descartarlas. Como resultado se consigue una reducción de dimensionalidad donde las nuevas características, que no guardan relación de significado con las originales, representarán la parte más significativa de la varianza de los datos.

Otro método que se podría haber utilizado para reducir la dimensionalidad de los datos es *Near Zero Variance*, que elimina predictores con varianza cercana a 0, ya que esto suele indicar que su valor es casi constante a lo largo de la muestra y apenas aportan información. En el caso de este conjunto de datos muchas de las características representan frecuencias de aparición de palabras concretas, por lo que es de esperar que en muchas observaciones estos valores sean 0; sin embargo, este hecho no implica que sus contribuciones no tengan importancia.

Una vez explicadas las transformaciones, veamos cómo se traduce en la práctica a código:

```
# Lectura de datos y obtención del conjunto de entrenamiento
spam = leer_datos_spam()
indices_train = which(spam$conjuntos == 0)

# Función que encapsula el preprocesado de datos
preprocesar_datos = function(datos,indices_train,metodos,umbral_varianza){

    preprocess_obj = preProcess(datos[indices_train,],method=metodos,umbral_varianza)
    nuevosDatos = predict(preprocess_obj,datos)

}

# Preprocesamiento (Yeo-Johnson, centrado, escalado, análisis de componentes principales...)
spam_procesado = preprocesar_datos(spam$datos,indices_train,
                                   c("YeoJohnson","center","scale","pca"),0.85)
# Sin PCA
spam_procesado_sin_pca = preprocesar_datos(spam$datos,indices_train,
                                           c("YeoJohnson","center","scale"),0.85)
```

Ya que existe la posibilidad de que el análisis de componentes principales elimine características relevantes, vamos a realizar experimentos separados preprocesando los datos con y sin esta técnica.

1.2. Clases de funciones a utilizar.

Ya que esta práctica trata de ajustes de modelos lineales, las clases de funciones a utilizar serán, en principio, las correspondientes a polinomios de grado 1.

No obstante, si observásemos indicios de no linealidad, podríamos plantearnos realizar algún tipo de transformación no lineal utilizando, por ejemplo, los polinomios de *Legendre*.

1.3. Regularización.

Es bastante común enfrentarnos al sobreajuste derivado de considerar una clase demasiado compleja sobre los datos de entrenamiento. Según el principio de la *navaja de Ockham*, la explicación más simple suele ser la correcta, por lo que una posible aproximación puede ser la simplificación.

La regularización toma este concepto para proponer una solución al sobreajuste. La metodología se basa en penalizar los valores altos en las componentes del vector de pesos de la función (ya sea forzando que algunos pesos sean 0 ó restringiendo los valores de todos ellos, entre otras formas).

En el caso de las funciones lineales, la regularización no parece en principio necesaria, debido a que únicamente consideraremos hiperplanos, el tipo más simple de clasificadores. Aun así, comprobaremos los efectos de aplicar esta técnica para asegurarnos que no es necesaria; en concreto, utilizaremos *weight decay*, proporcionado por el paquete `glmnet`.

1.4. Modelos utilizados.

En primer lugar, es necesario hablar de `regsubsets`, función mediante la cual seleccionaremos los conjuntos de características con las que construiremos distintos modelos para predecir la variable de respuesta. Dicha función proporciona, para un parámetro n, los conjuntos $\{C_1, C_2, \ldots, C_i, \ldots, C_n : |C_i| = i\}$ de los mejores predictores que encuentra. Esta selección de características se puede realizar de varias maneras diferentes:

- Hacia delante: se comienza sin variables y se añade, tras probarlas todas, la que produce una mejora más significativa del ajuste. Se repite este proceso hasta que ninguna incrementa la efectividad de forma estadísticamente notable.
- Hacia atrás: se empieza contando con todas las variables y se elimina la que empeora de forma menos significativa la calidad del ajuste. El proceso se repite hasta que ninguna variable se puede eliminar sin deteriorar seriamente el modelo.
- *Sequential Replacement*: una combinación de variables predictoras (modelo) se representa mediante un vector y está asociada a una medida de su calidad. En cada iteración se realizan todos los reemplazos posibles de variables originales por otras variables y se escoge la mejor nueva combinación, que será el punto de partida de la siguiente iteración. Este proceso se repite hasta que no haya mejora (convergencia).
- Exhaustiva: se utiliza *Branch and Bound* para realizar una búsqueda lo más completa posible manteniendo un nivel aceptable de eficiencia.

En los casos en los que se haya realizado una reducción de dimensionalidad importante usaremos una búsqueda exhaustiva. Sin embargo, si tenemos un número considerable de características deberemos optar por métodos más rápidos ya que el tiempo de ejecución la convertiría en inviable; en particular, si no utilizamos análisis de componentes principales, usaremos la búsqueda hacia delante.

La función que utiliza `regsubsets` para obtener los subconjuntos de variables con los que vamos a probar en las distintas regresiones es la siguiente:

```
subconjuntos_formulas = function(datos,max_tam,metodo="exhaustive"){
    # Obtenemos los subconjuntos de variables
    subsets = regsubsets(etiquetas~.,data=datos,method=metodo,nvmax=max_tam)
    # Obtenemos la matriz de características seleccionadas por grupos de tamaño [1,nvmax]
    matriz_subsets = summary(subsets)$which[,-1]
    # Guardamos, para cada fila, las columnas cuyas variables han sido seleccionadas.
    seleccionados = apply(matriz_subsets,1,which)
    # Obtenemos los nombres de esas columnas (para utilizarlos en la regresión)
    seleccionados = lapply(seleccionados,names)
    # Construimos la suma de las variables que usaremos en la regresión lineal
    seleccionados = mapply(paste,seleccionados,MoreArgs=list(collapse="+"))
    # Construimos strings equivalentes a las fórmulas que usaremos en la regresión lineal
    formulas = mapply(paste,rep("etiquetas~",max_tam),seleccionados,USE.NAMES = FALSE)
    # Construimos objetos fórmula
    formulas = apply(matrix(formulas,nrow=length(formulas)), 1, as.formula)
    list(formulas=formulas,cp=summary(subsets)$cp,bic=summary(subsets)$bic)
}
```

En esta función comenzamos obteniendo una matriz que nos indica qué variables se seleccionan para cada tamaño; a continuación, extraemos los nombres de las características para posteriormente construir sus fórmulas asociadas y así poder utilizarlas como argumento en las etapas siguientes de la experimentación.

Como modelos básicos de ajuste hemos utilizado la regresión lineal con una familia de funciones gaussiana que implementa la función `lm` y la regresión lineal logística proporcionada como una de las posibilidades de la función `glm`. Asimismo, hemos considerado el uso de regularización para comprobar empíricamente si es útil en nuestro caso.

Para realizar los experimentos sobre una base común, la misma colección de conjuntos de predictores obtenida mediante `regsubsets` será empleada tanto con `lm` como con `glm`. Para ambas funciones, la decisión de qué conjunto representa el mejor modelo se regirá por distintos criterios: la proporción de etiquetas incorrectas frente al total y los estimadores *Bayesian information criterion* (BIC) y C_p.

Comenzando por la regresión lineal de la función `lm`, los únicos parámetros que necesita son los datos de entrenamiento ya procesados, sus etiquetas y las variables que va a utilizar para calcular la regresión. La función que encapsula el proceso hasta el cálculo del porcentaje de error es la siguiente:

```
evalua_lm = function(formula,datos,subconjunto,fp=1,fn=1){
    # do.call pasa los parámetros a lm de forma correcta
    reg_lin = do.call("lm", list(formula=formula, data=substitute(datos),
                                 subset=substitute(subconjunto)))
    prediccion_test = evaluar_regresion(reg_lin,datos[-subconjunto,-ncol(datos)])
    porc_error = porcentaje_error(categorizar(prediccion_test),
                                  datos[-subconjunto,ncol(datos)],fp,fn)
    list(formula=formula, error = porc_error)
}
```

Para utilizar lo anterior en los datos con y sin selección de características sólo necesitamos las siguientes líneas:

```
ajustes_lm = mapply(evalua_lm, formulas, MoreArgs = list(datos = spam_procesado,
                                                         subconjunto = indices_train))
ajustes_lm_sin_pca = mapply(evalua_lm, formulas_sin_pca,
                            MoreArgs = list(datos = spam_procesado_sin_pca,
                                            subconjunto = indices_train))
```

Del mismo modo, utilizaremos la función `glm` para calcular regresiones logísticas. Los parámetros requeridos son los mismos que para `lm` aunque con la inclusión de la familia de funciones que queremos utilizar, que por defecto es la gaussiana (correspondiente a la regresión lineal). Nosotros emplearemos la binomial, ya que trata con la probabilidad de etiquetar una observación en una de dos posibles categorías; en otras palabras, implementa la regresión logística. De nuevo, la función de `R` que encapsula el uso de `glm` se muestra a continuación:

```
evalua_glm = function(formula,datos,subconjunto,fp=1,fn=1,familia=binomial()){
    reg_lin = do.call("glm", list(formula=formula, data=substitute(datos),
                                  subset=substitute(subconjunto),familia))
    prediccion_test = evaluar_regresion(reg_lin,datos[-subconjunto,-ncol(datos)])
    porc_error = porcentaje_error(categorizar(prediccion_test),
                                  datos[-subconjunto,ncol(datos)],fp,fn)
    list(formula=formula, error=porc_error,reg=reg_lin)
}
```

Las llamadas a función para calcular las regresiones logísticas quedan, pues, de la siguiente forma:

```
ajustes_glm = mapply(evalua_glm, formulas,
                     MoreArgs = list(datos = spam_procesado,
                                     subconjunto = indices_train))
ajustes_glm_sin_pca = mapply(evalua_glm, formulas_sin_pca,
                             MoreArgs = list(datos = spam_procesado_sin_pca,
                                             subconjunto = indices_train))
```

Finalmente, antes de aplicar regularización debemos estudiar en qué circunstancias tiene sentido. Una vez realizado el análisis de componentes principales, el cual ya elimina presumiblemente características que conduzcan a un sobreajuste, la regularización podría provocar *underfitting* o infraajuste. No obstante, si omitimos la etapa de PCA este problema tiene menos posibilidades de presentarse, por lo que realizaremos el experimento en este último caso.

El paquete `glmnet` implementa la funcionalidad que necesitamos. La regularización precisa de un hiperparámetro λ que hemos de calcular previamente usando la función `cv.glmnet`, la cual aplica validación cruzada para obtener el mejor valor de λ (el que obtiene una menor tasa de error).

```
# Creamos la matriz de datos en el formato que necesita glmnet
x = model.matrix(etiquetas~.,spam_procesado_sin_pca)[,-ncol(spam_procesado_sin_pca)]
y = spam_procesado_sin_pca$etiquetas
# Obtenemos los errores de validación cruzada en el conjunto
cv.out = cv.glmnet(x[indices_train,],y[indices_train],alpha=0)
# Guardamos el lambda que ha dado menor error de validación cruzada
bestlambda = cv.out$lambda.min
```

La función `cv.glmnet` devuelve, entre otros datos, un modelo que no consideraremos ya que es resultado del proceso de validación cruzada; esto significa que ha sido ajustado con un subconjunto del conjunto de entrenamiento y posiblemente no tenga la calidad suficiente.

Por ello, vamos a hacer uso de la función `glmnet`; ésta recibe como parámetros los datos y etiquetas en el mismo formato que `cv.glmnet`, además de un α que varía entre 0 y 1 e indica la proporción en la que se aplican respectivamente *weight decay* y *LASSO*, y un conjunto de valores de λ que se encargará de probar (en nuestro caso, sólo el mejor λ obtenido como ya hemos visto).

```
# Obtenemos un modelo de Ridge
modelo_ridge = glmnet(x,y,alpha=0,lambda=bestlambda)
# Calculamos las predicciones y el error asociado a ellas
modelo_ridge.pred = predict(modelo_ridge,s=bestlambda,newx=x[-indices_train,])
error_ridge = porcentaje_error(categorizar(modelo_ridge.pred),
                               spam_procesado[-indices_train,ncol(spam_procesado)],fp=1)
```

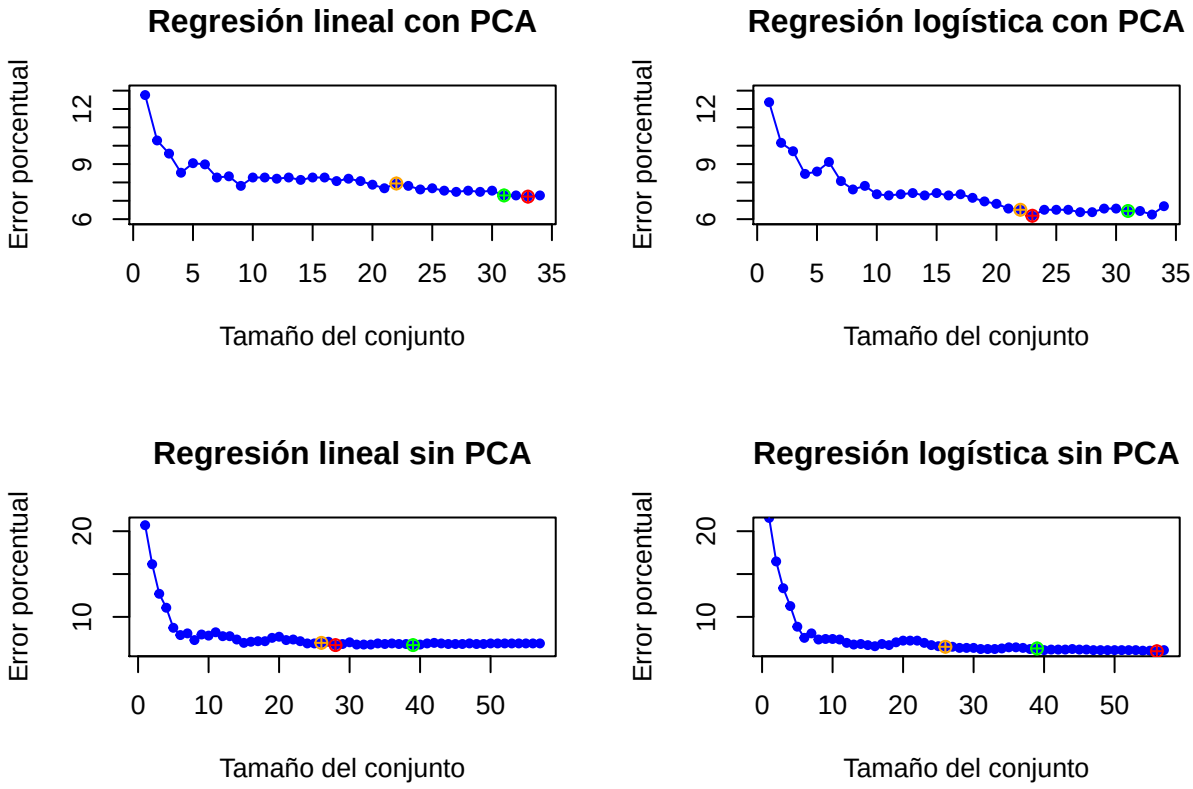
Una vez obtenidos los modelos, es el momento de estudiar sus tasas de error y elegir el mejor de ellos; lo haremos, pues, en la siguiente sección.

1.5. Selección del modelo final.

Para seleccionar un modelo estudiaremos en primer lugar las tasas de error. El cálculo de dicho error hace uso de la matriz de confusión y se ha implementado como se muestra a continuación, donde `fp` y `fn` corresponden a las penalizaciones que se le dan a falsos positivos y falsos negativos respectivamente. En este caso ambos tendrán el mismo coste ya que, aunque la semántica de nuestro problema sugiere que los falsos positivos se penalicen sobre los falsos negativos, no se proporciona una matriz de coste.

```
porcentaje_error = function(clasificados,reales,fp=1,fn=1){
    reales[reales == 0] = -1
    t = table(clasificados,reales)
    total_predicciones = sum(t)
    t[1,2] = t[1,2]*fn
    t[2,1] = t[2,1]*fp
    100*(1-sum(diag(t))/total_predicciones)
}
```

Como hemos generado varios conjuntos de modelos utilizando las funciones `lm` y `glm`, podemos comenzar comparando y eligiendo un modelo que represente cada uno de estos conjuntos (el mejor de ellos). Las siguientes gráficas muestran los errores obtenidos con diferentes subconjuntos de predictores para los datos con y sin análisis de componentes principales:

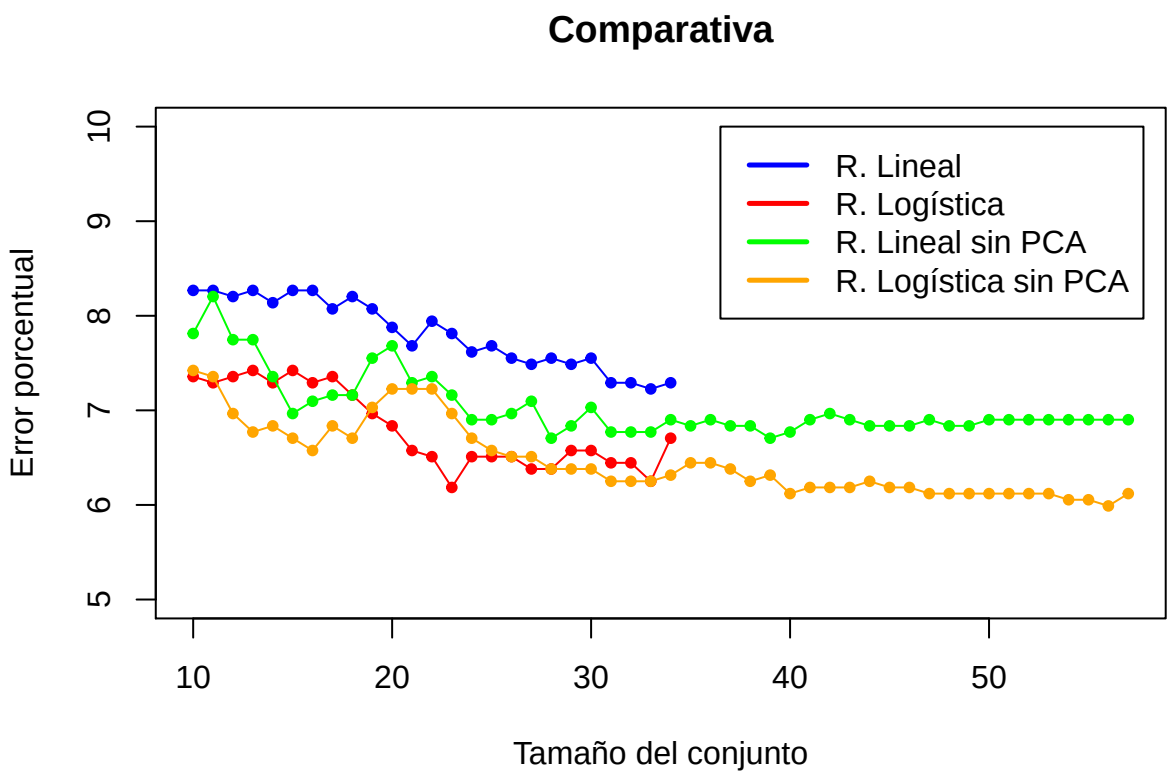


Los modelos con los que se obtiene el mínimo E_{test} se han marcado en rojo; adicionalmente, se han destacado en verde los obtenidos con C_p y en naranja los obtenidos con *BIC*, ambos proporcionados por `regsubsets`. El estimador C_p, además de utilizar el error cuadrático estimado del ajuste realizado por un subconjunto de predictores, añade una penalización al número de características escogidas. Su objetivo es evitar el sobreajuste, puesto que un número de predictores elevado puede resultar en que características que no son de relevancia jueguen un papel importante en el modelo, obteniendo resultados pobres al aplicarse a nuevas observaciones. *BIC* es un estimador similar que se diferencia del primero en el cálculo de la penalización. Para ello, tiene en cuenta el número total de observaciones además del número de características seleccionadas, por lo que generalmente elige modelos con menos predictores que C_p.

En la siguiente tabla se muestran, para los distintos tipos de ajuste y datos, los modelos seleccionados siguiendo los tres criterios descritos. Para cada modelo se proporcionan el E_{test} y el número de predictores utilizado.

	Error Mínimo		Estimador C_p		Estimador *BIC*	
	E_{test}	Tamaño	E_{test}	Tamaño	E_{test}	Tamaño
Ajustes lm	7.226562	33	7.291667	31	7.942708	22
Ajustes glm	6.184896	23	6.445312	31	6.510417	22
Ajustes lm sin PCA	6.705729	28	6.705729	39	6.966146	26
Ajustes glm sin PCA	5.989583	56	6.315104	34	6.510417	26

En la tabla podemos observar una diferencia de calidad entre el modelo de regresión logística sin PCA de 56 características y los demás. Esto podría conducirnos a elegirlo sin más como nuestro modelo definitivo, ya que en general parece razonable escoger el modelo que da el menor error en la partición de test. Sin embargo, en algunos casos podemos preferir un modelo con un error ligeramente superior a cambio de una reducción sustancial de dimensionalidad. Por ello, vamos a representar las cuatro gráficas anteriores conjuntamente para ver mejor si existe una alternativa de este tipo:



Visualmente podemos deducir lo que un análisis más detallado de los números nos diría: asumiendo un pequeño aumento de E_{test} (6.12 frente a 5.98), podríamos quedarnos con un modelo de regresión logística sin PCA con hasta 16 características menos; otras alternativas serían un E_{test} de 6.05 con 2 características menos o un E_{test} de 6.18 con 33 características menos con regresión logística usando PCA.

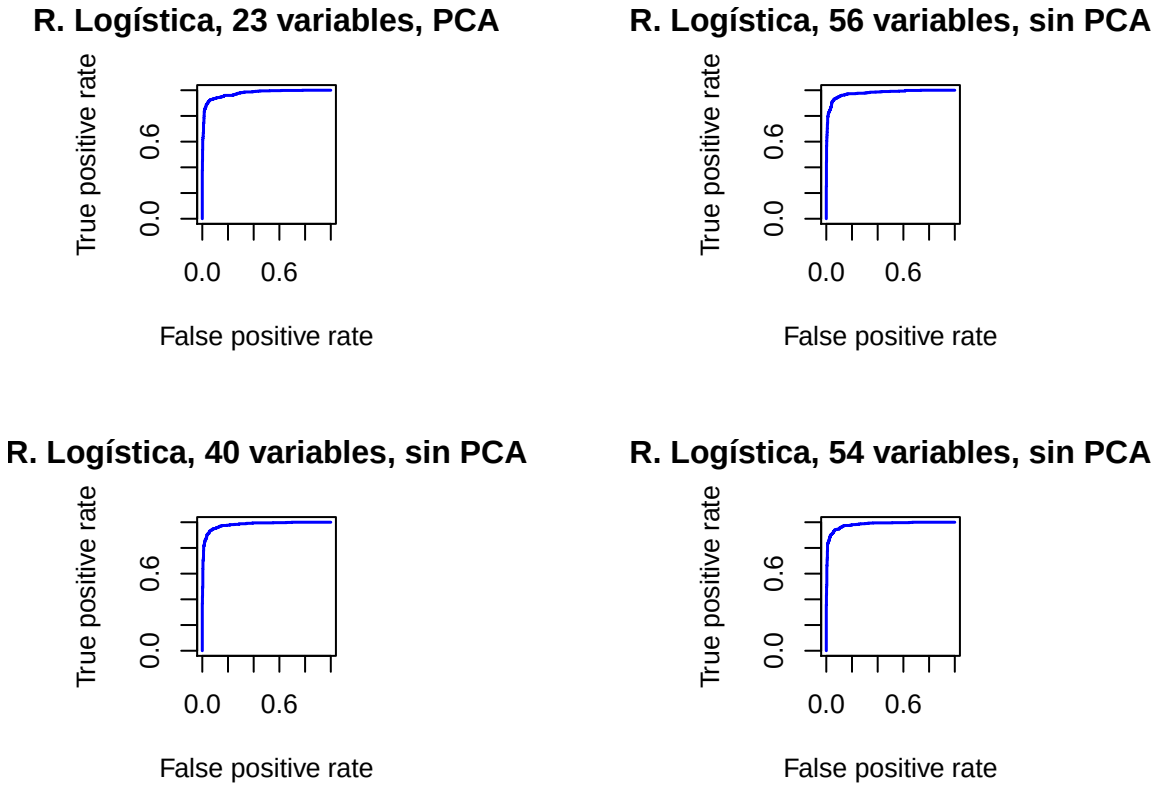
Adicionalmente, antes de considerar cuál de los cuatro modelos anteriores vamos a escoger, sería interesante recordar el ajuste con regularización y las transformaciones no lineales que hemos mencionado en secciones anteriores para saber si nos pueden aportar algo nuevo. En el caso de la regularización, el porcentaje de error que presenta es de un 6.57 %, pobre en comparación con los otros modelos. Esto es debido a que, como ya habíamos adelantado, la poca complejidad de nuestras funciones reduce la utilidad de esta técnica; por ello, tomamos la decisión de descartar la regularización. Del mismo modo, teniendo en cuenta el bajo error de los modelos ya estudiados, no merece la pena aumentar en vano la complejidad de la clase de funciones, por lo que no se aplicarán transformaciones no lineales.

Por último, vamos a realizar sobre los cuatro modelos otro tipo de análisis, el basado en la curva *ROC* (*Receiver Operating Characteristic*); esta curva representa la tasa de verdaderos positivos frente a la tasa de falsos positivos, y el área bajo ella nos dará una medida de lo acertado de las predicciones de un modelo.

En `R` podemos trabajar con curvas *ROC* usando el paquete `ROCR`. Vamos a ver una función que a partir de unas predicciones y unas etiquetas reales nos devuelve el área y la curva lista para ser representada con `plot`:

```
calcula_curva_roc = function(pred,truth){
    predob = prediction(pred,truth)
    area = performance(predob,"auc")
    curva = performance(predob,"tpr","fpr")
    list(curva=curva,area=area)
}
```

Las cuatro curvas *ROC* generadas quedan de la siguiente manera:



En las gráficas anteriores se puede observar una gran similitud, por otra parte esperada, entre los cuatro modelos. Ya que no podemos distinguir de esta forma cuál es el mejor según este criterio, vamos a obtener el área bajo las curvas (*AUC*) para trabajar con una medida objetiva.

	23 variables, PCA	40 variables sin PCA	54 variables sin PCA	56 variables sin PCA
AUC	0.9458017	0.9807312	0.9819779	0.9771073

Basándonos tanto en las tasas de error E_{test} como en las áreas bajo las curvas *ROC*, hemos decidido que el modelo de regresión logística con análisis de componentes principales utilizando 23 predictores presenta el balance más razonable entre dimensionalidad y precisión.

1.5.1. Descripción del modelo seleccionado.

En primer lugar, vamos a ver el error dentro de la muestra que presenta nuestro modelo. Esto se calcula mediante el siguiente código:

```
mejor_reg = ajustes_glm[3,23]
etiquetas_train = evaluar_regresion(mejor_reg,
                                    spam_procesado[indices_train,-ncol(spam_procesado)])
error_mejor_reg = porcentaje_error(categorizar(unlist(etiquetas_train)),
                                   spam_procesado[indices_train,ncol(spam_procesado)])
```

El error E_{in} obtenido es de 6.39478 %, muy similar a E_{test}, lo que nos indica que el modelo generaliza correctamente.

Es el momento asimismo de estimar E_{out}. Con este propósito utilizaremos la cota basada en la desigualdad de *Hoeffding*:

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

Donde N es el número de observaciones de la partición de test y $\mathbb{P}[|E_{test}(g) - E_{out}(g)| > \epsilon]$ corresponde a una tolerancia δ que fijaremos a 0.05.

$$\delta \leq 2e^{-2N\epsilon^2}$$

Junto con el error en la partición de test obtendremos, despejando ϵ de la siguiente forma, un intervalo en el que se encuentra E_{out} con una tolerancia $\delta = 0.05$.

$$\sqrt{\frac{log(2) - log(\delta)}{2N}} \geq \epsilon$$

Esta desigualdad nos da un valor de $\epsilon = 0.0346$ y, por tanto, un intervalo para E_{out} de $6.1848 \pm 0.0346 = [6.1502, 6.2194]$.

El modelo obtenido emplea las siguientes variables con sus pesos asociados:

Variables	Pesos	Variables	Pesos
PC1	-1.59929266	PC18	-0.24185342
PC2	-0.43305137	PC22	0.26230726
PC3	0.78132498	PC23	0.20279272
PC4	0.69798947	PC24	0.14152461
PC5	-0.67353870	PC26	0.33330240
PC6	0.31172429	PC27	-0.39325017
PC7	0.38812438	PC28	0.60798142
PC11	-0.63229755	PC31	0.35697994
PC12	0.33770499	PC32	0.15659858
PC13	-0.29214977	PC33	-0.26189766
PC16	0.13648889	PC34	-0.03502565
PC17	0.14671441	Intercept	-1.36529004

2. Regresión.

Para el caso de regresión utilizaremos la base de datos de *Los Angeles ozone*, cuyas observaciones están compuestas por las siguientes características:

- Altura geopotencial sobre la base aérea de Vandenberg para una presión de 500 milibares.
- Velocidad del viento en millas por hora.
- Porcentaje de humedad.
- Temperatura en la base aérea de Sandburg.
- Altura base de inversión térmica.
- Gradiente barométrico desde el aeropuerto internacional de Los Ángeles hasta Daggett.
- Temperatura base de inversión térmica.
- Visibilidad (en millas).
- Día del año.
- Variable de respuesta: concentración horaria media de ozono atmosférico en Upland, California.

Esta base de datos no contiene valores perdidos; tampoco se nos provee de particiones prefijadas de entrenamiento y test, por lo que dividiremos la muestra en dos subconjuntos elegidos aleatoriamente (aunque con una semilla inicializada a un valor concreto) con una proporción 70 %-30 % respectivamente:

```
ozono = leer_datos_ozono()
o_indexes_train = sample(nrow(ozono$datos),round(0.7*nrow(ozono$datos)))
o_labels = ozono$etiquetas
```

2.1. Preprocesamiento de datos.

Siguiendo los mismos razonamientos que en la sección dedicada a clasificación, emplearemos también transformaciones de *Yeo-Johnson*, centrado, escalado y análisis de componentes principales; en el caso de este último, volveremos a analizar la diferencia entre aplicarlo y no aplicarlo mediante dos conjuntos de datos preprocesados:

```
o_procesados = preprocesar_datos(ozono$datos,o_indexes_train,
                                 c("YeoJohnson","center","scale","pca"),0.9)
o_procesados_sin_pca = preprocesar_datos(ozono$datos,o_indexes_train,
                                         c("YeoJohnson","center","scale"),0.9)
```

Debido al número de características de este *dataset*, bastante pequeño en comparación con el empleado en clasificación, es de esperar que la reducción de dimensionalidad no sea tan drástica. Otra consecuencia de esto es que el riesgo de eliminar predictores relevantes es mayor.

2.2. Clases de funciones a utilizar.

De nuevo, teniendo en cuenta que esta práctica trata de ajustes de modelos lineales, las clases de funciones que emplearemos serán los polinomios de grado 1.

Si observamos indicios de no linealidad en las gráficas de errores residuales nos plantearemos hacer transformaciones no lineales, principalmente con la función `poly`.

2.3. Regularización.

Al igual que en clasificación, aparentemente no es necesaria esta técnica; no obstante, si decidiésemos aplicar transformaciones no lineales para poder trabajar con modelos más complejos, sería interesante estudiar lo que la regularización nos puede aportar (en particular, *weight decay*).

2.4. Selección y análisis de modelos.

Sobre los datos procesados ejecutaremos `regsubsets` para obtener subconjuntos de características de la misma forma que en el experimento anterior. Ya que en esta ocasión disponemos de pocos predictores, podremos aplicar un método de selección exhaustivo. La función de `R` para automatizar el proceso de obtención de fórmulas será la misma que habíamos definido (`subconjuntos_formulas`).

Dada la naturaleza de este problema, no podemos usar regresión logística, por lo que nos limitaremos a la función `lm`, que ofrece regresiones lineales gaussianas.

Una vez aclarados estos detalles, vamos a pasar a seleccionar un conjunto de modelos candidatos; para medir su eficacia utilizaremos el error cuadrático medio (*MSE*):

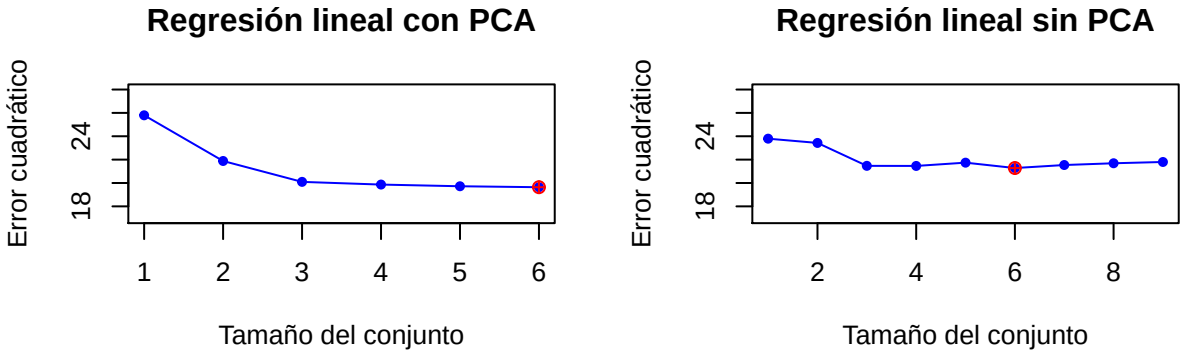
$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{y} - y)^2$$

La función de `R` para encapsular el cálculo de la regresión y su error cuadrático es muy similar a la utilizada en clasificación:

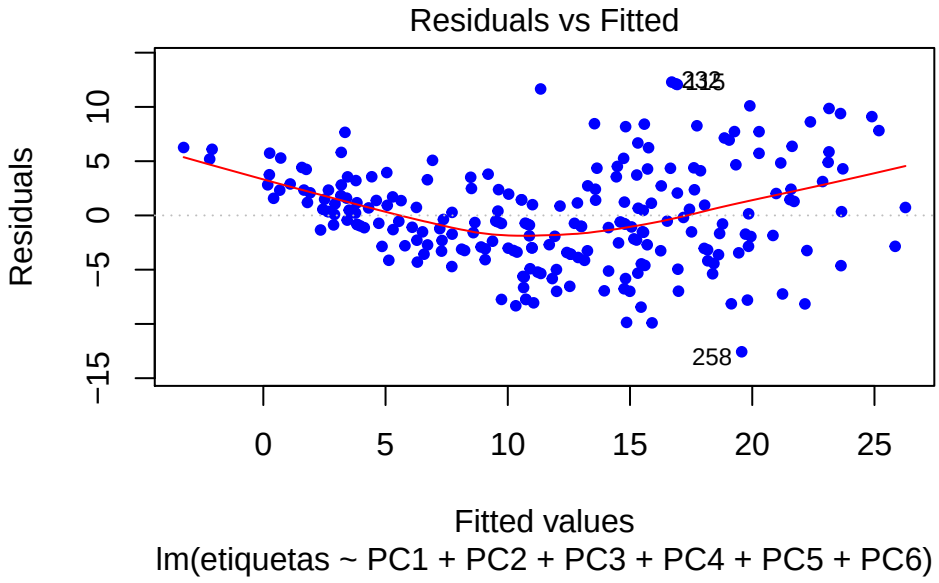
```
evalua_lm = function(formula,datos,subconjunto){
    reg_lin = do.call("lm", list(formula=formula, data=substitute(datos),
                                 subset=substitute(subconjunto)))
    prediccion_test = evaluar_regresion(reg_lin,datos[-subconjunto,-ncol(datos)])
    error = error_cuadratico_medio(prediccion_test,datos[-subconjunto,ncol(datos)])
    list(formula=formula, error = error, reg = reg_lin)
}

# Evaluamos todos los modelos
ajustes = mapply(evalua_lm, o_formulas, MoreArgs = list(datos = o_procesados,
                                                        subconjunto = o_indexes_train))
ajustes_sin_pca = mapply(evalua_lm, o_formulas_sin_pca,
                         MoreArgs = list(datos = o_procesados_sin_pca,
                                         subconjunto = o_indexes_train))
```

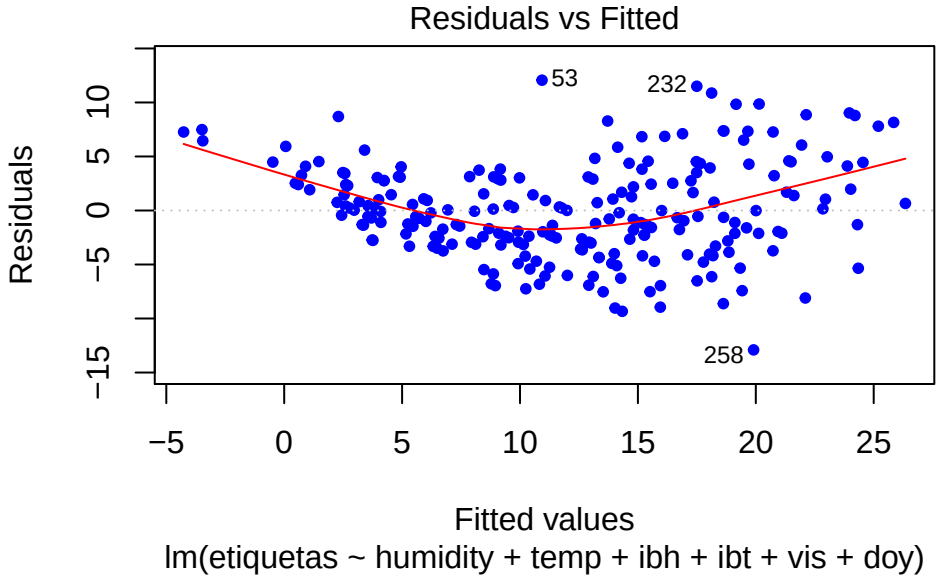
A continuación se muestran dos gráficas correspondientes al *MSE* de los modelos tanto con PCA como sin PCA:



En rojo se representa el modelo cuyo error cuadrático es menor. Ya que se trata de errores que no bajan del rango $[18, 21]$ (19.6393 en el caso de PCA y 21.2807 en el caso de sin PCA), vamos a estudiar sus errores residuales para tener más información al respecto. Primero vamos a ver los correspondientes al mejor modelo con PCA:



Podemos identificar patrones en la disposición de los elementos; la curva nos indica una no linealidad de los datos, mientras que la dispersión de los puntos a mayores valores de la variable predicha sugiere un fenómeno denominado heterocedasticidad que comentaremos después de arreglar la no linealidad.



En esta segunda gráfica, que representa los errores residuales del mejor modelo sobre los datos a los que no se les ha aplicado PCA, podemos observar las mismas tendencias, reafirmando nuestra hipótesis de que el modelo no es el más adecuado en estos momentos. Por lo tanto, es preciso solucionar este problema.

Para corregir nuestro modelo y así poder adaptarlo mejor a la no linealidad que presentan los datos, emplearemos la función poly, que calcula polinomios hasta un cierto grado a partir de las características originales. En nuestro caso, aplicaremos polinomios de grado dos (como sugieren las curvas rojas de las gráficas anteriores).

Con el objetivo de que la dimensionalidad no se dispare, vamos a aplicar las transformaciones sobre el subconjunto de características que nos ha proporcionado un error más bajo para cada conjunto de datos (con y sin PCA).

Dado que el proceso para utilizar poly es un poco críptico por el tipo de argumentos que recibe, vamos a mostrar únicamente la llamada para el modelo sin análisis de componentes principales, que es más ilustrativa (para más detalles, consultar el código):

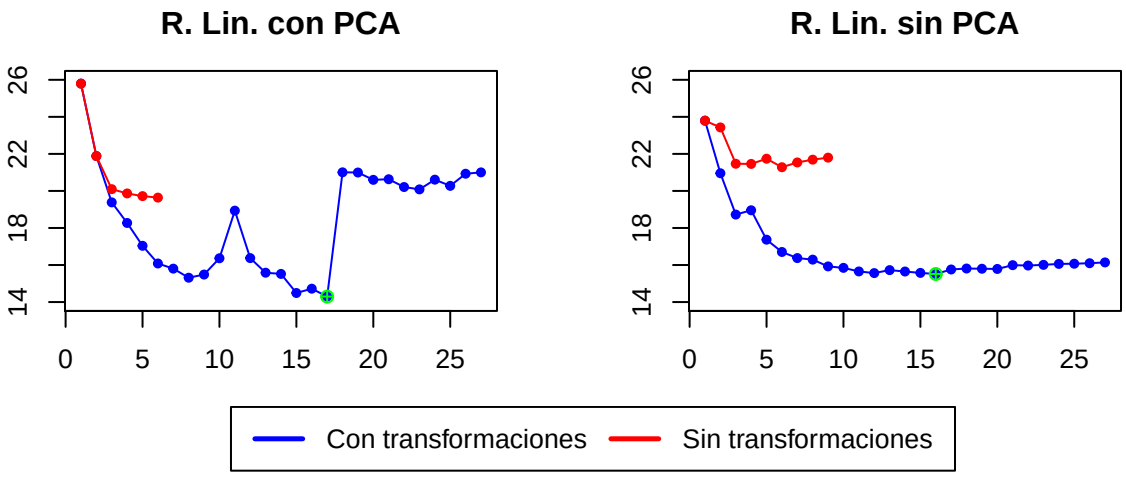
```
o_procesados_sin_pca_poly = poly(
    as.matrix(o_procesados_sin_pca[,c("humidity","temp","ibh","ibt","vis","doy")]),
    degree=2)
```

A continuación, obtenemos los subconjuntos de predictores más relevantes y los utilizamos para ajustar sus modelos correspondientes:

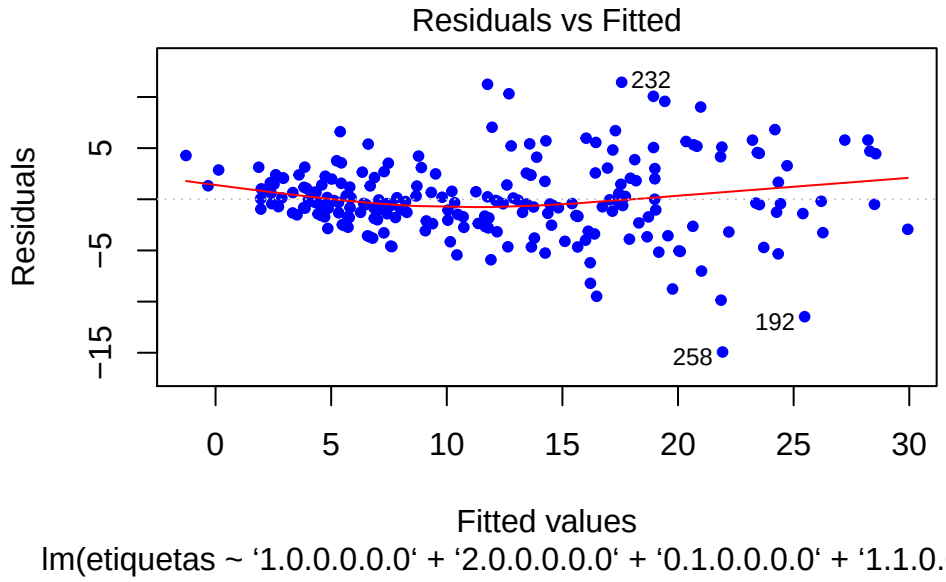
```
# Calculamos los nuevos conjuntos de fórmulas a partir de las nuevas características
max_car_poly = ncol(o_procesados_poly)-1
max_car_sin_pca_poly = ncol(o_procesados_sin_pca_poly)-1
formulas_poly = subconjuntos_formulas(o_procesados_poly,max_car_poly)$formulas
formulas_sin_pca_poly = subconjuntos_formulas(o_procesados_sin_pca_poly,
                                              max_car_sin_pca_poly)$formulas

# Ajustamos todos los modelos
ajustes_poly = mapply(evalua_lm,formulas_poly,
                      MoreArgs = list(datos = o_procesados_poly,
                                      subconjunto = o_indexes_train))
ajustes_sin_pca_poly = mapply(evalua_lm,formulas_sin_pca_poly,
                              MoreArgs = list(datos = o_procesados_sin_pca_poly,
                                              subconjunto = o_indexes_train))
```

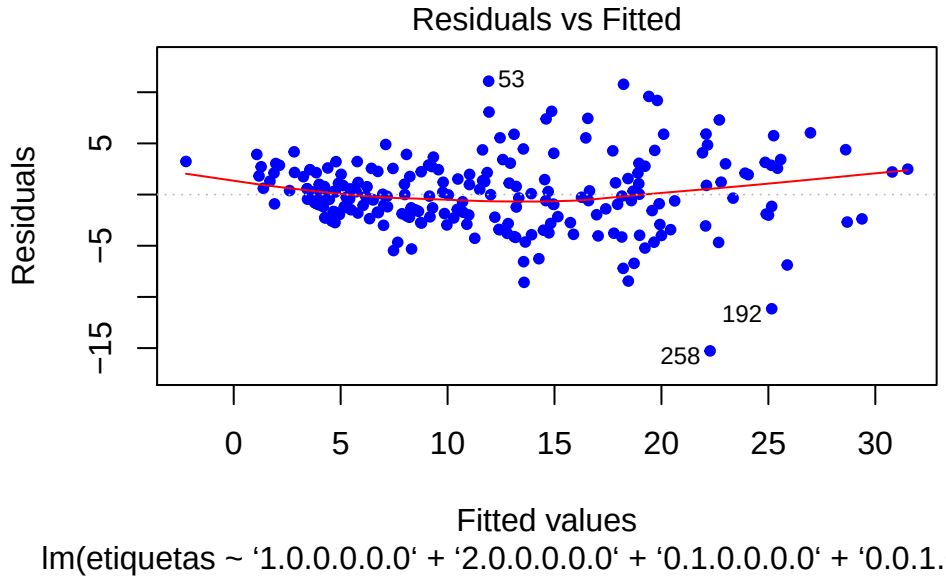
Veamos cómo quedan los errores de estos nuevos modelos frente a los que no tenían transformaciones no lineales:



Es evidente que hemos logrado mejorar el ajuste añadiendo polinomios de orden dos a nuestros conjuntos de predictores. Como consecuencia de esto, las gráficas de errores residuales de los mejores modelos (marcados en verde en las gráficas anteriores) se deberían ver un poco diferentes:



Vemos que, tanto en la gráfica correspondiente al modelo con análisis de componentes principales (arriba) como en la correspondiente al modelo sin dicho análisis (abajo), la curva se ha suavizado; esto nos indica que el modelo ajusta mejor los datos. Por ello, los errores mínimos que obtenemos son, respectivamente, 14.2961 y 15.5141.



Una vez mejorado el modelo mediante transformaciones, hemos de pasar al siguiente problema. Como ya comentábamos, ambos modelos presentan heterocedasticidad, que es una varianza no constante a lo largo de un conjunto (en este caso, los errores). Una forma de remediar esto es homogeneizar los valores de las etiquetas aplicándoles una función de tipo logaritmo o raíz cuadrada, de forma que las variaciones más extremas se vean atenuadas en mayor medida que las variaciones más pequeñas; esto nos dará en la práctica una varianza más reducida. Vamos a solucionar también este problema.

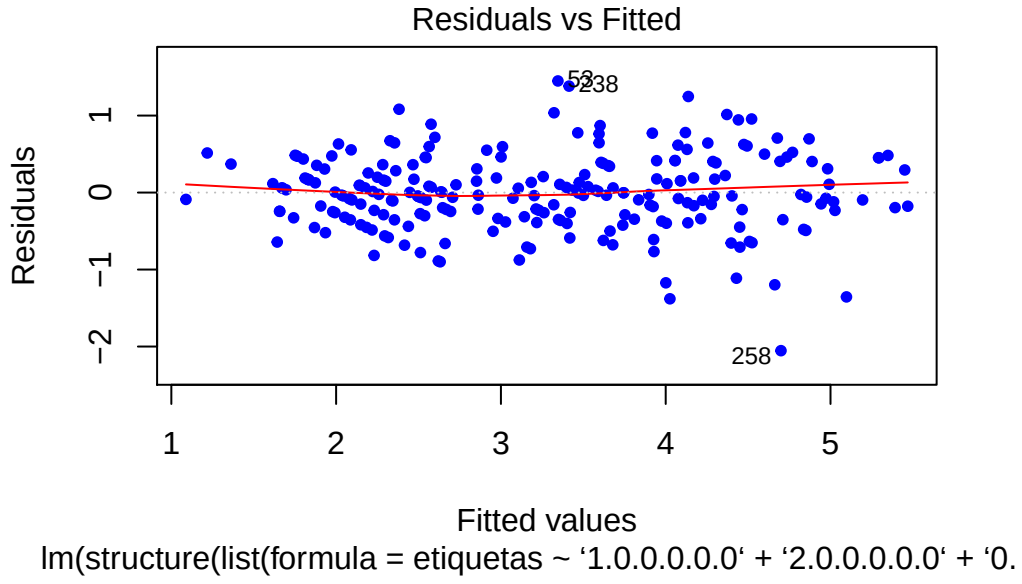
Para ello, aplicaremos la raíz cuadrada a las etiquetas antes de ajustar los dos mejores modelos en base a ellas:

```
o_procesados_poly[,ncol(o_procesados_poly)] =
    sqrt(o_procesados_poly[,ncol(o_procesados_poly)])
o_procesados_sin_pca_poly[,ncol(o_procesados_sin_pca_poly)] =
    sqrt(o_procesados_sin_pca_poly[,ncol(o_procesados_sin_pca_poly)])
```

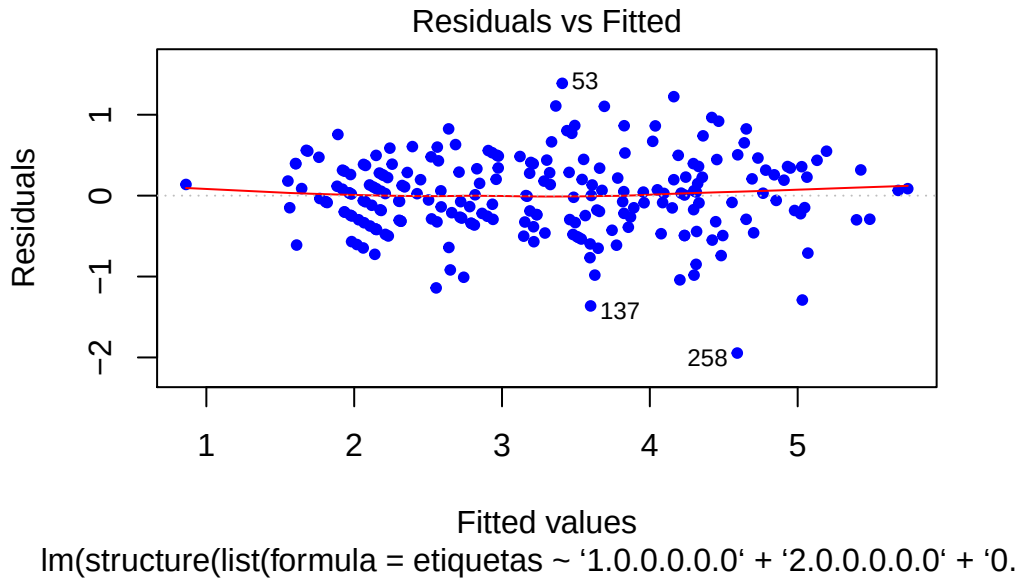
De la misma forma que hemos aplicado una transformación logarítmica a las etiquetas, las predicciones estarán también en una escala logarítmica. Para encontrar las predicciones reales y poder calcular el error cuadrático verdadero debemos realizar el proceso inverso, es decir, una transformación exponencial:

```
error_sqrt_mejor_modelo_poly = error_cuadratico_medio(etiquetas_sqrt_mejor_modelo_poly^2,
                                                      o_labels[-o_indexes_train])
error_sqrt_mejor_modelo_sin_pca_poly =
    error_cuadratico_medio(etiquetas_sqrt_mejor_modelo_sin_pca_poly^2,
                           o_labels[-o_indexes_train])
```

Las gráficas de errores residuales ahora muestran los puntos distribuidos de manera más uniforme:



Como se puede observar en ambas gráficas, la varianza ya no se comporta de forma diferente según la zona de la población en la que nos fijemos; esto quiere decir que la heterocedasticidad ya no ocurre.



En términos de error cuadrático medio, esto se traduce en 13.7345 para el modelo con PCA y en 14.8126 para el modelo sin PCA. Hay que destacar la ligera mejora respecto a los modelos originales con heterocedasticidad; además, las estimaciones proporcionadas tras estas transformaciones deberían ser estadísticamente más fiables ya que una de las suposiciones del cálculo de regresiones lineales es que la varianza se mantiene uniforme en todo el conjunto de datos.

Aun obteniendo una buena tasa de error, teniendo en cuenta que hemos utilizado polinomios de grado dos, vamos a probar la regularización para ver si nos ofrece algún beneficio. Emplearemos *weight decay* con el mejor modelo calculado hasta el momento.

Con este propósito reutilizaremos el código asociado de la sección de clasificación:

```
# Creamos la matriz de datos en el formato que necesita glmnet
o_x = model.matrix(mejor_formula_pca_poly$formula,o_procesados_poly)[,-ncol(o_procesados_poly)]
o_y = o_procesados_poly$etiquetas
# Obtenemos los errores de validación cruzada en el conjunto
o_cv.out = cv.glmnet(o_x[o_indexes_train,],o_y[o_indexes_train],alpha=0)
# Guardamos el lambda que ha dado menor error de validación cruzada
o_bestlambda = o_cv.out$lambda.min
# Obtenemos un modelo de regresión ridge
o_modelo_ridge = glmnet(o_x,o_y,alpha=0,lambda=o_bestlambda)
# Calculamos las predicciones y el error asociado a ellas
o_modelo_ridge.pred = predict(o_modelo_ridge,s=o_bestlambda,newx=o_x[-o_indexes_train,])
o_error_ridge =
    error_cuadratico_medio(o_modelo_ridge.pred^2,
                           o_procesados_poly[-o_indexes_train,ncol(o_procesados_poly)])
```

El error cuadrático medio es de 97.5945, así que podemos descartar la regularización en nuestra búsqueda del mejor modelo final. Esto es, por otra parte, posiblemente debido a que ya habíamos determinado la complejidad más adecuada para ajustar los datos en base a los patrones detectados en los errores residuales. La regularización es apta para situaciones en las que la complejidad de la clase de funciones utilizada es excesiva, lo que puede provocar problemas de generalización por sobreajuste. No obstante, teniendo ya una clase de funciones suficientemente general, una simplificación de la misma produce una situación de *underfitting* o infraajuste.

2.5. Descripción del modelo final.

El modelo que vamos a elegir como definitivo es el derivado de realizar transformaciones no lineales polinómicas de grado dos a los datos (escalados, centrados y con PCA respecto al conjunto de entrenamiento de este problema) y de raíz cuadrada a las etiquetas; para hacer nuevas predicciones sería necesario aplicar las mismas transformaciones. Recordemos que este modelo hace uso de 17 predictores para obtener un error medio cuadrático (*MSE*) de 13.7345, que significa que en promedio nos equivocamos en 3.7 puntos en la escala de la variable de respuesta.

Respecto al error dentro de la muestra, E_{in}, obtenemos un valor de 13.4503, lo que pone de manifiesto que el conjunto de entrenamiento es suficientemente representativo de la población. Lo hemos obtenido mediante el siguiente código:

```
o_etiquetas_train = evaluar_regresion(sqrt_mejor_modelo_poly$reg,
                        o_procesados_poly[o_indexes_train,-ncol(o_procesados_poly)])
o_error_mejor_reg = error_cuadratico_medio(unlist(o_etiquetas_train)^2,
                                           o_labels[o_indexes_train])
```

Una cota para E_{out} es la basada en E_{test}. Procediendo equivalentemente a la sección de clasificación, definimos esta cota gracias a la desigualdad de *Hoeffding*.

Dicha desigualdad nos da un valor de $\epsilon = 0.1364$ y, consecuentemente, un intervalo para E_{out} de $13.7345 \pm 0.1364 = [13.598, 13.8709]$.

A continuación se muestran los pesos asociados a los predictores que caracterizan el modelo elegido:

Característica	Peso	Característica	Peso
$PC1$	94.06544	$PC1 \cdot PC4$	241.89806
$PC1^2$	14.74783	$PC3 \cdot PC4$	-348.44470
$PC2$	24.50623	$PC4^2$	-15.64003
$PC1 \cdot PC2$	542.79954	$PC1 \cdot PC5$	224.02592
$PC2^2$	-21.53054	$PC2 \cdot PC5$	-185.91827
$PC3$	25.39582	$PC2 \cdot PC6$	200.61627
$PC1 \cdot PC3$	447.12311	$PC3 \cdot PC6$	165.04666
$PC3^2$	-35.49378	$PC6^2$	-16.43187
$PC4$	15.25194	Intercept	11.67902

3. Apéndice.

3.1. Funciones auxiliares.

Aquí se listan algunas funciones que no se han tratado explícitamente a lo largo de la memoria por encapsular tareas sencillas auxiliares cuyos detalles de implementación no son imprescindibles para comprender el contenido.

3.1.1. Cálculo del porcentaje de error.

```
porcentaje_error = function(clasificados,reales,fp=1,fn=1){

    reales[reales == 0] = -1
    t = table(clasificados,reales)
    total_predicciones = sum(t)
    t[1,2] = t[1,2]*fn
    t[2,1] = t[2,1]*fp
    100*(1-sum(diag(t))/total_predicciones)

}
```

3.1.2. Cálculo del error cuadrático medio.

```
error_cuadratico_medio = function(clasificados,reales){

    mean((clasificados-reales)^2)

}
```

3.1.3. Clasificar en dos categorías según un umbral.

```
categorizar = function(clasificados,umbral=0.5){

    clasificados[clasificados < umbral] = -1
    clasificados[clasificados >= umbral] = 1
    clasificados

}
```

3.1.4. Predecir etiquetas para unos datos con una regresión lineal.

```
evaluar_regresion = function(regresion,datos){

    predict(regresion,datos) # Los datos no deben incluir las etiquetas

}
```

3.2. Funcionamiento de algunas transformaciones de preprocesamiento.

Para mostrar el funcionamiento de ciertos métodos de preprocesamiento utilizados en este documento, vamos a generar aleatoriamente unos datos de prueba de la siguiente forma:

```
edad = sample(18:90,50)
salario_anual = sample(10000:80000,50)
estatura = sample(140:200,50)
data_matrix = cbind(edad,salario_anual,estatura)

print(head(data_matrix))
```

```
##      edad salario_anual estatura
## [1,]   45         12277      165
## [2,]   42         50062      191
## [3,]   64         30373      140
## [4,]   57         68947      174
## [5,]   21         44272      178
## [6,]   46         49331      200
```

El código proporcionado para estas demostraciones no pretende tener una eficiencia óptima sino ser ilustrativo.

3.2.1. Centrado.

El centrado sirve para uniformizar la media de todas las características. El proceso es simple: habrá que calcular la media de cada una de ellas y restársela a cada instancia particular:

```
media_edad = mean(edad)
media_salario_anual = mean(salario_anual)
media_estatura = mean(estatura)

data_matrix[,1] = data_matrix[,1] - media_edad
data_matrix[,2] = data_matrix[,2] - media_salario_anual
data_matrix[,3] = data_matrix[,3] - media_estatura

print(head(data_matrix))
```

```
##        edad salario_anual estatura
## [1,]  -7.88     -33524.72    -4.94
## [2,] -10.88       4260.28    21.06
## [3,]  11.12     -15428.72   -29.94
## [4,]   4.12      23145.28     4.06
## [5,] -31.88      -1529.72     8.06
## [6,]  -6.88       3529.28    30.06
```

```
print(mean(data_matrix[,1]))
```

```
## [1] -2.55726e-15
```

```
print(mean(data_matrix[,2]))
```

```
## [1] -1.164056e-12
```

```
print(mean(data_matrix[,3]))
```

```
## [1] 2.272844e-15
```

Podemos observar que, en efecto, la media es uniforme ya que ronda un valor muy cercano a 0 en todos los casos.

3.2.2. Escalado.

El escalado sirve para uniformizar la varianza de las variables independientes y así evitar que los algoritmos que calculan modelos favorezcan las características con mayores rangos. Para llevar a cabo un escalado, obtenemos la desviación típica de cada predictor y dividimos cada instancia del mismo entre ella.

```
sd_edad = sd(edad)
sd_salario_anual = sd(salario_anual)
sd_estatura = sd(estatura)

data_matrix[,1] = data_matrix[,1] / sd_edad
data_matrix[,2] = data_matrix[,2] / sd_salario_anual
data_matrix[,3] = data_matrix[,3] / sd_estatura

print(head(data_matrix))
```

```
##            edad salario_anual   estatura
## [1,] -0.3919726   -1.75793622 -0.2743855
## [2,] -0.5412007    0.22339636  1.1697488
## [3,]  0.5531390   -0.80903601 -1.6629762
## [4,]  0.2049400    1.21366938  0.2255071
## [5,] -1.5857977   -0.08021395  0.4476816
## [6,] -0.3422299    0.18506491  1.6696414
```

```
print(sd(data_matrix[,1]))
```

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

```
print(sd(data_matrix[,2]))
```

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

```
print(sd(data_matrix[,3]))
```

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

De nuevo comprobamos que la técnica ha conseguido su objetivo: la desviación típica de todas las características es 1.

Finalmente, como consecuencia natural, la media de toda la matriz de datos tiende a 0 y la desviación típica a 1:

```
print(mean(data_matrix))
```

```
## [1] -2.321598e-17
```

```
print(sd(data_matrix))
```

```
## [1] 0.9932659
```

La desventaja de aplicar técnicas de preprocesado es que las características pierden mucha interpretabilidad, como hemos podido observar a lo largo del proceso. Por otro lado, la ventaja es que muchos modelos suponen una distribución normal de los datos, por lo que estamos adecuándolos para un correcto funcionamiento.