PROYECTO FINAL: AJUSTE DE MODELOS NO-LINEALES

Nuria Rodríguez Barroso, Juan Luis Suárez Díaz.

09 de junio de 2017

Contents

Comprensión del problema a resolver: Human Activity Recognition Using Smartphones Data Set. 2
Preprocesado de datos. 2
Analisis de componentes principales (PCA) 4
Regresión LASSO (least absolute shrinkage and selection operator). 5

Modelo lineal. **5**

Redes Neuronales **6**

Máquina de Vectores Soporte **8**

Boosting **9**

RandomForest **10**

Análisis de los resultados y conclusiones **13**

Comprensión del problema a resolver: Human Activity Recognition Using Smartphones Data Set.

La base de datos elegida para el problema recoge la información de un grupo 30 voluntarios con edades comprendidas entre 18-48 años. Cada uno de estos individuos tenía que realizar una serie de actividades con el teléfono móvil enganchado en su cintura. Así, utilizando el dispositivo se podían registrar la aceleración lineal y angular en 3-ejes. Los experimentos se graban en vídeo para poder etiquetar los datos de forma manual. De entre los datos recogidos, se seleccionaron de forma aleatoria el 70% de estos para hacer de conjunto de training mientras que el otro 30% se seleccionaron para el test. Así, la base de datos se compone de 7352 muestras en el conjunto de train y 2947 en el conjunto de test, formando un total de 10299 muestras recogidas de estos 30 voluntarios y se consideran un total de 561 atributos.

Observamos el resumen de los diez primeros atributos:

En cuanto a las etiquetas, nos encontramos ante un problema de clasificación multiclase, dado que las etiquetas toman valores enteros en el intervalo [1,6]. Cada etiqueta clasifica el movimiento registrado por el dispositivo móvil y lo clasifica en seis tipos: 1. WALKING 2. WALKING_UPSTAIRS 3. WALKING_DOWNSTAIRS 4. SITTING 5. STANDING 6. LAYING

Podemos observar fijándonos en lo que representa cada etiqueta que los tres primeros valores de las etiquetas hacen referencia a actividades muy similares, relacionadas con andar, mientras que las otras tres tienen la misma propiedad, representando estados de no movimiento. Podemos intuir por tanto, en vista a lo que representan las distintas clases, que los datos de las tres primeras clases serán fácilmente separables de los datos de las tres restantes, si las medidas que se han tomado son representativas del movimiento, mientras que a priori será más complicado distinguir dentro de cada grupo de clases, cuál será la asignación correcta.

Al tratarse de un problema de clasificación multiclase, para la aplicación de aquellos métodos que precisen de clasificación binaria para un buen funcionamiento utilizaremos la técnica de *One vs One*, la cual consistirá en dividir nuestro problema de clasificación multiclase en $\binom{6}{2} = 15$ problemas de clasificación binaria de la forma clase_i vs clase_j con $i, j \in \{1, \ldots, 6\}$,

Así, el cálculo del error para cada modelo propuesto consistirá en el número de errores cometidos. Esto es, número de muestras con etiqueta asignada de forma errónea. Para aquellos modelos que precisen de una adaptación a subproblemas de clasificación binaria, obtendremos quince vectores con las etiquetas predichas por cada modelo de 1vs1, y finalmente asignaremos como etiqueta final a cada dato el valor más votado entre los 15 vectores. A partir de aquí calcularemos el de la misma forma que para el resto de modelos.

Preprocesado de datos.

En primer lugar, observamos que nuestra base de datos no contiene valores perdidos (NaNs), por lo cual no será necesario el trato especial de estos valores. Lo mismo ocurre con las variables cualitativas.

En cuanto al desbalanceo, observemos el porcentaje de elementos de cada clase en el train y después en el test:

```
## [1] "El porcentaje de elementos con etiqueta 1 en el train es: "
## [1] 16.67573
## [1] "El porcentaje de elementos con etiqueta 2 en el train es: "
## [1] 14.59467
## [1] "El porcentaje de elementos con etiqueta 3 en el train es: "
## [1] 13.41132
## [1] "El porcentaje de elementos con etiqueta 4 en el train es: "
## [1] 17.49184
```

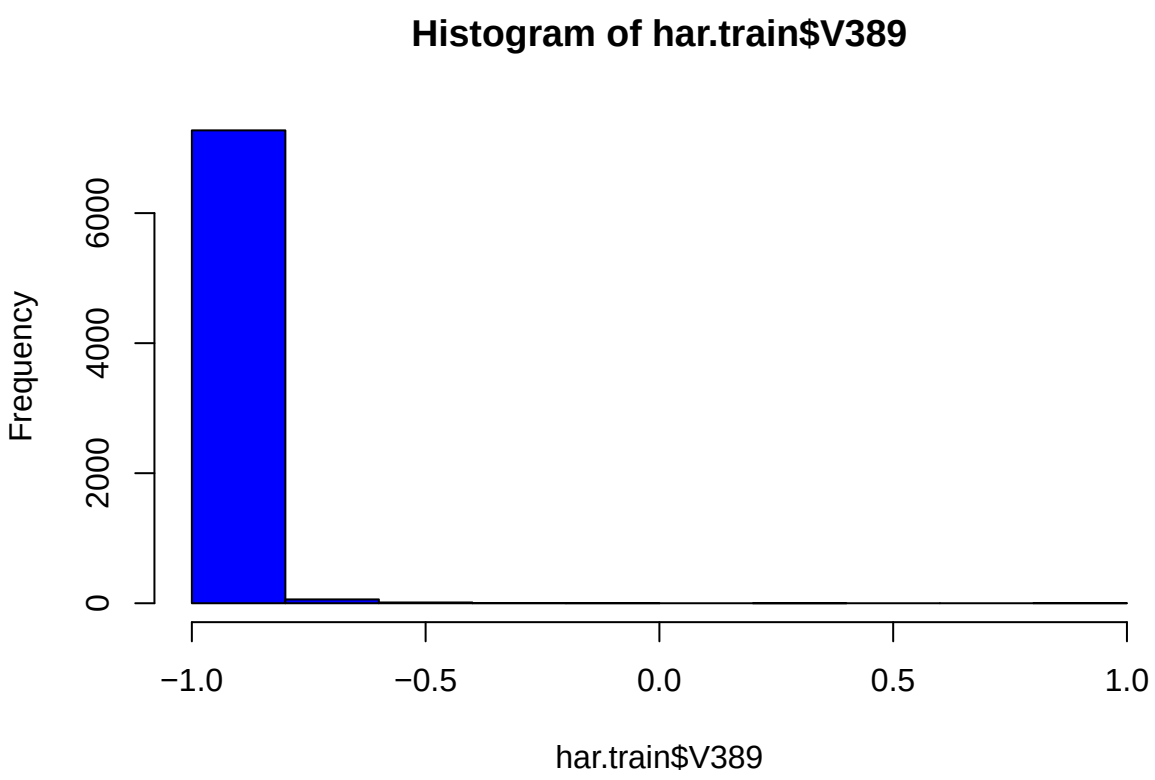
```
## [1] "El porcentaje de elementos con etiqueta 5 en el train es: "
## [1] 18.68879
## [1] "El porcentaje de elementos con etiqueta 6 en el train es: "
## [1] 19.13765
## [1] "El porcentaje de elementos con etiqueta 1 en el test es: "
## [1] 16.83068
## [1] "El porcentaje de elementos con etiqueta 2 en el test es: "
## [1] 15.98235
## [1] "El porcentaje de elementos con etiqueta 3 en el test es: "
## [1] 14.25178
## [1] "El porcentaje de elementos con etiqueta 4 en el test es: "
## [1] 16.66101
## [1] "El porcentaje de elementos con etiqueta 5 en el test es: "
## [1] 18.05226
## [1] "El porcentaje de elementos con etiqueta 6 en el test es: "
## [1] 18.22192
```

Podemos observar que no existe desbalanceo entre las seis clases contempladas, luego no podemos despreciar ninguna.

Aunque los datos están ya normalizados y escalados en [-1,1], estos presentan una gran asimetría como podemos observar:

```
##     V389     V479      V60      V47      V44     V485
## 14.70005 12.33718 12.18477 11.05100 10.70115 10.30984
```

Si representamos el atributo con mayor asimetría de todos obtenemos:



Aunque los datos presentan una gran asimetría, no los quitamos pues las transformaciones son realizadas atributo a atributo y modifican las dependencias entre ellos pudiendo empeorar los métodos empleados.

En cuanto a la presencia de ruido, los datos son clasificados por especialistas, luego es razonable pensar que no hay etiquetas mal colocadas (no hay ruido en las etiquetas). Por otra parte, los datos tomados provienen de medidas en un SmartPhone, que según su calidad, dichas medidas pueden tener distinta precisión, así que, aunque no podemos descartar que haya algo de ruido en los datos, con el avance de la tecnología los instrumentos de medida son cada vez más precisos, por lo que la presencia de ruido, si hay, parece ser pequeña.

La base de datos presenta un número demasiado alto de atributos, lo cual puede ser un gran problema a la hora utilizar métodos costosos para la predicción de las etiquetas. Con el objetivo de reducir el número de atributos sin perder información relevante, vamos a probar con dos métodos:

Analisis de componentes principales (PCA)

En primer lugar, probaremos a aplicar PCA a los datos con el objetivo de encontrar dependencias entre las variables y simplificar el conjunto de datos reduciendo los tiempos de cómputo. Los nuevos atributos considerados tras aplicar este método son combinaciones lineales de los atributos originales que recogen un tanto por ciento de la varianza acumulada de los datos pasado como argumento (*thresh*). Para la aplicación vamos a considerar *thresh = 0.95 ...*

Utilizaremos la función *preProcess* pasándole como argumento *method = c("pca").*

```
## Created from 7352 samples and 561 variables
##
## Pre-processing:
##   - centered (561)
```

```
##   - ignored (0)
##   - principal component signal extraction (561)
##   - scaled (561)
##
## PCA needed 102 components to capture 95 percent of the variance
```

Para mantener el 95% de la varianza se necesitan mantener 102 atributos.

Regresión LASSO (least absolute shrinkage and selection operator).

Lasso es un método que lleva a cabo la regularización a la misma vez que realiza selección de características.

El objetivo se basa en reducir el error de predicción, para ello, se ocupa de reducir la función:

$$R(\beta) = \sum_{i=1}^{n}(y_i - x_i\omega)^2 + \lambda \sum_{i=1}^{p} |\omega_i|$$

donde n es el número de muestras, p el número de atributos y w el vector de pesos solución. Así, obtiene diferentes valores de λ. Entre estos valores devueltos, podemos considerar dos de ellos:

- *lambda.min*, que nos devuelve el valor del λ para el que se minimiza el error obtenido.
- *lambda.1se*. Aunque el λ anterior sea menor, presenta una mayor dependencia de las particiones seleccionadas en la validación cruzada. La regla `1se` (one standard error), elige un valor de λ para lo suficiente cercano a `lambda.min` para el cual el modelo obtenido sea lo suficiente simple, reduciendo así la dependencia del error respecto a las validaciones.

Podemos contemplar cualquiera de estos valores de λ para regularizar nuestro modelo.

Para obtener el modelo LASSO, realizaremos validación cruzada con tres particiones (dada la elevada dimensión de la base de datos). Tras obtener el modelo, considerando cada uno de los valores de *lambda* anteriormente comentado, obtenemos seis vectores de coeficientes donde cada uno de las componentes de estos vectores representa la relevancia de cada atributo para cada clase. Así, los atributos de los cuales podremos prescindir serán aquellos en los que la componente correspondiente a dicho atributo en todos los vectores sea nula. Realizando estas simplificaciones obtenemos que tras simplificar atributos las dimensiones obtenidas son:

```
## [1] "El número de atributos tras aplicar la reducción con lambda.min es: "
```

```
## [1] 172
```

```
## [1] "El número de atributos tras aplicar la reducción con lambda.1se es: "
```

```
## [1] 188
```

Por tanto, aunque sería preferible aplicar una reducción de atributos basada en la regresión LASSO dado que los atributos obtenidos se corresponden con los atributos originales del modelo, obtenemos que tras comparar las tres reducciones de dimensionalidad la que mayor reducción produce es la aplicación de *PCA*. En conclusión, a partir de ahora trabajaremos con el conjunto de datos resultante de aplicar la reducción con *PCA*.

Modelo lineal.

Vamos a ajustar un modelo de regresión logística como modelo lineal para los datos observados. Como ya hemos comentado, tenemos 6 etiquetas distintas y aplicaremos la técnica de clasificación 1 vs 1 para descomponer el problema en problemas binarios.

Los resultados obtenidos son:

```
## [1] "Etest obtenido por el modelo lineal:"
```

```
## [1] 0.08720733
```

```
## [1] "Matriz de consusión obtenida por el modelo lineal:"
```

```
##
##       1   2   3   4   5   6
##   1 472  59   9   0   0   0
##   2   2 386  37   2   0   0
##   3  22  26 374   0   0   0
##   4   0   0   0 426  37   0
##   5   0   0   0  63 495   0
##   6   0   0   0   0   0 537
```

Como podemos observar, el modelo lineal ya produce unos resultados bastante buenos, con un error de clasificación solo de un 8 %. Además, hemos obtenido este error sin necesidad de realizar ninguna transformación en las características. Esto nos indica que los datos, una vez aplicadas las transformaciones realizadas, presentan un comportamiento lineal, que ha sido bien aprovechado por la regresión logística. Los distintos modelos no lineales que aplicaremos tendrán que intentar reducir este error de clasificación, que ya es bastante aceptable.

Redes Neuronales

Las redes neuronales son un modelo formado por una estructura compleja con conexiones entre las distinas unidades o neuronas, y con una gran capacidad de aprendizaje. Sin embargo, la complejidad de las redes neuronales conduce también a distintos problemas. Por una parte, es necesario elegir de forma acertada el número de capas sobre el que aprender nuestros datos. De no hacerlo, un número demasiado alto de capas puede conducirnos a sobreaprender. Por otra parte, el proceso de aprendizaje en una red neuronal es costoso computacionalmente.

Debido a las dimensiones de nuestra base de datos, formular una arquitectura de red neuronal con más de una capa o con muchas unidades en la capa oculta se hace inmanejable. Además, si consideramos más de una capa oculta el número de conexiones puede dispararse, pudiendo obtener, para un número suficientemente grande de neuronas por capa, un número de pesos en torno al número de instancias del problema, luego apenas tendríamos un dato por peso durante el aprendizaje. Por ello, definimos nuestro modelo de red neuronal por una arquitectura con una única capa oculta y con un número fijo de unidades en la capa oculta igual a 5. Utilizaremos la función `neuralnet` proporcionada por R, de la que destacamos los siguientes parámetros:

- **hidden:** Número de capas ocultas (hidden = 5).
- **threshold:** Criterio de parada. Umbral entre las derivadas parciales de la función de error en dos iteraciones consecutivas. Lo dejaremos por defecto, 0.1.
- **stepmax:** Número máximo de pasos. Estableceremos también este criterio de parada para evitar quedar atrapados en mínimos locales cuando usamos valores de tasa de aprendizaje muy bajas.
- **rep:** Número de repeticiones que realizamos del aprendizaje de los pesos a partir de los datos de train. Realizaremos tres repeticiones para cada valor de la tasa de aprendizaje.
- **learningrate:** Tasa de aprendizaje. Será el valor a estimar y tomará valores entre [0.8, 1.2]. Dependiendo de este valor, los saltos entre los pesos serán más o menos pronunciados. Se establecerá con un único valor en el intervalo dependiendo del ganador tras la estimación.
- **lifesign:** Para establecer lo que queremos que imprima la función durante el aprendizaje de la red neuronal. Lo utilizaremos con dos valores diferentes: 'minimal' para el cálculo de la tasa de aprendizaje óptima y 'full' cuando analicemos más profundamente.
- **err.fct:** Función de error usada para el cálculo del error. Usaremos la función por defecto, `sse`: suma de errores cuadráticos.
- **act.fct:** Función de activación utilizada en las neuronas. Usaremos la función por defecto: la función logística.

- **linear.output:** Lo pondremos a *FALSE* para que interprete el problema como un problema de clasificación.

Como ya hemos comentado, para establecer el modelo calcularemos la tasa de aprendizaje óptima para la arquitectura de una capa con cinco unidades en la capa:

```
## hidden: 5    thresh: 0.01    rep: 1/3    steps:    7853  error: 13.01818 time: 1.22 mins
## hidden: 5    thresh: 0.01    rep: 2/3    steps: stepmax  min thresh: 0.03587812901
## hidden: 5    thresh: 0.01    rep: 3/3    steps:    1134  error: 65.98646 time: 10.43 secs
## hidden: 5    thresh: 0.01    rep: 1/3    steps: stepmax  min thresh: 0.03391593212
## hidden: 5    thresh: 0.01    rep: 2/3    steps:    8872  error: 27.02617 time: 1.37 mins
## hidden: 5    thresh: 0.01    rep: 3/3    steps:    1493  error: 39.00314 time: 14.8 secs
## hidden: 5    thresh: 0.01    rep: 1/3    steps:    4608  error: 17.40283 time: 42.86 secs
## hidden: 5    thresh: 0.01    rep: 2/3    steps:    4018  error: 21.70001 time: 37.35 secs
## hidden: 5    thresh: 0.01    rep: 3/3    steps: stepmax  min thresh: 0.02968517139
## hidden: 5    thresh: 0.01    rep: 1/3    steps:    5941  error: 20.50604 time: 55.36 secs
## hidden: 5    thresh: 0.01    rep: 2/3    steps:    1332  error: 30.39057 time: 12.25 secs
## hidden: 5    thresh: 0.01    rep: 3/3    steps: stepmax  min thresh: 0.04018070282
## hidden: 5    thresh: 0.01    rep: 1/3    steps:    4201  error: 32.95402 time: 39.09 secs
## hidden: 5    thresh: 0.01    rep: 2/3    steps: stepmax  min thresh: 0.1277930806
## hidden: 5    thresh: 0.01    rep: 3/3    steps: stepmax  min thresh: 0.01874093181
```

Los errores obtenidos en las diferentes pruebas han sido:

```
##               err.tasa
## [1,] 0.8 0.05487528345
## [2,] 0.9 0.05668934240
## [3,] 1.0 0.06802721088
## [4,] 1.1 0.05124716553
## [5,] 1.2 0.05804988662
```

Luego, el mejor valor para la tasa de aprendizaje es $\eta = 1.1$. Definimos el modelo con esta tasa de aprendizaje.

```
## hidden: 5    thresh: 0.01    rep: 1/1    steps:    1000  min thresh: 0.3289771287
##                                                  2000  min thresh: 0.2536274022
##                                                  2625  error: 98.50664 time: 35.38 secs
```

```
## [1] "Etest obtenido por la red neuronal:"
```

```
## [1] 0.1143535799
```

```
## [1] "Matriz de confusión obtenida en la red neuronal:"
```

```
##
## pr.nn_2   1   2   3   4   5   6
##       1 453  87   6   0   0   0
##       2   7 358  37   1   0   0
##       3  36  23 376   0   0   0
##       4   0   0   0 409  55   0
##       5   0   2   0  73 477   0
##       6   0   1   1   8   0 537
```

Observamos que el error obtenido es mayor que el obtenido con el modelo lineal. Una de las razones es que, debido a la complejidad computacional, no hemos podido probar modelos con mayor número de unidades, y por tanto es posible que esta no sea la mejor estructura de red neuronal para el problema. Empleando un mayor tiempo de cómputo podría ser posible reducir el error, pero en tal caso habría que decidir si la reducción del error compensa el tiempo empleado, en comparación con otros modelos que hayan sido más eficientes y con errores de clasificación cercanos, como el lineal.

Máquina de Vectores Soporte

Las máquinas de vectores soporte son una de las técnicas más utilizadas para aprender distintos conjuntos de datos, por su simplicidad, su capacidad de maximizar márgenes y su facilidad para actuar en clasificaciones no lineales usando funciones kernel. Suelen proporcionar mejores resultados cuando los datos se reparten de forma homogénea y con poco ruido, permitiendo así controlar los márgenes de separación a partir de un conjunto pequeño de datos, los vectores soporte. Si el conjunto de datos es más heterogéneo, el ajuste será más complicado, puesto que la cantidad de vectores soporte puede aumentar, y serán más los vectores a adaptar.

En nuestro caso, ya intuíamos previamente que los datos de las clases 1,2 y 3 pueden ser a priori fácilmente distinguibles de los de las clases 4,5 y 6, mientras que dentro de cada grupo de clases la separabilidad de los datos puede ser más complicada. Los vectores soporte es posible que se concentren en torno a los márgenes internos a cada grupo de clases. Según la capacidad de separación que tengan para los datos medidos los resultados para SVM serán mejores o no.

Vamos a elegir por validación cruzada 5-fold el modelo más adecuado, fijando un kernel RBF-Gaussiano y probando distintos valores para el hiperparámetro. Los valores que probaremos para el parámetro γ serán 0.01, 0.1, 1 y 10. Los errores de validación obtenidos son:

```
## [1] "Resultados de tune sobre SVM:"
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 5-fold cross validation
##
## - best parameters:
##  gamma
##   0.01
##
## - best performance: 0.03536462308
```

```
## [1] "Errores obtenidos para cada parámetro en la validación cruzada:"
```

```
##   gamma         error     dispersion
## 1  0.01 0.03536462308 0.001597748161
## 2  0.10 0.64132780236 0.038404535914
## 3  1.00 0.81392805117 0.006764805437
## 4 10.00 0.81392805117 0.006764805437
```

Aprendemos el mejor modelo de SVM obtenido en la validación cruzada, es decir, para $\gamma = 0,01$:

```
##
## Call:
## svm.default(x = har.train, y = factor(lhar.train[, 1]), kernel = "radial",
##     gamma = 0.01)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##       gamma:  0.01
##
## Number of Support Vectors:  4255
##
##  ( 656 707 710 733 745 704 )
```

```
##
##
## Number of Classes:  6
##
## Levels:
##  1 2 3 4 5 6
```

En el resumen anterior podemos ver bastante información sobre el modelo que hemos obtenido. Por un lado, vemos los parámetros que se han utilizado, y por otra parte, vemos el número de vectores soporte obtenidos. Además, en el vector del resumen se indica cuántos vectores soporte son separadores en cada clase. Podemos ver que más de la mitad de los datos son vectores soporte y que el número de vectores soporte se distribuye de forma similar en las distintas clases. Lo primero nos indica que los datos de las distintas clases están separados por márgenes pequeños, y además es una causa de la poca eficiencia del algoritmo en este caso, pues ha requerido de la adaptación de una cantidad considerable de vectores soporte. Lo segundo puede ser consecuencia de que los datos no están desbalanceados, y por tanto la distribución de vectores soporte es similar para cada clase.

Finalmente, evaluamos los datos test para el modelo obtenido.

```
## [1] "Etest obtenido en SVM:"
```

```
## [1] 0.06277570411
```

```
## [1] "Matriz de confusión obtenida en SVM:"
```

```
##
## svm.pred   1   2   3   4   5   6
##        1 475  31   2   0   2   0
##        2   0 419   9   1   0   0
##        3  21  21 409   5   1   3
##        4   0   0   0 430  34   0
##        5   0   0   0  51 495   0
##        6   0   0   0   4   0 534
```

Como vemos, el error obtenido es el menor hasta el momento, superando ligeramente al lineal, aunque el tiempo de cómputo requerido ha sido mayor. También si contamos el tiempo para estimación de parámetros, se ha requerido una cantidad de tiempo considerable. Por otra parte, la minimización del riesgo estructural del SVM nos confirma que hay una separación importante entre por lo menos los grupos de clases mencionados anteriormente, y dicha separación contribuye a disminuir el error.

Boosting

Boosting es una técnica de aprendizaje basada en clasificadores débiles que se combinan para producir buenos resultados. La técnica AdaBoost sigue la mecánica de Boosting de forma iterativa, adaptándo los nuevos clasificadores según los errores obtenidos. Tiene una gran capacidad de ajuste, pero a la vez tiene una gran capacidad para maximizar márgenes, lo que le permite generalizar de forma bastante buena. Sin embargo, puede ser muy sensible al ruido en los datos, y la elección del clasificador simple influye mucho en su correcto funcionamiento.

En nuestro caso, utilizaremos como clasificador funciones stamp, o árboles de decisión de un nivel. A priori no conocemos la cantidad de ruido en los datos, aunque, como ya se ha comentado, es posible que haya algún ruido ligero consecuencia de errores de precisión en las medidas tomadas por los dispositivos. Si este ruido es lo suficientemente destacable, AdaBoost no producirá buenos resultados.

Utilizamos dos funciones de R para obtener estos modelos de Boosting. Por un lado, utilizamos AdaBoost comparando las clases mediante la técnica 1 vs 1 ya realizada con los modelos lineales, y por otra parte,

utilizaremos la librería de R maboost, que proporciona una versión generalizada de AdaBoost para problemas de clasificación con múltiples etiquetas.

En primer lugar aprendemos los distintos modelos para el primer caso, con la función de R `gbm` junto con la técnica 1 vs 1. Los resultados obtenidos son:

```
## [1] "Etest obtenido en AdaBoost - gbm:"
```

```
## [1] 0.1839158466
```

```
## [1] "Matriz de confusión obtenida en AdaBoost - gbm:"
```

```
##
##       1   2   3   4   5   6
##   1 461  51 118   0   0   0
##   2   3 414  42   2   1   0
##   3  32   6 260   0   0   0
##   4   0   0   0 295  64  23
##   5   0   0   0 183 466   5
##   6   0   0   0  11   1 509
```

Vemos que el error es bastante alto, y mucho más alto que todos los obtenidos anteriormente. Como ya hemos comentado, el boosting pude ser muy sensible al ruido en los datos, pero como también hemos comentado, no parece que haya demasiado ruido en nuestro conjunto de datos. Por ello, para comprobar si realmente los malos resultados se deben al ruido, para la siguiente función de boosting, `maboost`, utilizaremos un parámetro de regularización. Con esto, se añade un término de regularización basado en la norma 1 para solucionar el ruido. Aprendemos los datos de entrenamiento con este modelo y los validamos sobre el conjunto test, el error obtenido es:

```
## [1] "Multiclass boosting is selected"
```

```
## [1] "Etest obtenido en maboost:"
```

```
## [1] "Matriz de confusión obtenida en maboost:"
```

Como vemos, el error apenas se ha reducido en un 2 % y sigue siendo bastante grande. Por tanto, es posible que los resultados obtenidos no se deban al ruido, sino a otros factores a los que también es sensible el boosting, como que el clasificador débil elegido (las funciones stamp) no sean adecuadas para el conjunto de datos.

RandomForest

Random Forest es una técnica de aprendizaje basada en clasificadores simples utilizando bagging. Es una técnica con una gran componente aleatoria, la cual permite obtener buenos resultados cuando trabajamos con datos más difusos o con más ruido. También puede ser bastante eficaz para clasificar problemas con múltiples etiquetas, aunque puede necesitar un gran número de ejemplos para que la aleatoriedad del algoritmo permita generalizar bien para la clasificación de nuevos datos.

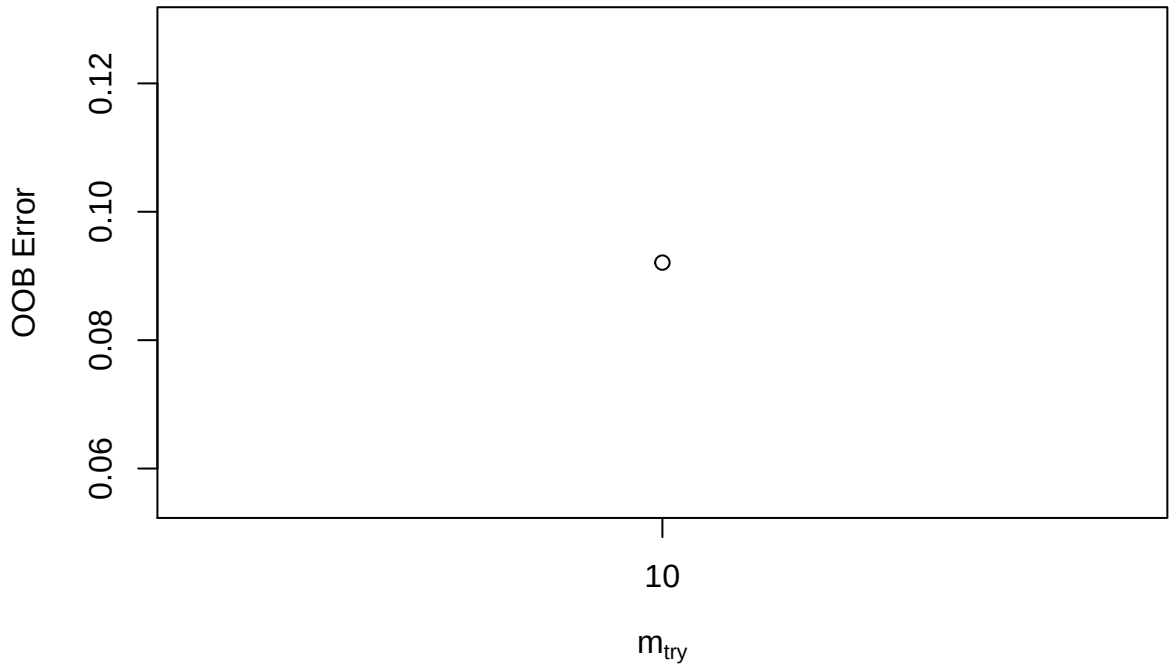
En nuestro caso, disponemos tanto de un problema con múltiples etiquetas como de bastantes ejemplos, por lo que puede funcionar bien. En cuanto a la presencia de ruido, ya hemos visto que no parece haber mucho ruido, salvo el que pueda obtenerse de errores de medida. En cualquier caso, Random Forest es una técnica para la cual los resultados sobre nuestro conjunto de datos pueden ser buenos.

A continuación nos planteamos definir un modelo con RandomForest. Utilizaremos la función `randomForest` proporcionada por R, de la que destacamos los siguientes parámetros:

- **ntree:** Número de árboles considerados. Para estimar este número, realizaremos validación cruzada haciendo uso de la función `tune` proporiconada por R.

- **mtry:** Número de variables elegidas de forma aleatoria como candidatas para cada partición. Según los parámetros visto en teoría, para los problemas de clasificación el valor óptimo sería $sqrt(p)$ donde p sería el número de atributos considerados, por tanto, para nuestro problema *mtry = 10*.

Para comprobar que se corresponde el valor óptimo con el valor establecido teóricamente, utilizamos la función `tuneRF` que nos devuelve el mejor valor de *mtry*.




```
## [1] "El valor óptimo de mtry calculado es: "
```

```
## [1] 10
```

Luego, el valor óptimo calculado del *mtry* se corresponde con el valor óptimo teórico. De aqui en adelante, utilizamos $mtry = 10$.

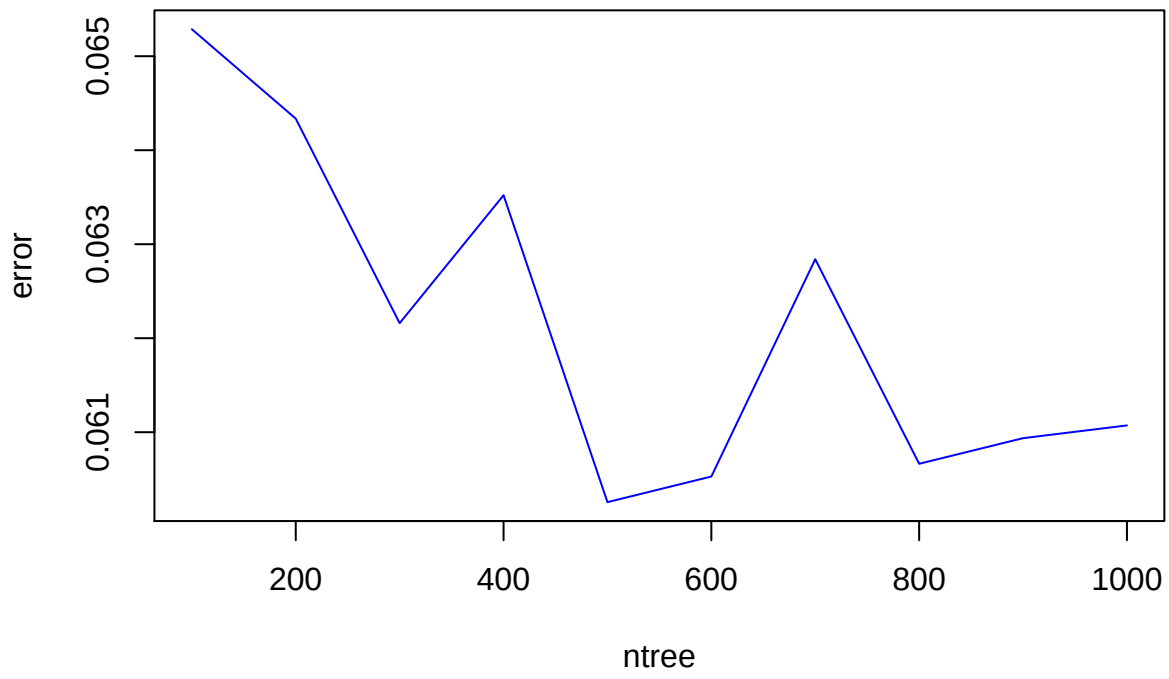
A continuación, experimentamos con el número de árboles óptimo para la definición de nuestro modelo de Random Forest final, consideraremos los valores de número árboles entre [100,1000]. Relizamos validación cruzada (5 folds) con la función tune y obtenemos lo siguiente:

```
##    ntree mtry         error     dispersion
## 1    100   10 0.06528669932 0.007985712592
## 2    200   10 0.06433598320 0.003829239947
## 3    300   10 0.06215975989 0.004472149105
## 4    400   10 0.06352067408 0.006433908427
## 5    500   10 0.06025583041 0.004704918951
## 6    600   10 0.06052840171 0.004121364255
## 7    700   10 0.06284030948 0.004402629302
## 8    800   10 0.06066399367 0.006427406698
## 9    900   10 0.06093628750 0.007799393336
## 10  1000   10 0.06107187947 0.003708439894
```

```
##
## Parameter tuning of 'randomForest':
##
## - sampling method: 5-fold cross validation
##
## - best parameters:
##  ntree mtry
##    500   10
##
## - best performance: 0.06025583041
```

Como podemos ver, el número óptimo de áboles obtenido ha sido 800, aunque el error no ha variado mucho.

Podemos visualizar el progreso del error en función del número de árboles utilizado en la siguiente gráfica:



Aunque podemos observar que las diferencias entre los errores obtenidos son pequeñas, como 800 árboles sigue siendo un número de árboles tratable en cuanto a tiempo de cómputo, elegimos este número como valor del parámetro *ntree*. Así, definimos nuestro modelo ganador de Random Forest con 800 árboles y *mtry = 10*.

Evaluamos sobre los datos test el mejor modelo obtenido en la validación cruzada.

```
##                Length Class  Mode
## call              5  -none- call
## type              1  -none- character
## predicted      7352  factor numeric
## err.rate       5600  -none- numeric
## confusion        42  -none- numeric
## votes         44112  matrix numeric
## oob.times      7352  -none- numeric
```

```
## classes             6  -none- character
## importance        102  -none- numeric
## importanceSD        0  -none- NULL
## localImportance     0  -none- NULL
## proximity           0  -none- NULL
## ntree               1  -none- numeric
## mtry                1  -none- numeric
## forest             14  -none- list
## y                7352  factor numeric
## test                0  -none- NULL
## inbag               0  -none- NULL
```

```
## [1] "Etest obtenido en Random Forest:"
```

```
## [1] 0.1102816423
```

```
## [1] "Matriz de confusión obtenida en Random Forest:"
```

```
##
## rf.pred   1   2   3   4   5   6
##       1 479  32  60   0   0   0
##       2   1 430  41   1   0   0
##       3  16   9 319   0   0   0
##       4   0   0   0 383  33  24
##       5   0   0   0 105 499   1
##       6   0   0   0   2   0 512
```

En este caso, el error obtenido es comparable al obtenido en la red neuronal, siendo de nuevo algo mayor que el lineal. En este caso, el número de datos no es el suficiente como para que los árboles aprendan lo suficiente para saber generalizar en mayor medida.

Análisis de los resultados y conclusiones

Finalmente, comparamos conjuntamente todos los resultados obtenidos.

```
## Regresión Logística        Red neuronal                 SVM
##       0.08720732949       0.11435357991       0.06277570411
##       Random Forest      Adaboost - BGM  Adaboost - MABoost
##       0.11028164235       0.18391584662       0.15167967424
```

```
## $`Regresión Logística`
##
##       1   2   3   4   5   6
##   1 472  59   9   0   0   0
##   2   2 386  37   2   0   0
##   3  22  26 374   0   0   0
##   4   0   0   0 426  37   0
##   5   0   0   0  63 495   0
##   6   0   0   0   0   0 537
##
## $`Red neuronal`
##
## pr.nn_2   1   2   3   4   5   6
##       1 453  87   6   0   0   0
##       2   7 358  37   1   0   0
##       3  36  23 376   0   0   0
```

```
##       4   0   0   0 409  55   0
##       5   0   2   0  73 477   0
##       6   0   1   1   8   0 537
##
## $SVM
##
## svm.pred   1   2   3   4   5   6
##        1 475  31   2   0   2   0
##        2   0 419   9   1   0   0
##        3  21  21 409   5   1   3
##        4   0   0   0 430  34   0
##        5   0   0   0  51 495   0
##        6   0   0   0   4   0 534
##
## $`Random Forest`
##
## rf.pred   1   2   3   4   5   6
##       1 479  32  60   0   0   0
##       2   1 430  41   1   0   0
##       3  16   9 319   0   0   0
##       4   0   0   0 383  33  24
##       5   0   0   0 105 499   1
##       6   0   0   0   2   0 512
##
## $`Adaboost - BGM`
##
##       1   2   3   4   5   6
##   1 461  51 118   0   0   0
##   2   3 414  42   2   1   0
##   3  32   6 260   0   0   0
##   4   0   0   0 295  64  23
##   5   0   0   0 183 466   5
##   6   0   0   0  11   1 509
##
## $`Adaboost - MABoost`
##
## maboost.pred   1   2   3   4   5   6
##            1 464  43  76   0   1   0
##            2   3 418  50   2   0   1
##            3  28   9 293   0   0   0
##            4   1   0   0 372  59  49
##            5   0   1   1 108 472   6
##            6   0   0   0   9   0 481
```

En primer lugar, si nos fijamos en las matrices de confusión obtenidas en todos los modelos evaluados, vemos que todas verifican la propiedad de que las submatrices derecha superior e izquierda inferior tienen la mayoría de sus valores nulos o muy cercanos a 0. Esto nos confirma lo que habíamos intuido previamente, y es que los datos pertenecientes a alguna de las tres primeras clases son fácilmente separables de los datos de las tres últimas clases. Todos los algoritmos son capaces de realizar esta separación salvo pequeñas diferencias en el número de errores.

Por otro lado, tras haber comparado todos los algoritmos obtenemos que SVM es el que mejores resultados ha proporcionado en cuanto a error de clasificación obtenido. Es conocido que SVM tiene una gran capacidad de generalización, y se comprueba efectivamente sobre nuestro conjunto de datos. Sin embargo, la ganancia en clasificación trae consigo un coste computacional importante.

Volviendo al tema del coste computacional, hemos obtenido también que el modelo lineal ha obtenido unos resultados muy buenos (solo superados por SVM) y con un coste menor derivado de la simplicidad del modelo. Luego la regresión logística es también un modelo a tener en cuenta y una muy buena opción para predecir nuestro problema.

Por útltimo, hay que recalcar la inferioridad en las tasas de clasificación obtenidas por los modelos de boosting en este problema, pues son con diferencia los métodos que peores resultados proporcionan, a la vez que traen consigo un tiempo de cómputo considerable.

En conclusión, a la hora de ajustar un modelo para estos datos, siempre que el tiempo de cómputo no sea un problema el modelo más adecuado es SVM, aunque si esto supone alguna complicación podemos conformarnos con el modelo lineal, cuyos resultados son igualmente aceptables.