SISTEMA INTELIGENTES PARA LA GESTIÓN EN LA EMPRESA

Universidad de Granada

Práctica 1: Pre-procesamiento de datos y clasificación binaria

Sergio Samaniego Martínez

Curso 2018/2019

Contenido

1. Exploración

En este problema se trabajará con un conjunto de datos de préstamos que se nos proporcipona.

Disponemos de un conjunto de datos en el que tenemos 200 columnas y 200.000 filas. Entre estas variables disponemos de una de ellas llamada “target” que nos indicará si un cliente realizará una transacción en un futuro o no.

De estas variables no tenemos información alguna, ya que todos los nombres de las mismas son “var_0”, “var_1”, etc.

Lo único que podemos observar es que todas ellas son numéricas.

2. Pre-procesamiento

En el preprocesamiento vamos a comenzar por estudiar las variables. Para ello usaremos la función df_status con el conjunto de datos para obtener una salida de esta forma:

```
> status <- df_status(data_raw)
    variable q_zeros p_zeros q_na p_na q_inf p_inf      type unique
1    ID_code       0    0.00    0 0.00     0     0 character 200000
2     target  181989   90.99    0 0.00     0     0   numeric      2
3      var_0       0    0.00   17 0.01     0     0   numeric  94670
4      var_1       1    0.00   11 0.01     0     0   numeric 108931
5      var_2       0    0.00   19 0.01     0     0   numeric  86554
6      var_3       0    0.00   16 0.01     0     0   numeric  74593
7      var_4       0    0.00   22 0.01     0     0   numeric  63513
8      var_5       3    0.00   21 0.01     0     0   numeric 141017
9      var_6       0    0.00   21 0.01     0     0   numeric  38599
10     var_7       0    0.00   20 0.01     0     0   numeric 103059
11     var_8       1    0.00   17 0.01     0     0   numeric  98615
12     var_9       0    0.00   19 0.01     0     0   numeric  49417
13    var_10       1    0.00   12 0.01     0     0   numeric 128760
14    var_11       2    0.00   24 0.01     0     0   numeric 130183
15    var_12       0    0.00   26 0.01     0     0   numeric   9560
16    var_13       0    0.00   20 0.01     0     0   numeric 115175
17    var_14       0    0.00   20 0.01     0     0   numeric  79119
18    var_15       0    0.00   20 0.01     0     0   numeric  19810
19    var_16       0    0.00   21 0.01     0     0   numeric  86914
```

En esta salida podemos observar para cada columna el número de ceros que hay, su porcentaje, el número de valores perdidos, su porcentaje, el número y porcentaje de valores infinitos, el tipo de la variable y los valores únicos.

Con estos datos, lo primero que podemos observar es que hay algunas variables con un alto porcentaje de valores perdidos, por lo cual, pasaremos a eliminarlas.

```
## columnas con NAs
na_cols <- status %>%
  filter(p_na > 70) %>%
  select(variable)
```

Estamos seleccionando aquellas columnas que tienen un porcentaje de valores perdidos mayor al 70%, ya que creo que no nos dan la información suficiente como para trabajar con ellas.

Seguidamente pasamos a seleccionar las columnas con una gran cantidad de valores únicos, concretamente más de un 80%. Ya que estas variables no tienen demasiada relevancia, ya que no nos sirven para predecir el comportamiento.

```
## columnas con valores diferentes
dif_cols <- status %>%
  filter(unique > 0.8 * nrow(data_raw)) %>%
  select(variable)
```

Seleccionadas todas estas variables, pasamos a eliminarlas y como vemos en la siguiente tabla, las variables eliminadas son las siguientes:

	variable
1	ID_code
2	var_45
3	var_74
4	var_117

Ahora se va a pasar a tratar las instancias con valores perdidos. Antes de nada vemos que hay 3493 valores NA de 200.000 instancias. Vemos que la proporción es bastante pequeña, por lo que podemos optar por dos alternativas:

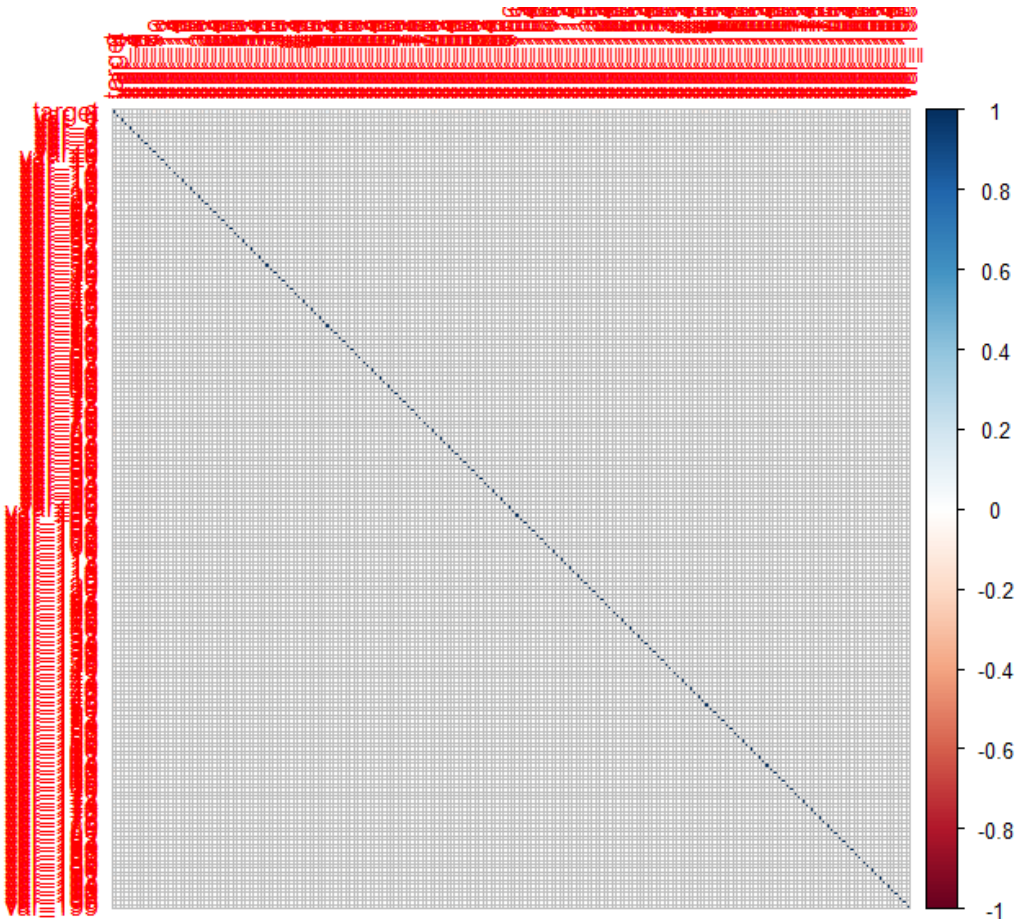
- Tratar los valores perdidos
- Eliminar las instancias que contengan valores perdidos, ya que al ser un porcentaje tan pequeño no afectaría al conjunto de datos.

En mi caso he optado por eliminar los valores perdidos, con **na.exclude()**.

Tras hacer el tratamiento de los datos, vemos que ya no tenemos valores perdidos.

El siguiente paso que se va a realizar va a ser la correlación del as variables con **target.**

Para ello usamos la función de R **cor()** para obtener una matriz de correlaciones. Una vez tenemos la matriz de correlaciones la representamos con **corrplot** y obtenemos el siguiente resultado.



Al ser una matriz tan grande no se diferencian del todo bien las celdas, pero a simple vista vemos que la única correlación que existe está en la diagonal principal, es decir, de la variable con sí misma y que con el resto de variables no existe correlación. Lo podemos comprobar además accediendo a ver los valores de la matriz de correlación para confirmar lo que vemos en el gráfico.

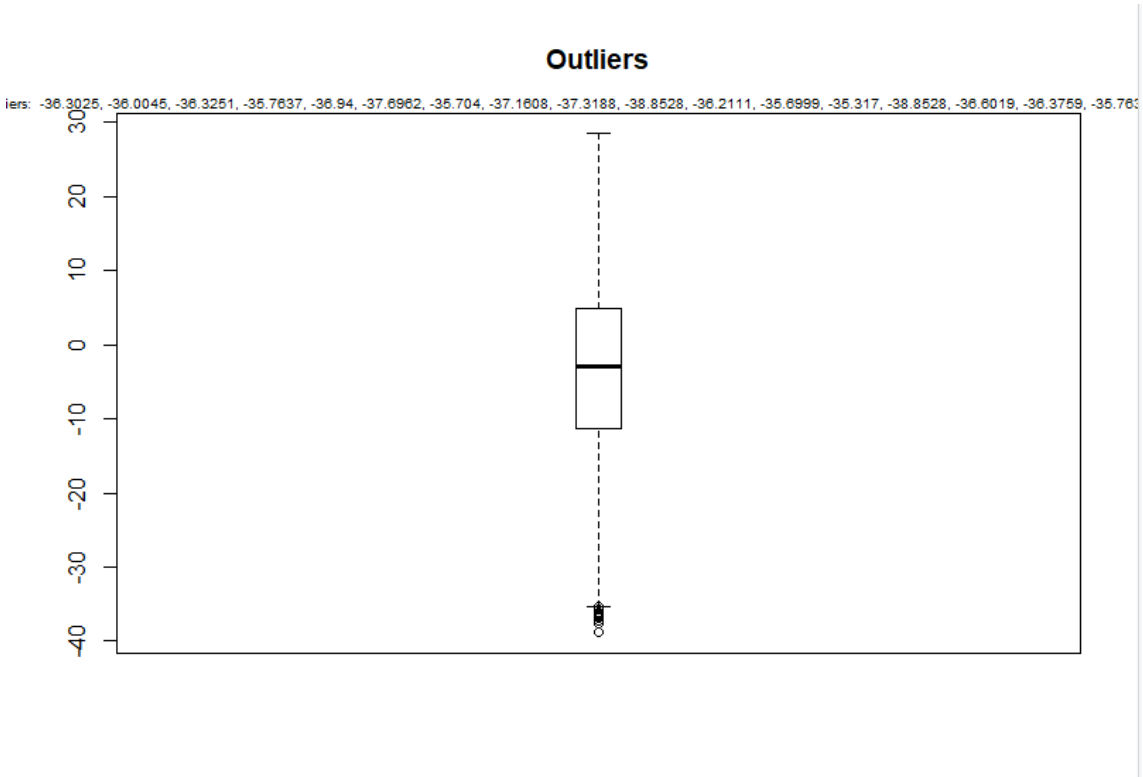
	target	var_0	var_1	var_2	var_3	var_4	var_5
target	1.0000000000	4.961552e-02	4.874570e-02	5.340174e-02	1.121657e-02	1.116165e-02	2.976539e-02
var_0	0.0496155174	1.000000e+00	-1.067634e-03	7.202743e-03	3.889587e-03	1.742087e-03	3.449142e-03
var_1	0.0487456971	-1.067634e-03	1.000000e+00	3.912048e-03	3.999314e-04	2.106570e-04	-1.156854e-03
var_2	0.0534017418	7.202743e-03	3.912048e-03	1.000000e+00	7.296699e-04	7.701138e-04	1.140961e-03
var_3	0.0112165702	3.889587e-03	3.999314e-04	7.296699e-04	1.000000e+00	-6.972724e-06	3.506839e-03
var_4	0.0111616491	1.742087e-03	2.106570e-04	7.701138e-04	-6.972724e-06	1.000000e+00	-9.180963e-04
var_5	0.0297653894	3.449142e-03	-1.156854e-03	1.140961e-03	3.506839e-03	-9.180963e-04	1.000000e+00
var_6	0.0641506338	6.777407e-03	3.879088e-03	5.422608e-04	-1.362145e-03	-3.750077e-05	2.603966e-03
var_7	-0.0027749435	2.524771e-03	1.048505e-03	-1.030459e-03	2.488731e-03	4.633483e-03	-3.462785e-04
var_8	0.0188263271	4.899833e-03	3.954072e-03	2.911572e-03	3.965588e-03	9.267524e-04	1.995730e-04
var_9	-0.0405043105	-2.615644e-03	-9.141614e-04	-1.955487e-03	-7.786189e-04	-7.391443e-04	-4.830761e-03
var_10	-0.0007055629	2.762522e-04	3.230502e-03	-1.119950e-03	-6.703643e-04	-3.536546e-03	5.534053e-05
var_11	0.0238088214	3.088199e-03	4.547472e-03	5.622655e-03	6.048904e-04	1.094348e-03	-8.575033e-04
var_12	-0.0645802224	-2.293675e-03	-1.774178e-03	-3.946505e-03	-1.558025e-03	7.216797e-04	-1.733352e-04
var_13	-0.0510948577	-2.737025e-03	-1.055136e-03	-9.213297e-03	-5.505183e-03	2.412264e-03	-4.197428e-03
var_14	-0.0063943295	-4.795851e-03	-1.832301e-03	-3.243106e-03	6.798852e-04	-1.185021e-03	-1.326863e-03
var_15	0.0176638468	-3.038628e-04	4.547854e-03	3.814296e-03	3.182330e-03	-6.542192e-04	3.010532e-03
var_16	0.0083605051	1.006289e-03	-2.454222e-03	9.041958e-04	2.874062e-03	1.457292e-03	-6.003368e-04
var_17	0.0006768967	-1.584731e-03	-8.388549e-04	3.138064e-04	3.719618e-03	-4.545067e-04	-2.618479e-03

Vemos claramente, que las correlaciones siempre son cercanas al 0.

Por lo tanto, no podemos eliminar variables ya que no hay correlación entre ellas. Podemos decir que cada variable es independiente de las demás.

Visto que no podemos eliminar variables, intentaremos eliminar datos con ruido. En este caso, se va a realizar un tratamiento de los outliers.

Vamos a ver por ejemplo la distribución de “var_199”.

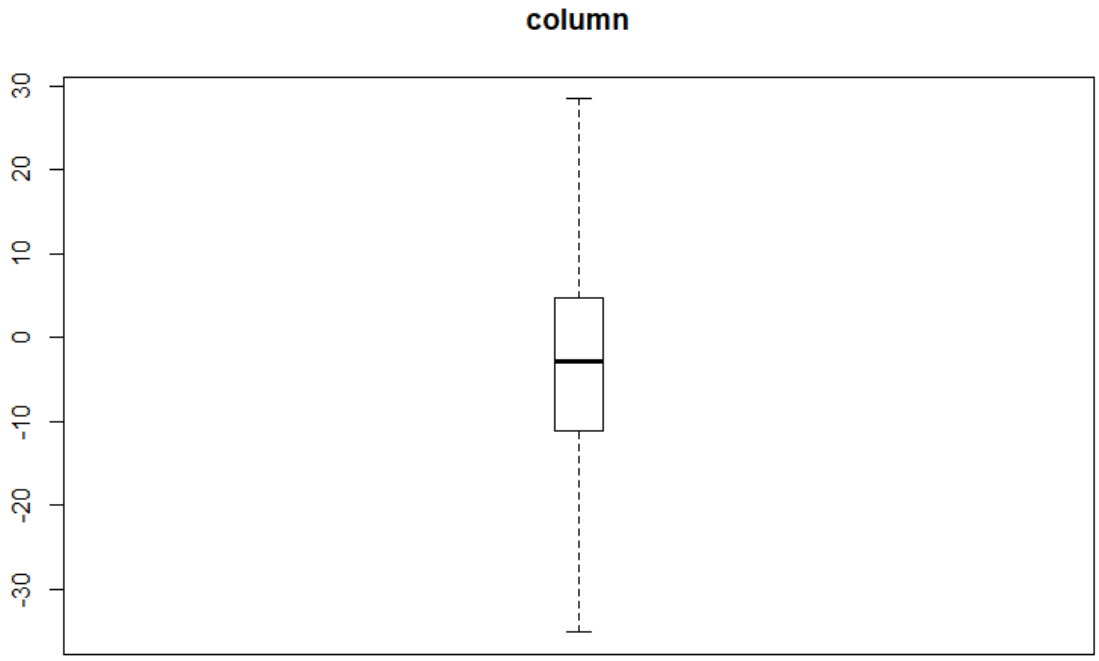


Como vemos hay datos, los cuales los escribimos en la parte superior del gráfico que los tratamos como outliers. Esto ocurre en prácticamente todas las variables por lo que se pasará a eliminarlos. El tratamiento a realizar será el siguiente:

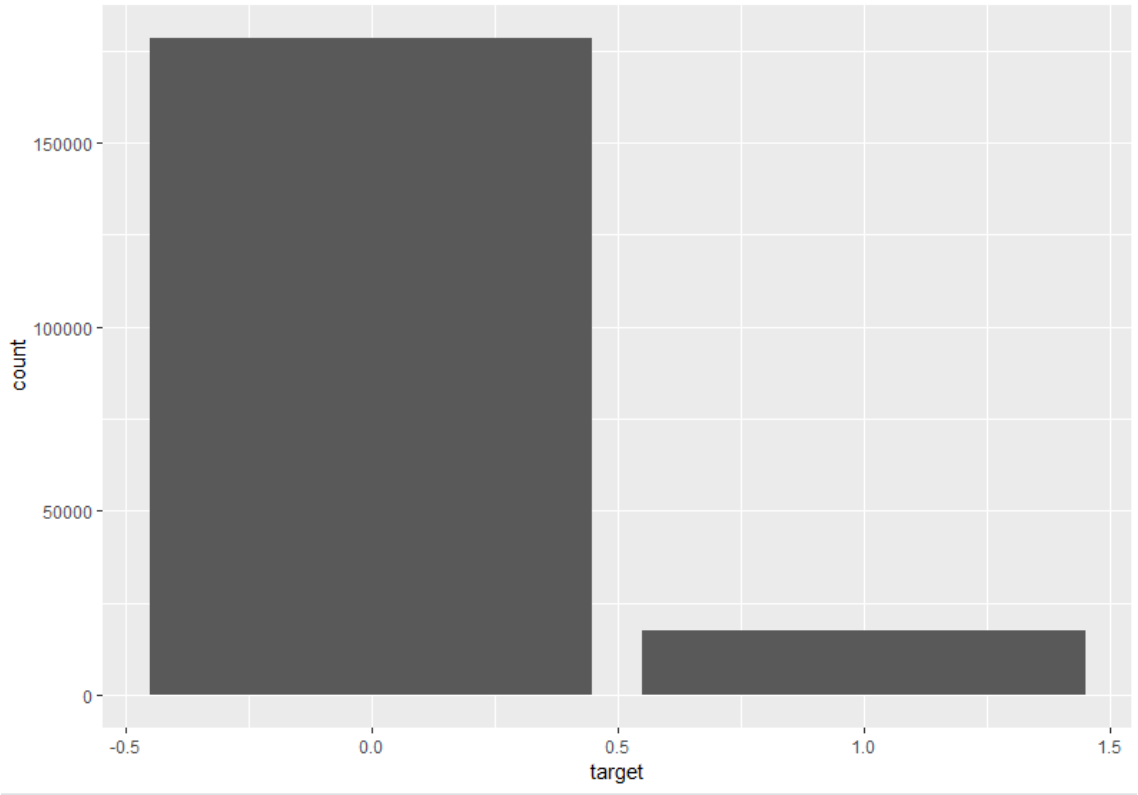
```
#Iteramos por todas las columnas para quitar los outliers
for(column in colnames(data_imp)){
  if(column != "target"){
    x <- data_imp[[column]]
    qnt <- quantile(x, probs=c(.25, .75), na.rm = T)
    caps <- quantile(x, probs=c(.05, .95), na.rm = T)
    H <- 1.5 * IQR(x, na.rm = T)
    x[x < (qnt[1] - H)] <- caps[1]
    x[x > (qnt[2] + H)] <- caps[2]

    boxplot(x, main=column, boxwex=0.1)
    data_no_outliers[[column]] <- x

  }

}
```

Por cada columna que no sea "**target**", los valores que se encuentran fuera de los límites de 1.5 * IQR (Rango intercuantílico), podríamos limitarlos, reemplazando todos aquellos valores que estén por debajo del 5% de Percentil y por encima del 95% del percentil.

De esta forma, las variables quedarían acotadas sin outliers, como vemos en la imagen tras ser tratados.



Finalmente pasamos a estudiar el equilibrio entre las clases.



Como se observa en la imagen hay un gran desequilibrio en la clase **target** ya que hay **178426** datos con valor 0 y **17670** con valor 1.

Por lo tanto para realizar la clasificación se hará uso de la técnica de **downsamplig** ya que tenemos una gran cantidad de datos y es preferible reducir los datos con valor 0 y equilibrar las clases en lugar de aumentar la cantidad de datos con valor 1, que aumentaría la cantidad de instancias y la clasificación nos tardaría muchísimo tiempo.

3. Clasificación

3.1 Random Forest

Random forest es una combinación de árboles predictores tal que cada árbol depende de los valores de un vector aleatorio probado independientemente y con la misma distribución para cada uno de estos. Es una modificación sustancial de bagging que construye una larga colección de árboles no correlacionados y luego los promedia.

Esta técnica para nuestro caso es una técnica que nos puede dar unos resultados buenos, debido a que nuestras variables son independientes y no conocemos nada acerca de ellas. Por lo tanto, debido a que generamos árboles aleatorios y posteriormente se hace el promedio, nos puede resultar muy útil para este problema.

Lo primero que haremos será entrenar nuestros datos de entrenamiento con Random Forest. Para ello se ha usado la función que se nos proporcionó en las prácticas de la asignatura SIGE del problema Loans.

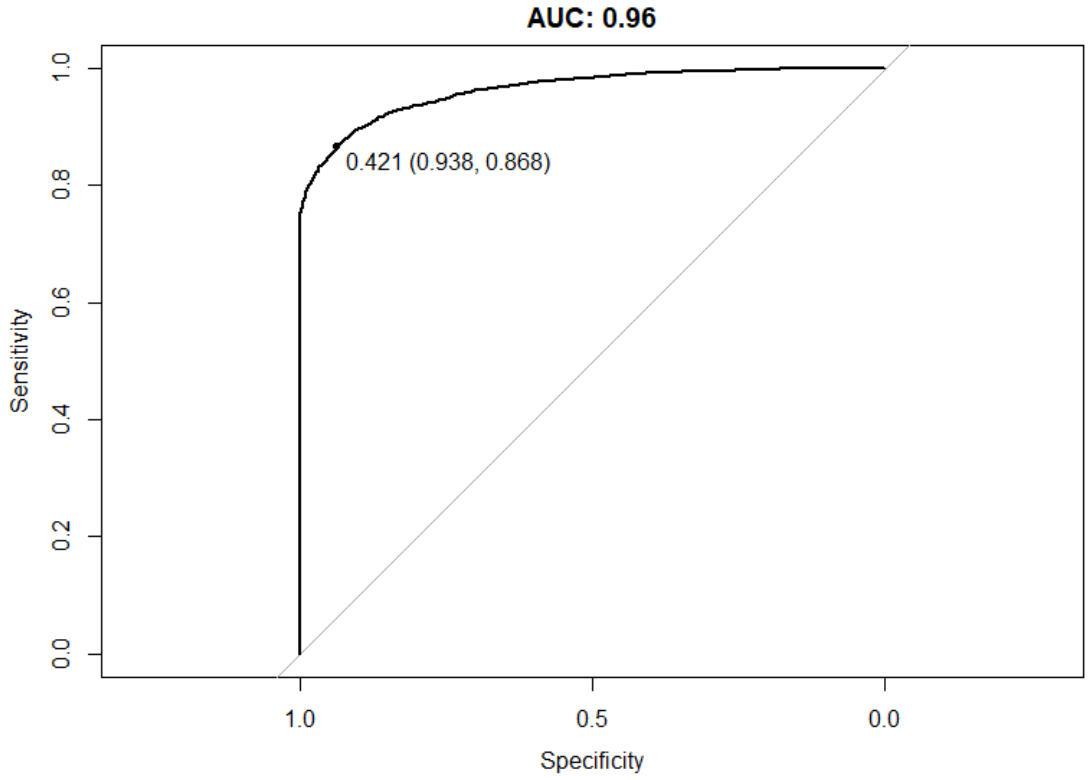
La función es la siguiente:

```
trainRF <- function(train_data, rfCtrl = NULL, rfParametersGrid = NULL) {
  if(is.null(rfCtrl)) {
    rfCtrl <- trainControl(
      verboseIter = F,
      classProbs = TRUE,
      method = "repeatedcv",
      number = 10,
      repeats = 1,
      summaryFunction = twoClassSummary)
  }
  if(is.null(rfParametersGrid)) {
    rfParametersGrid <- expand.grid(
      .mtry = c(sqrt(ncol(train))))
  }

  rfModel <- train(
    target ~ .,
    data = train_data,
    method = "rf",
    metric = "ROC",
    trControl = rfCtrl,
    tuneGrid = rfParametersGrid)

  return(rfModel)
}
```

Le indicamos que queremos entrenar con Random Forest y la métrica será una curva ROC. Además le pasamos los parámetros trControl, donde le indicamos que queremos que haga 10 árboles y una única repetición y el parámetro tuneGrid que es un marco de datos con posibles valores de afinación.

Con esto, haremos la predicción, usando la función **predict** donde le pasaremos el modelo entrenado, los datos de validación y que nos muestre la matriz de probabilidades.

Con esto, ya podremos dibujar la curva ROC de nuestro modelo y obtendremos el siguiente resultado.



Como se observa, tenemos un muy buen resultado, además si mostramos la matriz de confusión obtenemos estos datos:

```
          No  Yes
No  2729  882
Yes  190 3421
```

3.2 Conditional Tree

Esta segunda parte usaremos un “Conditional Tree”. Esta técnica nos permite hacer una partición recursiva de variables continuas y ver el árbol de decisión que se ha generado. Al tener variables continuas, se decide probar con este método y estudiar los resultados.

Lo que haremos será usar la función **ctree()** donde le pasamos la variable a clasificar y los datos de entrenamiento. Con este modelo entrenado, lo predecimos igual que hicimos con Random Forest y veremos los datos de la curva ROC.

Tras entrenar, el árbol generado nos da estos datos:

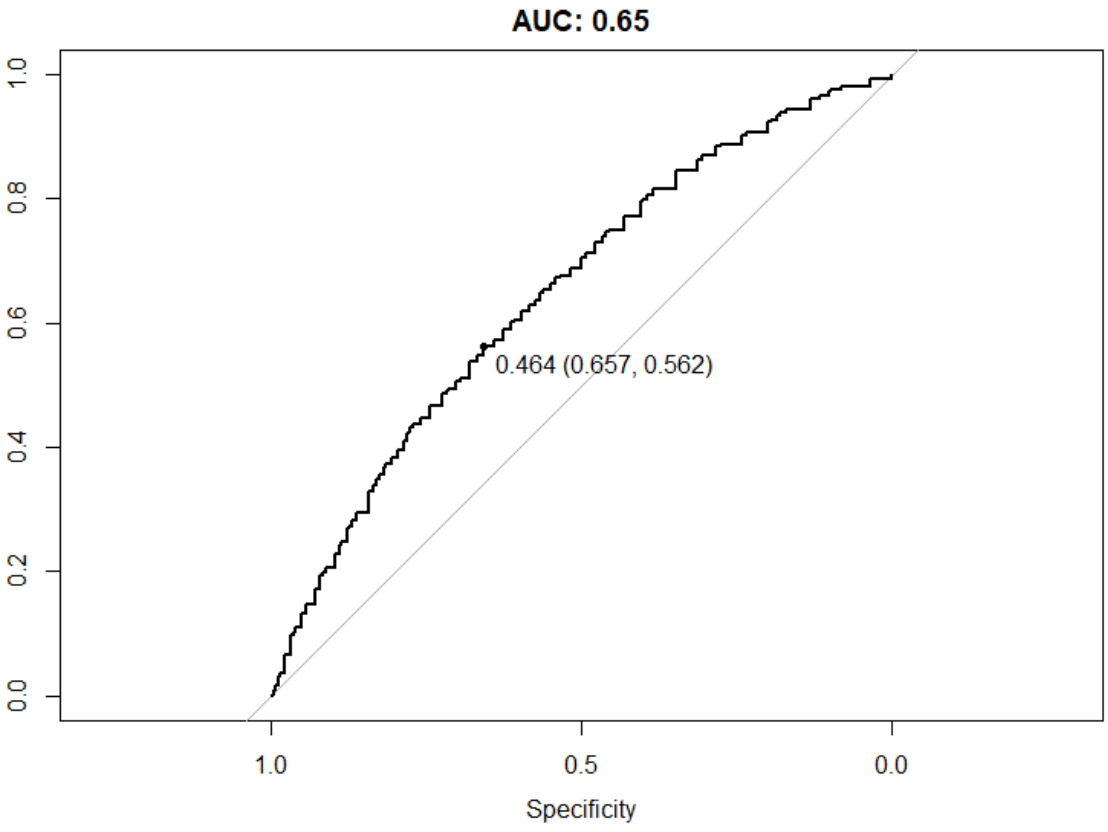
```
Number of inner nodes:    96
Number of terminal nodes: 97
```

Vemos que se ha generado un árbol grande que si lo intentamos representar no veremos nada, ya que los nodos se superponen unos con otros. Pero podemos ver una parte de la toma de decisiones que ha tomado, como por ejemplo:

```
[1] root
|   [2] var_81 <= 11.7703
|   |   [3] var_110 <= 10.9468
|   |   |   [4] var_6 <= 6.1978
|   |   |   |   [5] var_2 <= 13.9066
|   |   |   |   |   [6] var_169 <= 5.535
|   |   |   |   |   |   [7] var_94 <= 9.9545
|   |   |   |   |   |   |   [8] var_76 <= 5.5539: Yes (n = 136, err = 32.4%)
|   |   |   |   |   |   |   [9] var_76 > 5.5539: No (n = 126, err = 41.3%)
|   |   |   |   |   |   [10] var_94 > 9.9545: Yes (n = 495, err = 30.5%)
|   |   |   |   |   [11] var_169 > 5.535
|   |   |   |   |   |   [12] var_33 <= 8.3753: Yes (n = 94, err = 19.1%)
|   |   |   |   |   |   [13] var_33 > 8.3753
|   |   |   |   |   |   |   [14] var_172 <= 9.8389: Yes (n = 147, err = 32.7%)
|   |   |   |   |   |   |   [15] var_172 > 9.8389
|   |   |   |   |   |   |   |   [16] var_133 <= 7.3651
|   |   |   |   |   |   |   |   |   [17] var_13 <= 4.9538
|   |   |   |   |   |   |   |   |   |   [18] var_34 <= 10.9499: Yes (n = 59, err = 20.3%)
|   |   |   |   |   |   |   |   |   |   [19] var_34 > 10.9499: No (n = 159, err = 44.7%)
|   |   |   |   |   |   |   |   |   [20] var_13 > 4.9538: No (n = 533, err = 39.4%)
```

Vemos que va definiendo umbrales en las variables y según esto va clasificando.

Con el árbol pasaremos a hacer la predicción y la curva ROC que obtenemos es la siguiente.



4. Discusión de resultados

Claramente los resultados obtenidos con Random Forest son mucho mejores que los que se han obtenido con Conditional Tree.

Esto puede ocurrir debido a que Conditional Tree intenta establecer parámetros con los que ir clasificando mientras que Random Forest va generando estos de forma aleatoria.

En nuestro ejemplo, ya vimos que las variables no tienen correlación alguna, por lo que es difícil sacar conclusiones y sacar umbrales de decisión como hace Conditional Tree, lo que hace que este llegue a fallar en muchos casos.

Mientras tanto, Random Forest, genera árboles aleatorios que nos viene muy bien para nuestro problema, ya que no tenemos conocimiento alguno de las variables y como hemos visto en la sección anterior, los resultados obtenidos son bastante aceptables.

5. Conclusiones

Nos hemos enfrentado ante un conjunto de datos que podría simular lo que nos encontramos en casos reales. Tenemos clases desbalanceadas, variables que no conocemos qué significan y tampoco podemos hacer demasiado con ellas.

Además de esto se suma que el conjunto de datos era grande, teníamos 300 MB de datos, los cuales para muchas operaciones podían costarnos muchas horas.

En el caso de los datos con valores nulos, en un principio comencé a usar la función **mice()** pero tras más de una hora ejecutándose decidí pararla, ya que realmente veíamos que eran un porcentaje muy pequeño y que se podían eliminar.

Igual que esta decisión comentada, se han tenido que ir haciendo otras muchas ya que debías conocer muy bien que se quiere hacer y pensar cómo conseguirlo sin tener varias horas el ordenador trabajando.

6. Bibliografía

[1] http://ugrad.stat.ubc.ca/R/library/randomForest/html/predict.randomForest.html

[2] https://www.rdocumentation.org/packages/party/versions/0.8-8/topics/Conditional%20Trees

[3] https://www.rdocumentation.org/packages/raster/versions/2.8-19/topics/predict

[4] https://www.rdocumentation.org/packages/caret/versions/4.47/topics/train

[5] http://r-statistics.co/Outlier-Treatment-With-R.html