Sistemas Inteligentes para la Gestión en la Empresa

(SIGE)

Práctica 1: Pre-procesamiento de datos y clasificación binaria

Curso 2018-2019

En colaboración con:

Autor

Jonathan Martín Valera – jmv742@correo.ugr.es

Índice

1. Introducción

En este documento se va a describir el proceso de análisis y resolución del problema de pre-procesamiento y aprendizaje automático propuesto en esta primera práctica.

El conjunto de datos con el que se va a trabajar es una variación del ofrecido en la competición de *Kaggle Santander Customer Transaction Prediction* (https://www.kaggle.com/c/santander-customer-transaction-prediction).

El problema consiste en predecir si un cliente realizará una transacción en el futuro (`target`) a partir del resto de variables (200). El conjunto de datos se tratará como un problema de clasificación binaria, con dos posibles salidas: *Yes, No*.

2. Exploración de los datos

2.1 Lectura de datos

Comenzaremos utilizando el fichero [*train_ok.csv*] proporcionado en esta práctica, donde encontramos un conjunto de 200.000 instancias que tienen 202 características o atributos que se procesarán posteriormente para crear el modelo de predicción.

```
data_raw <- read_csv('data/train_ok.csv')
```

Una vez que se ha cargado los datos, lo primero que se ha realizado ha sido comprobar las dimensiones del dataframe que hemos importado.

Como se puede observar en la siguiente salida, las dimensiones de los datos son 200.000 muestras, y cada una tiene un total de 202 características.

```
dim(data_raw)
```

```
## [1] 200000    202
```

A continuación, dado que el conjunto de datos es demasiado extenso, se ha visualizado las 10 primeras muestras para observar qué características representan, y si podemos deducir a primera vista alguna información o correlación entre dichas variables.

2.2 Visualización de datos

```
head(data_raw,10)
```

```
## # A tibble: 10 x 202
##    ID_code target var_0   var_1 var_2 var_3 var_4   var_5 var_6 var_7
##    <chr>    <dbl> <dbl>   <dbl> <dbl> <dbl> <dbl>   <dbl> <dbl> <dbl>
##  1 train_0      0  8.93 -6.79   11.9   5.09 11.5   -9.28   5.12  18.6
##  2 train_1      0 11.5  -4.15   13.9   5.39 12.4    7.04   5.62  16.5
##  3 train_2      0  8.61 -2.75   12.1   7.89 10.6   -9.08   6.94  14.6
##  4 train_3      0 11.1  -2.15    8.95  7.20 12.6   -1.84   5.84  14.9
##  5 train_4      0  9.84 -1.48   12.9   6.64 12.3    2.45   5.94  19.3
##  6 train_5      0 11.5  -2.32   12.6   8.63 11.0    3.56   4.53  15.2
##  7 train_6      0 11.8  -0.0832  9.35  4.29 11.1   -8.02   6.20  12.1
##  8 train_7      0 13.6  -7.99   13.9   7.60  8.65   0.831  5.69  22.3
##  9 train_8      0 16.1   2.44   13.9   5.63  8.80   6.16   4.45  10.2
## 10 train_9      0 12.5   1.97    8.90  5.45 13.6  -16.3    6.06  16.8
## # ... with 192 more variables: var_8 <dbl>, var_9 <dbl>, var_10 <dbl>,
```

Como se ha podido comprobar en los resultados anteriores, los nombres de las variables no son representativos y no nos aportan ningún valor semántico. Si comprobamos los valores de cada una de estas características vemos que son valores continuas.

Seguidamente, se ha visualizado un resumen genérico de los datos para ver si se puede extraer más información.

```
summary(data_raw)
```

```
##    ID_code              target            var_0              var_1
##  Length:200000      Min.   :0.00000   Min.   : 0.4084   Min.   :-15.043
##  Class :character   1st Qu.:0.00000   1st Qu.: 8.4536   1st Qu.: -4.740
##  Mode  :character   Median :0.00000   Median :10.5248   Median : -1.608
##                     Mean   :0.09005   Mean   :10.6799   Mean   : -1.628
```

```
##                    3rd Qu.:0.00000   3rd Qu.:12.7582   3rd Qu.:  1.359
##                    Max.   :1.00000   Max.   :20.3150   Max.   : 10.377
##                                      NA's   :17        NA's   :11
##      var_2            var_3             var_4            var_5
##  Min.   : 2.117   Min.   :-0.0402   Min.   : 5.075   Min.   :-32.5626
##  1st Qu.: 8.722   1st Qu.: 5.2542   1st Qu.: 9.883   1st Qu.:-11.1997
##  Median :10.580   Median : 6.8251   Median :11.108   Median : -4.8333
##  Mean   :10.715   Mean   : 6.7965   Mean   :11.078   Mean   : -5.0652
##  3rd Qu.:12.517   3rd Qu.: 8.3241   3rd Qu.:12.261   3rd Qu.:  0.9244
##  Max.   :19.353   Max.   :13.1883   Max.   :16.671   Max.   : 17.2516
##  NA's   :19       NA's   :16        NA's   :22       NA's   :21
##      var_6           var_7           var_8             var_9
##  Min.   :2.347   Min.   : 5.35   Min.   :-10.5055   Min.   : 3.970
##  1st Qu.:4.768   1st Qu.:13.94   1st Qu.: -2.3178   1st Qu.: 6.619
##  Median :5.385   Median :16.46   Median :  0.3937   Median : 7.630
##  Mean   :5.409   Mean   :16.55   Mean   :  0.2843   Mean   : 7.567
##  3rd Qu.:6.003   3rd Qu.:19.10   3rd Qu.:  2.9379   3rd Qu.: 8.584
##  Max.   :8.448   Max.   :27.69   Max.   : 10.1513   Max.   :11.151
##  NA's   :21      NA's   :20      NA's   :17         NA's   :19
##    ...
```

Como podemos observar, tampoco se puede extraer información relevante sobre las variables. Cada una de ellas tiene distintos valores numéricos con las que a priori no se puede extraer ninguna información.

2.3 Estado del conjunto de datos

Para observar el estado de los datos, y comprobar si existen valores perdidos, nulos... se ha hecho uso de la librería `funmodeling` y de la función `df_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
### ... with 192 more variables:
```

Como se puede observar en el resultado anterior, el porcentaje de valores perdidos de todas las variables es sumamente nulo a primera vista.

Si se calcula el mayor y menor porcentaje de todo los datos, se obtiene lo siguiente.

```
x<-df_status(data_raw)

max_pNA_value <- max(x$p_na)
min_pNA_value <- min(x$p_na)
sprintf("El mayor porcentaje de valores perdidos es --> %f",max_pNA_value)
sprintf("El menor porcentaje de valores perdidos es --> %f",min_pNA_value )

max_pINF_value <- max(x$p_inf)
min_pINF_value <- min(x$p_inf)
sprintf("El mayor porcentaje de valores infinitos es --> %f",max_pINF_value)
sprintf("El menor porcentaje de valores infinitos es --> %f",min_pINF_value)
```

```
## [1] "El mayor porcentaje de valores perdidos es --> 0.020000"
## [2] "El menor porcentaje de valores perdidos es --> 0.000000"
## [3] "El mayor porcentaje de valores infinitos es --> 0.000000"
## [4] "El menor porcentaje de valores infinitos es --> 0.000000"
```

Como conclusión de los resultados anteriores, podemos observar que el número de valores perdidos es muy poco significativo, ya que el mayor porcentaje de valores perdidos por variable es de un **0.02 %**. Respecto a los valores infinitos, podemos comprobar que el dataset no tiene ningún valor infinito, y como los tipos de las variables son numéricos, no podemos descartar a priori las variables con valor 0.

2.4 Balanceo de los datos

Respecto al balanceo de los datos según nuestro valor objetivo (`target`), se ha podido comprobar que **existe un gran desequilibrio** entre el número de datos cuyo target es 1 y 0.

Esto se ha comprobado de la siguiente forma:

```
table(data_raw$target)
```

```
##
##      0      1
## 181989  18011
```

Como se puede observar, hay un gran desequilibrio entre dichos valores. Si lo comprobamos mediante porcentajes obtenemos que aproximadamente un 91 % de los datos son transacciones no realizadas y un 9 % de las transacciones son realizadas.

```
prop.table(table(data_raw$target))
```

```
##
##        0        1
## 0.909945 0.090055
```

Podemos visualizar esta diferencia gráficamente:

```
plotdata <-
  data_raw %>%
  mutate(target = as.factor(target))
```

```
ggplot(plotdata,aes(x=target, fill =  target)) +
  geom_bar(width = 0.8)+
      xlab("Realizar transacción")+
      ylab("Total")+
      labs(fill = "target")
```

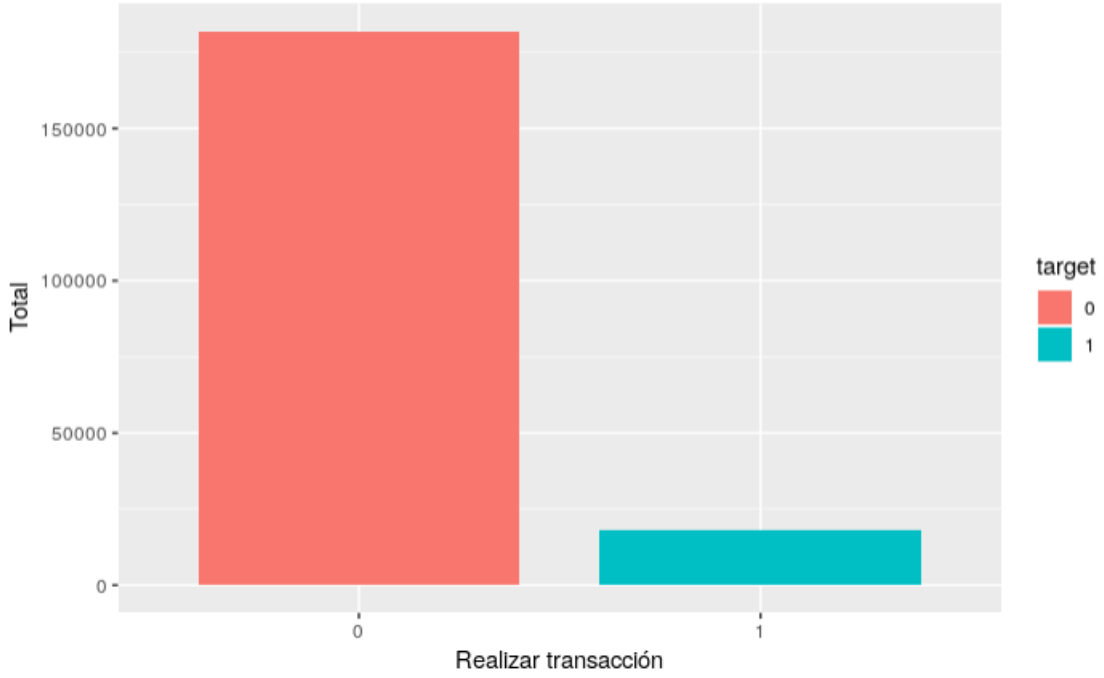


Figura 1: Desequilibrio de valores objetivo

2.5 Conclusión

Como conclusión general de la exploración de los datos, podemos concluir que **apenas se puede obtener ninguna información acerca de los datos**, ya que los nombres de las columnas no tienen ningún valor semántico, los valores de las variables son numéricos y continuos, y realizando un resumen de todos los datos, no se puede obtener ningún patrón ni elemento a destacar.

También se ha observado que el dataset no tiene apenas valores perdidos, y **está considerablemente desbalanceado**.En la etapa de preprocesamiento de los datos habrá que adoptar distintas estrategias para poder intentar balancear dichos datos y eliminar variables que tengan poca relevancia con nuestro objetivo (`target`).

3. Preprocesamiento

3.1 Limpieza de valores nulos

Tal y como hemos observado en la exploración de los datos, la cantidad de valores nulos es sumamente pequeña (el porcentaje máximo por variable era del 0.02 %) por lo que se ha optado por eliminar todas aquellas filas que tengan algún valor nulo.

Para ello, vamos a filtrar los datos que tengan valor `NA`

```
data <- na.omit(data_raw)
```

Si observamos la dimensión de los nuevos datos, podemos comprobar que se han eliminado 3959 filas.

```
dim(data)
```

```
## [1] 196041    202
```

Ahora, si volvemos a comprobar el porcentaje de valores nulos de los datos, efectivamente observamos que el porcentaje es del 0 %.

```
max_pNA_value <- max(x$p_na)
sprintf("El mayor porcentaje de valores perdidos es --> %f",max_pNA_value)
```

```
## [1] "El mayor porcentaje de valores perdidos es --> 0.000000"
```

> ***Nota***: Antes de tomar esta decisión, también se ha pensado en utilizar la librería `MICE` para realizar la imputación de dichos valores perdidos, pero dadas las dimensiones de nuestros datos y la cantidad de valores perdidos, se ha llegado a la conclusión de que **NO** vale la pena emplear un considerable tiempo de cálculo para imputar dichos valores perdidos.

3.2 Búsqueda de correlaciones

Una de las técnicas que se va a utilizar para reducir el número de variables es eliminar las variables menos correlacionadas con nuestra variable objetivo `target`.

Para ello, en primer lugar se ha generado una tabla de **correlaciones entre nuestra variable objetivo `target` y el resto**.

```
cor_target <-correlation_table(data_raw, target='target')
cor_target
```

```
##     Variable target
## 1     target   1.00
## 2      var_6   0.06
## 3     var_22   0.06
## 4     var_26   0.06
## 5     var_53   0.06
## 6    var_110   0.06
## 7      var_0   0.05
## 8      var_1   0.05
## 9      var_2   0.05
## 10    var_40   0.05
## ...
```

Como se puede observar, **no existe ninguna variable que esté altamente correlacionada** con nuestra variable objetivo `target`, por lo que a priori no podemos destacar una gran importancia de ninguna variable.

A continuación, se va a seleccionar las 100 variables que tengan mayor correlación con la variable objetivo `target` y se eliminará el resto.

> ***Nota***: El número 100 se ha escogido aleatoriamente para reducir el número de variables. Más adelante se irá aumentando o decrementando este número para observar el comportamiento del modelo en función de este parámetro.

Para la selección de las 100 variables más correladas, se escogerán los mayores valores en valor absoluto.

En primer lugar, se ordenan las variables de forma decreciente por valor absoluto.

```
cor_target <-correlation_table(data, target='target') %>%
    arrange(-abs(target))
cor_target
```

```
##     Variable target
## 1     target   1.00
## 2     var_81  -0.08
## 3    var_139  -0.07
## 4      var_6   0.06
## 5     var_22   0.06
## 6     var_26   0.06
## 7     var_53   0.06
## 8    var_110   0.06
## 9     var_12  -0.06
## 10    var_21  -0.06
## ...
```

El número de variables que tenemos hasta este momento es de 201.

```
dim(cor_target)
```

```
## [1] 201   2
```

A continuación, se establece un parámetro de corte (en este caso 100, tal y como se ha comentado anteriormente) y acotamos el número de variables descartando las 100 primeras y almacenando el resto.

Como se puede comprobar, el número de variables que se van a descartar es de 100.

```
split_parameter <- 100 # Parámetro para determinar el punto de corte
cor_target <- cor_target[(split_parameter+1):length(cor_target),]
```

```r
dim(cor_target)
```

```
## [1] 100   2
```

Finalmente, se eliminan dichas variables del conjunto de datos y resultamos con un total de 102 columnas.

```r
data <- data %>%
  select(-one_of(cor_target$Variable))
dim(data)
```

```
## [1] 196041    102
```

3.3 Borrado de columnas no útiles

Tras eliminar las columnas poco correladas con nuestra variable objetivo `target`, se ha procedido a eliminar columnas que no nos van a aportar información relevante a la hora de generar el modelo de predicción.

En primer lugar, se ha 'protegido' la variable objetivo `target` para que no sea procesada por los siguientes procedimientos de filtrado.

El primer filtro que se ha realizado ha sido el de seleccionar las filas cuyos valores sean distintos en una proporción del 70 %.

Observamos como se han seleccionado un total de 26 columnas.

```r
status <- status %>%
  filter(variable != 'target')

dif_cols <- status %>%
  filter(unique > 0.7 * nrow(data)) %>%
  select(variable)

dim(dif_cols)
```

```
## [1] 26  1
```

El siguiente filtro que se ha utilizado ha sido el de seleccionar valores con muy poca variabilidad, en concreto se ha establecido un valor del 5 %.

En este caso, se han seleccionado 2 columnas.

```
eq_cols <- status %>%
  filter(unique < 0.05 * nrow(data)) %>%
  select(variable)

dim(eq_cols)
```

```
## [1] 2 1
```

En último lugar, se ha aplicado un filtro para eliminar las columnas cuyos valores tengan un 90 % o más de ceros.

Como se puede observar, no ha habido ninguna selección de este tipo.

```
zero_cols <- status %>%
  filter(p_zeros > 0.9 * nrow(data)) %>%
  select(variable)

dim(zero_cols)
```

```
## [1] 0 1
```

A continuación se ha procedido a eliminar del conjunto de datos todas estas columnas seleccionadas.

Como se puede observar, hemos reducido el número de columnas a **74**.

```
remove_cols <- bind_rows(
  list(
    zero_cols,
    eq_cols,
```

```
    dif_cols
  )
)

data <- data%>%
  select(-one_of(remove_cols$variable))

dim(data)
```

```
## [1] 196041     74
```

3.4 Transformaciones de columnas

Para poder trabajar posteriormente en la construcción del modelo de predicción, es necesario convertir la variable objetivo `target` en una variable de tipo factor.

Para ello, lo que se ha hecho ha sido sustituir todos los valores 0 en No y los valores un en Si.

```
data <- data%>%
  mutate(target = as.factor(ifelse(target== 1, 'Yes', 'No')))
summary(data)
```

```
##  target
##  No :178377   ...
##  Yes: 17664
```

3.5 Balanceo de datos

Tal y como se ha podido observar en la figura 1, existe un gran desequilibrio en la clase `target` objetivo.

Con el fin de poder mejorar nuestro modelo de aprendizaje, se ha realizado un balanceo de datos utilizando la función `SMOTE` de la librería `DMwR`.

En este caso, se ha decidido realizar 3 tipos de balanceos diferentes para ir comparando posteriormente sus comportamientos en los diferentes modelos de aprendizaje.

```
data_1.1 <- SMOTE(target~., as.data.frame(data), perc.over=100)
data_3.1 <- SMOTE(target~., as.data.frame(data), perc.over=300, dataperc.under=100)
data_5.1 <- SMOTE(target~., as.data.frame(data), perc.over=500, dataperc.under=100)
```

A continuación, vamos a graficar la proporción entre las clases del objetivo `target`.

```
plotdata1.1 <-
   data_1.1 %>%
   mutate(target = as.factor(target))


plotdata3.1 <-
    data_3.1 %>%
    mutate(target = as.factor(target))


plotdata5.1 <-
    data_5.1 %>%
    mutate(target = as.factor(target))
```

En primer lugar, tenemos un balanceo equitativo entre las clases. Esto se ha realizado utilizando el parámetro `perc.over=100` tal y como dice en [REFERENCIA].

Tras realizar el balanceo, vamos a observar la nueva dimensión de los datos

```
dim(data_1.1)
```

```
## [1] 70656    74
```

Como podemos observar, se han reducido los datos considerablemente de 196041 a 70656. Este cambio ha sido debido a que se ha realizado un *downsampling* de la clase mayoritaria y un *upsampling* de la clase minoritaria hasta equilibrarse ambos.

Si mostramos la gráfica, observamos como están correctamente balanceados.

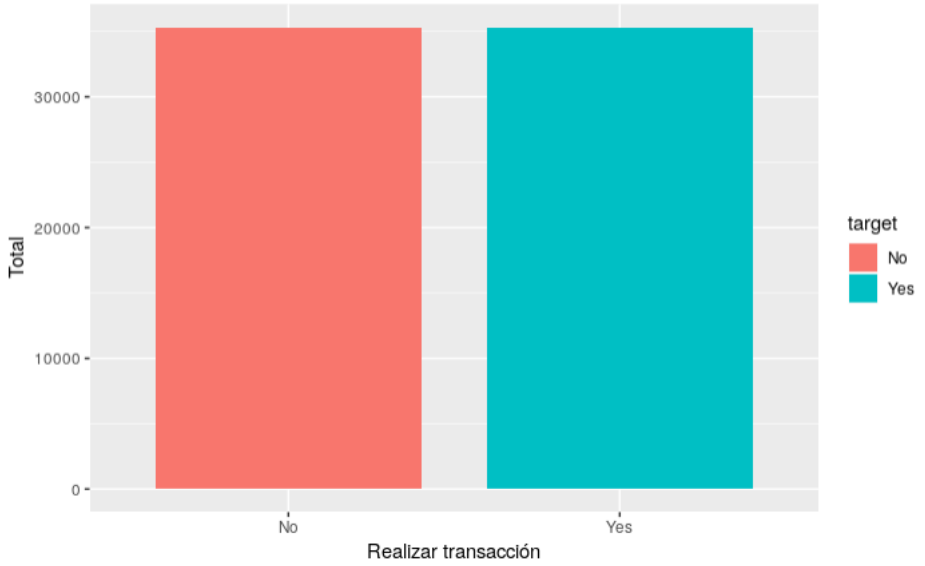


Figura 2: Balanceo de datos igualitario

El segundo balanceo que se ha realizado ha sido estableciendo los parámetros a `perc.over=300, dataperc.under=100`. De esta forma obtenemos un *upsampling* y *downsampling* más moderado obteniendo la siguiente proporción:

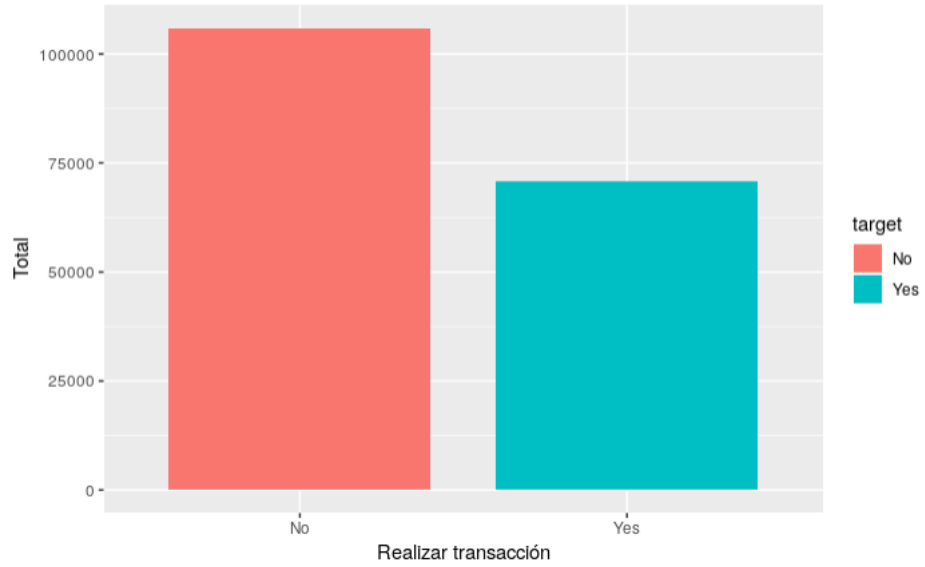


Figura 3: Balanceo de datos 2

Si observamos la nueva dimensión de estos datos observamos que tenemos una dimensión parecida a la que se tenía inicialmente antes de aplicar dicho balanceo.

```
dim(data_3.1)
```

```
## [1] 176640     74
```

Por último, se ha realizado un nuevo balanceo estableciendo como parámetros `perc.over=500, dataperc.under=100`

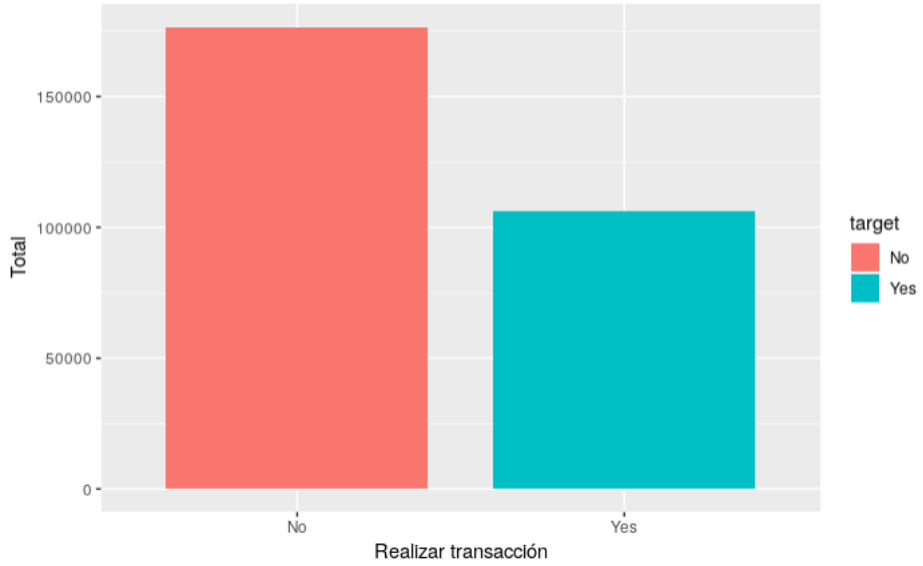


Figura 4: Balanceo de datos 3

En este nuevo balanceo, observamos como se mantiene un poco la misma proporción respecto al balanceo anterior (aunque en este caso es inferior), pero se ha aumentado el número de muestras considerablemente.

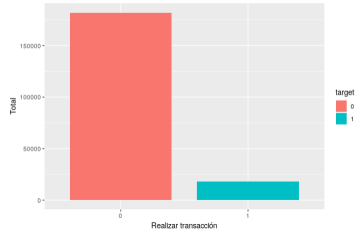
```
dim(data_5.1)
```

```
## [1] 282624     74
```

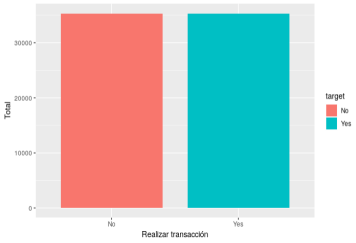
Estos conjuntos de datos que se han generado como resultado del balanceo, serán testeados a la hora de generar los modelos de predicción para observar su comportamiento.

Como resumen de este apartado, observamos como inicialmente teníamos un conjunto **totalmente desbalanceado** y aplicando este proceso se han obtenido varios conjuntos

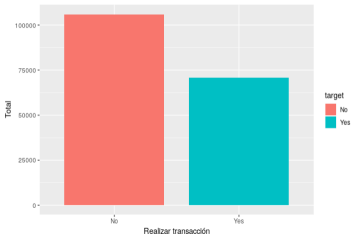
de datos con distintas muestras y proporción entre clases.En la siguiente figura podemos ver los distintos cambios.



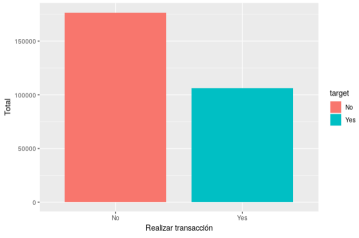
(a) Datos iniciales



(b) Primer balanceo



(c) Segundo balanceo



(d) Tercer balanceo

Figura 5: Conjuntos de datos

3.6 Reordenación de los datos

Otra medida que se ha llevado a cabo en el preprocesamiento, ha sido el de **reordenar** los datos.

Considero que esta es una buena práctica, debido a que puede que los datos estén ordenados por alguna etiqueta y que al realizar la partición de los datos en los conjuntos de entrenamiento y validación, el modelo de aprendizaje no generalice lo suficiente o que el conjunto de validación esté totalmente desbalanceado.

Por este motivo, he decidido 'barajar' los datos, de forma que el reparto de éstos sea de forma aleatoria y que el modelo de aprendizaje no aprenda según el orden, sino que generalice lo máximo posible.

```
# Semilla de aleatoriedad
set.seed(7)
# Aleatorizar filas
data <- data[sample(1:nrow(data)), ]
```

4. Clasificación

Una vez que se ha realizado el preprocesamiento de los datos, ya estamos preparados para construir modelos de aprendizaje automático realizado una serie de técnicas.

Para realizar esto y cumpliendo con las propuestas del guión de prácticas, se va a utilizar la librería `caret`, ya que nos proporciona una serie de métodos que nos facilita la construcción y validación de varios modelos de aprendizaje.

Durante la construcción de los modelos de aprendizaje, se han realizado una gran cantidad de pruebas en las que se han ido modificando parámetros y evaluando los resultados para obtener unas conclusiones acerca de la predicción de datos.

Concretamente, las pruebas que se van a especificar a continuación se han realizado para los modelos **rpart** y **rf** de la librería de caret que se corresponde con los métodos de clasificación llamados árboles de regresión y random forest. Decir que también se han estado realizando pruebas con el modelo **svm**, pero que desafortunadamente no se ha obtenido ningún resultado, ya que el tiempo de procesamiento de dicho algoritmo ha sido bastante elevado hasta el punto de tener que cancelar su ejecución.

Antes de empezar a construir los modelos de aprendizaje, se ha procedido a particionar el conjunto de datos en dos: *Entrenamiento y test*. El conjunto de entrenamiento dispone del 80 % del total de los datos, y el conjunto de test tiene el 20 % restante.

```
# Particiones entrenamiento / test
trainIndex <- createDataPartition(data$
              target, p = .8, list = FALSE, times = 1)
data_train <- data[ trainIndex, ]
data_test   <- data[-trainIndex, ]

dim(data_train)
dim(data_test)
```

```
[1] 156834     74
[2] 39207      74
```

Clasificación de datos no balanceados

En primer lugar, se han estado realizando análisis sobre los datos preprocesados quitando la etapa de balanceamiento. El principal motivo de esta decisión es tener una primera toma de contacto sobre el comportamiento de los modelos teniendo en cuenta el alto grado de desbalance que hay.

El primer modelo que se ha generado ha sido utilizando `rpart` como método de entrenamiento y `ROC` como métrica de validación.

```
# Parámetros
rpartCtrl <- trainControl(verboseIter = F, classProbs = TRUE,
                          summaryFunction = twoClassSummary)

rpartParametersGrid <- expand.grid(.cp = c(0.01, 0.05,0.1))

# Entrenamiento del modelo
rpartModel <- train(target~ ., data = data_train, method = "rpart",
                    metric = "ROC", trControl = rpartCtrl,
                    tuneGrid = rpartParametersGrid)

# Validación
predictionValidationProb_rpart <- predict(rpartModel,
                                          data_test, type = "prob")
auc_rpart <- roc(data_test$target, predictionValidationProb_rpart[["Yes"]],
                 levels = unique(data_test[["target"]]))
auc_rpart
roc_validation_rpart <- plot.roc(auc_rpart, ylim=c(0,1), type = "S" ,
                                 print.thres = T, main=paste('Validation AUC:',
                                 round(auc_rpart$auc[[1]], 2)))

# Predicción y matriz de confusión
prediction_rpart <- predict(rpartModel, data_test, type = "raw")
```

```
confusion_matrix_rpart <- confusionMatrix(table(prediction_rpart,
                                                data_test[["target"]]))
confusion_matrix_rpart

# Visualización del modelo
rpartModel_party <- as.party(rpartModel$finalModel)
plot(rpartModel_party)
fancyRpartPlot(rpartModel$finalModel)
```

Los resultados que se han obtenido con este modelo son relativamente malos. Como se puede observar, el área bajo la curva ha sido de 0.5, y como se puede observar en la matriz de confusión, vemos el 100 % de sus predicciones ha sido la respuesta 'NO' o 0.

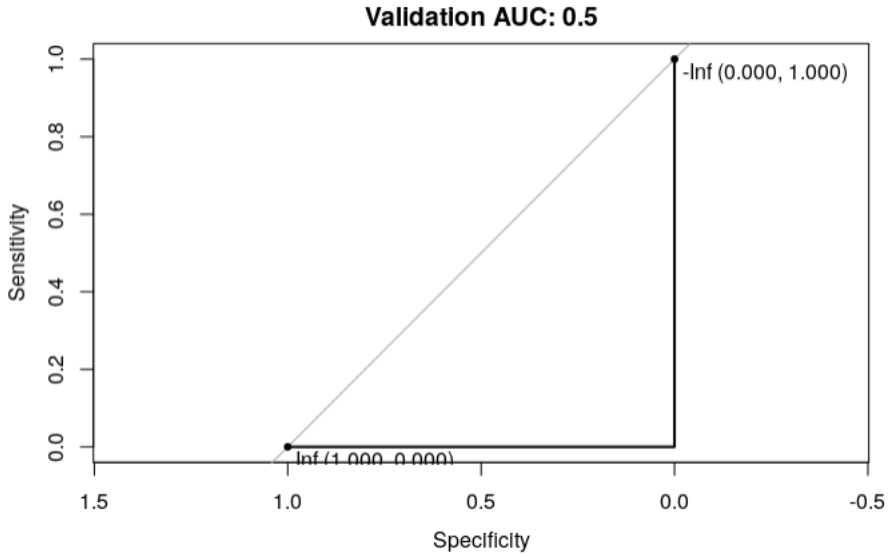


Figura 6: Curva ROC con rpart de los datos desbalanceados

El principal motivo por el cual el modelo siempre predice que no, es que **hay un gran desbalance entre las clases de los datos**, y es por ello que el modelo no se ha entrenado correctamente y siempre predice la clase mayoritaria. Esto era previsible y por ello se ha tenido esta consideración en el preprocesamiento y se han generado conjuntos de datos adicionales balanceados más equitativamente.

```
Confusion Matrix and Statistics


prediction_rpart    No   Yes
             No  35675  3532
             Yes     0     0

               Accuracy : 0.9099
                 95% CI : (0.907, 0.9127)
    No Information Rate : 0.9099
    P-Value [Acc > NIR] : 0.5045

                  Kappa : 0
 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 1.0000
            Specificity : 0.0000
         Pos Pred Value : 0.9099
         Neg Pred Value :    NaN
             Prevalence : 0.9099
         Detection Rate : 0.9099
   Detection Prevalence : 1.0000
      Balanced Accuracy : 0.5000

       'Positive' Class : No
```

Figura 7: Matriz de confusión con rpart de los datos desbalanceados

Tras esta prueba, se ha vuelto a construir otro modelo utilizando **rf** de `caret`.

```
 # RANDOMFOREST


# Parámetros
control_rf <- trainControl(method="none", number=3,
                           repeats=1, search="random",classProbs = TRUE)
mtry <- sqrt(ncol(data_train))


# Entrenamiento
rfModel <- train(target~., data=data_train, method="rf",
                 metric="ROC", trControl=control_rf,ntree=50)


# Predicciones y validación


predictionValidationProb_rf <- predict(rfModel,
                                       data_test, type = "prob")
```

```
auc_rf <- roc(data_test$target, predictionValidationProb_rf[["Yes"]],
              levels = unique(data_test[["target"]]))
auc_rf
roc_validation <- plot.roc(auc_rf, ylim=c(0,1), type = "S" ,
                           print.thres = T, main=paste('Validation AUC:',
                                                       round(auc_rf$auc[[1]], 2)))

# Matriz de confusión
prediction_rforest<- predict(rfModel, data_test, type = "raw")
confusion_matrix_rf<- confusionMatrix(table(prediction_rforest,
                                            data_test[["target"]]))
confusion_matrix_rf
```

Al igual que antes, los resultados obtenidos no son realmente buenos debido al desbalance de los datos, y tal y como pasó en el modelo con `rpart`, casi siempre realiza una predicción de 'No' o 0.

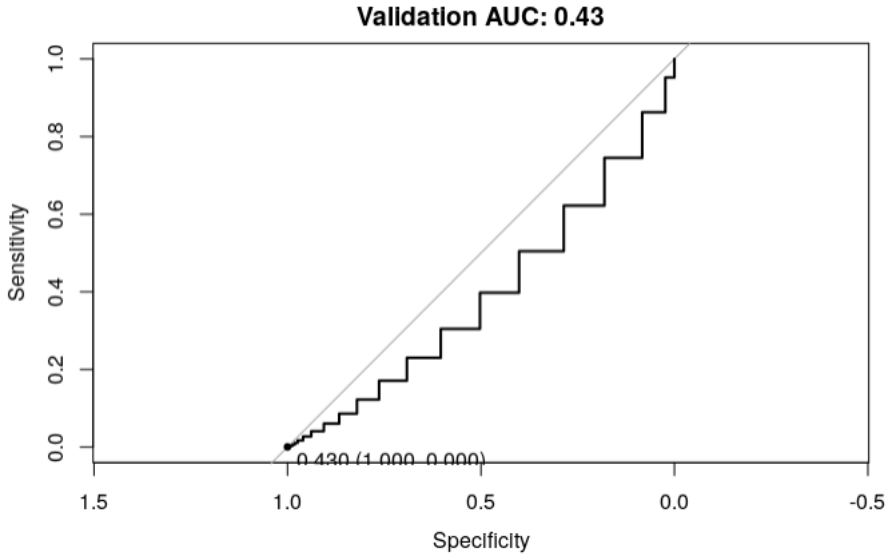


Figura 8: Curva ROC con rpart de los datos desbalanceados

```
Confusion Matrix and Statistics


prediction_rforest    No   Yes
               No  35673  3532
               Yes     2     0

               Accuracy : 0.9099
                 95% CI : (0.907, 0.9127)
    No Information Rate : 0.9099
    P-Value [Acc > NIR] : 0.5185

                  Kappa : -1e-04
 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.9999
            Specificity : 0.0000
         Pos Pred Value : 0.9099
         Neg Pred Value : 0.0000
             Prevalence : 0.9099
         Detection Rate : 0.9099
   Detection Prevalence : 0.9999
      Balanced Accuracy : 0.5000

       'Positive' Class : No
```

Figura 9: Matriz de confusión con rforest de los datos desbalanceados

Clasificación de datos balanceados al 50-50 %

En este caso, se va a proceder a describir el mismo procedimiento que se ha seguido con los datos desbalanceados y se va a comentar los cambios que ha habido en los resultados.

En primer lugar, se ha generado el modelo usando `rpart` y la métrica de validación `ROC`. Igual que en el caso anterior, se ha establecido un grid de parámetros para que se vaya probando a generar el modelo utilizando distintos valores de `cp`.

Tras generar el modelo y realizar el proceso de validación (se omite el código fuente debido a que es el mismo que en el proceso anterior, diferiendo en el conjunto de datos seleccionado) se han obtenido los siguientes resultados.

```
Area under the curve: 0.5576
```

Como se puede observar, ahora el resultado ha cambiado respecto a la prueba anterior. En este caso el valor AUC ha mejorado pero aún no sigue siendo bueno.

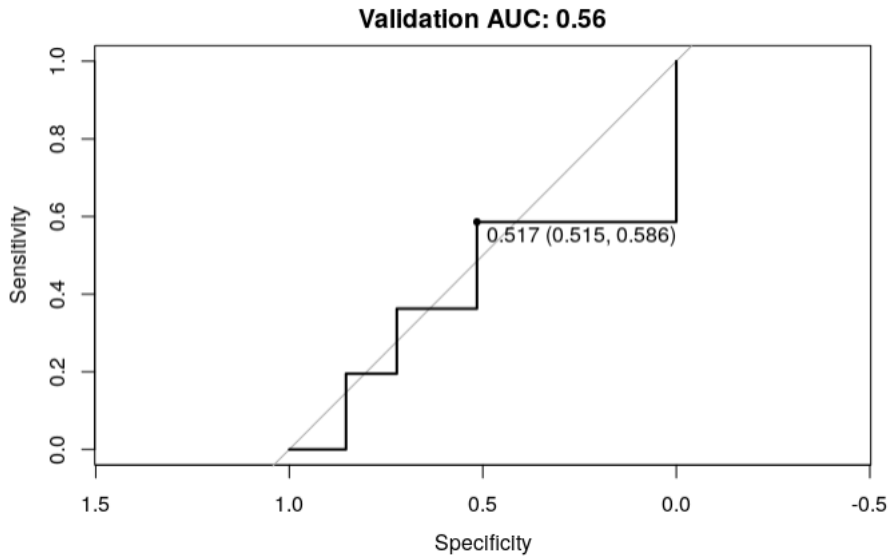


Figura 10: Curva ROC con rpart de los datos balanceados 1

Sin embargo, podemos comprobar en la matriz de confusión que en esta vez si ha intentado predecir considerablemente el valor *Yes* (Obvio debido a que en este caso, tenemos el mismo número de muestras tanto de *yes* como no *no*).

```
Confusion Matrix and Statistics


prediction_rpart   No  Yes
             No  4141 3426
             Yes 2924 3639

               Accuracy : 0.5506
                 95% CI : (0.5424, 0.5588)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.1012
 Mcnemar's Test P-Value : 3.234e-10

            Sensitivity : 0.5861
            Specificity : 0.5151
         Pos Pred Value : 0.5472
         Neg Pred Value : 0.5545
             Prevalence : 0.5000
         Detection Rate : 0.2931
   Detection Prevalence : 0.5355
      Balanced Accuracy : 0.5506

       'Positive' Class : No
```

Figura 11: Matriz de confusión con rpart de los datos balanceados 1

También se puede observar las variables más importantes por las que el modelo ha realizado la clasificación.

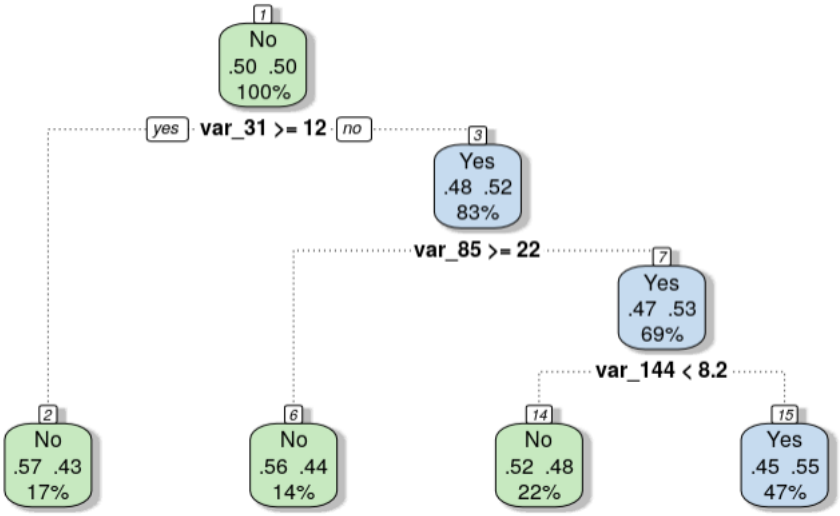


Figura 12: Matriz de confusión con rpart de los datos balanceados 1

Observando la figura anterior, vemos que el modelo ha realizado la primera partición teniendo en cuenta el valor de la variable 31 y 12, y así sucesivamente con el resto de valores.

A continuación, se ha probado a construir un modelo `rf`(randomForest). (Al igual que en el caso anterior, se omite el código debido a que es el mismo que el utilizado en el conjunto de datos sin balancear).

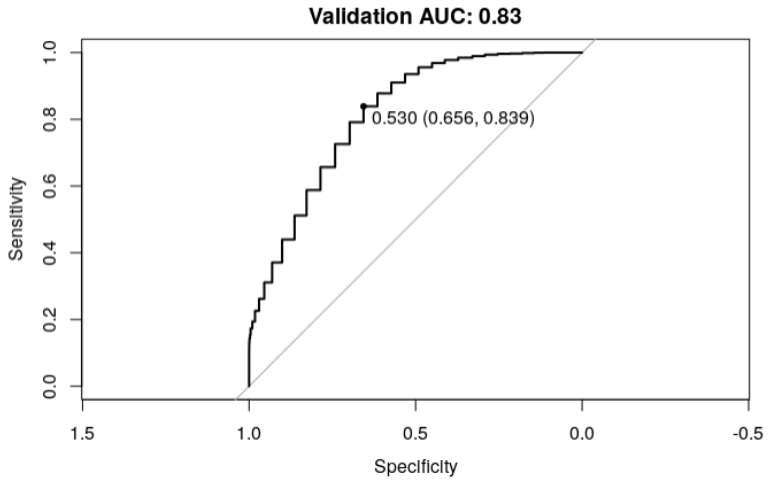


Figura 13: Curva ROC con rf de los datos balanceados 1

En este caso podemos comprobar como el modelo ha mejorado sustancialmente en su efectividad. Pasamos de tener un valor **0.56** usando `rpart` a **0.83** usando `rf`.

Si comprobamos la matriz de confusión, vemos como en este caso las predicciones entre clases están repartidas y son más acertadas.

```
Confusion Matrix and Statistics


prediction_rforest   No  Yes
               No  5436 2015
               Yes 1629 5050

               Accuracy : 0.7421
                 95% CI : (0.7348, 0.7493)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.4842
 Mcnemar's Test P-Value : 1.796e-10

            Sensitivity : 0.7694
            Specificity : 0.7148
         Pos Pred Value : 0.7296
         Neg Pred Value : 0.7561
             Prevalence : 0.5000
         Detection Rate : 0.3847
   Detection Prevalence : 0.5273
      Balanced Accuracy : 0.7421

       'Positive' Class : No
```

Figura 14: Matriz de confusión con rf de los datos balanceados 1

Clasificación de datos balanceados al 60-40 %

Este nuevo conjunto de datos tiene un balance del 60 % de la clase *No* y un 40 % de la clase *yes*.

A continuación se va a observar el comportamiento de este balance respecto a los modelos de predicción anteriormente descritos utilizando los mismos parámetros.

Comentar en primer lugar, que el resultado obtenido utilizando `rpart` ha sido extraño, ya que la proporción del balance es parecida al modelo anterior y los resultados son diferentes.

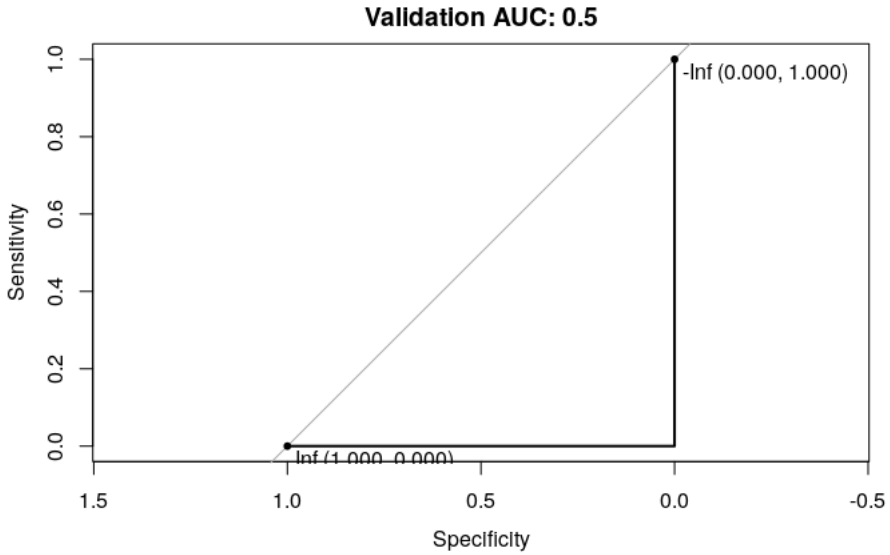


Figura 15: Curva ROC con rpart de los datos balanceados 2

```
Confusion Matrix and Statistics


prediction_rpart    No   Yes
             No  21196 14131
             Yes     0     0

               Accuracy : 0.6
                 95% CI : (0.5949, 0.6051)
    No Information Rate : 0.6
    P-Value [Acc > NIR] : 0.5023

                  Kappa : 0
 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 1.0
            Specificity : 0.0
         Pos Pred Value : 0.6
         Neg Pred Value : NaN
             Prevalence : 0.6
         Detection Rate : 0.6
   Detection Prevalence : 1.0
      Balanced Accuracy : 0.5

       'Positive' Class : No
```

Figura 16: Matriz de confusión con rpart de los datos balanceados 2

Como se puede observar en la figura anterior, el modelo ha vuelto a predecir en todas las ocasiones el valor *No*.

Sin embargo, empleando el método **rf** se han obtenido resultados bastantes buenos.

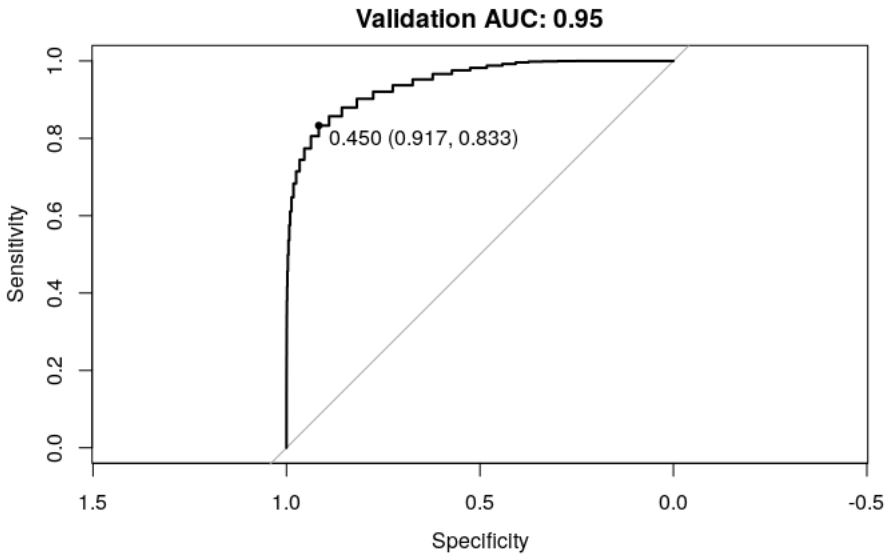


Figura 17: Curva ROC con rf de los datos balanceados 2

Como se puede observar, este modelo ha dado un buen `AUC` de 0.95 y un `accuracy` de 0.87

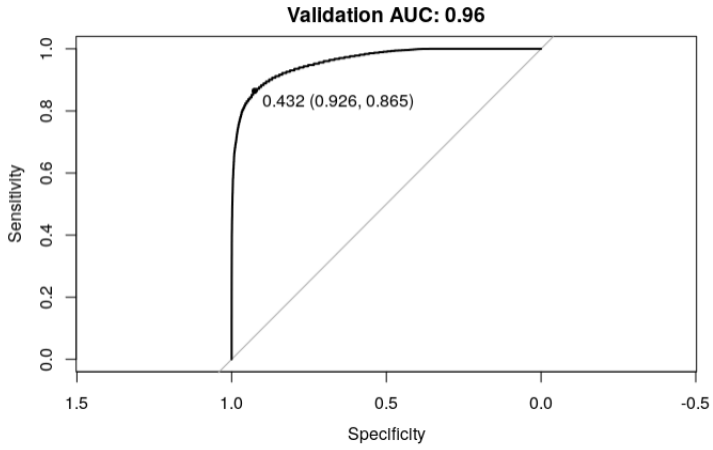
```
Confusion Matrix and Statistics


prediction_rforest    No   Yes
               No  20388  3473
               Yes   808 10658

               Accuracy : 0.8788
                 95% CI : (0.8754, 0.8822)
    No Information Rate : 0.6
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.7393
 Mcnemar's Test P-Value : < 2.2e-16

            Sensitivity : 0.9619
            Specificity : 0.7542
         Pos Pred Value : 0.8544
         Neg Pred Value : 0.9295
             Prevalence : 0.6000
         Detection Rate : 0.5771
   Detection Prevalence : 0.6754
      Balanced Accuracy : 0.8581

       'Positive' Class : No
```

Figura 18: Matriz de confusión con rf de los datos balanceados 2

Decir que para generar este modelo, se ha utilizado el mismo modelo que se comentó en la sección 1 con un parámetro ntree de 50.

Comentar que también se ha realizado pruebas usando diferentes parámetros y el mejor resultado que se ha obtenido ha sido con un `ntree=200`

Como resultado se ha obtenido un valor de **0.96** de `AUC` y un **89 %** de `acc`.




```
Confusion Matrix and Statistics


prediction_rforest    No   Yes
               No  20674  3292
               Yes   522 10839

               Accuracy : 0.892
                 95% CI : (0.8888, 0.8953)
    No Information Rate : 0.6
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.7675
 Mcnemar's Test P-Value : < 2.2e-16

            Sensitivity : 0.9754
            Specificity : 0.7670
         Pos Pred Value : 0.8626
         Neg Pred Value : 0.9541
             Prevalence : 0.6000
         Detection Rate : 0.5852
   Detection Prevalence : 0.6784
      Balanced Accuracy : 0.8712

       'Positive' Class : No
```

Figura 19: Mejor resultado con este balance 60-40 %

Este mismo modelo se ha testado con los data_test de `kaggle`. En primer lugar se han descargado los datos de https://www.kaggle.com/c/santander-customer-transaction-prediction/data) y se ha cargado.

Una vez se ha cargado el conjunto de test, se ha procedido a predecir la variable objetivo `target` y a almacenar dichos datos en un fichero *submission* formateado para subirlo a kaggle y observar la puntuación. El código que se ha empleado para ello es el siguiente:

```
# Kaggle test


# Se cargan los datos
kaggle_test <- read_csv('data/kaggle_test.csv')
```

```
# Comprobamos la dimensión de los datos de test
dim(kaggle_test)
```

```
## [1] 200000    201
```

```
# Obtenemos la columna de ID_code
submission_id <-kaggle_test$ID_code


# Se realiza la predicción para el conjunto kaggle_test
prediction_rforest<- predict(rfModel2, kaggle_test, type = "raw")
```

```r
# Se crea un dataframe con las columnas de ID_code y target
submission <- data.frame(submission_id,prediction_rforest)

# Se modifica el nombre a las columnas, ya que por defecto coge el nombre de las va
colnames(submission)<-c("ID_code", "target")

# Se modifica la predicción Yes/No a 1/0
submission <- submission %>%
  mutate(target=as.numeric(ifelse(target == 'Yes',1,0)))

# Observamos el número de predicciones positivas y negativas
table(submission$target)
```

```
##
##      0      1
## 192207   7793
```

```r
prop.table(table(submission$target))
```

```
##
##        0        1
## 0.961035 0.038965
```

```r
# Se almacenan las predicciones formatedas en un fichero listo para enviar a kaggle
write_csv(submission, 'data/sample_submission.csv')
```

Tras subir el fichero *sample_submission.csv* a `kaggle` se ha obtenido una puntuación de **0.5**.

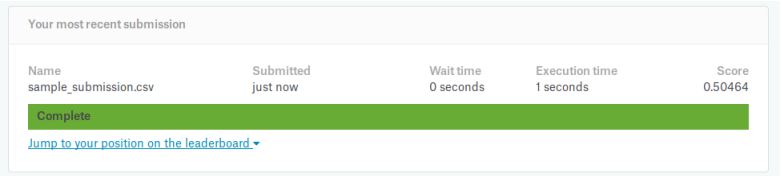


Figura 20: Puntuación de kaggle sobre las predicciones

Clasificación de datos balanceados al 62-38 %

Este nuevo ejemplo de balanceo tiene un porcentaje similar al anterior (60-40 %), pero como se comentó anteriormente en la etapa de preprocesamiento, la función SMOTE ha añadido una gran cantidad de muestras de ambas clases, y se ha pasado de tener una cantidad de 176640 a 282624 muestras respecto al conjunto anterior. El objetivo de este análisis es ver como se comporta el modelo ante estos nuevos cambios.

En primer lugar, se ha construido un modelo con **rpart**. En este caso ha ocurrido como en el caso de los datos desbalanceados y datos con balance 60-40 %. El modelo no ha generalizado correctamente y ha dado a todas las predicciones el valor 0.

Sin embargo, se ha obtenido un buen resultado utilizando el método **rf**.

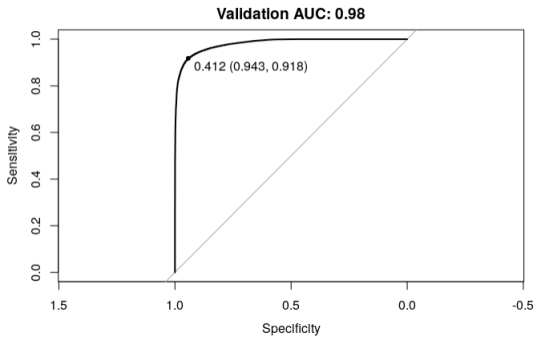


Figura 21: Curva ROC con rf en los datos balanceados 3.

Como podemos observar, la curva ROC es bastante buena, con un valor de `auc` de **0.98**

Si miramos la matriz de confusión, se puede comprobar que en general ha realizado muy buenas predicciones con un acierto del **92.5 %**.

```
Confusion Matrix and Statistics


prediction_rforest    No   Yes
               No  34834  3709
               Yes   494 17487

               Accuracy : 0.9256
                 95% CI : (0.9234, 0.9278)
    No Information Rate : 0.625
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.8364
 Mcnemar's Test P-Value : < 2.2e-16

            Sensitivity : 0.9860
            Specificity : 0.8250
         Pos Pred Value : 0.9038
         Neg Pred Value : 0.9725
             Prevalence : 0.6250
         Detection Rate : 0.6163
   Detection Prevalence : 0.6819
      Balanced Accuracy : 0.9055

       'Positive' Class : No
```

Figura 22: Matriz de confusión con rf en los datos balanceados 3.

Este nuevo modelo se ha utilizado para realizar el test de kaggle y tras subirlo se ha obtenido un resultado similar a la vez anterior.

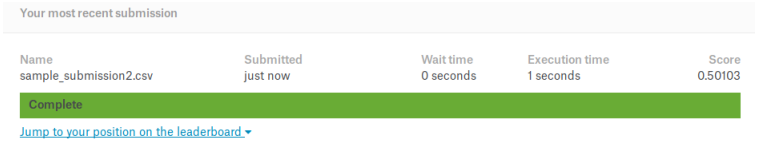


Figura 23: Puntuación en kaggle del modelo rf con un balance 62-38 %

Métodos y pruebas adicionales

Con fin de que la documentación no sea muy extensa, se ha obviado muchas pruebas que se han ido testeando con diferentes parámetros y métodos alternativos.

A continuación se muestra el código de los métodos alternativos que se han usado para construir el modelo de aprendizaje y los resultados obtenidos.

Linear discriminant analysis (lda)

```
# método LDA

lda_model <- train(target ~ .,
      data = data_train,
      method = "lda",
      prior = c(0.5, 0.5))



# Predicciones y validación

predictionValidationProb_lda <- predict(lda_model, data_test, type = "prob")
auc_lda <- roc(data_test$target, predictionValidationProb_lda[["Yes"]],
               levels = unique(data_test[["target"]]))
auc_lda
roc_validation <- plot.roc(auc_lda, ylim=c(0,1), type = "S" , print.thres = T,
                           main=paste('Validation AUC:',
```

```
                                        round(auc_lda$auc[[1]], 2)))


# Matriz de confusión
prediction_lda<- predict(lda_model, data_test, type = "raw")
confusion_matrix_lda<- confusionMatrix(table(prediction_lda, data_test[["target"]]))
confusion_matrix_lda
```

Los resultados obtenidos con este método en general no han sido demasiado buenos. El mejor resultado que se ha obtenido ha sido el siguiente.

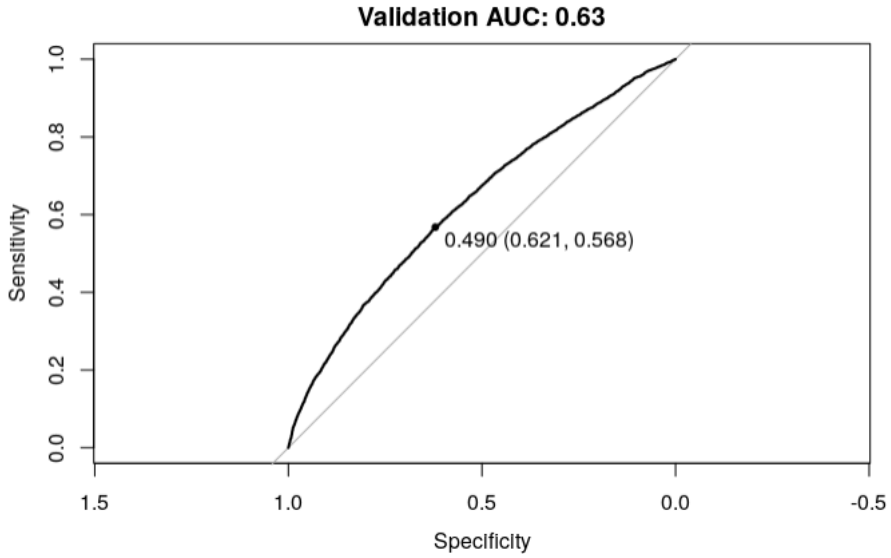


Figura 24: Curva ROC con lda usando los datos balanceados al 50-50 %

```
Confusion Matrix and Statistics


prediction_lda   No  Yes
           No  4235 2938
           Yes 2830 4127

               Accuracy : 0.5918
                 95% CI : (0.5836, 0.5999)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.1836
 Mcnemar's Test P-Value : 0.1589

            Sensitivity : 0.5994
            Specificity : 0.5841
         Pos Pred Value : 0.5904
         Neg Pred Value : 0.5932
             Prevalence : 0.5000
         Detection Rate : 0.2997
   Detection Prevalence : 0.5076
      Balanced Accuracy : 0.5918

       'Positive' Class : No
```

Figura 25: Matriz de confusión con lda usando los datos balanceados al 50-50 %

Generalized linear model (glm)

Los resultados obhttps://topepo.github.io/caret/train-models-by-tag.htmltenidos con este método en general tampoco han sido demasiado buenos. El mejor resultado que se ha obtenido ha sido de un `auc` **0.62** y **0.58 %** de `acc`.

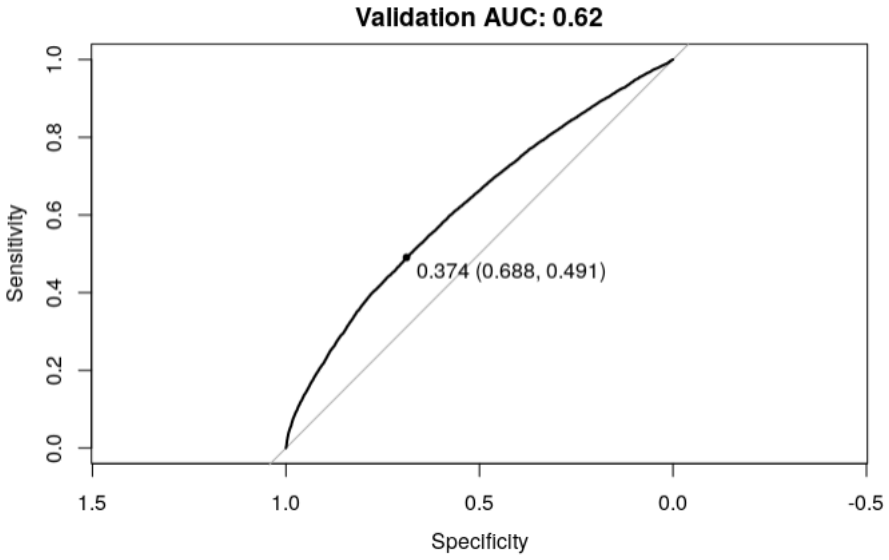


Figura 26: Curva ROC con glm usando los datos balanceados al 50-50 %

```
Confusion Matrix and Statistics


prediction_glm    No   Yes
           No  18276 10837
           Yes  2920  3294

               Accuracy : 0.6106
                 95% CI : (0.6055, 0.6157)
    No Information Rate : 0.6
    P-Value [Acc > NIR] : 2.432e-05

                  Kappa : 0.1052
 Mcnemar's Test P-Value : < 2.2e-16

            Sensitivity : 0.8622
            Specificity : 0.2331
         Pos Pred Value : 0.6278
         Neg Pred Value : 0.5301
             Prevalence : 0.6000
         Detection Rate : 0.5173
   Detection Prevalence : 0.8241
      Balanced Accuracy : 0.5477

       'Positive' Class : No
```

Figura 27: Matriz de confusión con glm usando los datos balanceados al 50-50 %

5. Conclusiones

Tras haber realizado toda la labor de preprocesamiento y clasificación de los datos, se aportan las siguientes conclusiones que se han obtenido tras haber realizado todo el proceso.

- El conjunto de datos que se ha analizado presenta un gran número de variables y todas son numéricas. Al visualizar los datos no se puede extraer ninguna información que se pueda interpretar a primera vista.

- Los datos presentan una baja correlación con la variable objetivo `target` por lo que a priori no se ha obtenido información relevante tras las exploración de los datos

- Se ha comprobado como inicialmente los datos estaban bastantes desbalanceados y los modelos de predicción siempre predecían el valor de la clase mayoritaria.

- Se ha observado como los modelos de predicción son significativamente mejores (en conjunto de validación) tras haber realizado un balanceo de los datos.

- Dada la gran cantidad de datos y variables (aún habiéndolos reducido) hay métodos de clasificación que no han sido computacionalmente viables y se ha tenido que cancelar su ejecución, como por ejemplo `SVM` o `KNN`.

- En general no se han obtenido buenos resultados utilizando regresión con `rpart`, o con métodos como `lda` o `glm`. Sin duda, el método que ha dado mejores resultados ha sido `rf` (randomforest).

- El mejor resultado de clasificación que se ha obtenido ha sido utilizando randomforest sobre un conjunto de datos balanceados. Se ha obtenido un `auc` de **0.98** , y un precisión del **92.5 %**

- Se ha utilizado el mejor modelo para predecir el conjunto de test de `kaggle` y se ha obtenido una puntuación de **0.5**. Esta gran diferencia obtenida entre el valor obtenido en validación, y el valor real obtenido con el test de `kaggle`, es debido a que al realizar el balanceo de datos, se han introducido muchas muestras que no han generalizado lo suficiente y esto ha sesgado el modelo, siendo bastante efectivo para el conjunto de validación (obvio ya que es obtenido de las muestras de los datos ya balanceados), y menos efectivo para el conjunto de test.

Referencias

[1] Diego Calvo, eliminar NA o valores nulos de R , disponible en http://www.diegocalvo.es/eliminar-na-o-valores-nulos-en-r/

[2] RDocumentation, SMOTE algorithm for unbalanced classification problems, disponible en https://www.rdocumentation.org/packages/DMwR/versions/0.4.1/topics/SMOTE.

[3] Jason Brownlee,Tune Machine Learning Algorithms in R, disponible en https://machinelearningmastery.com/tune-machine-learning-algorithms-in-r/.

[4] José R. Berrendero, RPubs, Introducción a CARET, disponible en https://rpubs.com/joser/caret.

[5] topepo.github.io, The caret package, disponible en https://topepo.github.io/caret/train-models-by-tag.html.