Tutorial sobre Tareas de Clasificación No Balanceada

Seminario Permanente de Formación en Inteligencia Artificial Aplicada a la Defensa

Alberto Fernández Hilario alberto@decsai.ugr.es
Universidad de Granada - DaSCI

01 junio 2020

Contents

Introducción **2**

Instalar los paquetes requeridos **2**

Paso uno. Obtener los datos **3**
Cargar datos de la manera más fácil . . . 4
Cargar desde CSV . . . 4

Paso dos. Conocer los datos: Resumir el conjunto de datos **4**
Dimensiones del conjunto de datos . . . 5
Tipos de Atributos . . . 5
Vistazo rápido a los Datos . . . 5
Niveles de la clase . . . 6
Resumen estadístico . . . 6

Visualizar el conjunto de datos **6**
Gráficas univariadas . . . 6
Gráficos multivariados . . . 8

Evaluar algunos algoritmos **10**
Validación de modelos . . . 11
Construir modelos . . . 11
Aprender y almacenar diferentes modelos de los datos . . . 12
Selecciona el mejor modelo . . . 19

Algoritmos avanzados en clasificación no balanceada: Utilizando el paquete "imbalance" **22**

Comentarios finales **49**

Este es un Cuaderno o NoteBook de R que incluye una tutorial práctico sobre "*Clasificación No Balanceada*" dentro del "*Seminario Permanente de Formación en Inteligencia Artificial Aplicada a la Defensa*" conjunto entre el Mando de Adiestramiento y Doctrina y la Universidad de Granada.

Existen bloques de texto (como este) y bloques de código (con instrucciones en R). Cuando se ejecuta el código dentro del cuaderno, los resultados aparecen debajo del mismo.

Introducción

Cualquier conjunto de datos con una distribución de clases desigual está técnicamente desequilibrado. Sin embargo, se dice que un conjunto de datos está desequilibrado cuando hay una desproporción significativa, o en algunos casos extrema, entre el número de ejemplos de cada clase del problema. En otras palabras, el desequilibrio de clases se produce cuando el número de ejemplos que representan una clase es mucho menor que los de las otras clases. Por lo tanto, una o más clases pueden estar subrepresentadas en el conjunto de datos. Una definición tan simple ha atraído mucha atención de los investigadores y profesionales debido al número de aplicaciones en el mundo real en las que los datos en bruto recopilados cumplen esta definición.

En este tutorial paso a paso, se pretende que usted:

1. Obtenga los paquetes más útiles para abordar la clasificación desequilibrada con el aprendizaje automático en R.
2. Cargue un conjunto de datos y entienda su estructura usando resúmenes estadísticos y visualización de datos.
3. Cree varios modelos de aprendizaje automático en sinergia con técnicas de preprocesamiento para datos desequilibrados, elegir los mejores y crear confianza en que el rendimiento predictivo es fiable.

Tal como se indicó anteriormente, para facilitar esta sesión "práctica", el documento contiene trozos de código fuente de R que pueden ser ejecutados directamente. Intente ejecutar este primer trozo haciendo clic en el botón *Run* dentro del trozo o colocando el cursor dentro de él y pulsando *Cmd+Shift+Enter* (*Ctrl+Shift+Enter* en Windows).

```
print("Welcome to your first R NoteBook")
```

```
## [1] "Welcome to your first R NoteBook"
```

Si quiere añadir un nuevo trozo de código, sólo tiene que hacer clic en el botón *Insertar trozo* (*Insert Chunk*) de la barra de herramientas o pulsando *Cmd+Opción+I* (o *Ctrl+Alt+I* en Windows). De esta manera, puede añadir su propio código si así lo necesita.

Cuando guarde el cuaderno, se guardará un archivo HTML con el código y la salida junto a él (haz clic en el botón *Previsualizar* (*preview*) o pulsando *Cmd+Mayús+K* para obtener una vista previa del archivo HTML). En realidad, esta es una buena manera de compilar todas las tareas desarrolladas durante el tutorial.

Aquí hay un resumen de lo que se pretende cubrir en este *Tutorial sobre Clasificación No Balancedad*:

1. Instalación de la plataforma R.
2. Cargando el conjunto de datos.
3. Resumiendo el conjunto de datos.
4. Visualizando el conjunto de datos.
5. Evaluar algunos algoritmos.
6. Comparación entre diferentes soluciones.

Tómese su tiempo. Trabaje en cada paso.

Instalar los paquetes requeridos

Instale los paquetes que vamos a usar hoy. Los paquetes son complementos o bibliotecas de terceros que se pueden usar en R.

NOTA: Puede que necesite otros paquetes, pero Caret debería preguntarnos si queremos cargarlos. Si tiene problemas con los paquetes, puede instalar los paquetes de caret y todos los paquetes que pueda necesitar escribiendo lo siguiente (elimine el carácter de comentario #):

```
#install.packages("caret", dependencies=c("Depends", "Suggests"),repos = "http://cran.r-project.org")
```

Ahora, cargue los paquetes que va a usar en este tutorial, los paquetes de `caret` e `imbalance`, entre otros.

```
library(caret)
```

```
## Loading required package: lattice
```

```
## Loading required package: ggplot2
```

```
library(imbalance)
```

```
## Warning: package 'imbalance' was built under R version 3.6.2
```

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(pROC)
```

```
## Type 'citation("pROC")' for a citation.
```

```
##
## Attaching package: 'pROC'
```

```
## The following objects are masked from 'package:stats':
##
##     cov, smooth, var
```

```
library(tidyr)
```

```
## Warning: package 'tidyr' was built under R version 3.6.2
```

```
library(rlist)
library(Rtsne)
```

Como quizá ya sabe, el paquete `caret` proporciona una interfaz consistente en cientos de algoritmos de Machine Learning y ofrece métodos útiles y convenientes para la visualización de datos, el remuestreo de datos, la puesta a punto de modelos y la comparación de modelos, entre otras características. Es una herramienta imprescindible para los proyectos de aprendizaje automático en R.

Una de sus ventajas para el uso del `caret` es que integra directamente el uso de técnicas de preprocesamiento al establecer el método de control (se describirámás adelante) para la etapa de entrenamiento.

Para más información sobre el paquete R caret, ver la página web del paquete caret.

Paso uno. Obtener los datos

Para establecer un escenario controlado, utilizaremos dos datos artificiales diferentes, a saber, los datos de "círculo" y "subclúster", que pueden encontrarse en diferentes estudios sobre clasificación desequilibrada. Se trata de problemas binarios, tanto en términos de atributos como de clases.

Además, podemos cargar algunos datos del paquete de `imbalance`, como el del glass0 (véase Wikipedia). En este caso, podemos observar el comportamiento de los diferentes métodos en un caso de estudio real.

Esto es lo que vamos a hacer en este paso:

0. Cargar los datos del glass0 de la manera más fácil (con el paquete `imbalance`). Esto es un adelante de la segunda parte del tutorial exclusiva de la citada biblioteca.
1. Cargar los datos de *círculo* desde CSV.
2. Separar los datos en un conjunto de datos de entrenamiento y un conjunto de datos de validación.

Para los datos de "*subcluster*", siga el mismo procedimiento que en el caso de "*círculo*".

Cargar datos de la manera más fácil

Afortunadamente, el paquete de desequilibrio nos proporciona el conjunto de datos de *Glass0*, entre otros. Cargue el conjunto de datos de la siguiente manera:

```
# adjunto el dataset glass0 al entorno
data(glass0)
# renombre el dataset por comodidad
dataset <- glass0
```

Ahora tiene el conjunto de datos *glass0* cargado en R y accesible a través de la variable de `dataset`.

Una ventaja de utilizar un nombre de variable genérico, como en este caso, es porque resulta útil si se desea copiar y pegar código entre proyectos y el conjunto de datos siempre tiene el mismo nombre.

Cargar desde CSV

En caso que haya descargado el conjunto de datos previamente, puede que quieras cargar los datos directamente desde un archivo CSV. También lo podemos hacer desde un enlace web.

1. Descargue el conjunto de datos de los círculos (y subcluster) desde los enlaces.
2. Guarde el archivo como `circle.csv` (y `subclus`) en su directorio de proyecto.
3. Cargue el conjunto de datos del archivo CSV de la siguiente manera:

Otra alternativa hubiese sido leer el CSV directamente desde la dirección Web (URL) del siguiente modo:

```
# cargar el fichero CSV desde local o directorio Web
dataset <- read.csv("https://www.dropbox.com/s/06k36xri8c1bepf/circle.csv?raw=1",header = TRUE)
# poner el nomber de los atributos en los datos (deben conocerse a priori o estar en el fichero)
colnames(dataset) <- c("Att1", "Att2", "Class")
#para asegurar que aparece como la primera clase
dataset$Class <- factor(dataset$Class,levels=c("positive","negative"))
```

Ahora tiene los datos del *círculo* cargados en R y accesibles a través de la variable `dataset`.

Paso dos. Conocer los datos: Resumir el conjunto de datos

Ahora es el momento de echar un vistazo a los datos. Esta etapa se ha realizado parcialmente en otras sesiones prácticas del Curso. En cualquier caso, siempre es positivo dar un repaso a esta tarea tan importante dentro del ciclo de Ciencia de Datos.

En concreto, en este paso vamos a echar un vistazo a los datos de diferentes maneras:

1. Dimensiones del conjunto de datos.
2. Tipos de atributos.
3. Echar un vistazo a los datos en sí.
4. Niveles del atributo de clase.

5. Desglose de las instancias de cada clase.
6. Resumen estadístico de todos los atributos.

No se preocupe, cada inspección sobre los datos es un breve comando o instrucción. Se mostrarán por tanto comandos útiles que puede usar una y otra vez en futuros proyectos.

Dimensiones del conjunto de datos

Podemos hacernos una idea rápida de cuántas instancias (filas) y cuántos atributos (columnas) contienen los datos con la función `dim`.

```
# dimensions of dataset
dim(dataset)
```

```
## [1] 2390    3
```

Debería ver 2390 instancias y 3 atributos en el caso de los datos del "*círculo*".

Tipos de Atributos

Es una buena idea tener una idea de los tipos de los atributos. Podrían ser dobles (reales), enteros, cadenas, factores y otros tipos.

Conocer los tipos es importante ya que le dará una idea de cómo resumir mejor los datos que tiene y los tipos de transformaciones que podría necesitar para preparar los datos antes de modelarlos.

En este ejemplo, debería ver que todos los datos de entrada son dobles y que el valor de la clase es un factor.

```
# Chequear la estructura y tipo de cada atributo
str(dataset)
```

```
## 'data.frame':    2390 obs. of  3 variables:
##  $ Att1 : num  248 229 229 312 300 ...
##  $ Att2 : num  222 219 221 238 254 ...
##  $ Class: Factor w/ 2 levels "positive","negative": 1 1 1 1 1 1 1 1 1 ...
```

Vistazo rápido a los Datos

También es siempre una buena idea observar sus datos. Debería ver las primeras 5 filas de los datos de la siguiente manera:

```
# Observar las 5 primeras filas de datos
head(dataset)
```

```
##       Att1     Att2    Class
## 1 248.2755 221.7979 positive
## 2 228.5710 218.6520 positive
## 3 228.6188 220.5286 positive
## 4 311.8650 238.2219 positive
## 5 300.3109 253.5445 positive
## 6 309.0332 247.3192 positive
```

Niveles de la clase

La variable de clase es un factor. Un factor es una clase que tiene múltiples etiquetas de clase o niveles. Veamos los niveles. Fíjese en cómo podemos referirnos a un atributo por su nombre como una propiedad del conjunto de datos. En los resultados podemos ver que la clase tiene 2 etiquetas diferentes:

```
# listas los niveles para la clase
levels(dataset$Class)
```

```
## [1] "positive" "negative"
```

Este es un problema de clasificación binaria.

Resumen estadístico

Ahora, finalmente, podemos echar un vistazo a un resumen de cada atributo.

Esto incluye la media, los valores mínimo y máximo, así como algunos percentiles (25, 50 o media y 75, e.g., los valores en estos puntos si ordenamos todos los valores de un atributo). Podemos ver aquí la distribución desigual entre las clases: 2335 vs. 55 (datos del *círculo*). Esto se confirma calculando el Ratio de Desequilibrio (IR).

```
# resumir la distribución de los atributos
summary(dataset)
```

```
##       Att1              Att2              Class
##  Min.   :  4.442   Min.   :  0.5926   positive:  55
##  1st Qu.:118.820   1st Qu.:118.3459   negative:2335
##  Median :254.135   Median :249.1955
##  Mean   :254.997   Mean   :253.1674
##  3rd Qu.:389.010   3rd Qu.:389.8631
##  Max.   :508.239   Max.   :505.8055
```

```
imbalanceRatio(dataset)
```

```
## [1] 0.0235546
```

Visualizar el conjunto de datos

Ahora tenemos una idea básica sobre los datos. Necesitamos ampliarla con algunas visualizaciones.

Vamos a ver dos tipos de gráficos:

1. Gráficos univariados para entender mejor cada atributo.
2. Gráficas multivariadas para entender mejor las relaciones entre los atributos.

Gráficas univariadas

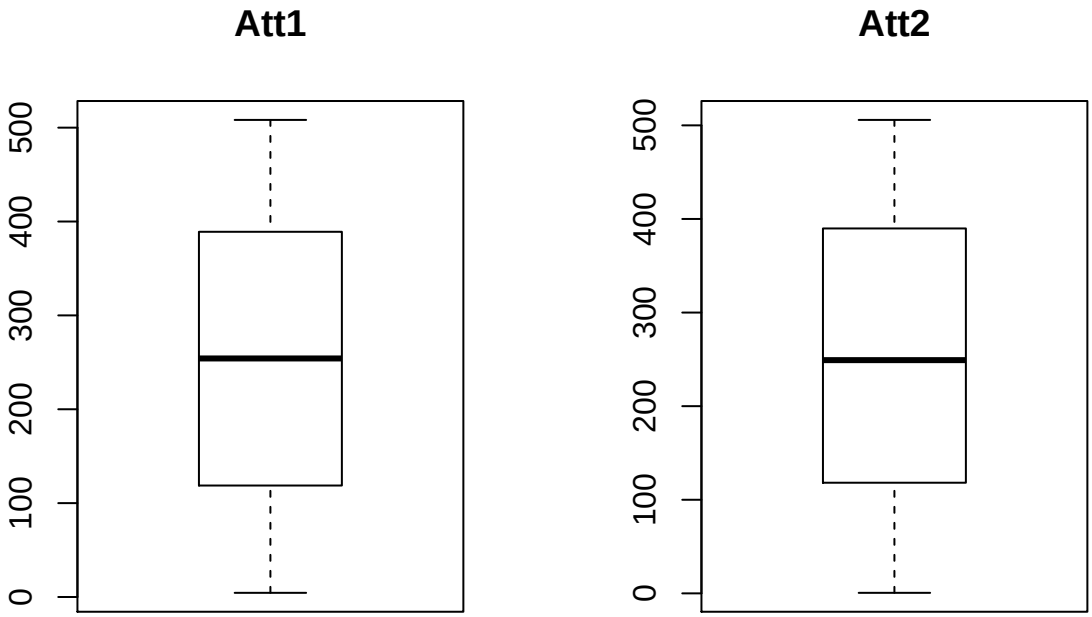
Empezamos con algunos gráficos univariados, es decir, gráficos de cada variable individual.

Es útil para la visualización tener una forma de referirse sólo a los atributos de entrada y sólo a los atributos de salida. Configurémoslo y llamemos a los atributos de entrada x y al de salida (o clase) y.

```
# Dividir entre variables de entrada y salida
x <- dataset[,1:2]
y <- dataset[,3]
```

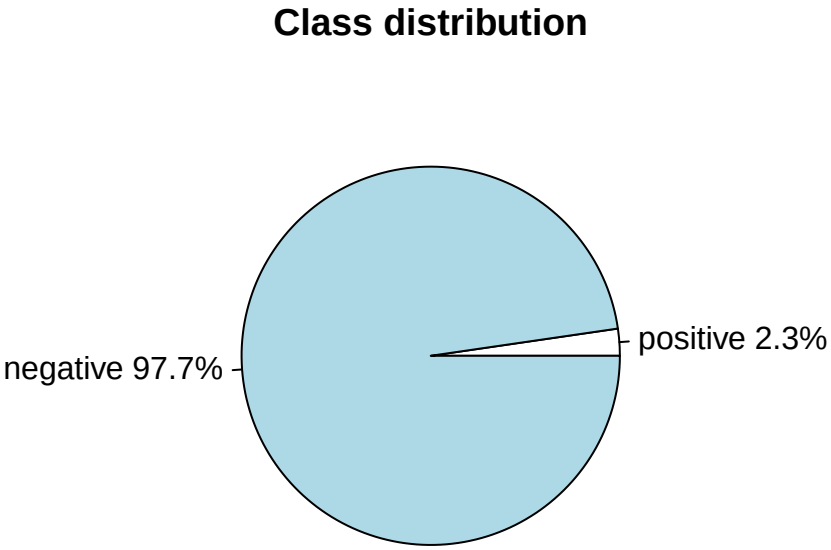
Dado que las variables de entrada son numéricas, podemos crear gráficos de caja y bigote (*boxplots*) de cada una. Esto nos da una idea mucho más clara de la distribución de los atributos de entrada:

```
# boxplot para cada atributo en una imagen
par(mfrow=c(1,2))
  for(i in 1:2) {
  boxplot(x[,i], main=names(dataset)[i])
}
```



También podemos crear un *barplot* diagrama de barras (mejor un gráfico de queso) de la variable de la clase para obtener una representación gráfica de la distribución de la clase. Esto confirma lo que aprendimos en la última sección, que las instancias están distribuidas de manera desigual entre las dos clases.

```
# simple barplot para ver las clases
# plot(y)

# Un diagrama de queso es más apropiado
n_classes <- c(sum(y=="positive"),sum(y=="negative"))
pct <- round(n_classes/sum(n_classes)*100,digits=2)

lbls <- levels(dataset$Class)
lbls <- paste(lbls, pct) # add percents to labels
lbls <- paste(lbls,"%",sep="") # ad % to labels

pie(n_classes,labels = lbls, main="Class distribution")
```

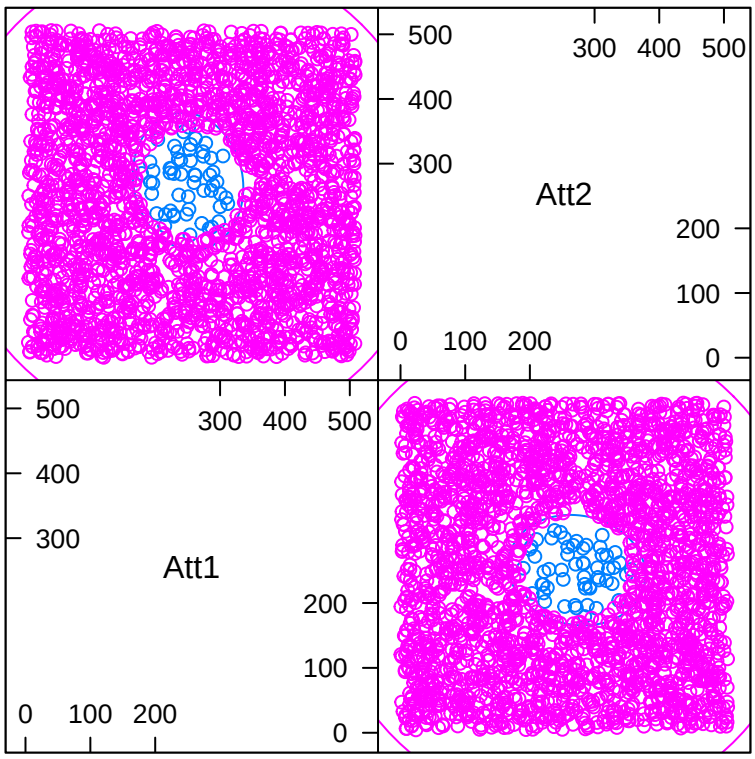


Gráficos multivariados

Ahora podemos ver las interacciones entre las variables.

Primero veamos los diagramas de dispersión de todos los pares de atributos y coloreemos los puntos por clase. Además, como los gráficos de dispersión muestran que los puntos de cada clase están generalmente separados, podemos dibujar elipses alrededor de ellos. Nuestro objetivo es ver las relaciones entre los atributos de entrada (tendencias) y entre los atributos y los valores de las clases (elipses). En este caso los datos son tan simples que no hay una conclusión clara.

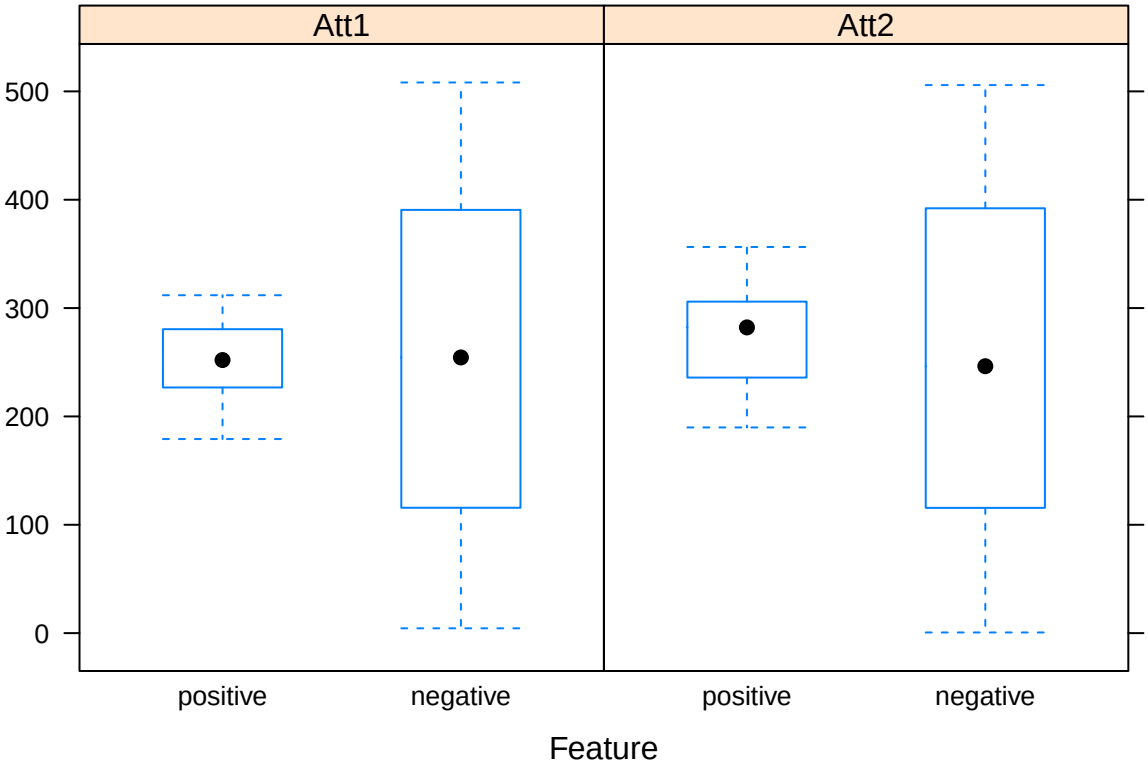
```
# scatterplot matrix
featurePlot(x=x, y=y, plot="ellipse")
```



Scatter Plot Matrix

También podemos mirar los gráficos de cajas y bigotes de cada variable de entrada de nuevo, pero esta vez divididos en gráficos separados para cada clase. Esto puede ayudar a determinar las separaciones lineales obvias entre las clases.

```
# boxplots para cada atributo
featurePlot(x=x, y=y, plot="box")
```



Las gráficas de dispersión pueden dar una gran idea de lo que se está tratando: puede ser interesante ver cuánto afecta una variable a otra. En otras palabras, se pretende ver si hay alguna correlación entre dos variables. Puede hacer gráficos de dispersión con el paquete ggvis, por ejemplo.

```
# Load in `ggvis`
library(ggvis)
```

```
##
## Attaching package: 'ggvis'
```

```
## The following object is masked from 'package:ggplot2':
##
##     resolution
```

```
# Dataset scatter plot
dataset %>% ggvis(~Att1, ~Att2, fill = ~Class) %>% layer_points()
```

Renderer: SVG | Canvas

Download

Evaluar algunos algoritmos

Ahora es el momento de crear algunos modelos de los datos y estimar su capacidad de predicción sobre datos no vistos.

Esto es lo que vamos a cubrir en este paso:

1. Preparar el conjunto de prueba o test a partir de la validación *hold-out*. No es la mejor práctica, pero nos servirá para propósitos de didácticos.

2. Aplicar diferentes enfoques de preprocesamiento a los datos de entrenamiento. El test debe mantenerse sin cambios.
3. Construir un modelo kNN para predecir la clase a partir de cada uno de los datos de entrenamiento.
4. Comparar y seleccionar la mejor metodología.

Validación de modelos

Como ya se ha comentado, nos centraremos en una partición tipo *hold-out*. De esta manera, tendremos sólo un entrenamiento y una partición de prueba, lo que puede causar un sesgo en nuestras conclusiones. Sin embargo, se muestra cómo proceder con un procedimiento adecuado de validación cruzada.

Debemos reajustar la semilla de números aleatorios antes de cada ejecución para asegurarnos de que la evaluación de cada algoritmo se realiza utilizando exactamente las mismas divisiones de datos. Esto asegura que los resultados sean directamente comparables.

```
set.seed(42) #Para asegurar la misma salida (en el mismo equipo)

#Una forma fácil de crear "particiones de datos":
trainIndex <- createDataPartition(dataset$Class, p = .75,list = FALSE, times = 1)

trainData <- dataset[ trainIndex,]
testData  <- dataset[-trainIndex,]

#Chequear el IR para asegurar una partición estratificada
imbalanceRatio(trainData)
```

```
## [1] 0.0239726
```

```
imbalanceRatio(testData)
```

```
## [1] 0.02229846
```

```
#Ad hoc FCV
#testIndices <- createFolds(dataset$Class, k=5)
#First partition
#dataTrain <- dataset[-testIndices[[1]],]
#dataTest  <- dataset[testIndices[[1]],]
```

Construir modelos

No sabemos qué algoritmos serían buenos en este problema o qué configuraciones usar.

Evaluemos 3 metodologías diferentes con kNN:

1. Conjunto de datos originales (datos en bruto)
2. Técnicas de muestreo triviales (sobre y submuestreo aleatorio).
3. Sobremuestreo SMOTE.

En primer lugar, necesitamos crear dos funciones auxiliares para las etapas de aprendizaje y predicción. Por favor, tenga en cuenta que estamos optimizando el parámetro k de kNN a través de un procedimiento "*grid search*". Si eliminamos esta parte, también podemos construir una función general para cualquier posible algoritmo de clasificación.

1) Aprendizaje

```
# a) Learning function
learn_model <-function(dataset, classifier, hyperp, ctrl, message){
```

```
  model.fit <- train(Class ~ ., data = dataset, method = classifier, trControl = ctrl, preProcess =
                       c("center","scale"), metric="ROC",tuneGrid = hyperp)
  model.pred <- predict(model.fit,newdata = dataset)
  #Get the confusion matrix to see accuracy value and other parameter values
  model.cm <- confusionMatrix(model.pred, dataset$Class,positive = "positive")
  model.probs <- predict(model.fit,newdata = dataset, type="prob")
  model.roc <- roc(dataset$Class,model.probs[,"positive"],color="green")
  return(model.fit)
}
```

2) Predicción:

```
# b) Estimation function
test_model <-function(dataset, model.fit,message){
  model.pred <- predict(model.fit,newdata = dataset)
  #Get the confusion matrix to see accuracy value and other parameter values
  model.cm <- confusionMatrix(model.pred, dataset$Class,positive = "positive")
  print(model.cm)
  model.probs <- predict(model.fit,newdata = dataset, type="prob")
  model.roc <- roc(dataset$Class,model.probs[,"positive"])
  #print(knn.roc)
  plot(model.roc, type="S", print.thres= 0.5,main=c("ROC Test",message),col="blue")
  #print(paste0("AUC Test ",message,auc(model.roc)))
  return(model.cm)
}
```

Aprender y almacenar diferentes modelos de los datos

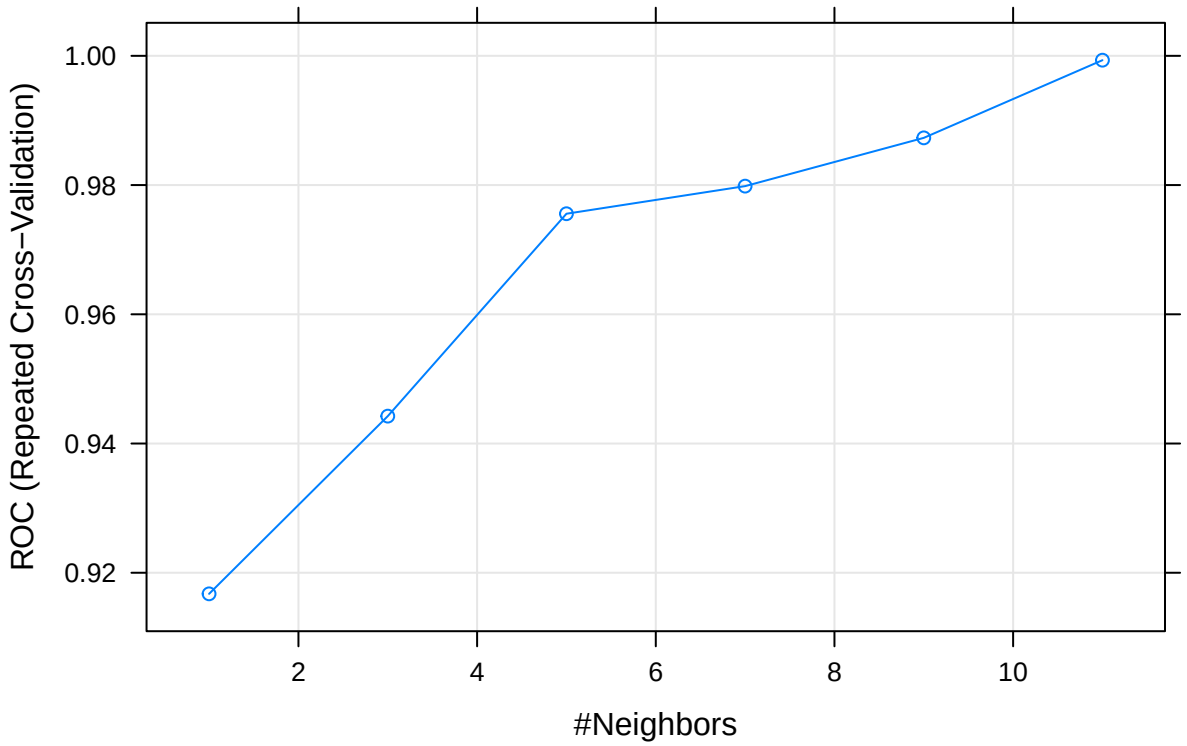
Primero, comprobamos la obtención de los resultados de los datos originales. Por favor, recuerde que una validación interna de CV se utiliza para establecer los mejores parámetros para el modelo de kNN (véase más arriba). Esto se indica en la llamada de `trainControl`.

```
#Execute model ("raw" data)
ctrl <- trainControl(method="repeatedcv",number=5,repeats = 3, classProbs=TRUE,summaryFunction = twoCla
#ctrl <- trainControl(method="none",classProbs=TRUE,summaryFunction = twoClassSummary)
classifier = "knn"
hyperp = data.frame(k = c(1,3,5,7,9,11))
#hyperp = data.frame(k = 3)

#classifier = "rpart"
#hyperp = data.frame(cp = 0.05)
model.raw <- learn_model(trainData,classifier,hyperp,ctrl,"RAW ")
```

```
## Setting levels: control = positive, case = negative
```

```
## Setting direction: controls > cases
```

```
#Podemos decidir representar los resultados del Grid search de los hiperparámetros del modelo
if (dim(hyperp)[1] > 1){
  plot(model.raw,main="Grid Search RAW")
}
```




```
print(model.raw)
```

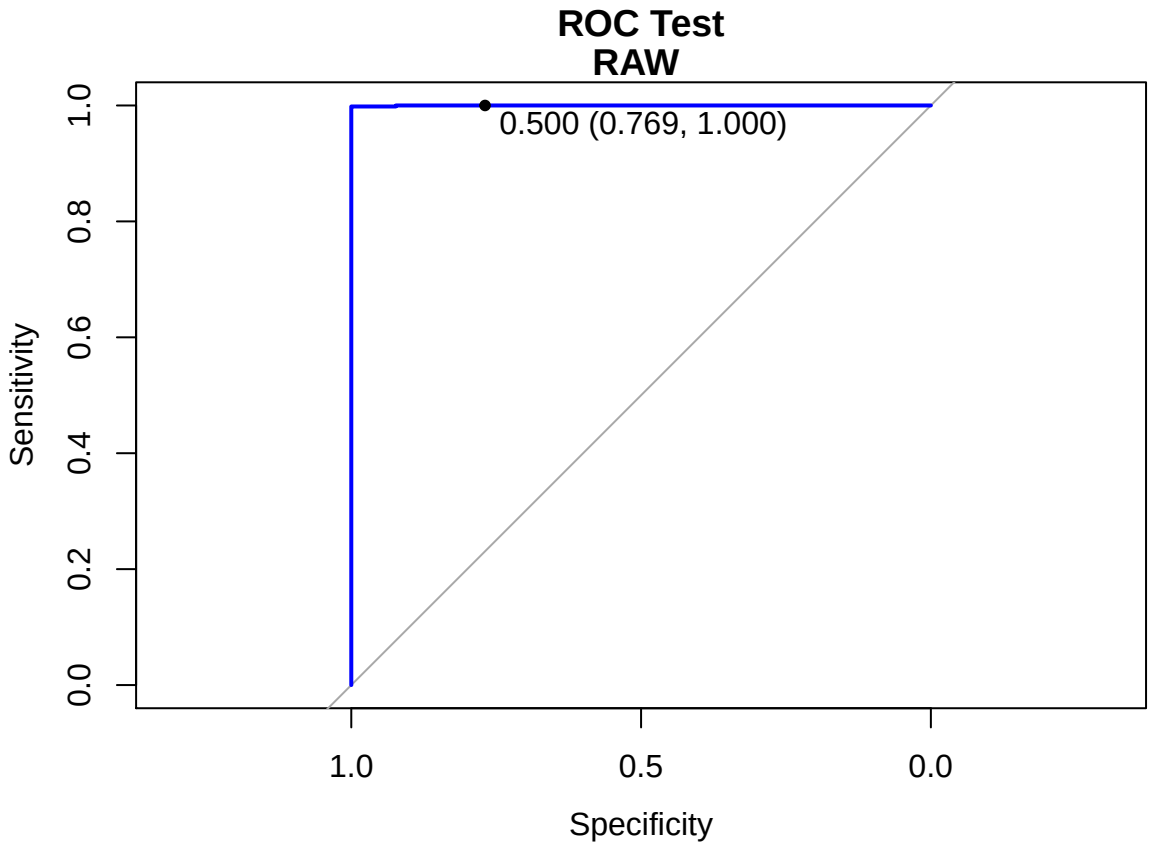
```
## k-Nearest Neighbors
##
## 1794 samples
##    2 predictor
##    2 classes: 'positive', 'negative'
##
## Pre-processing: centered (2), scaled (2)
## Resampling: Cross-Validated (5 fold, repeated 3 times)
## Summary of sample sizes: 1436, 1434, 1435, 1436, 1435, 1435, ...
## Resampling results across tuning parameters:
##
##   k   ROC        Sens       Spec
##    1  0.9167365  0.8351852  0.9982879
##    3  0.9442402  0.7861111  0.9992386
##    5  0.9755631  0.7240741  1.0000000
##    7  0.9798329  0.6833333  1.0000000
##    9  0.9873061  0.6287037  1.0000000
##   11  0.9993274  0.6212963  1.0000000
##
## ROC was used to select the optimal model using the largest value.
## The final value used for the model was k = 11.
```

```
cm.raw <- test_model(testData,model.raw,"RAW ")
```

```
## Confusion Matrix and Statistics
##
```

```
##           Reference
## Prediction positive negative
##   positive       10        0
##   negative        3      583
##
##                Accuracy : 0.995
##                  95% CI : (0.9854, 0.999)
##     No Information Rate : 0.9782
##     P-Value [Acc > NIR] : 0.0009625
##
##                   Kappa : 0.867
##
##  Mcnemar's Test P-Value : 0.2482131
##
##             Sensitivity : 0.76923
##             Specificity : 1.00000
##          Pos Pred Value : 1.00000
##          Neg Pred Value : 0.99488
##              Prevalence : 0.02181
##          Detection Rate : 0.01678
##    Detection Prevalence : 0.01678
##       Balanced Accuracy : 0.88462
##
##        'Positive' Class : positive
##
```

```
## Setting levels: control = positive, case = negative
## Setting direction: controls > cases
```



Ahora, incluimos la etapa de preprocesamiento en el método de control de `caret` y obtenemos nuevos modelos. Primero con una simple técnica de submuestreo. El uso de *RUS* debe obtener un conjunto de entrenamiento perfectamente equilibrado eliminando instancias de la clase mayoritaria de forma aleatoria.

```
#Execute model ("preprocessed" data)
#Undersampling
ctrl <- trainControl(method="repeatedcv",number=5,repeats = 3,classProbs=TRUE,summaryFunction = twoClass

model.us <- learn_model(trainData,classifier,hyperp,ctrl,"US ")
```

```
## Setting levels: control = positive, case = negative
```

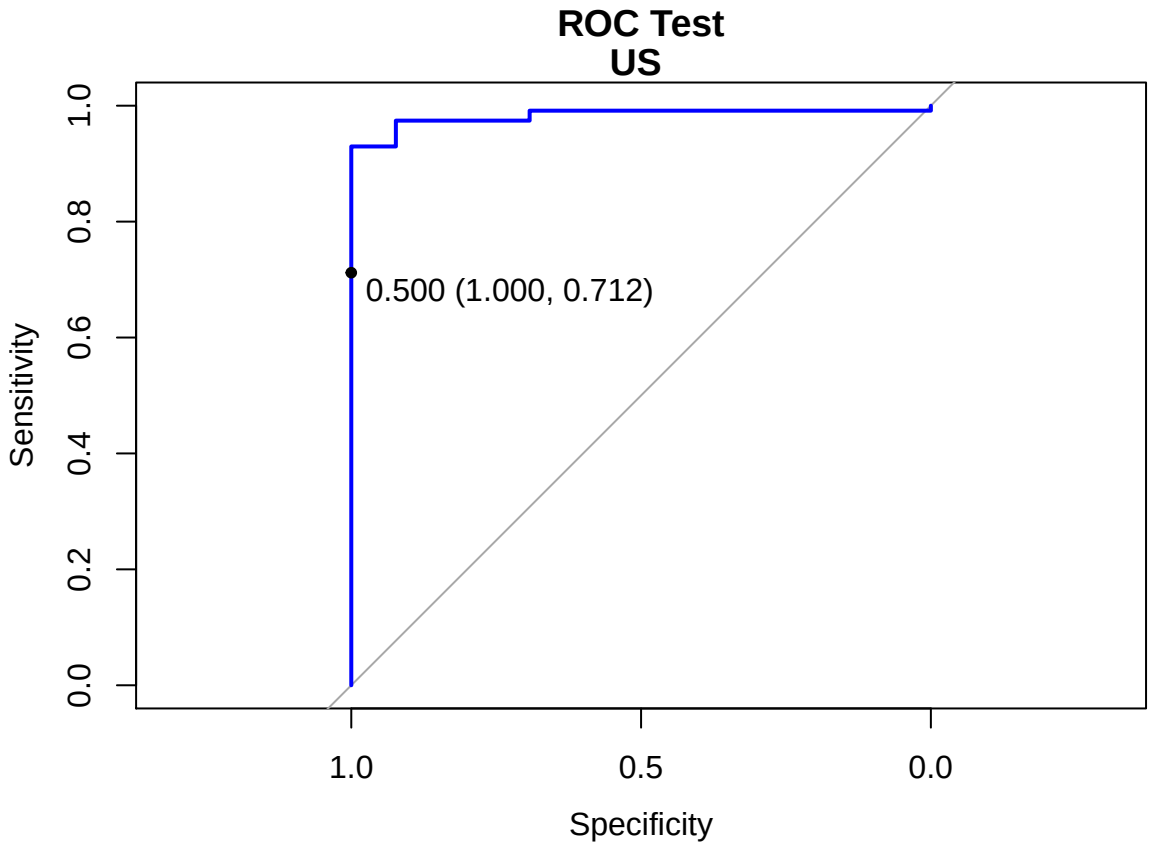
```
## Setting direction: controls > cases
```

```
cm.us <- test_model(testData,model.us,"US ")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction positive negative
##   positive       13      168
##   negative        0      415
##
##                Accuracy : 0.7181
##                  95% CI : (0.6801, 0.7539)
##     No Information Rate : 0.9782
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.0973
```

```
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 1.00000
##             Specificity : 0.71184
##          Pos Pred Value : 0.07182
##          Neg Pred Value : 1.00000
##              Prevalence : 0.02181
##          Detection Rate : 0.02181
##    Detection Prevalence : 0.30369
##       Balanced Accuracy : 0.85592
##
##        'Positive' Class : positive
##
```

```
## Setting levels: control = positive, case = negative
## Setting direction: controls > cases
```



Ahora debemos comprobar el comportamiento del enfoque de sobremuestreo aleatorio. El uso del *ROS* debe obtener un conjunto de entrenamiento perfectamente equilibrado al replicar instancias de la clase minoritaria de forma aleatoria.

```
#Oversampling
ctrl <- trainControl(method="repeatedcv",number=5,repeats = 3,
                     classProbs=TRUE,summaryFunction = twoClassSummary,sampling = "up")
model.os <- learn_model(trainData,classifier,hyperp,ctrl,"OS ")
```

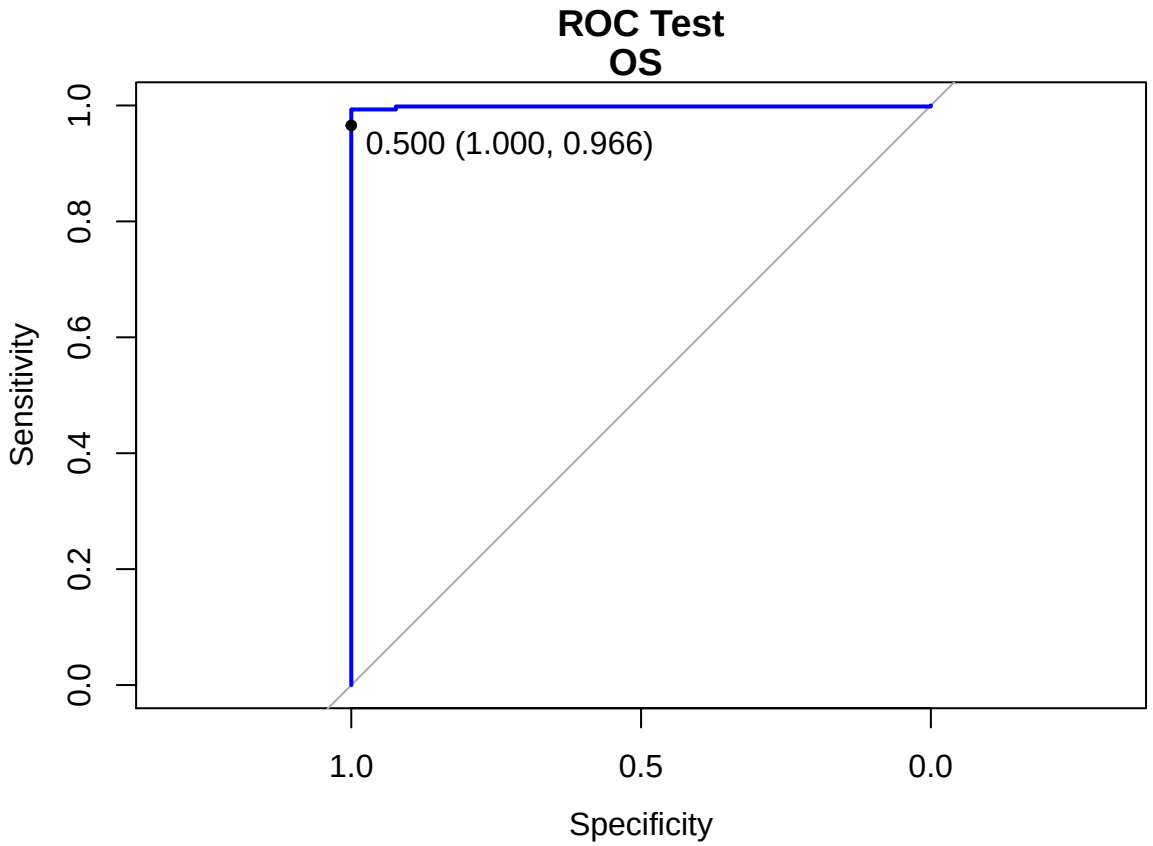
```
## Setting levels: control = positive, case = negative
```

```
## Setting direction: controls > cases
```

```
cm.os <- test_model(testData,model.os,"OS ")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction positive negative
##   positive       13       20
##   negative        0      563
##
##                Accuracy : 0.9664
##                  95% CI : (0.9486, 0.9794)
##     No Information Rate : 0.9782
##     P-Value [Acc > NIR] : 0.9763
##
##                   Kappa : 0.5512
##
##  Mcnemar's Test P-Value : 2.152e-05
##
##             Sensitivity : 1.00000
##             Specificity : 0.96569
##          Pos Pred Value : 0.39394
##          Neg Pred Value : 1.00000
##              Prevalence : 0.02181
##          Detection Rate : 0.02181
##    Detection Prevalence : 0.05537
##       Balanced Accuracy : 0.98285
##
##        'Positive' Class : positive
##
```

```
## Setting levels: control = positive, case = negative
## Setting direction: controls > cases
```



Finalmente, chequeamos la solución del estado del arte conocida como SMOTE. La aplicación de SMOTE debería obtener un conjunto de entrenamiento perfectamente equilibrado, creando nuevas instancias de la clase minoritaria

```
#SMOTE
ctrl <- trainControl(method="repeatedcv",number=5,repeats = 3, classProbs=TRUE,summaryFunction = twoCla
model.smt <- learn_model(trainData,classifier,hyperp,ctrl,"SMT ")
```

```
## Loading required package: grid
```

```
## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo
```

```
## Setting levels: control = positive, case = negative
```

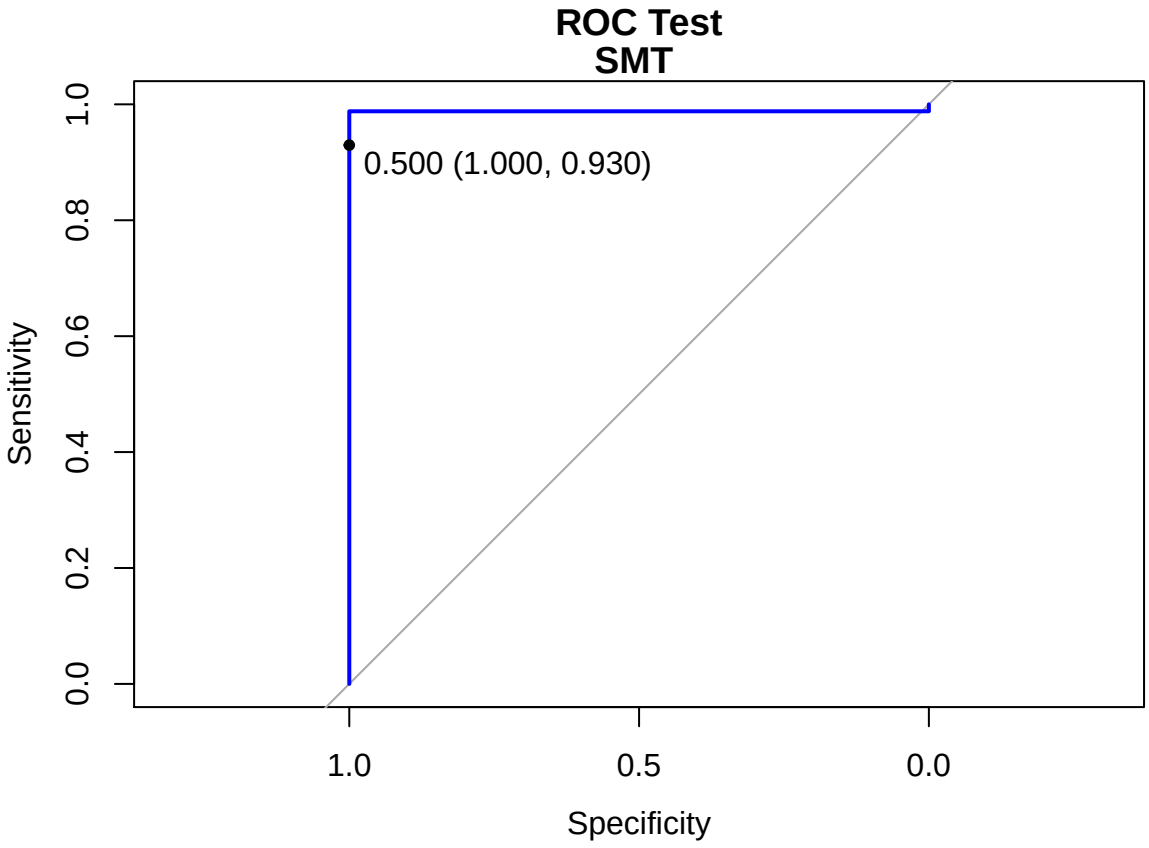
```
## Setting direction: controls > cases
```

```
cm.smt <- test_model(testData,model.smt,"SMT ")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction positive negative
##   positive       13       41
##   negative        0      542
##
##                Accuracy : 0.9312
##                  95% CI : (0.9078, 0.9502)
##     No Information Rate : 0.9782
```

```
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.3658
##
##  Mcnemar's Test P-Value : 4.185e-10
##
##             Sensitivity : 1.00000
##             Specificity : 0.92967
##          Pos Pred Value : 0.24074
##          Neg Pred Value : 1.00000
##              Prevalence : 0.02181
##          Detection Rate : 0.02181
##    Detection Prevalence : 0.09060
##       Balanced Accuracy : 0.96484
##
##        'Positive' Class : positive
##
```

```
## Setting levels: control = positive, case = negative
## Setting direction: controls > cases
```



Selecciona el mejor modelo

Ahora tenemos 4 modelos aprendidos del mismo conjunto de datos pero con diferentes opciones de preprocesamiento. Cada modelo comprende una estimación de precisión diferente, y por lo tanto necesitamos comparar los modelos entre sí y seleccionar el más "exacto" en términos de métricas de rendimiento desequilibrado, por supuesto.

Podemos informar sobre el rendimiento de cada modelo creando primero una lista de los modelos creados y utilizando la función de resumen:

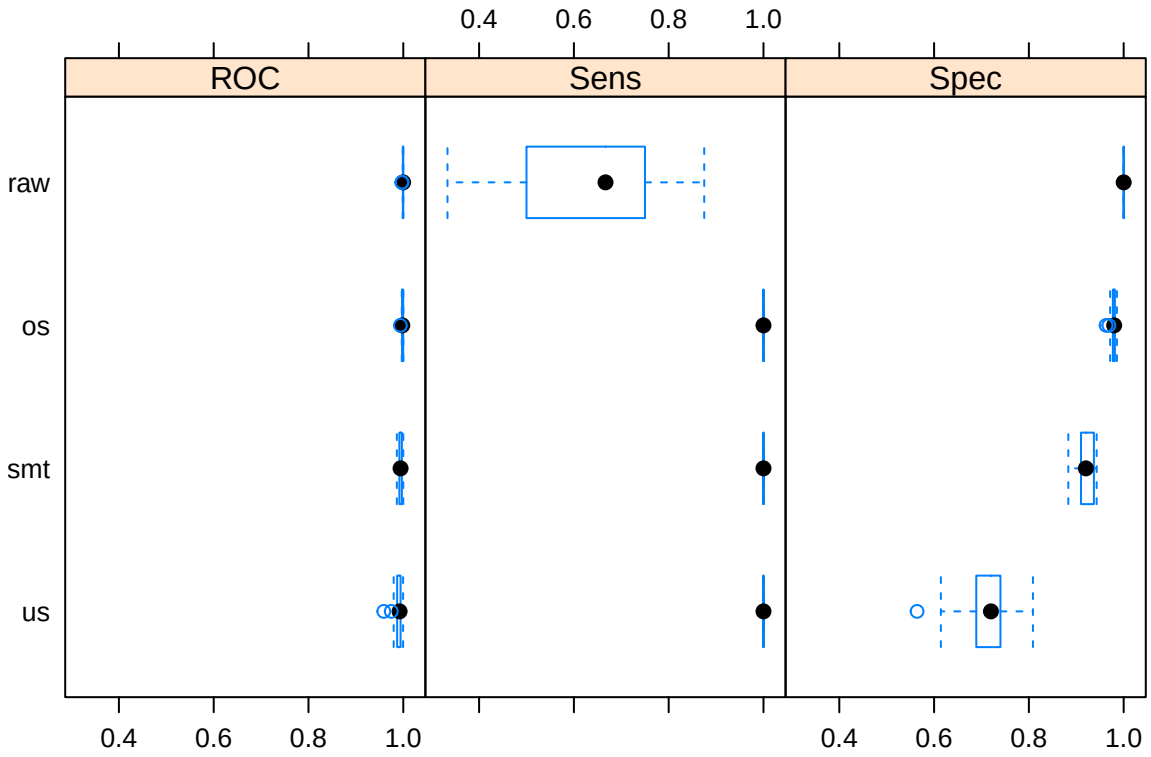
```
# resumen del acierto de los modelos

models <- list(raw = model.raw,us = model.us,os = model.os,smt = model.smt)
results <- resamples(models)
summary(results)
```

```
##
## Call:
## summary.resamples(object = results)
##
## Models: raw, us, os, smt
## Number of resamples: 15
##
## ROC
##          Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## raw 0.9968254 0.9991097 0.9996429 0.9993274 1.0000000 1.0000000    0
## us  0.9588675 0.9870818 0.9922222 0.9886787 0.9942857 0.9996429    0
## os  0.9939286 0.9975000 0.9979365 0.9982000 0.9998413 1.0000000    0
## smt 0.9864672 0.9918365 0.9942857 0.9942990 0.9970635 1.0000000    0
##
## Sens
##          Min. 1st Qu.    Median      Mean 3rd Qu.  Max. NA's
## raw 0.3333333     0.5 0.6666667 0.6212963    0.75 0.875    0
## us  1.0000000     1.0 1.0000000 1.0000000    1.00 1.000    0
## os  1.0000000     1.0 1.0000000 1.0000000    1.00 1.000    0
## smt 1.0000000     1.0 1.0000000 1.0000000    1.00 1.000    0
##
## Spec
##          Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## raw 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000    0
## us  0.5641026 0.6890110 0.7200000 0.7079669 0.7400000 0.8085714    0
## os  0.9628571 0.9771429 0.9800000 0.9779297 0.9814571 0.9857143    0
## smt 0.8831909 0.9100000 0.9202279 0.9216182 0.9371429 0.9430199    0
```

También podemos crear un gráfico de los resultados de la evaluación del modelo y comparar la dispersión y el rendimiento medio de cada modelo. En el caso de los datos de *círculos*, todos los métodos de preprocesamiento son igualmente buenos en términos de sensibilidad (reconocimiento de clase positiva), pero el sobremuestreo es un poco mejor para la especificidad (reconocimiento de clase negativa).

El procedimiento ideal es obtener una población de medidas de rendimiento para cada algoritmo, porque cada algoritmo se evalúa varias veces (validación cruzada de *k folds*).

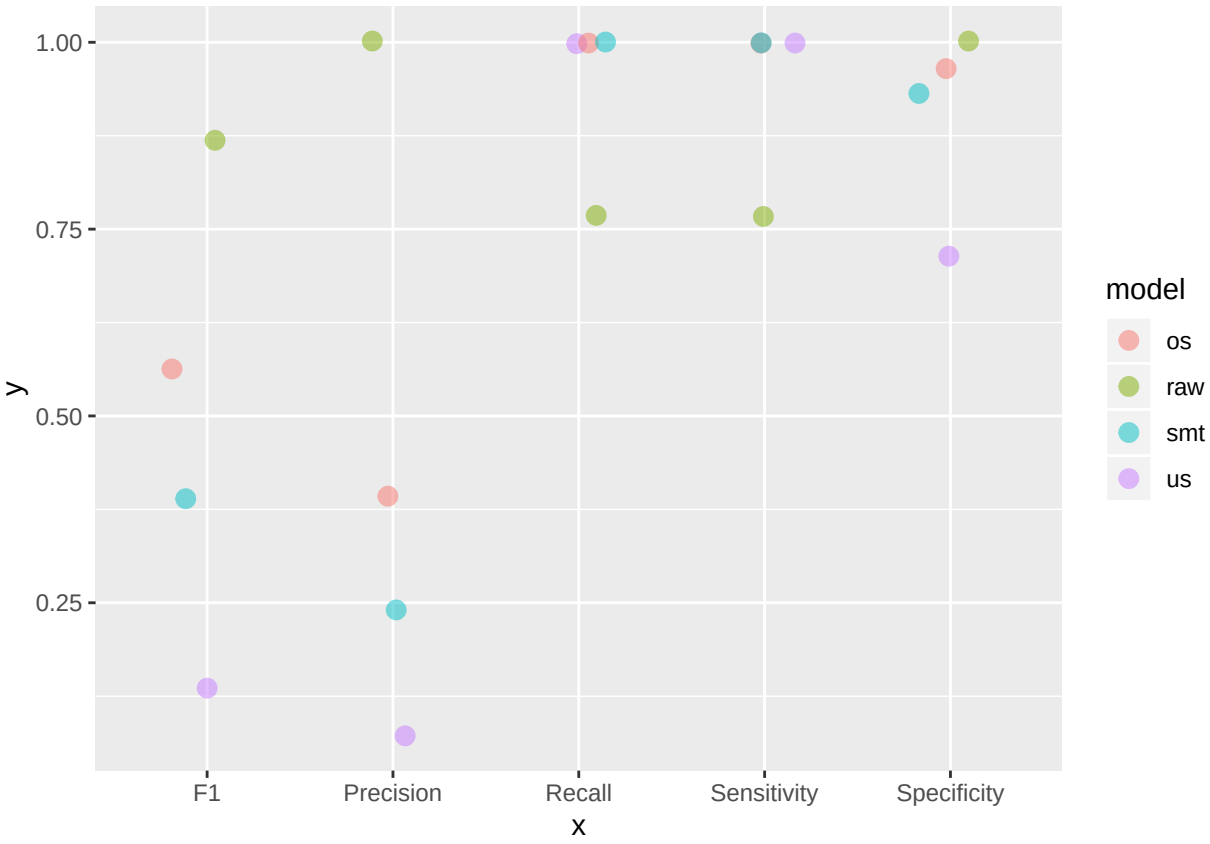
```
# comparar el acierto de los distintos modelos
bwplot(results)
```




```
#dotplot(results)
```

Finalmente podemos hacer otro gráfico diferente para comparar métricas adicionales para la clasificación no balanceada, como la precisión, el recall y la F1.

```
#LLevar a cabo una comparación sobre todas las métricas de imbalanced
comparison <- data.frame(model = names(models),
                         Sensitivity = rep(NA, length(models)),
                         Specificity = rep(NA, length(models)),
                         Precision = rep(NA, length(models)),
                         Recall = rep(NA, length(models)),
                         F1 = rep(NA, length(models)))

for (name in names(models)) {
  cm_model <- get(paste0("cm.", name))

  comparison[comparison$model == name, ] <- filter(comparison, model == name) %>%
    mutate(Sensitivity = cm_model$byClass["Sensitivity"],
           Specificity = cm_model$byClass["Specificity"],
           Precision = cm_model$byClass["Precision"],
           Recall = cm_model$byClass["Recall"],
           F1 = cm_model$byClass["F1"])
}

comparison %>%
  gather(x, y, Sensitivity:F1) %>%
  ggplot(aes(x = x, y = y, color = model)) +
  geom_jitter(width = 0.2, alpha = 0.5, size = 3)
```



Algoritmos avanzados en clasificación no balanceada: Utilizando el paquete "imbalance"

Como comentamos inicialmente, existe un paquete CRAN denominado `imbalance` que implementa algunas de las técnicas de preprocesamiento de datos más conocidas para la clasificación con clases no balanceadas. Debemos mirar más de cerca la documentación tanto en la página principal del paquete de desequilibrio, como en la función de ayuda

```
help("imbalance") #ver la documentación en la esquina inferior derecha
```

Utilizando la biblioteca `imbalance` podemos considerar la aplicación de técnicas avanzadas basadas en SMOTE. Para ello, debemos centrarnos en la función de `oversample` ("sobremuestreo"):

```
help("oversample") #ver la documentación en la esquina inferior derecha
```

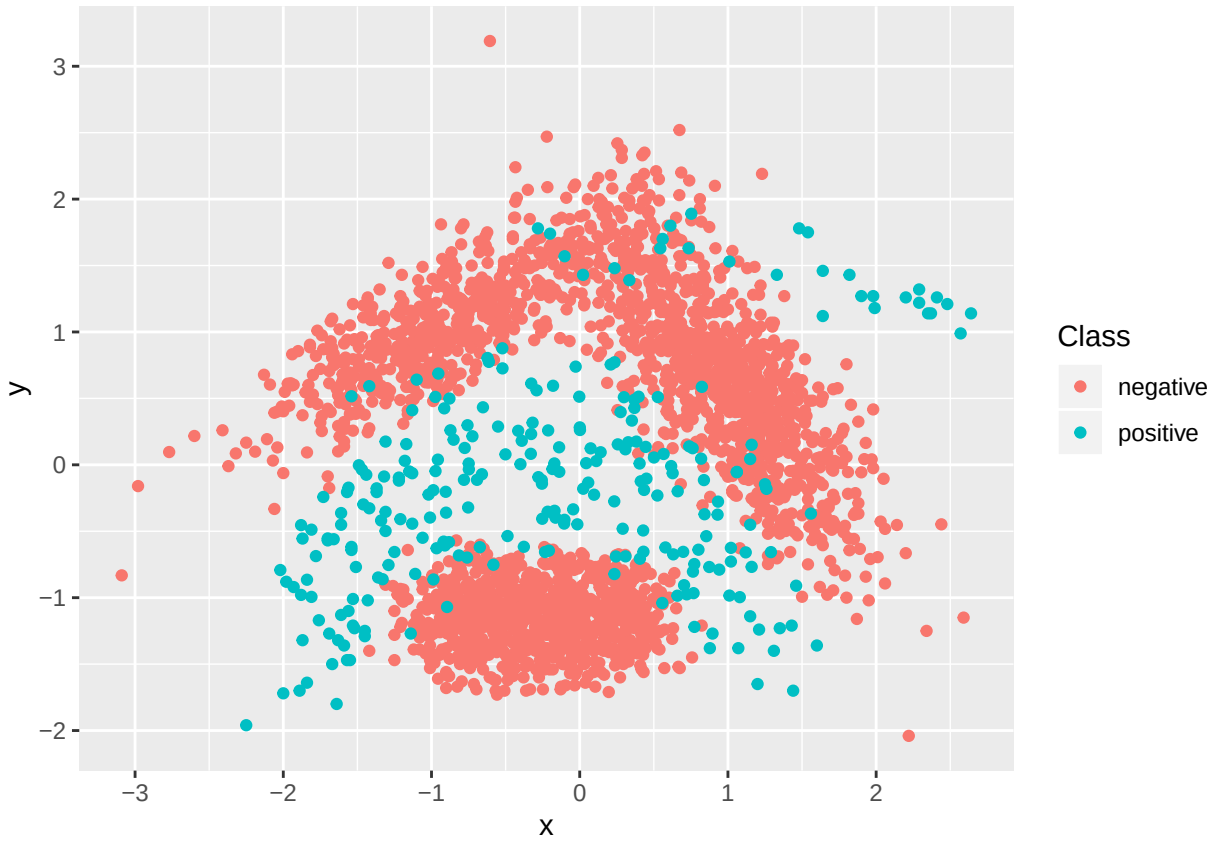
En primer lugar, se actualiza la función de comparación para hacerlo todo más compacto:

```
#Funcion para comparar la salida de las distintas técnicas de Oversampling

perform.comparison.smote <- function(dataset, imb.ratio, methods){
  ## Train index generation
  trainIndex <- createDataPartition(dataset$Class, p = .75,
  list = FALSE, times = 1)
  ## train-test separation
  trainData <- dataset[trainIndex,]
  testData <- dataset[-trainIndex,]
  ## Basic model training (no grid search)
```

```
  ctrl <- trainControl(method="none",classProbs=TRUE, summaryFunction = twoClassSummary)
  ## Change for another classifier (or include as parameter)
  classifier = "rpart"
  hyperp = data.frame(cp = 0.05)
  basic.model <- learn_model(trainData, classifier,hyperp,ctrl, "RAW")
  basic.model <- test_model(testData, basic.model, "RAW")

  ## Model training with each data generation policy
  cm.models <- sapply(methods, function (x) {
    aug.trainData <- oversample(trainData, ratio=imb.ratio, method=x, )
    ctrl <- trainControl(method="none", classProbs=TRUE, summaryFunction = twoClassSummary)
    model <- learn_model(aug.trainData, classifier,hyperp, ctrl, x)
    test_model(testData, model, x)
    }, simplify = F)

  cm.models <- list.prepend(cm.models, RAW=basic.model)

  ## Metrics gathering
  comparison <- lapply(cm.models, function(x){
      x$byClass[c("Balanced Accuracy", "F1", "Precision","Recall", "Specificity")]
  })

  ## Transformation into dataframe
  comparison <- as.data.frame(do.call(rbind, comparison))
  comparison$model <- rownames(comparison)
  comparison
}
```

A continuación, cargamos alguno de los conjuntos de datos proporcionados por el paquete (banana, o glass0, por ejemplo). Seleccionamos varias técnicas o variantes SMOTE diferentes, para aplicarlas sobre los problemas anteriores.

```
# Load the data
dataset <- banana
dataset <- unique(dataset)
dataset[,-length(dataset)] <- sapply(dataset[,-length(dataset)], as.numeric)
repr.data <- dataset
colnames(repr.data) <- c("x", "y", "Class")
ggplot(repr.data) + geom_point(aes(x=x, y=y, color=Class))
```




```r
# Apply preprocessing with oversample function
imb.ratio <- 0.65
methods <- c("SMOTE", "BLSMOTE", "DBSMOTE", "MWMOTE")
comparison <- perform.comparison.smote(dataset, imb.ratio, methods)
```

```
## Setting levels: control = negative, case = positive

## Setting direction: controls < cases

## Confusion Matrix and Statistics
##
##           Reference
## Prediction negative positive
##   negative      580       32
##   positive       12       34
##
##                Accuracy : 0.9331
##                  95% CI : (0.9113, 0.951)
##     No Information Rate : 0.8997
##     P-Value [Acc > NIR] : 0.001699
##
##                   Kappa : 0.5719
##
##  Mcnemar's Test P-Value : 0.004179
##
##             Sensitivity : 0.51515
##             Specificity : 0.97973
##          Pos Pred Value : 0.73913
```

```
##          Neg Pred Value : 0.94771
##              Prevalence : 0.10030
##          Detection Rate : 0.05167
##    Detection Prevalence : 0.06991
##       Balanced Accuracy : 0.74744
##
##        'Positive' Class : positive
##
```

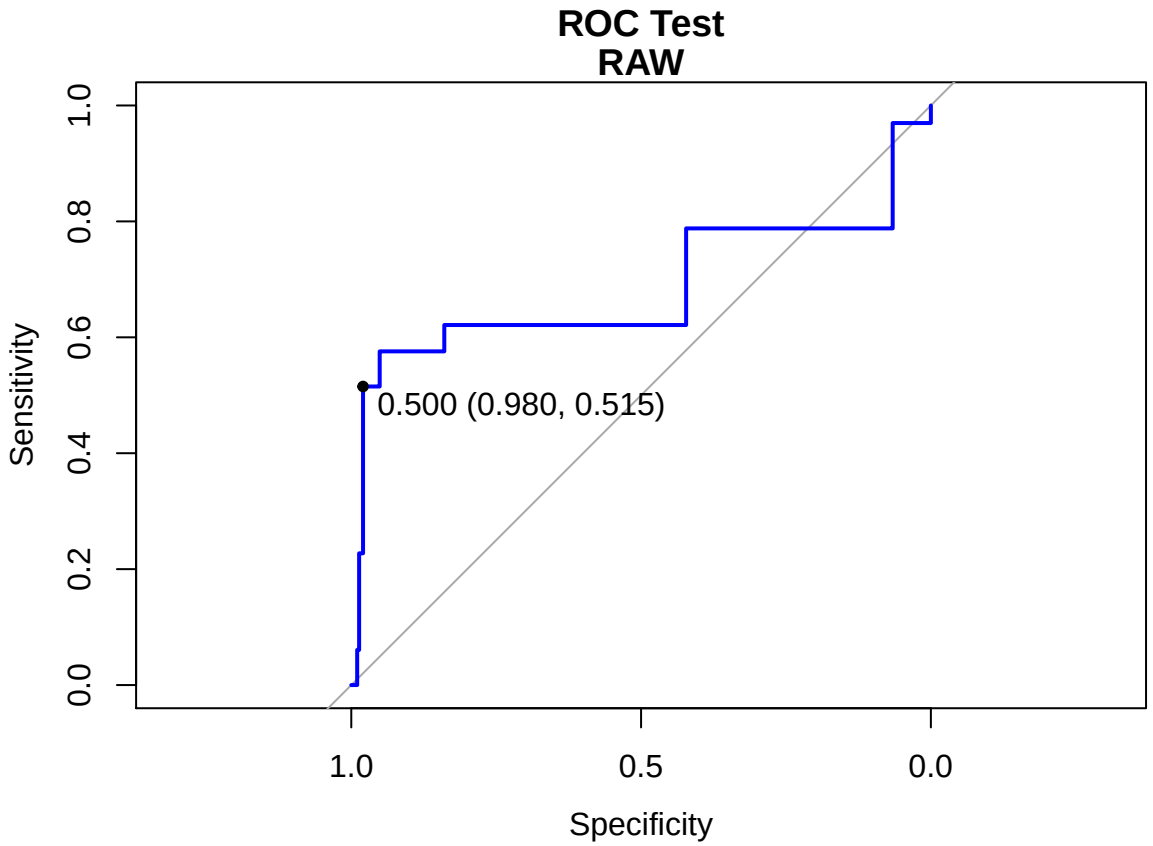
```
## Setting levels: control = negative, case = positive
## Setting direction: controls < cases
```

```
## Setting levels: control = negative, case = positive
```

```
## Setting direction: controls < cases
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction negative positive
##   negative      555       19
##   positive       37       47
##
##                Accuracy : 0.9149
##                  95% CI : (0.8909, 0.9351)
##     No Information Rate : 0.8997
##     P-Value [Acc > NIR] : 0.1069
##
##                   Kappa : 0.5794
##
##  Mcnemar's Test P-Value : 0.0231
##
##             Sensitivity : 0.71212
##             Specificity : 0.93750
##          Pos Pred Value : 0.55952
##          Neg Pred Value : 0.96690
##              Prevalence : 0.10030
##          Detection Rate : 0.07143
##    Detection Prevalence : 0.12766
##       Balanced Accuracy : 0.82481
##
##        'Positive' Class : positive
##
```

```
## Setting levels: control = negative, case = positive
## Setting direction: controls < cases
```



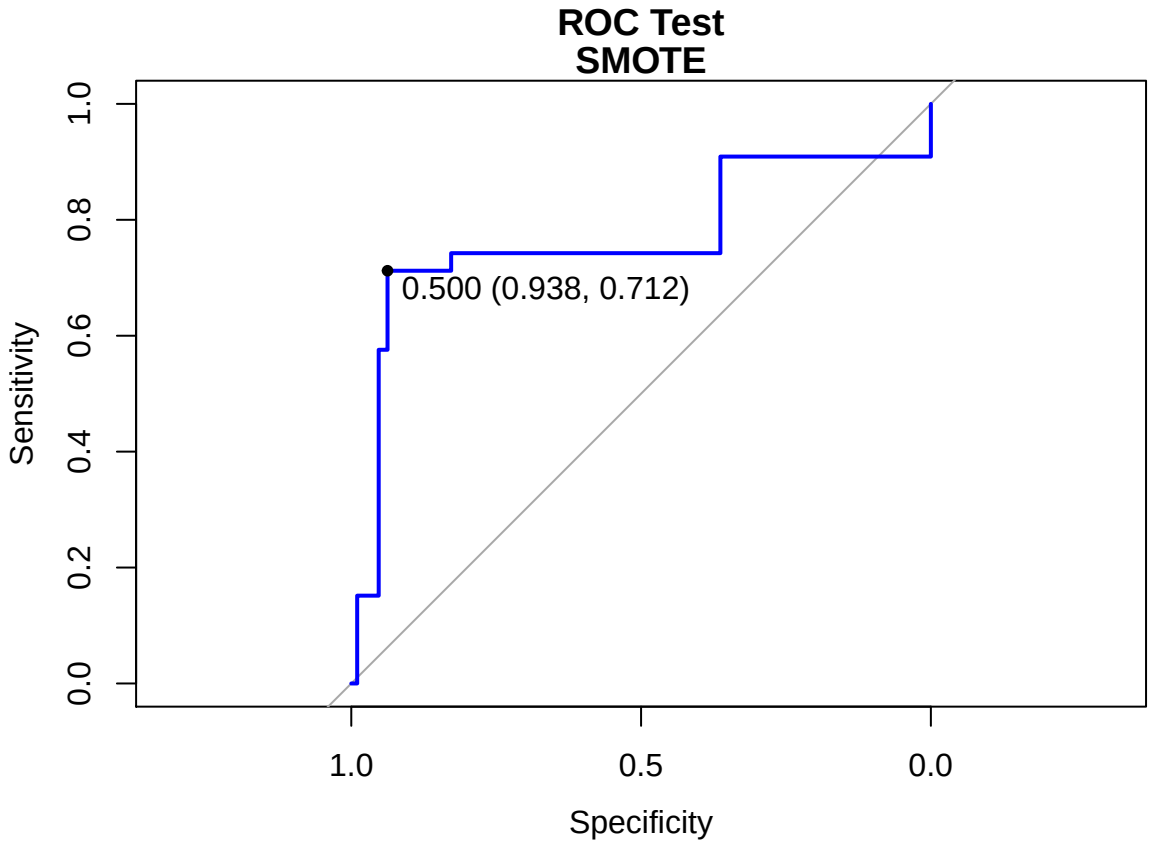

```
## [1] "Borderline-SMOTE done"
```

```
## Setting levels: control = negative, case = positive
## Setting direction: controls < cases
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction negative positive
##   negative      535       29
##   positive       57       37
##
##                Accuracy : 0.8693
##                  95% CI : (0.8411, 0.8941)
##     No Information Rate : 0.8997
##     P-Value [Acc > NIR] : 0.994891
##
##                   Kappa : 0.3907
##
##  Mcnemar's Test P-Value : 0.003597
##
##             Sensitivity : 0.56061
##             Specificity : 0.90372
##          Pos Pred Value : 0.39362
##          Neg Pred Value : 0.94858
##              Prevalence : 0.10030
##          Detection Rate : 0.05623
##    Detection Prevalence : 0.14286
```

```
##       Balanced Accuracy : 0.73216
##
##        'Positive' Class : positive
##
```

```
## Setting levels: control = negative, case = positive
## Setting direction: controls < cases
```



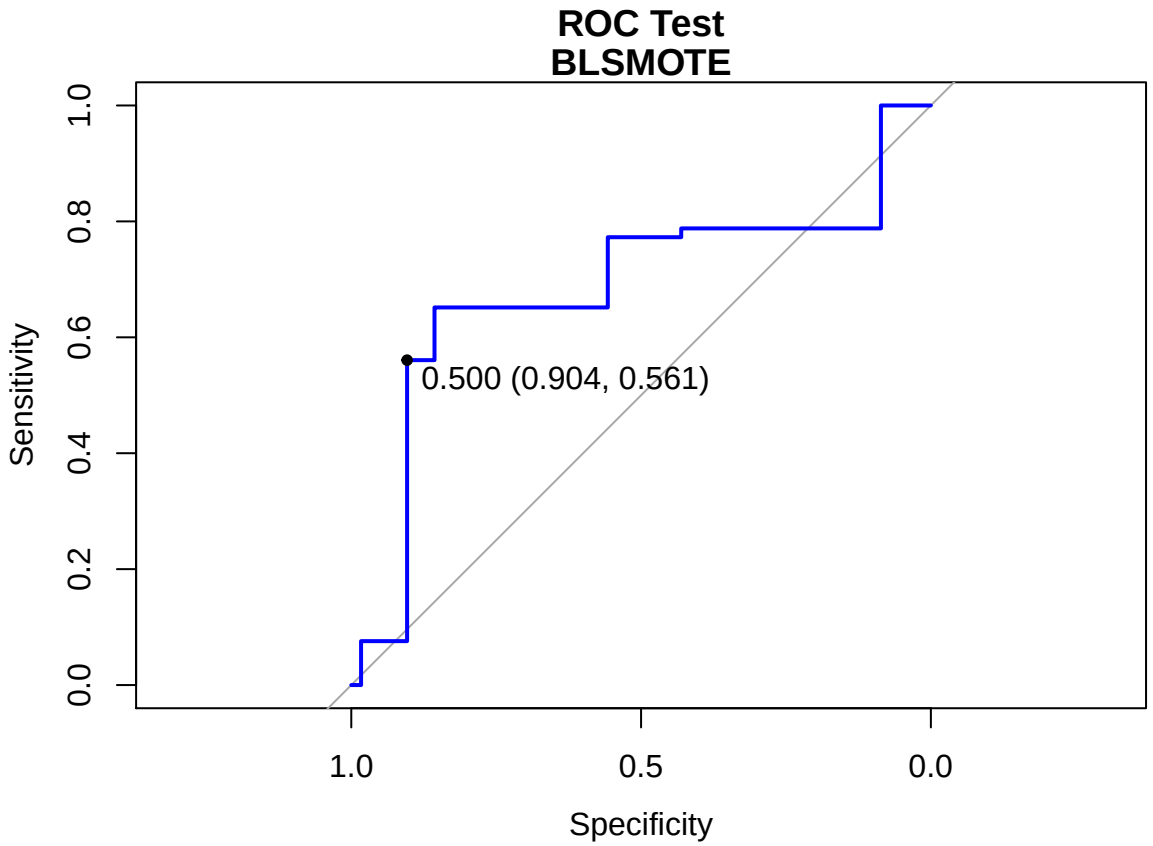

```
## [1] 8
## [1] 9
## [1] 8
## [1] 13
## [1] 8
## [1] 9
## [1] 5
## [1] 6
## [1] 7
## [1] 2
## [1] 5
## [1] 9
## [1] 9
## [1] 9
## [1] 11
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

```
## [1] "DBSMOTE is Done"
```

```
## Setting levels: control = negative, case = positive
## Setting direction: controls < cases
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction negative positive
##   negative      570       38
##   positive       22       28
##
##                Accuracy : 0.9088
##                  95% CI : (0.8842, 0.9297)
##     No Information Rate : 0.8997
##     P-Value [Acc > NIR] : 0.24033
##
##                   Kappa : 0.4338
##
##  Mcnemar's Test P-Value : 0.05281
##
##             Sensitivity : 0.42424
##             Specificity : 0.96284
##          Pos Pred Value : 0.56000
##          Neg Pred Value : 0.93750
##              Prevalence : 0.10030
##          Detection Rate : 0.04255
##    Detection Prevalence : 0.07599
##       Balanced Accuracy : 0.69354
##
##        'Positive' Class : positive
##
```

```
## Setting levels: control = negative, case = positive
## Setting direction: controls < cases
```

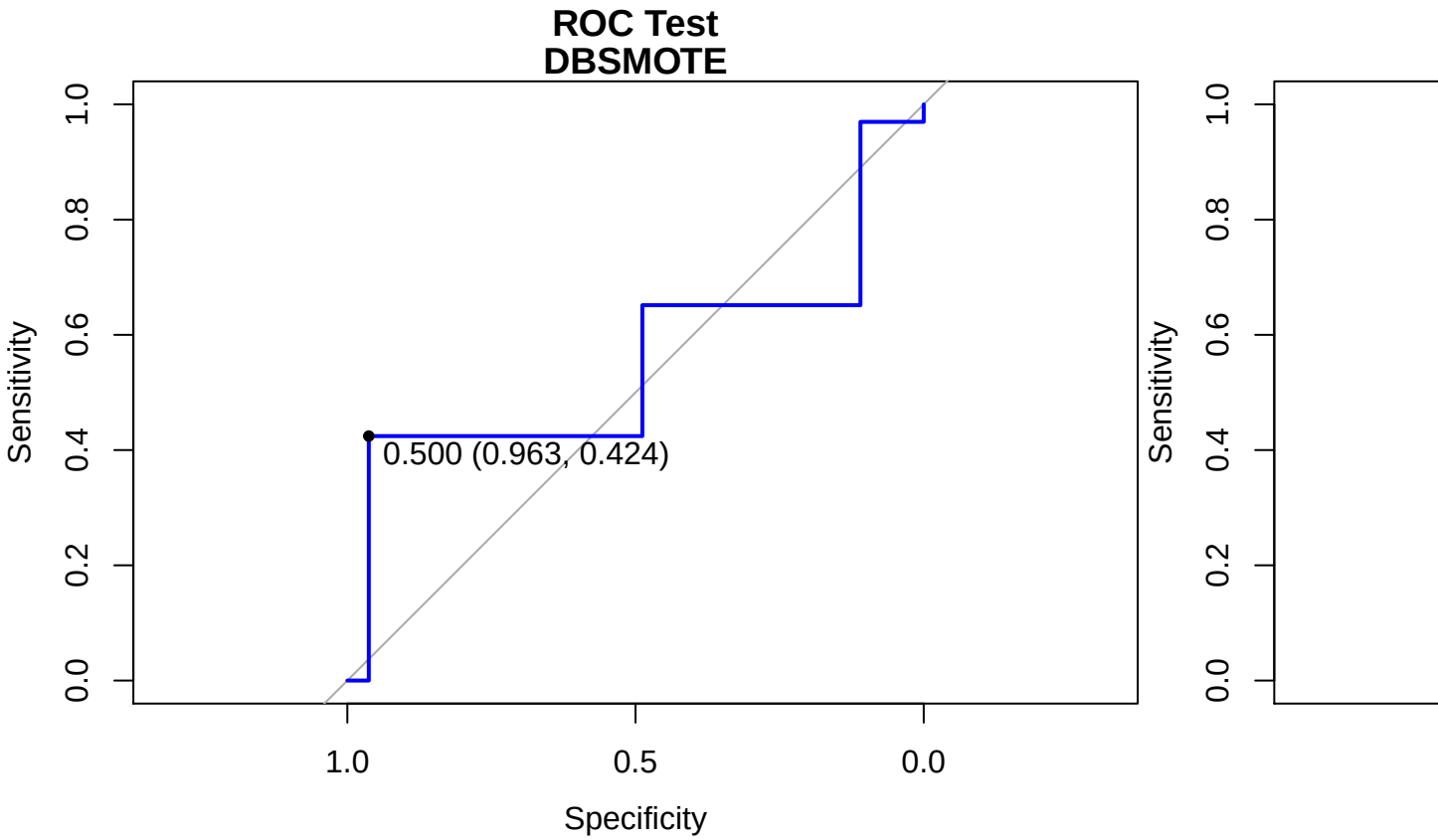



```
## Setting levels: control = negative, case = positive
## Setting direction: controls < cases
```

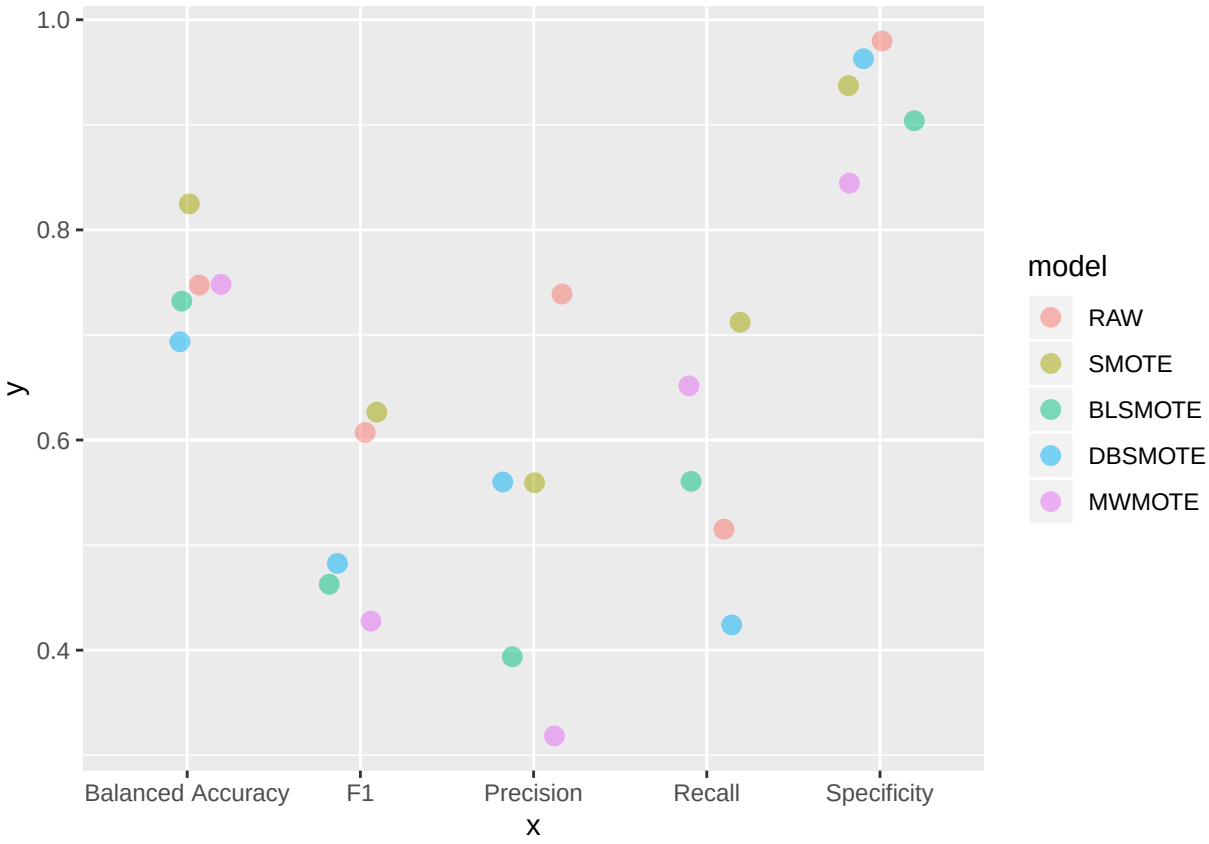
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction negative positive
##   negative      500       23
##   positive       92       43
##
##                Accuracy : 0.8252
##                  95% CI : (0.794, 0.8535)
##     No Information Rate : 0.8997
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.3388
##
##  Mcnemar's Test P-Value : 2.282e-10
##
##             Sensitivity : 0.65152
##             Specificity : 0.84459
##          Pos Pred Value : 0.31852
##          Neg Pred Value : 0.95602
##              Prevalence : 0.10030
##          Detection Rate : 0.06535
##    Detection Prevalence : 0.20517
##       Balanced Accuracy : 0.74805
```

```
##
##        'Positive' Class : positive
##
```

```
## Setting levels: control = negative, case = positive
## Setting direction: controls < cases
```




```
# Check results with kNN, DT or any other classifier (included in previous chunk)
comparison$model <- factor(comparison$model, levels=comparison$model)
comparison %>%
gather(x, y, "Balanced Accuracy":"Specificity") %>%
ggplot(aes(x = x, y = y, color = model)) +
geom_jitter(width = 0.2, alpha = 0.5, size = 3)
```



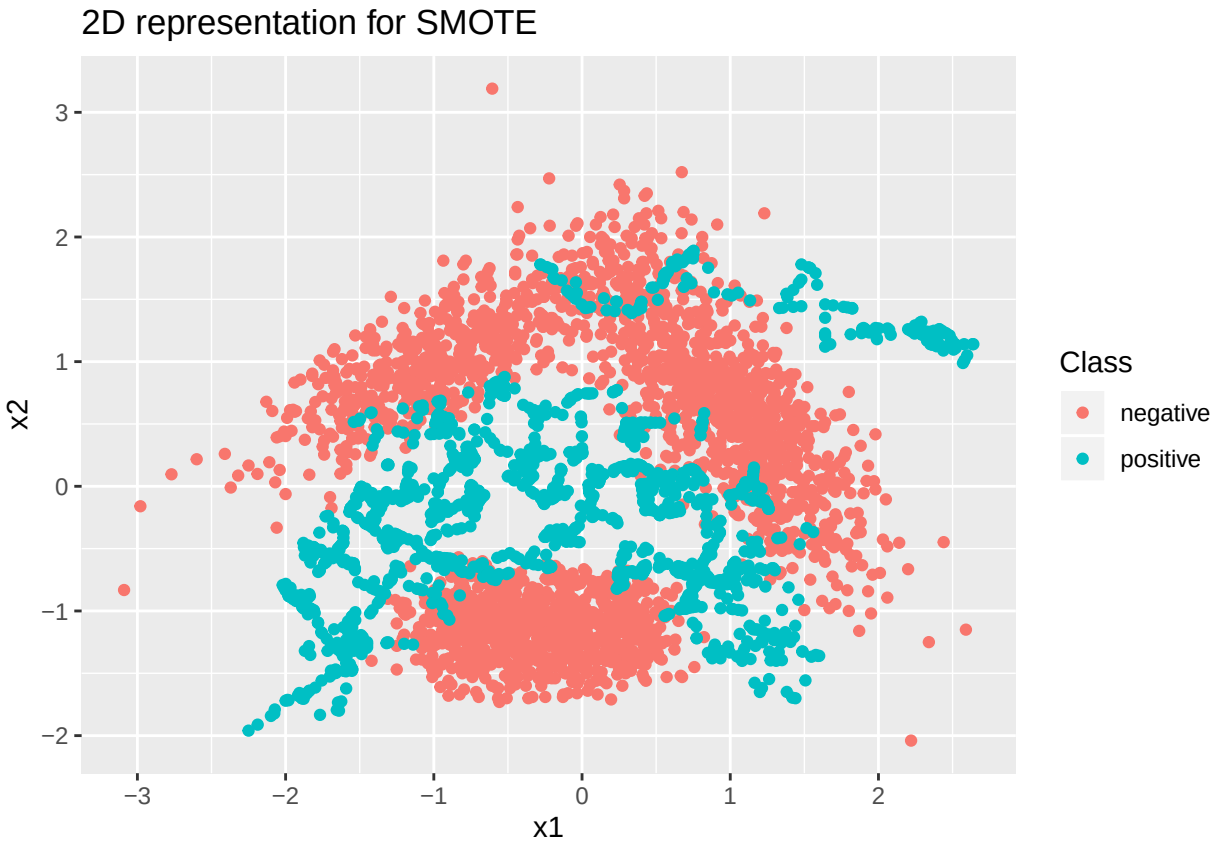
Ahora, haga una comparación gráfica entre el conjunto de datos original y el preprocesado (sólo para SMOTE, por ejemplo). Recordemos que, siendo un gráfico 2D, sólo debe seleccionar dos de las columnas de entrada.

```
# Visualize the data distribution between original and preprocess data.

representation <- function (ov_technique, data) {
    aug.data <- oversample(data, ratio=imb.ratio, method=ov_technique, )
    repr.data <- aug.data
    colnames(repr.data) <- c("x1", "x2", "Class")
    p <- ggplot(repr.data) + geom_point(aes(x=x1, y=x2, color=Class))
    p + ggtitle(paste0("2D representation for ",ov_technique)) # for the main title)
    }

sapply(methods, representation, data=dataset, simplify = F)
```

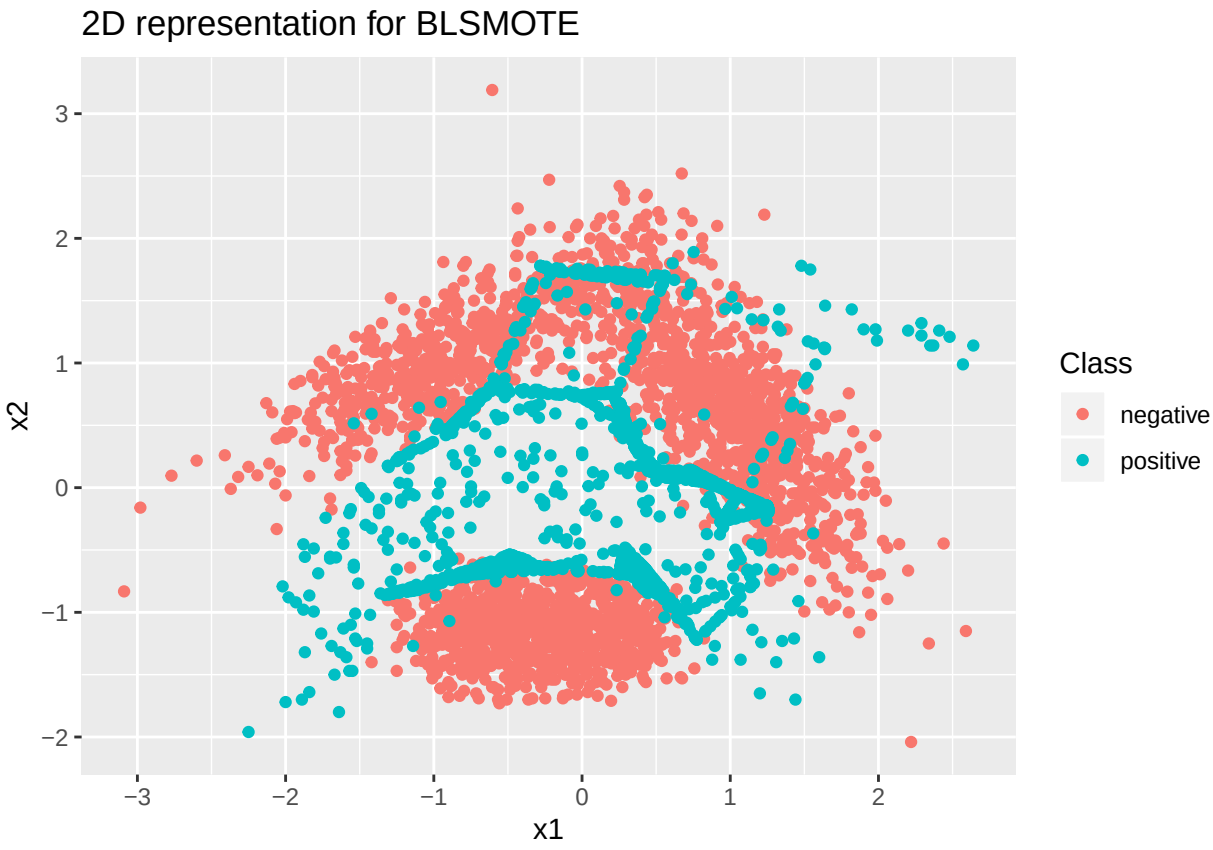
```
## [1] "Borderline-SMOTE done"
## [1] 6
## [1] 6
## [1] 5
## [1] 5
## [1] 2
## [1] 4
## [1] 8
## [1] 11
## [1] 6
## [1] 6
## [1] 8
## [1] 8
```

```
## [1] 4
## [1] 2
## [1] 3
## [1] 2
## [1] 4
## [1] 2
## [1] 2
## [1] 2
## [1] 3
## [1] 2
## [1] 3
## [1] 3
## [1] 2
## [1] "DBSMOTE is Done"
```

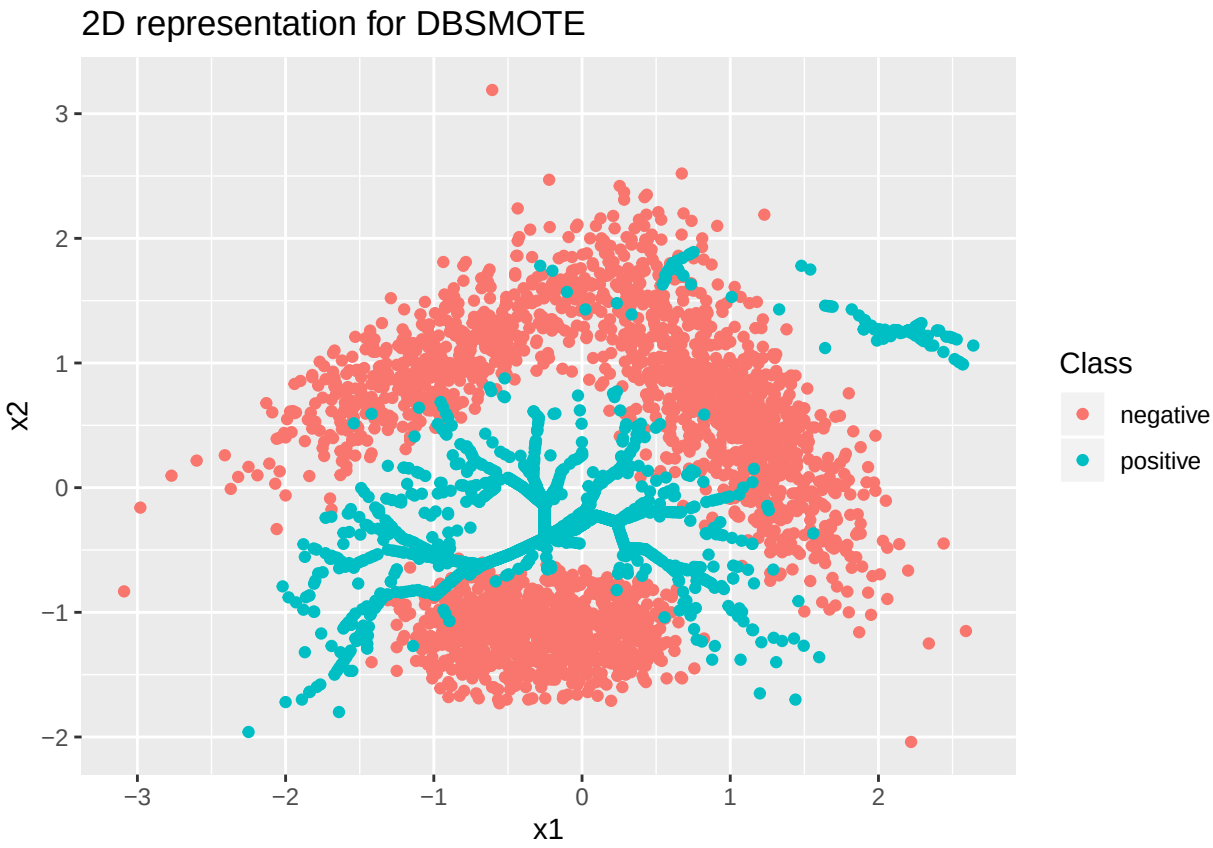
```
## $SMOTE
```



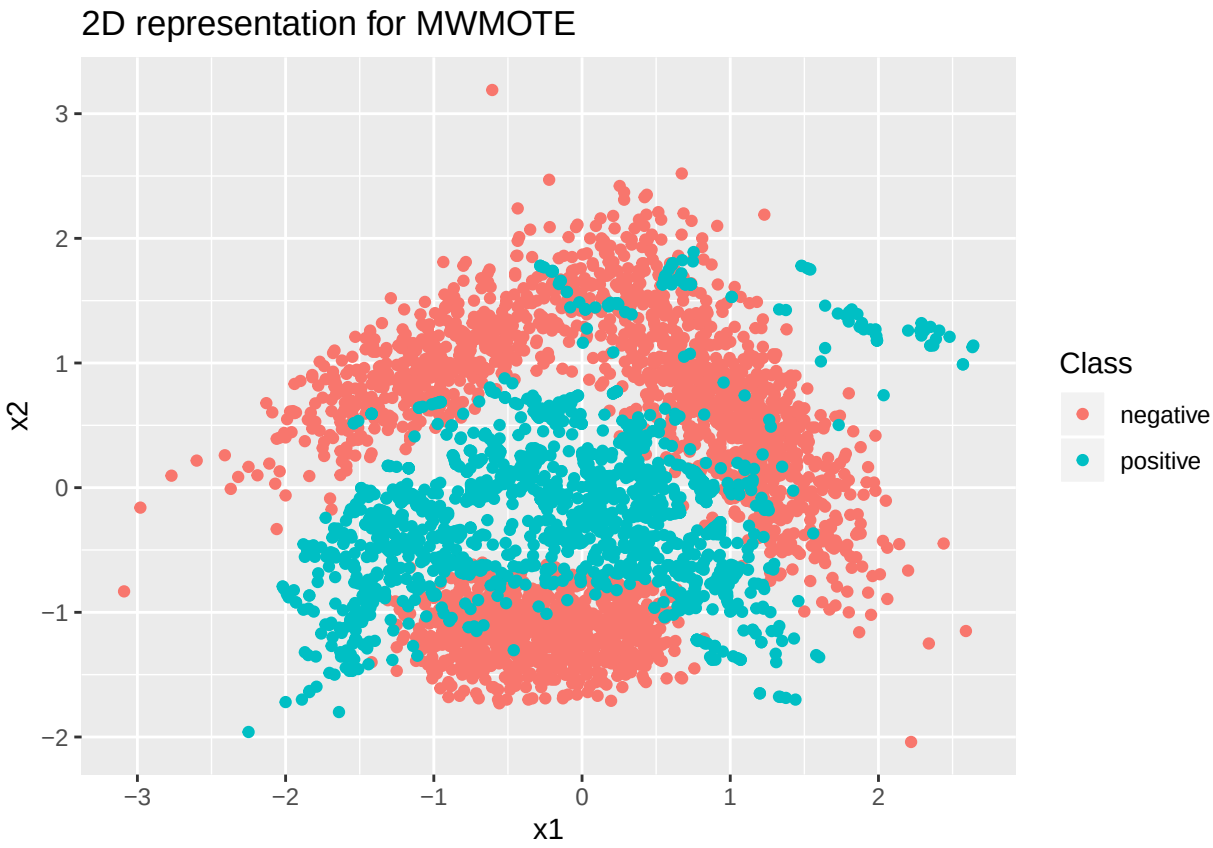

```
##
## $BLSMOTE
```


2D representation for BLSMOTE
x2
x1
3
2
1
0
−1
−2
−3
−2
−1
0
1
2
Class
negative
positive

```
##
## $DBSMOTE
```

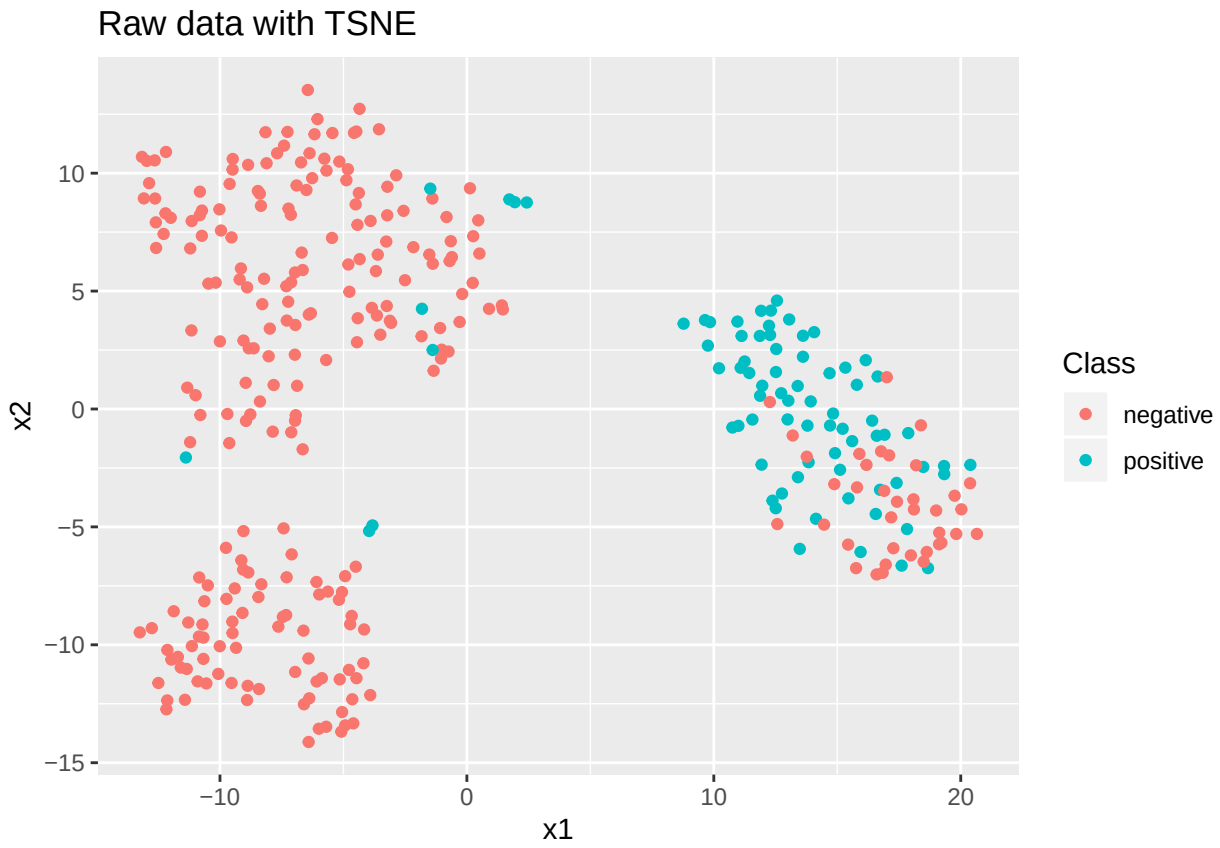

2D representation for DBSMOTE
x2
x1
3
2
1
0
−1
−2
−3
−2
−1
0
1
2
Class
negative
positive

```
##
## $MWMOTE
```



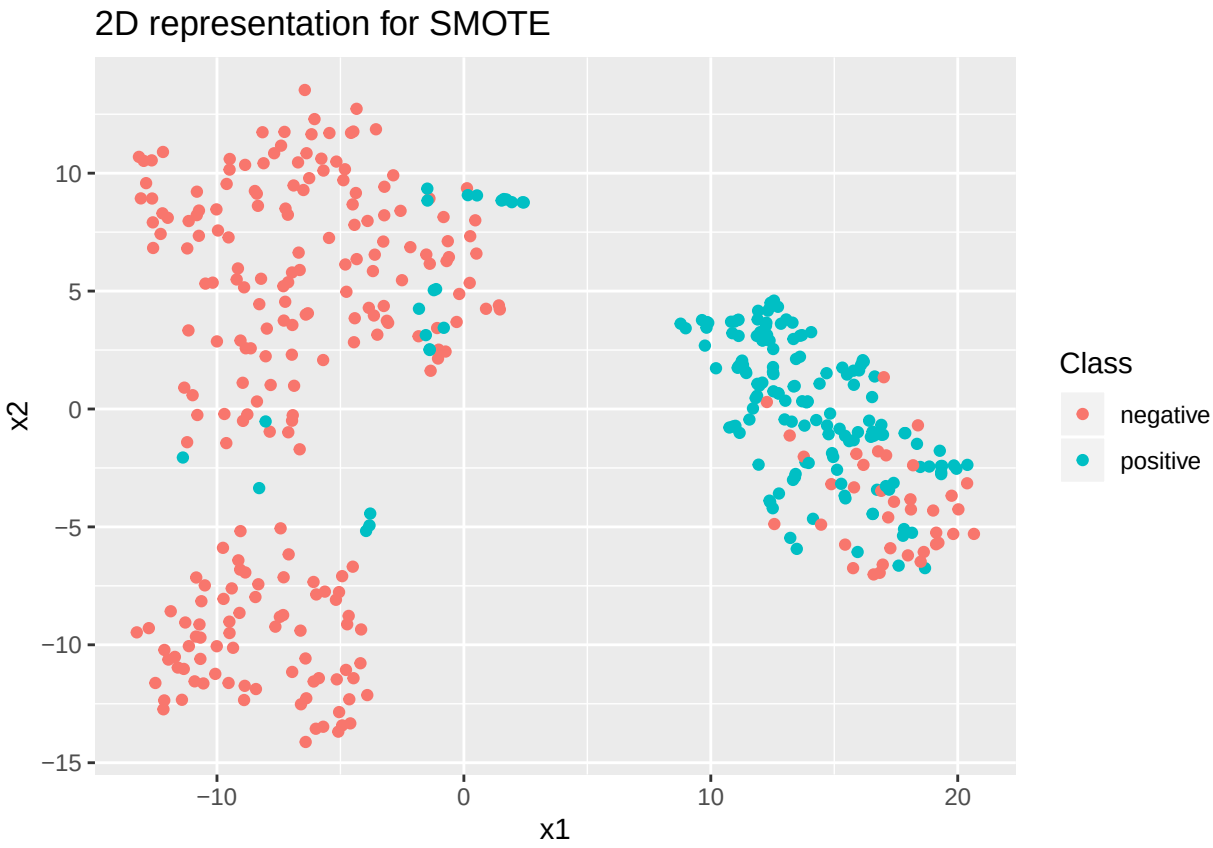
Alternativamente, puede aplicar el método `tsne` antes del gráfico para extraer dos únicas características.

```
library(Rtsne)
dataset <- ecoli1
#Dos variables que con un valor único son eliminadas
dataset <- subset(dataset, select=-c(Lip, Chg))
#Chequear posibles muestras respetidas
dataset <- unique(dataset)
#Por si hubiese algún valor numérico erróneamente cargado
dataset[,-length(dataset)] <- sapply(dataset[,-length(dataset)], as.numeric)

#No debería haber problemas puesto que hemos utilizado "unique"
tsne.suitable.ind <- !duplicated(dataset[,-length(dataset)])
data.2d <- as.data.frame(Rtsne(dataset[tsne.suitable.ind,-length(dataset)])$Y)

colnames(data.2d) <- c("x1", "x2")
#La clase no estaba inicialmente incluida
data.2d$Class <- dataset$Class[tsne.suitable.ind]

ggplot(data.2d) + geom_point(aes(x=x1, y=x2, color=Class)) + labs(title="Raw data with TSNE")
```

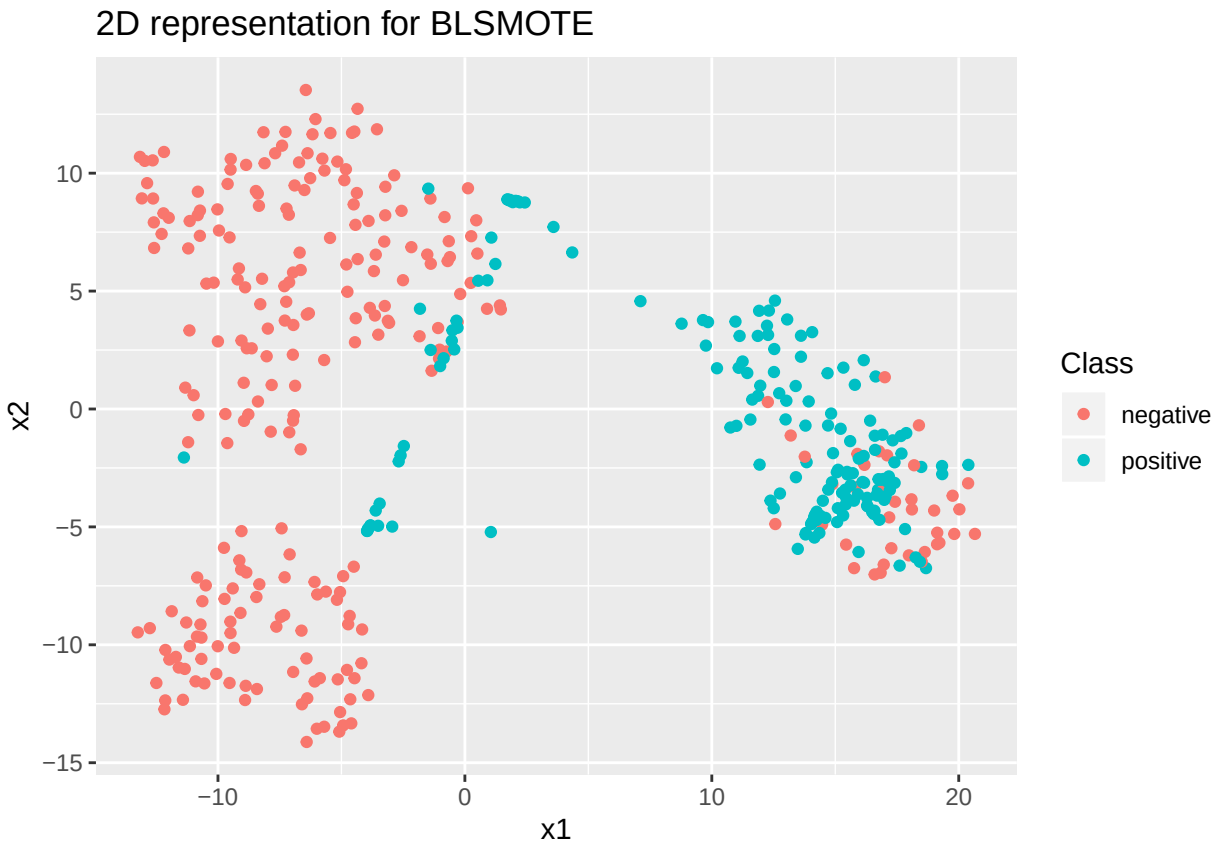



```
sapply(methods, representation, data=data.2d, simplify = F)
```

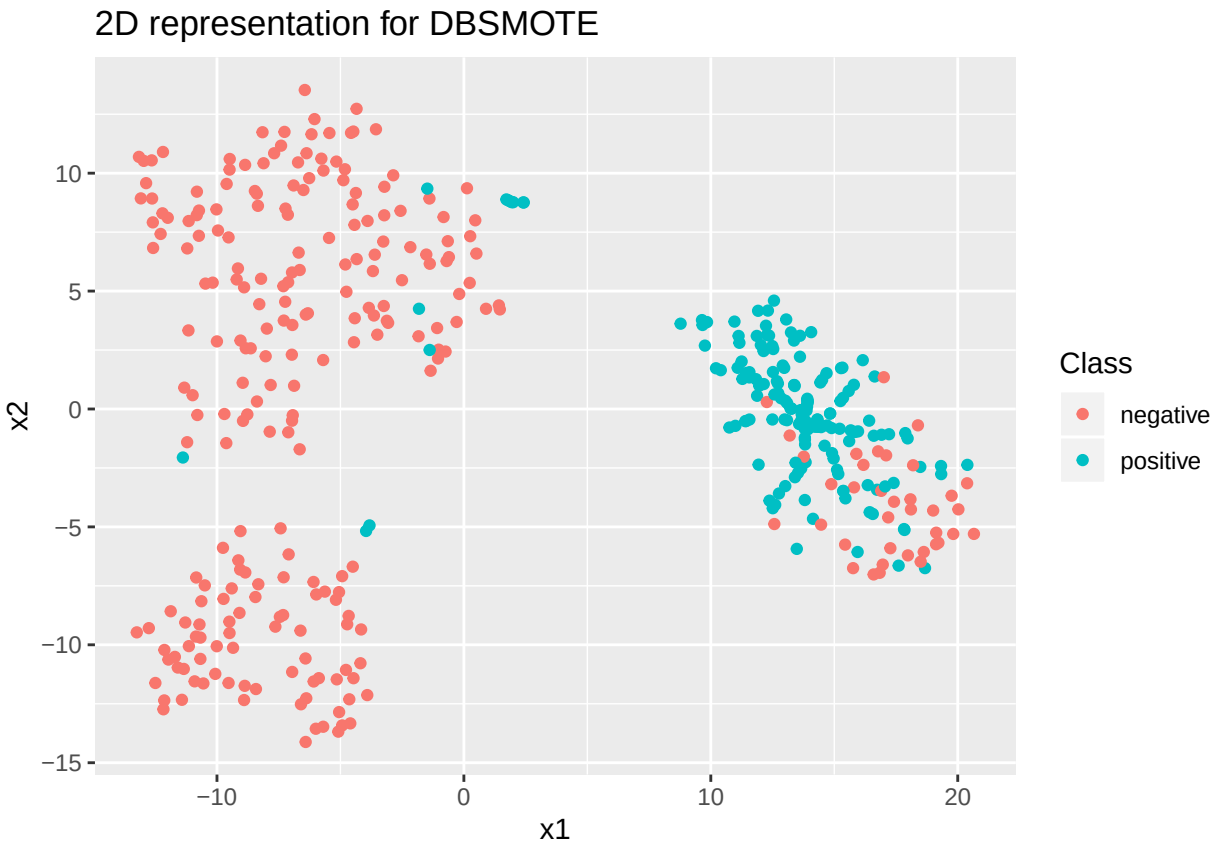
```
## [1] "Borderline-SMOTE done"
## [1] 6
## [1] 2
## [1] 4
## [1] 2
## [1] 4
## [1] 4
## [1] 5
## [1] 2
## [1] 6
## [1] 4
## [1] 4
## [1] 3
## [1] 4
## [1] 4
## [1] 4
## [1] 3
## [1] 4
## [1] 2
## [1] 2
## [1] 4
## [1] 3
## [1] 4
## [1] 3
## [1] 2
```


2D representation for SMOTE
x2
x1
10
5
0
−5
−10
−15
−10
0
10
20
Class
negative
positive

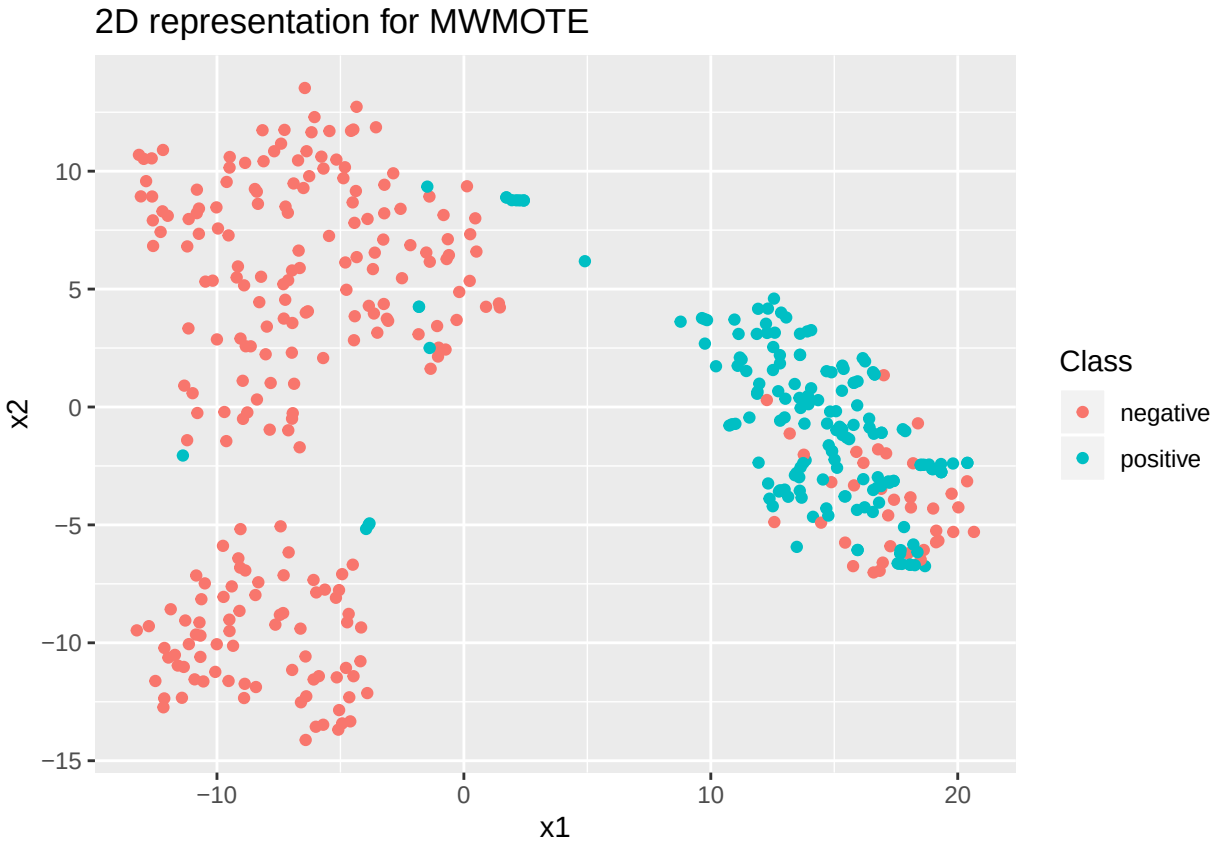
```
##
## $BLSMOTE
```


2D representation for BLSMOTE
10
5
0
−5
−10
−15
x2
−10
0
10
20
x1
Class
negative
positive

```
##
## $DBSMOTE
```

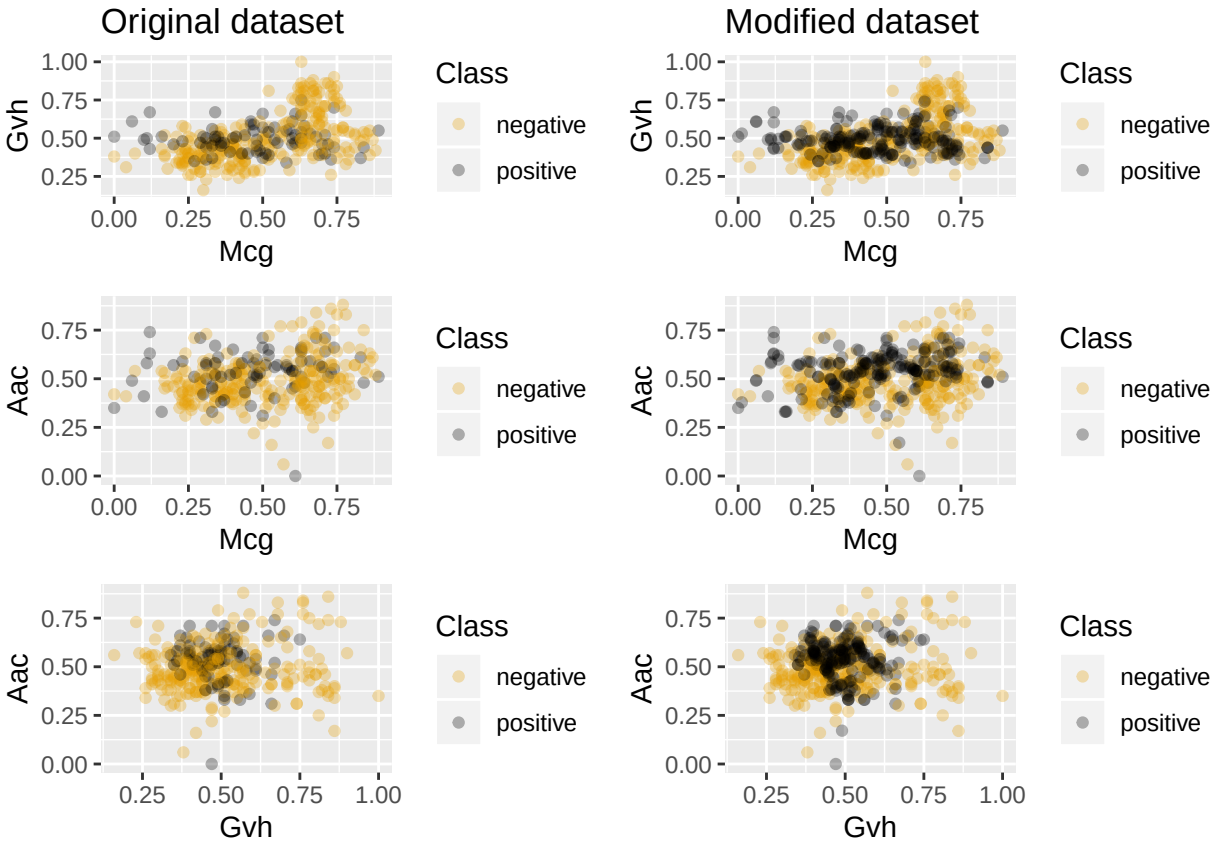

2D representation for DBSMOTE
10
5
0
−5
−10
−15
x2
−10
0
10
20
x1
Class
negative
positive

```
##
## $MWMOTE
```



Por último, es posible utilizar la propia función `plotComparison` incluida en el paquete para observar las diferentes entre el conjunto de datos original y el preprocesado, siempre realizando una representación en dos dimensiones realizando todas las combinaciones de atributos del conjunto de datos.

```
#dataset corresponde con "ecoli1"
aug.data <- oversample(dataset, ratio=0.75, method="SMOTE")
plotComparison(dataset, aug.data, attrs = names(dataset)[1:3], cols = 2, classAttr = "Class")
```



Comentarios finales

Espero que hayan disfrutado este tutorial sobre la Clasificación No Balanceada en Aprendizaje Automático. Si necesita más detalles sobre cómo realizar cualquier tipo de tarea, por favor pregúnteme por correo electrónico en la siguiente dirección: alberto@decsai.ugr.es