Tutorial on Imbalanced Classification Tasks

Alberto Fernandez (alberto@decsai.ugr.es)

January 27, 2020

This is an R Notebook a practical session on "Data Mining: Advanced Aspects" course from the "Master in Data Science and Computer Engineering" at University of Granada.

When you execute code within the notebook, the results appear beneath the code.

#Introduction Any dataset with an unequal class distribution is technically imbalanced. However, a dataset is said to be imbalanced when there is a significant, or in some cases extreme, disproportion among the number of examples of each class of the problem. In other words, the class imbalance occurs when the number of examples representing one class is much lower than the ones of the other classes. Hence, one or more classes may be underrepresented in the dataset. Such a simple definition has brought along a lot of attention from researchers and practitioners due to the number of real-world applications where the raw data gathered fulfill this definition.

In this step-by-step tutorial you will:

1. Get the most useful packages for addressing imbalanced classification with machine learning in R.
2. Load a dataset and understand it's structure using statistical summaries and data visualization.
3. Create several machine learning models in synergy with pre-processing techniques for imbalanced data, pick the best and build confidence that the predictive performance is reliable.

For the sake of easing this "hands-on" session, the document contains chunks of R source code that can be run directly. Try executing this first chunk by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing *Cmd+Shift+Enter* (*Ctrl+Shift+Enter* in Windows).

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

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

If you like to a new chunk, just click the *Insert Chunk* button on the toolbar or by pressing *Cmd+Option+I* (or *Ctrl+Alt+I* in Windows). This way, you may add your own code if requested.

When you save the notebook, an HTML file containing the code and output will be saved alongside it (click the *Preview* button or press *Cmd+Shift+K* to preview the HTML file). This is actually a nice way to compile all the tasks developed during the tutorial.

Here is an overview what we are going to cover in this *Tutorial on Imbalanced Classification*:

1. Installing the R platform.
2. Loading the dataset.
3. Summarizing the dataset.
4. Visualizing the dataset.
5. Evaluating some algorithms.
6. Comparison among different solutions.

Take your time. Work through each step.

#Install Required Packages Install the packages we are going to use today. Packages are third party add-ons or libraries that we can use in R.

```
install.packages(c("caret","dplyr","pROC","tidyr","imbalance"),repos = "http://cran.r-project.org")
```

```
## Installing packages into 'C:/Users/Eilder Jorge/Documents/R/win-library/3.6'
## (as 'lib' is unspecified)

## package 'caret' successfully unpacked and MD5 sums checked
## package 'dplyr' successfully unpacked and MD5 sums checked
## package 'pROC' successfully unpacked and MD5 sums checked
## package 'tidyr' successfully unpacked and MD5 sums checked
## package 'imbalance' successfully unpacked and MD5 sums checked
##
## The downloaded binary packages are in
##  C:\Users\Eilder Jorge\AppData\Local\Temp\Rtmp6nLki3\downloaded_packages
```

NOTE: We may need other packages, but caret should ask us if we want to load them. If you are having problems with packages, you can install the caret packages and all packages that you might need by typing (remove the # character):

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

Now, let's load the packages that we are going to use in this tutorial, the caret and imbalance packages, among others.

```
library(caret)
```

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

## Loading required package: lattice

## Loading required package: ggplot2
```

```
library(imbalance) #to be used in the optional part
```

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

```
library(dplyr)
```

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

##
## 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)
```

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

```
## 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
```

As you might already know, the caret package provides a consistent interface into hundreds of machine learning algorithms and provides useful convenience methods for data visualization, data resampling, model tuning and model comparison, among other features. It's a must have tool for machine learning projects in R.

One of its advantages for the use of caret is that it directly integrates the use of pre-processing techniques when setting up the control method for the training stage.

For more information about the caret R package see the caret package homepage.

#Step One. Get your data

For the sake of establishing a controlled scenario, we will make use of two different artificial data, namely the "circle" and "subclus" data, which can be found in different studies on imbalanced classification. These are binary problems, both in terms of attributes and classes.

In addition, we can load some data from the imbalance package, such as the glass0 one (see Wikipedia). In this case, we can observe the behavior of the different methods on a real case study.

Here is what we are going to do in this step:

0. Load the glass0 data the easy way (with imbalance package). This will only serve for the individual tasks to be accomplished afterwards.
1. Load the circle data from CSV (optional, for purists).
2. Separate the data into a training dataset and a validation dataset.

For the "subclus" data, please follow the same procedure as in the case of "circle".

##Load Data The Easy Way Fortunately, the imbalance package provides the glass0 dataset for us, among others. Load the dataset as follows:

```
# attach the glass0 dataset to the environment
data(glass0)
# rename the dataset
dataset <- glass0
```

You now have the glass0 loaded in R and accessible via the dataset variable.

I like to name the loaded data "dataset". This is helpful if you want to copy-paste code between projects and the dataset always has the same name.

##Load From CSV In the case you have previously downloaded the dataset, you may want to load the data just from a CSV file.

1. Download the circle (and subclus) dataset from PRADO (UGR).
2. Save the file as circle.csv (and subclus) in your project directory.
3. Load the dataset from the CSV file as follows:

```
# load the CSV file from the local or web directory
dataset <- read.csv("circle.csv",header = FALSE)
# set the column names in the dataset (you must know them a priori)
colnames(dataset) <- c("Att1", "Att2", "Class")
dataset$Class <- relevel(dataset$Class,"positive") #to ensure it appears at the first class
```

You now have the circle data loaded in R and accessible via the dataset variable.

#Step Two. Know your data: Summarize Dataset Now it is time to take a look at the data.

In this step we are going to take a look at the data a few different ways:

1. Dimensions of the dataset.
2. Types of the attributes.
3. Peek at the data itself.
4. Levels of the class attribute.
5. Breakdown of the instances in each class.
6. Statistical summary of all attributes.

Don't worry, each look at the data is one command. These are useful commands that you can use again and again on future projects.

##Dimensions of Dataset

We can get a quick idea of how many instances (rows) and how many attributes (columns) the data contains with the dim function.

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

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

You should see 2390 instances and 3 attributes in the case of the "circle" data.

##Types of Attributes It is a good idea to get an idea of the types of the attributes. They could be doubles, integers, strings, factors and other types.

Knowing the types is important as it will give you an idea of how to better summarize the data you have and the types of transforms you might need to use to prepare the data before you model it.

In this example, you should see that all of the inputs are double and that the class value is a factor.

```
# Check the structure and type of each attribute
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 ...
```

##Peek at the Data It is also always a good idea to actually eyeball your data. You should see the first 5 rows of the data:

```
# take a peek at the first 5 rows of the data
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
```

##Levels of the Class The class variable is a factor. A factor is a class that has multiple class labels or levels. Let's look at the levels. Notice how we can refer to an attribute by name as a property of the dataset. In the results we can see that the class has 3 different labels:

```
# list the levels for the class
levels(dataset$Class)
```

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

This is a binary classification problem.

##Statistical Summary Now finally, we can take a look at a summary of each attribute.

This includes the mean, the min and max values as well as some percentiles (25th, 50th or media and 75th e.g. values at this points if we ordered all the values for an attribute). We can see here the uneven distribution among classes: 2335 vs. 55 (circle data). This is confirmed by computing the Imbalanced Ratio (IR).

```
# summarize attribute distributions
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
```

#Visualize Dataset We now have a basic idea about the data. We need to extend that with some visualizations.

We are going to look at two types of plots:

1. Univariate plots to better understand each attribute.
2. Multivariate plots to better understand the relationships between attributes.

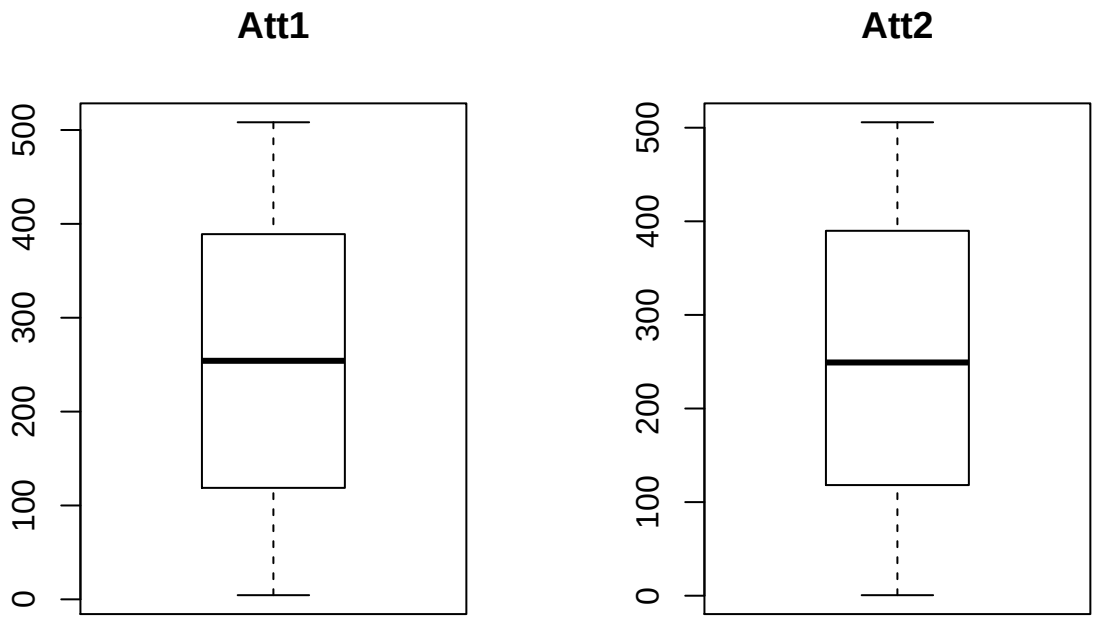
##Univariate Plots We start with some univariate plots, that is, plots of each individual variable.

It is helpful with visualization to have a way to refer to just the input attributes and just the output attributes. Let's set that up and call the inputs attributes x and the output attribute (or class) y.

```
# split input and output
x <- dataset[,1:2]
y <- dataset[,3]
```

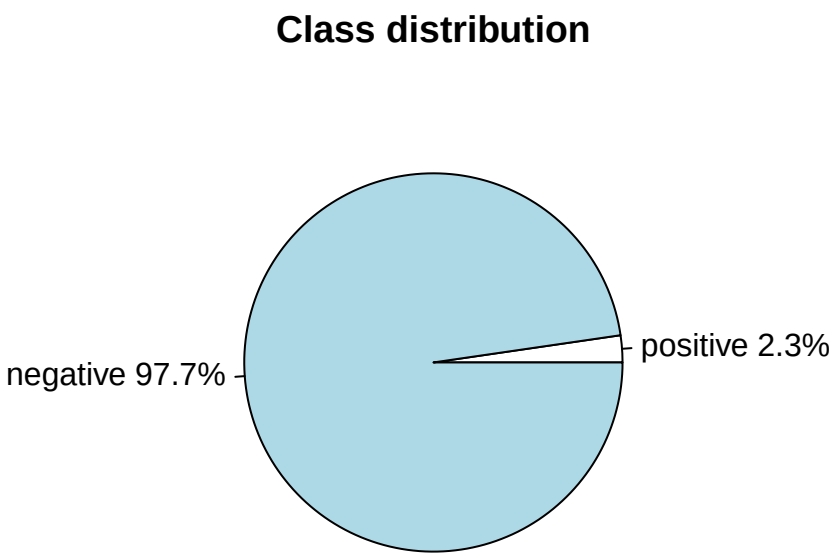
Given that the input variables are numeric, we can create box and whisker plots of each. This gives us a much clearer idea of the distribution of the input attributes:

```
# boxplot for each attribute on one image
par(mfrow=c(1,2))
  for(i in 1:2) {
  boxplot(x[,i], main=names(dataset)[i])
}
```



We can also create a barplot (better a pie chart) of the class variable to get a graphical representation of the class distribution. This confirms what we learned in the last section, that the instances are unevenly distributed across the two classes.

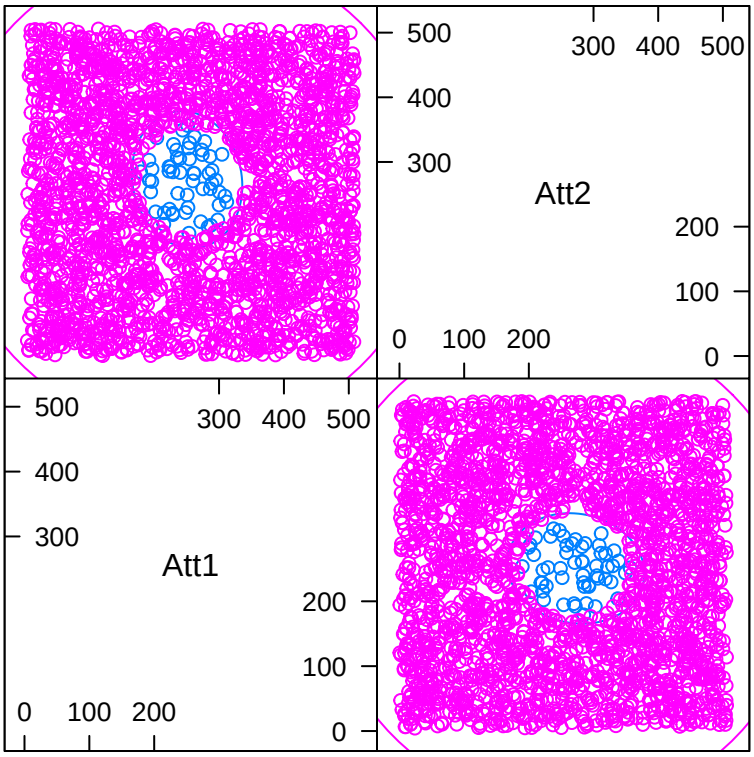
```
# barplot for class breakdown
# plot(y)

# A pie chart is best
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")
```



##Multivariate Plots Now we can look at the interactions between the variables.

First let's look at scatterplots of all pairs of attributes and color the points by class. In addition, because the scatterplots show that points for each class are generally separate, we can draw ellipses around them. We aim to see relationships between the input attributes (trends) and between attributes and the class values (ellipses). In this case the data is so simple that there is no clear conclusion.

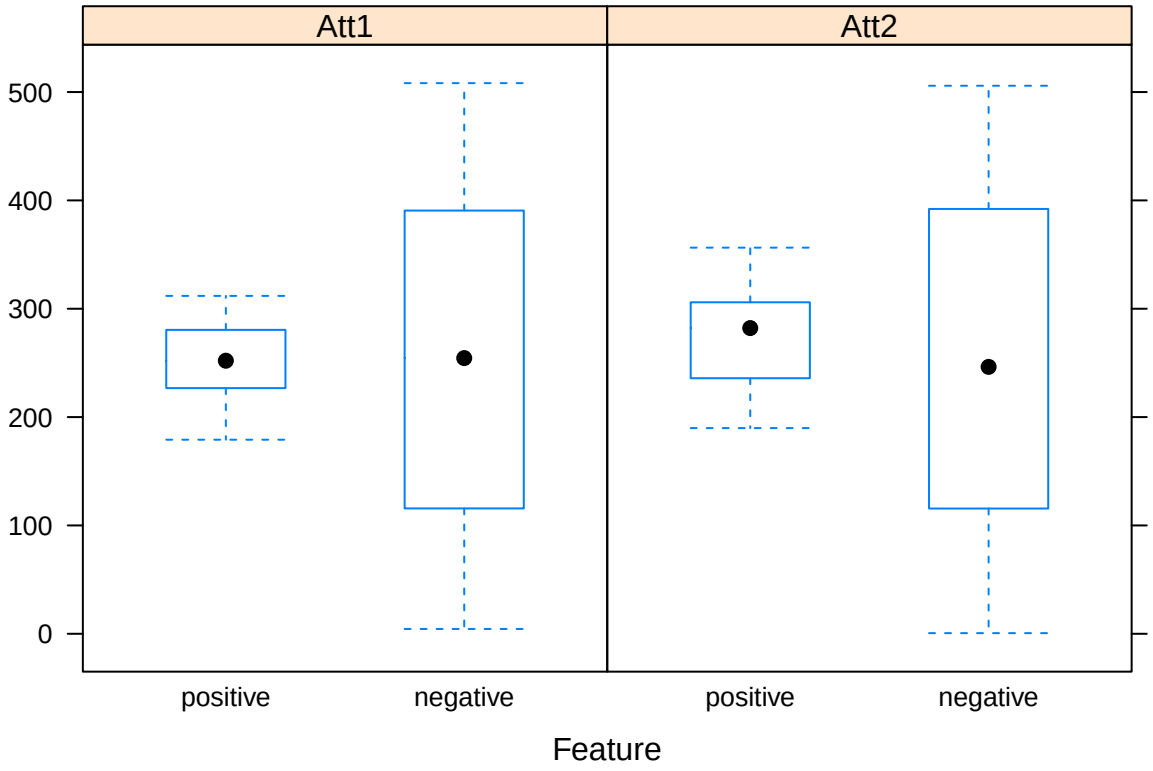
```
# scatterplot matrix
featurePlot(x=x, y=y, plot="ellipse")
```



Scatter Plot Matrix

We can also look at box and whisker plots of each input variable again, but this time broken down into separate plots for each class. This can help to tease out obvious linear separations between the classes.

```
# box and whisker plots for each attribute
featurePlot(x=x, y=y, plot="box")
```



Scatter plots can give you a great idea of what you're dealing with: it can be interesting to see how much one variable is affected by another. In other words, you want to see if there is any correlation between two variables. You can make scatterplots with the ggvis package, for example.

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

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

```
##
## 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

#Evaluate Some Algorithms Now it is time to create some models of the data and estimate their predictive ability on unseen data.

Here is what we are going to cover in this step:

1. Set-up the test harness to use hold-out validation. It is not the best practice, but will serve us for teaching purposes.
2. Apply different pre-processing approaches to the training data. Test must be kept unchanged.
3. Build a kNN model to predict the class from each different training data.
4. Compare and select the best methodology.

##Test Harness

As previously commented, we will focus on a hold-out partiton. This way, we will have just one training and one test split, which may cause a bias in our conclusions. However, it is shown how to proceed with a proper cross validation procedure.

We must reset the random number seed before each run to ensure that the evaluation of each algorithm is performed using exactly the same data splits. It ensures the results are directly comparable.

```
set.seed(42) #To ensure the same output

#An easy way to create split "data partitions":
trainIndex <- createDataPartition(dataset$Class, p = .75,
                                  list = FALSE,
                                  times = 1)
trainData <- dataset[ trainIndex,]
testData  <- dataset[-trainIndex,]

#Check IR to ensure a stratified partition
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]],]
```

##Build Models We don't know which algorithms would be good on this problem or what configurations to use.

Let's evaluate 3 different methodologies with kNN:

1. Original dataset (raw data)
2. Trivial sampling techniques (random over and under sampling).
3. SMOTE oversampling.

First of all, we need to create two auxiliar functions for the learning and estimation stages. Please take into account that we are optimising the k parameter of kNN via "grid search". If we remove this part, we may also build a general function for any possible classification algorithm.

1) Learning

```
# a) Learning function
learn_model <-function(dataset, ctrl,message){
  model.fit <- train(Class ~ ., data = dataset, method = "knn",
                     trControl = ctrl, preProcess = c("center","scale"), metric="ROC",
                     tuneGrid = expand.grid(k = c(1,3,5,7,9,11)))
  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) Prediction:

```
# 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)
}
```

##Learn and store different models of the data

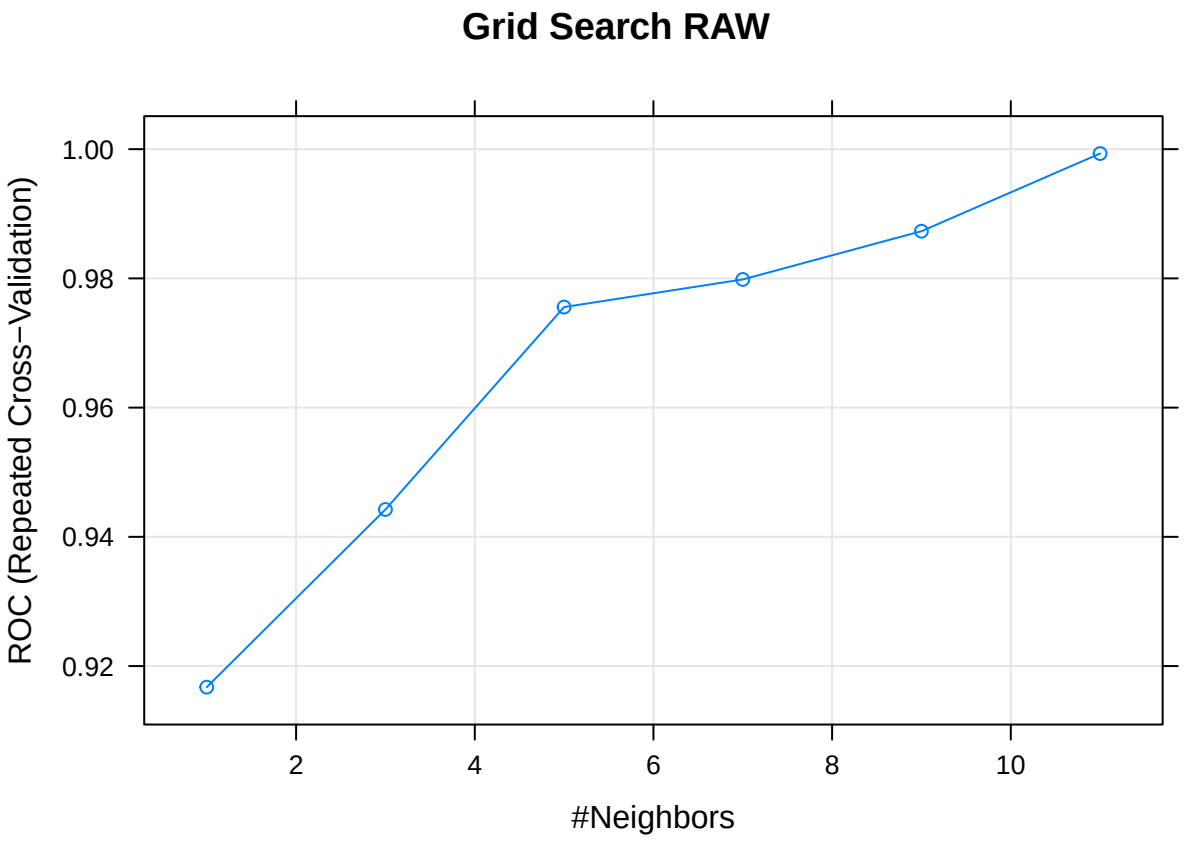
First, we check the obtain the results from original data. Please recall that an internal CV validation is used to set the best parameters for the kNN model (see above). This is stated in the trainControl call.

```
#Execute model ("raw" data)
ctrl <- trainControl(method="repeatedcv",number=5,repeats = 3,
                     classProbs=TRUE,summaryFunction = twoClassSummary)
model.raw <- learn_model(trainData,ctrl,"RAW ")
```

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

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

```
#We may decide to plot the results from the grid search of the model's parameters
plot(model.raw,main="Grid Search RAW")
```



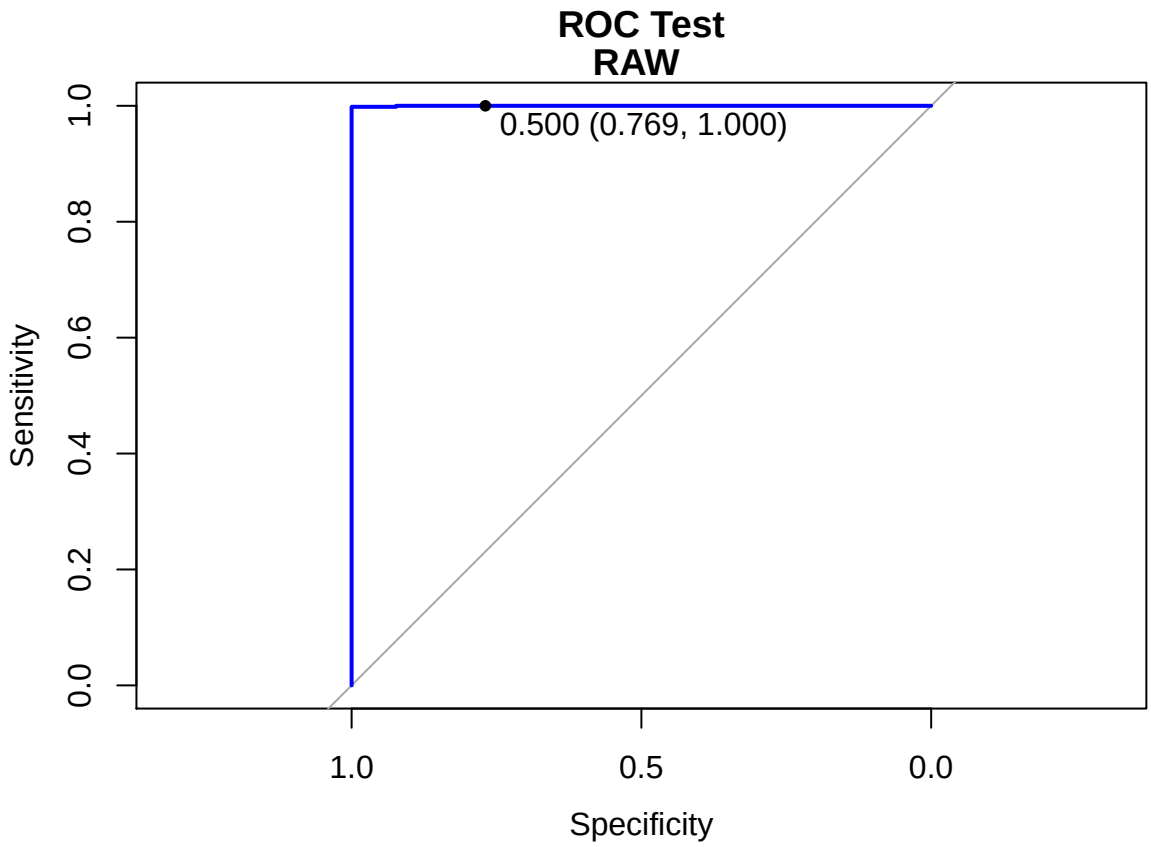

```
print(model.raw)
```

```
## 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
```



Now, we include the preprocessing stage into the control method of caret and obtain new models. First with a simple undersampling technique. The use of RUS should obtain a training set that is perfectly balanced by removing instances from the majority class in a random way.

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

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

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

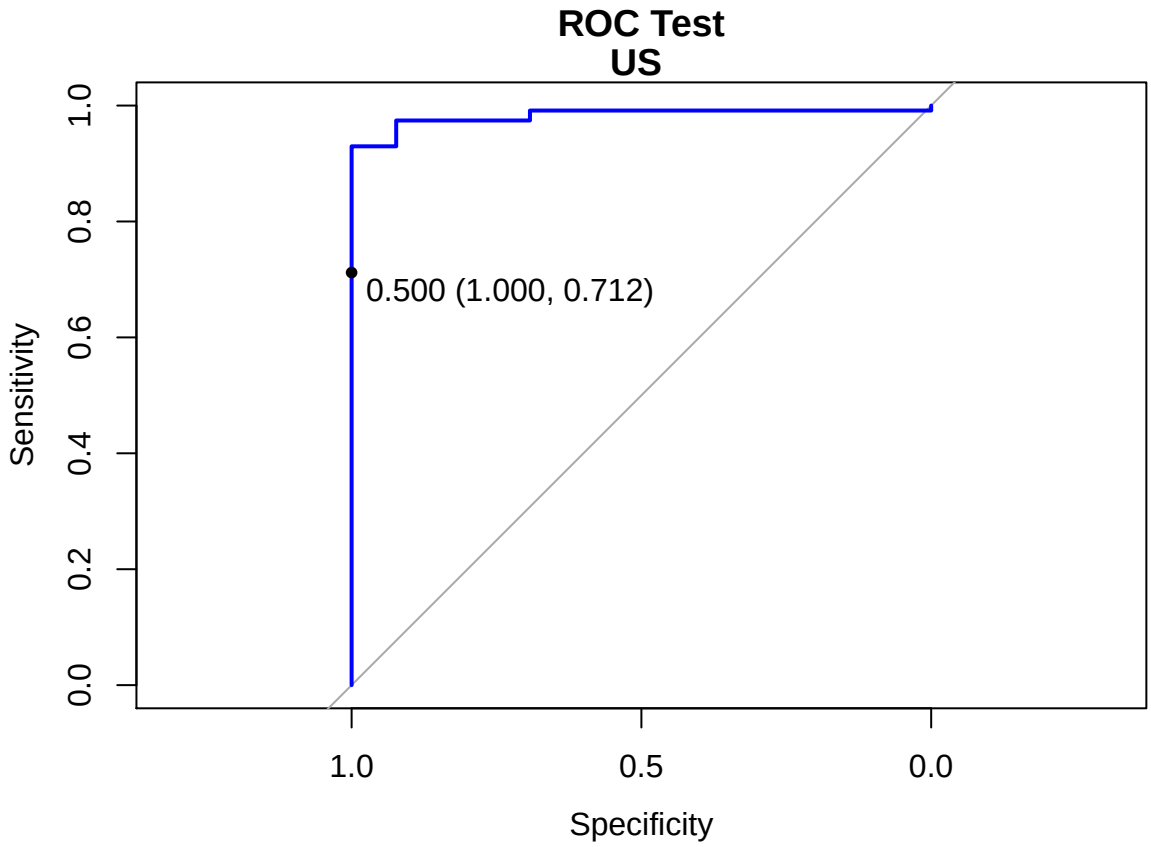
```
## Setting direction: controls > cases
```

```
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
```



Now we must check the behavior of the random oversampling approach. The use of ROS should obtain a training set that is perfectly balanced by replicating instances from the minority class in a random way.

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

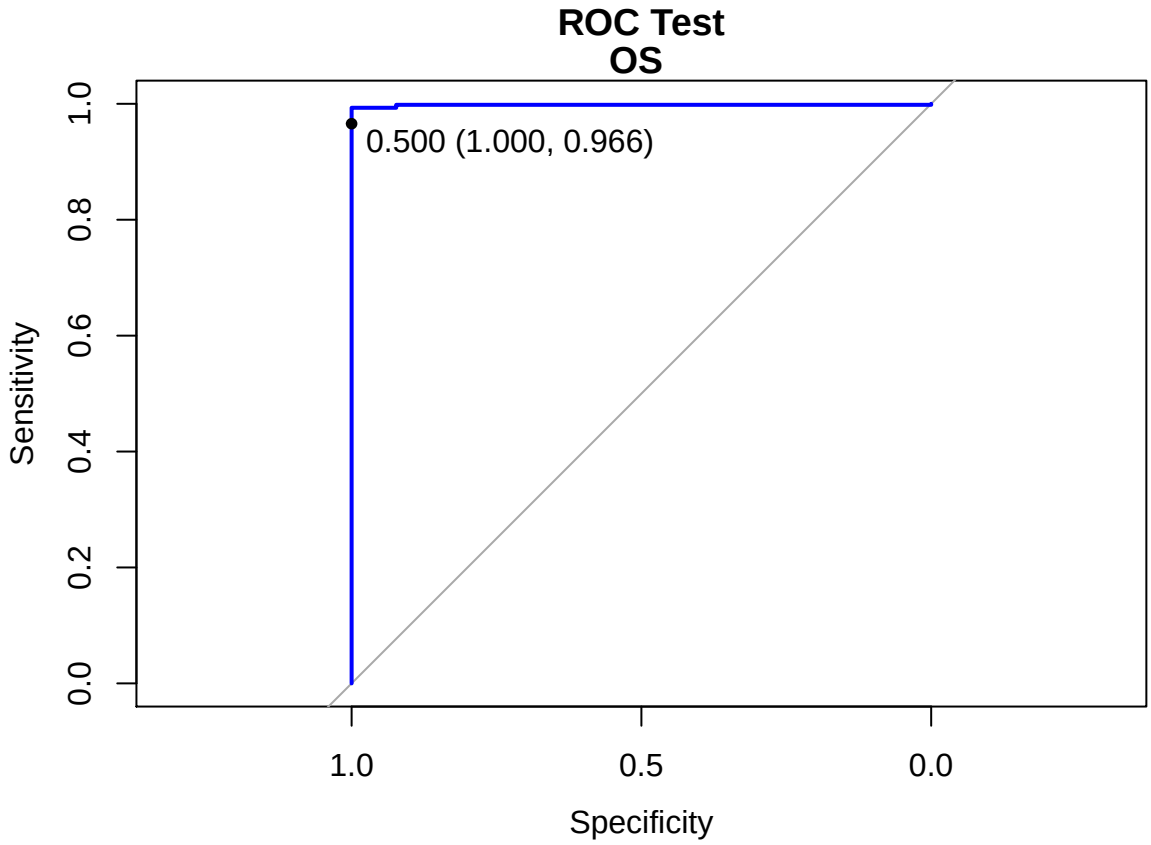
```
## Setting levels: control = positive, case = negative
```

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

```r
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
```



Finally we test the SMOTE state-of-the-art solution. 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 = twoClassSummary,sampling = "smote")
model.smt <- learn_model(trainData,ctrl,"SMT ")
```

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

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

```
## Registered S3 method overwritten by 'xts':
##   method     from
##   as.zoo.xts zoo
```

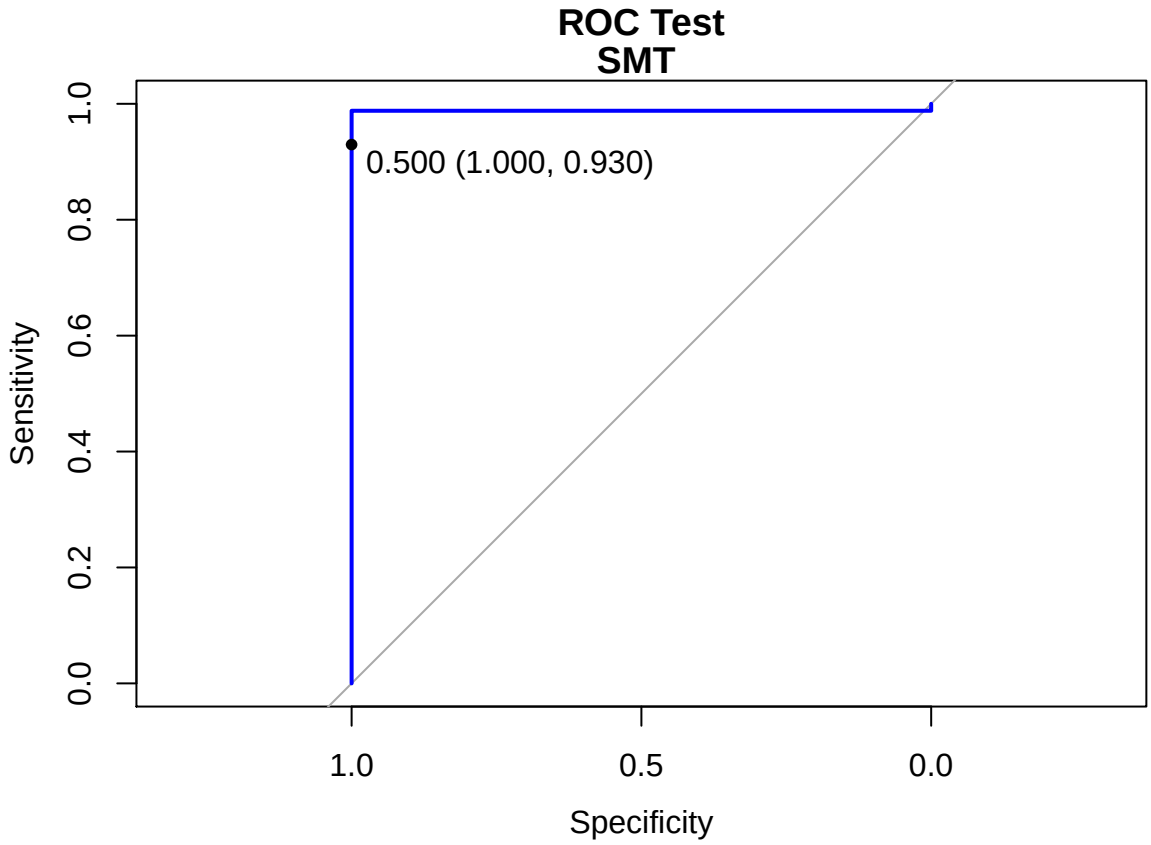
```
## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo
```

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

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

```r
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
```



##Select Best Model We now have 4 models learned from the same dataset but with different pre-processing options. Each model comprises a different accuracy estimation, and thus we need to compare the models to each other and select the most "accurate" in terms of imbalanced performance metrics, of course.

We can report on the performance of each model by first creating a list of the created models and using the summary function:

```
# summarize accuracy of models
#results <- resamples(list(lda=fit.lda, cart=fit.cart, knn=fit.knn))
models <- list(raw = model.raw,us = model.us,os = model.os,smt = model.smt)
results <- resamples(models)
summary(results)
```

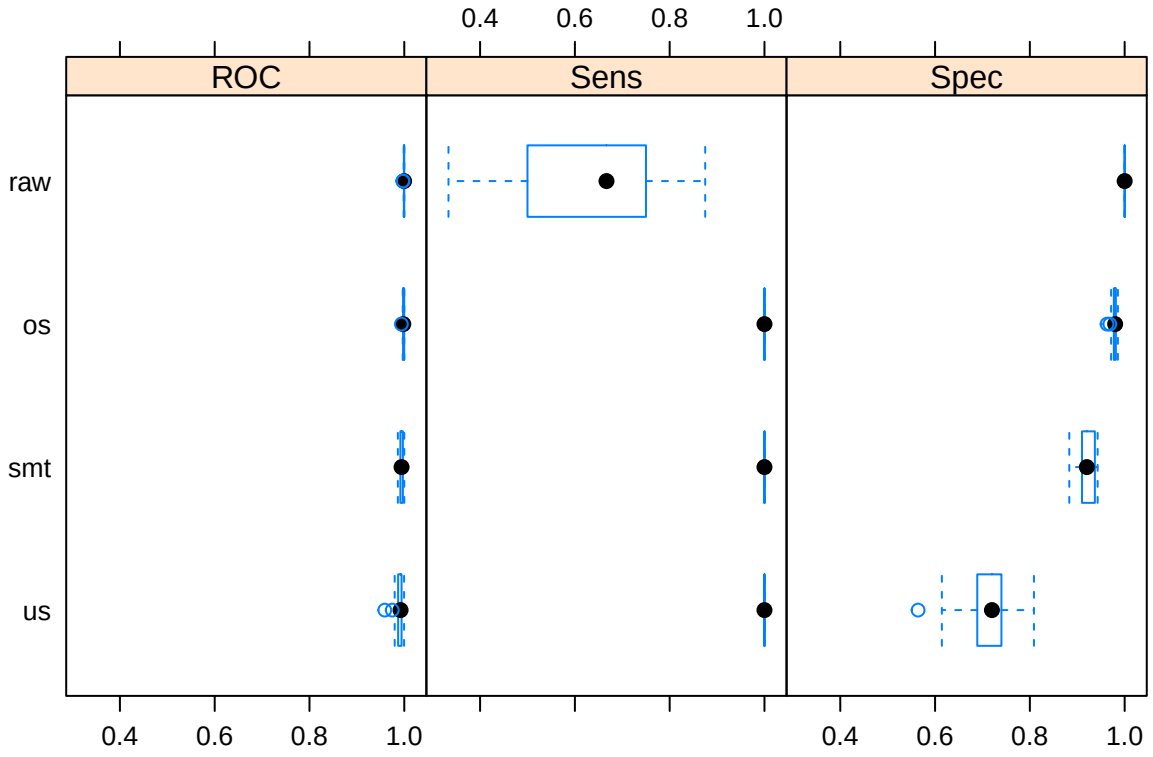
```
##
## Call:
## summary.resamples(object = results)
##
## Models: raw, us, os, smt
## Number of resamples: 15
##
## ROC
##          Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## raw 0.9968254 0.9991097 0.9996429 0.9993274 1.0000000 1.0000000    0
## us  0.9588675 0.9870818 0.9922222 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
```

We can also create a plot of the model evaluation results and compare the spread and the mean performance of each model. In the case of circle data, all preprocessing methods are equallly good in terms of sensitivity (positive class recognition), but oversampling is a bit better for specificity (negative class recognition).

The ideal procedure is to get a population of performance measures for each algorithm because each algorithm when evaluating several times (k fold cross validation).

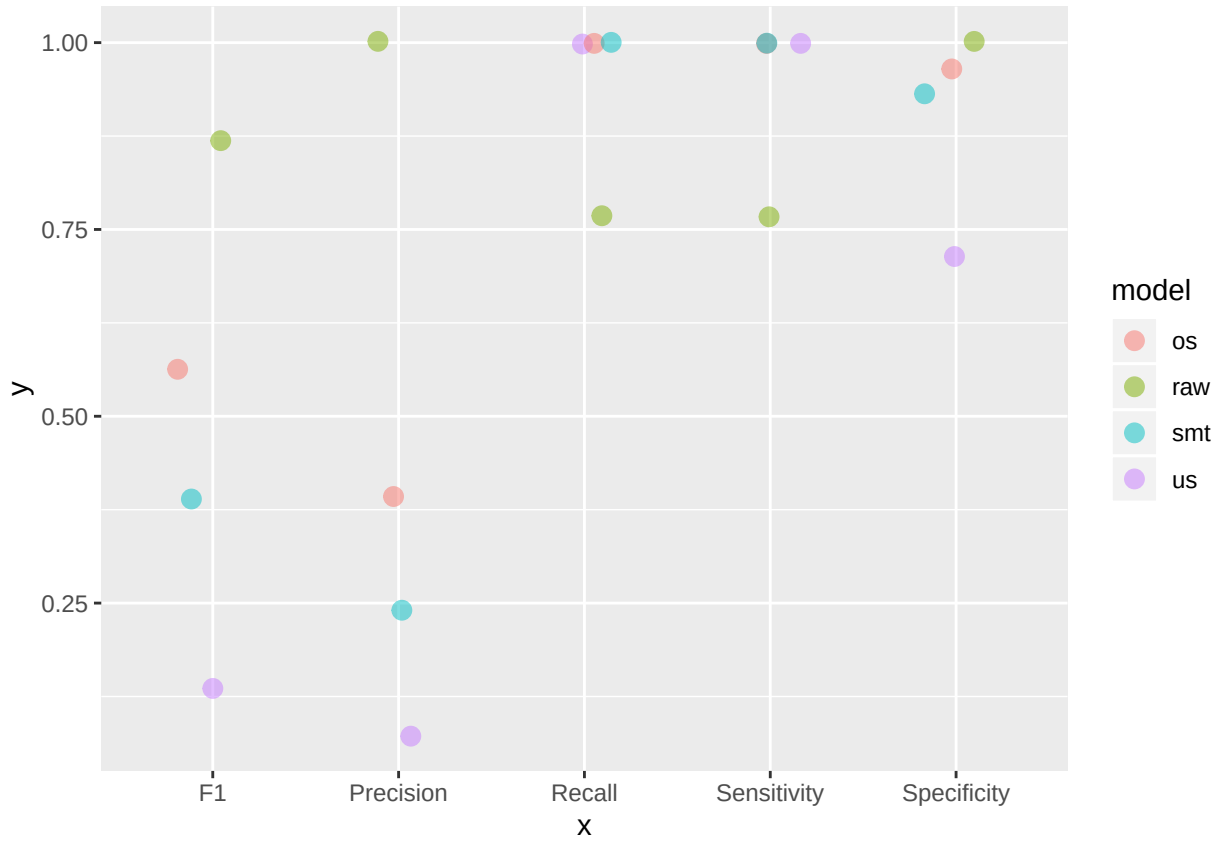
```
# compare accuracy of models
bwplot(results)
```



```r
#dotplot(results)
```

We can finally make another different plot to compare additional metrics for imbalanced classification, such as precision, recall and F1.

```r
#Carry out a comparison over all imbalanced metrics
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)
```



#Additional activities

The former tutorial served as a guide to understand the whole procedure to be carried out when addressing a classification problem that presents an uneven class distribution. Below, several tasks are proposed for extending this Rmd file to check whether all concepts have been acquired correctly.

Activity 1: Extension with additional datasets. (mandatory)

Repeat the whole procedure carried out with "circle" data using now the subclus problem. Please be sure to avoid unnecesary operations / R code chunks.

```
# load the CSV file from the local directory
data <- read.csv("subclus.csv",header = FALSE)
# set the column names in the dataset
colnames(data) <- c("Att1", "Att2", "Class")
data$Class <- relevel(data$Class,"positive") #to ensure it appears at the first class
# summarize statistical values
summary(data)
```

```
##       Att1              Att2             Class
##  Min.   :-84.00   Min.   :-282.0   positive:100
##  1st Qu.: 65.75   1st Qu.: 155.8   negative:500
##  Median :213.00   Median : 572.5
##  Mean   :214.06   Mean   : 574.5
##  3rd Qu.:365.50   3rd Qu.: 961.2
##  Max.   :483.00   Max.   :1481.0
```

```r
imbalanceRatio(data)
```

```
## [1] 0.2
```

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

Renderer: SVG | Canvas

Download

```r
# Create split "data partitions":
set.seed(13)

trainIndex <- createDataPartition(data$Class, p = .85, list = FALSE, times = 1)
trainData <- data[ trainIndex,]
testData  <- data[-trainIndex,]

#Check IR to ensure a stratified partition
imbalanceRatio(trainData)
```

```
## [1] 0.2
```

```r
imbalanceRatio(testData)
```

```
## [1] 0.2
```

```r
# Execute models
ctrl <- trainControl(method="repeatedcv",number=5,repeats = 3,classProbs=TRUE,summaryFunction = twoClass
model.raw <- learn_model(trainData,ctrl)
```

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

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

```r
ctrl <- trainControl(method="repeatedcv",number=5,repeats = 3,classProbs=TRUE,summaryFunction = twoClass
model.us <- learn_model(trainData,ctrl)
```

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

```r
ctrl <- trainControl(method="repeatedcv",number=5,repeats = 3,classProbs=TRUE,summaryFunction = twoClass
model.os <- learn_model(trainData,ctrl)
```

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

```r
ctrl <- trainControl(method="repeatedcv",number=5,repeats = 3,classProbs=TRUE,summaryFunction = twoClass
model.smt <- learn_model(trainData,ctrl)
```

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

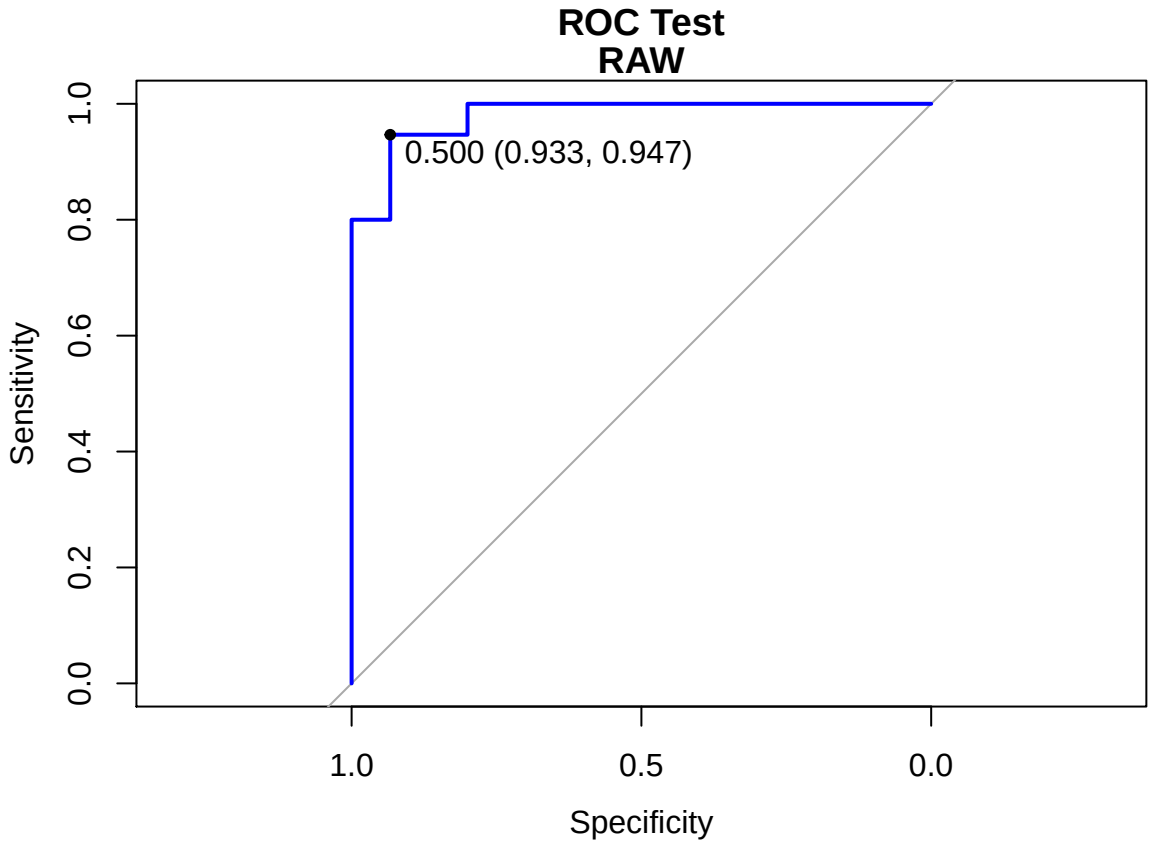
```r
# Summarize performance of models
models <- list(raw = model.raw,undersampling = model.us,oversampling = model.os,smote = model.smt)
results <- resamples(models)
summary(results)
```

```
##
## Call:
## summary.resamples(object = results)
##
## Models: raw, undersampling, oversampling, smote
## Number of resamples: 15
##
## ROC
##                    Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## raw           0.8958478 0.9243945 0.9397924 0.9411534 0.9608997 0.9809689
## undersampling 0.8598616 0.9074394 0.9235294 0.9247059 0.9463668 0.9636678
## oversampling  0.8965398 0.9500000 0.9532872 0.9540023 0.9671280 0.9754325
## smote         0.8993080 0.9382353 0.9508651 0.9485582 0.9641869 0.9858131
##               NA's
## raw              0
## undersampling    0
## oversampling     0
## smote            0
##
## Sens
##                    Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## raw           0.3529412 0.5882353 0.6470588 0.6549020 0.7352941 0.8823529
## undersampling 0.8235294 0.8823529 0.9411765 0.9411765 1.0000000 1.0000000
## oversampling  0.8823529 0.9411765 1.0000000 0.9725490 1.0000000 1.0000000
## smote         0.6470588 0.8235294 0.9411765 0.8980392 0.9705882 1.0000000
##               NA's
## raw              0
## undersampling    0
## oversampling     0
## smote            0
##
## Spec
##                    Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## raw           0.9294118 0.9411765 0.9529412 0.9537255 0.9647059 0.9882353
## undersampling 0.7411765 0.7882353 0.8000000 0.8094118 0.8411765 0.9058824
## oversampling  0.7058824 0.7764706 0.8000000 0.7984314 0.8352941 0.8588235
## smote         0.7764706 0.8352941 0.8823529 0.8745098 0.9058824 0.9529412
##               NA's
## raw              0
## undersampling    0
## oversampling     0
## smote            0
```

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

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction positive negative
##   positive       14        4
##   negative        1       71
##
##                Accuracy : 0.9444
##                  95% CI : (0.8751, 0.9817)
##     No Information Rate : 0.8333
##     P-Value [Acc > NIR] : 0.001441
##
##                   Kappa : 0.8148
##
##  Mcnemar's Test P-Value : 0.371093
##
##             Sensitivity : 0.9333
##             Specificity : 0.9467
##          Pos Pred Value : 0.7778
##          Neg Pred Value : 0.9861
##              Prevalence : 0.1667
##          Detection Rate : 0.1556
##    Detection Prevalence : 0.2000
##       Balanced Accuracy : 0.9400
##
##        'Positive' Class : positive
##
```

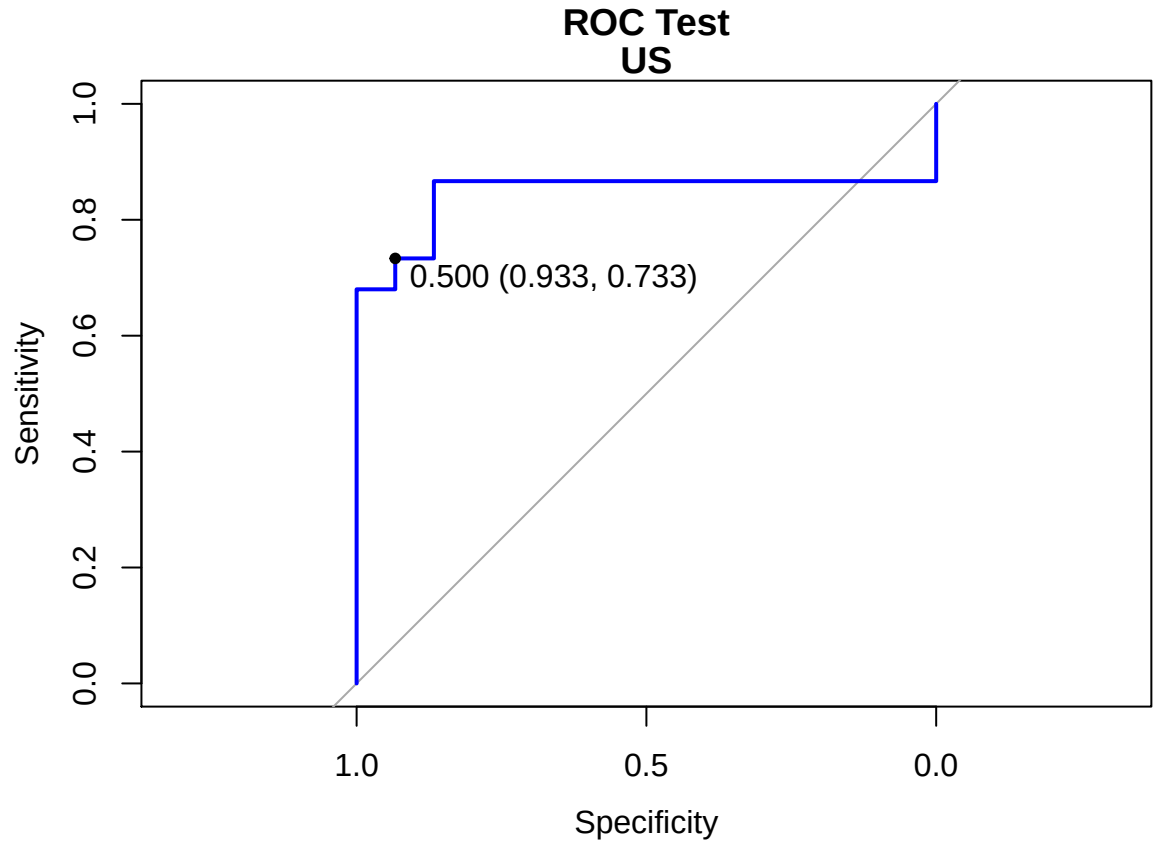
```
## Setting levels: control = positive, case = negative
## Setting direction: controls > cases
```




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

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction positive negative
##   positive       14       20
##   negative        1       55
##
##                Accuracy : 0.7667
##                  95% CI : (0.6657, 0.8494)
##     No Information Rate : 0.8333
##     P-Value [Acc > NIR] : 0.9623
##
##                   Kappa : 0.4425
##
##  Mcnemar's Test P-Value : 8.568e-05
##
##             Sensitivity : 0.9333
##             Specificity : 0.7333
##          Pos Pred Value : 0.4118
##          Neg Pred Value : 0.9821
##              Prevalence : 0.1667
##          Detection Rate : 0.1556
##    Detection Prevalence : 0.3778
```

```
##       Balanced Accuracy : 0.8333
##
##        'Positive' Class : positive
##

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

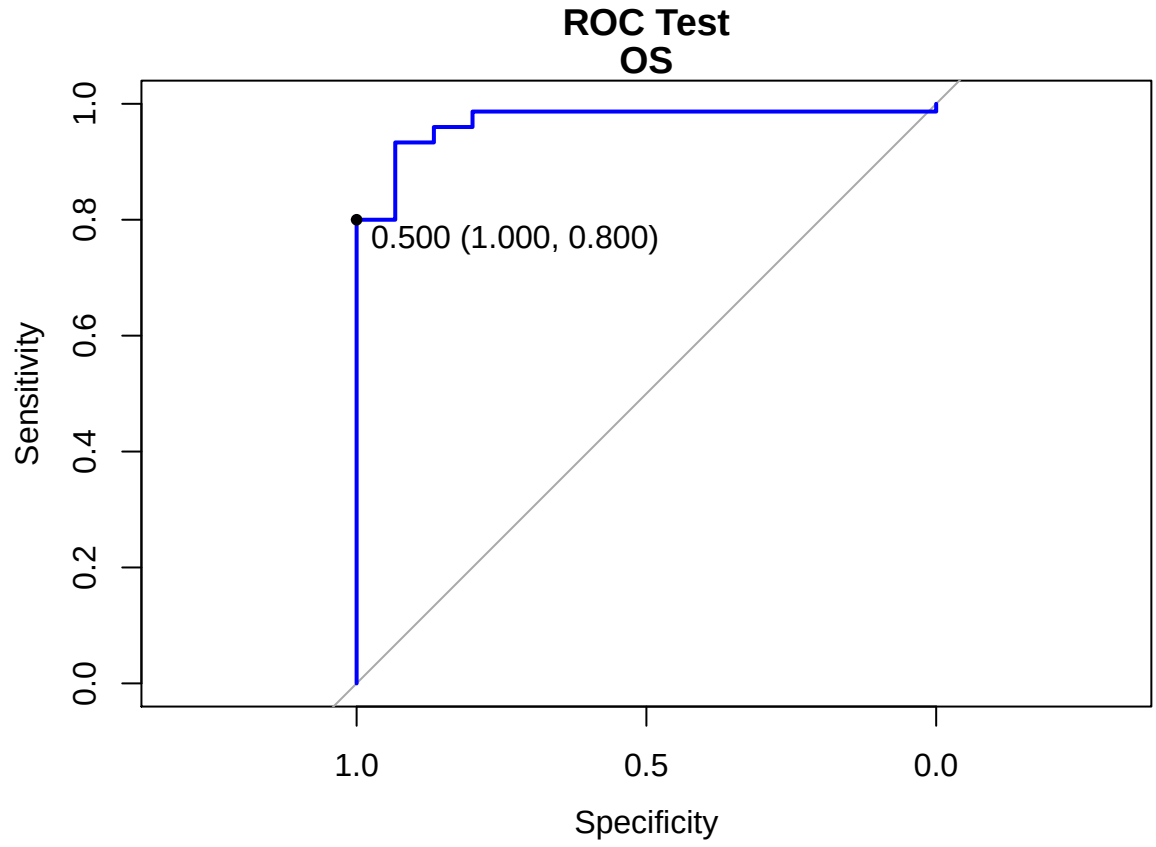



```
test.oversampling <- test_model(testData,model.os,"OS")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction positive negative
##   positive       15       16
##   negative        0       59
##
##                Accuracy : 0.8222
##                  95% CI : (0.7274, 0.8948)
##     No Information Rate : 0.8333
##     P-Value [Acc > NIR] : 0.6735536
##
##                   Kappa : 0.5514
##
##  Mcnemar's Test P-Value : 0.0001768
```

```
##
##             Sensitivity : 1.0000
##             Specificity : 0.7867
##          Pos Pred Value : 0.4839
##          Neg Pred Value : 1.0000
##              Prevalence : 0.1667
##          Detection Rate : 0.1667
##    Detection Prevalence : 0.3444
##       Balanced Accuracy : 0.8933
##
##        'Positive' Class : positive
##
```

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

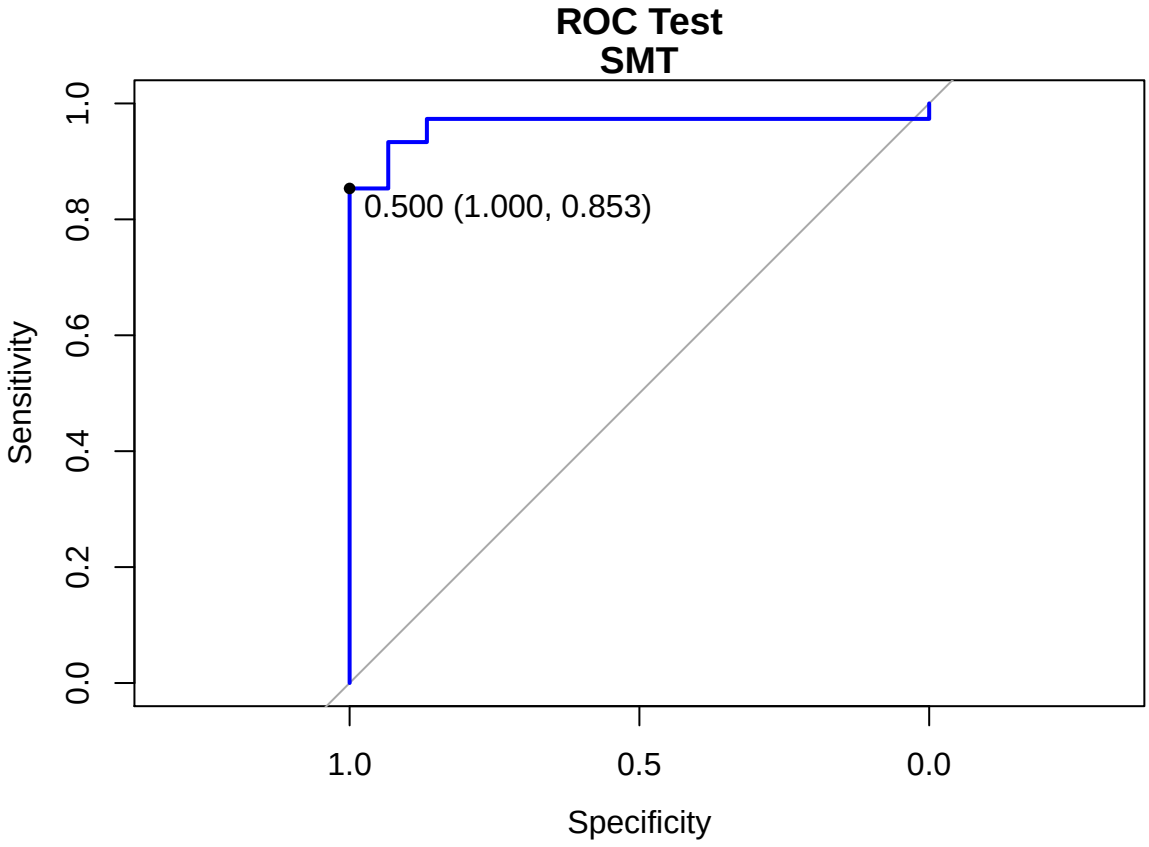



```
test.smote <- test_model(testData,model.smt,"SMT")
```

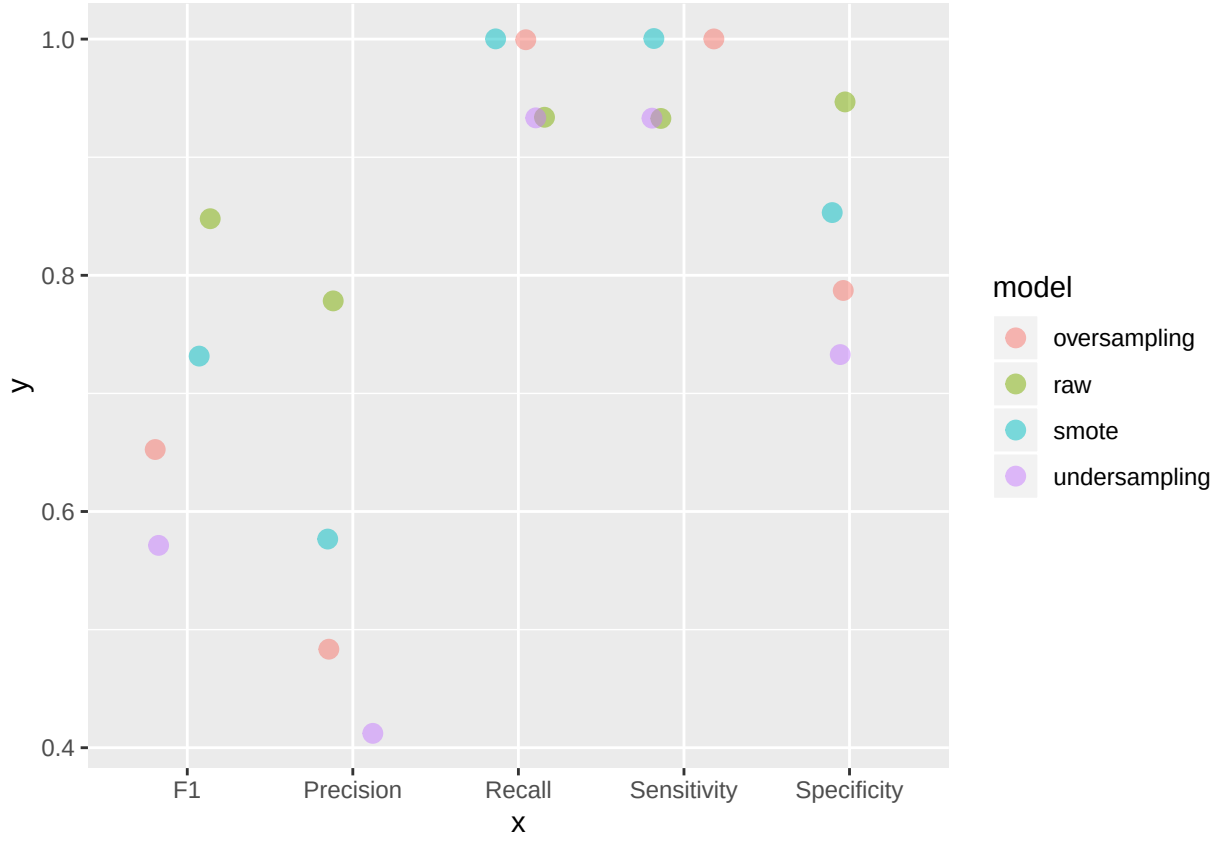
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction positive negative
##   positive       15       11
##   negative        0       64
##
```

```
##                Accuracy : 0.8778
##                  95% CI : (0.7918, 0.9374)
##     No Information Rate : 0.8333
##     P-Value [Acc > NIR] : 0.160939
##
##                   Kappa : 0.6598
##
##  Mcnemar's Test P-Value : 0.002569
##
##             Sensitivity : 1.0000
##             Specificity : 0.8533
##          Pos Pred Value : 0.5769
##          Neg Pred Value : 1.0000
##              Prevalence : 0.1667
##          Detection Rate : 0.1667
##    Detection Prevalence : 0.2889
##       Balanced Accuracy : 0.9267
##
##        'Positive' Class : positive
##
```

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



```r
# Carry out a visual comparison over all imbalanced metrics

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)) {
  test_model <- get(paste0("test.", name))
  comparison[comparison$model == name, ] <- filter(comparison, model == name) %>%
    mutate(Sensitivity = test_model$byClass["Sensitivity"],
           Specificity = test_model$byClass["Specificity"],
           Precision = test_model$byClass["Precision"],
           Recall = test_model$byClass["Recall"],
           F1 = test_model$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)
```



Write a short comment on the main conclusions obtained throughout the experimental analysis. Focus on the most interesting behavior achieved by the methods applied, observing whether there are some special

capabilities to be stressed by any particular approach.

The analysis shows that in imbalanced datasets using learning algorithms with raw data will give high precision, specially on the predominant class, at the cost of poor precision in the class with less examples. This can be mitigated using preprocessing algorithms such as undersampling, oversampling and SMOTE. All 3 can be used in R with the caret library with ease thanks to the sampling parameter of the train control, allowing us to test them in a dataset without much difficulty. Undersampling can give drastic results, at the cost of low precision and specificity, but giving very high recall and sensitivity values, which means it can be useful on cases where false positives are preferable over assigning negative to true positive, such as terminal disease tests.

Oversampling gives better results overall, at the cost of having slightly lower specificity, which might be preferable in general data.

SMOTE strikes a balance between both of them, with high values in all unbalanced metrics, but not as optimal as undersampling or oversampling for specific cases.

In general the application of one or the other will depend on the problem to be solved.

Activity 2: Using the "imbalance" library (mandatory)

As we have commented, there exists a CRAN package named as "imbalance" that implements some of the most well-known data preprocesing techniques for imbalanced classification. We must take a closer look to the documentation in both the imbalance package homepage. or the help function

```
help("imbalance") #see documentation in the right botton corner
```

```
## starting httpd help server ... done
```

By using the "imbalance" library we may consider the application of advanced techniques based on SMOTE. For that purpose, we must focus on the "oversample" function:

```
help("oversample") #see documentation in the right botton corner
```

It is your turn to select up to four different SMOTE techniques and apply them over some of the datasets provided by the package (ecoli1, glass0, haberman or yeast4, for example). Are there any significant differences among the results?

```
# Load the data
data(yeast4)
yeast<-yeast4
yeast$Erl=NULL #factor, can't be used with the interpolation of SMOTE methods
yeast$Class <- relevel(yeast$Class,"positive")
# Apply preprocessing with oversample function
BLSmoteyeast<-oversample(yeast,ratio=0.75,method="BLSMOTE")
```

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

```
DBSmoteyeast<-oversample(yeast,ratio=0.75,method="DBSMOTE")
```

```
## [1] 2
## [1] 4
```

```
## [1] 4
## [1] 2
## [1] 2
## [1] 2
## [1] 2
## [1] 3
## [1] 4
## [1] 2
## [1] 2
## [1] 2
## [1] 2
## [1] 3
## [1] 3
## [1] 2
## [1] 2
## [1] 4
## [1] 2
## [1] 5
## [1] 3
## [1] 2
## [1] 3
## [1] 2
## [1] 3
## [1] 4
## [1] 3
## [1] 3
## [1] 2
## [1] 2
## [1] 2
## [1] 2
## [1] 3
## [1] 2
## [1] 4
## [1] 2
## [1] 2
## [1] 4
## [1] 2
## [1] 4
## [1] 3
## [1] 3
## [1] 3
## [1] 4
## [1] 2
## [1] 4
## [1] 2
## [1] 3
## [1] 3
## [1] "DBSMOTE is Done"
```

```
Smoteyeast<-oversample(yeast,ratio=0.75,method="SMOTE")
ANSmoteyeast<-oversample(yeast,ratio=0.75,method="ANSMOTE")
```

```
## [1] "ANS is done"
```

```r
# Check results with kNN, DT or any other classifier
ctrl <- trainControl(method="repeatedcv",number=5,repeats = 3,
                     classProbs=TRUE,summaryFunction = twoClassSummary)
model.bl <- learn_model(BLSmoteyeast,ctrl)
```

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

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

```r
model.db <- learn_model(DBSmoteyeast,ctrl)
```

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

```r
model.sm <- learn_model(Smoteyeast,ctrl)
```

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

```r
model.an <- learn_model(ANSmoteyeast,ctrl)
```

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

```r
models <- list(BLSmote = model.bl,DBSmote = model.db,Smote = model.sm,ANSmote = model.an)
results <- resamples(models)
summary(results)
```

```
##
## Call:
## summary.resamples(object = results)
##
## Models: BLSmote, DBSmote, Smote, ANSmote
## Number of resamples: 15
##
## ROC
##              Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## BLSmote 0.9810145 0.9879953 0.9905594 0.9891066 0.9909457 0.9939746    0
## DBSmote 0.9774005 0.9840706 0.9883883 0.9871671 0.9912770 0.9936494    0
## Smote   0.9752317 0.9862977 0.9889312 0.9882782 0.9915666 0.9939146    0
## ANSmote 0.9800340 0.9845203 0.9874614 0.9867504 0.9881247 0.9933485    0
##
## Sens
##              Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## BLSmote 0.9767442 0.9837209 0.9906977 0.9872868 0.9906977 0.9953488    0
## DBSmote 0.9767442 0.9883721 0.9906977 0.9900775 0.9953488 1.0000000    0
## Smote   0.9906977 0.9976744 1.0000000 0.9984496 1.0000000 1.0000000    0
## ANSmote 0.9860465 0.9930233 0.9953488 0.9953488 1.0000000 1.0000000    0
##
## Spec
```

```
##              Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## BLSmote 0.9059233 0.9127397 0.9372822 0.9320793 0.9441534 0.9547038    0
## DBSmote 0.8745645 0.8935455 0.9020979 0.8999781 0.9075071 0.9163763    0
## Smote   0.8188153 0.8603774 0.8745645 0.8732130 0.8902439 0.9024390    0
## ANSmote 0.8601399 0.8693380 0.8846154 0.8848615 0.8902439 0.9265734    0
```

It looks like BLSmote is very good for this dataset, giving a high ROC curve mean, max, and the highest specificity, just sacrificing a very small amount of sensitivity. The other 3 SMOTE methods give very similar results, with Smote and ANSmote being identical in several values.

Now, make a plot comparison between the original and preprocessed dataset (only for SMOTE, for example). Recall that, being a 2D plot, you must only two of the input columns. Alternatively, you can carry out a "tsne" prior to the plot.

```
# Visualize the data distribution between original and preprocess data.
yeast %>% ggvis(~Mcg, ~Mit, fill=~Class) %>% layer_points()
```

Renderer: SVG | Canvas

Download

```
Smoteyeast %>% ggvis(~Mcg, ~Mit, fill=~Class) %>% layer_points()
```

Renderer: SVG | Canvas

Download

```
#Due to the nature of R markdown, I couldn't find a way to plot both at the same time and make them rea
```

Activity 3: Analyze the behavior of SMOTE preprocessing (optional)

In this last part of the practice, we intend to analyse the influence of the different parameters of SMOTE. In fact, during the theoretical classes, it was indicated that many of the SMOTE parameters could have a certain importance in terms of the quality of the new synthetic examples generated on the training set.

Therefore, the objective of this task is to contrast some of them to see which ones can have more influence, or which ones can be significant values to observe differences in the results.

To carry out this activity, the first issue is to determine the experimental framework. As for the datasets to be used, "subclus" and "circle" can be selected by default, although the study will be more relevant when the number of problems, both synthetic and real, is greater. In any case, the student should use a cross validation technique to check the results.

As a base classifier to analyze the behavior, the student may use by default kNN with K = 1 or K = 3. It could also be interesting to analyze the results with the C4.5 decision tree or even Random Forest or any other quality technique that the student consider to be appropriate.

Finally, it would remain to be discussed which parameters are appropriate for the study, and what range of values to use. The most direct parameters would be K for the number of neighbors chosen (for example, between K = 1, K = 5, K = N/2 with N number of positive instances, etc.), and the percentage of oversampling (double the minority class, 50-50 class ratio, 50% minority over majority class, among others).

To do so, you can either perform an ad hoc implementation of SMOTE, or analyze among those available in the different R packages, the one that allows you to perform a "Racing" of the parameters (clue: check the current available imbalanced packages in CRAN).

Build the corresponding tables of results and make a brief analysis if any interesting patterns are observed during the experimental analysis.

#Final Comments Hope you enjoyed this tutorial about Imbalanced Classification in Machine Learning. If you need further details on how to perform any kind of task, please ask me via email at alberto@decsai.ugr.es