


---

# **Imbalanced classification**

Minería de datos: aspectos avanzados

Francisco Luque Sánchez

21/12/2019

The slide features a white background with a large yellow triangle on the right side and two overlapping orange triangles at the bottom left corner.

## 1 Introduction

In this report, the problem of imbalanced classification will be addressed. In the first section, we will show a typical workflow to solve the imbalance classification problem, using the dataset *Subclus*. This dataset is an artificially generated two dimensional dataset whose positive class is grouped in a few small subgroups. In the second section, the performance of some SMOTE-based oversampling methods will be tested over that same dataset. Finally, in the third section, we will dig deeper in the classic version of SMOTE, trying to understand how the solution of the problem is influenced by its parameters.

## 2 Standard imbalanced classification pipeline

In this section, a classical pipeline of imbalanced classification will be shown. We begin by loading the dataset and renaming the variables properly:

```
## Dataset loading and column names setting
dataset <- read.csv("subclus.csv")
colnames(dataset) <- c("Att1", "Att2", "Class")
dataset$Class <- relevel(dataset$Class, "positive")
```

At first, we are interested in knowing about the dataset (variables, types, dimensions...):

```
## DATASET SUMMARY
## Dimensions
dim(dataset)
```

```
## [1] 599  3
```

```
## Structure and type
str(dataset)
```

```
## 'data.frame':  599 obs. of  3 variables:
## $ Att1 : int  187 290 194 204 196 201 289 116 199 174 ...
## $ Att2 : int   34 -57 -80 89 -81 -17 4 -95 38 33 ...
## $ Class: Factor w/ 2 levels "positive","negative": 1 1 1 1 1 1 1 1 1 1 ...
```

```
## First rows of the data
```

```
kable(head(dataset))
```

Att1	Att2	Class
187	34	positive
290	-57	positive
194	-80	positive
204	89	positive
196	-81	positive
201	-17	positive

```
## Class levels
```

```
levels(dataset$Class)
```

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

As we can see in the output of previous commands, our dataset is composed of 599 examples of 3 variables (two numeric and the class). It is a binary classification problem, with classes named *negative* and *positive*.

```
## Columns summarization
```

```
summary(dataset)
```

```
##      Att1      Att2      Class
##  Min.   :-84.0   Min.   :-282.0 positive: 99
## 1st Qu.: 65.5   1st Qu.: 156.5 negative:500
## Median :213.0   Median : 576.0
## Mean   :214.2   Mean    : 575.3
## 3rd Qu.:366.0   3rd Qu.: 961.5
## Max.   :483.0   Max.    :1481.0
```

```
## Imbalance ratio
```

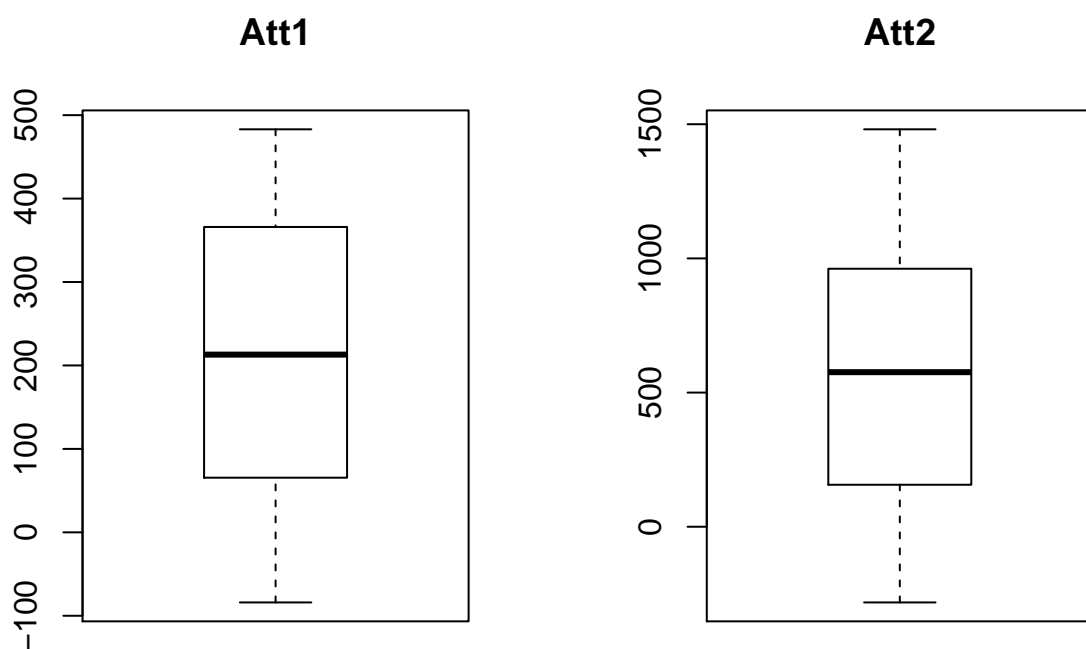
```
imbalanceRatio(dataset)
```

```
## [1] 0.198
```

The imbalance ratio of the dataset is not very pronounced (approximately 1 to 5). It is far from the 1 to 40 that we had in other examples. However, it is important enough to be addressed as an imbalanced dataset. Now, we will try and visualize the data. We begin with a boxplot of the attributes and a piechart of the classes distribution:

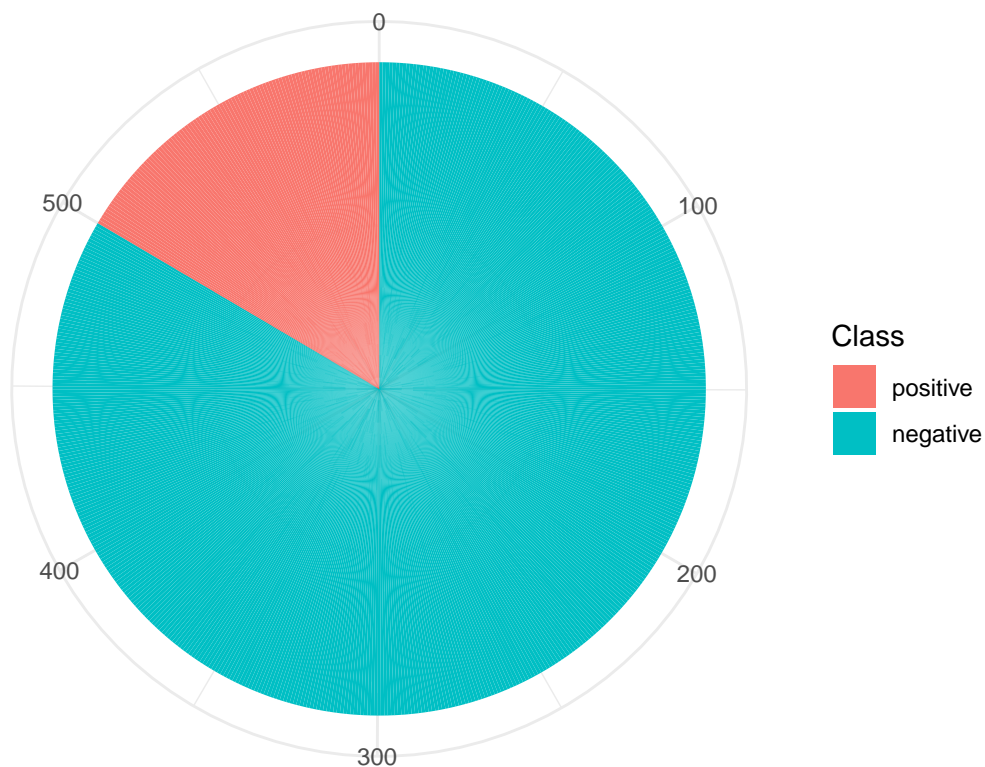
```
## Dataset visualization
x <- dataset[,1:2]
y <- dataset[,3]

## Attributes boxplot
par(mfrow=c(1,2))
out <- sapply(1:2, function(i) boxplot(x[,i], main=names(dataset)[i]))
```



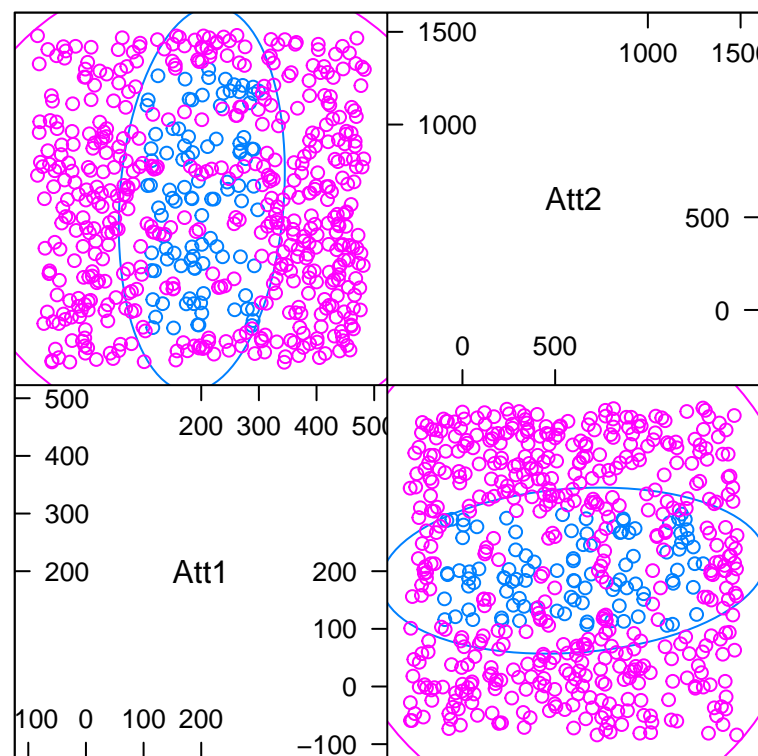
The data distribution along the variables is slightly different. The second attribute is much sparser than the first, with a range three times bigger.

```
## Classes piechart
ggplot(dataset, aes(x="", y=1, fill=Class))+
  geom_bar(width = 1, stat = "identity")+
  coord_polar("y", start=0)+ theme_minimal()+
  theme(axis.title.x=element_blank(), axis.title.y = element_blank())
```



As we said before, in the chart can be seen that there are approximately 5 times more data in the negative class than in the positive one. Now, we will plot the data in a scatter plot, marking the examples with the class they belong to:

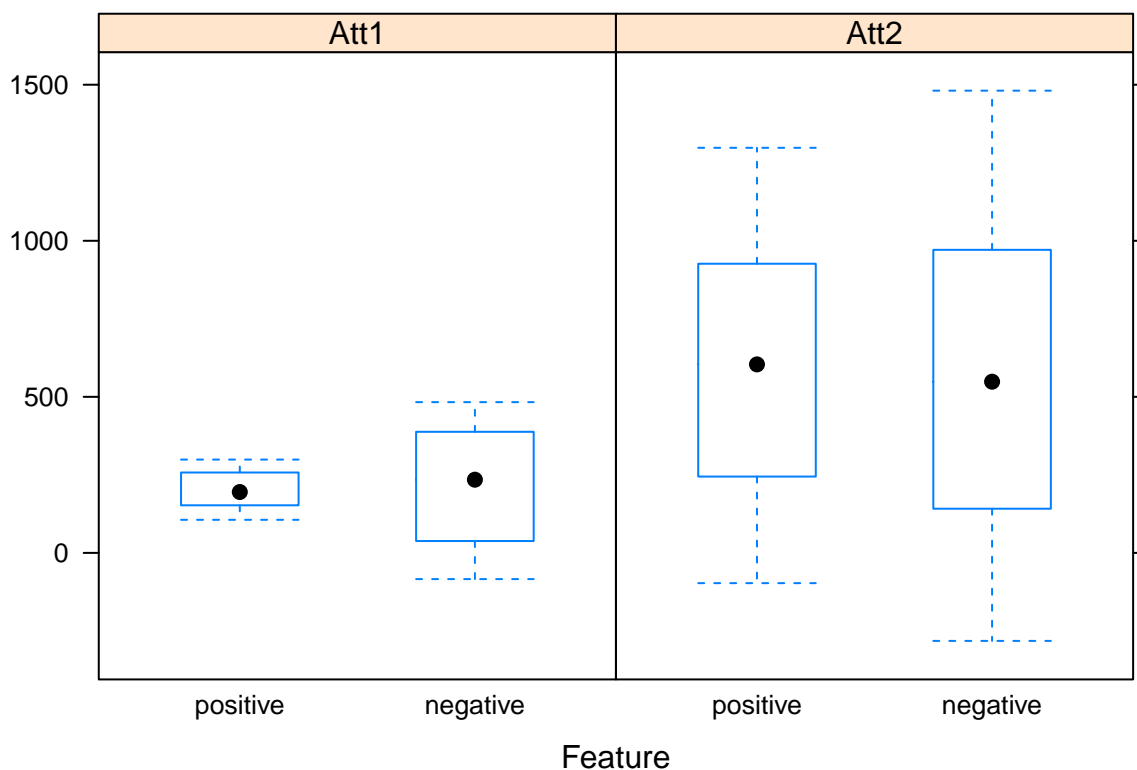
```
## Scatterplot
featurePlot(x=x, y=y, plot="ellipse")
```



Scatter Plot Matrix

The graph shows that the positive class is placed in the center of the variables space, forming a big vertical ellipse. However, if we look closely, we can find five smaller groups separated by negative examples. This disposition will undermine the performance of the classical SMOTE method for oversampling, since some of the artificial instances will lay outside the boundaries of the minority class.

```
## Per class boxplot
featurePlot(x=x, y=y, plot="box")
```



The previous boxplot exhibits the distribution of variables depending on the class. As we have previously observed in the scatter plot, data restricted to the minority class covers a less wide range of points, and thus they form a group of points in the center of the space.

After a first step of data exploration, a phase of models evaluation will be performed. Our aim is to conclude which balancing criteria produces better results with our dataset. At first, we will compare four different approaches:

- No balancing: Training the model with raw data
- Undersampling: Randomly deleting elements of the majority class
- Oversampling: Randomly replicating elements of the minority class
- SMOTE: Classical implementation of SMOTE to generate synthetic examples of the minority class

Since we are interested in testing the influence of sampling methods, we will fix the data preprocessing step and the learning algorithm, together with the evaluation criteria. Specifically, we will split the dataset into train and test using stratified sampling, keeping 75 % of the data for training, and the rest for testing. The data preprocessing will be a standardization, and the learning algorithm will be a k-NN with  $k \in [1, 3, 5, 7, 9, 11]$ . This parameter will be inferred using 5 fold cross validation with three repetitions (meaning that the whole procedure of CV will be repeated three times).

We begin by separating the dataset into train and test and checking that the imbalance ratio remains similar in both subsets, so the stratification is correct:

```
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] 0.2
```

```
imbalanceRatio(testData)
```

```
## [1] 0.192
```

Afterwards, we define two functions to perform the model training and testing respectively.

```
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)
}

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)

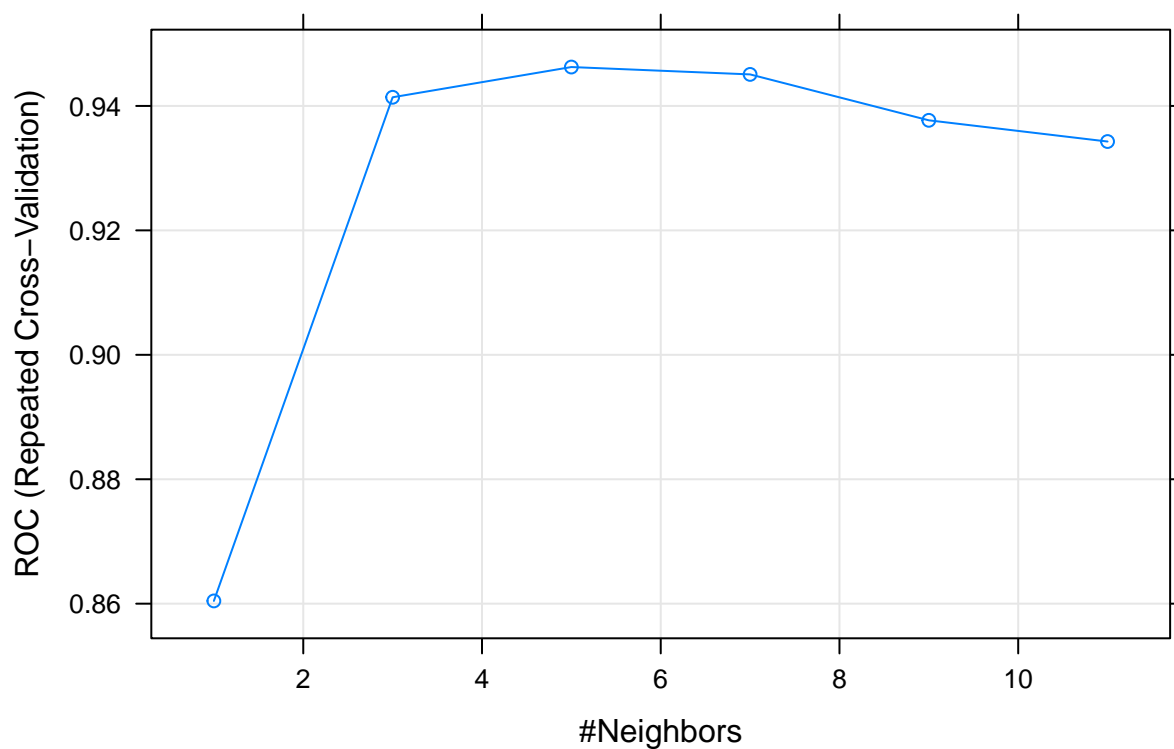
model.probs <- predict(model.fit,newdata = dataset, type="prob")
model.roc <- roc(dataset$Class,model.probs[, "positive"])
plot(model.roc, type="S", print.thres= 0.5,main=c("ROC Test",message),
      col="blue")

return(model.cm)
}
```

After both functions are defined, we will use them to train and test the defined pipelines. We begin with the raw data model:

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

## Grid Search RAW



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

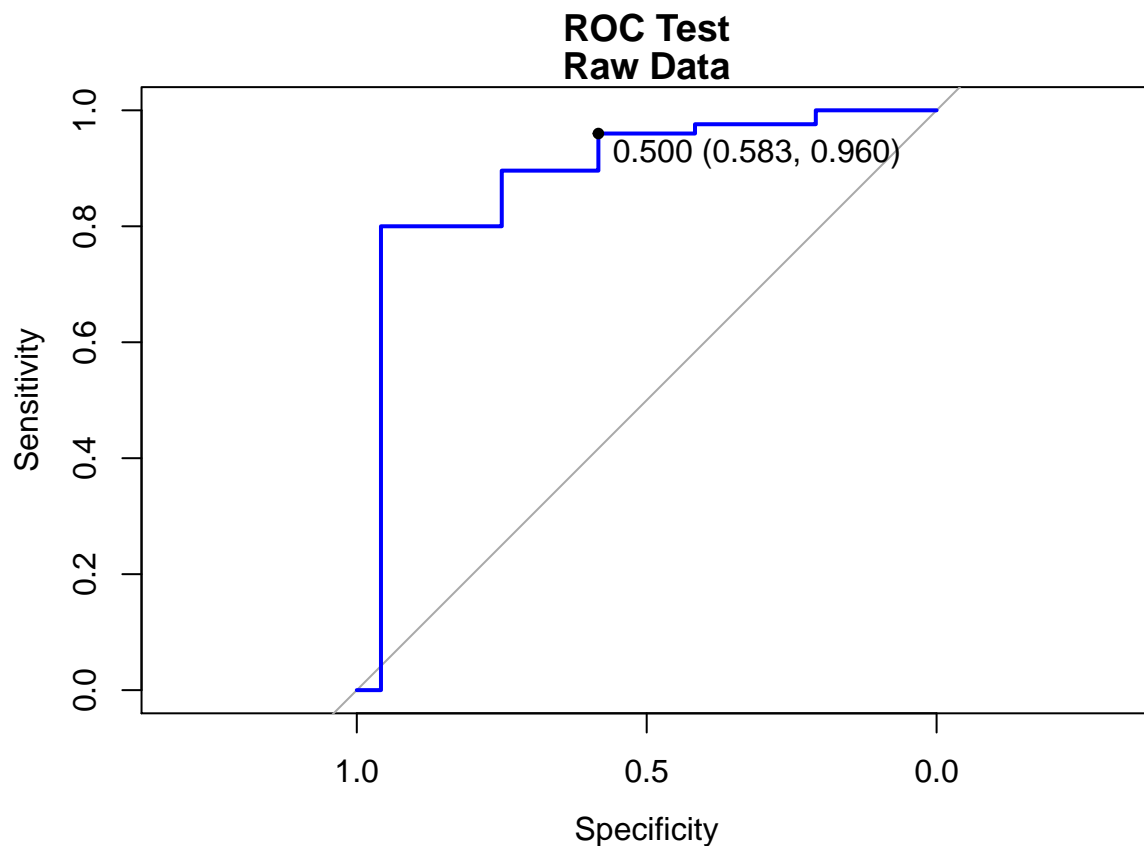
```
## k-Nearest Neighbors
##
## 450 samples
## 2 predictor
## 2 classes: 'positive', 'negative'
##
## Pre-processing: centered (2), scaled (2)
## Resampling: Cross-Validated (5 fold, repeated 3 times)
## Summary of sample sizes: 360, 360, 360, 360, 360, 360, ...
## Resampling results across tuning parameters:
##
##  k    ROC      Sens      Spec
##  1  0.8604444  0.7644444  0.9564444
##  3  0.9413926  0.6977778  0.9502222
##  5  0.9462519  0.6977778  0.9475556
##  7  0.9450667  0.6622222  0.9404444
##  9  0.9376889  0.6488889  0.9288889
## 11  0.9342815  0.6222222  0.9271111
##
## ROC was used to select the optimal model using the largest value.
## The final value used for the model was k = 5.
```

With the previous information (evolution of ROC in the graph and summary of the model), we can have an idea of the model performance). The specific value of  $k$  used in the inference phase is selected using one of the previous calculated metrics over cross validation. In this case, area under ROC for  $k = 5$  is the highest value, and then 5 is the selected  $k$ . It is interesting to point out that depending on the chosen metric, the election could have been different. For example, the model with the highest specificity is the obtained for  $k = 1$ , so we would have chosen  $k = 1$  if the important metric for our specific problem was that one.

After the model is trained, we can perform inference over our test set:

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

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction positive negative
## positive      14         5
## negative      10        120
##
##           Accuracy : 0.8993
##           95% CI : (0.8394, 0.9426)
##           No Information Rate : 0.8389
##           P-Value [Acc > NIR] : 0.02414
##
##           Kappa : 0.5933
##
## Mcnemar's Test P-Value : 0.30170
##
##           Sensitivity : 0.58333
##           Specificity : 0.96000
##           Pos Pred Value : 0.73684
##           Neg Pred Value : 0.92308
##           Prevalence : 0.16107
##           Detection Rate : 0.09396
##           Detection Prevalence : 0.12752
##           Balanced Accuracy : 0.77167
##
##           'Positive' Class : positive
##
```



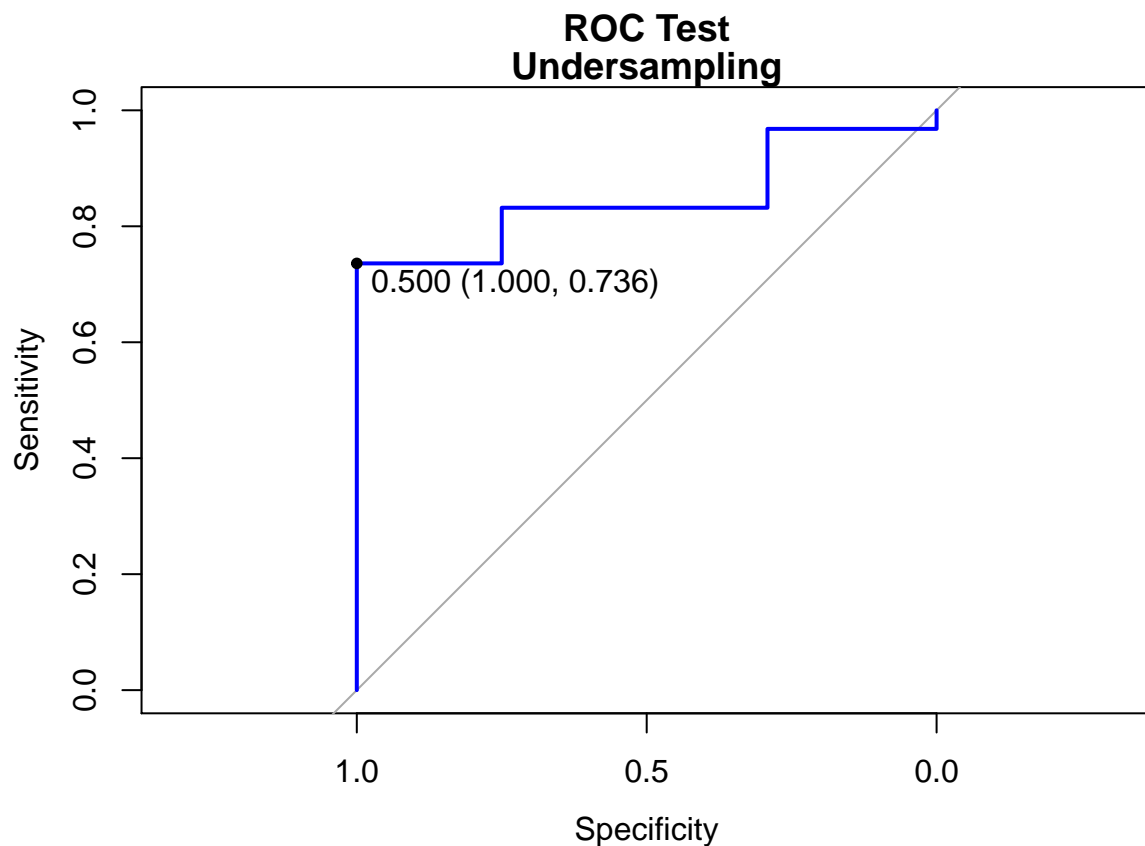
The output of the function shows different performance information related to the model. At first, we have the confusion matrix of the prediction. In it, we can observe an important number of false negatives (10, close to the 14 true positives detected), and some false positives also, but in a much more reduced number. After that matrix, some performance metrics are calculated (accuracy, sensitivity, specificity...), and plots de ROC curve.

For the following models, we won't comment the output since it will be the same for all of them. We will collect the results for the models and compare them afterwards. We perform next the evaluation over the random undersampling strategy.

```
## Undersampling
ctrl <- trainControl(method="repeatedcv", number=5, repeats=3,
                     classProbs=TRUE, summaryFunction = twoClassSummary,
                     sampling = "down")
model.us <- learn_model(trainData,ctrl,"Undersampling")
cm.us <- test_model(testData,model.us,"Undersampling")
```

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

```
##
##           Reference
## Prediction positive negative
## positive      24      33
## negative       0      92
##
##           Accuracy : 0.7785
##           95% CI : (0.7033, 0.8424)
##           No Information Rate : 0.8389
##           P-Value [Acc > NIR] : 0.9795
##
##           Kappa : 0.4732
##
## Mcnemar's Test P-Value : 2.54e-08
##
##           Sensitivity : 1.0000
##           Specificity : 0.7360
##           Pos Pred Value : 0.4211
##           Neg Pred Value : 1.0000
##           Prevalence : 0.1611
##           Detection Rate : 0.1611
##           Detection Prevalence : 0.3826
##           Balanced Accuracy : 0.8680
##
##           'Positive' Class : positive
##
```

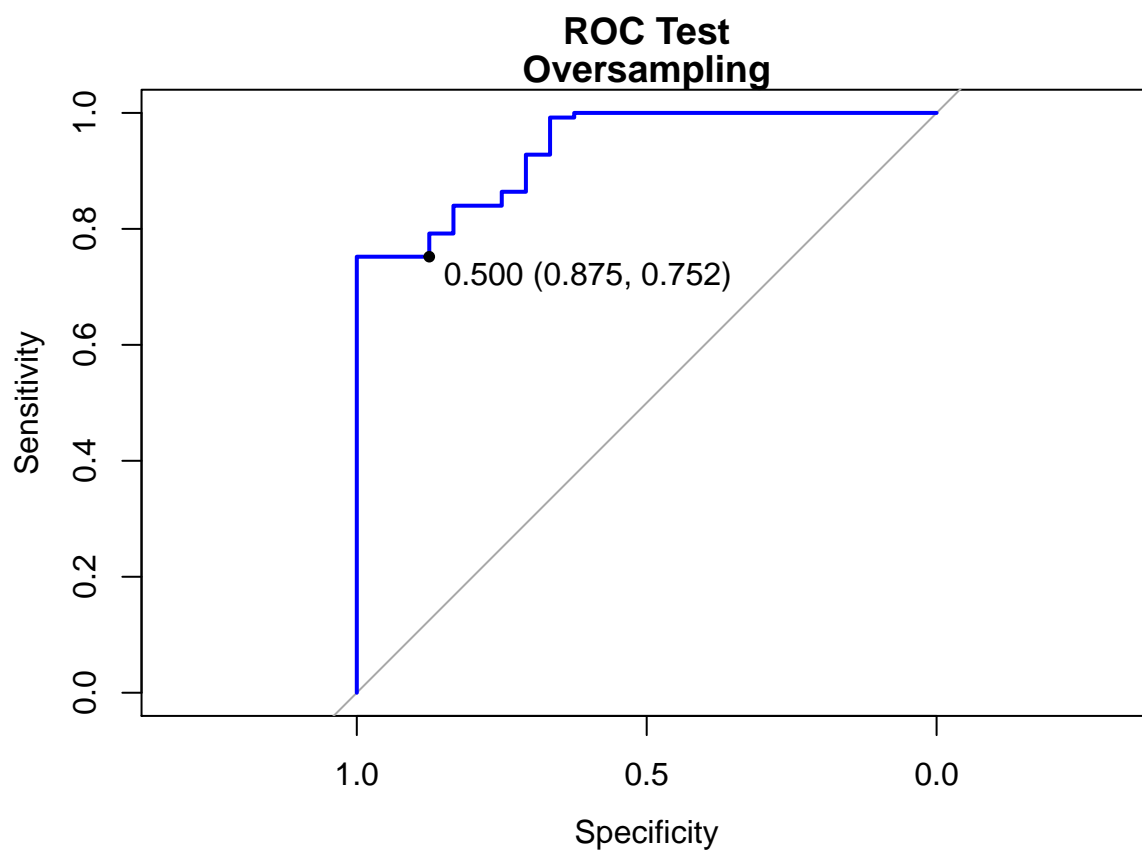


Then, oversampling politic:

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

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction positive negative
## positive      21      31
## negative       3      94
##
##           Accuracy : 0.7718
##           95% CI : (0.696, 0.8365)
##       No Information Rate : 0.8389
##       P-Value [Acc > NIR] : 0.9877
```

```
##
##           Kappa : 0.4261
##
## Mcnemar's Test P-Value : 3.649e-06
##
##           Sensitivity : 0.8750
##           Specificity : 0.7520
##           Pos Pred Value : 0.4038
##           Neg Pred Value : 0.9691
##           Prevalence : 0.1611
##           Detection Rate : 0.1409
##           Detection Prevalence : 0.3490
##           Balanced Accuracy : 0.8135
##
##           'Positive' Class : positive
##
```

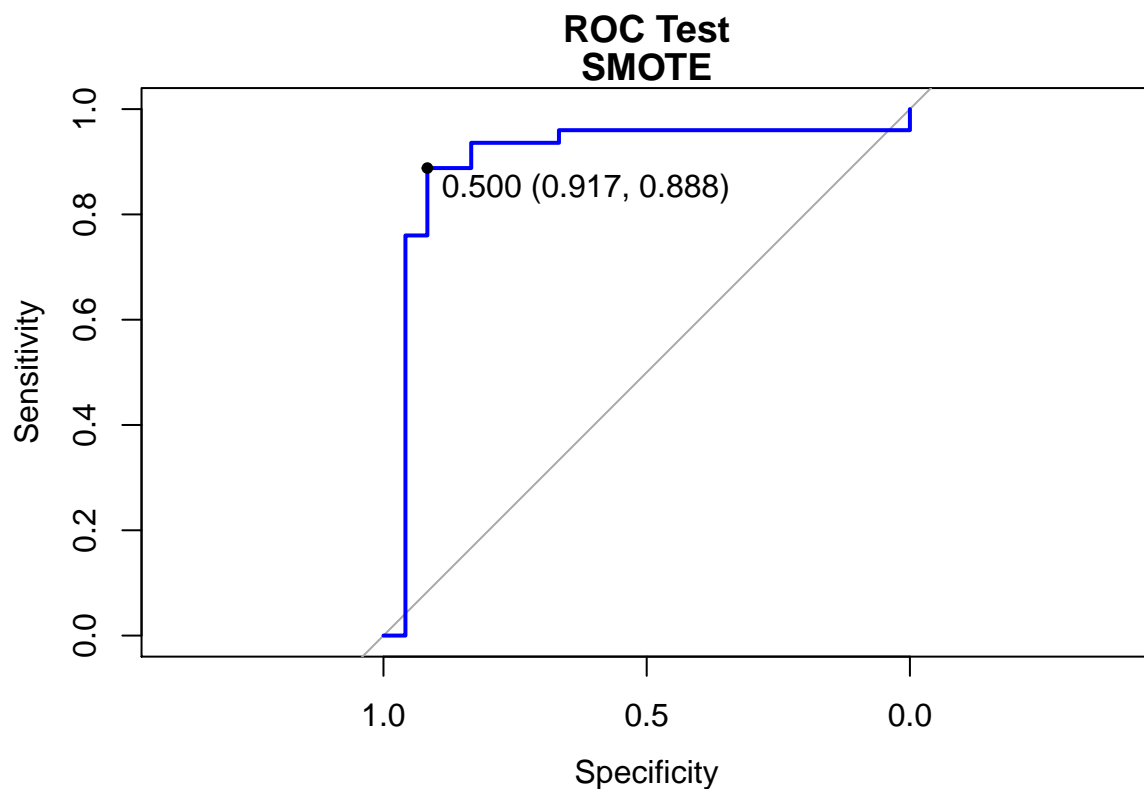


And finally, classic SMOTE:

```
## SMOTE
ctrl <- trainControl(method="repeatedcv",number=5,repeats = 3,
  ↪ classProbs=TRUE,
  summaryFunction = twoClassSummary,sampling = "smote")
model.smt <- learn_model(trainData,ctrl,"SMOTE")
cm.smt <- test_model(testData,model.smt,"SMOTE")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction positive negative
## positive      22      14
## negative       2     111
##
##           Accuracy : 0.8926
##           95% CI : (0.8315, 0.9374)
##      No Information Rate : 0.8389
##      P-Value [Acc > NIR] : 0.04215
##
##           Kappa : 0.6694
##
## Mcnemar's Test P-Value : 0.00596
##
##           Sensitivity : 0.9167
##           Specificity : 0.8880
##           Pos Pred Value : 0.6111
##           Neg Pred Value : 0.9823
##           Prevalence : 0.1611
##           Detection Rate : 0.1477
##           Detection Prevalence : 0.2416
##           Balanced Accuracy : 0.9023
##
##           'Positive' Class : positive
##
```





Now that we have performed training over all of the proposed methods, we will try to compare the obtained results in order to decide which model is the best. Using the function `resamples` from `caret` package, we can group and analyze the performance of the trained models:

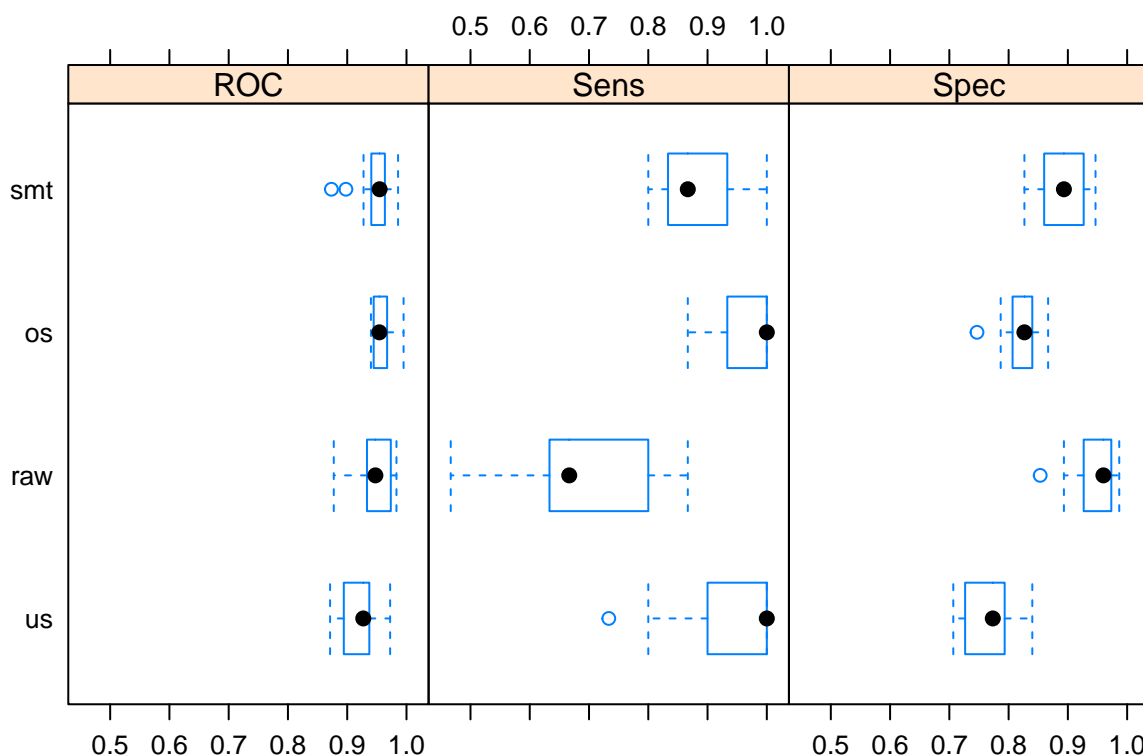
```
## summarize accuracy of models
models <- list(raw = model.raw, us = model.us,
               os = model.os, smt = model.smt)
results <- resamples(models)
summary(results)

##
## 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 0.8773333 0.9333333 0.9475556 0.9462519 0.9735556 0.9831111 0
## us  0.8711111 0.8942222 0.9271111 0.9188444 0.9373333 0.9724444 0
## os  0.9400000 0.9446667 0.9542222 0.9590815 0.9673333 0.9951111 0
```

```
## smt 0.8733333 0.9406667 0.9546667 0.9480593 0.9635556 0.9857778    0
##
## Sens
##      Min.   1st Qu.   Median     Mean   3rd Qu.     Max. NA's
## raw 0.4666667 0.6333333 0.6666667 0.6977778 0.8000000 0.8666667    0
## us  0.7333333 0.9000000 1.0000000 0.9333333 1.0000000 1.0000000    0
## os  0.8666667 0.9333333 1.0000000 0.9733333 1.0000000 1.0000000    0
## smt 0.8000000 0.8333333 0.8666667 0.8800000 0.9333333 1.0000000    0
##
## Spec
##      Min.   1st Qu.   Median     Mean   3rd Qu.     Max. NA's
## raw 0.8533333 0.9266667 0.9600000 0.9475556 0.9733333 0.9866667    0
## us  0.7066667 0.7266667 0.7733333 0.7688889 0.7933333 0.8400000    0
## os  0.7466667 0.8066667 0.8266667 0.8222222 0.8400000 0.8666667    0
## smt 0.8266667 0.8600000 0.8933333 0.8915556 0.9266667 0.9466667    0
```

By default, the stored information is ROC area, sensitivity and specificity of the models for all the cross validation executions (15 in total per model, since we performed 3 repetitions of 5-CV for each). We can also plot that information in a boxplot, so we get the information in a visual way:

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



For the moment, the information shown relates to the training set. It is more interesting to study the performance of the model in test set, since it will give us a more realistic view of the model performance over unseen data. The information of performance metrics is stored in a dataframe and represented in a plot:

```
## Carry out a comparison over all imbalanced metrics
comparison <- data.frame(model = names(models),
  BalancedAccuracy = 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$model == name, ] <- filter(comparison,
                                                    model == name) %>%
    mutate(BalancedAccuracy = cm_model$byClass["Balanced Accuracy"],
```

```

    Specificity = cm_model$byClass["Specificity"],
    Precision = cm_model$byClass["Precision"],
    Recall = cm_model$byClass["Recall"],
    F1 = cm_model$byClass["F1"])
}

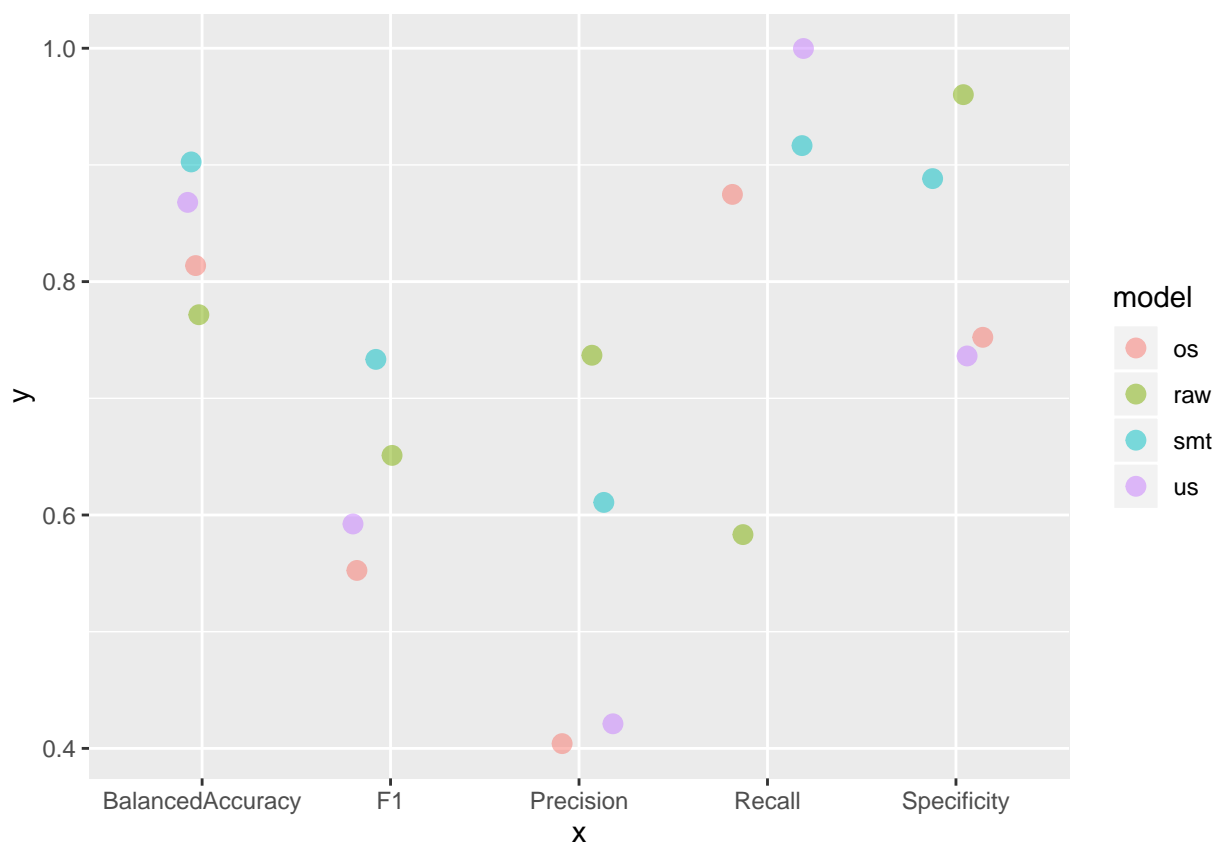
```

```
comparison %>%
```

```

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

```



In the previous plot, some performance metrics obtained by the models in the test set are shown. The most basic displayed metrics are recall and specificity. These metrics show the proportion of elements of each class that are correctly identified (recall for positive examples and specificity for negative ones). A high recall means that the algorithm is capable to detect elements in the positive class when they belong to it. A high specificity shows that the algorithm detects properly elements in the negative class.

Similar to this metrics, precision shows the proportion of elements classified as positive class that actually belong to that class. A high precision score shows that the model discriminates well when enclosing an example in the positive class, not committing much false positives.

Finally, we have the combined metrics. Those metrics are statistical summaries of the simple metrics before. The first one shown is the balanced accuracy. That metric is a correction of the classical accuracy metric that takes into account the imbalance present in the dataset. Instead of calculating the classic ratio of correctly classified examples by the total number of examples, the balanced accuracy is defined by the mean of specificity and recall. This correction improves the accuracy score by giving the same importance to both classes. If the dataset is balanced, both scores are the same. If not, the balanced accuracy penalizes the score stronger than the classical accuracy if the minority class is misclassified.

The other combined metric is  $F_1$  score. This metric calculates the harmonic mean between recall and specificity, instead of arithmetic mean, as balanced accuracy did. It is usually the most employed measure for imbalanced classification.

Now that we have commented the metrics we will use to analyze our models, we will comment the results obtained. For the raw data model, we can see that the specificity is the highest among all models. This is because k-NN overfits a lot in presence of imbalance. Since a lot of examples belong to the negative class, most of the neighbors of a given point will be of that class, and thus most of examples will be classified as such. Because of that, most of the negative examples in the test set will be correctly classified and the specificity is high. On the other hand, recall drops significantly for the same reason, and this model is the one with the worst score in this column.

The case of precision is curious. Despite the fact that not many positive examples are correctly identified, the number of examples incorrectly classified as positive is low, because the distribution of the data remains the same. This is not the case of the other models, which suffer from the modification in the data distribution, provoking an augment of false negatives, and thus an important drop in the precision. Talking about the composed metrics, due to the bad performance in terms of recall when we train with raw data, the model is the worst when balanced accuracy is measured.  $F_1$  score, however, is not too low, being the second model in this case.

Now, for the random undersampling model, we can observe an important drop in specificity. This is because the boundaries of the majority data are modified when random undersampling is performed, and because of that a lot of points in that zone are incorrectly classified as belonging to the minority class. Recall, on the other hand, is greatly improved, because the zone occupied by the positive class gets cleaned of negative points, making the classification of points of this class much easier. Nevertheless, precision is too bad, because a lot of points from the negative class are misclassified. Due to the bad performance in specificity, combined metrics are poor, specially  $F_1$ . Balanced accuracy is fairly good, because of the high recall value.

For the random oversampling model we obtain the worst results. Specificity is a bit better than the one obtained by undersampling, but in return the recall drops significantly. Also, the precision is the worst one, and the combined metrics are bad in both cases. The problem here lies in the learning model. Neighboring based algorithms are not much affected by oversampling, because the only effect the replication has is replicating the same neighbor for some points. Points that aren't related with a multiplied point before oversampling are unaffected afterwards, and then the result is not specially good.

SMOTE, by contrast, improves the results significantly. Comparing it to the raw model, specificity is a bit lower, but the vast improvement in recall measure justifies the loss. Not losing 10 percent points in specificity, recall rises more than 30 points. Precision score is reduced, but not to the extent of undersampling or oversampling. Finally, the improvement in basic measures produces that SMOTE is the best algorithm in both combined metrics, and then we can conclude that it greatly improves the results in our dataset.

In the next section, we will further study some extensions to SMOTE algorithm that overcome some of the problem produced by the random sample of the original points.

### 3 SMOTE variants using the `imbalace` library

In this section we will explore some of the variants of the SMOTE algorithm. This extensions try to fix some of the problems that present the original algorithm, produced by the random selection of the points, mostly.

In order to perform the comparison, we will encapsulate the pipeline in a function that receives the dataset, the target imbalance ratio, and a list of methods, performs stratified train-test sampling, and then for each specified method performs data augmentation, trains a kNN, and predicts the test set. Afterwards, the comparison dataframe is constructed with the metrics we used in the previous section. Also, classification using the raw data is given as a baseline. The function is the following:

```
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  
  ctrl <- trainControl(method="repeatedcv", number=5, repeats = 3,
```

```

        classProbs=TRUE, summaryFunction =
        ↪ twoClassSummary)
basic.model <- learn_model(trainData, ctrl, "RAW")
basic.model <- test_model(testData, basic.model, "RAW")

## Model training with each data generation politic
cm.models <- sapply(methods, function (x) {
  aug.trainData <- oversample(trainData, ratio=imb.ratio, method=x, )
  ctrl <- trainControl(method="repeatedcv", number=5, repeats = 3,
    classProbs=TRUE, summaryFunction =
    ↪ twoClassSummary)
  model <- learn_model(aug.trainData, 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
}

```

Now that our function is built, model comparison is easy. We will compare the performance of SMOTE with other three variants:

- BLSMOTE (Borderline SMOTE): One of the earliest modifications of SMOTE, which only creates examples using the minority class examples close to the class boundary.
- DBSMOTE (Density-Based SMOTE): This method combines the DBSCAN clustering algorithm with the SMOTE algorithm, by creating instances in the segment joining the original data with the centroid of the clusters discovered in the minority class with DBSCAN.
- MWMOTE (Majority Weighted Minority Oversample): This modification generates examples by assigning weights to the minority examples by the difficulty to classify them (by checking its distance to the majority class examples). Afterwards, it generates synthetic data using a clustering approach to select close original points.

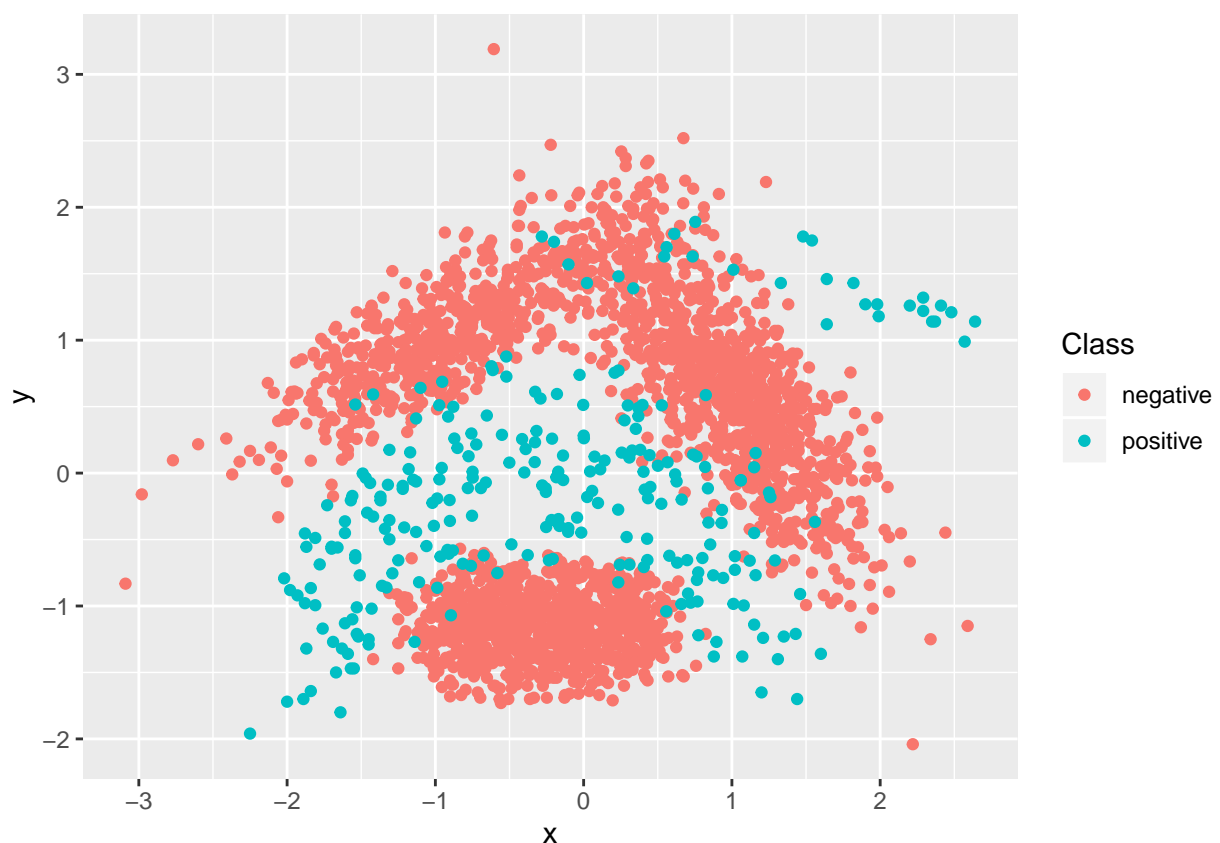
Also, the results with the raw data will be reported as baseline. We will test the algorithms over different dataset, in order to highlight its differences.

At first, we will perform the comparison over the banana dataset. The distributions of points in the dataset set is the following:

```
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))
```



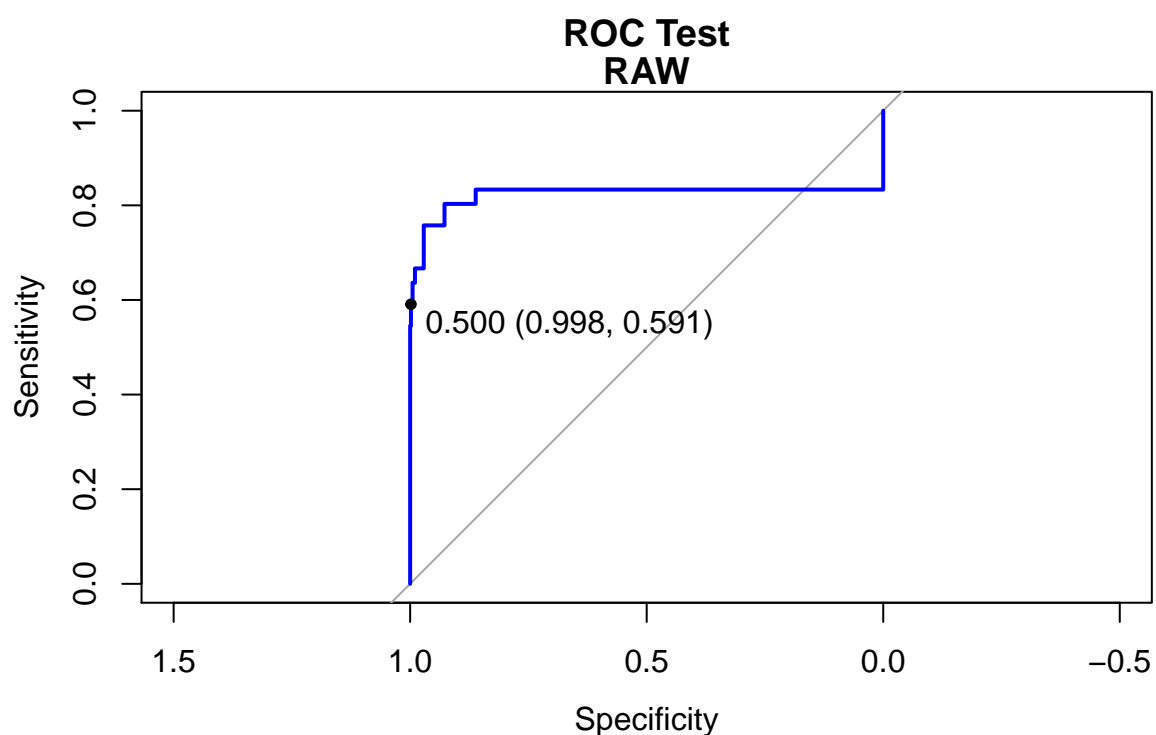
Its name comes from the shape of the minority class, which resembles a banana in its origin. Actually, this version of the dataset is an imbalanced one, not the original. As we can observe, the minority class

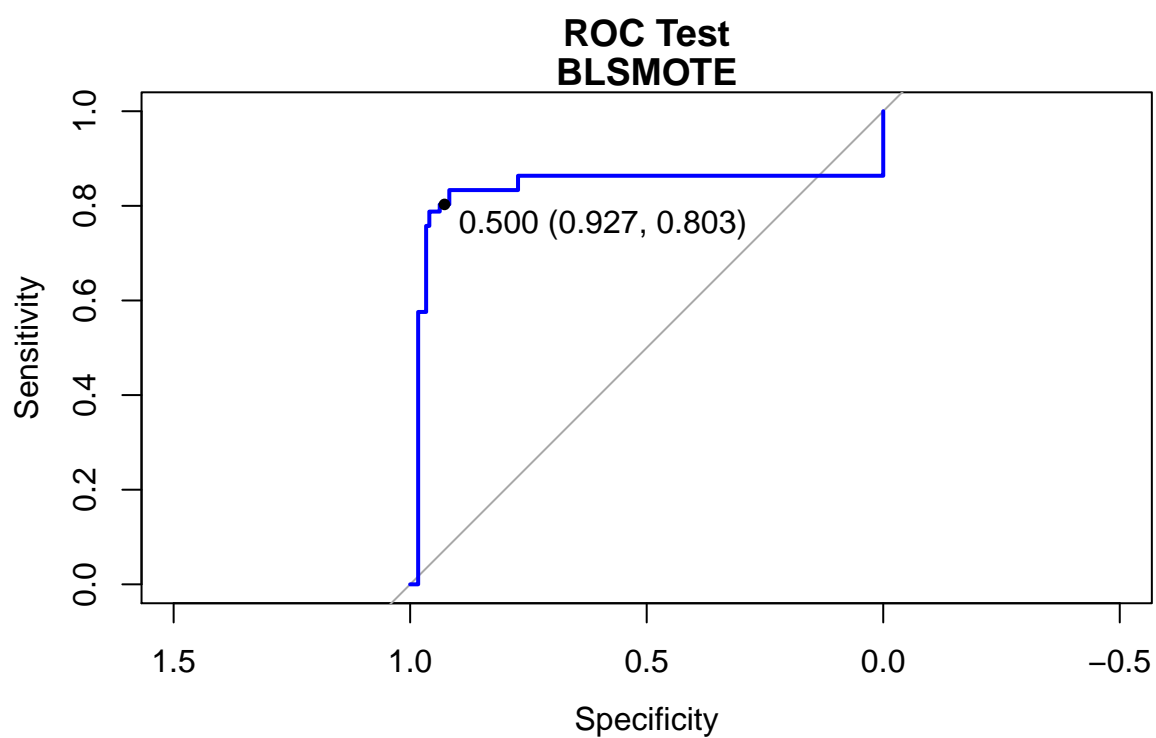
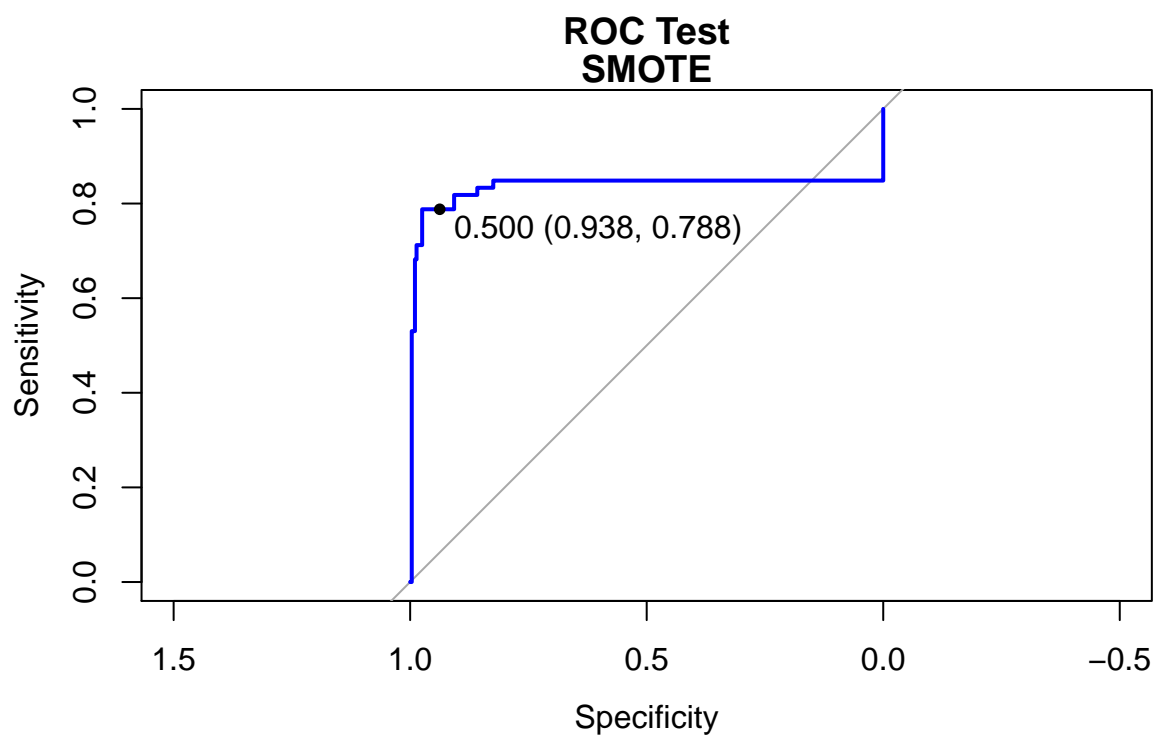


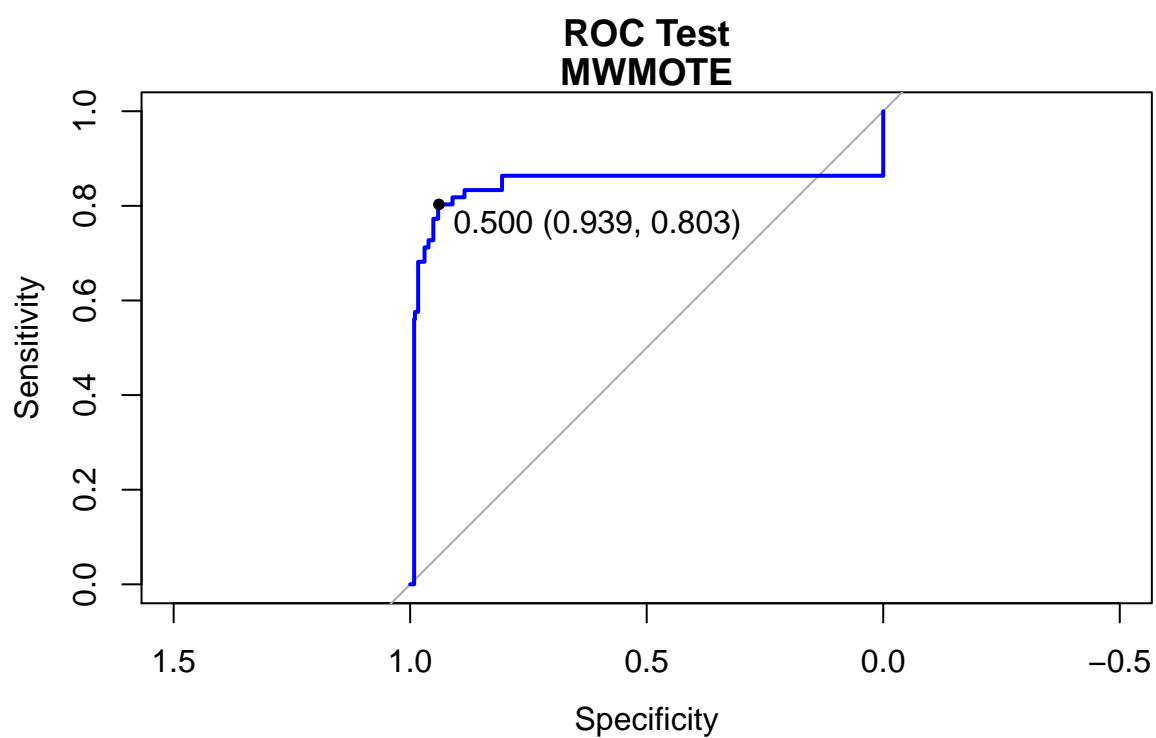
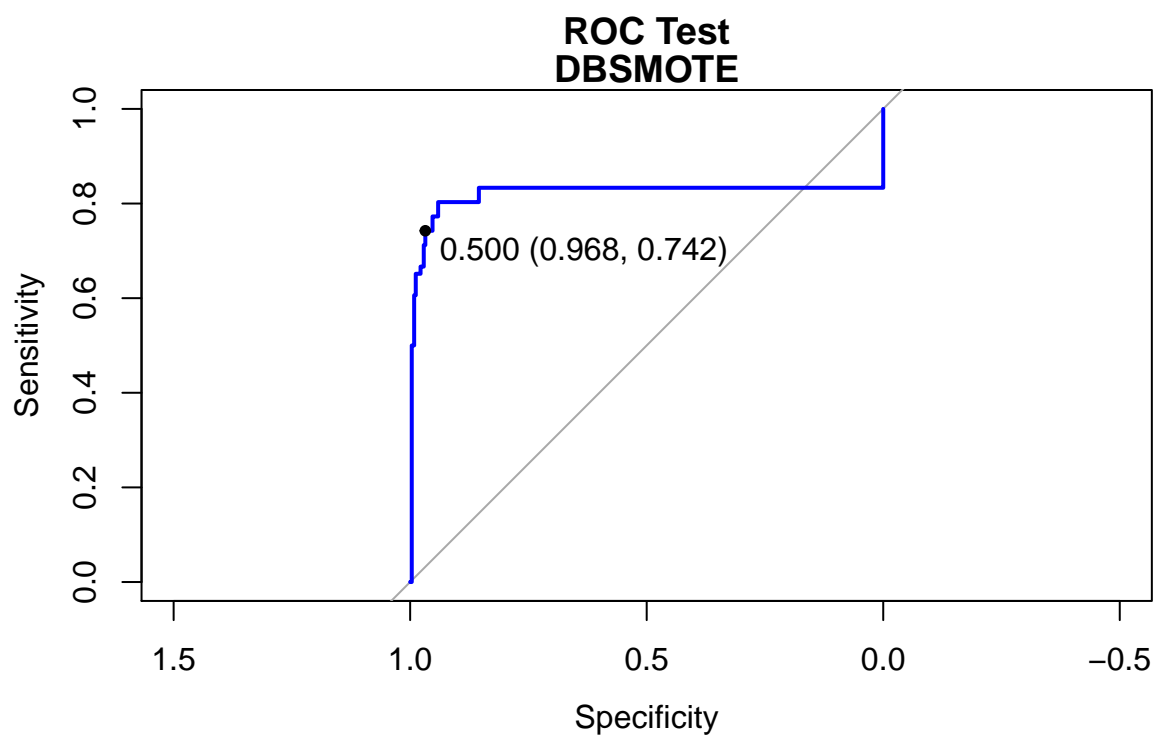
is placed between two big clusters of the majority one, and a small group of points in the top part. We have to be careful with our synthetic instances generation, in order to avoid creating instances inside the majority cluster at the top. If our algorithm chooses the original points completely at random, the joining segment between them will likely lie inside that cluster, and then the synthetic data will pollute the boundaries of our classes, undermining the performance.

As we have stated, we will now execute the models commented before over the banana dataset. Code and output is not shown because some of the methods employed print a lot of undesired information. Only the resulting ROC curve will be shown.

```
imb.ratio <- 0.65  
methods <- c("SMOTE", "BLSMOTE", "DBSMOTE", "MWMOTE")  
comparison <- perform.comparison.smote(dataset, imb.ratio, methods)
```

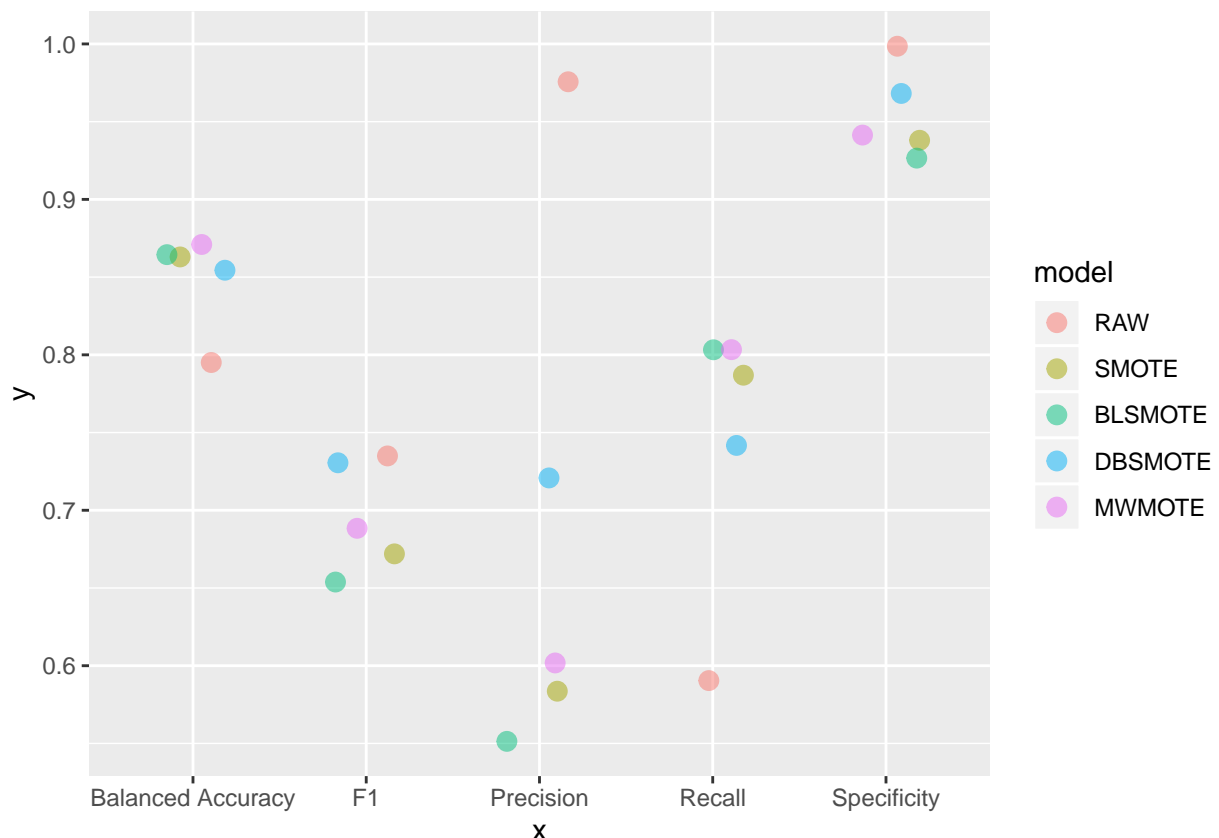






The results obtained by the models are the following:

```
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)
```

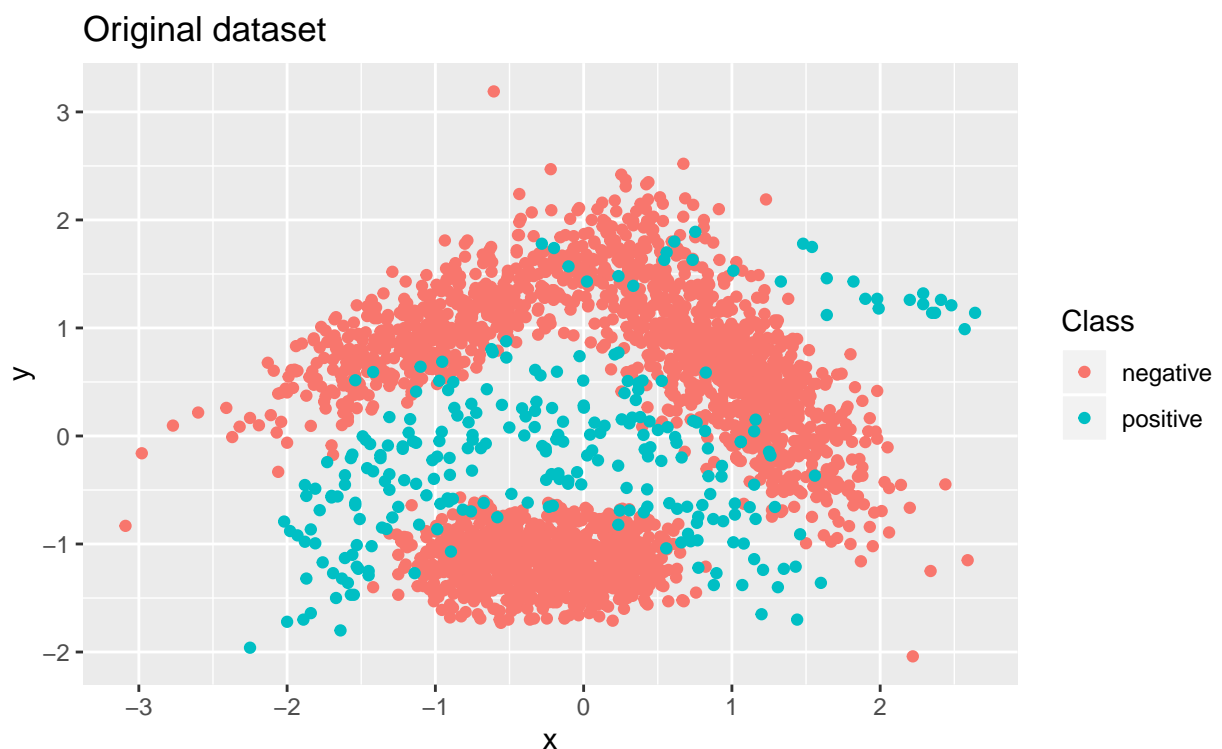


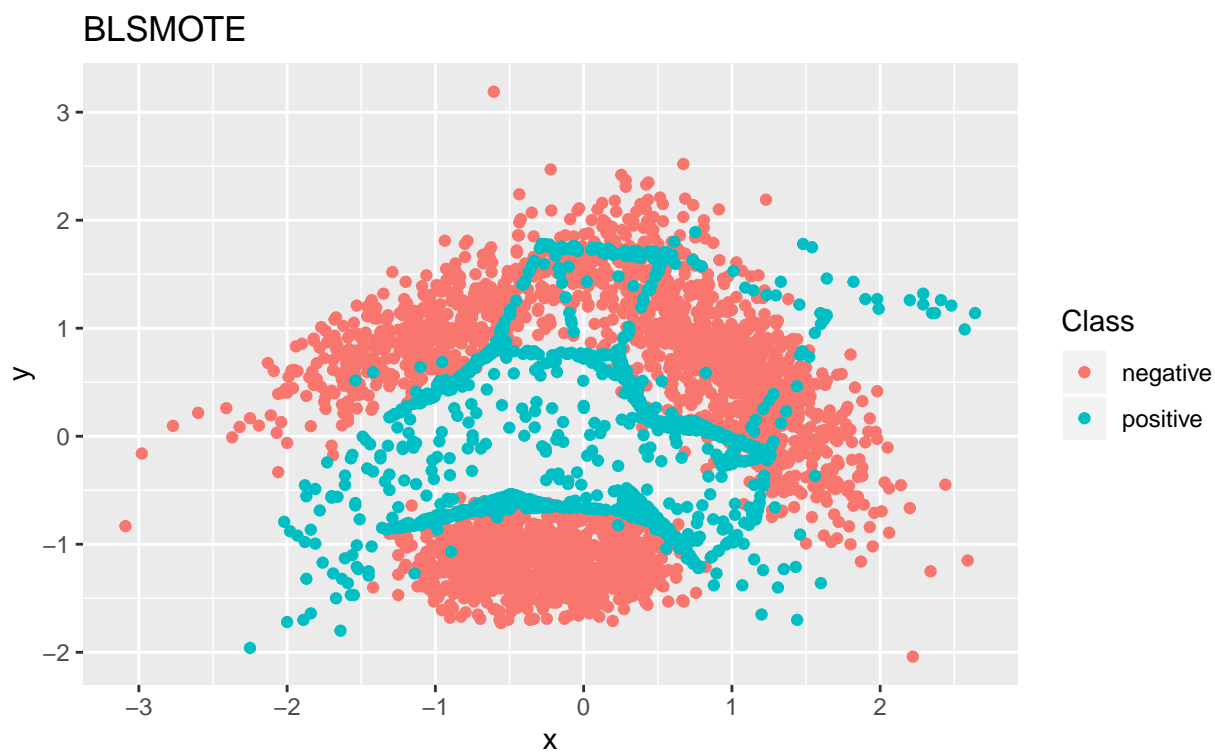
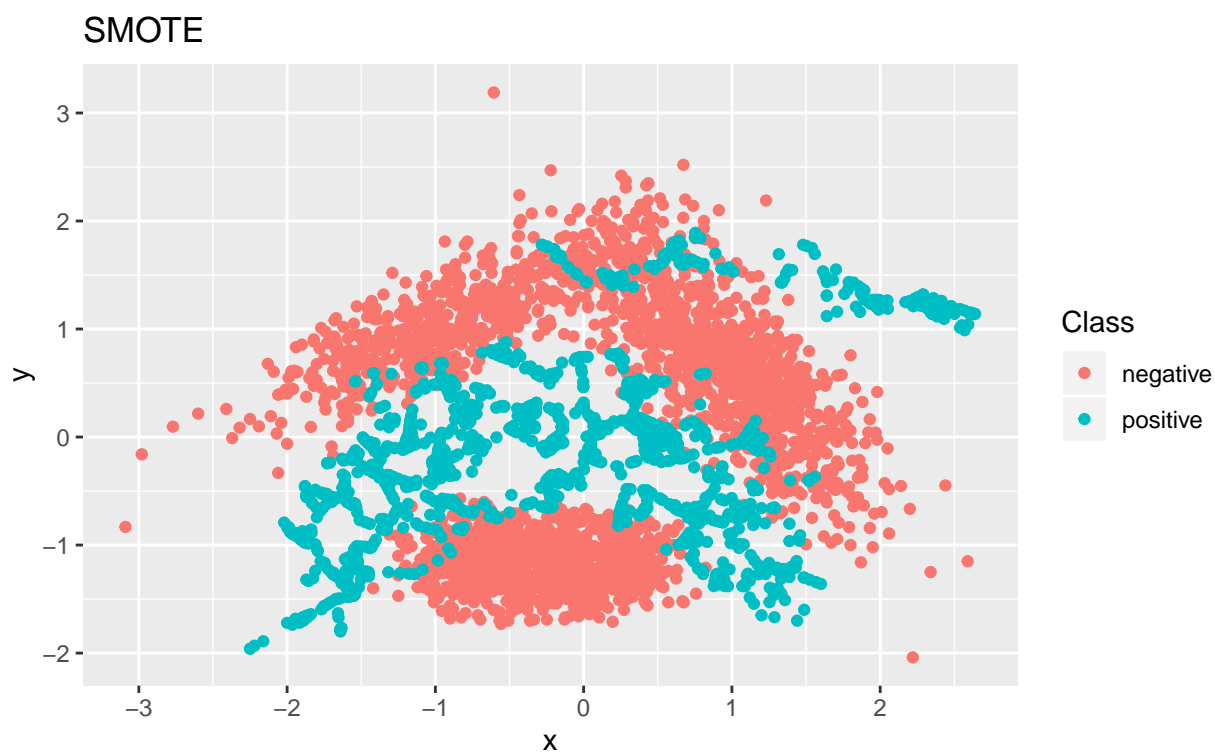
We can now analyze the results over our dataset. The first thing we can observe is that the behavior of the raw data model is similar to the one we observed in the previous section. Specificity of the model is great, but recall is poor. This is due to the overfitting in favor of the majority class (negative one). Precision is good because the number of false positives is low, since data boundaries haven't been modified and there is not much overlap. Talking about the combined metrics, the good result in specificity makes the  $F_1$  score to be the highest among all models. Balanced accuracy, on the other hand, is the lowest, due to the bad result in recall. This is one of the problems of  $F_1$  score. This metric tends to punish less the difference between class metrics than balanced accuracy does.

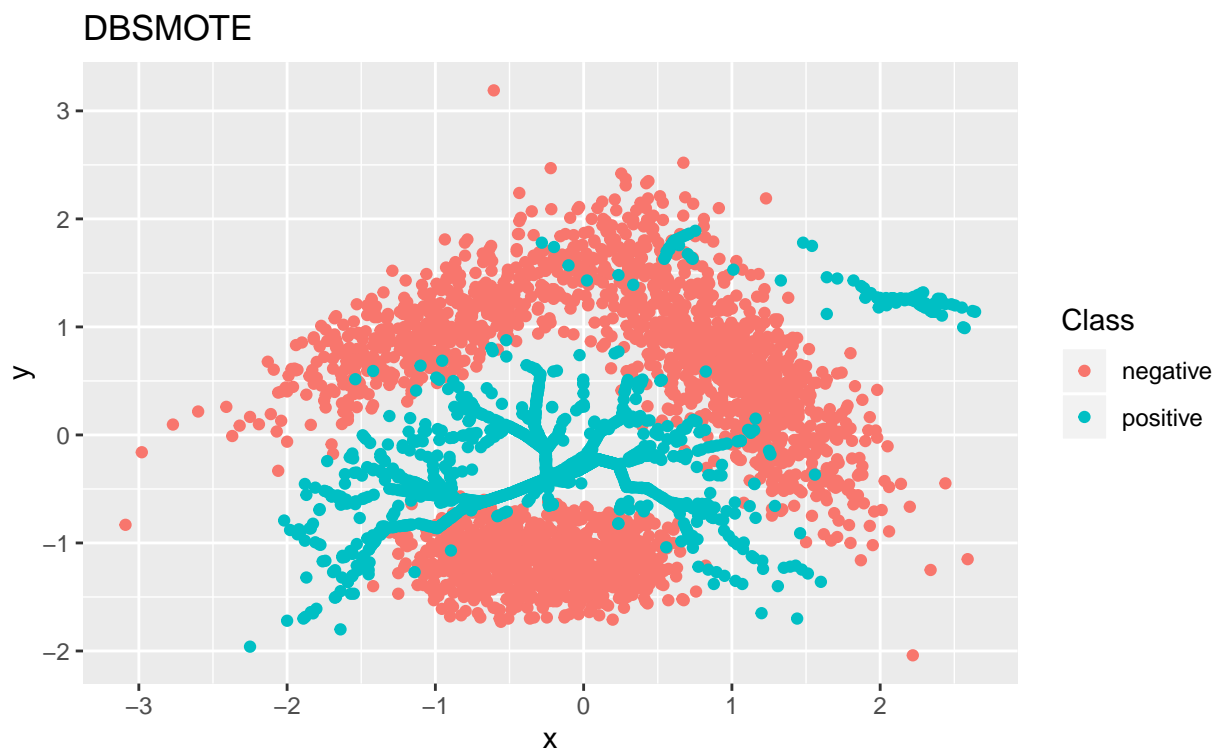
Talking about the oversampling methods, we can see that the behavior of all of them is very similar. The most different one is DBSMOTE, which produces a slightly better specificity, but losing a bit of

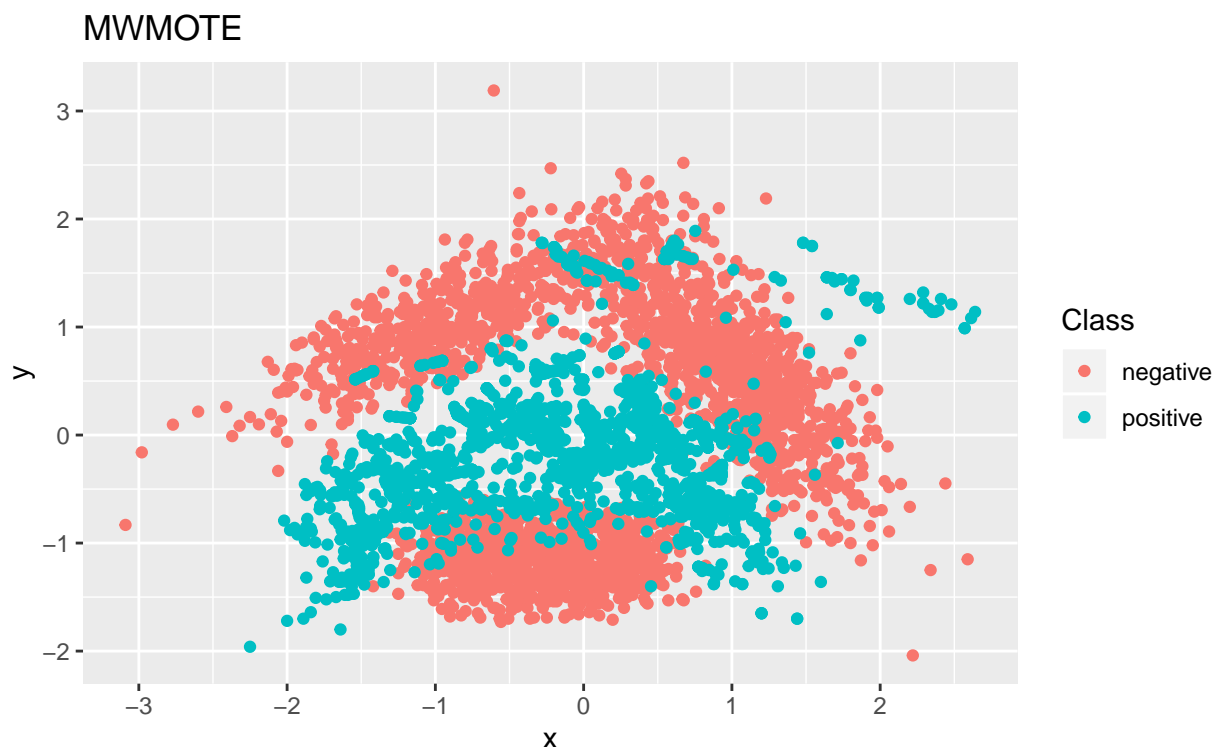
recall respect to the other models. Precision is much better for this model, however. In terms of combined metrics, DBSCAN scores a similar balanced accuracy as the other models, but due to the small improvement in specificity and the loss in recall,  $F_1$  for this model is slightly superior, close to the one of the raw data model.

Now, we are going to visualize the data generated by these methods, in order to infer the obtained results from the data.









As we can observe from the synthetic datasets, DBSMOTE is the one that produces less overlap between classes. If we pay attention to the points of the minority class at the center of the top majority cluster, that zone is strongly polluted by other methods. Specially, BLSMOTE, which generates points using borderline examples, pollutes a lot the top majority cluster. This fact justifies its scores in recall and specificity also. When data is generated using this method, the classifier will improve its classification rate in the minority class, since BLSMOTE will generate synthetic data near the complex examples, but the performance in majority class will be heavily affected due to class overlapping. This is why BLSMOTE is the algorithm with worst specificity, best recall, and worst precision (a lot of false positives occur).

It is also interesting to point out the patterns in DBSMOTE. It is visually clear that the examples are generated in the segment that joins examples and centroids. The main centroid of the algorithm is the point at the bottom center of the minority class, and the synthetic data resemble a tree structure with that point as root.

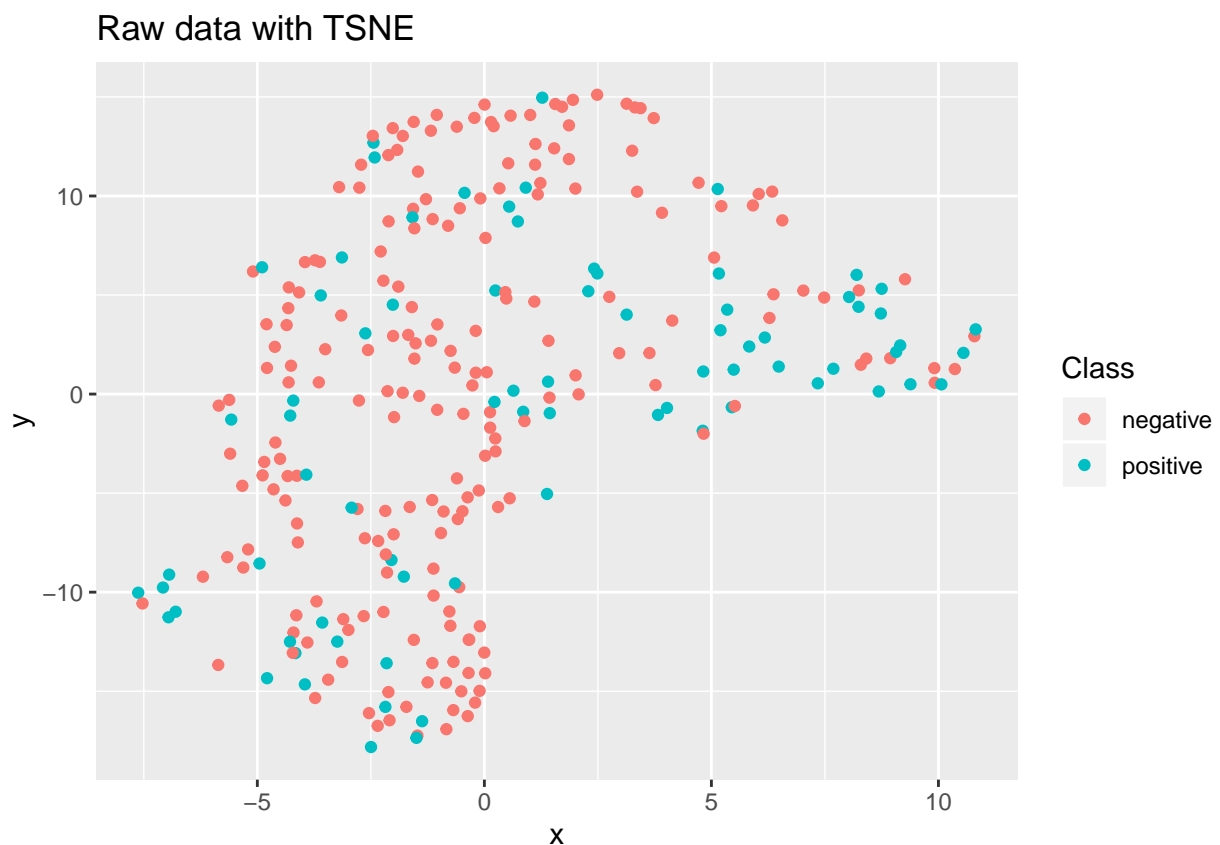
We will test our oversampling politics over other datasets. The next explored dataset is haberman. Since there are three variables, we cannot represent the information in a 2D graph. We will perform TSNE over the data previously in order to obtain a 2D representation of the dataset. After TSNE, the obtained dataset is the following:



```
dataset <- haberman
dataset <- unique(dataset)
dataset[, -length(dataset)] <- sapply(dataset[, -length(dataset)],
  ↪ as.numeric)

tsne.suitable.ind <- !duplicated(dataset[, -length(dataset)])
repr.data <- as.data.frame(Rtsne(
  dataset[tsne.suitable.ind, -length(dataset)]))$Y
)
colnames(repr.data) <- c("x", "y")
repr.data$Class <- dataset$Class[tsne.suitable.ind]

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



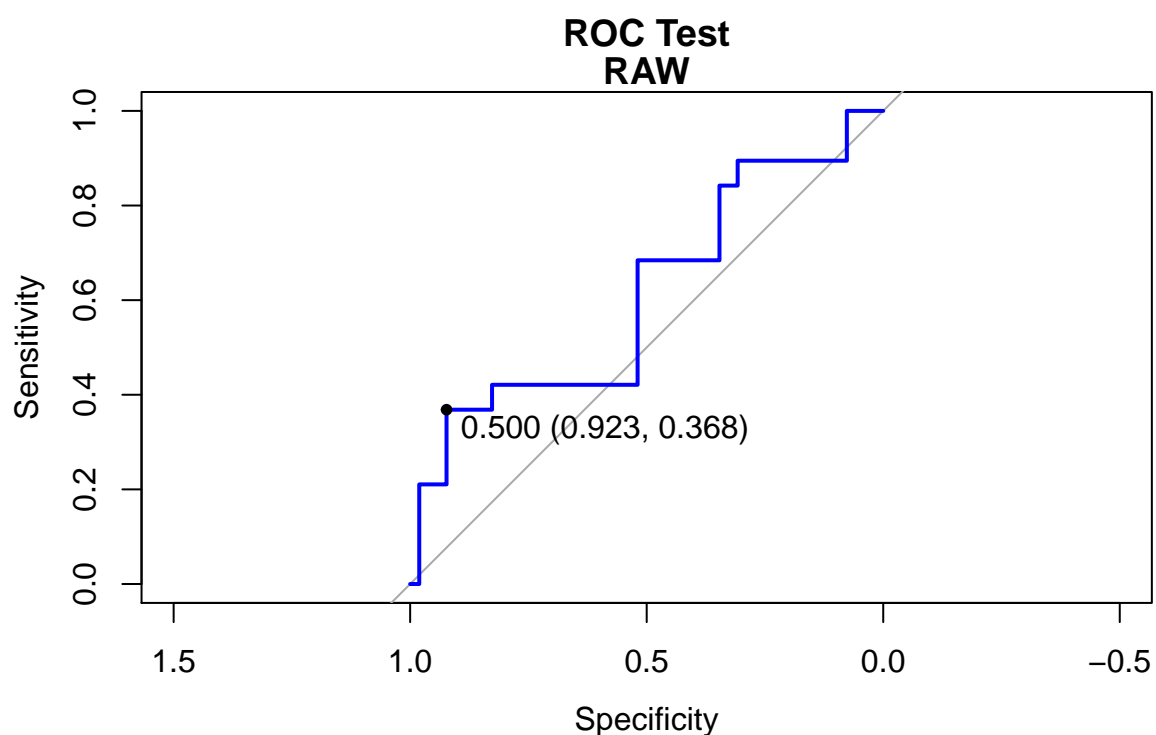
As we can observe in this dataset, the problem is not the scarcity of data but the dispersion of it.

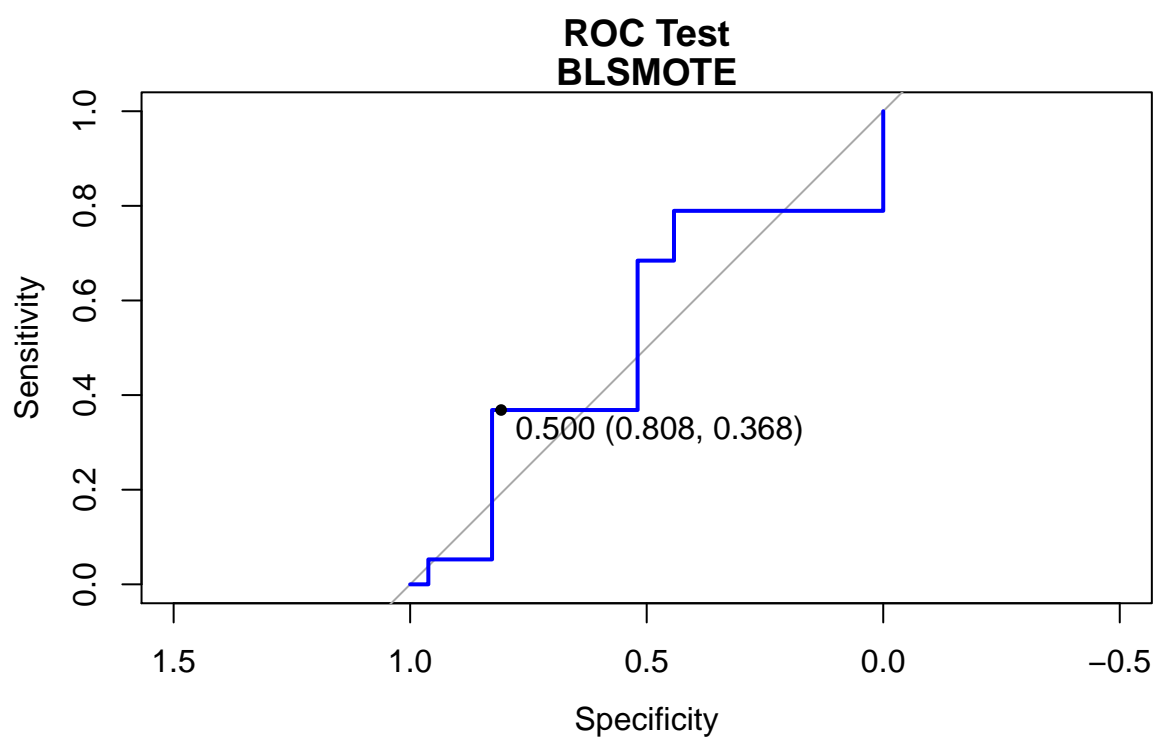
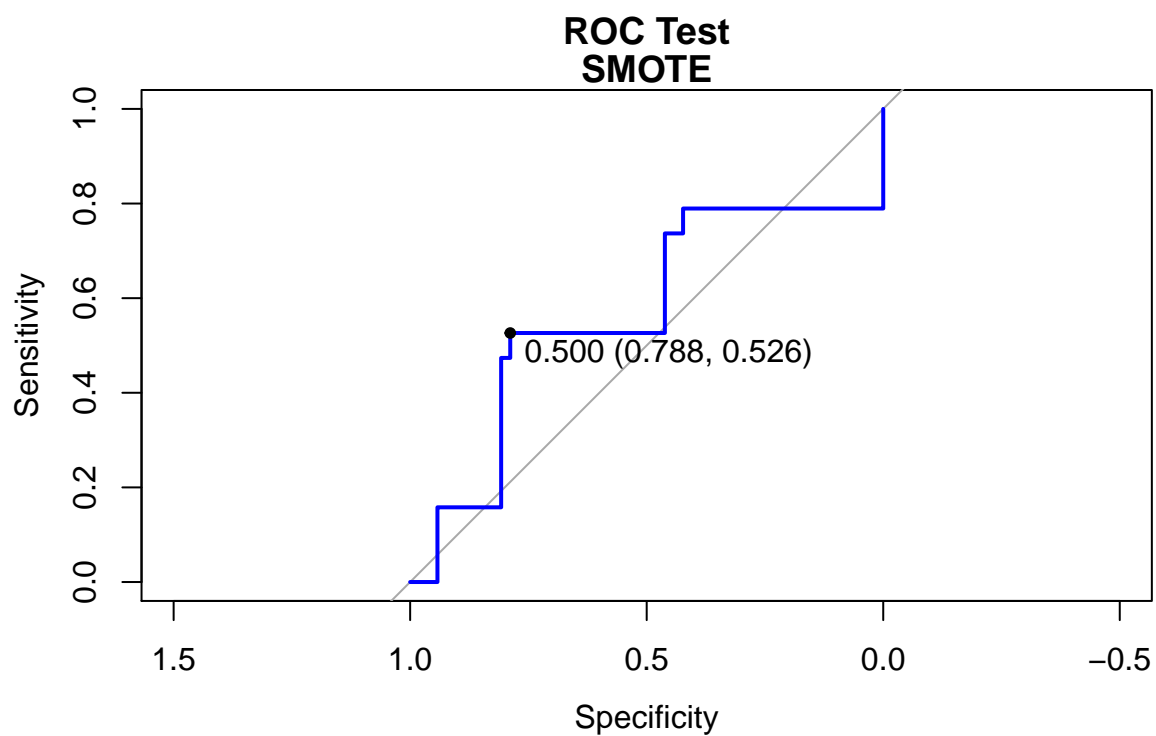
```
imbalanceRatio(dataset)
```

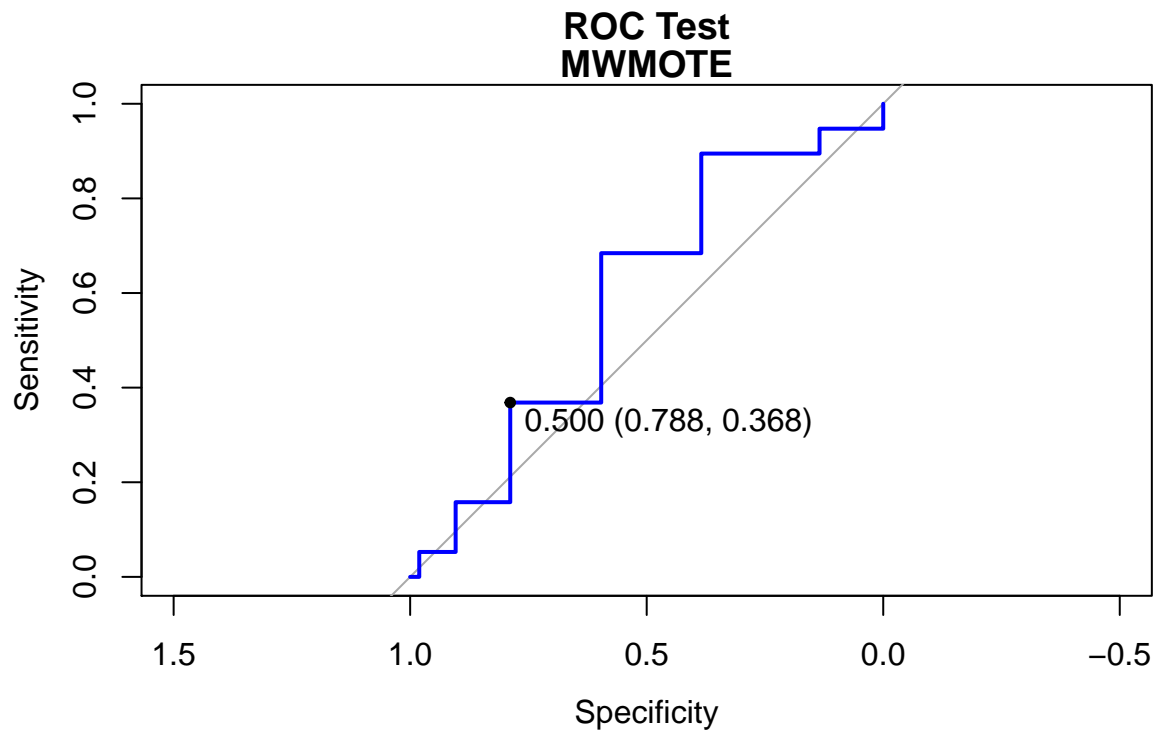
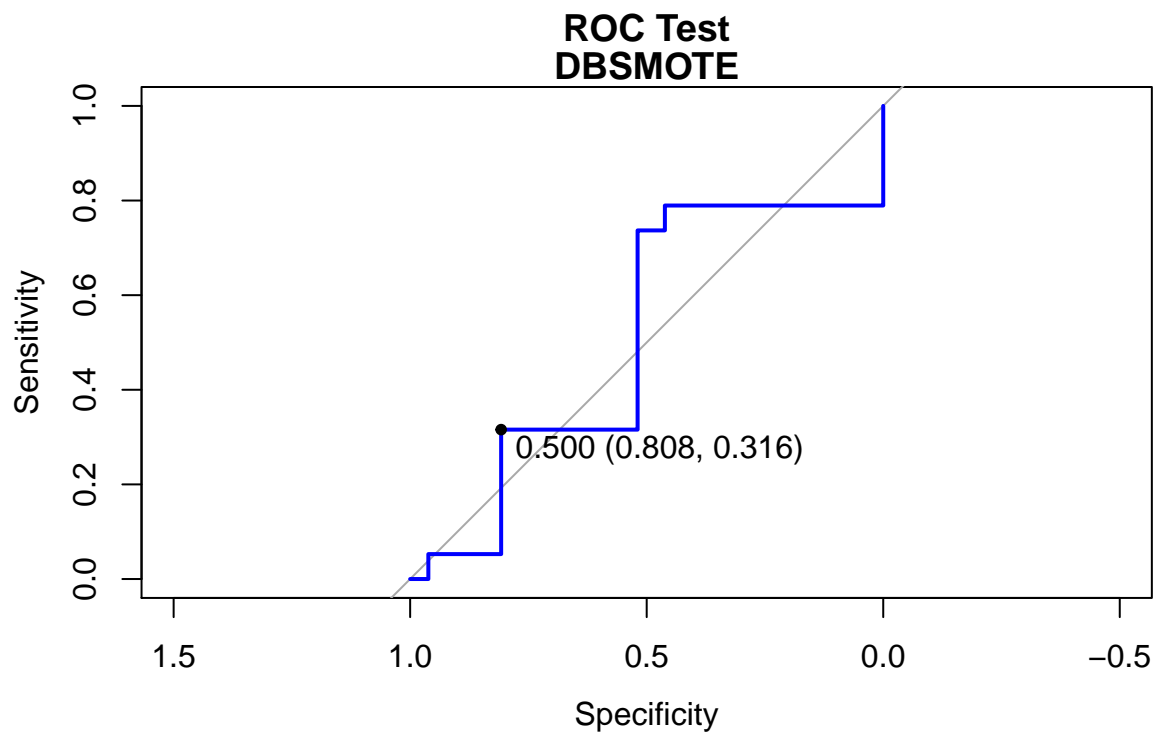
```
## [1] 0.3761905
```

In fact, the imbalance ratio is close to the double of the first dataset we used. In banana, the imbalance ratio is 0.11, so the problem here is less relevant. As we have done before, we will train five different models over this dataset, one with the raw data, and four with the different SMOTE modifications we are testing. The obtained results are the following:

```
imb.ratio <- 0.6  
methods <- c("SMOTE", "BLSMOTE", "DBSMOTE", "MWMOTE")  
comparison <- perform.comparison.smote(dataset, imb.ratio, methods)
```







```
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)
```

