Learning Binary Classification by Simulations

There are numerous aspects of data science projects from posing the right questions, through identifying and preparing relevant data to applying suitable analytical techniques and finally validating the results. This article focuses on the importance of analytical techniques and demonstrates how much they differ using examples of binary classification and my simulation package *mlsim* written in R.

One of the most used and abused methods of binary classification is logistic regression. It is applied quite frequently in many areas ranging from medical research to engineering in belief that it can discern a variety of data patterns. Many users of binary logistic regression not trained in Statistics or Machine Learning are not aware that the surface that separates classes of interest obtained in the process of estimating parameters is in fact a hyper-plane and as such will in many cases poorly approximate the boundary between the classes for many non-linear problems and result in high classification error rates. The initial code in what was to become *mlsim* was just a few examples showing how badly logistic regression and other linear methods such as SVM with linear kernel can fail if applied to data with certain non-linear boundaries.

As a data scientist I often have to explain in simple terms how different machine learning algorithms work which can be quite challenging especially when dealing with non-technical audience. In order to make this task easier I created the R package *mlsim* for simulating different data patterns and demonstrating the ability of several Machine Learning algorithms to fit to the data.

The package can also be a useful tool for data scientists who try to build spatial intuitions behind different Machine Learning algorithms. It can be downloaded from my github repository https://github.com/jacekko/mlsim\_zip/blob/master/mlsim\_0.0-1.zip.

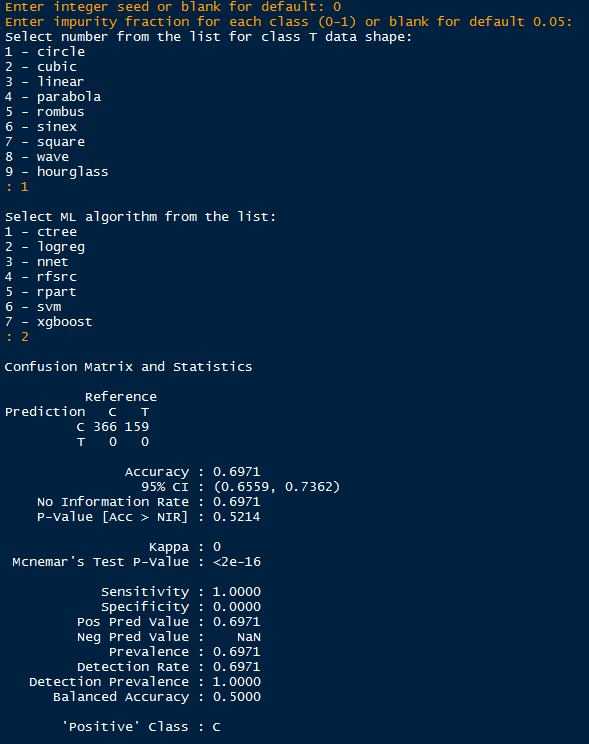
To install *mlsim* place the R binary build file *mlsim\_0.0-1.zip* in your working R directory and run

*install.packages("mlsim\_0.0-1.zip", repos=NULL)*

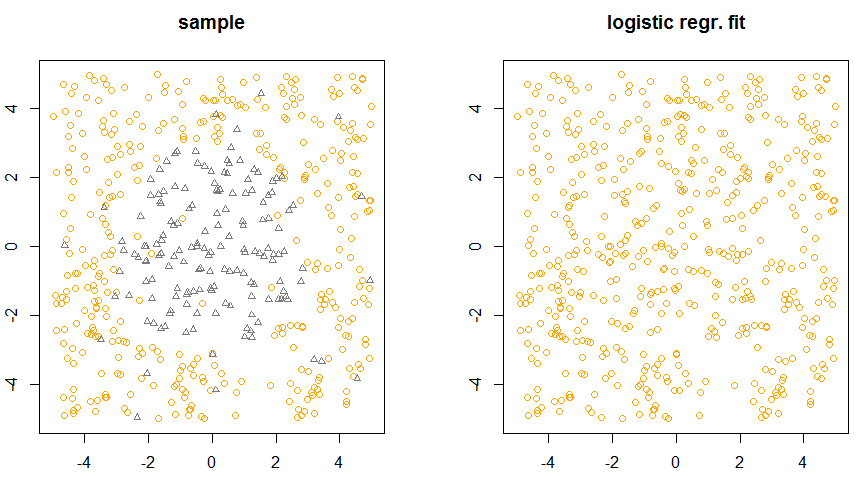
After loading the package with *library(mlsim)* you can run it by just typing *mlsim()* and following the prompts. It has been written in such a way that rather than downloading all the Machine Learning dependencies it prompts the user to do it. If the user decides to skip some libraries the corresponding options will not appear on the list of algorithms available for demonstration.

The program randomly generates data points marked as little triangles and circles representing two classes T and C separated by boundaries of different shapes such as circle, square, sinusoid etc. (a circle or square may not exactly be such shapes depending on the aspect ratio of you plotting area).

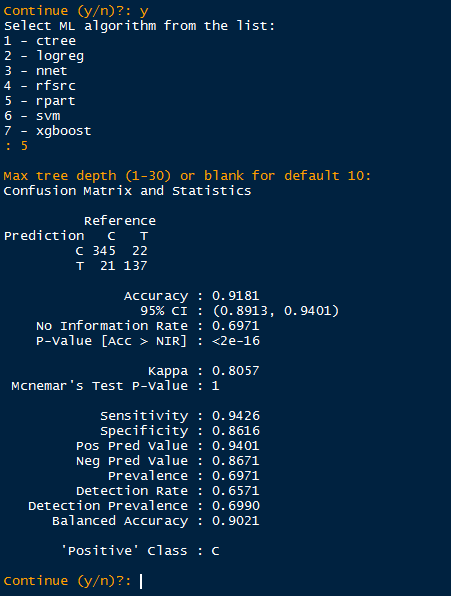
The program flow is shown below.

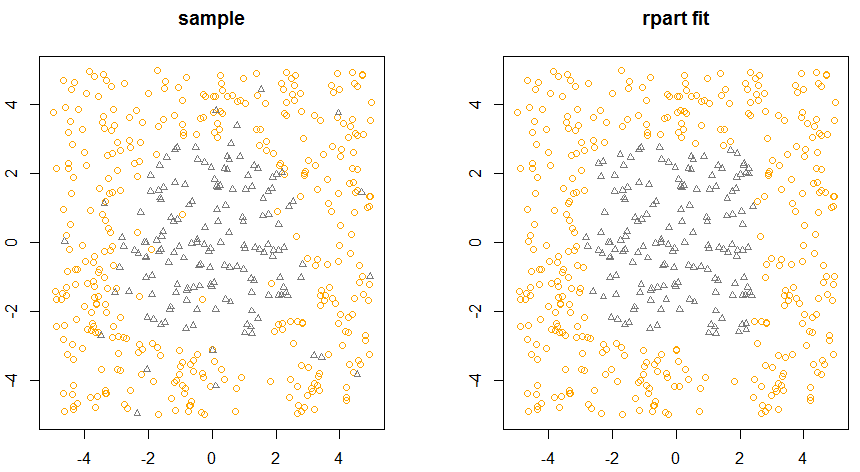


After the selected algorithm is executed the diagnostic information generated by *ConfusionMatrix* from package *caret* is printed and graphical output consisting of two plots presented as in the figure below. The plot on the left-hand side presents the randomly generated data and the one on the right-hand side presents the result of data fit. As we can see in this case logistic regression completely fails with separating the classes classifying everything as C i.e. circles.



We can then try another algorithm such as e.g. the recursive partitioning tree (rpart)





If you would like to create your own class boundary simply create a function of two variables x and y and give it a name with a suffix \_sh e.g.

*circle\_sh <- function (x, y, r = 3)*

*{*

*return(x^2 + y^2 - r^2)*

*}*

where the parameter *r* is the radius of the circle selected in such a way that the circle fits in the predefined plotting area.

You may try the following example:

*myshape\_sh <- function(x, y)*

*{*

*if(y != 0)*

*z = x/y*

*else*

*z = 0*

*return(z)*

*}*

The function *myshape\_sh* will be found by *mlsim* when you restart it and will appear on the list of shapes.

Obviously selecting the right algorithm, although important part of a data science project, does not guarantee that it will work well with a new data set. Even if an algorithm is flexible enough to capture complex data patterns this may be due to overfitting and lead to poor predictions when applied to a new data set. In order to minimise the likelihood of overfitting it is critical to perform hyper parameter tunning and cross-validation.