

1 Topic

How to perform Random Forest and Boosting with R and Python mainly, but also with Tanagra and Knime .

This tutorial follows the slideshow devoted to the "[Bagging, Random Forest and Boosting](#)". We show the implementation of these methods on a data file. We will follow the same steps as the slideshow i.e. we first describe the construction of a decision tree, we measure the prediction performance, and then we see how ensemble methods can improve the results. Various aspects of these methods will be highlighted: the measure of the variable importance, the influence of the parameters, the influence of the characteristics of the underlying classifier (e.g. controlling the tree size), etc.

As a first step, we will focus on R (rpart, adabag and randomforest packages) and Python (scikit-learn package). We can multiply analyses by programming. Among others, we can evaluate the influence of parameters on the performance. As a second step, we will explore the capabilities of software (Tanagra and Knime) providing turnkey solutions, very simple to implement, more accessible for people which do not like programming.

2 Dataset

We use the « [Image Segmentation Data Set](#) » from the UCI Machine Learning repository. The instances were drawn randomly from a database of 7 outdoor images. The images were hand segmented to create a classification for every pixel. There are 210 instances for the training sample, and 2100 instances for the test set.

Rather than manipulate two data files, we have gathered observations in the single data file "**image.txt**", with an additional column "**sample**" indicating their membership (train or test). Here are some rows and columns of the dataset. **REGION.TYPE** is the target attribute.

REGION.TYPE	EXGREEN.MEA	VALUE.MEAN	SATURATION.MEAN	HUE.MEAN	sample
GRASS	7.1111	18.5556	0.2927	2.7898	train
GRASS	13.2222	18.5556	0.4216	2.3925	train
GRASS	13.7778	17.5556	0.4454	1.8388	train
GRASS	17.2222	18.6667	0.5081	1.9109	test
GRASS	16.4444	19.2222	0.4633	1.9415	test
GRASS	14.5556	17.1111	0.4801	1.9879	test

3 Analysis with R

3.1 Data importation and preparation

We import the « `image.txt` » data file with the following parameters.

```
#data importation
setwd("... directory of the data file...")
image_all <- read.table("image.txt", sep="\t", dec=".", header=TRUE)
print(summary(image_all))
```

The `summary()` command provides an overview of the characteristics of the data and allows to detect eventual anomalies.

```
REGION_TYPE REGION_CENTROID_COL REGION_CENTROID_ROW REGION_PIXEL_COUNT SHORT_LINE_DENSITY_5
BRICKFACE:330 Min. : 1.0 Min. : 11.0 Min. :9 Min. :0.00000
CEMENT :330 1st Qu.: 62.0 1st Qu.: 81.0 1st Qu.:9 1st Qu.:0.00000
FOLIAGE :330 Median :121.0 Median :122.0 Median :9 Median :0.00000
GRASS :330 Mean :124.9 Mean :123.4 Mean :9 Mean :0.01433
PATH :330 3rd Qu.:189.0 3rd Qu.:172.0 3rd Qu.:9 3rd Qu.:0.00000
SKY :330 Max. :254.0 Max. :251.0 Max. :9 Max. :0.33333
WINDOW :330
SHORT_LINE_DENSITY_2 VEDGE_MEAN VEDGE_SD HEDGE_MEAN HEDGE_SD
Min. :0.000000 Min. : 0.0000 Min. : 0.0000 Min. : 0.0000 Min. : 0.0000
1st Qu.:0.000000 1st Qu.: 0.7222 1st Qu.: 0.3556 1st Qu.: 0.7778 1st Qu.: 0.4216
Median :0.000000 Median : 1.2222 Median : 0.8333 Median : 1.4444 Median : 0.9630
Mean :0.004714 Mean : 1.8939 Mean : 5.7093 Mean : 2.4247 Mean : 8.2437
3rd Qu.:0.000000 3rd Qu.: 2.1667 3rd Qu.: 1.8064 3rd Qu.: 2.5556 3rd Qu.: 2.1833
Max. :0.222222 Max. :29.2222 Max. :991.7184 Max. :44.7222 Max. :1386.3292

INTENSITY_MEAN RAWRED_MEAN RAWBLUE_MEAN RAWGREEN_MEAN EXRED_MEAN EXBLUE_MEAN
Min. : 0.000 Min. : 0.00 Min. : 0.000 Min. : 0.000 Min. : -49.667 Min. : -12.444
1st Qu.: 7.296 1st Qu.: 7.00 1st Qu.: 9.556 1st Qu.: 6.028 1st Qu.: -18.556 1st Qu.: 4.139
Median : 21.593 Median : 19.56 Median : 27.667 Median : 20.333 Median : -10.889 Median : 19.667
Mean : 37.052 Mean : 32.82 Mean : 44.188 Mean : 34.146 Mean : -12.691 Mean : 21.409
3rd Qu.: 53.213 3rd Qu.: 47.33 3rd Qu.: 64.889 3rd Qu.: 46.500 3rd Qu.: -4.222 3rd Qu.: 35.778
Max. :143.444 Max. :137.11 Max. :150.889 Max. :142.556 Max. : 9.889 Max. : 82.000

EXGREEN_MEAN VALUE_MEAN SATURATION_MEAN HUE_MEAN sample
Min. : -33.889 Min. : 0.00 Min. :0.0000 Min. : -3.044 test :2100
1st Qu.: -16.778 1st Qu.: 11.56 1st Qu.:0.2842 1st Qu.: -2.188 train: 210
Median : -10.889 Median : 28.67 Median :0.3748 Median : -2.051
Mean : -8.718 Mean : 45.14 Mean :0.4269 Mean : -1.363
3rd Qu.: -3.222 3rd Qu.: 64.89 3rd Qu.:0.5401 3rd Qu.: -1.562
Max. : 24.667 Max. :150.89 Max. :1.0000 Max. : 2.913
```

We note that we have a well balanced dataset (`REGION_TYPE`).

We subdivide the data into training and test sets using the column “`sample`”. Then, we exclude this last column from the data.frame.

```
#subdivision into training and test sets
image_train <- image_all[image_all$sample=="train",1:20]
image_test <- image_all[image_all$sample=="test",1:20]
print(summary(image_train$REGION_TYPE))
print(summary(image_test$REGION_TYPE))
```

We have the following class distribution for each sample:

```
> print(summary(image_train$REGION_TYPE))
BRICKFACE    CEMENT    FOLIAGE    GRASS    PATH    SKY    WINDOW
      30      30      30      30      30      30      30
> print(summary(image_test$REGION_TYPE))
BRICKFACE    CEMENT    FOLIAGE    GRASS    PATH    SKY    WINDOW
      300     300     300     300     300     300     300
```

3.2 Function for performance evaluation

We use the error rate to assess the quality of prediction. We write a function for this purpose. It takes as input the observed target variable and the prediction of a model.

```
#function for performance evaluation
error_rate <- function(yobs,ypred){
  #confusion matrix
  mc <- table(yobs,ypred)
  #error rate = 1 - success rate
  err <- 1.0 - sum(diag(mc))/sum(mc)
  return(err)
}
```

3.3 Classification tree

We use the “[rpart](#)” package for the construction of the classification tree. It is very popular and, important for our context, it is underlying to the ensemble methods packages for R (e.g. [adabag](#)). Thus, we can reuse the parameters defined in this section. We will have a consistent view of the results.

3.3.1 Classification tree with the default settings

We fit a first version of the trees with the default settings.

```
#classification tree
library(rpart)
arbre_1 <- rpart(REGION_TYPE ~ ., data = image_train)
print(arbre_1)
```

We obtain a tree with 9 leaves:

```
n= 210

node), split, n, loss, yval, (yprob)
* denotes terminal node

1) root 210 180 BRICKFACE (0.14 0.14 0.14 0.14 0.14 0.14 0.14)
 2) INTENSITY_MEAN< 79.037 180 150 BRICKFACE (0.17 0.17 0.17 0.17 0.17 0 0.17)
   4) EXGREEN_MEAN< 0.8889 150 120 BRICKFACE (0.2 0.2 0.2 0 0.2 0 0.2)
    8) REGION_CENTROID_ROW< 160.5 120 90 BRICKFACE (0.25 0.25 0.25 0 0 0 0.25)
```

```

16) HUE_MEAN>=-1.78935 37 8 BRICKFACE (0.78 0.027 0.027 0 0 0 0.16)
32) EXGREEN_MEAN< -7.05555 30 2 BRICKFACE (0.93 0.033 0.033 0 0 0 0) *
33) EXGREEN_MEAN>=-7.05555 7 1 WINDOW (0.14 0 0 0 0 0 0.86) *
17) HUE_MEAN< -1.78935 83 54 CEMENT (0.012 0.35 0.35 0 0 0 0.29)
34) EXGREEN_MEAN< -10.94445 29 3 CEMENT (0 0.9 0.034 0 0 0 0.069) *
35) EXGREEN_MEAN>=-10.94445 54 26 FOLIAGE (0.019 0.056 0.52 0 0 0 0.41)
70) HUE_MEAN< -2.0828 38 10 FOLIAGE (0 0.026 0.74 0 0 0 0.24)
140) SATURATION_MEAN>=0.50715 25 1 FOLIAGE (0 0 0.96 0 0 0 0.04) *
141) SATURATION_MEAN< 0.50715 13 5 WINDOW (0 0.077 0.31 0 0 0 0.62) *
71) HUE_MEAN>=-2.0828 16 3 WINDOW (0.062 0.12 0 0 0 0 0.81) *
9) REGION_CENTROID_ROW>=160.5 30 0 PATH (0 0 0 0 1 0 0) *
5) EXGREEN_MEAN>=0.8889 30 0 GRASS (0 0 0 1 0 0 0) *
3) INTENSITY_MEAN>=79.037 30 0 SKY (0 0 0 0 0 1 0) *

```

We read carefully the tree:

- the character "*" indicates the terminal nodes (leaves) of the tree;
- there are 9 leaves, thus 9 rules;
- some variables only among the 19 available have been used, some several times (e.g. EXGREEN_MEAN);
- we detail the reading of the node n°34: it contains 29 observations (**n**), with 3 counter-examples (**loss**), the conclusion is CEMENT (**yval**), which corresponds to $\approx 90\%$ ($(29-3)/29 = 0.8966$) (**yprob**) of the instances located on this node.

We calculate the prediction of the model on the test sample, and then we compare it with the observed target variable.

```

#prediction on the test set
pred_1 <- predict(arbre_1,newdata=image_test,type="class")

#error rate
print(error_rate(image_test$REGION_TYPE,pred_1))

```

The test error rate is **12.85%**.

3.3.2 Decision stump

A [decision stump](#) is a one-level decision tree. We have only two leaves if we fit a binary tree. It is not really adapted in our context of multiclass target attribute. But this kind of tree can be useful in the ensemble methods context that we study below, especially for the boosting approach which reduces the bias of the base classifier. Indeed, we note that we obtain an overall linear classifier with the boosting of decision stumps. In this section, we mostly want to identify the parameters that change the behavior of the rpart() procedure.

```
#decision stump
param_stump = rpart.control(cp=0,maxdepth=1,minsplit=2,minbucket=1)
arbre_2 <- rpart(REGION_TYPE ~ ., data = image_train,control=param_stump)
print(arbre_2)
```

The “control” option enables to set the parameters of the algorithm:

- “cp” acts as pre-pruning parameter during the growing of the tree. A split is accepted only if the relative reduction in the Gini index is greater than "cp". By setting its value to zero, we disable his action.
- “minsplit” indicates the minimal size (number of instances) of a node in order to attempt a split.
- “minbucket” corresponds to the minimum number of observations in any leaf.
- “maxdepth” corresponds to the maximum depth of any node of the final tree, with the root node counted as depth 0. With the value “maxdepth = 1”, we define a decision stump (one-level tree).

We obtain the following classification tree...

```
n= 210
node), split, n, loss, yval, (yprob)
  * denotes terminal node
1) root 210 180 BRICKFACE (0.14 0.14 0.14 0.14 0.14 0.14 0.14)
  2) INTENSITY_MEAN< 79.037 180 150 BRICKFACE (0.17 0.17 0.17 0.17 0.17 0 0.17) *
  3) INTENSITY_MEAN>=79.037 30 0 SKY (0 0 0 0 0 1 0) *
```

which is not really efficient...

```
#prediction and error rate
pred_2 <- predict(arbre_2,newdata=image_test,type="class")
print(error_rate(image_test$REGION_TYPE,pred_2))
```

... with a test error rate = **71.42%**. Only the class SKY is recognized.

3.3.3 Deeper tree

In this section, we try to learn a very deep tree with a maximum depth of 30 levels (which is the default value). We do not want that the settings about impurity reduction or node size interfere here. We put them to the minimum.

We set the following instructions:

```
#new settings: deeper tree
param_deep = rpart.control(cp=0,maxdepth=30,minsplit=2,minbucket=1)
```

```
arbre_3 <- rpart(REGION_TYPE ~ ., data = image_train,control=param_deep)
#prediction and error rate
pred_3 <- predict(arbre_3,newdata=image_test,type="class")
print(error_rate(image_test$REGION_TYPE,pred_3))
```

We obtain a tree with 21 leaves with a test error rate = **10.42%**. The large tree is better than the first. There is no overfitting. This is not very usual. This suggests that we have not noisy labels. The quality of the learning depends from the number of observations in the learning sample, which is especially determinant with regard to the performance of decision trees.

3.4 Bagging

We use the “[adabag](#)” package for the bagging process under R A thorough description of the package and the procedures has been published in Journal of Statistical Software¹.

3.4.1 Bagging with 20 trees (default parameter)

We begin with a bagging with 20 trees. We do not modify the other settings (parameters of the underlying tree learning algorithm).

```
#adabag package
library(adabag)

#bagging
bag_1 <- bagging(REGION_TYPE ~ ., data = image_train, mfinal=20)

#prediction
predbag_1 <- predict(bag_1,newdata = image_test)

#test error rate
print(error_rate(image_test$REGION_TYPE,predbag_1$class))
```

The **print()** of the object provides many information (e.g. individuals in each bootstrap sample, prediction, class membership probability, etc.) which are difficult to interpret. We will look at the most important elements in the following sections.

We perform the prediction on the test sample. We obtain a prediction object with several properties including the predicted class values (`$class`). The test error rate is **8.86%**. This is best result that we obtain up to now.

¹ E. Alfaro, M. Gamez, N. Garcia, « [adabag: An R Package for Classification with Boosting and Bagging](#) », in Journal of Statistical Software, 54(2), 2013.

3.4.2 Accessing the trees

The method generates a collection of trees. They are accessible with the property `$trees` of the result object. We access to the first generated tree.

```
#first tree
print(bag_1$trees[[1]])
```

We obtain:

```
n= 210

node), split, n, loss, yval, (yprob)
  * denotes terminal node

1) root 210 176 CEMENT (0.15 0.16 0.14 0.16 0.12 0.14 0.13)
 2) EXGREEN_MEAN< 0.8889 176 142 CEMENT (0.18 0.19 0.16 0 0.14 0.17 0.15)
 4) INTENSITY_MEAN< 78.16665 146 112 CEMENT (0.21 0.23 0.2 0 0.17 0 0.18)
 8) REGION_CENTROID_ROW< 160.5 121 87 CEMENT (0.26 0.28 0.24 0 0 0 0.22)
 16) HUE_MEAN>=-1.64025 27 1 BRICKFACE (0.96 0 0 0 0 0 0.037) *
 17) HUE_MEAN< -1.64025 94 60 CEMENT (0.053 0.36 0.31 0 0 0 0.28)
 34) EXGREEN_MEAN< -12.05555 36 7 CEMENT (0.11 0.81 0.028 0 0 0 0.056)
 68) SATURATION_MEAN< 0.3764 27 0 CEMENT (0 1 0 0 0 0 0) *
 69) SATURATION_MEAN>=0.3764 9 5 BRICKFACE (0.44 0.22 0.11 0 0 0 0.22) *
 35) EXGREEN_MEAN>=-12.05555 58 30 FOLIAGE (0.017 0.086 0.48 0 0 0 0.41)
 70) SATURATION_MEAN>=0.7639 26 3 FOLIAGE (0 0 0.88 0 0 0 0.12) *
 71) SATURATION_MEAN< 0.7639 32 11 WINDOW (0.031 0.16 0.16 0 0 0 0.66)
 142) REGION_CENTROID_ROW>=145.5 8 3 CEMENT (0.12 0.62 0 0 0 0 0.25) *
 143) REGION_CENTROID_ROW< 145.5 24 5 WINDOW (0 0 0.21 0 0 0 0.79)
 286) REGION_CENTROID_COL< 104.5 8 3 FOLIAGE (0 0 0.62 0 0 0 0.37) *
 287) REGION_CENTROID_COL>=104.5 16 0 WINDOW (0 0 0 0 0 0 1) *
 9) REGION_CENTROID_ROW>=160.5 25 0 PATH (0 0 0 0 1 0 0) *
 5) INTENSITY_MEAN>=78.16665 30 0 SKY (0 0 0 0 0 1 0) *
 3) EXGREEN_MEAN>=0.8889 34 0 GRASS (0 0 0 1 0 0 0) *
```

The bootstrap sample consists of 210 observations ($n = 210$ in the R output above). But, compared with the learning sample, some instances are repeated, others are absent. For this reason, we get a different tree than with `rpart()` on the original learning sample, even if the learning algorithm is based on the same default settings (section 3.3.1).

3.4.3 Importance of each variable

It is impossible to analyze all the trees to evaluate the influence of the predictor variables in the modeling. The "variable importance" measurement allows to overcome this drawback. We display them in descending order of importance here:

```
#variable importance
print(sort(bag_1$importance, decreasing=TRUE))
```

We obtain the following results:

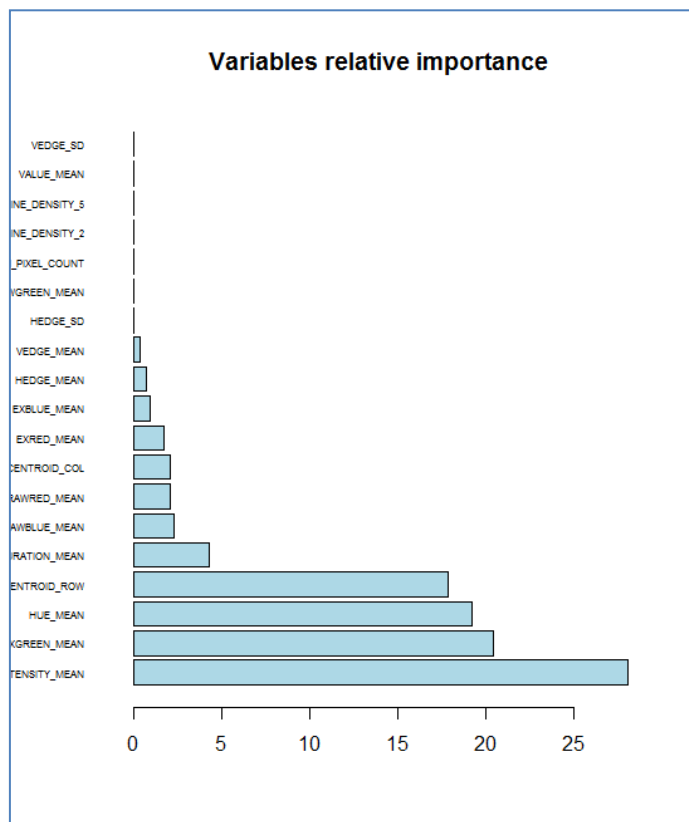
INTENSITY_MEAN	EXGREEN_MEAN	HUE_MEAN	REGION_CENTROID_ROW
28.0874825	20.4393368	19.2377527	17.8348773
SATURATION_MEAN	RAWBLUE_MEAN	RAWRED_MEAN	REGION_CENTROID_COL
4.2957024	2.2591761	2.1003289	2.0693125
EXRED_MEAN	EXBLUE_MEAN	HEDGE_MEAN	VEDGE_MEAN
1.6970205	0.9473870	0.6807030	0.3509204
HEDGE_SD	RAWGREEN_MEAN	REGION_PIXEL_COUNT	SHORT_LINE_DENSITY_2
0.0000000	0.0000000	0.0000000	0.0000000
SHORT_LINE_DENSITY_5	VALUE_MEAN	VEDGE_SD	
0.0000000	0.0000000	0.0000000	

INTENSITY_MEAN it is the most important variable in the sense that it induces the highest reduction of impurity in the trees where it appears. HEDGE_SD à VEDGE_SD have no influence (importance = 0) because they do not appear in any tree (we detail this comment below, section 3.4.4).

A graphical output is available with the command `importanceplot()` :

```
#graphical output
importanceplot(bag_1, cex.names=0.5, horiz=TRUE)
```

There is a discrepancy between the first, the following 3 variables, then the others.



3.4.4 Comment about the calculation of the variable importance

I had a doubt about the `adabag` calculation of the variable importance. Indeed, in accordance with the CART² methodology, `rpart` proposes a variable important measure that quantifies the influence of a variable, even if it does not appear in the tree. It is based on the surrogate split³ mechanism. I was wondering if `adabag` does not simply perform a sum of the values provided by `rpart`. I have therefore developed a bagged model with a single tree that I inspected.

```
#bagging with one tree
bag_seu1 <- bagging(REGION_TYPE ~ ., data = image_train,mfinal=1)
#the tree
print(bag_seu1$trees[[1]])
#variable importance
print(bag_seu1$importance)
```

We obtain the following classification tree,

```
n= 210

node), split, n, loss, yval, (yprob)
      * denotes terminal node

 1) root 210 177 BRICKFACE (0.16 0.13 0.15 0.15 0.14 0.13 0.14)
   2) EXGREEN_MEAN< 0.8889 179 146 BRICKFACE (0.18 0.16 0.17 0 0.16 0.16 0.17)
      4) REGION_CENTROID_ROW< 160.5 150 117 BRICKFACE (0.22 0.19 0.21 0 0 0.19 0.2)
          8) INTENSITY_MEAN< 79.037 122 89 BRICKFACE (0.27 0.23 0.25 0 0 0 0.25)
              16) HUE_MEAN>=-1.8422 44 11 BRICKFACE (0.75 0.023 0.068 0 0 0 0.16)
                  32) EXGREEN_MEAN< -7.05555 37 4 BRICKFACE (0.89 0.027 0.081 0 0 0 0) *
                      33) EXGREEN_MEAN>=-7.05555 7 0 WINDOW (0 0 0 0 0 0 1) *
                          17) HUE_MEAN< -1.8422 78 50 FOLIAGE (0 0.35 0.36 0 0 0 0.29)
                              34) EXGREEN_MEAN< -10.94445 30 4 CEMENT (0 0.87 0.067 0 0 0 0.067) *
                                  35) EXGREEN_MEAN>=-10.94445 48 22 FOLIAGE (0 0.021 0.54 0 0 0 0.44)
                                      70) HUE_MEAN< -2.2124 20 0 FOLIAGE (0 0 1 0 0 0 0) *
                                          71) HUE_MEAN>=-2.2124 28 7 WINDOW (0 0.036 0.21 0 0 0 0.75)
                                              142) RAWRED_MEAN< 0.7778 9 3 FOLIAGE (0 0 0.67 0 0 0 0.33) *
                                                  143) RAWRED_MEAN>=0.7778 19 1 WINDOW (0 0.053 0 0 0 0 0.95) *
                                                      9) INTENSITY_MEAN>=79.037 28 0 SKY (0 0 0 0 0 1 0) *
                                                          5) REGION_CENTROID_ROW>=160.5 29 0 PATH (0 0 0 0 1 0 0) *
                                                              3) EXGREEN_MEAN>=0.8889 31 0 GRASS (0 0 0 1 0 0 0) *
```

The following variables are present: EXGREEN_MEAN, REGION_CENTROID_ROW, INTENSITY_MEAN, HUE_MEAN, RAWRED_MEAN. In the variable importance list, only these

² L. Breiman, J. Friedman, R. Olshen, C. Stone, « Classification and Regression Trees », Wadsworth, 1984.

³ T. Therneau, E. Atkinson, « [An Introduction to Recursive Partitioning Using RPART Routines](#) », 2015 ; see section 3.4, page 11.

variables have an importance greater than 0. The most important variable is EXGREEN_MEAN because it appears repeatedly, especially at the root of the tree.

EXBLUE_MEAN	EXGREEN_MEAN	EXRED_MEAN	HEDGE_MEAN
0.000000	38.471715	0.000000	0.000000
HEDGE_SD	HUE_MEAN	INTENSITY_MEAN	RAWBLUE_MEAN
0.000000	22.219920	17.858486	0.000000
RAWGREEN_MEAN	RAWRED_MEAN	REGION_CENTROID_COL	REGION_CENTROID_ROW
0.000000	3.155757	0.000000	18.294122
REGION_PIXEL_COUNT	SATURATION_MEAN	SHORT_LINE_DENSITY_2	SHORT_LINE_DENSITY_5
0.000000	0.000000	0.000000	0.000000
VALUE_MEAN	VEDGE_MEAN	VEDGE_SD	
0.000000	0.000000	0.000000	

Obviously, "adabag" takes account only the variables that appear in the trees when it computes the variable importance.

3.4.5 Modifying the tree characteristics

According to the literature, Bagging affects only the variance component of the error, not the bias. Thus, a bagging of decision stumps is not a good idea; by contrast, increasing the size of the base classifications trees would have a positive effect. Let us examine that.

Bagging of decision stumps. We reuse the parameters defined above (section 3.3.2).

```
#bagging of decision stumps
bag_stump <- bagging(REGION_TYPE~., data=image_train, mfinal=20, control=param_stump)
#prediction
predbag_stump <- predict(bag_stump, newdata = image_test)
#test error rate
print(error_rate(image_test$REGION_TYPE, predbag_stump$class))
```

The idea is disastrous on our dataset with a test error rate = **85.71%**.

Bagging of deep trees. The aim is to reduce the bias of the base classification trees. We hope that the combination of the classifiers overcompensates the increase in variance.

```
#bagging of large trees
bag_deep <- bagging(REGION_TYPE~., data=image_train, mfinal=20, control=param_deep)
#prediction
predbag_deep <- predict(bag_deep, newdata = image_test)
#test error rate
print(error_rate(image_test$REGION_TYPE, predbag_deep$class))
```

The test error rate is equal to **6.14%**. Of course, we cannot generalize from a single result, but this kind of behavior is consistent with the theory.

3.4.6 Make varying the number of trees

The number of base classifiers is crucial in the ensemble methods. In our experiment, we try the following number of trees [$m = (1, 5, 10, 20, 50, 100, 200)$] and we measure the test error rate. Each m is evaluated **20 times** in order to have some stability in the results. We then calculate the average of the error rates.

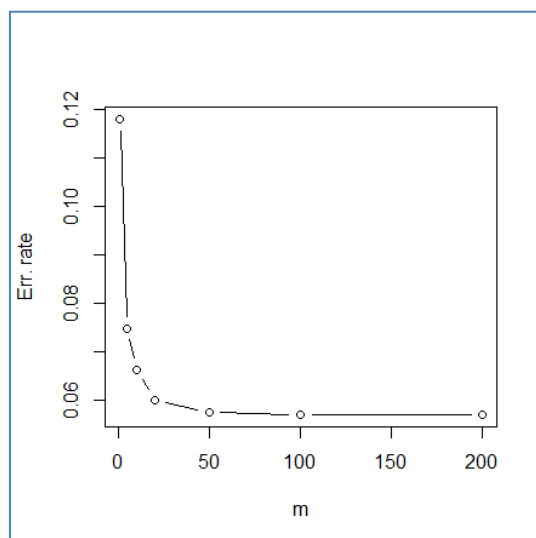
```
#various values of number of trees
m_a_tester <- c(1,5,10,20,50,100,200)

#learning and testing phases
train_test_bag <- function(m){
  bag <- bagging(REGION_TYPE ~ .,data=image_train,mfinal=m,control=param_deep)
  predbag <- predict(bag,newdata = image_test)
  return(error_rate(image_test$REGION_TYPE,predbag$class))
}

#evaluate 20 times each value of m
result <- replicate(20,sapply(m_a_tester,train_test_bag))

#graphical representation of the results
#horizontal axis: m, vertical axis: mean of the errors for each m
plot(m_a_tester,apply(result,1,mean),xlab="m",ylab="Err. rate",type="b")
```

Knowing programming in R becomes significant here!



Starting from $m = 50$, additional trees do not improve the performance. The error rate would be **5.7%** for $m = 50$. We must take with caution this value because we used the test set to select the best model. It is not really impartial. It would be more appropriate to use another procedure for the selection of models. Scikit-learn for Python for example uses the cross-

validation to detect the best combination of parameters. Thereafter, we can evaluate the best model identified in this way on the test set (see section 4).

3.5 Random Forest

In certain respects, [Random Forest](#) is an improved version of the bagging, where the underlying models are necessarily classification trees. A random disturbance is introduced in the learning process in order to "decorrelate" them. We use "randomForest" library for R. A very large tree is created with the default settings: when `maxnodes` is not specified, there is no limitation on the size of the tree; `nodesize` indicates the minimum number of observations into leaves (default value 1).

3.5.1 Random Forest with 20 trees

We fit and evaluate a model with 20 trees.

```
#random forest
library(randomForest)
rf_1 <- randomForest(REGION_TYPE ~ ., data = image_train, ntree = 20)

#prediction
predrf_1 <- predict(rf_1,newdata=image_test,type="class")

#test error rate
print(error_rate(image_test$REGION_TYPE,predrf_1))
```

The test error rate is **5.05%**.

3.5.2 Out-of-bag (OOB) error rate

Random Forest offers an internal mechanism for the error rate estimation. We do not need an additional dataset or an additional learning process. We have access to the confusion matrix and we can deduce the error rate.

```
# out-of-bag confusion matrix
print(rf_1$confusion)
# out-of-bag error rate
print(1-sum(diag(rf_1$confusion))/sum(rf_1$confusion))
```

We have **10.3%**. The out-of-bag error clearly overestimates the error on our dataset. We did well to use a separate test sample for our example.

3.5.3 Accessing to trees

We can access to the underlying trees.

```
#access to the first tree
print(getTree(rf_1,1))
```

The presentation is not intuitive:

	left daughter	right daughter	split var	split point	status	prediction
1	2	3	11	69.22220	1	0
2	4	5	19	0.91940	1	0
3	0	0	0	0.00000	-1	6
4	6	7	8	0.63890	1	0
5	0	0	0	0.00000	-1	4
6	8	9	11	5.11115	1	0
7	10	11	2	159.50000	1	0
8	12	13	11	0.11110	1	0
9	14	15	10	7.40740	1	0
10	16	17	13	23.05555	1	0
11	0	0	0	0.00000	-1	5
12	18	19	14	-0.77780	1	0
13	0	0	0	0.00000	-1	7
14	0	0	0	0.00000	-1	1
15	0	0	0	0.00000	-1	7
16	20	21	19	-1.85465	1	0
17	0	0	0	0.00000	-1	2
18	0	0	0	0.00000	-1	3
19	0	0	0	0.00000	-1	7
20	22	23	2	145.50000	1	0
21	24	25	18	0.38770	1	0
22	26	27	10	14.74075	1	0
23	28	29	12	7.55555	1	0
24	0	0	0	0.00000	-1	3
25	30	31	6	1.61110	1	0
26	32	33	7	0.10740	1	0
27	34	35	19	-2.19505	1	0
28	0	0	0	0.00000	-1	7
29	0	0	0	0.00000	-1	2
30	0	0	0	0.00000	-1	1
31	36	37	11	12.05555	1	0
32	0	0	0	0.00000	-1	7
33	0	0	0	0.00000	-1	3
34	0	0	0	0.00000	-1	3
35	38	39	17	28.55555	1	0
36	0	0	0	0.00000	-1	7
37	0	0	0	0.00000	-1	1
38	0	0	0	0.00000	-1	7
39	0	0	0	0.00000	-1	2

The nodes are numbered, splitting variable is identified by its number, an intermediate node corresponds to a status equal to 1, a leaf has a status equal to -1, the predicted class number is indicated in the prediction column.

3.5.4 Variable importance

The calculation of the variable importance is consistent with the bagging. In addition, the “randomForest” package indicates the number of occurrence of variables, knowing that a variable may be present multiple times in a tree.

```
#occurrence of variables
print(data.frame(cbind(colnames(image_train)[2:20],varUsed(rf_1))))

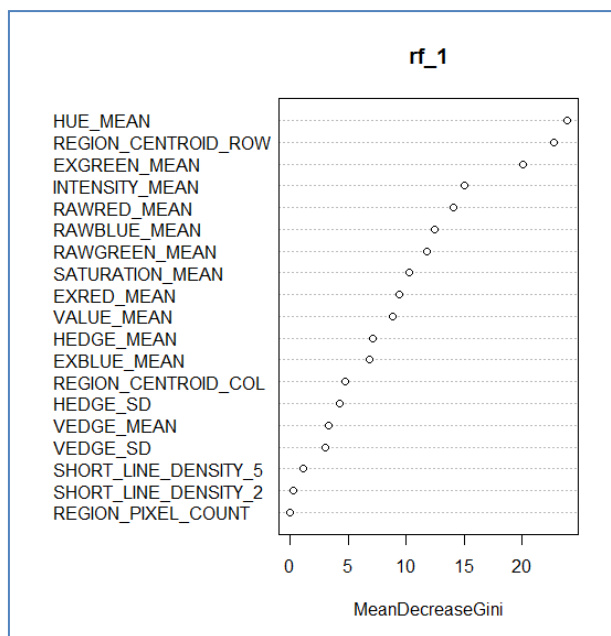
#variable importance
varImpPlot(rf_1)
```

REGION_PIXEL_COUNT is included in no tree despite the random selection mechanism....

		X1	X2
1	REGION_CENTROID_COL	27	
2	REGION_CENTROID_ROW	33	
3	REGION_PIXEL_COUNT	0	
4	SHORT_LINE_DENSITY_5	7	
5	SHORT_LINE_DENSITY_2	2	
6	VEDGE_MEAN	19	
7	VEDGE_SD	20	
8	HEDGE_MEAN	25	
9	HEDGE_SD	22	
10	INTENSITY_MEAN	27	
11	RAWRED_MEAN	27	
12	RAWBLUE_MEAN	21	
13	RAWGREEN_MEAN	24	
14	EXRED_MEAN	23	
15	EXBLUE_MEAN	16	
16	EXGREEN_MEAN	33	
17	VALUE_MEAN	14	
18	SATURATION_MEAN	28	
19	HUE_MEAN	45	

... its importance is logically null.

HUE_MEAN is the most relevant. This is not the case in the bagging where INTENSITY_MEAN seems to be the most relevant one (section 3.4.3).



3.5.5 Tree number

We reiterate the experimentation to identify the "optimal" number of trees. Compared to "adabag", "randomForest" is very quick.

```
# number of trees to try
m_a_tester <- c(1,5,10,20,50,100,200)

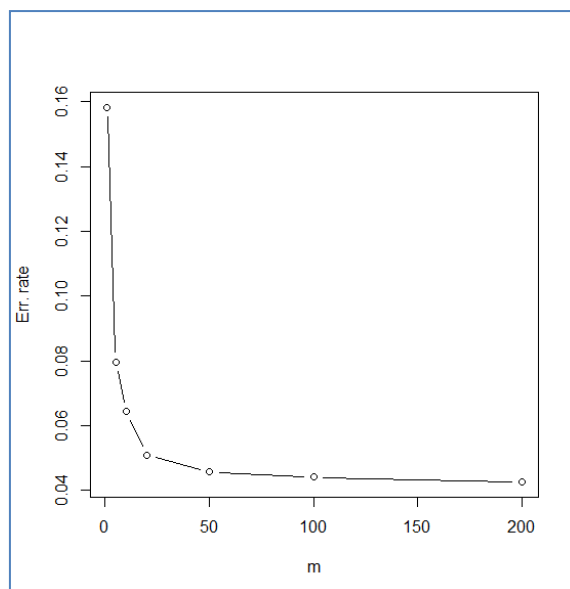
#training and testing phase
train_test_rf <- function(m){
  rf <- randomForest(REGION_TYPE ~ .,data=image_train,ntree=m)
  predrf <- predict(rf,newdata = image_test)
  return(error_rate(image_test$REGION_TYPE,predrf))
}

#evaluate 20 times for each value of m
result <- replicate(20,sapply(m_a_tester,train_test_rf))

#graphical representation
plot(m_a_tester,apply(result,1,mean),xlab="m",ylab="Err. rate",type="b")
```

Starting from $m = 100$ trees, the error decreases very slowly, but seems still decrease. There is no overfitting phenomenon when we add trees.

With $m = 200$, the test error rate is **4.25%**. But, again, because the test sample is used in order to detect the best model, the test error rate must be considered with caution.



3.6 Boosting

We use again the “`adabag`” package in order to implement boosting with R. The process is the same as the previous sections. The variable importance takes into account the weight of the classifiers here. The real issues are the setting of the underlying tree and the number of trees. Indeed, the boosting may be subject to overfitting.

3.6.1 Boosting with 20 trees (default settings for the trees)

```
#boosting
bo_1 <- boosting(REGION_TYPE ~ ., data = image_train,mfinal=20, boos=FALSE)

#prediction
predbo_1 <- predict(bo_1,newdata = image_test)

#test error rate
print(error_rate(image_test$REGION_TYPE,predbo_1$class))
```

The test error rate is **5.67%**. This result is better than any bagging we have tried. It is similar than the Random Forest with 20 trees (5.05%).

The option “`boos = FALSE`” plays an important role. It indicates that we are using the original version AdaBoost based on the weighting of all individuals. If it is equal to `TRUE`, the algorithm relies on a random sampling with replacement, but with unequal probability proportional to the weights. The result is not deterministic in this case.

3.6.2 Depth of the trees

Boosting with decision stumps. We use the SAMME algorithm here (`coeflearn = 'Zhu'`), more adapted to the multiclass problem.

```
#boosting with decision stumps
bo_stump <- boosting(REGION_TYPE ~ ., data = image_train,mfinal=20, coeflearn=
                    'Zhu', control=param_stump, boos=FALSE)

#prediction
predbo_stump <- predict(bo_stump,newdata = image_test)

#test error rate
print(error_rate(image_test$REGION_TYPE,predbo_stump$class))
```

This is not at all convincing with a test error rate = **61.6%**. Boosting allows to reduce bias, but in our context of a target attribute with 7 values, it is not efficient.

Boosting with deep trees. Let us see if, as for the bagging, the increase in the size of the tree influences positively the performance.

```
#boosting with deep trees
bo_deep <- boosting(REGION_TYPE ~ ., data = image_train,mfinal=20, boos=FALSE,
                   coeflearn= 'Zhu', control=param_deep)

#prediction
predbo_deep <- predict(bo_deep,newdata = image_test)

#test error rate
print(error_rate(image_test$REGION_TYPE,predbo_deep$class))
```

Here too, this solution is not convincing with a test error rate equal to **10.4%**. We are facing a overfitting problem when the underlying model is too complex, in accordance with the machine learning literature.

3.6.3 Tree number

We reiterate the experimentation on the number of trees.

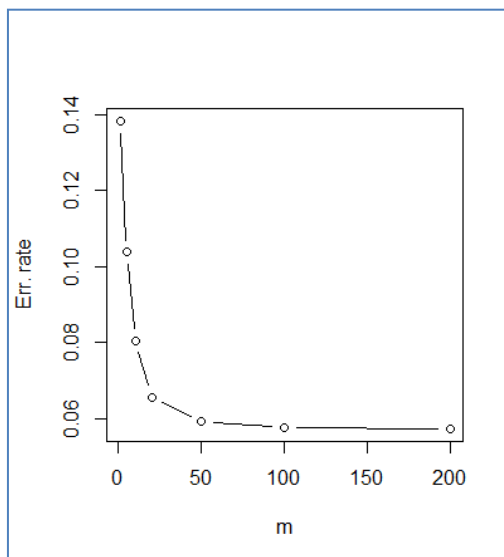
```
#number of trees
m_a_tester <- c(1,5,10,20,50,100,200)

#function for training and testing
train_test_boosting <- function(m){
  bo <- boosting(REGION_TYPE ~ .,data=image_train,mfinal=m,coeflearn='Zhu')
  predbo <- predict(bo,newdata = image_test)
  return(error_rate(image_test$REGION_TYPE,predbo$class))
}
```

```
#evaluate 20 times for each value of m
result <- replicate(20,sapply(m_a_tester,train_test_boosting))

#graphical representation
plot(m_a_tester,apply(result,1,mean),xlab="m",ylab="Err. rate",type="b")
```

We obtain the following graphical representation:



Starting from $m = 50$, the decreasing of the error rate is small. But there is not an overfitting phenomenon when we increase m (until $m = 200$ in any case).

4 Analysis with Python

In this section, we want to show the implementation of the different techniques in Python ([scikit-learn](#) package, [version 0.17](#)), without trying to reproduce all of the experiments that have been conducted under R. We presented the package “[scikit-learn](#)” in a previous tutorial⁴. This second document allows us to go further.

4.1 Data importation and preparation

We import the data file “image.txt” and we verify the shape of the table object.

```
#change the default directory
import os
os.chdir("... your directory ...")

#import the data file using the pandas library
```

⁴ Tanagra, “[Python - Machine Learning with scikit-learn \(slides\)](#)”, December 2015.

```
import pandas
image_all = pandas.read_table("image.txt",sep="\t",header=0,decimal= ".")

#shape of the table
print(image_all.shape)    # (2310, 21)
```

We use the “[pandas](#)” package for the importation. “`image_all`” is an object of the DataFrame class⁵, very similar to the R class⁶. We have a tabular data with 2310 rows and 21 columns.

We exploit the column "sample" in order to subdivide the dataset into training and test sets. We extract Numpy matrices and vectors that we use with scikit-learn procedures thereafter.

We extract the training set,

```
#training sample - select the instances
image_train = image_all[image_all["sample"]=="train"]
#remove the column 'sample'
image_train = image_train.iloc[:,0:20]
#checking
print(image_train.shape) # (210, 20)
#transformation into numpy matrix
d_train = image_train.as_matrix()
#vector for the target attribute
y_app = d_train[:,0]
#matrix for the predictive attributes
X_app = d_train[:,1:20]
```

The predictive variables and the target are divided in two different objects: **X_app** is a matrix with 210 rows and 19 columns; **y_app** is a vector with 210 values.

We proceed the same for the test set.

```
#test
image_test = (image_all[image_all["sample"]=="test"]).iloc[:,0:20]
print(image_test.shape)    # (2100, 20)
y_test = image_test.as_matrix()[:,0]
X_test = image_test.as_matrix()[:,1:20]
```

4.2 Function for performance evaluation

As for R, we define a function which calculates the error rate on the test sample. It is very generic because we use only the “scikit-learn” package. The signatures of the functions are the same regardless of the used machine learning algorithm.

⁵ pandas 0.17.1 documentation - <http://pandas.pydata.org/pandas-docs/stable/index.html>

⁶ See the comparison: http://pandas.pydata.org/pandas-docs/stable/comparison_with_r.html

The function takes as input: the model developed from the learning sample; for the test sample, the vector of the variable target and the matrix of predictive variables. We use the **metrics** module from the scikit-learn package to calculate the accuracy rate. The error rate is the complementary to one of the accuracy rate.

```
#module for the evaluation of the classifiers
from sklearn import metrics

#function for the performance evaluation
def error_rate(modele,y_test,X_test):
    #prediction
    y_pred = modele.predict(X_test)
    #error rate = 1 - accuracy rate (success rate)
    err = 1.0 - metrics.accuracy_score(y_test,y_pred)
    #return
    return err
#end fonction
```

4.3 Classification tree

4.3.1 Instantiation and settings

The process is always the same with scikit-learn, at least in the supervised learning task:

```
#Decision tree - importation of the class
from sklearn.tree import DecisionTreeClassifier

#instantiation
dtree = DecisionTreeClassifier()

#print the settings of the algorithm
print(dtree)
```

We import the class related to the method that we instantiate (by calling its constructor). We therefore get an object that we fit on the training set. The display of the instantiated object allows to visualize the algorithm's parameters and their default values.

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                        max_features=None, max_leaf_nodes=None, min_samples_leaf=1,
                        min_samples_split=2, min_weight_fraction_leaf=0.0,
                        presort=False, random_state=None, splitter='best')
```

3 parameters get our attention: `max_depth = None`, there is no limit to the depth of the tree (if we want a “decision stump”, we set `max_depth = 1`, the root is at the level 0); `min_samples_split = 2`, a node is split if it contains at least 2 observations; `min_samples_leaf = 1`, any leaves of the tree must contain at least 1 instance.

With such parameters, we will get a very deep tree.

4.3.2 Learning phase

We fit the model on the learning set:

```
#learning
dtree.fit(X_app,y_app)
```

The operation is apparently OK, but no message was sent. When I wanted to display the tree, I realized that the operation is not easy.

4.3.3 Visualization of the classification tree

There is no default text display with `print()`. The simplest way to visualize the tree seems to generate a file that can be transformed into graphic with the **GraphViz** software⁷. Therefore, we must first download and install this tool⁸.

With the following commands, we generate the “tree.dot” file.

```
#generation of the output -> .dot format
from sklearn import tree
tree.export_graphviz(dtree,out_file="tree.dot",feature_names=image_train.columns[1:20])
```

We load “tree.dot” into a text editor, we observe a not very legible description.

```
digraph Tree {
node [shape=box] ;
0 [label="EXGREEN_MEAN <= 0.8889\ngini = 0.8571\nsamples = 210\nvalue = [30, 30, 30, 30, 30, 30]\nclass = R"] ;
1 [label="REGION_CENTROID_ROW <= 160.5\ngini = 0.8333\nsamples = 180\nvalue = [30, 30, 30, 0, 30, 30, 30]\nclass = R"] ;
0 -> 1 [labeldistance=2.5, labelangle=45, headlabel="True"] ;
...
}
```

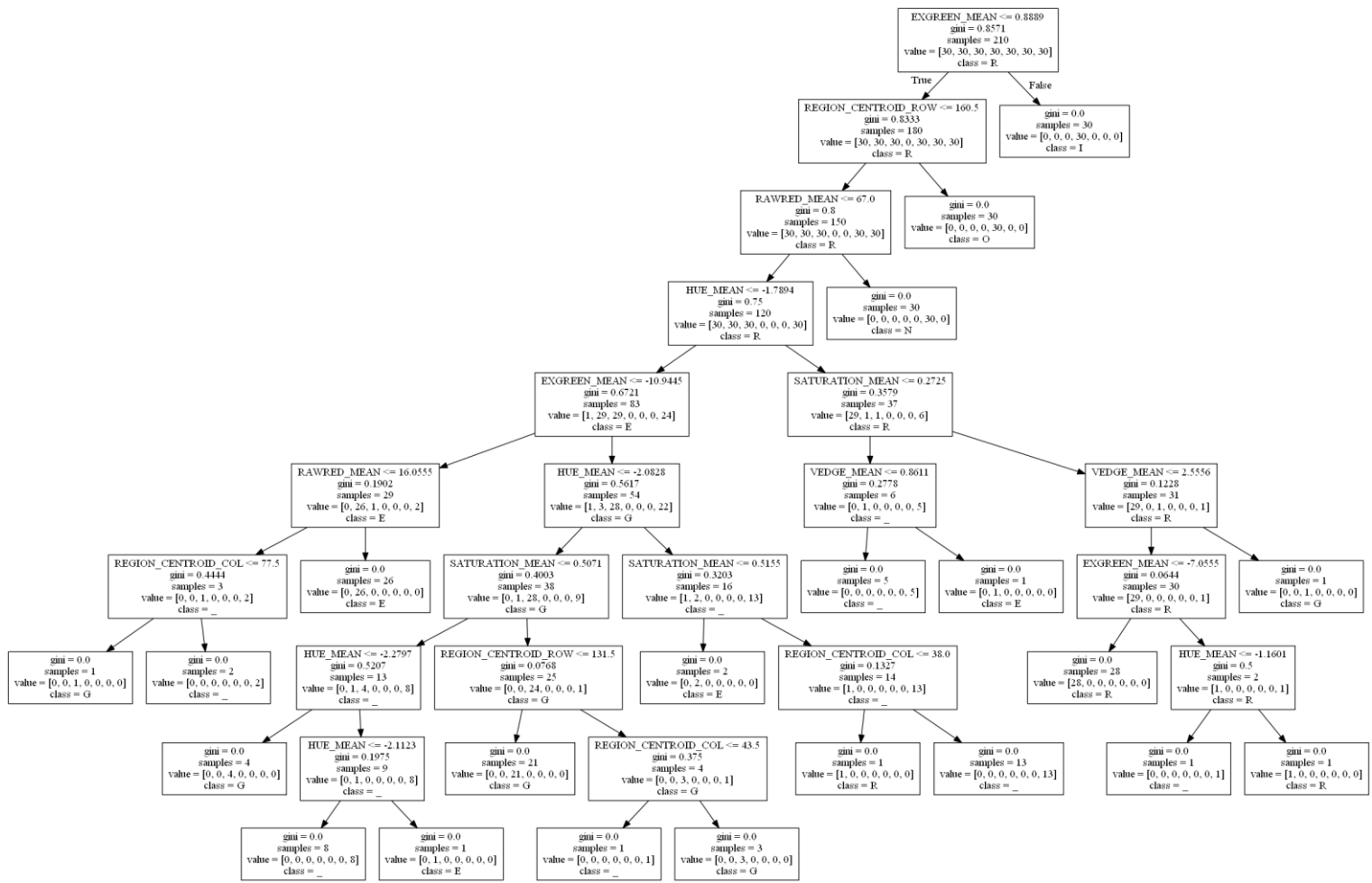
The file is converted with Graphviz using the following command (launched in a DOS command prompt),

```
dot -Tpng tree.dot -o tree.png
```

« dot » is the name of the executable file. A “tree.png” file has been generated. We can visualize it with any graphic software that supports PNG format.

⁷ http://scikit-learn.org/stable/modules/generated/sklearn.tree.export_graphviz.html

⁸ http://www.graphviz.org/Download_windows.php



It is so large that its reading is not really interesting. The main information here is that it is possible to get a graphical representation of the tree with a little extra effort.

4.3.4 Variable importance

We can get the importance of variables. Scikit-learn includes only the variables that appear explicitly in the tree.

```
#importance of variables - 0 when the variable does not appear into the tree
imp = {"VarName":image_train.columns[1:], "Importance":dtree.feature_importances_}
print(pandas.DataFrame(imp))
```

We put the results in a data frame structure in order to be able to match each variable name (obtained by using the `columns` property of the training set) with its importance (`feature_importances_`).

Importance	VarName
0	0.026058 REGION_CENTROID_COL
1	0.169000 REGION_CENTROID_ROW
2	0.000000 REGION_PIXEL_COUNT

3	0.000000	SHORT_LINE_DENSITY_5
4	0.000000	SHORT_LINE_DENSITY_2
5	0.019665	VEDGE_MEAN
6	0.000000	VEDGE_SD
7	0.000000	HEDGE_MEAN
8	0.000000	HEDGE_SD
9	0.000000	INTENSITY_MEAN
10	0.189911	RAWRED_MEAN
11	0.000000	RAWBLUE_MEAN
12	0.000000	RAWGREEN_MEAN
13	0.000000	EXRED_MEAN
14	0.000000	EXBLUE_MEAN
15	0.282588	EXGREEN_MEAN
16	0.000000	VALUE_MEAN
17	0.097552	SATURATION_MEAN
18	0.215226	HUE_MEAN

4.3.5 Prediction and evaluation

We use the `error_rate` function defined above (section 4.2) to measure the performance.

```
#error rate
print(error_rate(dtree,y_test,x_test))
```

The test error rate is **10.38%**, similar to the deep tree with R's `rpart` (10.42%).

4.4 Bagging of classification trees

4.4.1 Train and test

Instantiation. Bagging is a meta-classifier that can take as input any learning algorithm under scikit-learn. As a first step, we develop a bagging of 20 deep trees, equivalent to that carried out under R (section 3.4.5).

You must first import the class `BaggingClassifier`, and then instantiate it with as a parameter the desired underlying algorithm, a `DecisionTreeClassifier` in our case.

```
#class bagging
from sklearn.ensemble import BaggingClassifier

#instantiation
baggingTree = BaggingClassifier(DecisionTreeClassifier(),n_estimators=20)
print(baggingTree)
```

The print command displays both the characteristics of the meta-classifier (in blue), and those of the base (in purple) algorithm.

```
BaggingClassifier(base_estimator=DecisionTreeClassifier(class_weight=None,
criterion='gini', max_depth=None,
```

```

max_features=None, max_leaf_nodes=None, min_samples_leaf=1,
min_samples_split=2, min_weight_fraction_leaf=0.0,
presort=False, random_state=None, splitter='best'),
bootstrap=True, bootstrap_features=False, max_features=1.0,
max_samples=1.0, n_estimators=20, n_jobs=1, oob_score=False,
random_state=None, verbose=0, warm_start=False)

```

We note a very interesting option. It is possible to make calculations in parallel using the 'n_jobs' option, in order to take advantage of the possibilities of multi-core processors for example. On very large databases, the gain in speed is significant.

Train and test. We fit the model on the training set and we evaluate it on the test set.

```

#training
baggingTree.fit(X_app,y_app)

#test
print(error_rate(baggingTree,y_test,X_test))

```

The test error rate is **5.85%**, slightly better than that of R (6.14%, section 3.4.5).

4.4.2 Tree number

We can also program in Python. In this section, we try to reproduce the detection of the “best” number of trees for the bagging.

```

#train-test function for a given m
def train_test_bagging(m,X_app,y_app,X_test,y_test):
    #instantiation
    bag = BaggingClassifier(DecisionTreeClassifier(),n_estimators=m)
    #fit the model
    bag.fit(X_app,y_app)
    #prediction and calculation of the error rate
    return error_rate(bag,y_test,X_test)
#end train-test

#values of m to evaluate
m_a_tester = [1,5,10,20,50,100,200]

#initialization of the matrix for the results
import numpy
result = numpy.zeros(shape=(1,7))

#repeat 20 times the experiment for m
for expe in range(20):
    #evaluate each value of m
    res = [train_test_bagging(m,X_app,y_app,X_test,y_test) for m in m_a_tester]
    #the vector with 7 values is transformed in a matrix (1, 7)

```



```
res = numpy.asarray(res).reshape(1,7)
#add a new row in the matrix
result = numpy.append(result,res,axis=0)
#
#remove the first row
result = numpy.delete(result,0,axis=0)
#calculate the average of error rate for each m
mresult = numpy.mean(result,axis=0)
print(mresult)
```

There is a double loop: repeat 20 times the experience for each value of m, perform the analysis for different values of m.

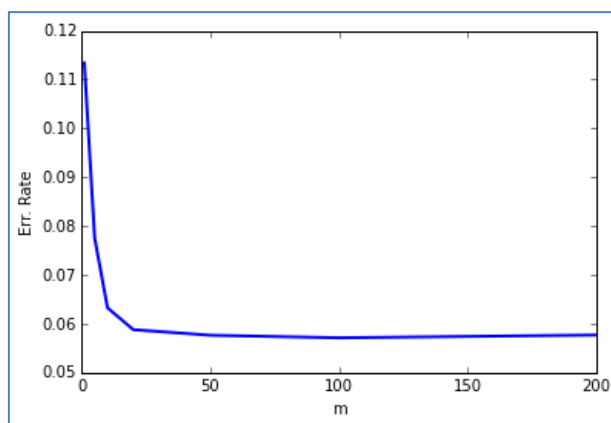
Here are the mean of the error rate for each value of m.

```
[ 0.11340476  0.07761905  0.0632619  0.05878571  0.05764286  0.05711905  0.05769048]
```

We create a graphical representation...

```
#graphical tool
import matplotlib.pyplot as plt
#label of the axes
plt.xlabel("m")
plt.ylabel("Err. Rate")
plt.plot(m_a_tester,mresult,linewidth=2)
```

... we have,



4.4.3 Grid search

By reading the documentation, I realized that scikit-learn proposes a tool for testing the effectiveness of different parameters values. The main interest is that it proceeds by cross-validation to evaluate the quality of the combination of parameters. Thus, our test sample

keeps its impartiality status since it is not used to detect the best configuration, but only to measure the error rate of the latter.

In what follows, for the same values of m of the previous section, we ask to Python to measure the success rate by cross-validation. Then we use the test sample in order to measure the error rate of the best configuration highlighted by the tool.

```
# detecting the "optimal" number of trees
# using the grid search tool
from sklearn.grid_search import GridSearchCV

# the parameters to make vary
# the name of the parameter must be explicit
# we enumerate the values to try
parameters = [{"n_estimators": [1, 5, 10, 20, 50, 100, 200]}]

# instantiate the classifier
bag = BaggingClassifier(DecisionTreeClassifier())

# instantiation of the grid search tool
# the metric used is the accuracy rate (error rate = 1 - accuracy rate)
grid_bag = GridSearchCV(estimator=bag, param_grid=parameters, scoring="accuracy")

# launching the exploration
grille_bag = grid_bag.fit(X_app, y_app)

# print the results
print(grille_bag.grid_scores_)
```

Despite the amount of calculations, the procedure is very fast. We obtain the following outputs:

```
[mean: 0.81905, std: 0.02935, params: {'n_estimators': 1}, mean: 0.88571, std: 0.04666, params: {'n_estimators': 5}, mean: 0.88095, std: 0.00673, params: {'n_estimators': 10}, mean: 0.89048, std: 0.03750, params: {'n_estimators': 20}, mean: 0.90000, std: 0.05084, params: {'n_estimators': 50}, mean: 0.91429, std: 0.04206, params: {'n_estimators': 100}, mean: 0.90000, std: 0.05084, params: {'n_estimators': 200}]
```

For $m = 1$, the accuracy rate in cross-validation is 81.90% ; for $m = 5$, we have 88.57%, etc.

We can directly identify the best configuration:

```
# best score
print(grille_bag.best_score_) # 0.91428

# parameter for the best score
print(grille_bag.best_params_) # {'n_estimators': 100}
```

The solution with $m = 100$ trees is the most effective. When we apply this solution on the test sample...

```
#valuation of the best solution on the test set
print(error_rate(grille_bag,y_test,X_test))
```

... we obtain the error rate **5.71%**, better (very slightly) than our first configuration with $m = 20$ trees (section **Erreur ! Source du renvoi introuvable.**).

4.4.4 Bagging with other base classifier

The bagging is generic under scikit-learn, we can pass any base classifier. For instance, if we want to perform a bagging of linear discriminant analysis (LDA), we will proceed as follows:

```
#import the discriminant analysis class
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
#instantiation
bag_lda = BaggingClassifier(LinearDiscriminantAnalysis(),n_estimators=20)
print(bag_lda)
#learning process
bag_lda.fit(X_app,y_app)
#evaluation
print(error_rate(bag_lda,y_test,X_test))
```

The tool sends a warning about the collinearity between the variables, but it still continue the learning process. The test error rate is **10.43%**. In comparison, if we perform a single instance of the LDA, the error rate is **9.57%**. So it is technically possible to make a bagging of any base classifier. But the gain is really convincing that when it is slightly biased and has a high variance, which is not really the case of the linear discriminant analysis.

4.5 Random Forest

Scikit-learn includes also the Random Forest method. We perform a simple analysis with 20 trees.

```
# RandomForest class
from sklearn.ensemble import RandomForestClassifier
# instantiation
rf = RandomForestClassifier(n_estimators=20)

# training phase
rf.fit(X_app,y_app)
# test error rate
print(error_rate(rf,y_test,X_test))
# importance of variables...
print(rf.feature_importances_)
```

```
# with their names
imp = {"VarName":image_train.columns[1:], "Importance":rf.feature_importances_}
print(pandas.DataFrame(imp))
```

The test error rate is **4.9%**, here are the importance of the variables:

	Importance	VarName
0	0.026045	REGION_CENTROID_COL
1	0.138857	REGION_CENTROID_ROW
2	0.000000	REGION_PIXEL_COUNT
3	0.001765	SHORT_LINE_DENSITY_5
4	0.001373	SHORT_LINE_DENSITY_2
5	0.018827	VEDGE_MEAN
6	0.018828	VEDGE_SD
7	0.033337	HEDGE_MEAN
8	0.025444	HEDGE_SD
9	0.082932	INTENSITY_MEAN
10	0.073985	RAWRED_MEAN
11	0.103093	RAWBLUE_MEAN
12	0.043970	RAWGREEN_MEAN
13	0.037872	EXRED_MEAN
14	0.039667	EXBLUE_MEAN
15	0.082905	EXGREEN_MEAN
16	0.044985	VALUE_MEAN
17	0.085668	SATURATION_MEAN
18	0.140448	HUE_MEAN

In trying to find the “optimal” number of trees with the GridSearchCV tool, it appears that **m = 100** is the best solution with a test error rate of **4.76%**. The gain, compared with $m = 20$, is negligible for Random Forest.

4.6 Boosting

We proceed also simply for boosting, specifying a decision tree as a base classifier. **Note:** a decision stump is used by default if the option "base_estimator" is omitted.

```
# Adaboost
from sklearn.ensemble import AdaBoostClassifier
# instantiation
ab=AdaBoostClassifier(algorithm="SAMME",n_estimators=20,base_estimator=
    DecisionTreeClassifier())
print(ab)

# training phase
ab.fit(X_app,y_app)
# test error rate
print(error_rate(ab,y_test,X_test))
```

The test error rate is **8.9%**.

5 Analysis with other tools

The interest of R and Python is that we have the ability to write programs. The possibilities of analysis are strongly increased. We had noticed that in the two previous sections. But know how to program requires a time of training which is sometimes not available for practitioners of data mining. The tools that we present in this section allow to reproduce our overall process, without having to write a single line of code. Some users appreciate to this characteristic.

5.1 Analysis with Tanagra

Various ensemble methods are available into Tanagra: bagging, arcing ([Breiman, 1998](#)), boosting, and also ensemble techniques which allow to take into account the misclassification costs (Cost Sensitive Bagging, MultiCost)⁹.

The underlying base classifier can be any learning algorithm. We seen previously that both bagging and boosting may be defined theoretically with any type of learning algorithm even if, in practice, a classification tree is the most commonly used.

Random Forest it is not defined as such. It corresponds to a bagging with a special induction tree algorithm (RndTree)¹⁰ where the tree is built as large as possible, with the particular variable selection process when splitting the nodes.

5.1.1 Data importation and preparation

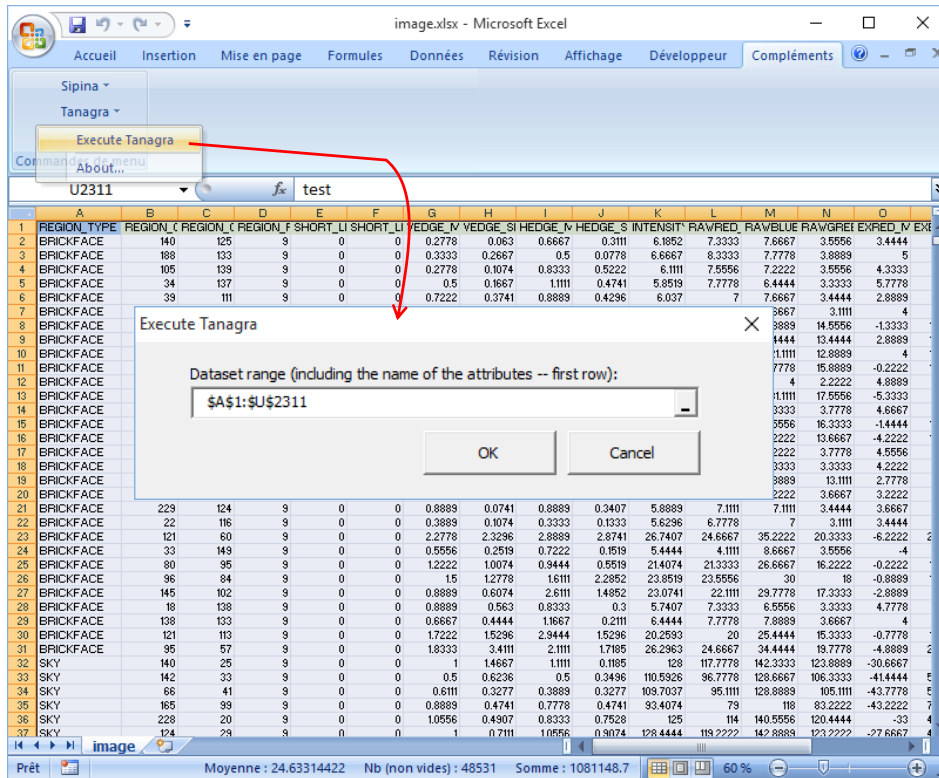
Importing the data file. We load the data file « **image.xlsx** » into Excel. We select the data range and we click on the menu COMPLEMENTS / TANAGRA / EXECUTE (ADD-INS / TANAGRA / EXECUTE in English) installed with the “**tanagra.xla**” add-in for Excel^{11,12}.

⁹ Tanagra, “[Cost-sensitive learning - Comparison of tools](#)”, March 2009.

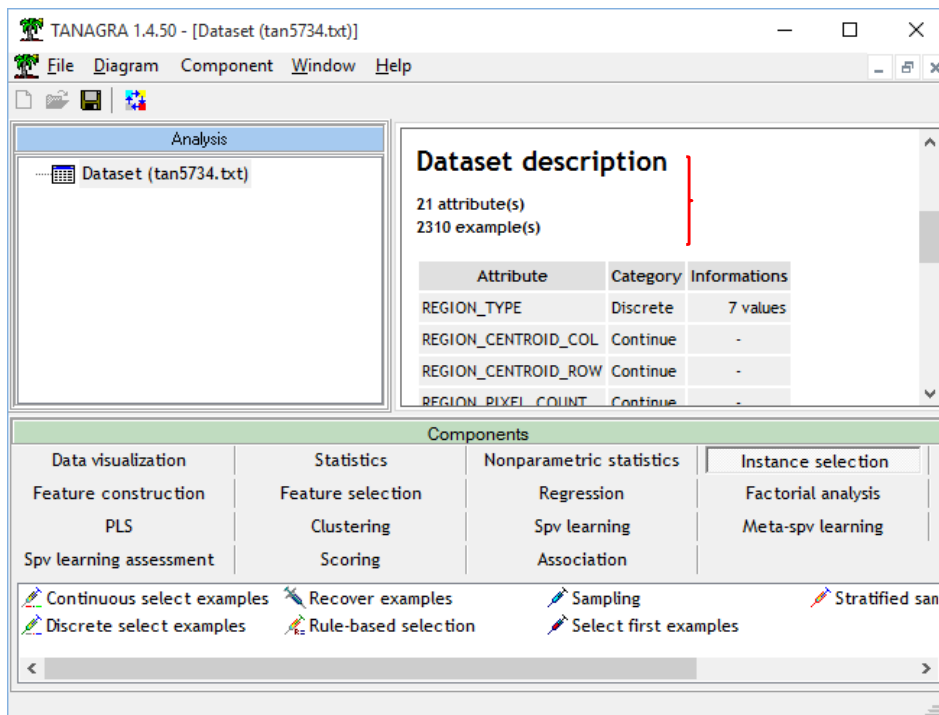
¹⁰ Tanagra, “[Random Forest](#)”, November 2008.

¹¹ Tanagra, “[Tanagra add-in for Excel 2010 - 64-bit version](#)”, December 2011.

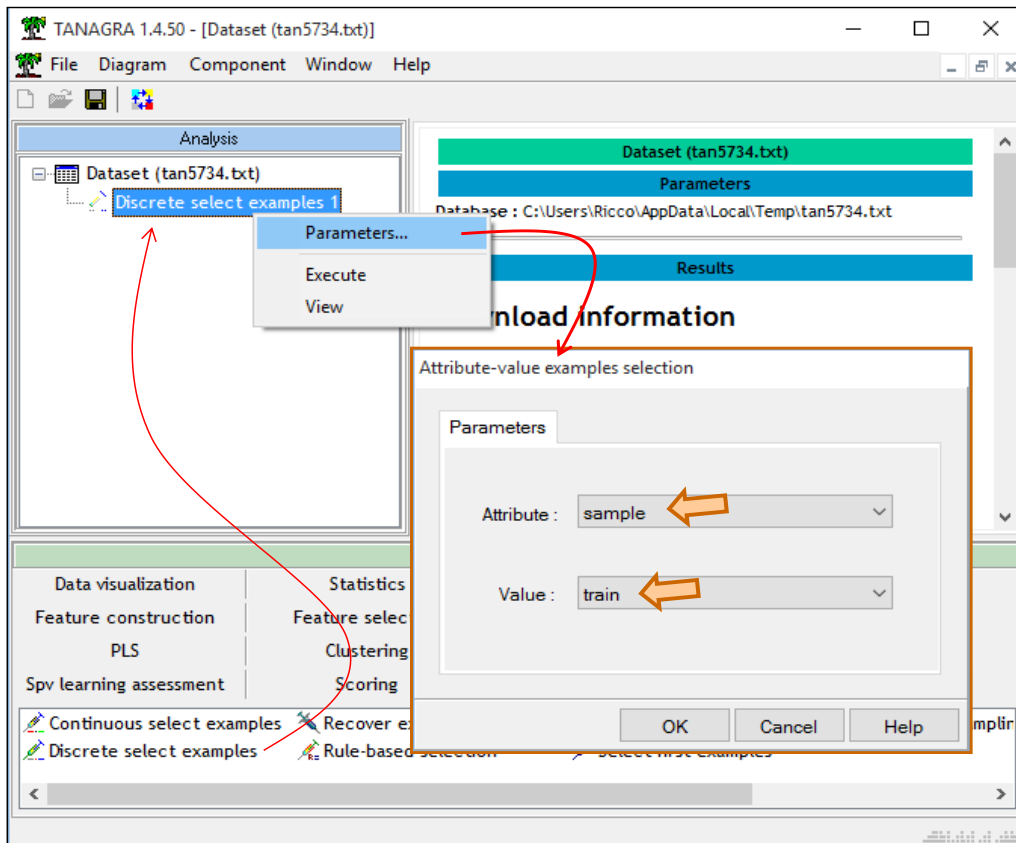
¹² Tanagra, “[Tanagra add-in for Office 2007 and Office 2010](#)”, August 2010.



We click on the OK button. Tanagra is automatically launched and the dataset is imported.

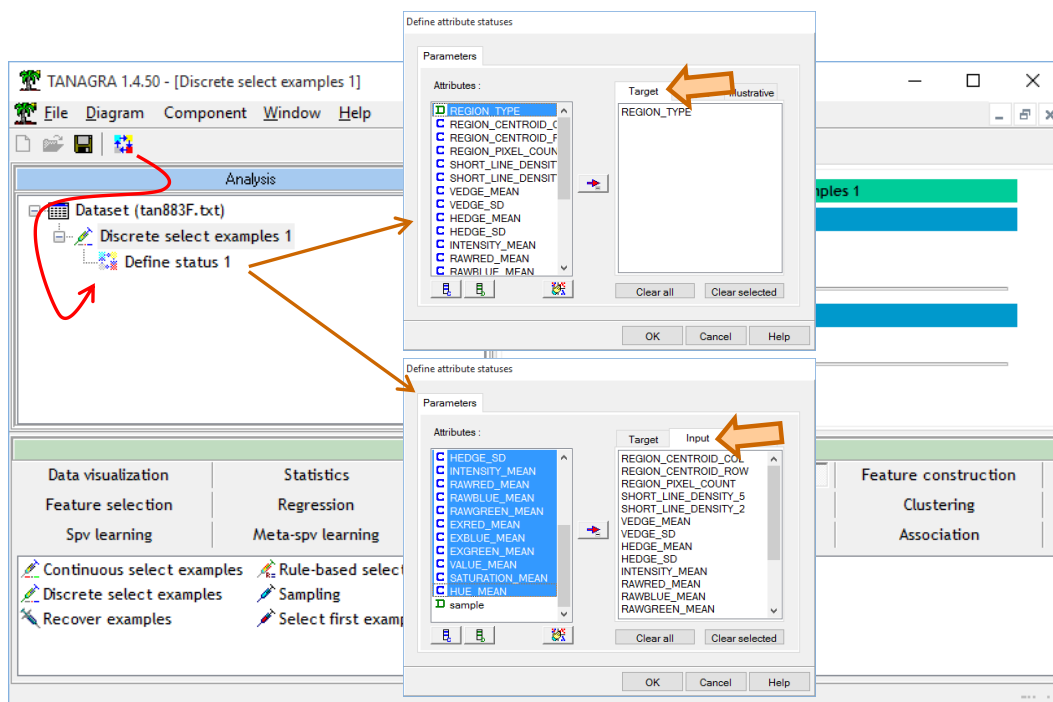


Subdivision into training and test sets. We use the DISCRETE SELECT EXAMPLES component in order to subdivide the dataset according to the “sample” column. We insert it into the diagram, then we click on the PARAMETERS menu.



“sample = train” corresponds to the training sample, with 210 instances.

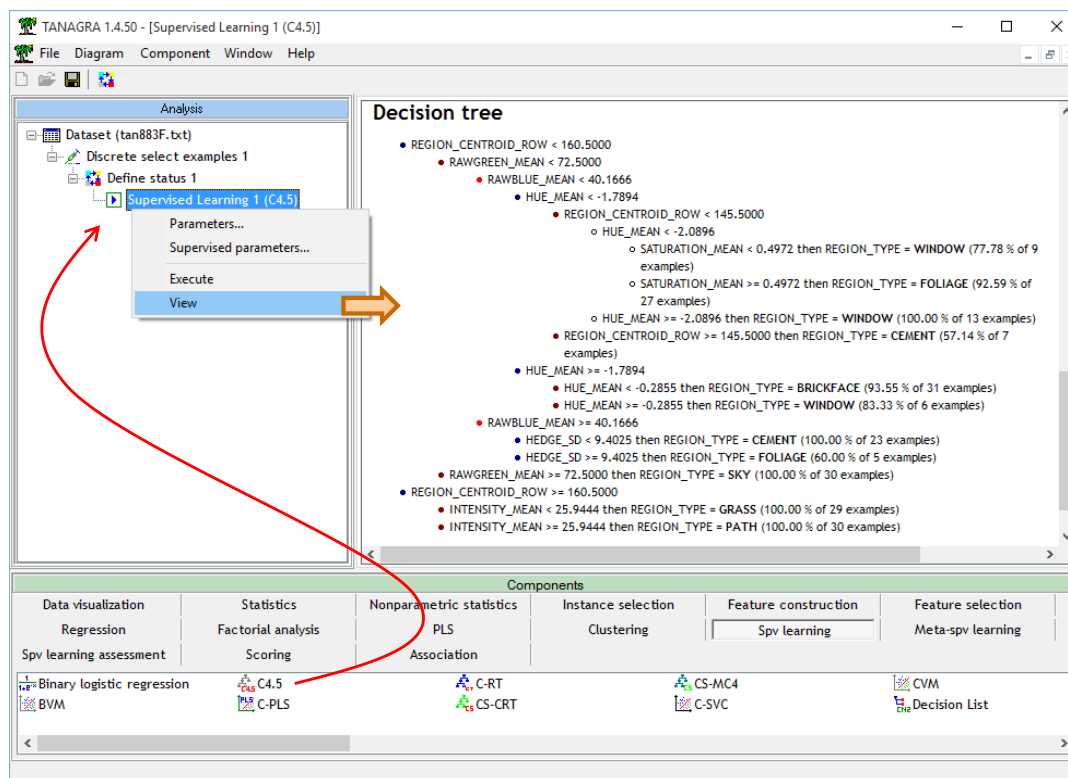
Role of the variables. With the DEFINE STATUS component, we set REGION_TYPE as target attribute, the other ones (with the exception of “sample”) as input attributes.



5.1.2 Classification tree

Learning phase. We insert the C4.5 component (SPV LEARNING tab) after DEFINE STATUS 1.

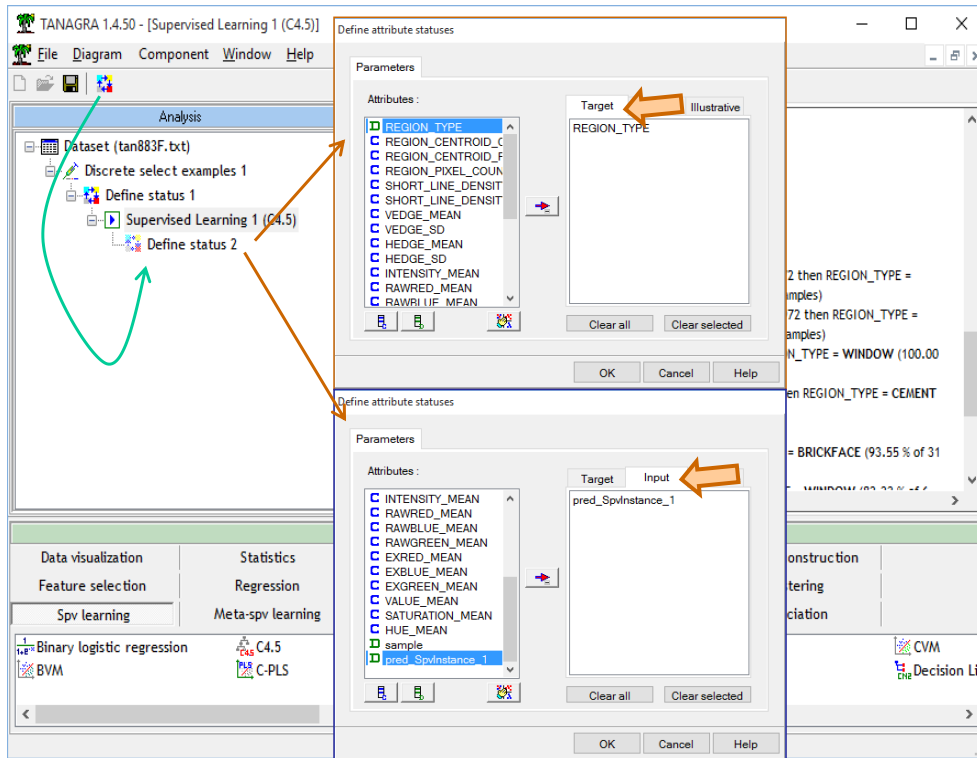
We click on the VIEW menu to visualize the results.



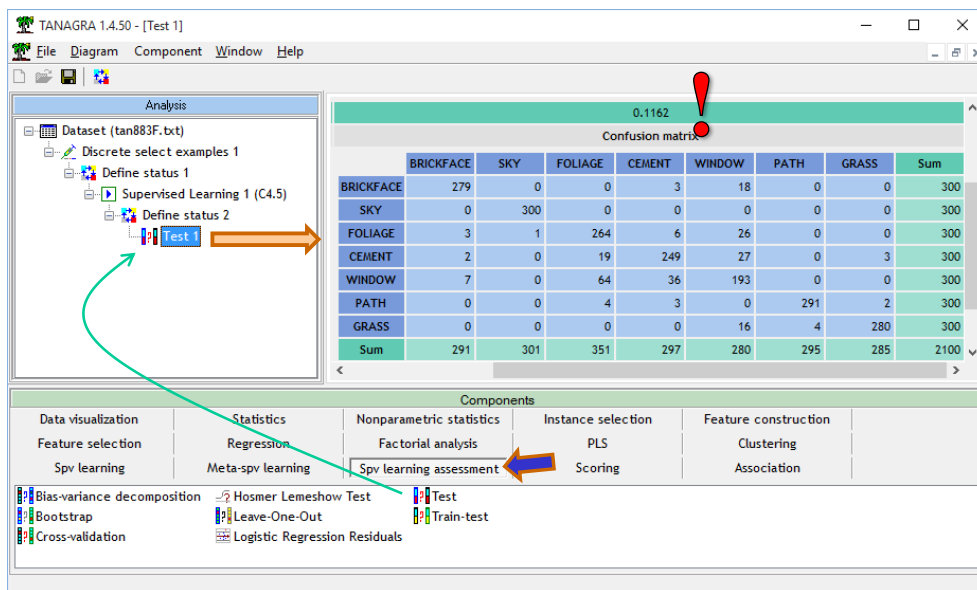
We obtain a classification tree with 11 leaves (11 decision rules):

- REGION_CENTROID_ROW < 160.5000
 - RAWGREEN_MEAN < 72.5000
 - RAWBLUE_MEAN < 40.1666
 - HUE_MEAN < -1.7894
 - REGION_CENTROID_ROW < 145.5000
 - HUE_MEAN < -2.0896
 - SATURATION_MEAN < 0.4972 then REGION_TYPE = WINDOW (77.78 % of 9 examples)
 - SATURATION_MEAN >= 0.4972 then REGION_TYPE = FOLIAGE (92.59 % of 27 examples)
 - HUE_MEAN >= -2.0896 then REGION_TYPE = WINDOW (100.00 % of 13 examples)
 - REGION_CENTROID_ROW >= 145.5000 then REGION_TYPE = CEMENT (57.14 % of 7 examples)
 - HUE_MEAN >= -1.7894
 - HUE_MEAN < -0.2855 then REGION_TYPE = BRICKFACE (93.55 % of 31 examples)
 - HUE_MEAN >= -0.2855 then REGION_TYPE = WINDOW (83.33 % of 6 examples)
 - RAWBLUE_MEAN >= 40.1666
 - HEDGE_SD < 9.4025 then REGION_TYPE = CEMENT (100.00 % of 23 examples)
 - HEDGE_SD >= 9.4025 then REGION_TYPE = FOLIAGE (60.00 % of 5 examples)
 - RAWGREEN_MEAN >= 72.5000 then REGION_TYPE = SKY (100.00 % of 30 examples)
- REGION_CENTROID_ROW >= 160.5000
 - INTENSITY_MEAN < 25.9444 then REGION_TYPE = GRASS (100.00 % of 29 examples)
 - INTENSITY_MEAN >= 25.9444 then REGION_TYPE = PATH (100.00 % of 30 examples)

Evaluation. To calculate the error rate on the set sample, we insert again the DEFINE STATUS component. We set REGION_TYPE as target, the prediction of the classifier (available for all the instances) PRED_SPVINSTANCE_1 as input.



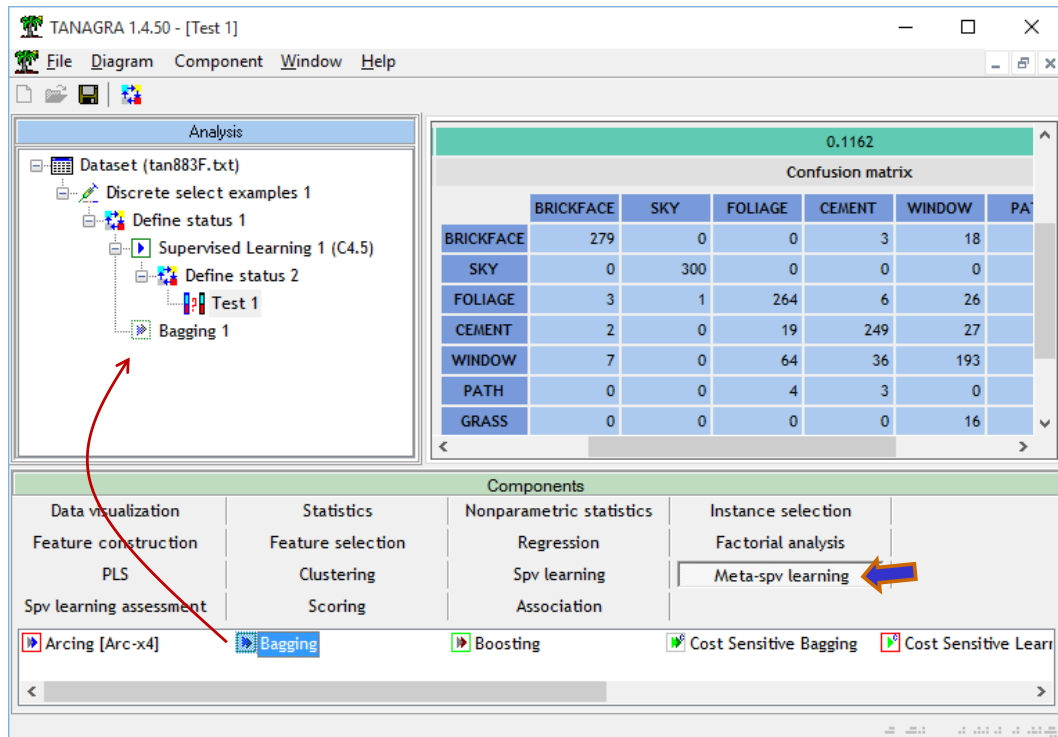
Then, we add the TEST component (SPV LEARNING ASSESSMENT tab) which calculates the confusion matrix and the error rate on the unselected instances i.e. the test set.



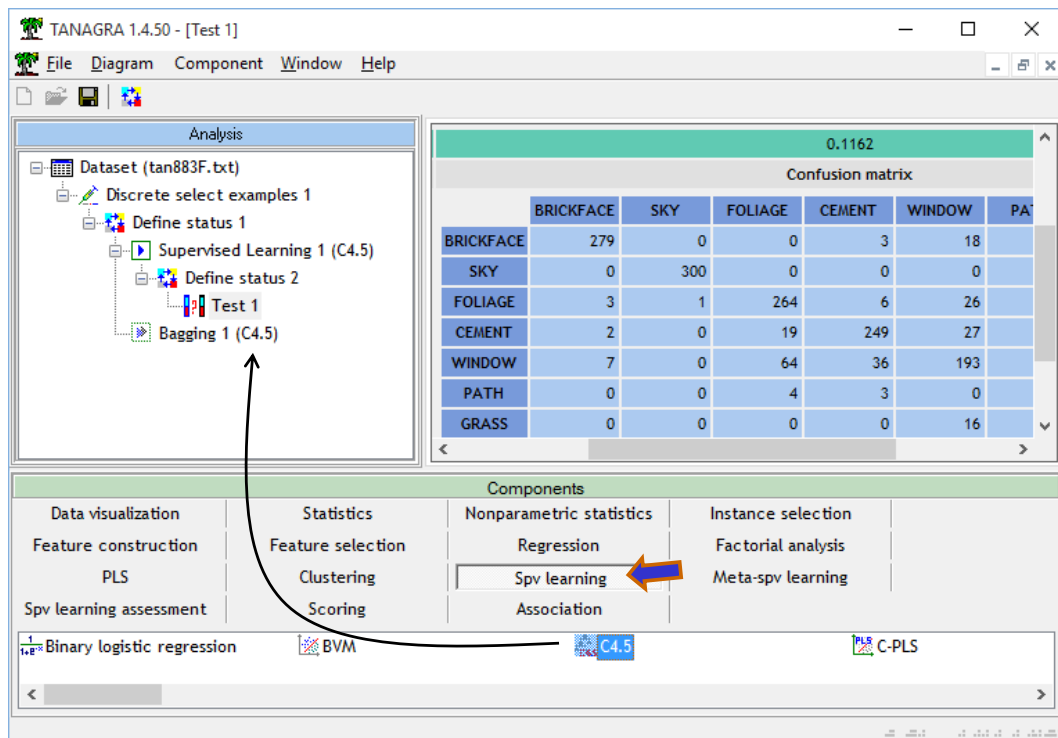
We click on the VIEW menu. The test error rate is **11.62%**.

5.1.3 Bagging

To perform a bagging of C4.5 algorithm, we must proceed in two stages. First, we add the meta learner BAGGING (META-SPV LEARNING tab) into the diagram.



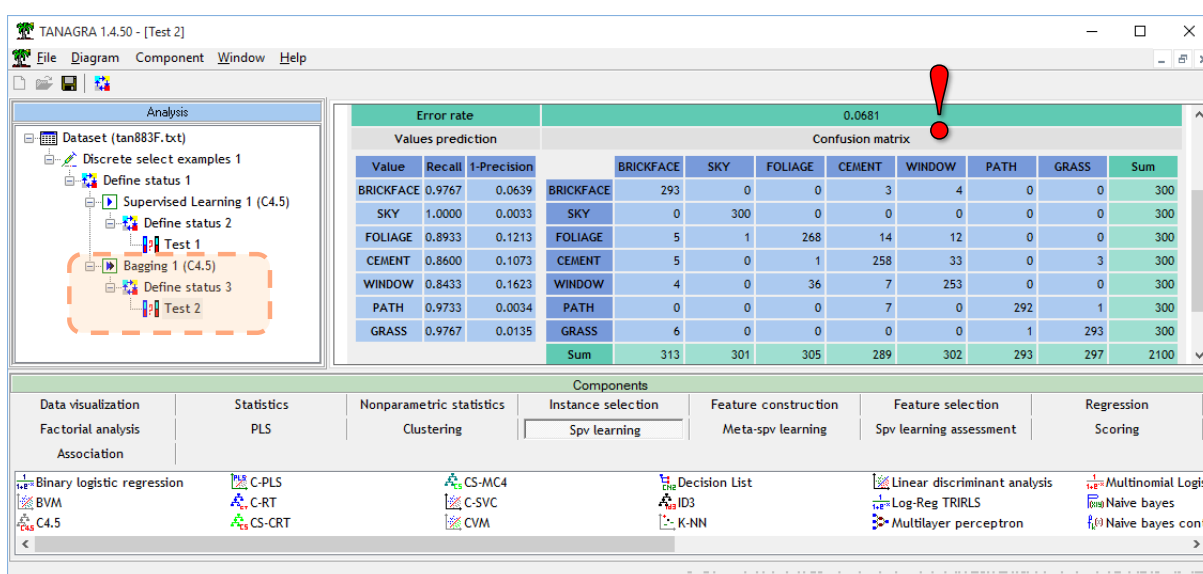
Second, we put the component C4.5 (SPV LEARNING ALGORITHM) into the BAGGING 1.



We can set the parameter of the bagging (contextual menu PARAMETERS, mainly the number of replications, default 25) or the underlying learning algorithm (contextual menu SUPERVISED PARAMETERS: minimal number of instance on the leaves and the confidence level for the calculation of the pessimistic error for the post pruning of C4.5).

Note: Like for scikit-learn, it is possible to perform a bagging with any base classifier. We have seen above that this is not always appropriate (e.g. bagging of linear discriminant analysis, section 4.4.4).

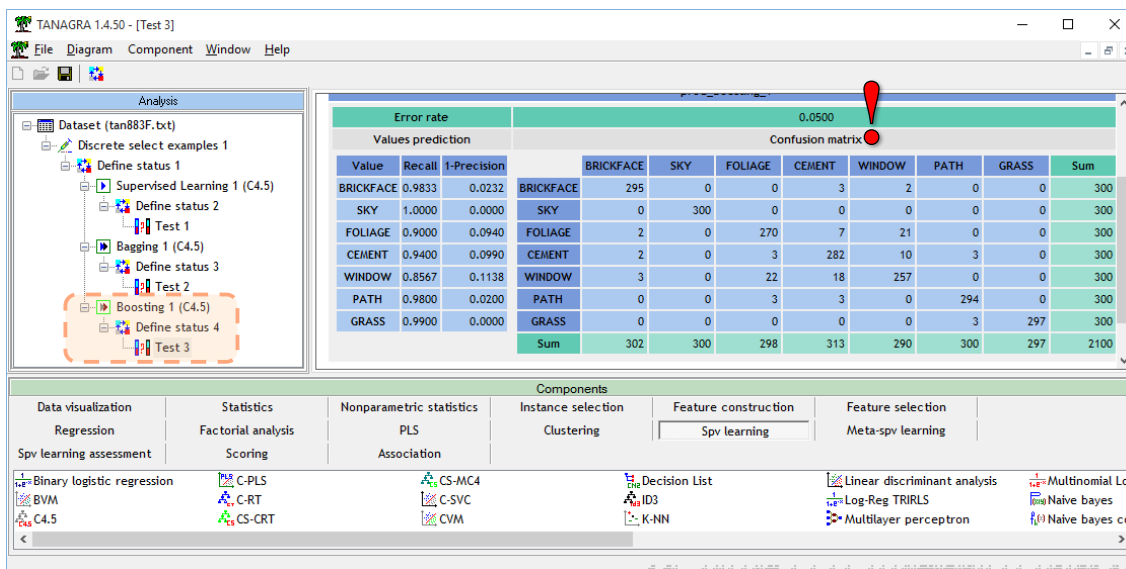
Again, we add the DEFINE STATUS component to compare the REGION_TYPE (target) and the prediction of the model (PRED_BAGGING_1, input). Then, we use the TEST component to calculate the error rate which is equal to **6.18%**.



5.1.4 Boosting

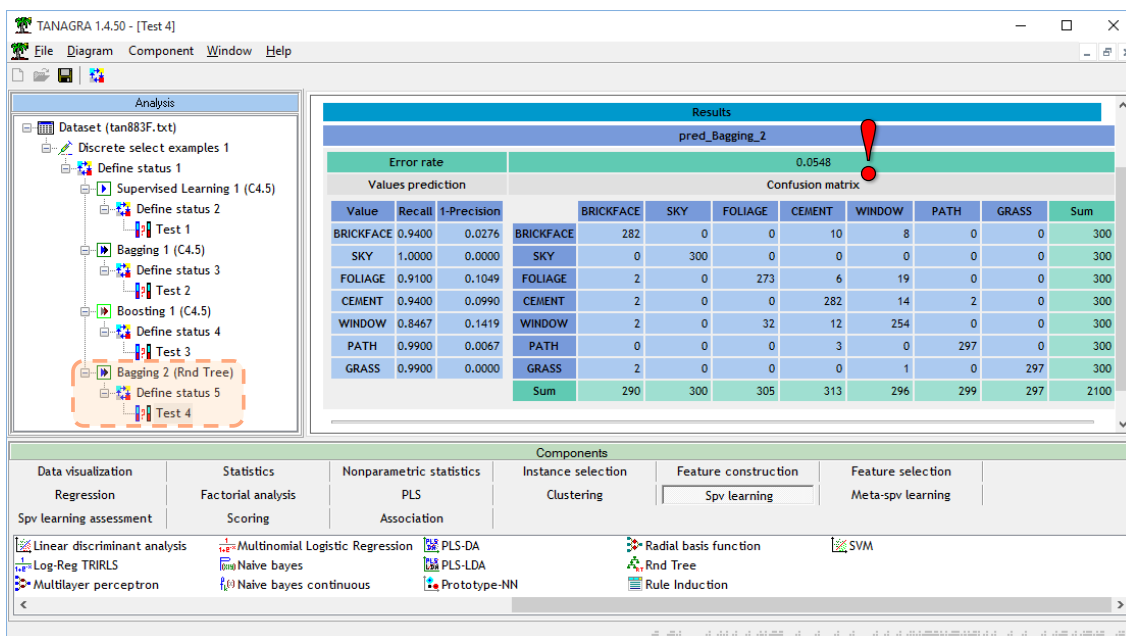
The component BOOSTING is available into the META-SPV LEARNING tab. It implements the ADABOOST.M1 approach. I think that I shall make improve it soon in such a way that it also incorporates the SAMME approach, a more natural method for multi-class problems.

Here also, we proceed in two stages to insert the method into the diagram (BOOSTING first + C4.5 second). We compare the REGION_TYPE and the prediction PRED_BOOSTING_1. The test error rate is **5%**.



5.1.5 Random Forest

To instantiate the Random Forest, we use the BAGGING component to which we associate the RND TREE learning algorithm (SPV LEARNING tab).

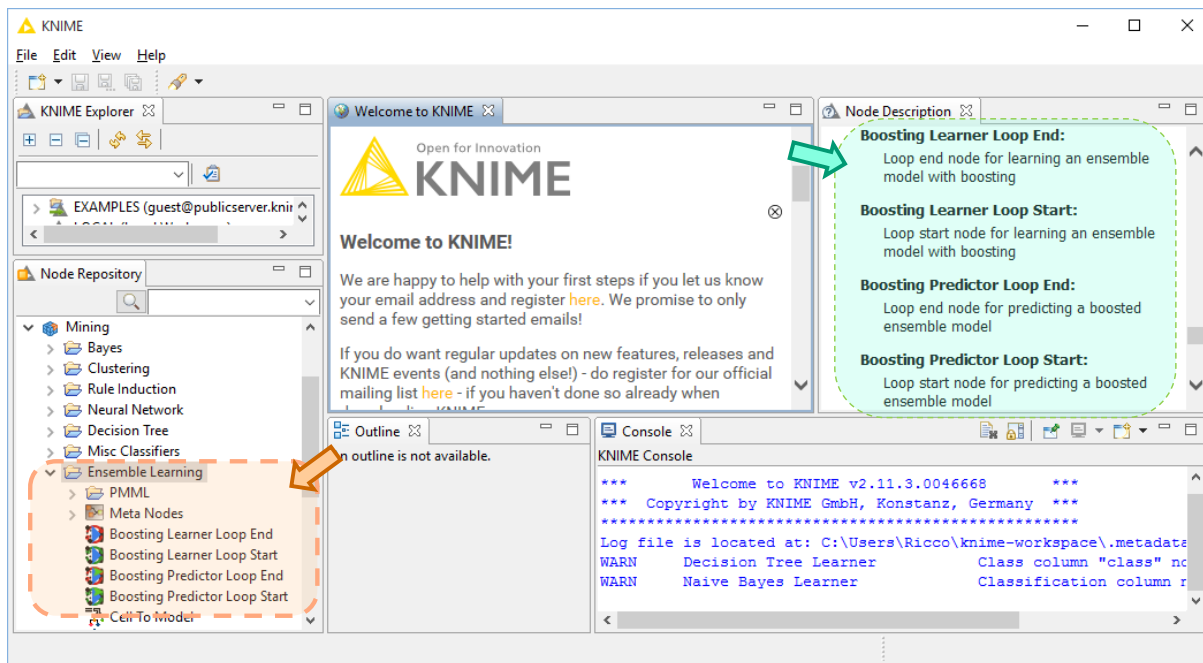


The test error rate is **5.48%** by comparing REGION_TYPE and PRED_BAGGING_2.

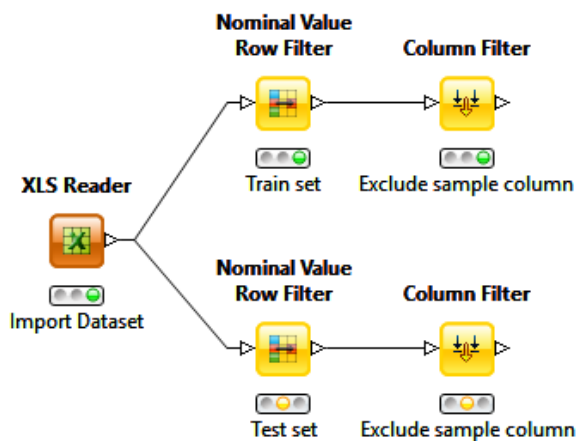
Note: Out of sheer curiosity, I perform a simple learning with RND TREE, the test error rate is 16.14%. This suggests that the proportion of relevant variables is high in the base. The random disturbance does not penalize the learning process. A boosting of RND TREE leads to a test error rate of 5.05%.

5.2 Analysis with Knime

The ENSEMBLE LEARNING package for [Knime](#) incorporates generic components for Bagging and Boosting. We must install the library first. These components can use any base classifier, like Python/Scikit-learn or Tanagra.



5.2.1 Data importation and preparation



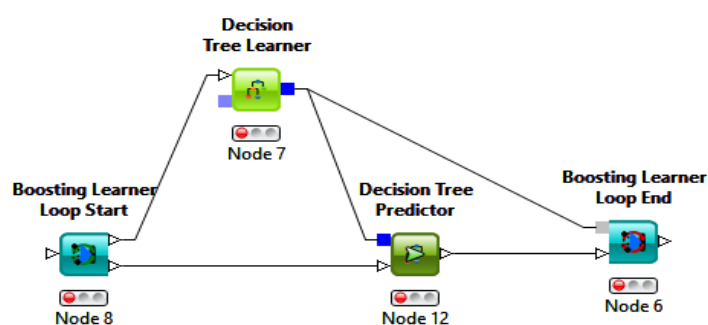
Knime can directly read XLSX files. We use the XLS READER component. With the NOMINAL VALUE ROW FILTER component, we filter the dataset according to the “sample” column to

define the test ("sample = test") and learning ("sample = train") samples. COLUMN FILTER allows to remove the column "sample" for the rest of the study.

5.2.2 Boosting with Knime

Boosting is built in the form of loop in Knime. META-NODES allows to summarize the operations, both for the learning and the prediction phase (Boosting Learner and Boosting Predictor). But I preferred to define the steps manually. Indeed, the process may seem confusing at first. However once we have understood the reasoning, the sequence of the tools into the diagram is pedagogically very interesting.

The sequence below define a Boosting of decision trees. The BOOSTING LEARNER LOOP START component starts the boosting loop boosting from the training set. It is connected to a DECISION TREE LEARNER learning algorithm tool, but also to a predictive DECISION TREE PREDICTOR tool. Indeed, the errors of prediction for the step (t) allows to set the weights of individuals to the step ($t + 1$). BOOSTING LEARNER LOOP END closes the loop and allows to move to the next iteration.



In term of parameters:

- for DECISION TREE LEARNER, we specify the target attribute REGION_TYPE ;
- for BOOSTING LEARNER LOOP END, we specify the target attribute [REGION_TYPE], the prediction of the boosting process [Prediction(REGION_TYPE)], and the number of iterations (25, so that the result would comparable with that of Tanagra).

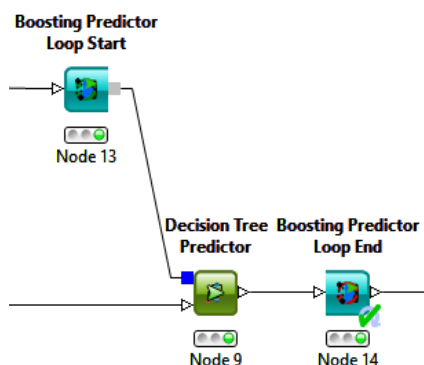
5.2.3 Evaluation on the test sample

Two treatments are necessary at this stage: use the meta-classifier to calculate the prediction on the sample test, then compare the observed class values with the predicted class values to form the confusion matrix and calculate the test error rate.

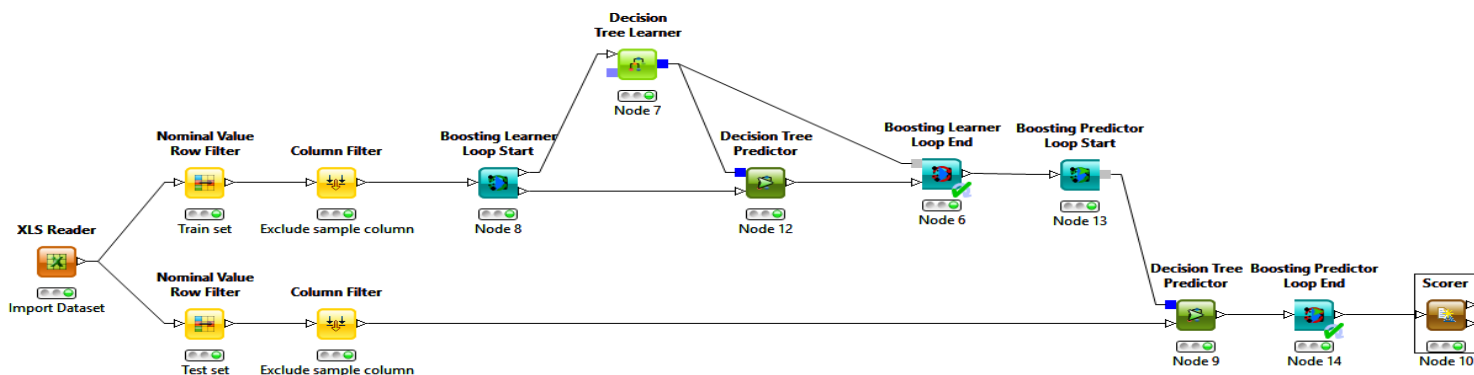
A new loop defines the prediction on the test sample. The BOOSTING PREDICTOR LOOP START tool takes as input the boosting learner loop. It is connected to a DECISION TREE PREDICTOR, which also takes input test data.

We can clearly see the idea. For the prediction, we must activate all the trees and make them vote (a weighted voting for boosting).

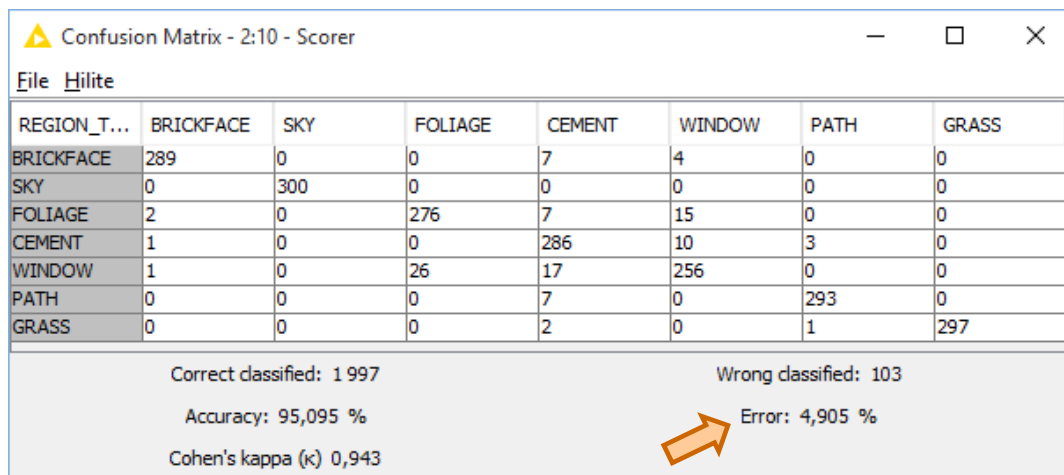
BOOSTING PREDICTOR LOOP END closes the loop.



Then we insert the SCORER tool into the diagram. It compares REGION_TYPE and Prediction(REGION_TYPE). Here is the diagram as a whole.



The test error rate reported by SCORER is **4.90%**.



REGION_T...	BRICKFACE	SKY	FOLIAGE	CEMENT	WINDOW	PATH	GRASS
BRICKFACE	289	0	0	7	4	0	0
SKY	0	300	0	0	0	0	0
FOLIAGE	2	0	276	7	15	0	0
CEMENT	1	0	0	286	10	3	0
WINDOW	1	0	26	17	256	0	0
PATH	0	0	0	7	0	293	0
GRASS	0	0	0	2	0	1	297

Correct classified: 1 997 Wrong classified: 103
Accuracy: 95,095 % Error: 4,905 %
Cohen's kappa (κ) 0,943

Knime always offers interesting solutions. The hard part is understanding the logical layout of the components. But once we understand the ideas, the pattern seems clear.

6 Conclusion

The first objective of this tutorial is to provide a practical touch to course material dedicated to the ensemble techniques that I have written [lately](#). I compare the specific libraries for R and Python, but also the tools provided by Tanagra and Knime. In the end, at least as regards the “image” dataset, these approaches are particularly effective. This is also somewhat true in general. Random Forest and Boosting often offer the best results in the challenges.

7 References

Package “[adabag](#)” for R.

Package “[randomForest](#)” for R.

Scikit-learn [ensemble methods](#) for Python.