1 Introduction

Implementation of the "Gradient Boosting" approach under R and Python.

This tutorial follows the course material devoted to the "Gradient Boosting" (GBM, 2016) to which we are referring constantly in this document. It also comes in addition to the supports and tutorials for Bagging, Random Forest and Boosting approaches (BRBC & BRBT, 2015).

The thread will be basic: after importing the data which are split into two data files (learning and testing) in advance, we build predictive models and evaluate them. The test error rate criterion is used to compare performance of various classifiers.

The question of parameters, particularly sensitive in the context of the gradient boosting, is studied. Indeed, there are many parameters, and their influence on the behavior of the classifier is considerable. Unfortunately, if we guess about the paths to explore to improve the quality of the models (more or less regularization), accurately identifying the parameters to modify and set the right values are difficult, especially because they (the various parameters) can interact with each other. Here, more than for other machine learning methods, the trial and error strategy takes a lot of importance.

We use R and Python with their appropriate packages.

2 Dataset and evaluation approach

2.1 Dataset

We use the "Optical Recognition of Handwritten Digits"[1] dataset from the UCI Repository. The aim is to recognize (K = 10) handwritten digits (number from 0 to 9) from (8 x 8) bitmaps (pixels numbered from 1 to 64). We have 5260 instances.

2.2 Evaluation process of the classifiers

We use only 200 instances for the training set (5420 for the test set). Thus, we have approximately 20 instances per class. A small number of training instances for 64-dimensional dataset make difficult the learning process. Overfitting may easily occur in this context. Set the right values of the parameters to combat this curse will be hard.

1 http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

3 Gradient boosting with R

3.1 Data importation and preparation

We import the train and test sets into two distinct data frames. We display the types of the variables. This information is crucial for some packages used in this tutorial.

```
#set the default directory
setwd("... votre répertoire ...")

#import the train and test sets
#into two distinct data frames
dtrain <- read.table("opt_digits_train.txt",header=T,sep="\t")
dtest <- read.table("opt_digits_test.txt",header=T,sep="\t")

#display the class of each variable
print(sapply(dtrain,class))
```

The class attribute CHIFFRE is recognized as a R "factor". The other columns are "integer" (http://www.statmethods.net/input/datatypes.html).

```
  chiffre      pix1      pix2      pix3      pix4      pix5      pix6      pix7 
 "factor" "integer" "integer" "integer" "integer" "integer" "integer" "integer" 
     pix8      pix9     pix10     pix11     pix12     pix13     pix14     pix15 
"integer" "integer" "integer" "integer" "integer" "integer" "integer" "integer" 
    pix16     pix17     pix18     pix19     pix20     pix21     pix22     pix23 
"integer" "integer" "integer" "integer" "integer" "integer" "integer" "integer" 
    pix24     pix25     pix26     pix27     pix28     pix29     pix30     pix31 
"integer" "integer" "integer" "integer" "integer" "integer" "integer" "integer" 
    pix32     pix33     pix34     pix35     pix36     pix37     pix38     pix39 
"integer" "integer" "integer" "integer" "integer" "integer" "integer" "integer" 
    pix40     pix41     pix42     pix43     pix44     pix45     pix46     pix47 
"integer" "integer" "integer" "integer" "integer" "integer" "integer" "integer" 
    pix48     pix49     pix50     pix51     pix52     pix53     pix54     pix55 
"integer" "integer" "integer" "integer" "integer" "integer" "integer" "integer" 
    pix56     pix57     pix58     pix59     pix60     pix61     pix62     pix63 
"integer" "integer" "integer" "integer" "integer" "integer" "integer" "integer" 
    pix64 
"integer" 
```

3.2 Function for the calculation of the error rate

We create a function for the calculation of the test error rate. It takes as input the observed class values and the prediction of the classifier. It computes the confusion matrix and returns the error rate expressed in percentage.

```
#function for the calculation of the error rate
```

```
err_rate <- function(D,prediction){
  #confusion matrix
  mc <- table(D$chiffre,prediction)
  #error rate
  #1 - sum(well classified instances) / total number of instances
  err <- 1 - sum(diag(mc))/sum(mc)
  print(paste("Error rate :",round(100*err,2),"%"))
}
```

3.3 Decision tree with the "rpart" package

We evaluate the behavior of the simple decision tree algorithm as a first step. Because most of the ensemble methods use this approach as underlying classifier, in principle, the ensemble methods should be more accurate.

3.3.1 Decision tree with the default settings of "rpart"

Learning process. We load the "rpart" package and launch the learning process with the default settings.

```
#rpart library
library(rpart)
m.tree <- rpart(chiffre ~ ., data = dtrain)
print(m.tree)
```

The tree has 13 leaves. Displaying the tree is not interesting in our context. We note however that we have approximately 200/13 ≈ 15 instances per leaf.

When we show the importance of the variables...

```
#variable importance
print(m.tree$variable.importance)
```

... we observe that the pix34 is the most important for the classification process, then come pix43, pix47, etc.

```
    pix34     pix43     pix47     pix42     pix27     pix37     pix31     pix29
27.858532 21.311758 19.180948 17.730346 14.640914 14.361170 13.867778 13.447534
    pix44     pix63     pix30     pix36     pix20     pix55     pix28     pix21
11.959184 11.402576 11.182230 11.172173 10.630079 10.548857 10.476497 10.305098
    pix61     pix39     pix26     pix22      pix7     pix62     pix18     pix46
 8.305127  7.075785  6.807937  6.607682  6.589464  6.413949  6.308128  5.857559
    pix19     pix35     pix56     pix11     pix51      pix6     pix54     pix12
 5.835374  5.701288  5.295399  5.171776  5.082832  4.858586  4.808232  4.507306
    pix14     pix64     pix53     pix13      pix4     pix45     pix52      pix3
 4.275966  4.275966  3.934008  3.878055  3.778900  3.660975  3.660975  3.576715
    pix38     pix59     pix10     pix60
 3.059784  2.699214  1.733579  1.417344
```

Some variables do not appear in the list, especially because they have only a unique value. They are not relevant in statistical analysis process. We can detect them easily when we calculate the standard deviation of the variables (comparing the min and the max values allows also to detect the variables with a unique value).

```
> sapply(dtrain[,2:65],sd)
      pix1       pix2       pix3       pix4       pix5       pix6       pix7 
0.00000000 0.88760306 4.66708511 3.93641676 4.51992796 5.46428330 3.04637110 
      pix8       pix9      pix10      pix11      pix12      pix13      pix14 
0.75020935 0.00000000 2.98065792 5.22200360 3.89148019 5.00452057 5.60339990 
     pix15      pix16      pix17      pix18      pix19      pix20      pix21 
3.76300091 0.23247629 0.07071068 3.51363461 5.72045611 5.91182913 6.31093611 
     pix22      pix23      pix24      pix25      pix26      pix27      pix28 
6.00551673 3.13414892 0.07071068 0.00000000 3.17556231 6.16905676 5.83534328 
     pix29      pix30      pix31      pix32      pix33      pix34      pix35 
6.32026135 5.78697203 3.76093381 0.00000000 0.00000000 3.51933282 6.22215203 
     pix36      pix37      pix38      pix39      pix40      pix41      pix42 
6.51658880 6.15805025 6.01503769 3.53530649 0.00000000 0.14142136 2.92103358 
     pix43      pix44      pix45      pix46      pix47      pix48      pix49 
6.55076619 6.49037749 6.33289053 5.77194523 4.58353784 0.14035132 0.45165803 
     pix50      pix51      pix52      pix53      pix54      pix55      pix56 
2.21560818 5.63367477 4.67362965 5.06511371 6.12393771 5.31117150 0.86651848 
     pix57      pix58      pix59      pix60      pix61      pix62      pix63 
0.07071068 0.87969844 5.01805534 3.92796634 4.72222373 5.77232176 4.10929820 
     pix64 
1.76919480 
```

Evaluation of the classifier. We apply the classifier on the test set. We compare the predictions with the observed values of the class attribute. We use the **err_rate()** function defined above (section 3.2).

```
#prediction
y.tree <- predict(m.tree, newdata = dtest, type = "class")

#error rate
err_rate(dtest,y.tree)
```

The test error rate is **36.59 %**. This is our baseline error rate. We should always do better with the methods that follow. We observe that the error rate of the default classifier is (1 - 1/K = 1 - 1/10 = 90%) because we have a balanced dataset. Even if the tree is not great on our dataset, it performs a relevant classification process, significantly better than the default classifier.

3.3.2 Modifying the settings of "rpart"

To check if the overfitting phenomenon can occur on our dataset, we create a tree deliberately oversized. If the error rate deteriorates, it means that we must select more regularized approach because, for instance, we have a noisy data. Otherwise, if the error rate is improved, we may think that we have non-noisy data. Complex classifiers with low bias should be more accurate.

We modify the settings of the tree algorithm, we decrease the minsplit and minbucket parameters. And we deactivate the cp parameter (cp = 0).

```
#new settings for the tree algorithm
param.tree.2 <- rpart.control(minsplit=5,minbucket=2,cp=0)

#learning process with the new settings
m.tree.2 <- rpart(chiffre ~ ., data = dtrain, control = param.tree.2)
print(m.tree.2)

#prediction
y.tree.2 <- predict(m.tree.2, newdata = dtest, type = "class")

#test error rate
err_rate(dtest,y.tree.2)
```

The tree has 23 leaves (8.7 instances per leaf in average). The error rate is **32.8 %**. Apparently, overfitting is not the first concern here. We can create classifier variants that fit closely the learning data. We will think about it when we have to set the parameters of the other methods that we will examine in the following sections.

3.4 Boosting with the "adabag" package

Boosting is the second baseline learning approach (BRBC). Indeed, "Gradient boosting" is a kind of generalization of the standard boosting. In the first place, I thought that gradient boosting was difficult to handle because of the high number of parameters, and that it is not usable in practice. I admit I was quite astonished when I read in recent publications that the "gradient boosting" have an excellent behavior in the data science challenges. This is a bit of what brought me to study the method closely (GBM).

To study the behavior of the boosting approach, we use the "adabag" package for R (BRBC). We reiterate the train and test steps.

```
#load the "adabag" library (must be installed first)
library(adabag)
```

```
#boosting - learning process
m.boosting <- boosting(chiffre ~ ., data = dtrain, boos = FALSE, mfinal = 100, coeflearn = 'Zhu')

#prediction
y.boosting <- predict(m.boosting, newdata = dtest)

#test error rate
err_rate(dtest,y.boosting$class)
```

We create ''mfinal = 100'' decision trees[2]. The approach is based on the SAMME algorithm ''coeflearn = Zhu'' (BRBC).

The test error rate is **10.83 %**. Compared with the standard decision tree learning algorithm, the improvement is dramatic. The test error rate is divided by a factor of 3. This shows, if necessary, that the ensemble methods are often very effective in most situations. The only criticism that can be addressed to them is the lack of an explicit model for the interpretation.

An open issue is the number of tree to create in the ensemble model (we use the default setting mfinal = 100 here). We study this issue when we analyze the behavior of the gradient boosting below.

3.5 Gradient boosting with the "gbm" package

We examine the "gbm" package in this section. The description of the approach is available online (Ridgeway, 2007).

3.5.1 Default settings

Learning process. As a first step, we launch the algorithm with the default settings. We specify nevertheless the "multinomial" distribution which correspond to the loss function "multinomial deviance" (GBM, page 13), so that the procedure interprets properly the target attribute with K = 10 classes.

```
#package "gbm"
library(gbm)

#learning process
#default settings
#loss function: multinomial deviance
```

2 We use systematically 100 trees for ensemble methods in this tutorial.

```
m.gbm.default <- gbm(chiffre ~ ., data = dtrain, distribution="multinomial")

#print the results
print(m.gbm.default)
```

R displays the following results.

```
gbm(formula = chiffre ~ ., distribution = "multinomial", data = dtrain)
A gradient boosted model with multinomial loss function.
100 iterations were performed.
There were 64 predictors of which 29 had non-zero influence.
```

29 descriptors among the 64 available variables are relevant for the classification process. It means that 35 variables have no influence! It is more restrictive than the 6 irrelevant variables detected for the decision tree (page 4). These 35 variables do not appear in any internal trees of the ensemble classifier.

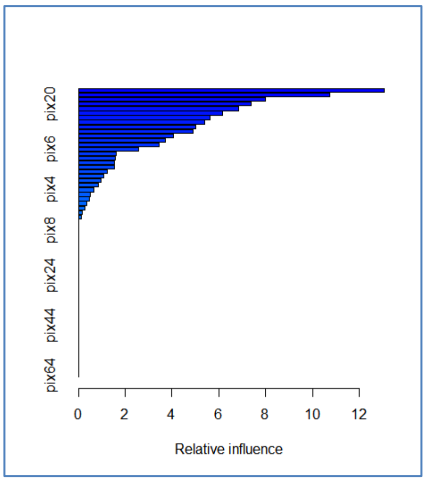
We obtain the importance of the variables with the summary() command.

```
#summary -> variable importance
print(head(summary(m.gbm.default),10))
```

Among the 10 best ones, the influence of the variable pix37 is emphasized, unlike for the decision tree algorithm (see page 3).

```
var   rel.inf
pix37 pix37 13.075552
pix61 pix61 10.739860
pix30 pix30  7.997531
pix20 pix20  7.383096
pix52 pix52  6.835593
pix34 pix34  6.167558
pix47 pix47  5.612913
pix38 pix38  5.414612
pix19 pix19  5.005820
pix29 pix29  4.895419
```

The importance of the variables can be shown with a bar graph. But we cannot distinguish the names of the variables when we have many ones.



Prediction. We launch the **predict()** procedure for the prediction on the test set.

```
#predict ==> score for each class
p.gbm.default <- predict(m.gbm.default,newdata=dtest,n.trees=m.gbm.default$n.trees)
print(head(p.gbm.default[,,1],6))
```

We must specify the number of trees to use for the prediction. We use the number returned by the learning process ($n.trees). **predict()** returns a table with, in row the instances to classify, in columns, the score for each class. We have the following values for the first 6 individuals of the test set.

```
             C0          C1          C2          C3          C4          C5
[1,] 0.49304415 -0.09812562  0.03519917 -0.06352704 -0.05290850 -0.07557961
[2,] 0.49304415 -0.09812562 -0.09573301 -0.06352704  0.15421028 -0.07243963
[3,] 0.05945927 -0.09812562 -0.09573301 -0.05673755  0.18897575 -0.07313190
[4,] 0.49304415 -0.09812562 -0.09573301 -0.05786361  0.18897575  0.01457892
[5,] 0.49304415 -0.09812562 -0.07952674 -0.05475276  0.20477354 -0.04971930
[6,] 0.05945927  0.03398715 -0.09573301 -0.08488862  0.08093326 -0.05908508
              C6         C7          C8           C9
[1,] -0.09706824 -0.1033490 -0.02731614 -0.072639129
[2,] -0.09706824 -0.1033490 -0.08116849 -0.072639129
[3,] -0.09706824  0.1226494 -0.08427735 -0.072639129
[4,] -0.09706824 -0.1033490 -0.07925893 -0.024321297
[5,]  0.14387216 -0.1004051 -0.07428739 -0.006123564
[6,] -0.09706824 -0.1033490 -0.07104859  0.303982942
```

For each row (instance to classify), the prediction corresponds to the column with the highest value. We use the following command to identify the prediction for each instance:

```
#transform the score in a prediction for each instance
y.gbm.default <- factor(levels(dtrain$chiffre)[apply(p.gbm.default[,,1],1,which.max)])
```

Here are the details of the command:

- apply() is applied to each row of the matrix provided by the predict() command. It searches the number of the column with the highest score.
- levels(dtrain$chiffre) returns the list of the values of the target attribute CHIFFRE i.e. {C0, C1, C2, …, C9}.
- By means of the R's replication mechanism, apply() returns a vector of strings containing the values {'C0'…'C9'}.
- factor() enables to transform the vector of strings into a factor data type.

Finally, we call the error rate function defined previously.

```
#test error rate
err_rate(dtest,y.gbm.default)
```

Unfortunately, the test error rate is **35.9%**. At the same level than the simple decision tree algorithm. This is a disappointing result. Before to point out the inefficiency of the "gbm" package, it would be time to take a close look at the parameters used by the procedure.

3.5.2 Modifying the parameters

We obtain the properties of a R object with the **attributes()** command.

```
#displaying the properties of the object provided by the learning process
print(attributes(m.gbm.default))
```

We have:

```
$names
 [1] "initF"             "fit"               "train.error"
 [4] "valid.error"       "oobag.improve"     "trees"
 [7] "c.splits"          "bag.fraction"      "distribution"
[10] "interaction.depth" "n.minobsinnode"    "num.classes"
[13] "n.trees"           "nTrain"            "train.fraction"
[16] "response.name"     "shrinkage"         "var.levels"
[19] "var.monotone"      "var.names"         "var.type"
[22] "verbose"           "classes"           "estimator"
[25] "data"              "Terms"             "cv.folds"
[28] "call"              "m"

$class
[1] "gbm"
```

Some of them seem interesting regarding what we know about the parameters of the gradient boosting methods (GBM, page 23). We display them.

```
#number of trees = 100
print(m.gbm.default$n.trees)
```

```
#depth of the trees = 1. By default, decision stump are used!!!
print(m.gbm.default$interaction.depth)

#shrinkage parameter = 0.001, very small correction applied
#for each tree learned (GBM, page 16)
print(m.gbm.default$shrinkage)

#proportion of instances selected randomly = 0.5
#for the learning of the trees at each step (GBM, page 16)
print(m.gbm.default$bag.fraction)
```

The key information that we observe that the procedure uses by default decision stumps as underlying classifier. For a multiclass problem, with K = 10, it is not appropriate. Each single tree is very bad. We are in an underfitting situation i.e. the learning set is under-exploited.

We modify the settings in order to build deeper individual trees (interaction.depth = 6). We improve also the speed of the convergence by emphasizing the correction for each individual tree (shrinkage = 0.1) because we do not worry about overfitting here (see section 3.3.2).

```
#modifying the settings of the learning algorithm
m.gbm.2 <- gbm(chiffre ~ ., data = dtrain, distribution="multinomial",interaction.depth=6,shrinkage=0.1)

#displaying the results
print(m.gbm.2)

#prediction
p.gbm.2 <- predict(m.gbm.2,newdata=dtest,n.trees=m.gbm.2$n.trees)
y.gbm.2 <- factor(levels(dtrain$chiffre)[apply(p.gbm.2[,,1],1,which.max)])

#test error rate
err_rate(dtest,y.gbm.2)
```

We were well advised since the test error rate is now **12.47 %[3].** We observe two essential facts:

- If we let 'shrinkage' option at 0.0001, the test error rate would have been to 24.21%. It would have been necessary to increase the number of trees to achieve satisfactory results again (Ridgeway, 2007 ; section 3.2).

3 Because there is a random part in the algorithm because of the ' bag.fraction < 1 ', we do not have exactly the same results at each execution of the procedure. With the option '' bag.fraction = 1 '', the algorithm becomes deterministic and the error rate is (therefore systematically) 14.26%.

- "Standard" boosting algorithm achieve error rate of 10.83% (the difference is significant on 5420 instances). But we do not need to manipulate finely the parameters. This relativizes the results.

Of course, by finely adjusting the gradient boosting parameters, we should achieve the same level of performance as the "standard" boosting here. But we need to be able to identify the right values for the right parameters. This is not obvious.

3.6 Gradient boosting with the "xgboost" package

We examine the "xgboost" package for R in this section. Some tutorials (ex. 1, 2) allow you to learn how to use it. A technical documentation is also available.

3.6.1 Learning process with default settings

A data preparation is needed before we can use the learning function of the package. We must transform the data frame into matrix type, and the vectors must be of the type numeric ("xgboost" does not take integer vectors).

```
#convert the descriptors in numeric data type (page 2)
#'-1' because the 1st column is the target attribute
#the list of vectors is transformed into a data frame
XTrain <- data.frame(lapply(dtrain[,-1],as.numeric))

#the data frame is transformed into a matrix
XTrain <- as.matrix(XTrain)

#recode the target attribute ("factor" type) into a numeric vector
#the values (1, 2, …, 10) are transformed into (0, 1, 2, …, 9)
yTrain <- unclass(dtrain$chiffre)-1
```

We can launch the learning process by asking 100 trees. We must specify the number of distinct values for the class attribute (num_class = 10). The softmax function is the loss function (GBM, page 13).

The test set must be transformed also to ensure that the predict function operates properly. We add the value 1 to the predicted values, defined on the range (0, ..., 9), so that the comparison with the observed values of the test set is possible.

```
#package xgboost
library(xgboost)

#learning process with the default settings (eta=0.3, max.depth=6)
m.xg.def <- xgboost(data=XTrain,label=yTrain,objective="multi:softmax",num_class=10,nrounds=100)
```

```
#preparation of the descriptors of the test set
XTest <- data.frame(lapply(dtest[,-1],as.numeric))
XTest <- as.matrix(XTest)

#prediction
y.xg.def <- predict(m.xg.def,newdata=XTest)+1

#test error rate
err_rate(dtest,y.xg.def)
```

The test error rate is **15.5 %**. Here also the results are disappointing compared with the standard boosting (10.83 %, section 3.4). They are even more because, by reading carefully the documentation, the default settings seem appropriate (shrinkage, eta = 0.3; depth of the trees, max_depth = 6).

3.6.2 Modifying the settings

We are bewildered. A first solution would be to reduce the shrinkage setting to avoid excessive corrections (eta = 0.1). But this change is not enough. A second solution explores an original feature that brings the "xgboost" closer to the implementation of the Random forest method (BRB, page 22). The idea is to make a random sampling of the variables used for the construction of individual trees. This increases the diversity of trees (they are decorrelated according to the terminology of Random Forest). Their combination will be more effective[4].

We launch again the learning process with the new version of the settings.

```
# learning with the new settings
# 0.125 = SQRT(64)/64, suggested by the settings of the "random forest"
m.xg.2 <- xgboost(data=XTrain,label=yTrain,objective="multi:softmax",num_class=10,nrounds=100,eta=0.1,colsample_bytree=0.125)

#prediction
y.xg.2 <- predict(m.xg.2,newdata=XTest)+1

#test error rate
err_rate(dtest,y.xg.2)
```

The test error rate is **10.44%**. The improvement is substantial but I confess that I am mixed. Modifying the parameters without knowing exactly why is not a satisfactory approach.

[4] With the exception that the Random forest makes the sampling at each node of the tree, while "xgboost" makes the sampling before the construction of each tree.

3.6.3 Variable importance

"xgboost" provides a tool which enables to obtain the variable importance. For our part, we compare the diversification of the trees when we introduce the variable sampling (colsample_bytree = 0.125) during the learning process.

```
#learning process
m.xg.def <- xgboost (data=XTrain, label=yTrain, objective="multi:softmax", num_class=10,nrounds=100)
#variable importance (15 first)
print(head (xgb.importance (colnames(XTrain), model=m.xg.def),15))
```

```
#learning process
m.xg.2 <- xgboost (data=XTrain, label=yTrain, objective="multi:softmax", num_class=10, nrounds=100, eta=0.1, colsample_bytree=0.125)
# variable importance (15 first)
print(head (xgb.importance (colnames(XTrain), model=m.xg.2),15))
```

```
Feature       Gain
pix37  0.08741986
pix44  0.06783863
pix20  0.05909602
pix22  0.05765523
pix30  0.05586736
pix61  0.04987202
pix29  0.04899898
pix52  0.04585716
pix34  0.04168518
pix47  0.04167961
pix27  0.03714287
pix38  0.03642339
pix3   0.03033923
pix6   0.03012236
pix11  0.02646798
```

```
Feature       Gain
pix28  0.05056194
pix31  0.03659583
pix21  0.03599328
pix22  0.03565025
pix20  0.03510729
pix44  0.03322517
pix37  0.03265284
pix59  0.03157483
pix45  0.03011384
pix3   0.03010671
pix47  0.02917668
pix36  0.02911094
pix29  0.02891033
pix38  0.02856075
pix19  0.02647185
```

The influence of the variables is better distributed (less concentration of "gain" on the first variables) with the introduction of the variable sampling. This is not surprising. In this case, it allows to improve the performance of the ensemble classifier. This is the main result.

3.7 Gradient boosting with the "mboost" package

"mboost" is the third package presented in our course material (GBM, page 21).

It operates properly on our illustrative example, which handles a binary problem (in the course material). But for our dataset (K = 10 classes) in this tutorial, an error occurs when we launch the learning process.

Here are the commands used.

```
#package mboost
library(mboost)

#learning process ➔ ERREUR SESSION R
m.mb.def <- blackboost(chiffre ~ ., data = dtrain, family=Multinomial())
```

I reiterate the experiments on the IRIS dataset [data(iris) from the ''dataset'' package]. The same error occurred. Yet, by reading the documentation, it seems that the processing of multiple-class problems is possible [option family = Multinomial()]. I stopped here my investigations.

4 Gradient boosting with Python

In this section, we process our dataset with "scikit-learn" (**version 0.17.1**) package for Python. The thread is the same that the one for the various packages for R software. In addition, we examine the grid search tool which enables to detect "automatically" the best settings for machine learning algorithms. We will see if it is efficient for our dataset.

The documentation for the "gradient boosting" procedure provided by the "scikit-learn" package is available online. I think that studying carefully the parameters is of interest to understand the nature of the algorithm implemented.

4.1 Data importation and preparation

As under R, we must import the two data files (learning and test sets). Then, we perform the data preparation required by the "scikit-learn" package.

```
#modifying the default directory
import os
os.chdir("... le dossier de vos données ...")

#data importation with the "pandas" package
import pandas
dtrain = pandas.read_table("opt_digits_train.txt",sep="\t",header=0,decimal=".")

#checking the dimension : 200 obs., 65 variables
print(dtrain.shape)
```

```
#target variable – learning sample
y_app = dtrain.as_matrix()[:,0]

#input variables – learning sample
X_app = dtrain.as_matrix()[:,1:64]

#importing the test set
dtest = pandas.read_table("opt_digits_test.txt",sep="\t",header=0,decimal=".")

#5420 obs. and 65 variables
print(dtest.shape)

#transformation into a matrix data type
y_test = dtest.as_matrix()[:,0]
X_test = dtest.as_matrix()[:,1:64]
```

4.2 Function for the calculation of the error rate

"Scikit-learn" provides efficient tools from the "metrics" module. We use them by transforming the success (accuracy) rate into error rate.

```
#function for the calculation of the error rate
#confronting observed values and prediction on the test set
from sklearn import metrics
err = 1.0 - metrics.accuracy_score(y.obs.test,y.pred.test)
print(err)
```

Note: In a previous tutorial (BRBT, section 4.2), taking advantage of the fact that the signatures of the functions of scikit-Learn are homogeneous, we defined a function that takes as input the test set and the classifier. So, the function incorporates the prediction and calculation of the error rate. This is a pretty elegant solution. But, in order to be consistent with the presentation for the R software in this tutorial, I prefer to make explicitly the prediction in the main program before to calculate the error rate.

4.3 Gradient boosting with "scikit-learn" package for Python

We are ready to start the modeling process. Several steps are needed.

Instantiation of the class. We must initialize an object of the class "Gradient boosting". We display its default settings.

```
#gb is an object of the gradient boosting type
#no settings are specified ➔ the object uses the default settings
from sklearn.ensemble import GradientBoostingClassifier
gb = GradientBoostingClassifier()
```

```
#displaying the settings of the object
print(gb)
```

We obtain the following results.

```
GradientBoostingClassifier(init=None, learning_rate=0.1, loss='deviance',
              max_depth=3, max_features=None, max_leaf_nodes=None,
              min_samples_leaf=1, min_samples_split=2,
              min_weight_fraction_leaf=0.0, n_estimators=100,
              presort='auto', random_state=None, subsample=1.0, verbose=0,
              warm_start=False)
```

We note among other, by referring to the documentation.

- The loss function used (loss = 'deviance') (GBM, page 13);
- 100 trees are created (n_estimators = 100);
- The learning rate (learning rate) ν = 0.1 (GBM, page 16);
- The maximal depth of the individual trees is 'max_depth = 3'. The trees are rather small. This will protect us against overfitting (GBM, page 23), but this can result in an underfitting;
- The minimum number of instances required to split and required to be at leaf is small (min_samples_split = 2, min_samples_leaf = 1) (GBM, page 23);
- There is not a sampling mechanism of instances for the creation of each individual trees (no "stochastic gradient boosting", [GBM, page 16]) i.e. all the instances are used (subsample = 1.0).
- There is not a sampling mechanism of variables (based on the Random Forest idea) for the creation of individual trees (max_features = None).

Learning process. We launch the learning process with the **fit()** procedure

```
#learning process
gb.fit(X_app,y_app)
```

Prediction and evaluation. We perform the prediction on the test set, then we calculate the error rate.

```
#prediction on the test set
y_pred = gb.predict(X_test)
#evaluation: calculation of the error rate
#error rate = 1 – accuracy rate
from sklearn import metrics
err = 1.0 - metrics.accuracy_score(y_test,y_pred)
print(err)
```

The test error rate is **19.85 %**. Again, it is disappointing. We must refine the settings to get better results.

4.4 Grille search for "optimal" parameters

Scikit-learn provides an interesting tool to detect the "optimal" values of the parameters. To the GridSearchCV tool is passed a list of parameters with the values to test. It tries to find the best combination in cross-validation. It uses a measure of performance evaluation for that (e.g. error rate, F-Score, etc.). The positive aspect is that the test sample is never used in this process. It keeps its status of arbitrator to evaluate the performance of the resulting classifier in generalization. The negative aspect is the risk of overfitting on the learning sample (200 instances for our problem).

Let us examine the use of the tool.

```
#GridSearchCV class
#3-fold cross validation – default parameter
from sklearn.grid_search import GridSearchCV

#List of parameter-value pairs to try
parametres                                                                    =
   {"learning_rate":[0.3,0.2,0.1,0.05,0.01],"max_depth":[2,3,4,5,6],"subsample":[1.0
   ,0.8,0.5],"max_features":[None,'sqrt','log2']}

#learning algorithm
gbc = GradientBoostingClassifier()

#object (instance of the class) for the grid search
grille = GridSearchCV(estimator=gbc,param_grid=parametres,scoring="accuracy")

#launch the search on the learning set
resultats = grille.fit(X_app,y_app)

#displaying the performance results (scores)
print(resultats.grid_scores_)

#best performances (scores)
print(resultats.best_score_)

#parameters-values pair for the best performances
print(resultats.best_params_)
```

For max_features, 'sqrt' and 'log2' correspond to square root and the binary logarithm of the number of descriptors. None corresponds to "no selection".

The tool evaluates the combination i.e. 5 x 5 x 3 x 3 = 225 configurations. By default, it uses a 3-folds cross-validation for the performance evaluation. Thus, there are 675 pairs of learning-test processes. It is better to have a good machine!

The 'grid_scores_' properties returns all the results. For each combination of parameters, we have the parameters used, the mean and the standard deviation of the success rate in 3-folds cross-validation. Here are for instance the 3 first results of the table:

```
[mean:  0.78500,  std:  0.01654,  params:  {'max_features':  None,  'max_depth':  2,
'subsample':  1.0,  'learning_rate':  0.3},  mean:  0.81000,  std:  0.01305,  params:
{'max_features':  None,  'max_depth':  2,  'subsample':  0.8,  'learning_rate':  0.3},
mean:  0.77500,  std:  0.01770,  params:  {'max_features':  None,  'max_depth':  2,
'subsample': 0.5, 'learning_rate': 0.3},
```

For *{'max_features': None, 'max_depth': 2, 'subsample': 1.0, 'learning_rate': 0.3}*, the average of the success rate is *78.5 %*; etc.

The best result is not easy to perceive in these outputs. Fortunately, scikit-learn provides automatically the best scores ('best_score_' = 89,5 %) and the corresponding combination of parameters ('best_params_'):

```
{'max_features': 'log2', 'max_depth': 6, 'subsample': 1.0, 'learning_rate': 0.05}
```

Is this success rate of 89.5% (error rate = 1 – 89.5% = 10.5%) credible? We apply the best model detected by the GridSearchCV tool on the test set.

```
#prediction on the test set
#with the best classifier
ypredc = resultats.predict(X_test)

#test error rate
err_best = 1.0 - metrics.accuracy_score(y_test,ypredc)
print(err_best)
```

The test error rate is **9.0 %**. Clearly, the method is efficient. This is the best result that we obtain until now. Despite the reticence about this kind of "crude" strategies, we can consider that it is a useful alternative approach when we do not master the characteristics of the learning algorithm to use.

4.5 Random Forest

In preparing this tutorial, I had tested different approaches to identify the best solutions (settings) for the gradient boosting. I realized that the Random Forest method was the most efficient. This result has led me on the introduction of the variables sampling during the construction of individual decision trees with “xgboost” under R (section 3.6.2) and “scikit-learn” under Python (section 4.4). This process (variables sampling) plays clearly an important role in our data.

Indeed, when we perform the Random Forest algorithm on our dataset…

```
#random forest class
from sklearn.ensemble import RandomForestClassifier

#RandomForest object
rf = RandomForestClassifier(n_estimators = 100)

#learning process on the learning sample
rf.fit(X_app,y_app)

#prediction on the test set
y_pred_rf = rf.predict(X_test)

#test error rate
err = 1.0 - metrics.accuracy_score(y_test,y_pred_rf)
print(err)
```

… we obtain a test error rate of **8.24 %**. This is the best result during our experiments!

5 Conclusion

The initial goal of this tutorial is to show the implementation of the gradient boosting using R and Python with easy to use packages. The process is rather easy if we consider a standard analysis with the default parameters. "Gradient boosting" is a predictive method as any the others.

Problems begin when we want to detect the parameters-values pairs which fit with the processed data. We realize that there are many parameters, that they do not always correspond to those described in the state-of-the-art books (e.g. Hastie and al., 2009). To unambiguously identify what is implemented in the packages and the potential for tuning, the only solution is to read carefully the documentation and carry out experiments. In the case of the gradient boosting, the work on the parameters is essential. We have found that they influence heavily the performance of the subsequent classifiers.

The automatic search tools for the best settings such as the scikit-learn's GridSearchCV can be of great help. The trick is to not use them wrongly and through by blindly relying on the indications they provide. Explicitly identifying the mechanisms of success of a data mining method on our data remains the best way to ensure the reproducibility of our results. But this is not always obvious, I concede.

Finally, we note that all the ensemble methods do better than the standard decision tree learning algorithm that have served as a baseline (section **Erreur ! Source du renvoi introuvable.**).

6 References

[BRBC] "Bagging, Random Forest, Boosting (slides)", December 2015.

[BRBT] "Random Forest & Boosting with R and Python", December 2015.

[GBM] "Gradient boosting - Slides", June 2016.

Hastie T., Tibshirani R., Friedman J., « The elements of Statistical Learning - Data Mining, Inference and Prediction », Springer, 2009; chapter 10.

Natekin A., Knoll A., « Gradient boosting machines - A tutorial », in *Frontiers in NeuroRobotics*, december 2013.

Wikipedia (EN), « Gradient boosting », visited at August 2017.