# 1 Objectif

**Mise en œuvre des Random Forest et Boosting sous R et Python principalement, mais aussi avec d'autres outils (Tanagra et Knime) .**

Ce tutoriel fait suite au support de cours consacré au « Bagging, Random Forest et Boosting »[1]. Nous montrons l'implémentation de ces méthodes sur un fichier de données. Nous suivrons à peu près la même trame que dans le support, c.-à-d. nous décrivons tout d'abord la construction d'un arbre de décision, nous mesurons les performances en prédiction, puis nous voyons ce que peuvent apporter les méthodes ensemblistes. Différents aspects de ces méthodes seront mis en lumière : l'importance des variables, l'influence du paramétrage, l'impact des caractéristiques des arbres sous-jacents, etc.

Dans un premier temps, nous mettrons l'accent sur R (packages rpart, adabag et randomforest) et Python (package scikit-learn)[2]. Disposer d'un langage de programmation permet de multiplier les analyses et donc les commentaires. Evaluer l'influence du paramétrage sur les performances sera notamment très intéressant. Dans un deuxième temps, nous étudierons les fonctionnalités des logiciels qui fournissent des solutions clés en main, très simples à mettre en œuvre, plus accessibles pour les personnes rebutées par la programmation, avec Tanagra et Knime.

# 2 Données

Nous utilisons les données « Image Segmentation Data Set » du dépôt UCI Machine Learning. Elles décrivent 7 types d'images d'extérieur à partir des paramètres qui ont été extraites. Les observations ont été subdivisées en échantillons d'apprentissage (30 exemples par type d'image, soit 210 individus) et de test (300 par type).

Plutôt que de manipuler deux fichiers, nous avons réuni les observations dans le seul fichier « **image.txt** », avec une colonne supplémentaire « sample » indiquant leur d'appartenance (apprentissage [train] ou test [test]). Voici quelques lignes et colonnes de l'ensemble de données, REGION.TYPE est la variable cible :

[1] R. Rakotomalala, « Bagging, Random Forest, Boosting - Diapos », novembre 2015. Nous y ferons référence tout au long de ce document.

[2] Pour les méthodes Bagging et Random Forest, vous n'obtiendrez pas exactement les mêmes résultats que dans ce document puisqu'il y a une part d'aléatoire dans l'élaboration des modèles.

| 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 Analyse avec R

## 3.1 Importation et préparation des données

Nous importons le fichier « **image.txt** » avec les paramètres adéquats.

```
#importation les données
setwd("… répertoire des données")
image_all <- read.table("image.txt",sep="\t",dec=".",header=TRUE)
print(summary(image_all))
```

La commande **summary()** permet d'avoir un aperçu des caractéristiques des données et de s'assurer de leur intégrité.

```
   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
```

On notera entres autres que les classes sont parfaitement équilibrées (REGION_TYPE), nous disposons de 210 observations en apprentissage et 2100 en test.

Nous partitionnons les données en échantillons d'apprentissage et de test en utilisant la colonne « sample ». Nous en profitons pour évacuer cette dernière des data.frame générés.

```
#partition apprentissage et test
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))
```

Nous avons respectivement les distributions de classes suivantes :

```
> 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
```

Nous sommes parés pour lancer les analyses.

## 3.2 Fonction d'évaluation des performances

Le taux d'erreur sera utilisé pour évaluer la qualité de la prédiction. Nous écrivons une fois pour toutes une fonction à cet effet. Elle prend en entrée la variable cible observée et la prédiction d'un modèle.

```
#fonction d'évaluation
error_rate <- function(yobs,ypred){
  #matrice de confusion
  mc <- table(yobs,ypred)
  #taux d'erreur
  err <- 1.0 - sum(diag(mc))/sum(mc)
  return(err)
}
```

## 3.3 Arbre de décision

Nous utilisons la librairie « rpart » pour construire les arbres de décision (page 7), parce qu'elle est très populaire, et surtout parce qu'elle est sous-jacente aux méthodes ensemblistes que nous verrons par la suite dans R. Les instructions de paramétrage sont réutilisables. Nous disposerons ainsi d'une vue cohérente des résultats.

### 3.3.1 Arbre avec les paramètres par défaut

Nous construisons une première version des arbres avec les paramètres par défaut.

```
#arbre de décision
library(rpart)
arbre_1 <- rpart(REGION_TYPE ~ ., data = image_train)
print(arbre_1)
```

Nous obtenons un arbre avec 9 feuilles :

```
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) *
```

Lisons attentivement l'arbre :

- le symbole « * » indique les nœuds terminaux (les feuilles) de l'arborescence ;
- il y a 9 feuilles, donc 9 règles ;
- quelques variables seulement parmi les 19 disponibles ont été utilisées, certaines plusieurs fois (ex. EXGREEN_MEAN) ;
- détaillons la lecture du sommet n°34 : il comporte 29 observations (**n**), avec 3 contre-exemples (**loss**), la conclusion est CEMENT (**yval**), ce qui correspond à ≈ 90% ((29-3)/29 = 0.8966) (**yprob**) des observations présentes sur le sommet.

Nous calculons la prédiction du modèle sur l'échantillon test, puis nous la confrontons avec la variable cible observée.

```
#prédiction sur échantillon test
pred_1 <- predict(arbre_1,newdata=image_test,type="class")

#taux d'erreur
print(error_rate(image_test$REGION_TYPE,pred_1))
```

Le taux d'erreur est **12.85%**.

### 3.3.2 Decision stump

Les « decision stump »[3] sont des arbres limités à un niveau c.-à-d à une seule segmentation, avec deux feuilles lorsqu'ils (les arbres) sont binaires. C'est un non-sens dans le cadre de la construction d'un arbre prédictif unique, surtout lorsque le nombre de modalités K de la variable cible est strictement supérieur à 2 (K > 2). L'approche se justifie en revanche dans le contexte d'une technique ensembliste de type « boosting ». En effet, cette dernière réduit la composante « biais » de l'erreur. De fait, « Boosting de decision stump » est une méthode de référence reconnue dans la littérature. On se rend compte en effet que le modèle global correspond à un classifieur linéaire. Cette caractéristique apparaît clairement lorsque les prédicteurs sont tous binaires.

Dans cette section, il s'agit avant tout d'identifier les paramètres qui permettent de modifier le comportement de la procédure rpart().

```
#décision 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)
```

L'option control permet de spécifier les paramètres de l'algorithme :

- « cp » intervient en pré-élagage lors de la construction de l'arbre, une segmentation est acceptée uniquement si la réduction relative de l'indice de Gini est supérieure à « cp ». En le mettant à zéro, nous désactivons son action.
- « minsplit » indique l'effectif minimum d'un somment pour tenter une segmentation.
- « minbucket » est l'effectif d'admissibilité. Dans notre cas, nous acceptons les feuilles qui comportent au moins 1 individu. C'est le minimum que l'on puisse faire.
- « maxdepth » indique la profondeur maximal de l'arbre, sachant que la racine est au niveau 0. Avec « maxdepth = 1 », nous définissons bien un « decision stump ».

Nous obtenons l'arbre suivant :

[3] https://en.wikipedia.org/wiki/Decision_stump

```
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) *
```

Qui s'avère catastrophique en prédiction…

```
#prédiction et taux d'erreur
pred_2 <- predict(arbre_2,newdata=image_test,type="class")
print(error_rate(image_test$REGION_TYPE,pred_2))
```

… avec un taux d'erreur de **71.42%**. C'était prévisible, seule la classe SKY est correctement reconnue.

### 3.3.3 Arbre plus profond

Voyons ce qu'il en est maintenant si nous produisons un arbre très profond avec une profondeur maximale de 30 niveaux (qui correspond d'ailleurs à la valeur par défaut). Il ne faut pas que les paramètres de réduction de la pureté ou d'effectifs interfèrent. Nous les mettons au minimum.

Revoici la nouvelle séquence des traitements :

```
#arbre profond
param_deep = rpart.control(cp=0,maxdepth=30,minsplit=2,minbucket=1)
arbre_3 <- rpart(REGION_TYPE ~ ., data = image_train,control=param_deep)

#prédiction et taux d'erreur
pred_3 <- predict(arbre_3,newdata=image_test,type="class")
print(error_rate(image_test$REGION_TYPE,pred_3))
```

Nous obtenons un arbre avec 21 feuilles (l'afficher n'a pas vraiment d'intérêt), avec un taux d'erreur de **10.42%**. L'arbre profond est meilleur que le premier. Il n'y a pas de sur-apprentissage. Ce n'est pas très habituel. Cela laisse à penser que les classes sont peu bruitées. La limitation de la qualité de l'apprentissage provient du nombre d'observations dans l'échantillon d'apprentissage, facteur particulièrement déterminant s'agissant des performances des arbres de décision.

## 3.4 Bagging

Nous utilisons le package « adabag » pour implémenter la méthode bagging (page 12) sur nos données. Une description approfondie du package et des méthodes programmées a été publiée dans Journal of Statistical Software[4].

### 3.4.1 Bagging avec 20 arbres (paramètres par défaut)

Nous commençons par un bagging de 20 arbres avec les paramètres par défaut.

```
#librairie adabag
library(adabag)

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

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

#taux d'erreur
print(error_rate(image_test$REGION_TYPE,predbag_1$class))
```

Le **print()** de l'objet emmène une profusion d'informations (ex. individus dans chaque échantillon bootstrap, prédiction, probabilités d'affectation, etc.) difficile à décrypter. Nous nous intéresserons aux éléments les plus importants dans les sections qui suivent.

Nous effectuons la prédiction sur l'échantillon test. Nous noterons que l'objet prédiction est un type complexe comprenant les classes prédites par le modèle (**$class**). Nous l'opposons aux classes observées, le taux d'erreur est de **8.86%**. C'est le meilleur résultat que nous ayons obtenu jusqu'ici.

### 3.4.2 Accès aux arbres individuels

La méthode génère une collection d'arbres, ils sont accessibles avec le champ **$trees** de l'objet résultat. Accédons au premier arbre généré.

```
#premier arbre
print(bag_1$trees[[1]])
```

Nous avons :

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

```
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) *
```

L'échantillon bootstrap est composé de 210 observations (n = 210 dans la sortie de R ci-dessus). Mais, de l'échantillon d'apprentissage initial, certains se répètent, d'autres sont absents. C'est pour cette raison que nous obtenons un arbre différent de celui élaboré avec rpart() sur l'échantillon d'apprentissage, pourtant basé sur les mêmes paramètres par défaut (section 3.3.1).

### 3.4.3 Importance des variables

Il est impossible d'analyser la multitude d'arbres pour évaluer l'influence des variables prédictives dans la modélisation. L'outil « importance des variables » permet de pallier cet inconvénient. Nous les affichons par ordre décroissant d'importance ici :

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

Nous avons :

```
      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
```

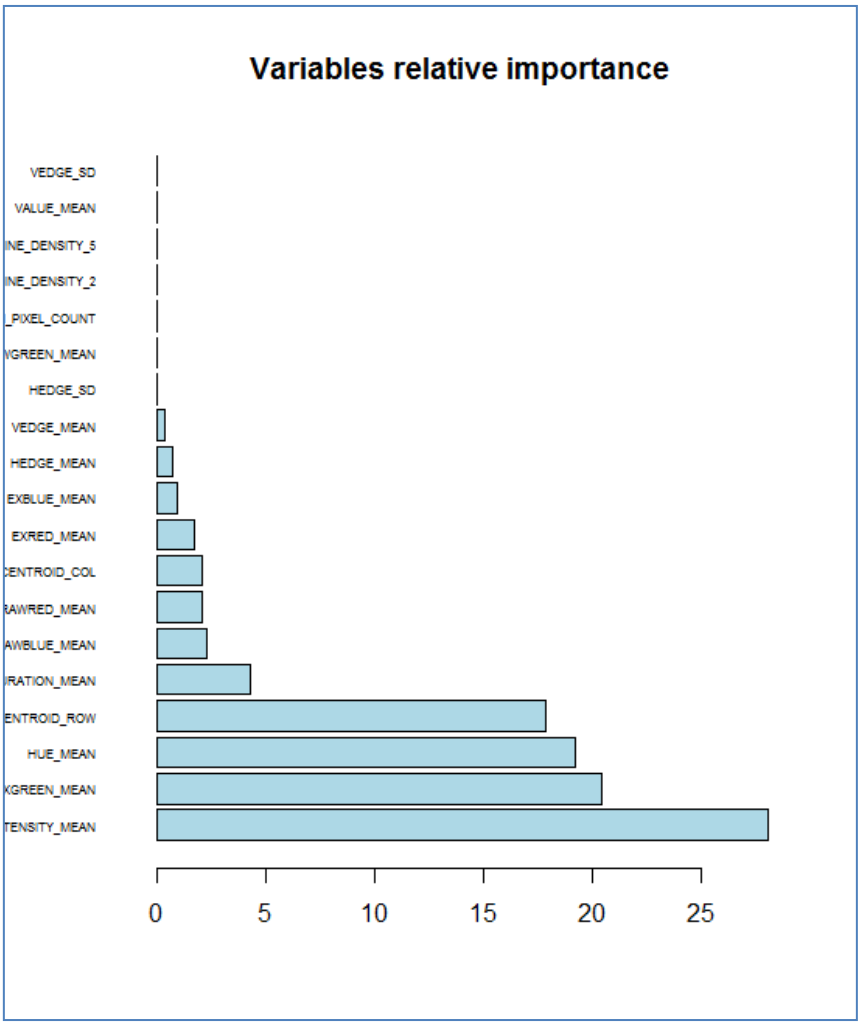
```
            HEDGE_SD        RAWGREEN_MEAN   REGION_PIXEL_COUNT SHORT_LINE_DENSITY_2
           0.0000000            0.0000000            0.0000000            0.0000000
SHORT_LINE_DENSITY_5           VALUE_MEAN             VEDGE_SD
           0.0000000            0.0000000            0.0000000
```

INTENSITY_MEAN est la variable la plus intéressante dans le sens où elle a induit la somme de réduction d'impureté (page 10) la plus élevée dans les arbres pour lesquels elle est apparue. HEDGE_SD à VEDGE_SD n'apparaissent dans aucun arbre. Leur influence est nulle (nous reviendrons sur ce commentaire plus loin, section 3.4.4).

Une sortie graphique est disponible avec la commande **importanceplot()** :

```
#sortie graphique
importanceplot(bag_1,cex.names=0.5,horiz=TRUE)
```

On note surtout un fort décalage entre la première, les 3 suivantes, puis les autres.



### 3.4.4 Commentaire sur le calcul de l'importance

J'avais un doute sur le mode de calcul de l'importance d'adabag. En effet, en accord avec la méthodologie CART[5], rpart propose une mesure d'importance qui quantifie l'influence d'une variable, même si elle n'apparaît pas dans l'arbre. Elle est basée sur le mécanisme des divisions suppléantes (surrogate split)[6]. Je me demandais si adabag ne procédait pas simplement à une somme des valeurs fournies par rpart.

J'ai donc élaboré un bagging avec un seul arbre que j'ai inspecté.

```
#bagging avec un seul arbre
bag_seul <- bagging(REGION_TYPE ~ ., data = image_train,mfinal=1)
#l'arbre
print(bag_seul$trees[[1]])
#importance
print(bag_seul$importance)
```

Dans l'arbre…

```
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) *
```

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

[6] T. Therneau, E. Atkinson, « An Introduction to Recursive Partitioning Using RPART Routines », 2015 ; voir section 3.4, page 11.

… apparaissent les variables EXGREEN_MEAN, REGION_CENTROID_ROW, INTENSITY_MEAN, HUE_MEAN, RAWRED_MEAN.

Seules ces variables présentent une importance supérieure à 0. La palme de la variable la plus importante revient à EXGREEN_MEAN parce qu'elle intervient plusieurs fois, et surtout dès la racine.

```
         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
```

C'est confirmé, « adabag » n'exploite que les variables qui apparaissent dans les arbres lorsqu'elle additionne les importances d'un arbre à l'autre.

### 3.4.5 Modifier les caractéristiques de l'arbre

Dixit la littérature (page 14), le bagging n'agit que sur la composante variance de l'erreur, pas sur le biais. A priori, un bagging de decision stump ne devrait pas donner grand-chose ; en revanche, augmenter la taille des arbres individuels devrait améliorer le méta-modèle. Voyons ce qu'il en est.

**Bagging de decision stump**. Nous réutilisons les paramètres définis plus haut (section 3.3.2).

```
#bagging de decision stump
bag_stump <- bagging(REGION_TYPE~.,data=image_train,mfinal=20,control=param_stump)
#prédiction
predbag_stump <- predict(bag_stump,newdata = image_test)
#taux d'erreur
print(error_rate(image_test$REGION_TYPE,predbag_stump$class))
```

L'idée n'est manifestement pas très brillante avec un taux d'erreur de **85.71%**, pire que l'arbre individuel.

**Bagging avec un arbre plus profond**. On réduit le biais des arbres individuels en espérant que le mécanisme de vote va plus que compenser l'augmentation de la variance.

```
#bagging arbre plus profond
bag_deep <- bagging(REGION_TYPE~.,data=image_train,mfinal=20, control=param_deep)
```

```
#prédiction
predbag_deep <- predict(bag_deep,newdata = image_test)
#taux d'erreur
print(error_rate(image_test$REGION_TYPE,predbag_deep$class))
```

L'erreur est passée à **6.14%**, le meilleur résultat obtenu jusqu'à présent. Ce n'est qu'un exemple sur un seul fichier, il ne faut pas non plus trop s'enflammer. Il reste que nous sommes parfaitement en accord avec la théorie.

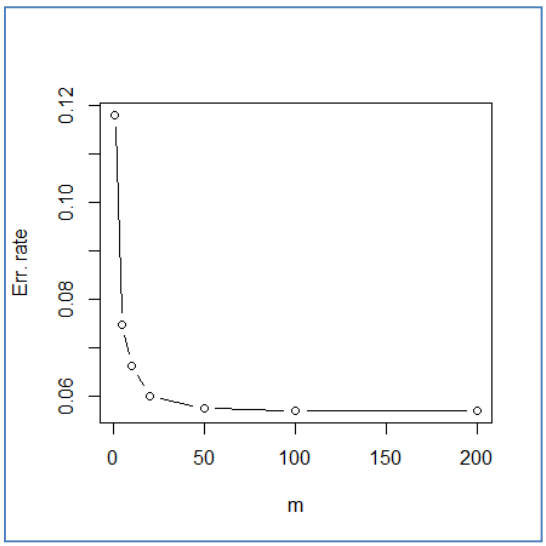
### 3.4.6 Jouer sur le nombre d'arbres

Reste la question cruciale du nombre *m* d'arbres. Nous allons le faire évoluer [m = (1, 5, 10, 20, 50, 100, 200)] et mesurer le taux d'erreur sur l'échantillon test. Chaque valeur de *m* est évaluée **20** fois pour disposer d'une certaine stabilité dans les résultats. Nous calculons alors la moyenne des taux d'erreurs obtenus.

```
#nombre m d'arbres à tester
m_a_tester <- c(1,5,10,20,50,100,200)

#apprentissage-test
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))
}
#évaluation 20 fois de chaque valeur de m
result <- replicate(20,sapply(m_a_tester,train_test_bag))
#graphique, abscisse : m, ordonnée : moyenne des erreurs pour chaque m
plot(m_a_tester,apply(result,1,mean),xlab="m",ylab="Err. rate",type="b")
```

Savoir programmer sous R prend tout son sens ici !



Le graphique relie le taux d'erreur avec le nombre de réplications. A partir de m = 50, les arbres additionnels n'améliorent pas les performances prédictives. Le taux d'erreur serait de **5.7%** pour m = 50. Nous devons prendre avec prudence cette valeur car nous avons utilisé l'échantillon test pour sélectionner le meilleur modèle. Il n'est plus vraiment impartial. Il serait plus judicieux d'utiliser une autre procédure pour la sélection de modèles. Scikit-learn de Python par exemple s'appuie sur la validation croisée (voir section 4).

## 3.5 Random Forest (Forêts aléatoires)

Par certains égards, Random Forest est une version améliorée du bagging, où le modèle sous-jacent sont forcément des arbres avec un processus de perturbation aléatoire pour les « décorréler » (page 22) (dans le bagging, le modèle sous-jacent peut être tout type de méthode bien que, dans les faits, les arbres sont quasiment systématiquement utilisés).

Nous utilisons la librairie « randomForest ». Un arbre de très grande taille est créée avec le paramétrage par défaut : lorsque *maxnodes* n'est pas spécifié, il n'y a pas de limitation sur la taille de l'arbre ; *nodesize* indique l'effectif minimal dans les feuilles (valeur par défaut 1).

### 3.5.1 Random Forest avec 20 arbres

Nous construisons et évaluons un modèle avec 20 arbres.

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

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

#erreur en test
print(error_rate(image_test$REGION_TYPE,predrf_1))
```

Le taux d'erreur est de **5.05%**. Nouveau record.

### 3.5.2 Erreur out-of-bag

Random Forest propose un mécanisme interne d'évaluation de l'erreur, sans avoir à passer par un échantillon test à part. Nous avons accès à la matrice de confusion, nous pouvons en déduire le taux d'erreur.

```
#matrice de confusion out-of-bag
print(rf_1$confusion)
#erreur out-of-bag
print(1-sum(diag(rf_1$confusion))/sum(rf_1$confusion))
```

Nous obtenons **10.3%**. L'erreur out-of-bag surestime manifestement l'erreur sur notre fichier de données. Nous avons bien fait de mettre à contribution un échantillon test distinct.

### 3.5.3 Accès aux arbres

Nous avons accès aux arbres sous-jacents.

```
#accès au 1er arbre
print(getTree(rf_1,1))
```

La présentation n'est pas vraiment 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
```

Les sommets sont numérotés, la variable de segmentation est identifiée par son numéro, un nœud intermédiaire correspond à un statut égal à 1, une feuille possède un statut égal à -1, le numéro de la classe est indiqué dans la colonne prédiction.

### 3.5.4 Importance des variables

Le calcul de l'importance des variables est conforme au bagging. Le package « randomForest » indique en plus le nombre d'apparition des variables, sachant qu'une variable peut être présente plusieurs fois dans un arbre.

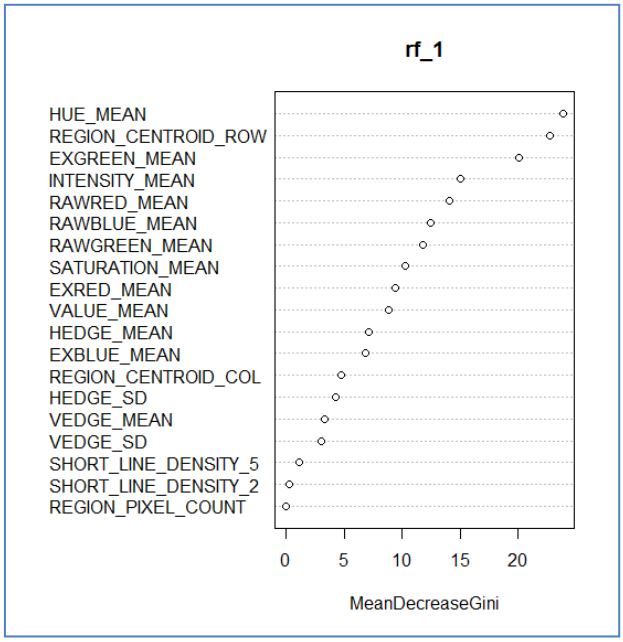
```
#apparition des variables
print(data.frame(cbind(colnames(image_train)[2:20],varUsed(rf_1))))

#importance des variables
varImpPlot(rf_1)
```

REGION_PIXEL_COUNT n'apparaît dans aucun arbre malgré le mécanisme de sélection aléatoire.

```
                    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
```

Elle présente logiquement une importance nulle.




HUE_MEAN semble la plus pertinente. Ce n'était pas le cas avec le bagging où INTENSITY_MEAN se démarquait (section 3.4.3).

### 3.5.5 Nombre d'arbres

Nous réitérons l'expérimentation permettant d'identifier le bon nombre d'arbres. Par rapport à « adabag », randomForest se révèle particulièrement véloce. Les calculs sont rapidement menés à leur terme.

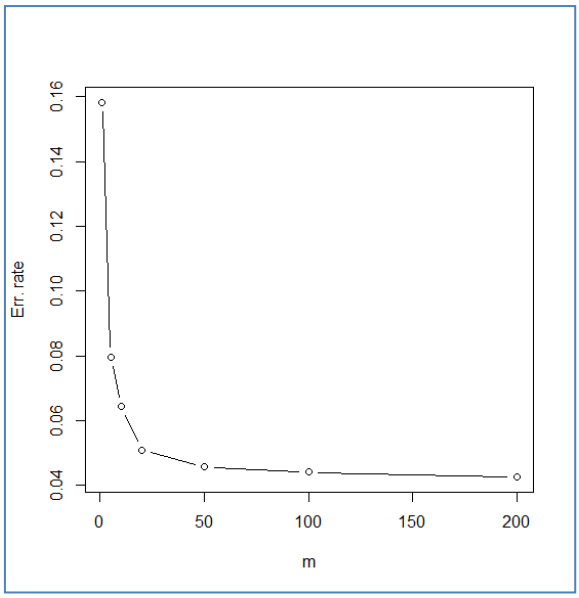
```
#nombre m d'arbres à tester
m_a_tester <- c(1,5,10,20,50,100,200)

#apprentissage-test
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))
}

#évaluation 20 fois de chaque valeur de m
result <- replicate(20,sapply(m_a_tester,train_test_rf))

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

A partir de m = 100 arbres, l'erreur baisse faiblement, mais semble vouloir baisser quand même. Plus il y en a, mieux ça vaut, il n'y a pas de phénomène de sur apprentissage.




Avec m = 200, nous avons une erreur de **4.25%**. Mais je le répète encore une fois, l'échantillon test ne joue plus vraiment le rôle d'arbitre dans cette configuration.

## 3.6 Boosting

Nous revenons au package « adabag » pour implémenter le boosting (page 28) sous R. Nous ne serons pas dépaysés. La trame est identique aux deux analyses précédentes. L'importance des variables tient compte du « poids » (page 31) de chaque arbre maintenant.

Les véritables enjeux tiennent au paramétrage des arbres sous-jacents et du nombre d'arbres. En effet, le boosting peut être sujet au sur apprentissage.

### 3.6.1 Boosting avec 20 arbres (paramétrage par défaut)

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

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

#taux d'erreur
print(error_rate(image_test$REGION_TYPE,predbo_1$class))
```

Le taux d'erreur est de **5.67%**. Meilleur que n'importe quel bagging que nous avions testé, moins bon (d'un souffle) en revanche que le random forest à 20 arbres (5.05%).

L'option « boos = FALSE » joue un rôle important. Il indique que nous utilisons la version originelle AdaBoost basée sur la pondération de tous les individus. S'il est égal à TRUE,

l'algorithme s'appuie sur un tirage avec remise, mais avec des probabilités inégales proportionnelles aux pondérations. Le résultat n'est plus déterministe dans ce cas.

### 3.6.2 Taille des arbres

**Boosting de decision stump**. Afin de dépasser la contrainte sur le taux d'erreur durant l'apprentissage qui peut être bloquante sur nos données (il devrait être inférieur à 0.5 pour pouvoir continuer le boosting), nous introduisons l'option « coeflearn = 'Zhu' » afin de mettre en œuvre la variante SAMME, plus adaptée au cas multi-classes (nombre de modalité de la cible K > 2).

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

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

#taux d'erreur
print(error_rate(image_test$REGION_TYPE,predbo_stump$class))
```

Ce n'est vraiment pas convaincant avec une erreur à **61.6%**. Le boosting permet de réduire le biais. Mais, visiblement, utiliser un boosting de decision stump dans un problème à K = 7 classes n'est pas vraiment adapté.

**Boosting sur un arbre profond**. Voyons si l'augmentation de la taille de l'arbre influe positivement sur les performances.

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

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

#taux d'erreur
print(error_rate(image_test$REGION_TYPE,predbo_deep$class))
```

Là non plus, ce n'est pas convaincant avec un taux d'erreur à **10.4%**. Nous sommes confrontés à un problème de sur apprentissage lorsque le modèle sous-jacent est trop complexe, conformément à ce que nous dit la théorie.

### 3.6.3 Nombre d'arbres

Nous réitérons l'expérimentation sur le nombre d'arbres.

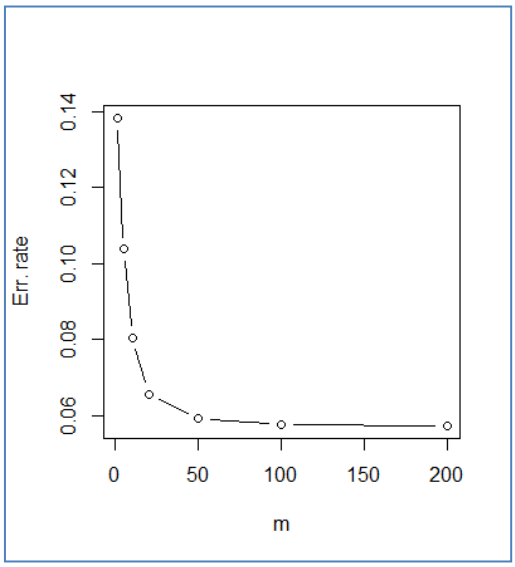
```
#nombre m d'arbres à tester
m_a_tester <- c(1,5,10,20,50,100,200)

#apprentissage-test
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))
}

#évaluation 20 fois de chaque valeur de m
result <- replicate(20,sapply(m_a_tester,train_test_boosting))

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

Nous avons :




A partir de m = 50, la réduction est très faible. Il n'y pas de phénomène de sur apprentissage sur notre jeu de données lorsqu'on augmente « m » (jusqu'à m = 200 en tous les cas).

# 4 Analyse avec Python

Dans cette section, nous souhaitons montrer l'implémentation des différentes techniques sous Python (package scikit-learn, **version 0.17**), sans chercher à reproduire toutes les expérimentations qui ont été menées sous R. Nous avions présenté le package « scikit-learn » dans un précédent tutoriel[7]. Cette seconde expérience nous permet d'aller plus loin.

[7] Tutoriel Tanagra, « Python - Machine learning avec scikit-learn », septembre 2015.

## 4.1 Importation et préparation des données

Nous importons le fichier « image.txt » et procédons à la vérification des dimensions.

```
#changer le dossier par défaut
import os
os.chdir("… votre dossier …")

#importation des données
import pandas
image_all = pandas.read_table("image.txt",sep="\t",header=0,decimal= ".")

#dimension des données
print(image_all.shape)    # (2310, 21)
```

Nous exploitons le package « pandas », « image_all » est de type DataFrame[8], très similaire à celui de R[9]. Nous obtenons un ensemble de données avec 2310 lignes et 21 colonnes.

La partition en échantillons d'apprentissage et de test relève de la même démarche que sous R, à la différence que nous extrayons des matrices et des vecteurs de type NumPy pour les analyses subséquentes avec scikit-learn.

Pour l'échantillon d'apprentissage :

```
#éch. d'apprentissage - sélectionner les lignes
image_train = image_all[image_all["sample"]=="train"]
#retirer la colonne sample
image_train = image_train.iloc[:,0:20]
#vérification
print(image_train.shape) # (210, 20)
#transformation en matrices numpy[10]
d_train = image_train.as_matrix()
y_app = d_train[:,0]
X_app = d_train[:,1:20]
```

Les variables prédictives et la cible sont dispatchés dans deux entités différentes : **X_app** est une matrice avec 210 lignes et 19 colonnes, **y_app** est un vecteur à 210 éléments.

Nous faisons de même pour l'échantillon test :

[8] pandas 0.17.1 documentation - http://pandas.pydata.org/pandas-docs/stable/index.html
[9] Voir les comparaisons : http://pandas.pydata.org/pandas-docs/stable/comparison_with_r.html
[10] **Attention (1)**, les indices commencent à 0 en Python. d_train[:,0] veut dire : toutes les lignes de d_train et la colonne n° 0 c.-à-d. la première.
**Attention (2)**, dans la notation « start:end », le premier (start) est inclus dans la sélection, le dernier (end) ne l'est pas c.à-d. dans « 1:20 », la colonne n°1 est prise en compte, et la dernière colonne est la n°19. Ainsi, d_train[ :,1 :20] veut dire : toutes les lignes de d_train, et les colonnes n°1 à 19.

```
#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 Fonction d'évaluation des performances

Comme pour R, nous définissons une fonction qui permet de calculer le taux d'erreur sur un échantillon test. Elle est très générique parce que nous n'utilisons que le seul package « scikit-learn ». Les signatures des fonctions sont les mêmes quel que soit l'algorithme de machine learning utilisé. Notre fonction prend en entrée : le modèle élaboré à partir de l'échantillon d'apprentissage et, pour l'échantillon test, le vecteur de la variable cible ainsi que la matrice des variables prédictive. Nous utilisons le module **metrics** du package sklearn pour calculer le taux de succès, converti en taux d'erreur.

```
#module pour évaluation des classifieurs
from sklearn import metrics

#fonction pour évaluation de méthodes
def error_rate(modele,y_test,X_test):
    #prediction
    y_pred = modele.predict(X_test)
    #taux d'erreur
    err = 1.0 - metrics.accuracy_score(y_test,y_pred)
    #return
    return err
#fin fonction
```

## 4.3 Arbre de décision

### 4.3.1 Instanciation et paramétrage

Le schéma est toujours le même sous scikit-learn, du moins en apprentissage supervisé :

```
#arbre de décision - importation de la classe
from sklearn.tree import DecisionTreeClassifier

#instanciation
dtree = DecisionTreeClassifier()

#paramètres de l'arbre
print(dtree)
```

Nous importons la classe associée à la méthode que nous souhaitons utiliser. Nous l'instancions (en appelant son constructeur). Nous obtenons donc un objet à partir duquel

nous lançons la modélisation sur les données d'apprentissage. L'affichage de l'objet instancié permet de visualiser les paramètres de l'algorithme et leurs valeurs par défaut.

```
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 paramètres retiennent notre attention : max_depth = None, il n'y a pas de limite dans la profondeur de l'arbre (**si on souhaite produire un arbre « decision stump », il faudrait fixer max_depth = 1, le premier sommet est au niveau 0**) ; min_samples_split = 2, un nœud est segmenté s'il a 2 observations ou plus, min_samples_leaf = 1, une feuille est validée si elle contient au moins une observation. Avec des paramètres pareils, il est prévoir que nous obtiendrons un arbre particulièrement développé.

### 4.3.2 Apprentissage

Lançons la modélisation :

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

L'opération s'est apparemment bien déroulée, aucun message n'a été envoyé. Lorsque j'ai voulu afficher l'arbre, je me suis rendu compte que l'opération était compliquée.

### 4.3.3 Visualisation de l'arbre

Il n'y a pas d'affichage par défaut textuel accessible avec **print()**. La solution la plus simple identifiée sur le web[11] consiste à générer un fichier que l'on peut transformer en graphique avec le logiciel GraphViz[12]. Il faut donc d'abord télécharger et installer l'outil en question.

Avec les commandes suivantes, un fichier « tree.dot » est bien généré sur le disque :

```
#génération de la sortie au format .dot
from sklearn import tree
tree.export_graphviz(dtree,out_file="tree.dot",feature_names=image_train.columns[1:20])
```

Chargé dans un éditeur de texte, nous observons « **tree.dot** »

```
digraph Tree {
node [shape=box] ;
0 [label="EXGREEN_MEAN <= 0.8889\ngini = 0.8571\nsamples = 210\nvalue = [30, 30,
30, 30, 30, 30, 30]\nclass = R"] ;
```

[11] http://scikit-learn.org/stable/modules/generated/sklearn.tree.export_graphviz.html
[12] http://www.graphviz.org/Download_windows.php

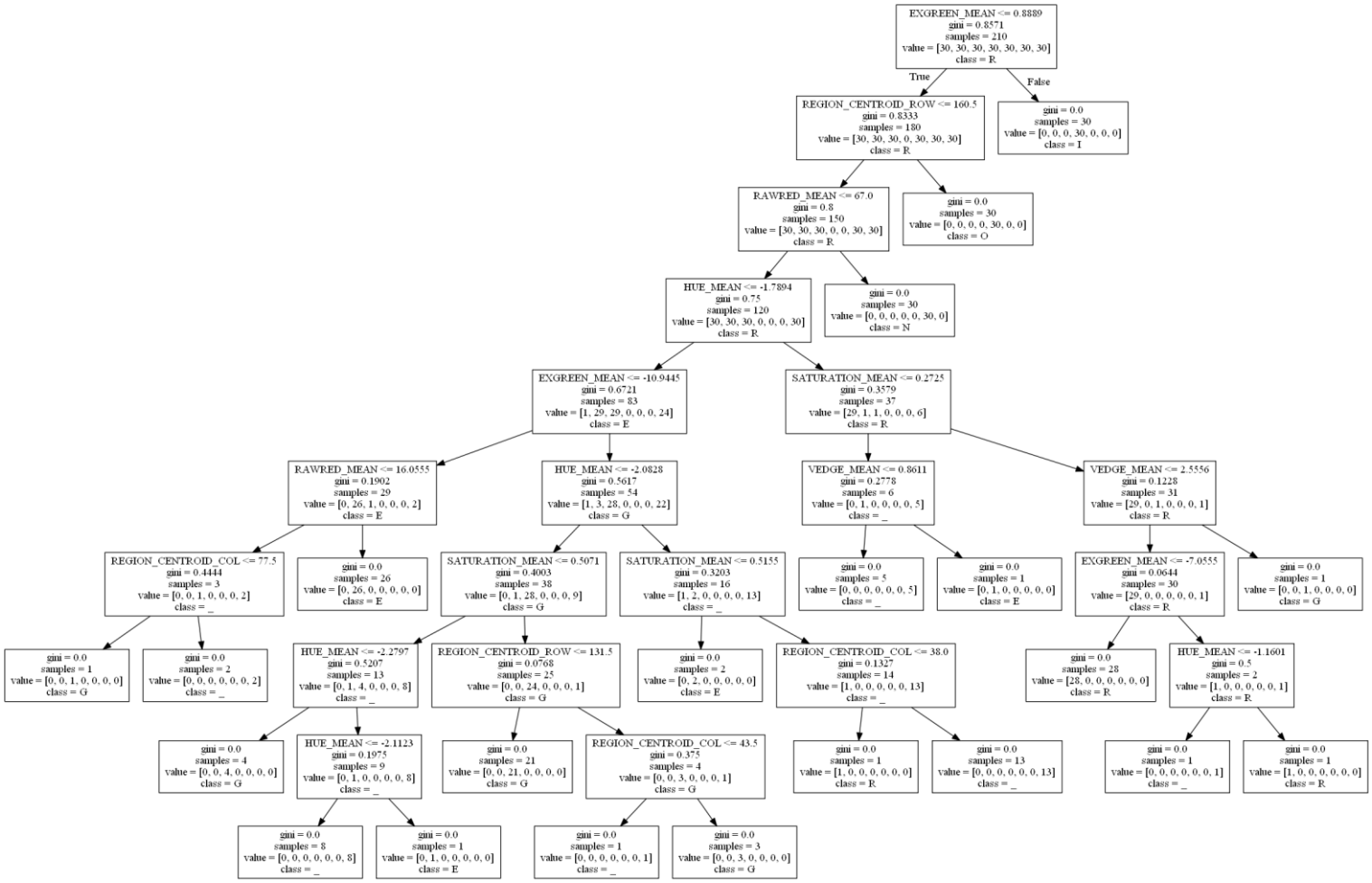
```
1 [label="REGION_CENTROID_ROW <= 160.5\ngini = 0.8333\nsamples = 180\nvalue = [30,
30, 30, 0, 30, 30, 30]\nclass = R"] ;
0 -> 1 [labeldistance=2.5, labelangle=45, headlabel="True"] ;
...
```

pas très lisible à vrai dire, d'autant plus que l'arbre est profond.

Le fichier est converti avec Graphviz à l'aide de la commande suivante (lancée dans une invite de commandes DOS),

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

« **dot** » est le nom de l'exécutable disponible après installation de GraphViz. Il faudra veiller à ce qu'il soit accessible dans le PATH de Windows. Un fichier « **tree.png** » a été généré. Nous pouvons le visualiser avec n'importe quel logiciel graphique supportant le format PNG.




Il est tellement grand que sa lecture ne présente pas beaucoup d'intérêt. La principale information ici est qu'il est possible d'obtenir une représentation graphique de l'arbre, moyennant une petite gymnastique informatique.

### 4.3.4 Importance des variables

Il est possible d'obtenir l'importance des variables. Scikit-learn ne s'intéresse qu'aux variables qui apparaissent explicitement dans l'arbre.

```
#importance des variables - 0 quand elle n'apparaît pas dans l'arbre
imp = {"VarName":image_train.columns[1:],"Importance":dtree.feature_importances_}
print(pandas.DataFrame(imp))
```

Nous avons mis les résultats dans une structure data.frame afin de pouvoir mettre en correspondance le nom de chaque variable (obtenu à l'aide de la propriété columns de l'ensemble de données d'apprentissage) avec son 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 Prédiction et performances

Il ne nous reste plus qu'à l'évaluer sur l'échantillon test. Nous utilisons la fonction error_rate développée ci-dessus (section 4.2).

```
#taux d'erreur
print(error_rate(dtree,y_test,X_test))
```

Nous obtenons un taux d'erreur de **10.38%**, du même ordre que l'arbre profond de rpart sous R (10.42%, section 3.3.3).

## 4.4 Bagging d'arbres

### 4.4.1 Apprentissage et évaluation des performances

**Instanciation**. Bagging est un meta-classifieur qui peut prendre en entrée n'importe quel algorithme d'apprentissage sous scikit-learn. Dans un premier temps, nous souhaitons élaborer un bagging de 20 arbres profonds, équivalent à celui réalisé sous R (section 3.4.5).

Il faut tout d'abord importer la classe BaggingClassifier, puis l'instancier avec l'algorithme sous-jacent souhaité, un DecisionTreeClassifier en ce qui nous concerne.

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

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

A l'affichage, nous avons à la fois les caractéristiques du méta-classifieur (en bleu), et celles de l'algorithme de base[13] (en violet) :

```
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)
```

Notons une option très intéressante. Il est possible de réaliser les calculs en parallèle en jouant sur l'option « n_jobs », pour tirer parti des possibilités des machines à processeurs multi-cœurs par exemple. Sur de très grandes bases de données, le gain en rapidité peut être conséquent.

**Apprentissage et test**. Nous construisons le classifieur sur l'échantillon d'apprentissage et nous l'évaluons sur l'échantillon test.

```
#apprentissage
baggingTree.fit(X_app,y_app)

#évaluation
print(error_rate(baggingTree,y_test,X_test))
```

[13] Traduction assez maladroite de « base classifier », j'en conviens, mais je n'en vois pas de plus simple.

Le taux d'erreur est **5.85%**, très légèrement meilleur que celui de R (6.14%, section 3.4.5).

### 4.4.2 Jouer sur le nombre d'arbres

On peut aussi bien programmer sous Python que sous R. Dans cette section, nous essayons de reproduire la démarche de recherche du nombre d'arbres optimal pour le bagging[14].

```
#fonction train-test pour un m donné
def train_test_bagging(m,X_app,y_app,X_test,y_test):
    #modélisation
    bag = BaggingClassifier(DecisionTreeClassifier(),n_estimators=m)
    #prédiction
    bag.fit(X_app,y_app)
    #taux d'erreur
    return error_rate(bag,y_test,X_test)
#fin train-test

#valeurs de m à tester
m_a_tester = [1,5,10,20,50,100,200]

#initialisation de la matrice de résultat
import numpy
result = numpy.zeros(shape=(1,7))


#répéter 20 fois l'expérience
for expe in range(20):
    #itérer sur ces valeurs
    res = [train_test_bagging(m,X_app,y_app,X_test,y_test) for m in m_a_tester]
    #le vecteur de 7 valeurs est transformé en une matrice 1 ligne et 7 colonnes
    res = numpy.asarray(res).reshape(1,7)
    #ajouter comme nouvelle ligne dans la matrice
    result = numpy.append(result,res,axis=0)
#

#retirer la première ligne de zéros
result = numpy.delete(result,0,axis=0)

#calculer les moyennes
mresult = numpy.mean(result,axis=0)
print(mresult)
```

[14] Ma maîtrise de Python est bien moindre que celle de R (pour l'instant). Je passe donc très classiquement par les boucles. Il y a peut-être (sûrement) moyen de faire mieux.

Il y a bien une double boucle : répéter 20 fois l'expérience pour chaque valeur de *m*, réaliser l'analyse pour différentes valeurs de *m*. L'analogie avec le code pour R est évidente après coup (section 3.4.6). La différence est que je ne bénéficie pas du mécanisme du **replicate()** dans mon code Python, je passe par une boucle « `for expe in range(20)` ».

Voici les erreurs observées pour chaque m :

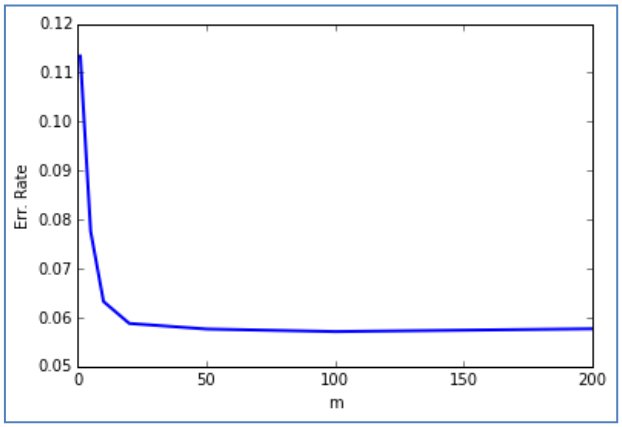
```
[ 0.11340476  0.07761905  0.0632619   0.05878571  0.05764286  0.05711905  0.05769048]
```

La visualisation est sympathique si l'on passe par un graphique de la librairie matplotlib…

```
#graphique
import matplotlib.pyplot as plt

#label des axes
plt.xlabel("m")
plt.ylabel("Err. Rate")
plt.plot(m_a_tester,mresult,linewidth=2)
```

… soit,




### 4.4.3 Grille de recherche

Savoir programmer, c'est bien. Savoir identifier les bons outils dans une librairie, c'est encore mieux. En farfouillant un peu, je me suis rendu compte que scikit-learn propose un dispositif pour tester l'efficacité des différentes valeurs de paramétrage. Ca nous dispense de programmer, c'est déjà une chose. Mais surtout, il procède par validation croisée pour évaluer la qualité des configurations. Ainsi, notre échantillon test garde bien son statut d'arbitre puisqu'il n'est pas utilisé pour détecter la meilleure configuration, mais uniquement pour mesurer le taux d'erreur de cette dernière.

Dans ce qui suit, pour les mêmes valeurs de *m* que dans la section précédente, nous demandons à Python d'évaluer le taux de succès en validation croisée puis, nous calculerons le taux d'erreur sur échantillon test de la meilleure configuration mise en lumière par l'outil.

```
#recherche du nombre "optimal" d'arbres
#outil grille de recherche
from sklearn.grid_search import GridSearchCV

#paramètres à manipuler
#le nom du paramère à manipuler est explicite
#on énumère alors les valeurs à essayer
parametres = [{"n_estimators":[1,5,10,20,50,100,200]}]

#instanciation du classifieur à inspecter
bag = BaggingClassifier(DecisionTreeClassifier())

#instanciation - ma mesure de performance sera l'accuracy : taux de succès
#rappelons que taux d'erreur = 1 - taux de succès
grid_bag = GridSearchCV(estimator=bag,param_grid=parametres,scoring="accuracy")

#lancer l'exploration
grille_bag = grid_bag.fit(X_app,y_app)

#affichage des résultats
print(grille_bag.grid_scores_)
```

Je suis assez bluffé par la rapidité de la fonction. Nous obtenons à la sortie :

```
[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}]
```

Pour m = 1, le taux de succès moyen (en validation croisée) est 81.90% ; pour m = 5, nous avons 88.57%, etc.

Il est possible d'identifier directement la meilleure configuration :

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

#meilleur paramétrage
print(grille_bag.best_params_) # {'n_estimators' : 100}
```

La configuration à m = 100 arbres est la plus intéressante. Voyons ce qu'il en est sur notre échantillon test :

```
#nouvelle évaluation
print(error_rate(grille_bag,y_test,X_test))
```

Nous obtenons un taux d'erreur de **5.71%**, meilleure (très légèrement) que notre première configuration à m = 20 arbres (section 4.4.1).

### 4.4.4 Bagging avec d'autres méthodes

Le bagging est générique sous Python, nous pouvons lui passer n'importe quel algorithme sous-jacent d'apprentissage supervisé. Mettons que l'on souhaite effectuer un bagging d'analyse discriminante linéaire (LDA), nous procéderons comme suit :

```
#analyse discriminante
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
#instanciation
bag_lda = BaggingClassifier(LinearDiscriminantAnalysis(),n_estimators=20)
print(bag_lda)
#apprentissage
bag_lda.fit(X_app,y_app)
#évaluation
print(error_rate(bag_lda,y_test,X_test))
```

L'outil nous averti que la LDA n'apprécie pas la colinéarité entre les variables, mais il poursuit quand même l'apprentissage. Le taux d'erreur est de **10.43%** pour le bagging d'analyse discriminante, à rapprocher avec la valeur de **9.57%** obtenue lorsque cette dernière est utilisée seule. Il est donc techniquement possible d'effectuer un bagging de n'importe quel classifieur. Mais le gain n'est vraiment convaincant que lorsqu'il est faiblement biaisé et à forte variance, ce qui n'est pas vraiment le cas de l'analyse discriminante.

## 4.5 Random Forest

Scikit-learn propose aussi les Random Forest. Nous procédons à une analyse très simple avec une version à 20 arbres.

```
#classe RandomForest
from sklearn.ensemble import RandomForestClassifier
#instanciation
rf = RandomForestClassifier(n_estimators=20)

#apprentissage
rf.fit(X_app,y_app)
#taux d'erreur
```

```
print(error_rate(rf,y_test,X_test))
#importance des variables…
print(rf.feature_importances_)
#avec leurs noms
imp = {"VarName":image_train.columns[1:],"Importance":rf.feature_importances_}
print(pandas.DataFrame(imp))
```

Le taux d'erreur est de **4.9%**, avec les importances de variables suivantes :

```
    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
```

En cherchant le nombre « optimal » d'arbre avec l'outil GridSearchCV, il apparaît que m = 100 est la meilleure solution avec un taux d'erreur en test de **4.76%.** Le gain, par rapport à m = 20, est négligeable pour Random Forest.

## 4.6 Boosting

Nous procédons tout aussi simplement pour le boosting, en spécifiant qu'un arbre de décision avec ses paramètres usuels jouera le rôle l'algorithme sous-jacent. **Remarque :** un decision stump est utilisé par défaut si l'option « base_estimator » est omis.

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

```
#apprentissage
ab.fit(X_app,y_app)
#evaluation
print(error_rate(ab,y_test,X_test))
```

Le taux d'erreur est de **8.9%**.

# 5 Analyse avec d'autres outils

L'intérêt de R et Python est que l'on peut programmer. Les possibilités d'analyses sont décuplées. Nous avons pu nous en apercevoir dans les deux sections précédentes. Mais encore faut-il pouvoir le faire. Au-delà des commandes, il savoir manier le langage et les principes de programmation (boucles, actions conditionnelles) pour définir des actions complexes. Cela nécessite un temps d'apprentissage dont ne disposent pas toujours les praticiens du data mining[15]. Les outils que nous présentons dans cette section, à commencer par Tanagra, permettent de reproduire simplement notre trame globale, sans avoir à écrire une seule ligne de code. Certains utilisateurs sont sensibles à cet atout.

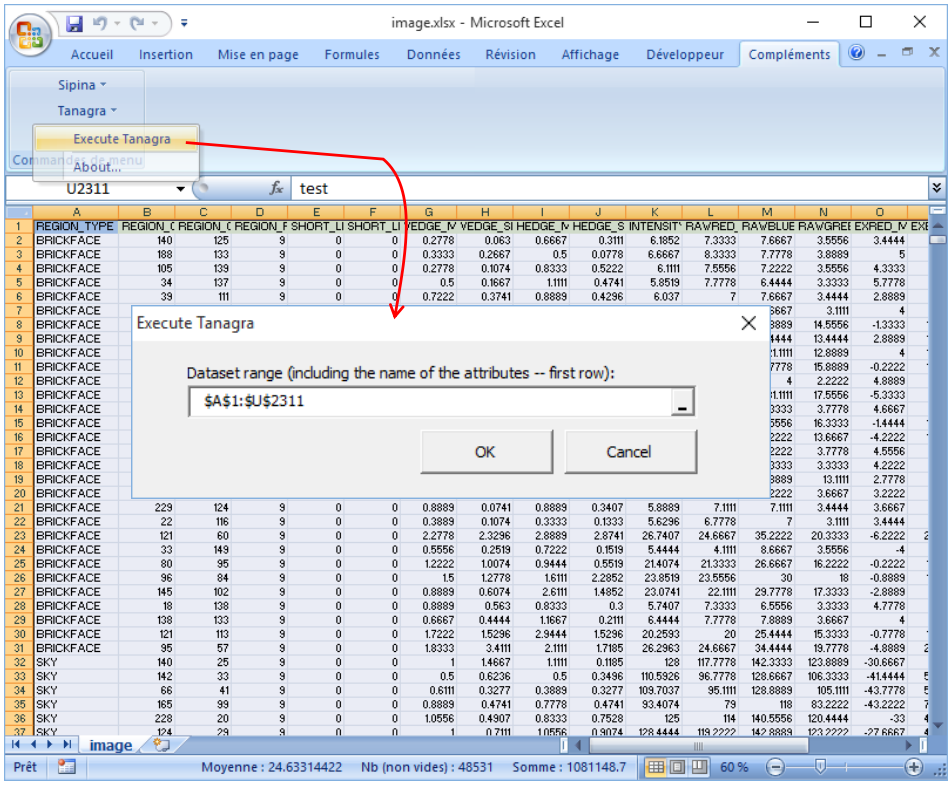
## 5.1 Analyse avec Tanagra

Tanagra dispose de plusieurs « meta-learner » qui permettent de combiner n'importe quel algorithme dans une procédure d'apprentissages répétés sur différentes versions des données. Nous disposons du bagging, de l'arcing (Breiman, 1998), du boosting, et de techniques ensemblistes ou non tenant compte des coûts de mauvais classement (Cost Sensitive Bagging, MultiCost)[16]. Random Forest est conçu comme un bagging d'un algorithme particulier d'induction (**RndTree**) où l'arbre est construit aussi grand que possible, avec le procédé de perturbation aléatoire de sélection des variables de segmentation sur les nœuds[17].

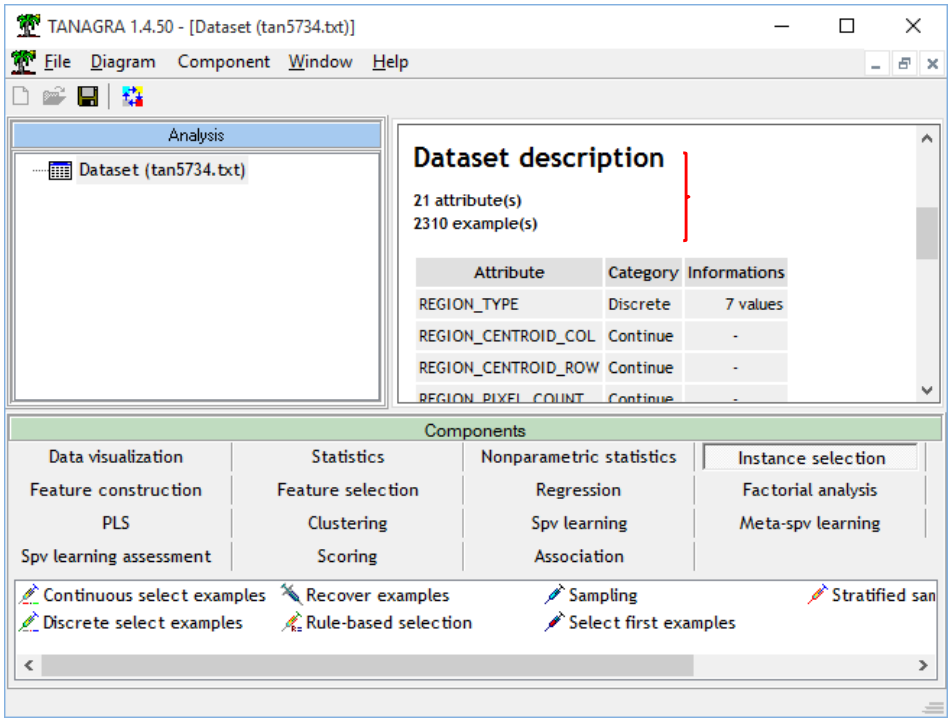
### 5.1.1 Importation et préparation des données

**Importation des données**. Nous chargeons le fichier « **image.xlsx** » dans le tableur Excel. Nous sélectionnons la plage de données puis nous activons le menu COMPLEMENTS / TANAGRA / EXECUTE installé à l'aide de l'add-in Tanagra pour Excel[18].

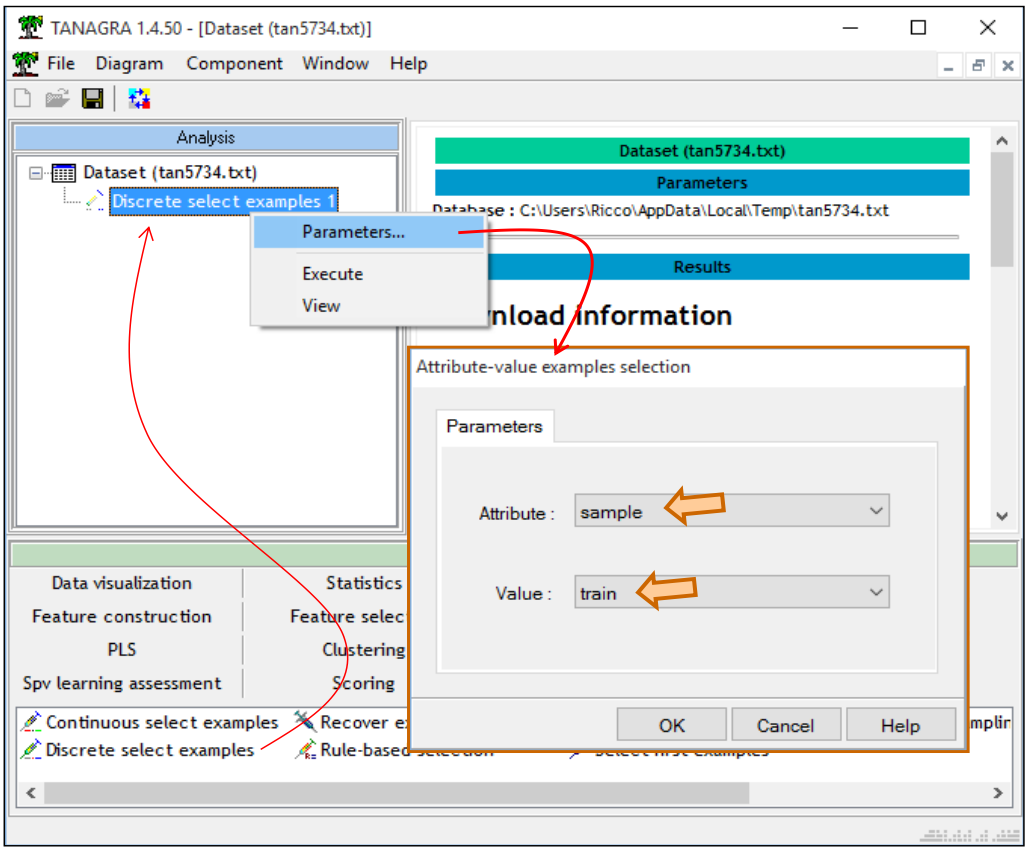
[15] C'est pour cela que j'ai réservé des heures spécifiques pour apprendre aux étudiants à programmer sous R dans mes enseignements (Programmation R).
[16] Tutoriel Tanagra, « Coûts de mauvais classement en apprentissage supervisé », janvier 2009.
[17] Tutoriel Tanagra, « Random Forests », mars 2008.
[18] Tutoriel Tanagra, « L'add-in Tanagra pour Excel 2007 et 2010 », août 2010.




Nous validons. Tanagra est automatiquement démarré et les données chargées.

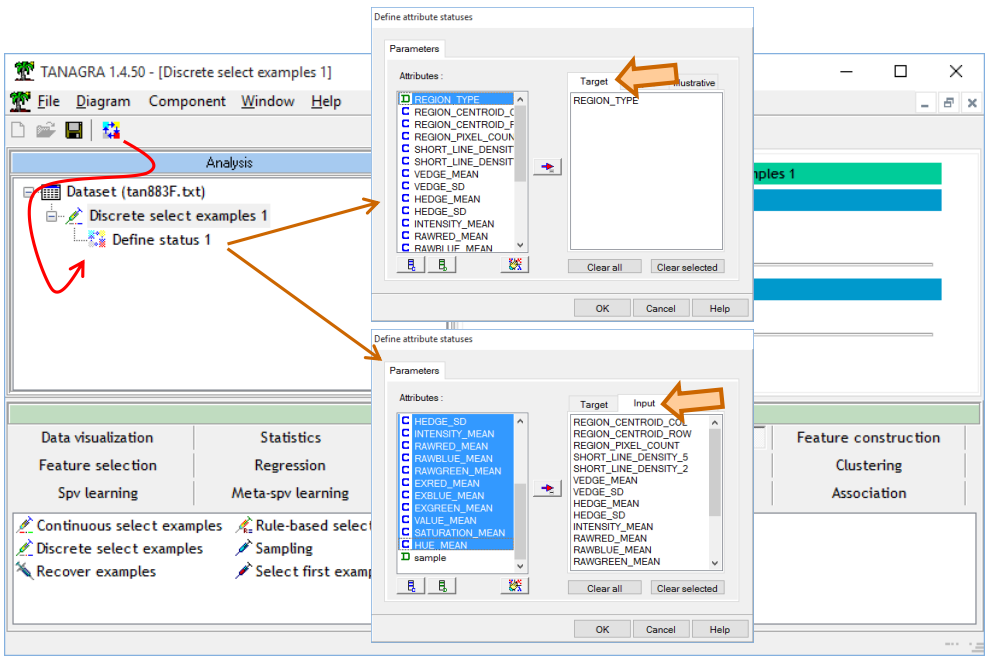



**Subdivision en échantillons d'apprentissage et de test**. Nous utilisons le composant DISCRETE SELECT EXAMPLES pour subdiviser les données selon la colonne ''sample''. Nous l'insérons dans le diagramme et nous actionnons le menu PARAMETERS.



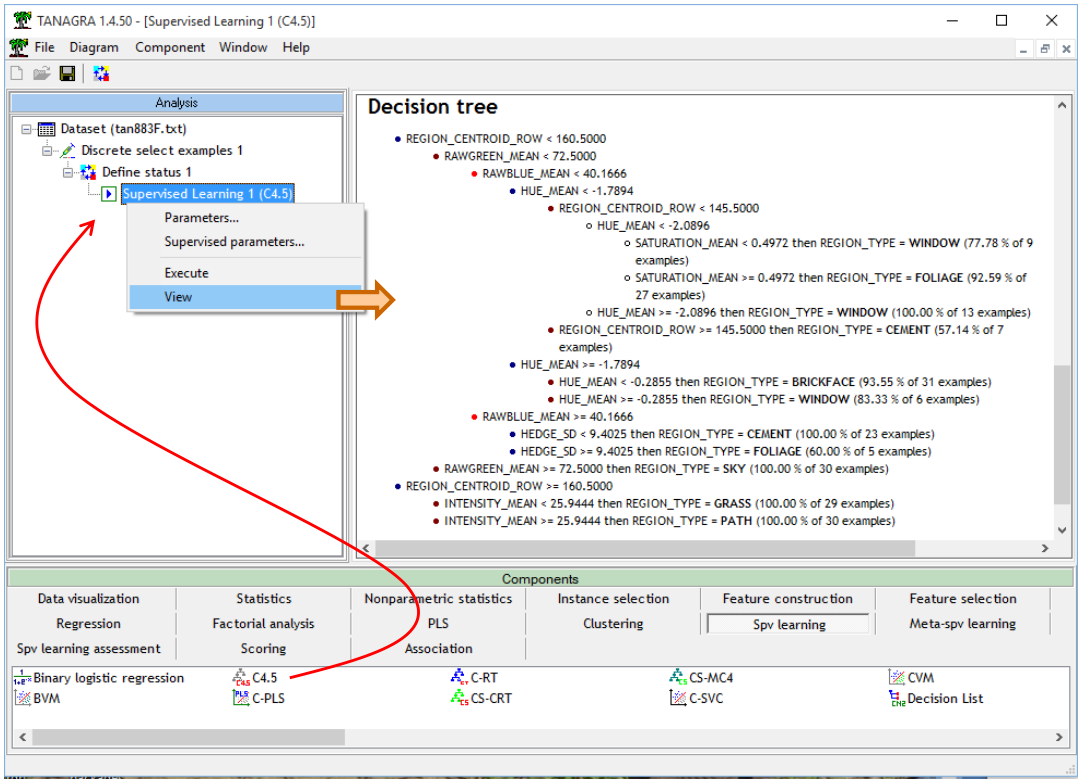

« sample = train » correspond à l'échantillon d'apprentissage, soit 210 observations.

**Rôle des variables**. Avec le composant DEFINE STATUS, nous désignons REGION_TYPE comme variable cible, les autres (REGION_CENTROID_COL…HUE_MEAN, à l'exception de ''sample'') comme prédictives.



### 5.1.2 Arbre de décision

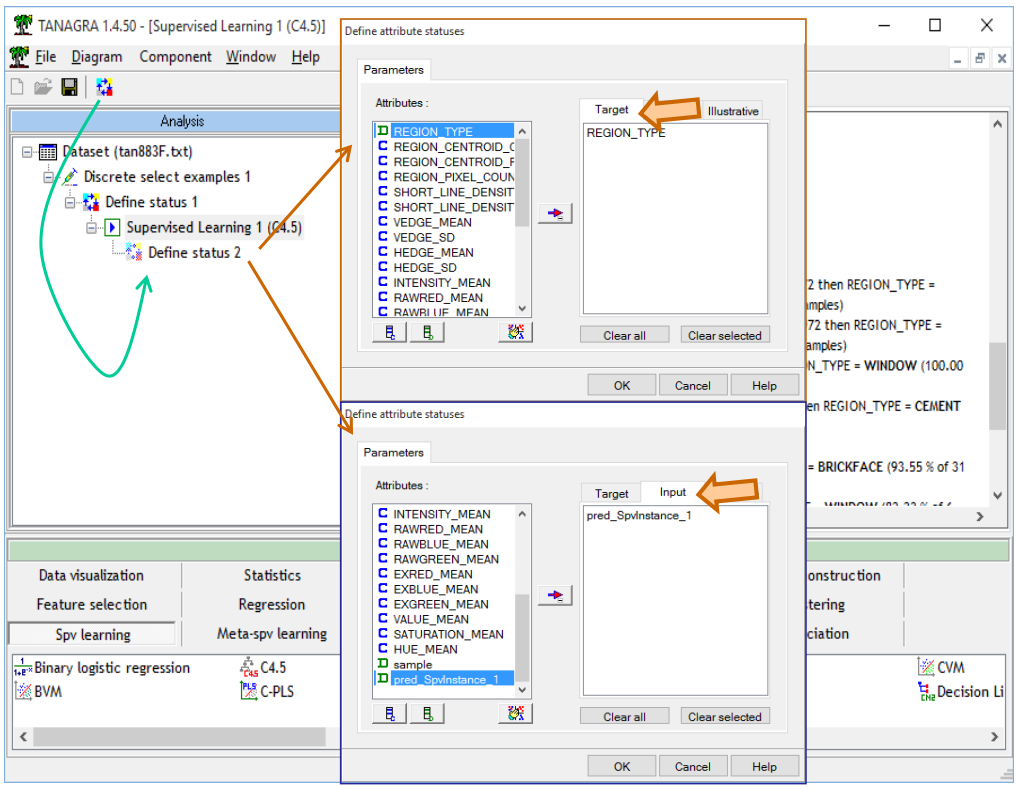
**Apprentissage**. Nous plaçons la méthode C4.5 de l'onglet SVP LEARNING à la suite de DEFINE STATUS 1. Nous actionnons le menu VIEW pour accéder aux résultats.



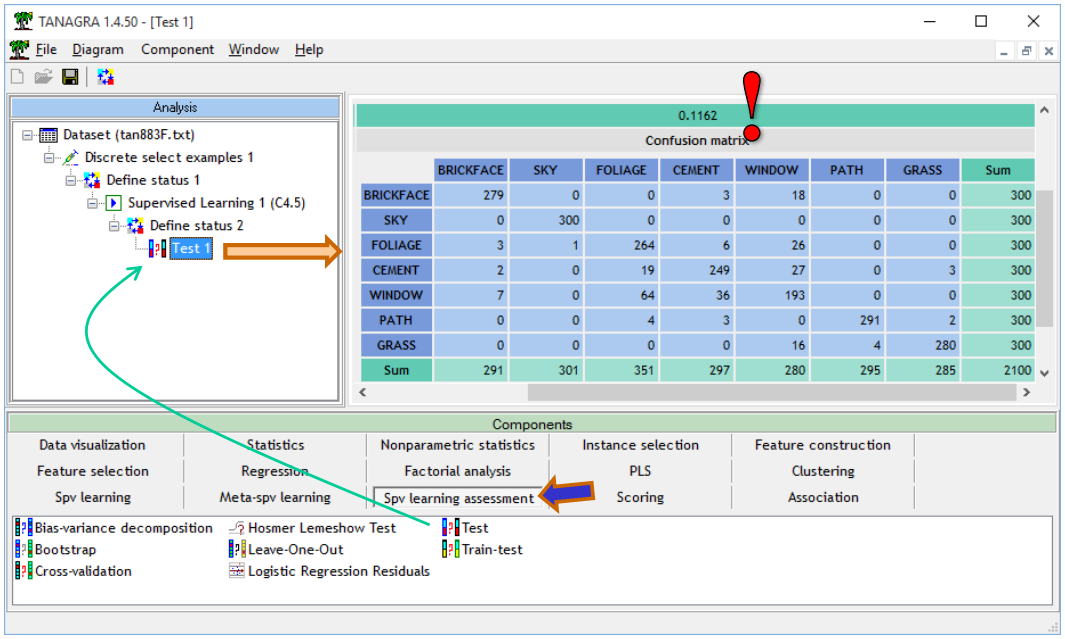

Nous obtenons un arbre à 11 feuilles (11 règles) dont voici le détail :

- 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**. Pour calculer le taux d'erreur en test, nous insérons un nouveau DEFINE STATUS à la suite de C4.5. Nous plaçons REGION_TYPE en ''target'', la prédiction du modèle (PRED_SPVINSTANCE_1) en ''input''.



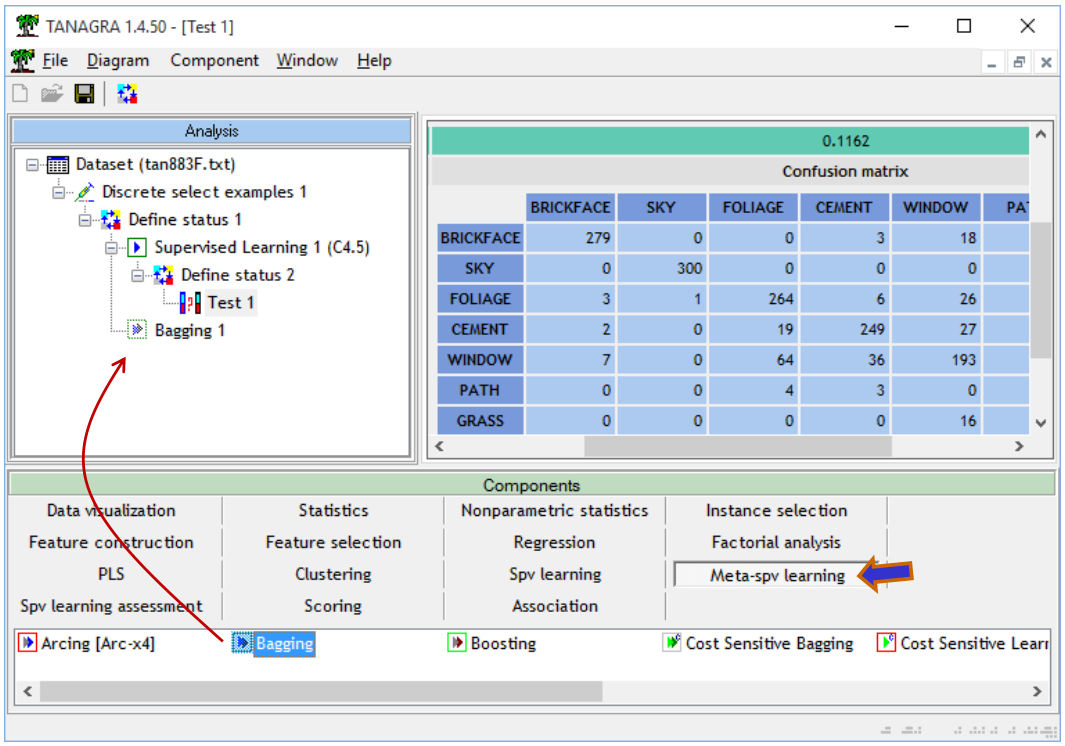

Puis le composant TEST de l'onglet SPV LEARNING ASSESSMENT qui se charge de croiser la cible et la prédiction sur l'échantillon test (les individus non sélectionnés précédemment). Nous actionnons le menu contextuel VIEW. Le taux d'erreur est de **11.62%**.



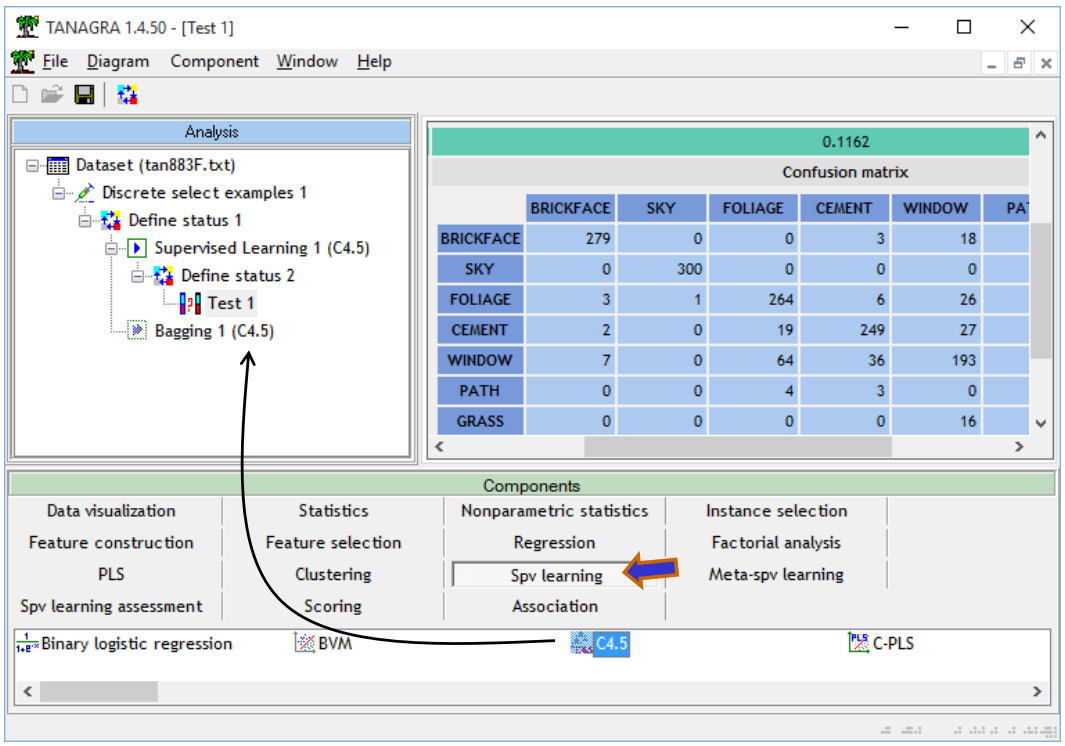

| | BRICKFACE | SKY | FOLIAGE | CEMENT | WINDOW | PATH | GRASS | Sum |
|---|---|---|---|---|---|---|---|---|
| BRICKFACE | 279 | 0 | 0 | 3 | 18 | 0 | 0 | 300 |
| SKY | 0 | 300 | 0 | 0 | 0 | 0 | 0 | 300 |
| FOLIAGE | 3 | 1 | 264 | 6 | 26 | 0 | 0 | 300 |
| CEMENT | 2 | 0 | 19 | 249 | 27 | 0 | 3 | 300 |
| WINDOW | 7 | 0 | 64 | 36 | 193 | 0 | 0 | 300 |
| PATH | 0 | 0 | 4 | 3 | 0 | 291 | 2 | 300 |
| GRASS | 0 | 0 | 0 | 0 | 16 | 4 | 280 | 300 |
| Sum | 291 | 301 | 351 | 297 | 280 | 295 | 285 | 2100 |

### 5.1.3 Bagging

Pour réaliser un Bagging d'abres C4.5, nous devons procéder en deux temps. Nous ajoutons d'abord le « meta-learner » BAGGING (onglet META-SPV LEARNING) dans le diagramme.



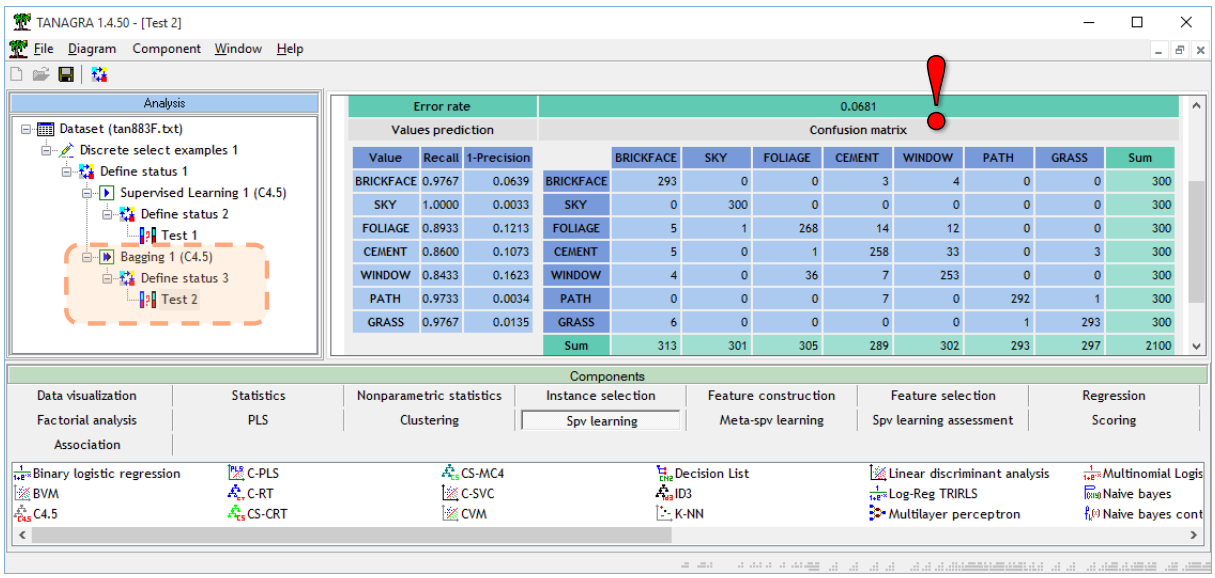

Puis, nous lui adjoignons le composant C4.5 (onglet SPV LEARNING ALGORITHM).



Nous pouvons paramétrer le bagging (menu contextuel PARAMETERS, principalement le nombre de réplications, 25 par défaut) ou l'algorithme d'apprentissage sous-jacent (menu contextuel SUPERIVSED PARAMETERES : effectif minimal sur les feuilles et niveau de confiance pour le calcul de l'erreur pessimiste en post élagage par exemple pour C4.5).

**Remarque :** Sous Tanagra, il est ainsi techniquement possible de réaliser un bagging de n'importe quelle méthode. Après, l'intérêt de la chose en termes de performances est une autre histoire comme nous avons pu le voir sous Python (bagging + analyse discriminante linéaire, section 4.4.4).

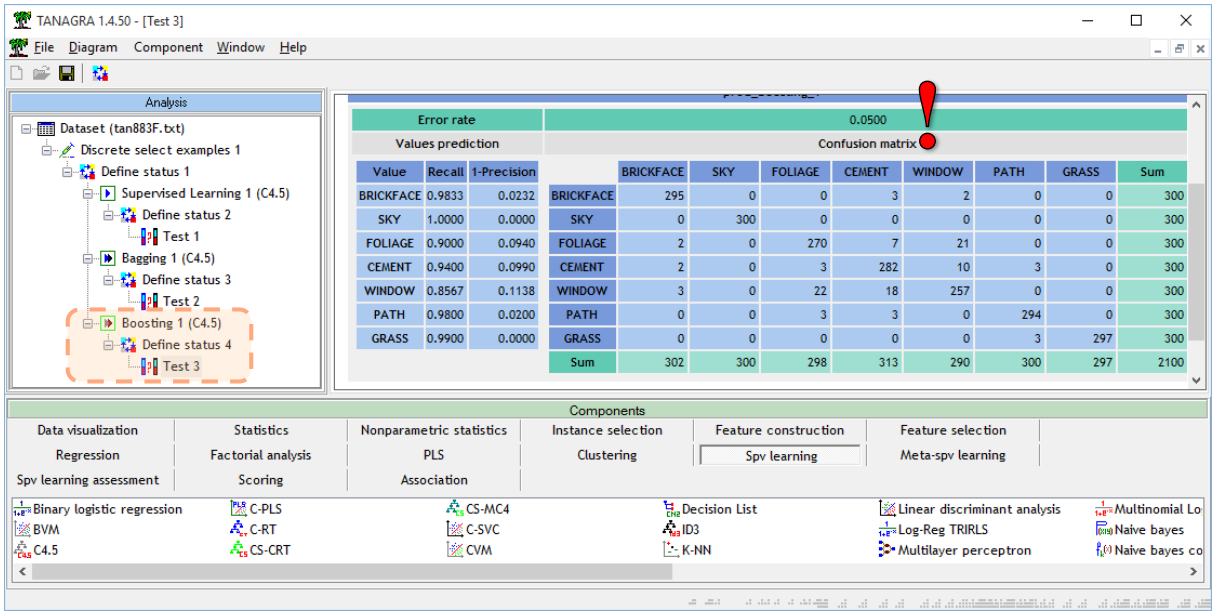
De nouveau, nous insérons le DEFINE STATUS pour croiser REGION_TYPE et PRED_BAGGING_1 généré par la modélisation, puis nous calculons le taux d'erreur en test avec le composant TEST. Il est de **6.81%**.




### 5.1.4 Boosting

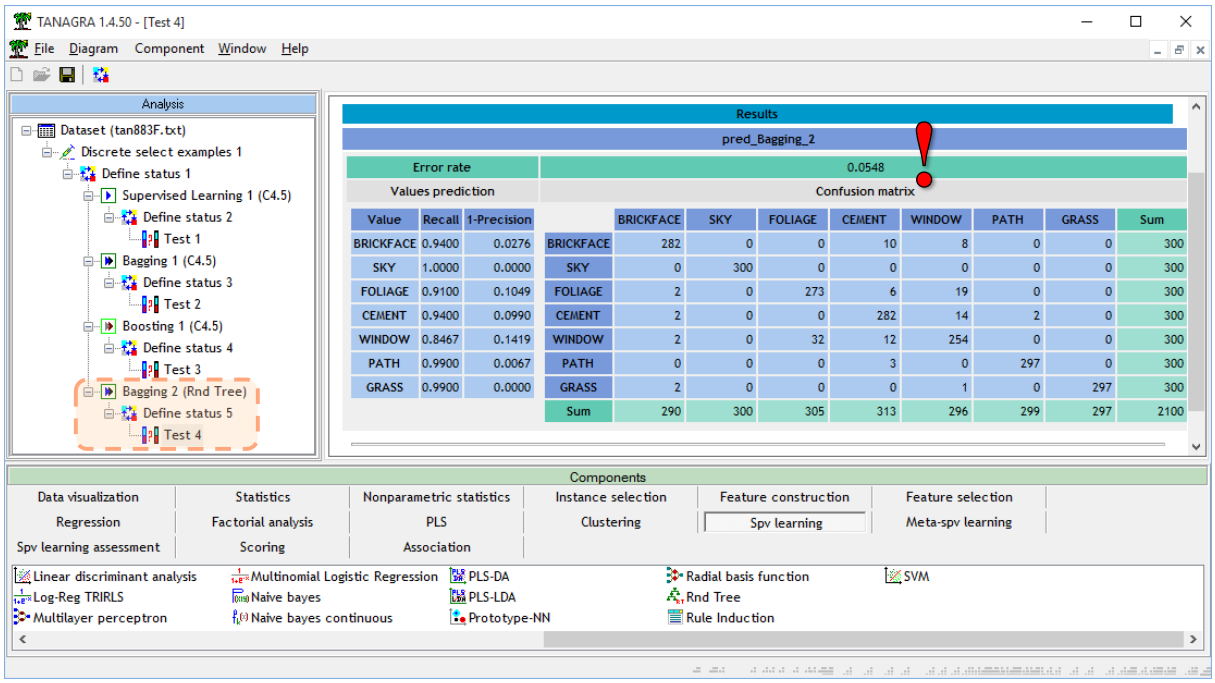
Le composant BOOSTING est disponible dans l'onglet META-SPV LEARNING. Il propose la méthode ADABOOST.M1. Il est à prévoir qu'il évolue en intégrant l'option SAMME, généralisation naturelle pour réaliser un boosting dans un cadre multi-classes (nombre de modalités de la variable cible K > 2).

L'insertion dans le diagramme se fait en deux temps toujours (BOOSTING + C4.5). Nous calculons toujours les performances en test en croisant REGION_TYPE et PRED_BOOSTING_1. Le taux d'erreur mesuré est de **5%**.




### 5.1.5 Random Forest

Pour instancier Random Forest, nous insérons le composant BAGGING (META-SPV LEARNING) auquel nous adjoignons RND TREE (onglet SPV LEARNING).

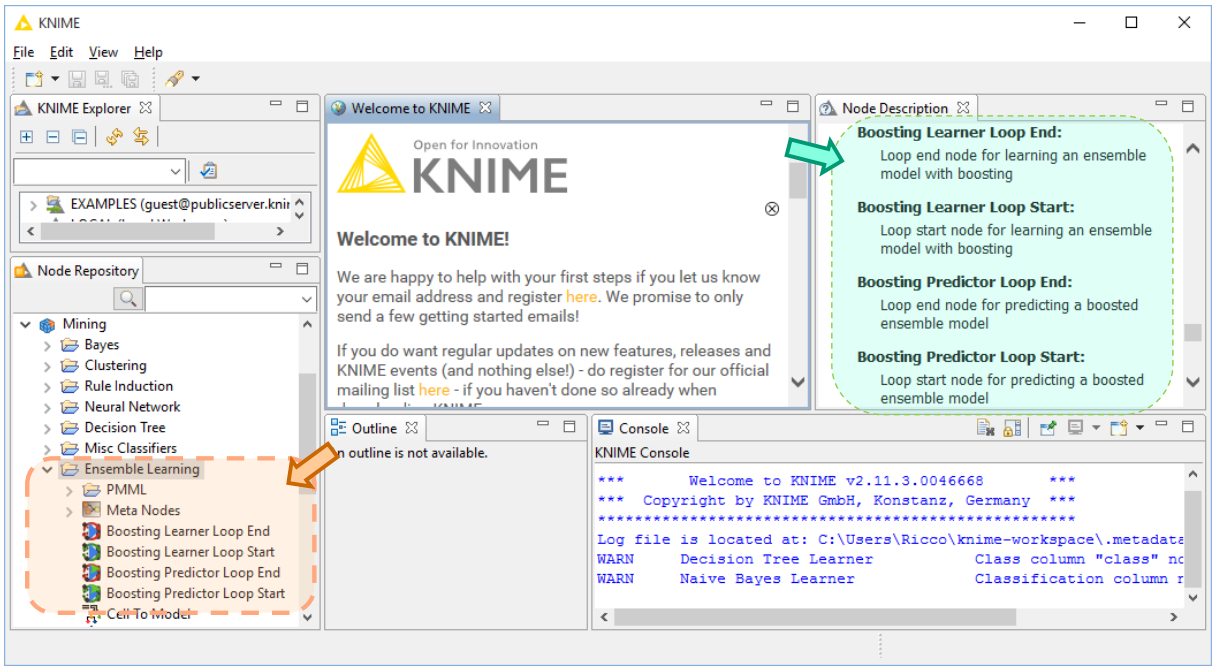



Le taux d'erreur est de **5.48%** en croisant REGION_TYPE et PRED_BAGGING_2.

**Remarque :** A titre de curiosité, j'ai réalisé un apprentissage simple avec RND TREE, le taux d'erreur est de **16.14%**, cela laisse à penser que la proportion de variables pertinentes est assez élevée dans la base. Et un boosting de RND TREE aboutit à une erreur de **5.05%**.

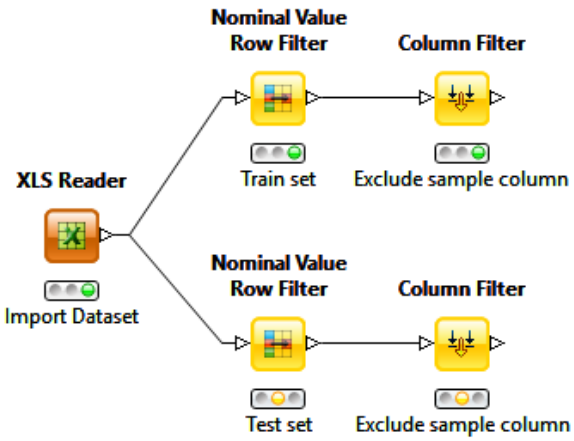
## 5.2 Analyse avec Knime

Knime propose des procédures génériques de Bagging et Boosting via le package ENSEMBLE LEARNING qu'il faut installer au préalable. Elles peuvent prendre n'importe quel algorithme d'apprentissage supervisé sous-jacent.




### 5.2.1 Importation et préparation des données

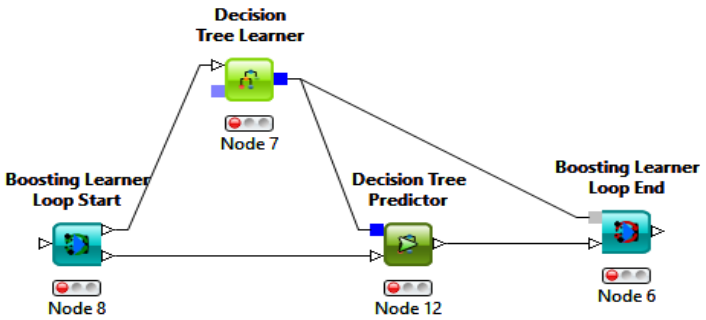
Knime sait lire directement les fichiers XLSX. Nous utilisons le composant **XLS READER**. A l'aide de **NOMINAL VALUE ROW FILTER**, nous filtrons la base selon la colonne ''sample'' pour produire les échantillons d'apprentissage (''sample = train'') et de test (''sample = test''). **COLUMN FILTER** permet d'évacuer la colonne ''sample'' pour le reste de l'étude.



### 5.2.2 Boosting - Construction du classifieur

Le boosting se construit sous forme de boucle dans Knime. Des META-NODES permettent de résumer les opérations, tant en apprentissage qu'en prédiction (Boosting Learner et Boosting Predictor). J'ai préféré définir les étapes manuellement. En effet, la trame peut paraître déroutante au premier abord mais, une fois que l'on a compris la logique, l'enchaînement des outils s'avère pédagogiquement très intéressant.

La séquence suivante défini une analyse Boosting d'arbres de décision. Le composant BOOSTING LEARNER LOOP START initie la boucle boosting à partir des données d'apprentissage. Il est connecté à un outil d'apprentissage DECISION TREE LEARNER, mais aussi à un outil prédictif DECISION TREE PREDICTOR. En effet, les erreurs de prédiction à l'étape $t$ permettent de définir les pondérations des individus à l'étape ($t + 1$). BOOSTING LEARNER LOOP END permet de boucler le tout et passer à l'itération suivante.




En termes de paramétrage :

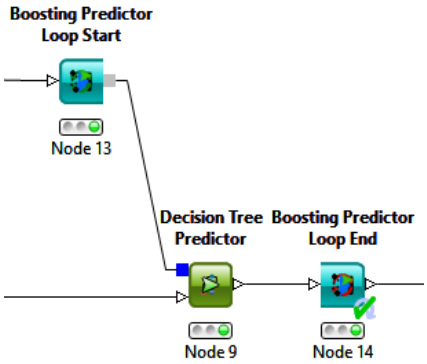
- dans DECISION TREE LEARNER, nous indiquons la variable cible REGION_TYPE ;
- dans BOOSTING LEARNER LOOP END, nous spécifions la cible observée [REGION_TYPE], la variable prédite par le processus [Prediction(REGION_TYPE)], et le nombre d'itérations (25 pour se caler sur Tanagra).

### 5.2.3 Evaluation sur l'échantillon test

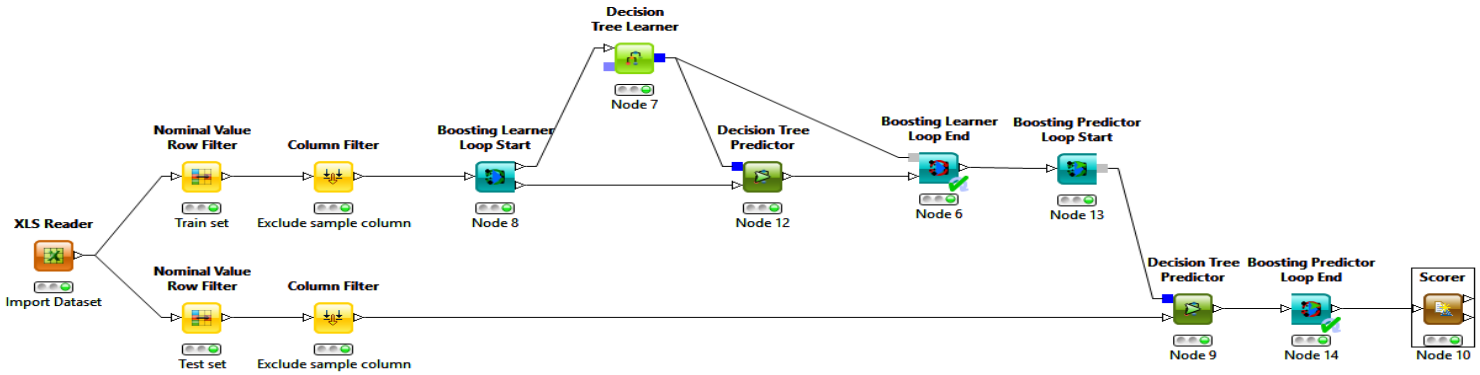
Deux opérations sont nécessaires à ce stade : utiliser le meta-classifieur pour prédire sur l'échantillon test, puis confronter la classe observée avec la classe prédite pour calculer la matrice de confusion et le taux d'erreur.

Une nouvelle boucle définit la prédiction sur l'échantillon test. L'outil BOOSTING PREDICTOR LOOP START prend en entrée le boosting d'apprentissage. Il est connecté à un DECISION TREE PREDICTOR, lequel prend en entrée également les données test. On voit bien l'idée, il

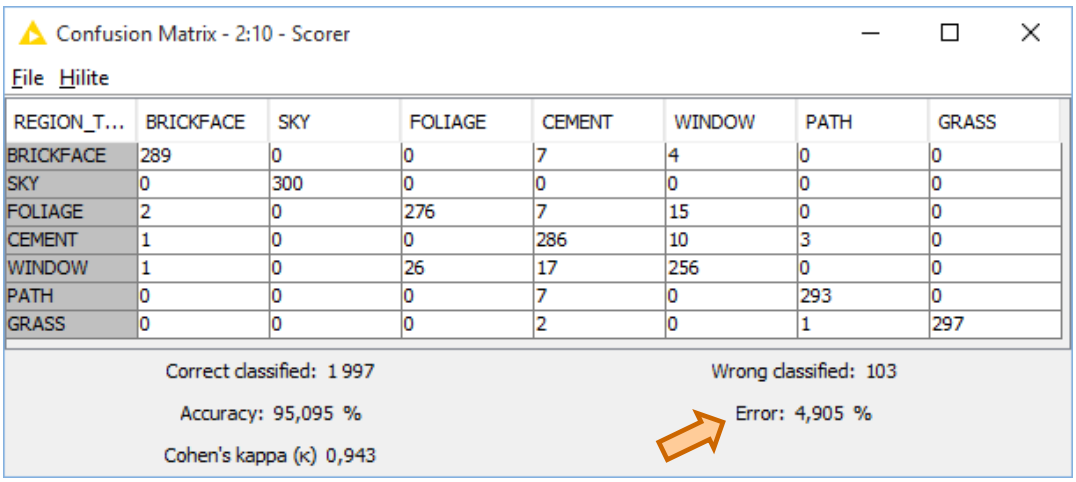
s'agit de passer en revue les arbres composant le modèle et de les faire voter (on a un vote pondéré dans le boosting). BOOSTING PREDICTOR LOOP END clôture la boucle.




Il ne reste plus qu'à placer le composant SCORER qui se charge de croiser REGION_TYPE et Prediction(REGION_TYPE). Voici le diagramme dans sa globalité.




Le taux d'erreur en test issu de SCORER est **4.90%**.



Confusion Matrix - 2:10 - Scorer

File Hilite

| 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 propose toujours des solutions intéressantes. Le plus dur est de comprendre la logique d'utilisation des composants. J'y ai mis le temps en ce qui me concerne. Mais une fois que j'ai compris les idées directrices, le schéma me paraît limpide.

# 6 Conclusion

Le premier objectif de ce tutoriel est d'apporter une touche concrète au support de cours consacré aux techniques ensemblistes de type bagging et boosting que j'ai rédigé dernièrement. J'en ai profité pour comparer les packages proposés par R (adabag, random forest) et Python (scikit-learn). Puis, tant qu'à faire, j'ai décrit également ce qui se faisait sous Tanagra (assez curieusement, je n'ai jamais écrit un tutoriel pour ces outils jusqu'à présent) et Knime. Au final, du moins en ce qui concerne le fichier « image », ces approches s'avèrent particulièrement efficaces. C'est aussi *un peu* vrai en général. Random Forest et Boosting se partagent souvent les meilleures places dans les challenges.

# 7 Références

Package ''adabag'' pour R.

Package ''randomForest'' pour R.

Les méthodes ensemblistes de scikit-learn pour Python.