1 Objectif

Mise en œuvre de la méthode « Gradient Boosting » sous R et Python.

Ce tutoriel fait suite au support de cours consacré au « Gradient Boosting » auquel nous nous référerons constamment [GBM, pour ''Gradient Boosting Machine'' en anglais]. Il vient également en complément des supports et tutoriels pour les techniques de bagging, random forest et boosting mis en ligne précédemment [BRBC et BRBT].

La trame sera très classique : après avoir importé les données préalablement scindées en deux fichiers (apprentissage et test), nous construisons les modèles prédictifs et nous les évaluons. Le critère taux d'erreur en test est privilégié pour comparer les performances.

La question du paramétrage, particulièrement délicate dans le cadre du gradient boosting, est étudiée. En effet, ils sont nombreux, et leur influence est considérable. Malheureusement, si l'on devine à peu près les pistes à explorer pour améliorer la qualité des modèles (plus ou moins « coller » à l'échantillon d'apprentissage c.-à-d. plus ou moins régulariser), identifier avec exactitude les paramètres à manipuler et fixer les bonnes valeurs relèvent un peu de la boule de cristal, surtout lorsqu'ils interagissent entre eux. Ici, plus que pour d'autres méthodes de machine learning, la stratégie essai-erreur prend beaucoup d'importance.

2 Données

2.1 Données utilisées

Nous utilisons les données « Optical Recognition of Handwritten Digits »[1] du serveur UCI. L'objectif est de reconnaître des caractères manuscrits (les chiffres 0 à 9 ; la variable cible CHIFFRE possède K = 10 modalités) à partir d'images bitmap transformées en vecteur de taille 64 (''pix1'' à ''pix64''). Nous disposons de 5620 observations en tout.

2.2 Procédure d'évaluation des approches

Pour corser l'affaire, nous modifions la répartition originelle des données : 200 observations sont utilisées pour l'apprentissage, 5420 pour le test. Les distributions étant à peu près

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

équilibrées, nous disposons donc d'une vingtaine d'observations par classe pour l'apprentissage.

200 individus dans un espace à 64 dimensions, voilà un excellent terreau pour le sur-apprentissage. Nous devrons faire preuve de beaucoup de clairvoyance pour orienter correctement les paramètres des méthodes que nous implémenterons.

3 Gradient boosting avec R

3.1 Importation et préparation des données

Nous importons les fichiers d'apprentissage et de test avec les paramètres adéquats. Nous affichons le type des données. Ces informations sont essentielles pour certains packages que nous utiliserons ci-dessous.

```
#modifier le répertoire par défaut
setwd("... votre répertoire ...")

#importer les échantillons d'apprentissage et de test
#stockés dans deux fichiers différents
dtrain <- read.table("opt_digits_train.txt",header=T,sep="\t")
dtest <- read.table("opt_digits_test.txt",header=T,sep="\t")

#affichage du type des variables
print(sapply(dtrain,class))
```

La variable cible CHIFFRE est reconnue comme « factor » de R. Les autres colonnes, toutes numériques, sont reconnues comme « integer ».

```
  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 Fonction pour le calcul du taux d'erreur

Nous écrivons une fonction pour calculer le taux d'erreur. Elle prend en entrée les données tests et la prédiction du modèle. Elle calcule la matrice de confusion et en déduit la proportion des observations mal classées (en pourcentage).

```
#fonction pour le calcul du taux d'erreur
err_rate <- function(D,prediction){
  #matrice de confusion
  mc <- table(D$chiffre,prediction)
  #taux d'erreur
  #1 - somme(individus classés correctement) / somme totale individus
  err <- 1 - sum(diag(mc))/sum(mc)
  print(paste("Error rate :",round(100*err,2),"%"))
}
```

3.3 Arbre de décision avec le package « rpart »

Nous évaluons le comportement d'un arbre de décision dans un premier temps. La méthode nous sert de référence car les techniques ensemblistes que nous analysons dans ce tutoriel l'utilisent comme classifieur sous-jacent c.-à-d. les classifieurs individuels sont des arbres. A priori, les approches ensemblistes devraient faire mieux.

3.3.1 Arbre de décision avec les paramètres par défaut de rpart

Construction du modèle. Nous chargeons le package « **rpart** » et nous lançons la modélisation avec les paramètres par défaut [ARB, page 9].

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

L'arbre comporte 13 feuilles. L'afficher n'a pas réellement d'intérêt dans notre contexte. On se contentera de remarquer que les feuilles couvrent 200 / 13 ≈ 15 individus en moyenne.

Lorsqu'on affiche l'importance des variables…

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

… nous constatons que pix34 est celle qui pèse le plus dans la prédiction, puis viennent 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
```

Certaines variables n'apparaissent pas dans la liste, notamment parce qu'elles sont constituées d'une unique valeur. Ce sont des constantes. On s'en rend très facilement compte en calculant les écarts-type (ex. pix1, pix9, pix25, etc.). Dans une étude réelle, on aurait dû commencer par là d'ailleurs : calculer des statistiques descriptives pour situer un peu les caractéristiques des variables et les relations qui peuvent exister entre elles.

```
> 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
```

Évaluation des performances. Nous appliquons le modèle sur l'échantillon test. Nous confrontons les prédictions avec les valeurs observées de la variable cible à l'aide de la fonction **err_rate()** définie ci-dessus (section 3.2).

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

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

Le taux d'erreur en test est de **36.59 %**. Ce sera notre valeur de référence. On devra toujours faire mieux dans ce qui suit. Notons que le classifieur par défaut consistant à affecter les individus à la classe majoritaire aurait comme taux d'erreur (1 - 1/K = 1 - 1/10 = 90%) puisque nos classes sont à peu près équilibrées. Même si ses performances ne sont pas mirifiques, 1 erreur pour 3 individus classés, l'arbre amène quand même de l'information pertinente pour la prédiction.

3.3.2 Modification des paramètres de l'arbre

Je m'appuie souvent sur le stratagème suivant pour caractériser les données que je manipule en analyse prédictive : je construis un arbre volontairement surdimensionné, je regarde les performances sur l'échantillon test. L'idée est simple, est-ce que le danger du sur-apprentissage nous guette dans notre contexte ? Si les performances s'effondrent, cela laisse à penser qu'il faut fortement régulariser les techniques d'apprentissage à mettre en œuvre. Dans le cas contraire, si elles se maintiennent ou mieux s'améliorent, on peut penser qu'on a affaire à un espace peu bruité. Il paraît plus rentable dans ce cas de travailler sur la composante « biais » de l'erreur plutôt que sur la « variance ».

Modifions les paramètres de l'arbre en diminuant les effectifs requis sur les nœuds et feuilles, et désactivons le paramètre de complexité 'cp' en le mettant à 0 [ARB, page 11].

```
#nouveaux paramètres pour l'arbre
param.tree.2 <- rpart.control(minsplit=5,minbucket=2,cp=0)

#modélisation avec les nouveaux paramètres
m.tree.2 <- rpart(chiffre ~ ., data = dtrain, control = param.tree.2)
print(m.tree.2)

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

#taux d'erreur
err_rate(dtest,y.tree.2)
```

L'arbre présente 23 feuilles (8.7 individus par feuilles en moyenne). Le taux d'erreur passe à **32.8 %**. Apparemment, le sur-apprentissage n'est pas la première préoccupation ici, il faudra créer des variantes qui « collent » plus aux données d'apprentissage lorsque nous aurons à paramétrer les méthodes que nous examinerons dans les sections suivantes.

3.4 Boosting avec le package « adabag »

La seconde méthode de référence est le boosting [BRBC, page 28 et suivantes]. Le « gradient boosting » est censé en être une généralisation. Je l'avais d'ailleurs identifié lorsque j'avais lu dans la première version du fameux ouvrage de Hastie et al. (« Elements of Statistical Learning », 2001 ; pages 320 à 322). Je voyais surtout là une vraie usine à gaz avec des paramètres additionnels qui rendaient la manipulation de l'ensemble assez ardue. J'avoue avoir été assez étonné lorsque j'ai lu dans des références récentes que le « gradient boosting » trustait les tableaux d'honneur dans les challenges. C'est un peu ce qui m'a poussé à étudier de près la méthode d'ailleurs.

Nous utilisons le package « adabag » décrit dans un précédent tutoriel [BRBT, section 3.6]. Nous réitérons notre séquence apprentissage-test.

```
#librairie « adabag » à installer au préalable
library(adabag)

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

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

#taux d'erreur en test
err_rate(dtest,y.boosting$class)
```

La méthode s'appuie sur ''mfinal = 100'' arbres de décision[2]. Il utilise l'algorithme SAMME ''coeflearn = Zhu'' [BRBC, page 32].

Le taux d'erreur en test est de **10.83 %**. Par rapport aux arbres individuels, l'amélioration des performances est spectaculaire ! Le taux d'erreur a été divisé par trois. Cela montre, si besoin était, que les méthodes ensemblistes sont souvent très performantes dans la grande majorité des situations. Le seul reproche qu'on peut leur adresser est l'absence d'un modèle explicite facile à interpréter et à manipuler en déploiement.

Nous devrions nous intéresser aux paramètres de ce type de méthode. Leurs valeurs semblent convenir puisque nous avons fortement amélioré les performances. Nous avons explicitement demandé 100 arbres, mais nous n'en connaissons pas les caractéristiques. Or,

2 Nous nous en tiendrons systématiquement 100 arbres tout au long de ce tutoriel.

elles pèsent énormément sur le comportement du modèle [BRBC, page 36]. Nous mettrons de côté cette question dans cette section pour mieux y revenir dans les parties dévolues au gradient boosting ci-dessous.

3.5 Gradient boosting avec « gbm »

Nous explorons le package « gbm » pour commencer. La description de la méthode réellement implémentée et de ses paramètres est accessible en ligne (Ridgeway, 2007).

3.5.1 Paramétrage par défaut

Construction du classifieur. Dans un premier temps, nous lançons l'algorithme en exploitant les paramètres par défaut. Nous spécifions quand même la distribution « multinomiale », qui correspond à la fonction de coût « déviance multinomiale » [GBM, page 13], pour que la procédure interprète correctement notre variable cible catégorielle à K = 10 modalités.

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

#modélisation sur les données d'apprentissage
#paramètres par défaut
#fonction de coût : déviance multinomiale
m.gbm.default <- gbm(chiffre ~ ., data = dtrain, distribution="multinomial")

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

R nous affiche les informations suivantes.

```
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 variables sur les 64 disponibles jouent un rôle effectif dans la modélisation. Cela veut dire surtout que 35 variables n'ont aucun impact ! Cela va au-delà des 6 constantes initialement identifiées (page 4), elles (ces 35 variables) n'apparaitraient dans aucun des arbres élaborés.

Avec la fonction summary()…

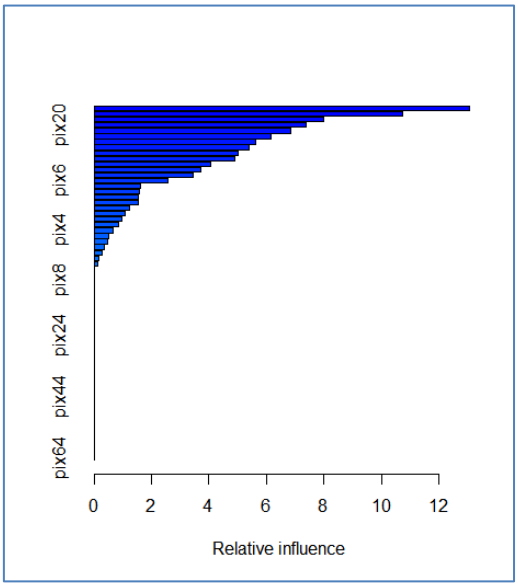
```
#summary -> variable importance
print(head(summary(m.gbm.default),10))
```

… nous disposons de l'importance des variables (les 10 premières, pix37 se démarque cette fois-ci, à la différence des arbres individuels, voir 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
```

L'importance des variables peut être représentée dans un diagramme en bâton. Il est plus folklorique qu'autre chose. Il est quasiment impossible de distinguer le nom des variables quand elles sont nombreuses.



Prédiction. Nous appelons la procédure **predict()** pour effectuer la prédiction sur l'échantillon test.

```
#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))
```

Nous devons spécifier le nombre d'arbres à utiliser. Nous utilisons la valeur renvoyée par la fonction suite à la modélisation ($n.trees). **predict()** renvoie un tableau avec, en ligne, les individus à classer, en colonne, les scores pour chaque classe. Nous disposons des valeurs suivantes pour les 6 premiers individus :

```
             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
```

Pour chaque ligne, la prédiction correspond à la colonne présentant la valeur la plus élevée. Nous utilisons la commande suivante pour identifier la prédiction pour chaque individu :

```
#transformer les scores en classe prédite
y.gbm.default <- factor(levels(dtrain$chiffre)[apply(p.gbm.default[,,1],1,which.max)])
```

Détaillons-la :

- apply() appliqué à chaque ligne de la matrice fournie par predict() recherche le numéro de colonne présentant le score le plus élevé ;
- levels(dtrain$chiffre) renvoie la liste des modalités de la variable cible CHIFFRE c.-à-d. (C0, C1, C2, …, C9).

Grâce au mécanisme de réplication de R, nous disposons donc d'un vecteur de chaîne de caractères contenant les valeurs {'C0'…'C9'}.

- factor() permet de transformer le vecteur de chaînes en un vecteur de type « factor » constituant la prédiction du modèle sur l'échantillon test.

Il ne nous reste plus qu'à appeler la fonction calculant le taux d'erreur :

```
#taux d'erreur en test
err_rate(dtest,y.gbm.default)
```

Patatras ! Nous obtenons un taux d'erreur de **35.9 %**. Tout ça pour ça serait-on tenté de dire. Avant de se gloser inopportunément sur l'inefficacité du package « gbm », il serait temps de s'intéresser aux paramètres utilisés par la procédure.

3.5.2 Modification des paramètres

Nous disposons des propriétés d'un objet R en faisant appel à **attributes()**

```
#afficher les propriétés de l'objet issu de la modélisation
print(attributes(m.gbm.default))
```

Nous obtenons.

```
$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"
```

Certaines nous interpellent au regard de ce que nous connaissons des paramètres des méthodes « gradient boosting » [GBM, page 23]. Nous les affichons.

```
#nombre d'arbres = 100
print(m.gbm.default$n.trees)

#profondeur des arbres = 1, par défaut « decision stump » sont utilisés !!
print(m.gbm.default$interaction.depth)

#paramètre de shrinkage = 0.001, corrections très faibles répercutées
#à chaque arbre construit [GBM, page 16]
print(m.gbm.default$shrinkage)

#proportion des observations choisis aléatoirement = 0.5
#pour la construction de l'arbre suivant à chaque étape [GBM, page 16]
print(m.gbm.default$bag.fraction)
```

La principale information qui saute aux yeux (« bondit aux yeux » même dirais-je) est que la procédure s'appuie sur des « decision stump » c.-à-d. des arbres de profondeur égale 1, avec une seule segmentation. Dans un problème à K = 10 classes, ce n'est manifestement pas approprié. Les arbres individuels sont très mauvais, on est dans une situation de sous-apprentissage (underfitting en anglais) c.-à-d. de sous exploitation des données disponibles.

Nous modifions les paramètres de manière à produire des arbres individuels plus profonds (interaction.depth = 6). Nous améliorons également la vitesse de convergence en donnant une influence plus forte aux corrections apportées par chaque arbre (shrinkage = 0.1) puisqu'on ne craint pas le sur-apprentissage sur nos données a priori (voir section 3.3.2).

```
#modifier les cararctéristiques de l'algorithme
m.gbm.2 <- gbm(chiffre ~ ., data = dtrain, distribution="multinomial",interaction.depth=6,shrinkage=0.1)

#affichage
print(m.gbm.2)

#prédiction
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)])

#taux d'erreur
err_rate(dtest,y.gbm.2)
```

Grand bien nous en a pris puisque le taux d'erreur est passé à **12.47 %**[3]. Ces corrections vont dans le bon sens.

Notons deux éléments importants :

- Si nous avions laissé 'shrinkage' à 0.001, le taux d'erreur aurait été de 24.21 %, il aurait fallu augmenter le nombre d'arbres pour obtenir de nouveau des performances satisfaisantes (Ridgeway, 2007 ; section 3.2).
- Boosting tout seul (section 3.4) faisait 10.83 % (l'écart est significatif sur 5420 observations) sans que l'on ait eu à tripatouiller les paramètres. Cela remet les choses en perspective.

Bien sûr, en ajustant finement les paramètres du gradient boosting, nous devrions atteindre le même niveau de performance que le boosting « classique ». Mais encore faut-il pouvoir identifier les bonnes valeurs des bons paramètres sans que cela ne s'apparente à un chemin de croix.

3.6 Gradient boosting avec « xgboost »

Nous examinons le package « xgboost » dans un second temps. Des tutoriels (ex. 1, 2) permettent de s'initier à son utilisation. Une documentation technique est également disponible en ligne.

3 Comme il y a une part aléatoire dans l'algorithme à cause du paramètre ''bag.fraction < 1'', nous n'avons pas exactement les mêmes résultats à chaque lancement de la procédure. Avec l'option ''bag.fraction = 1'', l'algorithme devient déterministe et le taux d'erreur est (donc systématiquement) de 14.26 %

3.6.1 Apprentissage avec les paramètres par défaut

Une préparation des données est nécessaire avant de pouvoir utiliser la fonction d'apprentissage du package. Nous devons traduire les data frame R en type « matrix », et tous les vecteurs doivent être de type numérique (xgboost n'accepte pas les types *integer*).

```
#convertir le type des descripteurs d'integer (page 2) en numeric
#'-1' parce que la variable n°1 (chiffre) ne doit pas être traité ici
#la liste de vecteurs est transformée en type data.frame
XTrain <- data.frame(lapply(dtrain[,-1],as.numeric))
#le data frame est transformé en type matrice
XTrain <- as.matrix(XTrain)

#recoder la variable cible (de type factor) en un vecteur numérique
#et les codes (1,2,…,10) sont transformés en (0,1,2,…,9)
yTrain <- unclass(dtrain$chiffre)-1
```

Nous pouvons lancer l'apprentissage en demandant 100 arbres. L'indication du nombre de classes est obligatoire (num_class = 10). La fonction softmax sert de fonction de coût [GBM, page 13]. Les données tests doivent être également transformées pour que la fonction predict() soit opérante. Les valeurs prédites, définies sur la plage (0, 1, …, 9), sont additionnées de '1' pour que la confrontation avec les valeurs observées de la cible sur l'échantillon test soit licite.

```
#package xgboost
library(xgboost)

#apprentissage avec les paramètres par défaut (eta=0.3, max.depth=6)
m.xg.def <- xgboost(data=XTrain,label=yTrain,objective="multi:softmax",num_class=10,nrounds=100)

#conversion des descripteurs de l'échantillon test
XTest <- data.frame(lapply(dtest[,-1],as.numeric))
XTest <- as.matrix(XTest)

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

#taux d'erreur en test
err_rate(dtest,y.xg.def)
```

Nous obtenons un taux d'erreur de **15.5 %**. Là aussi, les résultats sont décevants comparés à ceux du boosting (10.83 %, section 3.4). Ils le sont d'autant plus qu'en scrutant la documentation, nous constatons que le paramètres par défaut sont loin d'être aberrants (shrinkage, eta = 0.3 ; profondeur des arbres, max_depth = 6).

3.6.2 Modification des paramètres

On se perd un peu en conjectures à vrai dire. Une première piste serait de réduire le paramètre de shrinkage pour éviter les corrections excessives (eta = 0.1). Mais cette modification n'est pas suffisante. La seconde piste explore une fonctionnalité originale qui rapproche l'implémentation « xgboost » de la méthode des forêts aléatoires [BRB, page 22]. L'idée consiste à effectuer un échantillonnage aléatoire des variables utilisées pour la construction des arbres individuels. On accentue ainsi la diversité des arbres (elles sont décorrélées selon la terminologie des Random Forest). Leur combinaison n'en sera que plus efficace[4].

Nous avons relancé la méthode avec les nouveaux paramètres

```
# apprentissage avec les nouveaux paramètres
# 0.125 = SQRT(64)/64, inspiré du paramètrage des ''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)

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

#taux d'erreur en test
err_rate(dtest,y.xg.2)
```

Le taux d'erreur est de **10.44 %**. L'amélioration est substantielle. Mais j'avoue être assez partagé. Tripatouiller les paramètres - un peu - au petit bonheur la chance n'est pas vraiment une approche satisfaisante.

3.6.3 Importance des variables

« xgboost » propose également un outil permettant d'obtenir l'importance des variables. Pour nous, il nous servira surtout à identifier la « diversification » des arbres suite à l'introduction de l'option ''colsample_bytree = 0.125'' lors de l'apprentissage.

---

4 Pour être tout à fait honnête, j'avais testé différentes méthodes lors de la préparation de ce tutoriel. L'approche Random Forest s'est révélée la plus performante (voir section 4.5). Il a été facile d'identifier sa particularité - mécanisme de tirage aléatoire des variables durant la segmentation - qui lui permettait de se démarquer. Il ne restait plus qu'à trouver dans ''xgboost'' l'option qui permettait de s'en rapprocher. Le procédé n'est pas exactement le même ceci étant dit : pour Random Forest, l'échantillonnage intervient au niveau de la sélection de variables pour la segmentation ; pour xgboost, dixit la documentation, il intervient en amont de la construction de chaque arbre.

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

```
   Feature       Gain
1    pix37 0.08741986
2    pix44 0.06783863
3    pix20 0.05909602
4    pix22 0.05765523
5    pix30 0.05586736
6    pix61 0.04987202
7    pix29 0.04899898
8    pix52 0.04585716
9    pix34 0.04168518
10   pix47 0.04167961
11   pix27 0.03714287
12   pix38 0.03642339
13    pix3 0.03033923
14    pix6 0.03012236
15   pix11 0.02646798
```

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

```
   Feature       Gain
1    pix28 0.05056194
2    pix31 0.03659583
3    pix21 0.03599328
4    pix22 0.03565025
5    pix20 0.03510729
6    pix44 0.03322517
7    pix37 0.03265284
8    pix59 0.03157483
9    pix45 0.03011384
10    pix3 0.03010671
11   pix47 0.02917668
12   pix36 0.02911094
13   pix29 0.02891033
14   pix38 0.02856075
15   pix19 0.02647185
```

Leur rôle est mieux réparti (moins de concentration du « gain » sur les premières variables) avec l'introduction de l'échantillonnage des variables. Il pouvait difficilement en être autrement de toute manière. Dans le cas présent, cela induit une amélioration des performances.

3.7 Gradient boosting avec « mboost »

« mboost » est le troisième package pour R présenté dans le support de cours [GBM, page 21]. Il a parfaitement fonctionné pour l'exemple illustratif, qui est binaire précisons-le c.-à-d. la variable cible présente 2 modalités. Je l'ai lancé sur nos données. Malheureusement la session R a échoué. Voici le code utilisé.

```
#package mboost
library(mboost)

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

J'ai réitéré l'expérimentation sur des données IRIS [data(iris) dans R, package ''dataset''']. La même erreur est survenue. Pourtant, à la lecture de la documentation, il semble que le traitement des problèmes multi-classes [option family = Multinomial()] soit possible. J'ai stoppé là mes investigations.

4 Gradient boosting avec Python

Dans cette section, nous traitons les mêmes données sous Python à l'aide du package « scikit-learn » (**version 0.17.1**). La trame est globalement la même. En sus, nous mettrons à contribution l'outil « grille de recherche » de scikit-learn qui permet d'identifier « automatiquement » les meilleurs valeurs des paramètres des techniques de machine learning. Nous verrons s'il est vraiment efficace dans notre contexte.

La documentation de la procédure « gradient boosting » programmée dans le package scikit-learn est accessible en ligne. L'étudie attentive des paramètres est salutaire. Pour aller plus loin, je conseillerais la lecture de l'ouvrage de Biernat et Lutz (2015 ; chapitre 12, plus particulièrement à partir de la page 125).

4.1 Importation et préparation des données

Comme sous R, il nous faut tout d'abord importer les données d'apprentissage et de test, puis procéder aux transformations nécessaires à l'utilisation des procédures de scikit-learn.

```
#changement du répertoire par défaut
import os
os.chdir("... le dossier de vos données ...")

#importation avec le package ''pandas''
import pandas
dtrain = pandas.read_table("opt_digits_train.txt",sep="\t",header=0,decimal=".")

#vérification des dimensions : 200 obs., 65 variables
print(dtrain.shape)

#variable cible - échantillon d'apprentissage
y_app = dtrain.as_matrix()[:,0]
```

```
#descripteurs - échantillon d'apprentissage
X_app = dtrain.as_matrix()[:,1:64]

#importation de l'échantillon test
dtest = pandas.read_table("opt_digits_test.txt",sep="\t",header=0,decimal=".")

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

#conversion également en type 'matrix'
y_test = dtest.as_matrix()[:,0]
X_test = dtest.as_matrix()[:,1:64]
```

4.2 Fonction pour le calcul du taux d'erreur

Scikit-learn propose des outils performants avec la classe « metrics ». Nous nous contenterons de l'utiliser en convertissant le taux de succès (accuracy) en taux d'erreur.

```
#calcul du taux d'erreur avec la classe ''metrics''
#confrontation y observé vs. y prédit sur l'échantillon test
from sklearn import metrics
err = 1.0 - metrics.accuracy_score(y.obs.test,y.pred.test)
print(err)
```

Remarque : Dans un précédent tutoriel [BRBT, section 4.2], profitant du fait que les signatures des fonctions de scikit-learn sont homogènes, nous avions défini une fonction qui prend en entrée les données de test et le modèle : elle intègre alors la prédiction et le calcul du taux d'erreur. Cette solution est assez élégante. J'ai pris le parti de faire autrement ici pour être raccord avec R où nous enchaînons explicitement prédiction et calcul du taux dans le programme principal.

4.3 Gradient boosting avec scikit-learn

Nous sommes parés pour lancer le processus de modélisation. Plusieurs étapes sont nécessaires.

Instanciation de la classe. Il nous faut créer une instance de la classe « Gradient boosting », nous affichons les paramètres par défaut de l'algorithme.

```
#gb est un objet de type gradient boosting
#aucun paramètre n'est spécifié ➔ l'objet utilise les paramètres par défaut
from sklearn.ensemble import GradientBoostingClassifier
gb = GradientBoostingClassifier()

#afficher l'objet revient à afficher ses paramètres
```

```
print(gb)
```

Nous obtenons.

```
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)
```

Nous remarquons entres autres, en nous référant à la documentation :

- la fonction de coût déviance est utilisée (loss = 'deviance') [GBM, page 13] ;
- 100 arbres sont élaborés (n_estimators = 100) ;
- la constante d'apprentissage (learning rate) $\nu = 0.1$ [GBM, page 16] ;
- la profondeur maximal des arbres individuels est fixé à max_depth = 3, la valeur est assez faible et nous prémunit du sur-apprentissage [GBM, page 23] ;
- les effectifs sur les nœuds, avant et après segmentation, ne sont pas opérantes avec des valeurs aussi faibles (min_samples_split = 2, min_samples_leaf = 1) [GBM, page 23] ;
- il n'y a pas de tirage aléatoire des individus (pas de ''stochastic gradient boosting'', [GBM, page 16]) pour l'élaboration des arbres intermédiaires (subsample = 1.0) ;
- il n'y a pas de tirage aléatoire de variables *à la Random Forest* lors des segmentations dans les arbres individuels (max_features = None).

Apprentissage. Nous lançons la modélisation à l'aide de la méthode **fit()**

```
#modélisation à partir des données d'apprentissage
gb.fit(X_app,y_app)
```

Prédiction et évaluation. Et il ne nous reste plus qu'à évaluer sur l'échantillon test.

```
#prédiction sur l'échantillon test
y_pred = gb.predict(X_test)

#évaluation : calcul du taux d'erreur
#taux d'erreur = 1 - taux de succès
from sklearn import metrics
err = 1.0 - metrics.accuracy_score(y_test,y_pred)
print(err)
```

Le taux d'erreur est de **19.85 %**. Pour le coup, c'est aussi décevant. Encore une fois, il faudra affiner la modélisation en jouant sur les paramètres.

4.4 Grille de recherche des paramètres optimaux

Scikit-learn propose un outil intéressant pour détecter les paramètres « optimaux »[5]. On lui passe une liste de paramètres avec les valeurs à tester, il se charge de trouver la meilleure combinaison en validation croisée. Le point positif est que l'échantillon test n'est jamais mis à contribution dans ce processus, il garde son statut d'arbitre impartial. Le point négatif est qu'on augmente le risque de sur-dépendance à l'échantillon d'apprentissage (200 observations dans notre cas, rappelons-le). Il y a un risque que le meilleur modèle issu de ce processus généralise mal sur l'échantillon test, et donc dans la population.

Voyons ce qu'il en est :

```
#classe pour la grille de recherche des paramètres en validation croisée
#par défaut 3-fold validation croisée
from sklearn.grid_search import GridSearchCV

#combinaisons de paramètres à tester
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']}

#méthode d'apprentissage à tester
gbc = GradientBoostingClassifier()

#object (instance de classe) pour la grille de recherche
grille = GridSearchCV(estimator=gbc,param_grid=parametres,scoring="accuracy")

#lancer la recherche sur les données d'apprentissage
resultats = grille.fit(X_app,y_app)

#affichage des performances (scores)
print(resultats.grid_scores_)

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

#meilleurs paramètres au sens de la validation croisée
```

5 J'oscille entre « épatant » et « inquiétant ». « Epatant » parce que l'outil est quand même très pratique. La vitesse d'exécution est bluffante si l'on considère qu'il essaie toutes les combinaisons possibles. « Inquiétant » parce qu'il nous pousse à tenter tout et n'importe quoi sans vraiment réfléchir. Exactement le contraire de ce que j'essaie de transmettre à mes étudiants.

```
print(resultats.best_params_)
```

Pour max_features, 'sqrt' et 'log2' correspondent respectivement à la racine carrée et logarithme du nombre de prédicteurs candidats. None correspond à « pas de sélection ».

L'outil évalue toutes les combinaisons c.-à-d. 5 x 5 x 3 x 3 = 225 configurations. Par défaut, il y a 3 portions (folds) dans la validation croisée. Il y a donc 675 processus apprentissage-test. Mieux vaut avoir une bonne machine !

La propriété 'grid_scores_' renvoie le tableau de tous les résultats. Pour chaque composition, nous disposons des paramètres évalués, des moyennes et écarts-type du taux de succès en validation croisée sur les 3 folds. Voici par exemple les 3 premiers éléments du tableau :

```
[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},
```

Pour *{'max_features': None, 'max_depth': 2, 'subsample': 1.0, 'learning_rate': 0.3}*, le taux de succès moyen est de *78.5 %* ; etc.

Discerner dans ces sorties le meilleur résultat n'est pas aisé. Heureusement, scikit-learn fournit automatiquement le score optimal avec 'best_score_', nous avons 89,5 %, et la configuration associée, avec 'best_params_' :

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

Ce taux de succès de 89,5 %, soit un taux d'erreur de 10.5 %, est-il réellement digne de foi ? Si nous appliquons notre modèle sur la population, on doit s'attendre à ces performances ?

La seule manière de le savoir est d'appliquer le modèle sur l'échantillon test.

```
#prédiction sur l'échantillon test avec le modèle
#correspondant aux meilleurs paramètres
ypredc = resultats.predict(X_test)

#taux d'erreur en test
err_best = 1.0 - metrics.accuracy_score(y_test,ypredc)
print(err_best)
```

Le taux d'erreur en test est de **9.0 %**. La solution tient manifestement la route puisque c'est le meilleur résultat que nous ayons obtenu jusqu'à présent. Malgré les réserves que l'on

peut toujours émettre par rapport à ces stratégies « brutes » où la performance mesurée sert d'unique critère, force est de constater que le modèle obtenu est performant[6].

4.5 Random Forest

En préparant ce tutoriel, j'avais testé différentes approches pour identifier les meilleures solutions pour le gradient boosting. Je me suis rendu compte que la méthode Random Forest était la plus performante. Ce résultat m'a aiguillé sur l'introduction de l'échantillonnage des variables dans la construction des arbres intermédiaires avec xgboost sous R (section 3.6.2) et scikit-learn sous Python (section 4.4). Ce procédé joue manifestement un rôle important sur nos données.

En effet, en lançant la démarche apprentissage-test avec la méthode Random Forest…

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

#objet RandomForest
rf = RandomForestClassifier(n_estimators = 100)

#modélisation sur les données d'apprentissage
rf.fit(X_app,y_app)

#prédiction sur les données tests
y_pred_rf = rf.predict(X_test)

#taux d'erreur en test
err = 1.0 - metrics.accuracy_score(y_test,y_pred_rf)
print(err)
```

… nous obtenons un taux d'erreur de **8.24 %**, le meilleur résultat de tout notre tutoriel ! On se demande après coup pourquoi on s'est tant embêté finalement. Mais ça, c'est le lot de tout data scientist : « la pépite, elle se mérite » (ça rime en plus !).

5 Conclusion

L'objectif initial de ce tutoriel est de montrer la mise en œuvre du gradient boosting à l'aide des logiciels R et Python. L'opération est relativement aisée en définitive si l'on s'en tient à

6 Attention, puisqu'il y a une part stochastique dans la démarche, vous obtiendrez des résultats légèrement différents en relançant plusieurs fois le processus.

une analyse simple basée sur les paramètres par défaut. Il s'agit d'une méthode prédictive au même titre que les autres.

Les problèmes commencent lorsqu'on se met en tête de modifier les paramètres pour les mettre en adéquation avec les données traitées. On se rend compte qu'ils sont très nombreux, qu'ils ne correspondent pas toujours aux paramètres décrits dans les ouvrages qui font référence, celui de Hastie et al. (2009) en l'occurrence. Pour identifier clairement ce qui est réellement implémenté dans les packages et les potentialités en matière de paramétrage, la seule solution est de lire la documentation en détail et de réaliser des expérimentations. Dans le cas du gradient boosting, le travail sur les paramètres est indispensable. Nous avons constaté qu'ils pèsent énormément sur les performances des modèles subséquents.

Les outils de recherche automatique des meilleures configurations tels que la grille de recherche en validation croisée de scikit learn peuvent être d'une grande aide. Le tout est de ne pas les utiliser à tort et à travers en se fiant aveuglément aux indications qu'ils dispensent. Identifier explicitement les mécanismes du succès d'une méthode de data mining (avec son paramétrage) sur nos données reste la meilleure manière d'assurer la reproductibilité de nos résultats. Mais ce n'est pas toujours évident, je le concède.

Enfin, nous remarquerons que toutes les méthodes ensemblistes font mieux que les arbres individuels qui nous ont servi de référence (section 3.3).

6 Références

[ARB] « Introduction à R - Arbre de décision », mars 2012.

[BRBC] « Bagging, Random Forest, Boosting - Diapos », novembre 2015.

[BRBT] « Random Forest et Boosting avec R et Python », novembre 2015.

[GBM] « Gradient boosting - Diapos », avril 2016.

Biernat E, Lutz M., « Data Science : fondamentaux et études de cas - Machine learning avec Python et R », 2015 ; chapitre 12.

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

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

Wikipédia (EN), « Gradient boosting », consulté en mai 2016.