Stacking avec R

Tutoriel Tanagra

11 janvier 2019

1. Introduction

Ce tutoriel fait suite au support de cours consacré au stacking. L'idée, rappelons-le, est de faire coopérer des modèles en prédiction en espérant que les erreurs se compenseront. L'ensemble serait alors plus performant que les modèles sous-jacents qui le composent, pris individuellement. A la différence du boosting ou du bagging, nous créons les classifieurs à partir de la même version des données d'entraînement (sans pondération ou autres modifications), la diversité nécessaire à la complémentarité provient de l'utilisation d'algorithmes de familles différentes. Dans les exemples que nous explorerons, le pool comprendra un arbre de décision, une analyse discriminante linéaire, et un SVM avec un noyau RBF. On peut penser que ces approches possèdent des caractéristiques suffisamment dissemblables pour qu'en classement, elles ne soient pas constamment unanimes (si les modèles sont unanimes, les faire coopérer ne sert à rien).

Nous étudierons le stacking de deux manières dans ce document. Dans un premier temps, nous programmerons à partir de zéro les différentes manières de combiner les modèles (vote ''hard'', vote ''soft'', métamodèle). Au-delà du plaisir à le faire, l'objectif est de décortiquer les étapes pour s'assurer la bonne compréhension des approches. Ensuite seulement, dans un deuxième temps, nous utiliserons les outils spécialisés (caretEnsemble, H2O). Le fait de détailler les opérations précédemment permettra de mieux appréhender les paramètres et les opérations préalables que nécessitent l'appel des fonctions dédiées au stacking dans ces packages.

2. Importation et préparation des données

Nous utilisons la base de données Spambase bien connue des data scientists. L'objectif est d'identifier les courriels licencieux (spam = yes) à partir de leurs caractéristiques (fréquence relative des mots, des caractères, présence de lettres en majuscules, etc.).

2.1. Chargement des données

Nous chargeons le fichier de données "**spam_stacking.txt**". Nous en affichons les caractéristiques. Nous disposons 4601 observations et 56 variables.

```
#changement de répertoire
setwd("... votre dossier de travail ...")

#charger le fichier
spam.all <- read.table("spam_stacking.txt", header = T, sep="\t", dec=".")

#vérification
print(str(spam.all))

## 'data.frame':    4601 obs. of  56 variables:
##  $ wf_make                  : num  0 0 0 0.08 0 0 0 0 0 0.4 ...
##  $ wf_address               : num  0.52 0 0 0 0 0.36 0 0 0 0.4 ...
##  $ wf_all                   : num  0.52 0 0.66 0.16 0 0 0 1.85 0.53 0.26 ...
##  $ wf_3d                    : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_our                   : num  0.52 0 0 0 0 0 0 0 0 0.13 ...
##  $ wf_over                  : num  0 0 0.66 0.08 0 0.36 0 0 0.53 0.2 ...
##  $ wf_remove                : num  0 0 0 0 0 1.47 0 0 0 0.06 ...
##  $ wf_internet              : num  0 0 0 0.08 0 0 0 0 0 0.33 ...
##  $ wf_order                 : num  0 0 0 0.73 0 0 0 0 0 0 ...
##  $ wf_mail                  : num  0 0 0 0 0 0.36 0 1.85 0.53 1.14 ...
##  $ wf_receive               : num  0 0 0 0 0 0.36 0 0 0 0.33 ...
##  $ wf_will                  : num  0 0 0.66 0.24 0 0.73 0 1.85 0 1.07 ...
##  $ wf_people                : num  0 0 0 0 0 0 0 1.85 0.53 1 ...
##  $ wf_report                : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_addresses             : num  0 0 0 0 0 0 0 0 0 0.26 ...
##  $ wf_free                  : num  0.52 0 0 0 0 0.36 0 0 0 0.4 ...
##  $ wf_business              : num  0 0 0 0 0 0 0 0 0 0.06 ...
##  $ wf_email                 : num  0 0 0 0 0 1.1 0 0 0 0 ...
##  $ wf_you                   : num  1.56 3.19 1.66 0.32 0 2.2 3.33 5.55 2.15 4.1
...
##  $ wf_credit                : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_your                  : num  0 1.06 0.66 0.16 0 0.73 6.66 0 0.53 0.94 ...
##  $ wf_font                  : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_000                   : num  0 0 0 0 0 0.36 0 0 0 0.53 ...
##  $ wf_money                 : num  0 0 0 0 0 0 0 0 0 0.26 ...
##  $ wf_hp                    : num  0 0 0 0.49 0 0 0 0 0 0 ...
##  $ wf_hpl                   : num  0 0 0 0.57 0 0 0 0 0 0 ...
##  $ wf_lab                   : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_labs                  : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_telnet                : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_857                   : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_data                  : num  0 0 0 0.57 0 0 0 0 0 0 ...
##  $ wf_415                   : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_85                    : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_technology            : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_1999                  : num  0 0 0 0.16 0 0 0 0 0 0 ...
##  $ wf_parts                 : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_pm                    : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_direct                : num  0 0 0 0 0 0 0 0 0 0 ...
```

```
##  $ wf_cs                     : num  0 1.06 0 0 0 0 0 0 0 0 ...
##  $ wf_meeting                : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_original               : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_project                : num  0 0 0.33 0 0 0 0 0 0 0 ...
##  $ wf_re                     : num  0 1.06 0.33 0 0 0 0 0 1.07 0 ...
##  $ wf_edu                    : num  0 1.06 0 0 0 0 0 0 0 0 ...
##  $ wf_table                  : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ wf_conference             : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ cf_comma                  : num  0 0 0 0.126 0 0 0 0 0 0 ...
##  $ cf_bracket                : num  0.192 0 0 0.172 0 0.183 0 0 0.101 0.088 ...
##  $ cf_sqbracket              : num  0 0 0 0.057 0 0 0 0 0 0 ...
##  $ cf_exclam                 : num  0.867 0.398 0 0 0 0.367 0 0.692 0 1.06 ...
##  $ cf_dollar                 : num  0 0 0 0.022 0 0.061 0 0 0 0.151 ...
##  $ cf_hash                   : num  0 0 0 0 0 0.122 0 0 0 0.05 ...
##  $ capital_run_length_average: num  2.22 1.18 1.14 3.21 1 ...
##  $ capital_run_length_longest: int  20 5 4 44 1 36 3 5 16 123 ...
##  $ capital_run_length_total  : int  131 26 56 665 3 264 13 19 52 1045 ...
##  $ spam                      : Factor w/ 2 levels "no","yes": 2 1 1 1 1 2 1 1 1 2
```

2.2. Partition en échantillons d'apprentissage et de test

Nous choisissons de réserver 1601 observations pour l'élaboration des modèles, et 3000 pour leur évaluation. Cette répartition n'est pas très usuelle (on conseille généralement 2/3 vs. 1/3). Dans notre cas, notre étude vise avant tout à montrer l'intérêt du stacking. Nous plaçons les méthodes dans une position défavorable, avec peu (relativement) d'observations en apprentissage.

Pour partitionner les données, nous créons un index de numéros d'observations tirés au hasard dans le vecteur (1, 2, …, 4601). Ceux qui sont intégrés dans l'index feront partie de l'échantillon d'apprentissage, les autres seront assignés à la base de test.

```
#index pour partition
set.seed(1)
idx <- sample(1:nrow(spam.all),1601,replace=FALSE)

#apprentissage
spam.train <- spam.all[idx,]
print(dim(spam.train))

## [1] 1601   56

#test
spam.test <- spam.all[-idx,]
print(dim(spam.test))

## [1] 3000   56
```

set.seed() initialise le générateur de nombre aléatoires de R et permet de reproduire à l'identique l'expérimentation.

2.3. Standardisation des variables

Les descripteurs étant exprimés dans des unités différentes, nous devons les standardiser dans les deux échantillons, mais en utilisant les paramètres (moyennes et écarts-type) exclusivement calculés sur l'ensemble d'apprentissage.

Nous les calculons à partir de "spam.train" donc.

```
#calcul des moyennes sur l'échantillon d'apprentissage
mean.train <- sapply(spam.train[-ncol(spam.train)],mean)

#idem pour l'écart-type
sd.train <- sapply(spam.train[-ncol(spam.train)],sd)
```

Puis nous les appliquons sur "spam.train" :

```
#centrage-réduction de l'échantillon d'apprentissage
spam.train.cr <- data.frame(scale(spam.train[-
ncol(spam.train)],center=mean.train,scale=sd.train))

#adjonction de la variable cible
spam.train.cr$spam <- spam.train$spam
```

Et sur "spam.test" :

```
#centrage et réduction de l'échantillon test
#avec les paramètres (moyenne, écart-type) calculés sur l'échantillon d'apprentissage
spam.test.cr <- data.frame(scale(spam.test[-ncol(spam.test)], center = mean.train,
scale = sd.train))

#adjonction de la cible
spam.test.cr$spam <- spam.test$spam
```

3. Programmation "from scratch"

Dans cette section, nous programmons explicitement les différentes procédures d'assemblage des modèles. Les classifieurs utilisés sont identiques à ceux que nous mettrons en œuvre par la suite avec le package "caretEnsemble" ci-dessous. Reproduire ainsi les calculs permet de comprendre les étapes préalables que demandent "caretEnsemble" (section 4) et "H2O" (section 5) avant de pouvoir faire appel à la fonction dédiée au "stacking". La nécessité d'entraîner les modèles en validation croisée notamment peut paraître inutilement contraignante si l'on n'a pas compris la nature de l'approche et des opérations qui la compose.

3.1. Modèles individuels

Nous travaillons à partir de 3 méthodes supervisées.

Arbre de décision

Nous utilisons la méthode **rpart()** du package éponyme. L'arbre est entraîné à partir de "spam.train.cr". Mettre le paramètre cp (complexity parameter) à zéro permet de désactiver le pré-élagage basé sur le gain relatif du critère de pureté de l'arbre à chaque segmentation. Concrètement, nous aurons un arbre assez grand qui collera fortement aux données d'apprentissage.

```
#RPART pour rpart
library(rpart)
arbre.seul <- rpart(spam ~ ., data = spam.train.cr, control = rpart.control(cp = 0))
print(arbre.seul)

## n= 1601
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##    1) root 1601 629 no (0.60712055 0.39287945)
##      2) cf_dollar< -0.1417039 1179 259 no (0.78032231 0.21967769)
##        4) wf_remove< -0.1332013 1087 173 no (0.84084637 0.15915363)
##          8) cf_exclam< 0.3814195 964  96 no (0.90041494 0.09958506)
##           16) wf_money< -0.230327 937  77 no (0.91782284 0.08217716)
##             32) wf_free< -0.2782042 839  50 no (0.94040524 0.05959476)
##               64) capital_run_length_average< 0.110954 827  44 no (0.94679565 0.05320435)
##                 128) wf_your< 0.4280881 737  28 no (0.96200814 0.03799186) *
##                 129) wf_your>=0.4280881 90  16 no (0.82222222 0.17777778)
##                   258) wf_business< -0.1611129 82  11 no (0.86585366 0.13414634)
##                     516) wf_hp>=-0.2522626 26   0 no (1.00000000 0.00000000) *
##                     517) wf_hp< -0.2522626 56  11 no (0.80357143 0.19642857)
##                       1034) capital_run_length_total< -0.3320695 49   7 no (0.85714286 0.14285714) *
##                       1035) capital_run_length_total>=-0.3320695 7   3 yes (0.42857143 0.57142857) *
##                   259) wf_business>=-0.1611129 8   3 yes (0.37500000 0.62500000) *
##               65) capital_run_length_average>=0.110954 12   6 no (0.50000000 0.50000000) *
##             33) wf_free>=-0.2782042 98  27 no (0.72448980 0.27551020)
##               66) wf_people>=-0.2141915 20   0 no (1.00000000 0.00000000) *
##               67) wf_people< -0.2141915 78  27 no (0.65384615 0.34615385)
##                 134) cf_bracket>=-0.1533206 33   6 no (0.81818182 0.18181818) *
##                 135) cf_bracket< -0.1533206 45  21 no (0.53333333 0.46666667)
##                   270) wf_will>=-0.06246821 9   1 no (0.88888889 0.11111111) *
##                   271) wf_will< -0.06246821 36  16 yes (0.44444444 0.55555556)
##                     542) wf_email>=1.497757 7   1 no (0.85714286 0.14285714) *
##                     543) wf_email< 1.497757 29  10 yes (0.34482759 0.65517241)
##                       1086) capital_run_length_average< -0.1008844 11   4 no (0.63636364 0.36363636) *
##                       1087) capital_run_length_average>=-0.1008844 18   3 yes (0.16666667 0.83333333) *
##           17) wf_money>=-0.230327 27   8 yes (0.29629630 0.70370370)
##             34) wf_our< -0.4059387 15   7 no (0.53333333 0.46666667) *
##             35) wf_our>=-0.4059387 12   0 yes (0.00000000 1.00000000) *
##         9) cf_exclam>=0.3814195 123  46 yes (0.37398374 0.62601626)
##           18) capital_run_length_total< -0.3812102 44  11 no (0.75000000 0.25000000)
##             36) capital_run_length_average< -0.08229827 32   4 no (0.87500000 0.12500000) *
##             37) capital_run_length_average>=-0.08229827 12   5 yes (0.41666667 0.58333333) *
```

```
##         19) capital_run_length_total>=-0.3812102 79  13 yes (0.16455696 0.83544304)
##           38) cf_bracket>=0.6612154 8   3 no (0.62500000 0.37500000) *
##           39) cf_bracket< 0.6612154 71   8 yes (0.11267606 0.88732394) *
##       5) wf_remove>=-0.1332013 92   6 yes (0.06521739 0.93478261)
##        10) cf_exclam< -0.4404847 20   6 yes (0.30000000 0.70000000)
##          20) wf_remove< 2.786974 7   2 no (0.71428571 0.28571429) *
##          21) wf_remove>=2.786974 13   1 yes (0.07692308 0.92307692) *
##        11) cf_exclam>=-0.4404847 72   0 yes (0.00000000 1.00000000) *
##     3) cf_dollar>=-0.1417039 422  52 yes (0.12322275 0.87677725)
##       6) wf_hp>=-0.111882 29   3 no (0.89655172 0.10344828) *
##       7) wf_hp< -0.111882 393  26 yes (0.06615776 0.93384224)
##        14) wf_edu>=0.1654729 7   1 no (0.85714286 0.14285714) *
##        15) wf_edu< 0.1654729 386  20 yes (0.05181347 0.94818653)
##          30) capital_run_length_longest< -0.3501961 11   4 no (0.63636364 0.36363636) *
##          31) capital_run_length_longest>=-0.3501961 375  13 yes (0.03466667 0.96533333)
##            62) cf_exclam< -0.338901 64   8 yes (0.12500000 0.87500000)
##             124) wf_you< -0.2182566 25   8 yes (0.32000000 0.68000000)
##               248) wf_you>=-0.5570484 9   3 no (0.66666667 0.33333333) *
##               249) wf_you< -0.5570484 16   2 yes (0.12500000 0.87500000) *
##             125) wf_you>=-0.2182566 39   0 yes (0.00000000 1.00000000) *
##            63) cf_exclam>=-0.338901 311   5 yes (0.01607717 0.98392283) *
```

Nous avons un arbre assez grand (28 feuilles). Nous pouvons comptabiliser le nombre d'observations dans chaque feuille :

```
#numéros des feuilles
#et nombre d'observations dans chaque feuille
table(arbre.seul$where)

##
##   8  11  13  14  15  16  18  20  22  24  26  27  29  30  33  34  36  37
## 737  26  49   7   8  12  20  33   9   7  11  18  15  12  32  12   8  71
##  40  41  42  44  46  48  52  53  54  55
##   7  13  72  29   7  11   9  16  39 311
```

L'effectif médian dans les feuilles est :

```
#médiane du nombre d'observations dans les feuilles
median(table(arbre.seul$where))

## [1] 14
```

Nous appliquons l'arbre sur l'échantillon test pour obtenir les prédictions.

```
#prédiction des arbres
pred.arbre <- predict(arbre.seul,newdata=spam.test.cr,type="class")
print(table(pred.arbre))

## pred.arbre
##   no  yes
## 1921 1079
```

Sur les 3000 observations, 1921 sont prédites "saines" et 1079 "frauduleuses".

Confrontons ces prédictions avec les classes observées. Nous utilisons l'outil **confusionMatrix()** du package "caret" qu'il faudra installer au préalable s'il n'est pas présent sur votre machine.

```
#CARET pour évaluation
library(caret)

## Loading required package: lattice

## Loading required package: ggplot2

#taux de reconnaissance
caret::confusionMatrix(data=pred.arbre,reference=spam.test.cr$spam,positive="yes")

## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1714  207
##        yes  102  977
##
##                Accuracy : 0.897
##                  95% CI : (0.8856, 0.9077)
##     No Information Rate : 0.6053
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.7811
##  Mcnemar's Test P-Value : 3.292e-09
##
##             Sensitivity : 0.8252
##             Specificity : 0.9438
##          Pos Pred Value : 0.9055
##          Neg Pred Value : 0.8922
##              Prevalence : 0.3947
##          Detection Rate : 0.3257
##    Detection Prevalence : 0.3597
##       Balanced Accuracy : 0.8845
##
##        'Positive' Class : yes
##
```

La proportion de prédictions correctes (taux de reconnaissance, taux de succès, accuracy, à ne pas confondre avec précision qui est un autre type d'indicateur) est de **89.7%**. D'autres indicateurs sont disponibles, nécessitant l'indication de la modalité de référence (positive = "yes") lors de l'appel de la fonction. Ils ne nous intéressent pas vraiment dans notre contexte, nous les ignorerons par la suite.

Les méthodes d'agrégation peuvent également s'appuyer sur les probabilités d'appartenance aux classes (section 3.3). L'option "type = prob" de la fonction **predict()** permet de les produire.

```
#probabilité d'appartenance aux classes
proba.arbre <- predict(arbre.seul,newdata=spam.test.cr,type="prob")
print(head(proba.arbre))
```

```
##            no        yes
## 1  0.11267606 0.88732394
## 2  0.96200814 0.03799186
## 3  0.96200814 0.03799186
## 4  0.96200814 0.03799186
## 7  0.85714286 0.14285714
## 10 0.01607717 0.98392283
```

Nous avons une matrice de taille (3000 x 2), 3000 est l'effectif de l'échantillon test, 2 parce la variable cible est binaire.

Analyse discriminante linéaire

Nous utilisons la fonction **lda()** du package "MASS" pour l'analyse discriminante linéaire. La construction du modèle n'appelle pas de commentaires particulier, l'algorithme n'est pas paramétré.

```
#MASS pour lda
library(MASS)
adl.seul <- lda(spam ~ ., data = spam.train.cr)
print(adl.seul)
```

```
## Call:
## lda(spam ~ ., data = spam.train.cr)
##
## Prior probabilities of groups:
##        no       yes
## 0.6071205 0.3928795
##
## Group means:
##        wf_make  wf_address     wf_all       wf_3d     wf_our    wf_over
## no  -0.1132256  0.02118491 -0.1547263 -0.04874549 -0.1918875 -0.2266123
## yes  0.1749687 -0.03273725  0.2391001  0.07532689  0.2965257  0.3501863
##      wf_remove wf_internet   wf_order    wf_mail wf_receive      wf_will
## no  -0.3122369  -0.1413906 -0.2230975 -0.1119039 -0.1871421 -0.004811801
## yes  0.4825029   0.2184922  0.3447548  0.1729261  0.2891926  0.007435725
##      wf_people   wf_report wf_addresses    wf_free wf_business   wf_email
## no  -0.1054094 -0.02381489   -0.1637344 -0.1936171  -0.2207466 -0.1738400
## yes  0.1628902  0.03680139    0.2530205  0.2991984   0.3411219  0.2686368
##         wf_you  wf_credit    wf_your     wf_font     wf_000   wf_money
## no  -0.2241848 -0.1730198 -0.2930203 -0.07610661 -0.2873577 -0.2404590
## yes  0.3464351  0.2673692  0.4528071  0.11760830  0.4440567  0.3715837
```

```
##          wf_hp     wf_hpl     wf_lab    wf_labs  wf_telnet      wf_857
## no   0.1958504  0.1772099  0.1130300  0.1438991  0.1133507  0.09543147
## yes -0.3026496 -0.2738442 -0.1746664 -0.2223688 -0.1751620 -0.14747120
##        wf_data     wf_415      wf_85 wf_technology    wf_1999    wf_parts
## no   0.1131193  0.0940173  0.1041254     0.1141786  0.1526898  0.02336632
## yes -0.1748043 -0.1452859 -0.1609060    -0.1764413 -0.2359532 -0.03610821
##          wf_pm   wf_direct       wf_cs wf_meeting wf_original  wf_project
## no   0.1023310  0.05932150  0.07159287  0.1182131   0.1019003  0.06911393
## yes -0.1581331 -0.09167011 -0.11063318 -0.1826759  -0.1574675 -0.10680245
##          wf_re     wf_edu    wf_table wf_conference    cf_comma
## no   0.1288881  0.1098916  0.04285225    0.07783434  0.05237139
## yes -0.1991720 -0.1698166 -0.06622001   -0.12027818 -0.08093004
##      cf_bracket cf_sqbracket  cf_exclam  cf_dollar     cf_hash
## no   0.09719439   0.05335515 -0.2852665 -0.2700427 -0.06012122
## yes -0.15019546  -0.08245025  0.4408252  0.4172998  0.09290592
##     capital_run_length_average capital_run_length_longest
## no                  -0.07202909                 -0.2446327
## yes                  0.11130727                  0.3780333
##     capital_run_length_total
## no                 -0.1928704
## yes                 0.2980445
##
## Coefficients of linear discriminants:
##                                    LD1
## wf_make                     -0.088580368
## wf_address                  -0.036771727
## wf_all                       0.036414358
## wf_3d                        0.086621626
## wf_our                       0.284048583
## wf_over                      0.187737687
## wf_remove                    0.389454873
## wf_internet                  0.139458469
## wf_order                     0.116358647
## wf_mail                      0.021307481
## wf_receive                   0.032789748
## wf_will                     -0.062745574
## wf_people                   -0.050996359
## wf_report                    0.010665454
## wf_addresses                -0.032192305
## wf_free                      0.195852943
## wf_business                  0.104109380
## wf_email                     0.096811721
## wf_you                       0.106428170
## wf_credit                    0.079661207
## wf_your                      0.251820065
## wf_font                      0.196937265
## wf_000                       0.234142944
## wf_money                     0.243234580
## wf_hp                       -0.109914152
## wf_hpl                      -0.063837938
## wf_lab                      -0.002320882
## wf_labs                     -0.154682339
## wf_telnet                   -0.114918545
## wf_857                      -0.168264471
## wf_data                     -0.115045776
```

```
## wf_415                        0.071496360
## wf_85                        -0.044131125
## wf_technology                 0.112779694
## wf_1999                      -0.072670354
## wf_parts                     -0.073239022
## wf_pm                        -0.034378968
## wf_direct                     0.228516293
## wf_cs                         0.006340388
## wf_meeting                   -0.124553057
## wf_original                  -0.042490168
## wf_project                   -0.049937750
## wf_re                        -0.112614359
## wf_edu                       -0.100804740
## wf_table                     -0.083824380
## wf_conference                -0.067267185
## cf_comma                     -0.137920948
## cf_bracket                   -0.004759735
## cf_sqbracket                 -0.030694632
## cf_exclam                     0.325159163
## cf_dollar                     0.208721937
## cf_hash                       0.059329556
## capital_run_length_average    0.005682026
## capital_run_length_longest    0.100648837
## capital_run_length_total      0.209460438
```

Nous avons les moyennes conditionnelles des variables et les coefficients de la fonction discriminante.

Sur l'échantillon test,

```
#prédiction ADL
pred.adl <- predict(adl.seul,newdata=spam.test.cr)$class

#perfomances
caret::confusionMatrix(data=pred.adl,reference=spam.test.cr$spam)$overall["Accuracy"]

##  Accuracy
## 0.8853333
```

… le taux de reconnaissance est de **88.5%**, un peu moins bon que l'arbre de décision. Un écart de 1.2% sur 3000 représente quand-même 36 observations correctement classées supplémentaires.

Les probabilités d'appartenance aux classes en prédiction sont obtenues via un champ spécifique de l'objet fourni par **predict()**.

```
#probabilités d'appartenance aux classes
proba.adl <- predict(adl.seul,newdata=spam.test.cr)$posterior
head(proba.adl)
```

```
##            no        yes
## 1  0.74564517 0.25435483
## 2  0.91988132 0.08011868
## 3  0.86806212 0.13193788
## 4  0.93654190 0.06345810
## 7  0.39037116 0.60962884
## 10 0.09189726 0.90810274
```

SVM avec noyau RBF

Le package "kernlab" propose plusieurs algorithmes supervisés, dont les SVM (support vector machine). Nous optons pour un noyau RBF (radial basis function). Nous fixons les paramètres : C = 1.0, coût de l'erreur de classement ; sigma = 0.1, le paramètre d'influence de chaque point. Tous deux agissent sur les propriétés de régularisation de l'algorithme (cf. une description intéressante de ces paramètres dans la documentation du package scikit-learn pour Python ou encore dans un tutoriel de Haohan).

```
#KERNLAB pour SVM RBF
set.seed(1)
library(kernlab)

##
## Attaching package: 'kernlab'

## The following object is masked from 'package:ggplot2':
##
##     alpha

svm.seul <- ksvm(spam ~ ., data = spam.train.cr, kernel = "rbfdot", C = 1.0, kpar =
                 list(sigma = 0.1), prob.model = TRUE)
print(svm.seul)

## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.1
##
## Number of Support Vectors : 976
##
## Objective Function Value : -398.8322
## Training error : 0.023735
## Probability model included.
```

Le modèle est à la hauteur de l'arbre de décision en test.

```
#prédiction svm
pred.svm <- predict(svm.seul,newdata=spam.test.cr,type="response")
```

```
#performances en test
caret::confusionMatrix(data=pred.svm,reference=spam.test.cr$spam)$overall["Accuracy"]

##  Accuracy
## 0.8903333
```

L'option "type = probabilities" de **predict()** permet de produire les probabilités d'affectation.

```
#probas d'appartenance aux classes
proba.svm <- predict(svm.seul,newdata=spam.test.cr,type="probabilities")
print(head(proba.svm))

##                no         yes
## [1,] 0.116141015 0.883858985
## [2,] 0.986329299 0.013670701
## [3,] 0.977364959 0.022635041
## [4,] 0.991098771 0.008901229
## [5,] 0.877238067 0.122761933
## [6,] 0.007130893 0.992869107
```

3.2. Vote à partir des prédictions (vote "hard")

Nous disposons de 3 classifieurs (arbre, analyse discriminante, svm) avec pour taux de reconnaissance (accuracy) en test respectivement (89.7%, 88.5%, 89%). Est-ce que combiner leurs prédictions par un vote à la majorité simple des prédictions permetrait d'améliorer les performances. D'une certaine manière, nous créons un métamodèle où les classifieurs se voient attribuer la même importance.

Nous regroupons les prédictions dans une structure unique (une matrice), nous codons 1 les prédictions "yes", 0 les "no".

```
#prédiction regroupées
pred.all <- data.frame(arbre=as.numeric(pred.arbre=="yes"),
                       adl=as.numeric(pred.adl=="yes"),svm=as.numeric(pred.svm=="yes"))
print(head(pred.all))

##   arbre adl svm
## 1     1   0   1
## 2     0   0   0
## 3     0   0   0
## 4     0   0   0
## 5     0   1   0
## 6     1   1   1
```

Pour le premier individu, arbre et svm sont d'accord et prédisent "yes", contrairement à adl. Pour le second, il y a unanimité des décisions pour "no". Etc.

La combinaison n'est intéressante que s'il y a complémentarité entre les prédictions. En effet, si les décisions sont systématiquement unanimes, les prédictions individuelles des modèles ne sont jamais remises en cause, et il est impossible d'améliorer leurs taux de reconnaissances (qui seront strictement identiques d'un modèle à l'autre d'ailleurs).

Pour vérifier la disparité des prédictions, nous calculons la matrice des corrélations, le calcul est licite puisque les variables sont binaires 0/1.

```
#corrélation entre les prédictions
cor(pred.all)

##           arbre       adl       svm
## arbre 1.0000000 0.7345705 0.7377263
## adl   0.7345705 1.0000000 0.6904505
## svm   0.7377263 0.6904505 1.0000000
```

L'arbre est plutôt d'accord avec adl et svm, ces deux derniers légèrement moins. Mais il n'y a pas unanimité dans les prédictions sur l'échantillon test (les corrélations seraient égales à 1 sinon). On peut espérer (pure conjecture) une amélioration en les combinant.

Le métamodèle correspondant à un vote à la majorité des prédictions s'écrit formellement :

Si 1 x pred.yes.arbre + 1 x pred.yes.adl + 1 x pred.yes.svm > 1.5 alors "yes" sinon "non"

Nous le traduisons comme suit sous R en partant de la matrice des prédiction pred.all.

```
#métamodèle - vote "hard"
pred.hard <- factor(ifelse(rowSums(pred.all) > 1.5,"yes","no"))
print(table(pred.hard))

## pred.hard
##   no  yes
## 1939 1061
```

La classe "yes" est attribuée à 1061 observations, 1939 pour "no".

Mesurons le taux de reconnaissance en confrontant ces prédictions avec les classes observées sur l'échantillon test.

```
#performances du vote "hard"
caret::confusionMatrix(data=pred.hard,reference=spam.test.cr$spam)$overall["Accuracy"
]

##  Accuracy
## 0.9243333
```

Nous sommes "largement" meilleur que l'arbre qui présentait le taux de plus favorable (89.7%). L'écart (0.924 – 0.897) x 3000 = 81 observations correctement classés supplémentaires. Ça valait vraiment le coup de passer par la combinaison de modèles, même via une approche très fruste de vote à la majorité des prédictions.

3.3. Vote à partir des probabilités d'affectation (vote "soft")

Réitérons la même démarche, mais en utilisant les probabilités estimées d'affectation aux classes. On parle de vote "soft" cette fois-ci. Le métamodèle s'écrit maintenant :

Si 1 x proba.yes.arbre + 1 x proba.yes.adl + 1 x proba.yes.svm > 1.5 alors ''yes'' sinon ''no''

Tout d'abord, nous réunissons dans une structure unique les probabilités estimées P(Y = yes / X) fournies par les modèles sur l'échantillon test.

```
#probas. "yes" regroupées
proba.all <-
data.frame(arbre=proba.arbre[,"yes"],adl=proba.adl[,"yes"],svm=proba.svm[,"yes"])
print(head(proba.all))

##         arbre        adl         svm
## 1  0.88732394 0.25435483 0.883858985
## 2  0.03799186 0.08011868 0.013670701
## 3  0.03799186 0.13193788 0.022635041
## 4  0.03799186 0.06345810 0.008901229
## 7  0.14285714 0.60962884 0.122761933
## 10 0.98392283 0.90810274 0.992869107
```

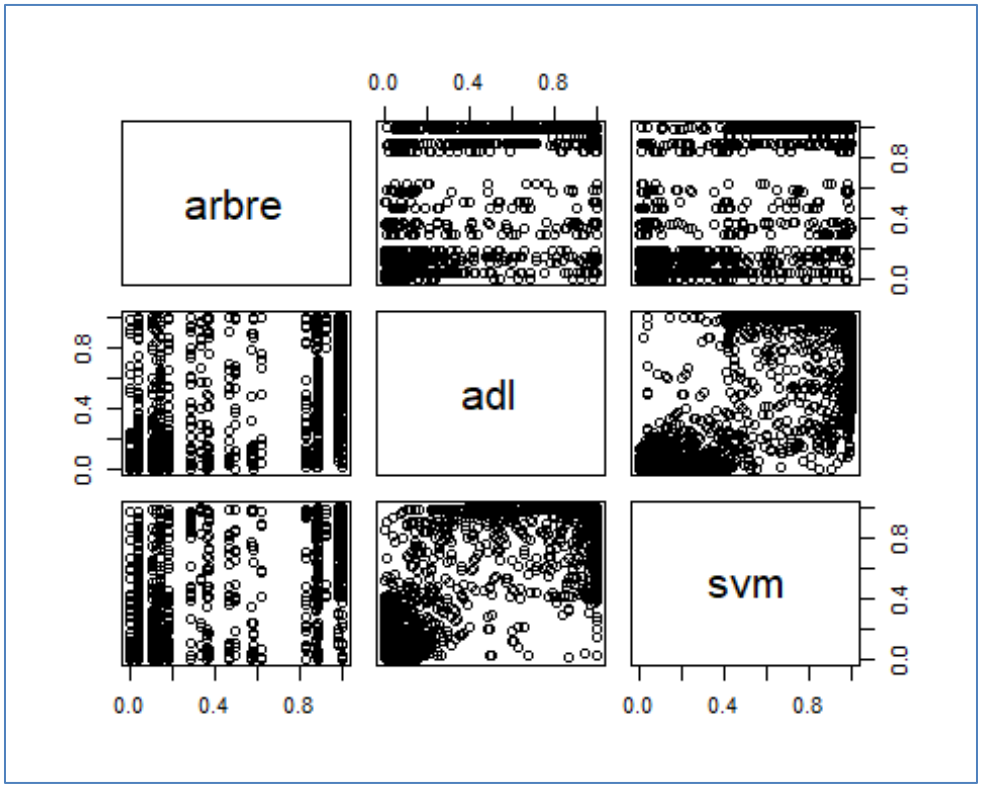
Voyons de nouveau le degré d'unanimité des valeurs en calculant la corrélation. Par rapport aux prédictions codées 0/1, les valeurs sont un peu différentes, mais la structure des corrélations reste la même.

```
#corrélations
cor(proba.all)

##           arbre      adl       svm
## arbre 1.0000000 0.826517 0.8195864
## adl   0.8265170 1.000000 0.8065760
## svm   0.8195864 0.806576 1.0000000
```

Puisque nous avons des probabilités, nous pouvons analyser plus finement les associations entre les modèles via des graphiques nuages de points.

```
#scatter croisés
pairs(proba.all)
```



Quelques commentaires :

- L'arbre présente de nombreuses valeurs ex-aequo. Ce n'est pas étonnant. Les individus situés dans la même feuille se voient attribuer la même probabilité d'appartenance aux classes.
- L'adl présente de nombreuses valeurs saturées à 0 et 1. Il y a un agglutinement fort autour des faibles valeurs. Nous en avons la confirmation en calcul les statistiques descriptives (**summary**) où la médiane des probabilités "yes" est 0.086.
- Svm aussi présente aussi des valeurs saturées à 1. L'asymétrie de la distribution est moins prononcée néanmoins.

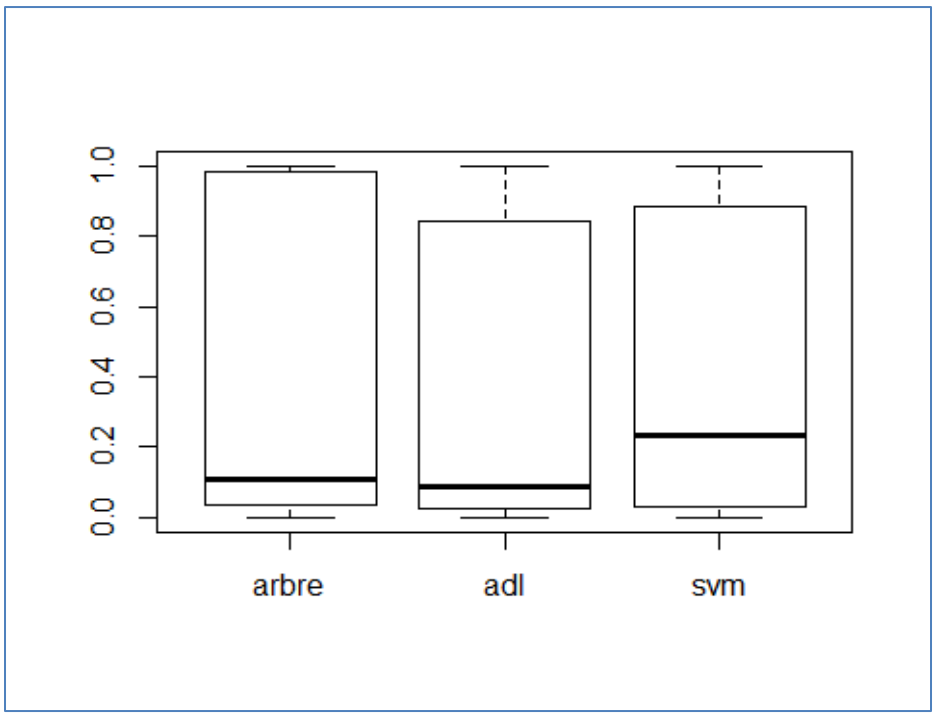
Nous avons une autre lecture de ces constatations avec **summary()**.

```
#calibration - comparabilité des probas
summary(proba.all)

##      arbre              adl                svm          
##  Min.   :0.00000   Min.   :0.0000002   Min.   :0.0003772  
##  1st Qu.:0.03799   1st Qu.:0.0269612   1st Qu.:0.0295652  
##  Median :0.11111   Median :0.0865870   Median :0.2324852  
##  Mean   :0.39414   Mean   :0.3522569   Mean   :0.3970326  
##  3rd Qu.:0.98392   3rd Qu.:0.8419586   3rd Qu.:0.8872958  
##  Max.   :1.00000   Max.   :1.0000000   Max.   :0.9999604
```

Des boxplot permettent également de confirmer le diagnostic.

```
#graphiquement avec des boxplot
boxplot(proba.all)
```



Même si les probabilités ne sont pas très bien calibrées (uniformément réparties entre [0,1]), les distributions d'un classifieur à l'autre restent comparables. Les associer reste possible, ne serait pas choquant tout du moins. Il y aurait des sérieux problèmes si par exemple, une distribution serait très resserrée autour de 0.5 pour un des modèles, et que d'autres présentent plutôt des valeurs saturées à 0 et/ou à 1.

Voyons les performances en prédiction du vote "soft".

```
#vote "soft"
pred.soft <- factor(ifelse(rowSums(proba.all) > 1.5, "yes", "no"))

#performances du vote "soft"
caret::confusionMatrix(data=pred.soft,reference=spam.test.cr$spam)$overall["Accuracy"]

##  Accuracy
## 0.9313333
```

Nous avons un léger gain par rapport au vote "hard" : (0.9313-0.9243) x 3000 = 19 individus bien classés supplémentaires. Utiliser les probabilités d'appartenance aux classes

pour combiner les modèles (abre, adl et svm) s'est avéré bénéfique pour notre jeu de donnes "spambase."

3.4. Stacking

L'idée du stacking est d'utiliser les données mêmes pour calculer les pondérations "optimales" des modèles dans le métaclassifieur. Nous avons un deuxième étage de modélisation où les variables prédictives sont constituées par les prédictions des modèles du premier niveau, la variable cible jouant son rôle usuel.

L'ennui est que nous ne pouvons pas utiliser l'échantillon test pour élaborer ce second étage. Il doit conserver son rôle de juge impartial, ne participant en aucune manière au processus de modélisation.

L'ennui (bis) est que nous ne pouvons pas utiliser les prédictions en resubstitution (prédiction sur les données ayant servi à l'apprentissage) non plus parce que ce serait favoriser les modèles ayant une propension au surapprentissage. En effet, collant exagérément aux données, leurs prédictions sont systématiquement correctes, la post-modélisation leur attribuerait alors un poids exagéré, qui ne reflète en rien leur comportement en généralisation.

La validation croisée est l'artifice qui permet de dépasser ces obstacles. Le processus est très bien résumé dans la documentation en ligne de H2O (Super Learner Algorithm). Pour générer les données utilisables pour le second étage de modélisation, nous regroupons les prédictions sur les k-èmes folds (blocs) dans un processus de K-fold validation croisée. Bien sûr, pour que les prédictions soient comparables, il faut utiliser exactement les mêmes répartitions en folds pour l'ensemble des modèles qui vont constituer le pool.

Nous souhaitons créer un index qui permet d'attribuer un numéro de bloc à chaque individu de l'échantillon d'apprentissage. Nous partons sur l'idée d'une 5-fold validation croisée, les blocs sont numérotés de 1 à 5. Voici la démarche : nous générons des valeurs aléatoires avec **runif()**, nous calculons la position de chaque valeur avec **order()** et nous la transformons en un ensemble de valeurs comprises entre 1 et 5 avec un modulo.

```
#nombre de folds
nfolds <- 5

#création des 5 blocs
```

```
set.seed(100)
index <- 1 + order(runif(nrow(spam.train))) %% nfolds
print(head(index,20))

##  [1] 2 4 3 3 4 5 4 5 3 3 2 5 4 1 5 3 2 4 4 2
```

Le premier individu de l'échantillon d'apprentissage est dans le bloc (fold) n°2, le second dans le n°4, etc.

Contrôlons les effectifs par blocs.

```
#nombre d'observations par bloc
print(table(index))

## index
##   1   2   3   4   5
## 320 321 320 320 320
```

La répartition est équilibrée. Toutefois, 1601 n'étant pas divisible par 5, l'individu supplémentaire a été attribué (au hasard) au bloc numéro 2.

Nous pouvons maintenant lancer la succession d'apprentissage-test de la validation croisée pour chaque algorithme. Nous obtenons une matrice de données avec 1601 lignes (taille de l'échantillon d'apprentissage) et 3 colonnes (3 modèles qui composent le pool).

```
#matrice pour collecter les résultats de prédictions en validation croisée
mPredCv <- matrix(0,nrow=nrow(spam.train),ncol=3)

#nommer les colonnes
colnames(mPredCv) <- c("arbre","adl","svm")

#sessions apprentissage-test
for (k in 1:nfolds){
  ############################
  #### ARBRE DE DECISION ####
  ############################
  #apprentissage arbre sur les autres que le fold numéro k
  arbreCv <- rpart(spam ~ ., data = spam.train.cr[index!=k,], control =
rpart.control(cp = 0))
  #prédiction arbre sur le fold numéro k
  predArbreCV <- as.numeric(predict(arbreCv,newdata=spam.train.cr[index==k,],type="class")=="yes")
  #intégration des prédictions dans la matrice
  mPredCv[index==k,"arbre"] <- predArbreCV
  ################################
  #### ANALYSE DISCRIMINANTE ####
  ################################
  #apprentissage
  adlCv <- lda(spam ~ ., data = spam.train.cr[index!=k,])
  #prédiction
  predAdlCv <- as.numeric(predict(adlCv,newdata=spam.train.cr[index==k,])$class == "yes")
  #récupération
```

```
  mPredCv[index==k,"adl"] <- predAdlCv
  ##################
  #### SVM RBF ####
  ##################
  #apprentissage
  svmCv <- ksvm(spam ~ ., data = spam.train.cr[index!=k,], scale = FALSE, kernel =
"rbfdot", C = 1.0, kpar = list(sigma = 0.1), prob.model = TRUE)
  #prédiction
  predSvmCv <- as.numeric(predict(svmCv,newdata=spam.train.cr[index==k,],type="response")=="yes")
  #récupération de la prédiction
  mPredCv[index==k,"svm"] <- predSvmCv
}

#vérification
print(head(mPredCv,20))
```

```
##       arbre adl svm
##  [1,]     0   0   0
##  [2,]     0   0   0
##  [3,]     0   0   0
##  [4,]     0   1   0
##  [5,]     0   0   0
##  [6,]     1   1   1
##  [7,]     1   0   1
##  [8,]     1   1   1
##  [9,]     0   0   0
## [10,]     0   1   1
## [11,]     0   0   0
## [12,]     0   0   0
## [13,]     0   0   0
## [14,]     1   0   0
## [15,]     1   1   1
## [16,]     1   1   1
## [17,]     1   1   0
## [18,]     0   0   0
## [19,]     1   0   1
## [20,]     0   0   0
```

Il y a unanimité des décisions pour les 3 premiers individus, par pour le 4ème, etc.

Nous accolons à cette matrice la variable cible :

```
#adjonction de la variable cible
dataCv <- data.frame(cbind(spam.train.cr["spam"],data.frame(mPredCv)))
print(head(dataCv))
```

```
##      spam arbre adl svm
## 1222   no     0   0   0
## 1712   no     0   0   0
## 2635   no     0   0   0
## 4176  yes     0   1   0
## 928    no     0   0   0
## 4129  yes     1   1   1
```

Et nous pouvons maintenant lancer la modélisation à l'aide de la régression logistique (**glm** de R) parce qu'elle peut fournir des coefficients qui s'apparentent à des pondérations attribuées aux modèles dans le vote.

```
#modélisation stacking
modelStack <- glm(spam ~ ., data = dataCv, family="binomial")
print(modelStack)

##
## Call:  glm(formula = spam ~ ., family = "binomial", data = dataCv)
##
## Coefficients:
## (Intercept)        arbre          adl          svm
##      -2.933        2.302        2.539        2.288
##
## Degrees of Freedom: 1600 Total (i.e. Null);  1597 Residual
## Null Deviance:       2145
## Residual Deviance: 775.4     AIC: 783.4
```

Le métamodèle, appris à partir des données, s'apparente à un vote pondéré. Il s'écrit maintenant :

Si 2.302 x pred.yes.arbre + 2.539 x pred.yes.adl + 2.288 x pred.yes.svm > 2.933 alors ''yes'' sinon ''no''

Finalement, les coefficients présentent des valeurs comparables. Le dispositif accorde à peu près la même importance aux modèles qui composent le pool. Nous sommes assez proches du vote simple.

Appliquons ces coefficients aux prédictions sur l'échantillon test avec la fonction **predict()** de glm.

```
#utilisation de ce modèle pour pondérer les prédictions "hard"
#valeur du logit
logit.pred.Stack <- predict(modelStack,newdata = pred.all,type="link")
print(head(logit.pred.Stack))

##          1          2          3          4          5          6
##  1.6571944 -2.9325105 -2.9325105 -2.9325105 -0.3933528  4.1963521
```

`predict()` produit le logit, nous le comparons à 0 pour prédire "yes ou"no.

```
#conversion en prédiction
pred.Stack <- factor(ifelse(logit.pred.Stack > 0, "yes", "no"))

#performances
caret::confusionMatrix(data=pred.Stack,reference=spam.test.cr$spam)$overall["Accuracy"]

##  Accuracy
## 0.9243333
```

Exactement le même niveau de performance que le vote simple finalement.

Quelques commentaires :

- La procédure est généralisable. Plutôt que la régression logistique pour l'élaboration du métamodèle, nous pouvons utiliser d'autres approches. Certains packages proposent Random Forest (caretEnsemble, H20) ou le gradient boosting (H20), etc.
- Dans notre cas, stacking ne fait pas mieux que le vote simple parce que les niveaux de performances sont proches d'une méthode à l'autre, avec des corrélations assez élevées dans les prédictions. Les modèles se voient attribuer le même poids dans le métamodèle.
- Nous pouvons raffiner l'approche en introduisant les probabilités d'affectation plutôt que sur les prédictions brutes dans le métamodèle, ce que fait le package "caretEnsemble" par exemple. Nous verrons si cette variante sera profitable.

Dans les sections suivantes, nous étudions deux packages qui réalisent automatiquement l'empilement des modèles. L'avantage, maintenant que nous les avons programmés explicitement, est que nous avons parfaitement identifié les étapes des calculs, nous comprendrons mieux le séquencement des opérations et les paramétrages que demanderont ces outils.

4. Stacking avec le package "caretEnsemble"

La package caretEnsemble permet d'associer des classifieurs implémentés dans caret, librairie que j'avais longuement exploré dans un précédent tutoriel (Avril, 2018).

4.1. Paramétrage des modèles

Nous paramétrons l'apprentissage des modèles individuels dans une première étape.

```
#paramètre pour l'apprentissage
fitControl <- caret::trainControl(method="cv",number=5, classProbs = TRUE,
savePredictions = "all")
```

Si on veut assembler les classifieurs par la suite, nous devons :

- Passer par la validation croisée (method = cv) ;
- Le nombre de folds sera (number = 5) ;

- Il faudra produire les probabilités d'affectation parce qu'elles seront utilisées dans la construction du métaclassifieur (classProbs = TRUE).
- Et il faut les sauvegarder pour pouvoir les réutiliser dans la construction du métaclassifieur (savePredictions = all).

On apprend donc que "caretEnsemble" passe par les probabilités d'appartenance aux classes et non pas par les prédictions pour l'élaboration du métamodèle. Nous avons une variante élaborée du vote "soft" ci-dessus (section 3.3) où les pondérations sont calculées à partir des données issues de la validation croisée.

Nous listons ensuite les algorithmes à utiliser et leurs hyperparamètres. Ils doivent faire partie des approches répertoriées par caret.

```
#caretEnsemble
library(caretEnsemble)

##
## Attaching package: 'caretEnsemble'

## The following object is masked from 'package:ggplot2':
##
##     autoplot

#liste des modèles à empiler
#https://topepo.github.io/caret/available-models.html
arbre <- caretEnsemble::caretModelSpec(method="rpart",tuneGrid=data.frame(cp=0))
adl <- caretEnsemble::caretModelSpec(method="lda")
svmRbf <- caretEnsemble::caretModelSpec(method="svmRadial",tuneGrid=data.frame(C=1,sigma=0.1))
```

J'ai vérifié, ce sont bien les mêmes méthodes implémentées individuellement plus haut (section 3.1), avec des paramètres identiques.

Attention, si nous ne spécifions pas explicitement les paramètres, "caret" teste un jeu de valeurs pour chaque méthode et sélectionne automatiquement la combinaison qui optimise une métrique d'évaluation en validation croisée. Pourquoi pas. Mais ces opérations rallongent d'autant le temps de calcul. Cela peut surprendre si nous ne savons pas qu'une recherche en grille est effectuée en sous-main.

4.2. Apprentissage en validation croisée

Nous pouvons lancer l'élaboration des modèles en validation croisée maintenant.

```
#lancement des méthodes en validation croisée
set.seed(1)
```

```
modeles <- caretEnsemble::caretList(spam ~ ., data = spam.train.cr, trControl =
fitControl, tuneList = list(arbre,adl,svmRbf))

## Warning in trControlCheck(x = trControl, y = target): indexes not defined
## in trControl. Attempting to set them ourselves, so each model in the
## ensemble will have the same resampling indexes.

print(modeles)

## $rpart
## CART
##
## 1601 samples
##   55 predictor
##    2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 1281, 1280, 1282, 1281, 1280
## Resampling results:
##
##   Accuracy   Kappa
##   0.8875931  0.7627017
##
## Tuning parameter 'cp' was held constant at a value of 0
##
## $lda
## Linear Discriminant Analysis
##
## 1601 samples
##   55 predictor
##    2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 1281, 1280, 1282, 1281, 1280
## Resampling results:
##
##   Accuracy   Kappa
##   0.8807025  0.7420127
##
##
## $svmRadial
## Support Vector Machines with Radial Basis Function Kernel
##
## 1601 samples
##   55 predictor
##    2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 1281, 1280, 1282, 1281, 1280
## Resampling results:
##
##   Accuracy   Kappa
```

```
##   0.8857064  0.7541621
##
## Tuning parameter 'sigma' was held constant at a value of 0.1
##
## Tuning parameter 'C' was held constant at a value of 1
##
## attr(,"class")
## [1] "caretList"
```

Un message d'avertissement (en rouge) nous précise que des blocs identiques ont été conçus pour l'élaboration des modèles en validation croisée. Heureusement car, dans le cas contraire, il ne serait pas possible d'assembler les probabilités d'affectation fournies par les modèles dans une structure unique, comme nous avons pu le faire pour notre prédictions (page 18, la matrice mPredCv).

L'objet "modeles" contient des informations détaillées sur les classifieurs individuels.

```
#liste des classifieurs dans modeles
attributes(modeles)

## $names
## [1] "rpart"     "lda"       "svmRadial"
##
## $class
## [1] "caretList"
```

Si nous souhaitons accéder à "lda" par exemple.

```
#attributs de lda
attributes(modeles$lda)

## $names
##  [1] "method"       "modelInfo"    "modelType"    "results"
##  [5] "pred"         "bestTune"     "call"         "dots"
##  [9] "metric"       "control"      "finalModel"   "preProcess"
## [13] "trainingData" "resample"     "resampledCM"  "perfNames"
## [17] "maximize"     "yLimits"      "times"        "levels"
## [21] "terms"        "coefnames"    "xlevels"
##
## $class
## [1] "train"         "train.formula"
```

Nous pouvons afficher les prédictions dans les blocs de validation croisée. Pour "lda",

```
#prédictions en validation croisée pour lda
head(modeles$lda$pred,20)

##    pred obs           no        yes rowIndex parameter Resample
## 1    no  no 0.9197899343 0.08021007        3      none    Fold1
## 2    no  no 0.8561632479 0.14383675        5      none    Fold1
## 3   yes yes 0.0006983142 0.99930169        8      none    Fold1
```

```
## 4    no  no 0.9742086243 0.02579138        20      none    Fold1
## 5    no  no 0.9367546565 0.06324534        22      none    Fold1
## 6   yes yes 0.0094532375 0.99054676        28      none    Fold1
## 7   yes  no 0.3550272108 0.64497279        37      none    Fold1
## 8    no yes 0.5350599755 0.46494002        44      none    Fold1
## 9    no  no 0.6994514770 0.30054852        48      none    Fold1
## 10   no  no 0.9796427179 0.02035728        50      none    Fold1
## 11   no  no 0.9308497092 0.06915029        53      none    Fold1
## 12  yes yes 0.4754487187 0.52455128        65      none    Fold1
## 13   no  no 0.9670912695 0.03290873        74      none    Fold1
## 14   no  no 0.9745493987 0.02545060        84      none    Fold1
## 15   no  no 0.9560218445 0.04397816        85      none    Fold1
## 16   no  no 0.9910620004 0.00893800        87      none    Fold1
## 17   no  no 0.9790383088 0.02096169        88      none    Fold1
## 18   no yes 0.7500463061 0.24995369        97      none    Fold1
## 19  yes yes 0.2600601336 0.73993987        98      none    Fold1
## 20  yes yes 0.0786130231 0.92138698       102      none    Fold1
```

Pour l'arbre,

```
#prédictions en validation croisée pour l'arbre
head(modeles$rpart$pred,20)
```

```
##    pred obs         no        yes rowIndex cp Resample
## 1    no  no 0.94254658 0.05745342        3  0    Fold1
## 2    no  no 0.94254658 0.05745342        5  0    Fold1
## 3   yes yes 0.01153846 0.98846154        8  0    Fold1
## 4    no  no 0.94254658 0.05745342       20  0    Fold1
## 5    no  no 0.94254658 0.05745342       22  0    Fold1
## 6   yes yes 0.01153846 0.98846154       28  0    Fold1
## 7    no  no 0.94254658 0.05745342       37  0    Fold1
## 8   yes yes 0.01153846 0.98846154       44  0    Fold1
## 9    no  no 0.83333333 0.16666667       48  0    Fold1
## 10   no  no 0.84931507 0.15068493       50  0    Fold1
## 11   no  no 0.94254658 0.05745342       53  0    Fold1
## 12  yes yes 0.01153846 0.98846154       65  0    Fold1
## 13   no  no 0.94254658 0.05745342       74  0    Fold1
## 14   no  no 0.87500000 0.12500000       84  0    Fold1
## 15   no  no 0.94254658 0.05745342       85  0    Fold1
## 16   no  no 0.94254658 0.05745342       87  0    Fold1
## 17   no  no 0.94254658 0.05745342       88  0    Fold1
## 18  yes yes 0.01153846 0.98846154       97  0    Fold1
## 19  yes yes 0.03571429 0.96428571       98  0    Fold1
## 20  yes yes 0.03571429 0.96428571      102  0    Fold1
```

Nous remarquons que les informations "rowIndex" et "Resample" (ID des folds dans la validation croisée) sont complètement cohérentes entre l'adl et l'arbre. Il sera ainsi possible d'assembler les résultats des différents modèles pour l'élaboration du métaclassifieur. C'était le sens du message d'avertissement (en rouge) produit par l'outil lors de l'apprentissage des modèles sous-jacents individuels.

4.3. Performances individuelles des modèles en validation croisée

Les performances des modèles (arbre, adl, svm) mesurées en validation croisée sont respectivement (88.76%, 88.07%, 88.57%) (section 4.2). Nous pouvons obtenir une vision détaillée des résultats des taux de reconnaissances (accuracy) dans les folds. Remarque : Kappa est une autre métrique d'évaluation des modèles, elle n'apporte pas grand-chose de plus dans notre étude.

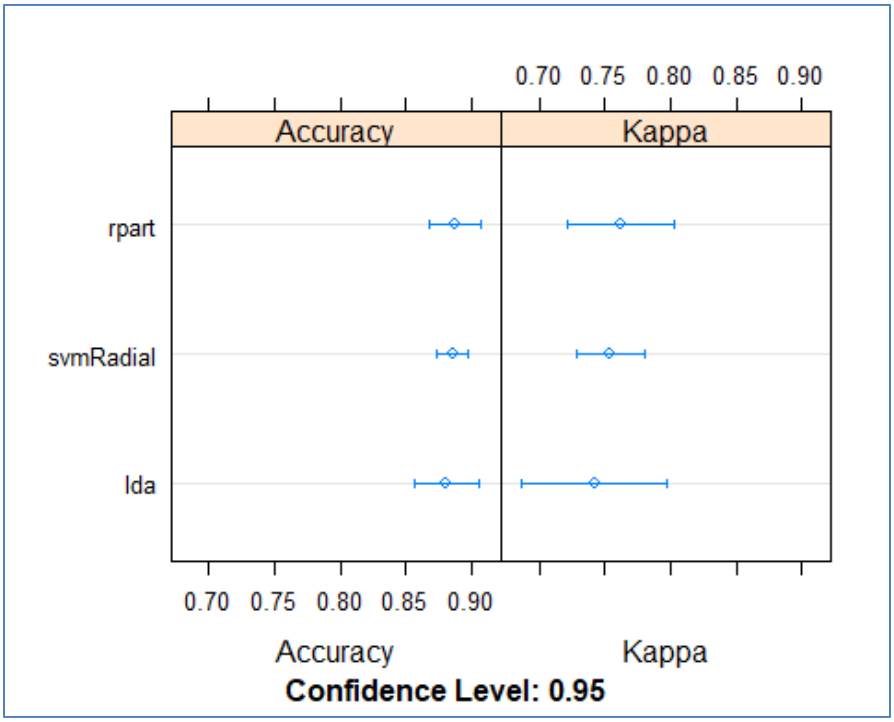
```
#collecter le détail des résultats en validation croisée
resultatsCV <- caret::resamples(modeles)

#affichage
print(summary(resultatsCV))

##
## Call:
## summary.resamples(object = resultatsCV)
##
## Models: rpart, lda, svmRadial
## Number of resamples: 5
##
## Accuracy
##                Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## rpart     0.878125 0.8781250 0.8816199 0.8875931 0.8847352 0.9153605    0
## lda       0.862500 0.8660436 0.8718750 0.8807025 0.8965517 0.9065421    0
## svmRadial 0.871875 0.8847352 0.8847352 0.8857064 0.8875000 0.8996865    0
##
## Kappa
##                 Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## rpart     0.7422127 0.7450772 0.7467929 0.7627017 0.7593167 0.8201090    0
## lda       0.6994536 0.7088896 0.7227622 0.7420127 0.7788585 0.8000996    0
## svmRadial 0.7251320 0.7509593 0.7516779 0.7541621 0.7596796 0.7833616    0
```

L'affichage graphique permet de comparer plus facilement les performances.

```
#affichage graphique "intervalle de variation"
dotplot(resultatsCV)
```



Les intervalles de confiance se chevauchent, il n'y a pas de différences significatives entre les taux de reconnaissance des modèles, évaluées en validation croisée tout du moins.

4.4. "Corrélation" des modèles

La notion de "corrélation" entre modèles en validation croisée est mise en avant dans certains tutoriels. En réalité, il s'agit des corrélations entre les taux de reconnaissances obtenus dans les folds de la validation croisée, calculés sur 5 valeurs donc en ce qui nous concerne.

La matrice des corrélations est la suivante :

```
#corrélation entre les résulats dans les folds
caret::modelCor(resultatsCV)

##                rpart       lda svmRadial
## rpart     1.0000000 0.5730294 0.8182971
## lda       0.5730294 1.0000000 0.3860206
## svmRadial 0.8182971 0.3860206 1.0000000
```

En accédant aux résultats détaillés dans les folds…

```
#taux de succès pour les folds
print(resultatsCV$values)
```

```
##   Resample rpart~Accuracy rpart~Kappa lda~Accuracy lda~Kappa
## 1    Fold1      0.8781250   0.7422127    0.8625000 0.6994536
## 2    Fold2      0.8816199   0.7467929    0.8660436 0.7088896
## 3    Fold3      0.9153605   0.8201090    0.8965517 0.7788585
## 4    Fold4      0.8781250   0.7450772    0.8718750 0.7227622
## 5    Fold5      0.8847352   0.7593167    0.9065421 0.8000996
##   svmRadial~Accuracy svmRadial~Kappa
## 1          0.8875000       0.7596796
## 2          0.8847352       0.7516779
## 3          0.8996865       0.7833616
## 4          0.8718750       0.7251320
## 5          0.8847352       0.7509593
```

… nous pouvons calculer explicitement la "corrélation" entre deux modèles (ex. arbre et adl) :

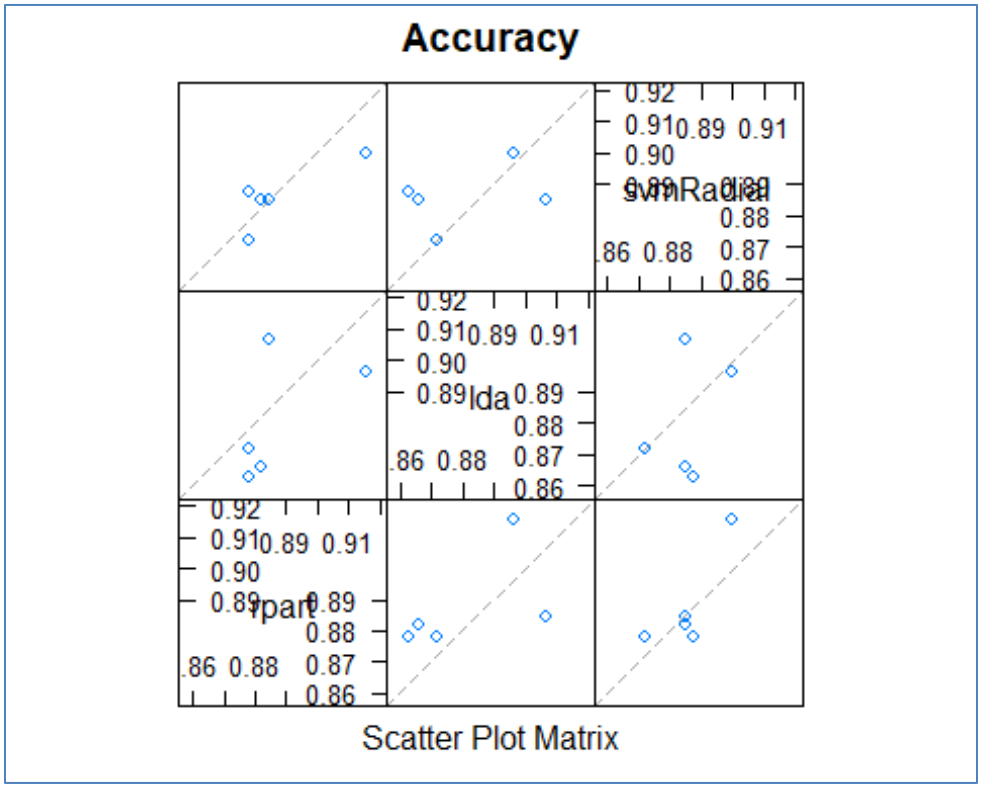
```
#vérification de la corrélation pour deux modèles
cor(resultatsCV$values["rpart~Accuracy"],resultatsCV$values["lda~Accuracy"])
```

```
##                lda~Accuracy
## rpart~Accuracy    0.5730294
```

De fait, une corrélation égale à 1, selon cette acception, ne veut pas dire que les modèles classent de la même manière. Elle indique seulement que nous avons obtenus les mêmes taux d'erreur dans chaque fold. Or, deux modèles qui présentent des taux de reconnaissance identiques n'ont pas forcément classé de la même manière tous les individus (le taux de reconnaissance dépend des faux positifs ET des faux négatifs). Je tenais à le préciser parce que la confusion est vite faite. Les tutoriels que l'on peut lire ici ou là en ligne ne sont pas très clairs sur cette histoire.

Il est possible de positionner graphiquement les taux de reconnaissance des modèles dans chaque fold (5 folds = 5 points) pour situer les modèles entre eux.

```
#affichage grahique croisé des résultats par fold
#"lattice" a été installé et chargé automatiquement
#en même temps de caret
lattice::splom(resultatsCV)
```



On notera surtout quelques disparités comme le Fold numéro 4 où l'arbre obtient de bons résultats (0.91536) contrairement à l'adl et le svm (resp. 0.89655 et 0.89968). Etc. Mais les valeurs restent comparables, il n'y a pas vraiment matière à discuter longuement ici.

4.5. Assemblage des modèles

Nous disposons des résultats en validation croisée de chaque modèle, nous pouvons lancer la construction du métaclassifieur. Pour ce second étage de modélisation, le rééchantillonnage n'est pas nécessaire, d'où l'option "method = none" dans la définition du **train.control()**. Comme dans la programmation précédemment (section 3.4), nous utilisons la régression logistique ("method = glm").

```
#stacking via la glm
#trainControl(method="none") - pas (nécessairement) besoin de resampling ici
mStack <- caretEnsemble::caretStack(modeles,method="glm",trControl=trainControl(method="none"))
print(mStack)

## A glm ensemble of 2 base models: rpart, lda, svmRadial
##
## Ensemble results:
## Generalized Linear Model
##
## 1601 samples
```

```
##    3 predictor
##    2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: None
```

Nous affichons les caractéristiques du métamodèle mStack.

```
#affichage détaillé du métamodèle (glm)
#visualisation des coefficients
summary(mStack)
```

```
##
## Call:
## NULL
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.0093  -0.2941  -0.2351   0.1914   2.6944
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -3.7881     0.1775 -21.339  < 2e-16 ***
## rpart         2.3283     0.2781   8.373  < 2e-16 ***
## lda           2.5717     0.3609   7.127 1.03e-12 ***
## svmRadial     3.5202     0.4062   8.667  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2145.40  on 1600  degrees of freedom
## Residual deviance:  699.29  on 1597  degrees of freedom
## AIC: 707.29
##
## Number of Fisher Scoring iterations: 6
```

A la lecture des coefficients, nous constatons que le svm se voit accorder plus d'importance dans le vote puisque son coefficient est plus élevé. Voyons si cela est justifié en l'appliquant sur notre échantillon test : prédiction tout d'abord avec **predict()**, puis confrontation avec les classes observées.

```
#prédiction et évaluation
pStack <- predict(mStack,newdata=spam.test.cr)
caret::confusionMatrix(spam.test.cr$spam,pStack)$overall["Accuracy"]
```

```
##  Accuracy
## 0.9333333
```

Nous sommes très proches (0.9333 vs 0.9313) du vote "soft" que nous avions mis en place plus haut (section 3.3). L'utilisation des probabilités d'affectation aux classes est réellement la bonne piste pour combiner les modèles (arbre, ald, svm) sur nos données ''spambase''.

5. Stacking avec le package "h2o"

H2O est une plateforme JAVA de machine learning. Je m'y suis beaucoup intéressé récemment (Janvier, 2019). Elle propose l'outil Stacked Ensembles pour combiner les modèles au sens du stacking.

5.1. Démarrage et préparation des données

Nous chargeons le package (après l'avoir installé) et nous démarrons le serveur en demandant toute la puissance disponible (nthreads = -1). Nous transformons les données au format reconnu par H2O (H2OFrame).

```
#library
library(h2o)

##
## ----------------------------------------------------------------------
##
## Your next step is to start H2O:
##     > h2o.init()
##
## For H2O package documentation, ask for help:
##     > ??h2o
##
## After starting H2O, you can use the Web UI at http://localhost:54321
## For more information visit http://docs.h2o.ai
##
## ----------------------------------------------------------------------

##
## Attaching package: 'h2o'

## The following objects are masked from 'package:stats':
##
##     cor, sd, var

## The following objects are masked from 'package:base':
##
##     %*%, %in%, &&, ||, apply, as.factor, as.numeric, colnames,
##     colnames<-, ifelse, is.character, is.factor, is.numeric, log,
##     log10, log1p, log2, round, signif, trunc

#initialisation
h2o.init(nthreads = -1)
```

```
##  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         47 minutes 48 seconds
##     H2O cluster timezone:       Europe/Paris
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.22.1.1
##     H2O cluster version age:    15 days
##     H2O cluster name:           H2O_started_from_R_Zatovo_cmi606
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   1.47 GB
##     H2O cluster total cores:    8
##     H2O cluster allowed cores:  8
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     H2O Internal Security:      FALSE
##     H2O API Extensions:         Algos, AutoML, Core V3, Core V4
##     R Version:                  R version 3.5.2 (2018-12-20)

#transformer en un format reconnu par h2o - training set
h2oTrain <- as.h2o(spam.train.cr)

##
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%

#idem pour le test set
h2oTest <- as.h2o(spam.test.cr)

##
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

5.2. Construction des modèles sous-jacents

Nous n'avons ni arbre de décision, ni analyse discriminante linéaire, ni SVM sous H2O. Pour le premier, nous rusons en demandant un gradient boosting machine avec 1 seul arbre. Pour le second, nous instancions un naive bayes classifier qui induit une séparation linéaire. Enfin, pour le troisième, nous créons un classifieur non linéaire avec un perceptron à 10 neurones dans l'unique couche cachée (deep learning).

Lors de la construction de ces modèles sous-jacents au stacking, trois paramètres sont essentiels pour pouvoir réaliser l'assemblage par la suite :

- Nous devons demander la validation croisée en spécifiant le nombre de folds (nfolds=5).
- Nous devons conserver les prédictions en validation croisée pour pouvoir les utiliser dans la construction du métamodèle (keep_cross_validation_predictions = TRUE).
- Et nous devons spécifier la même initialisation du générateur de nombre aléatoire pour que ces prédictions en validation croisée puissent être consolidées dans la même structure (seed = 1 ; en tous les cas, la même valeur pour les 3 modèles).

Pour l'arbre, nous créons le modèle et nous mesurons le taux de reconnaissance en test.

```
#premier modèle - 1 arbre via gradient boosting
h2o_arbre <- h2o.gbm(y="spam",training_frame=h2oTrain,ntrees=1,nfolds=5,
keep_cross_validation_predictions = TRUE,seed=1)

#ses performances en test - prédiction + confrontation avec les classes observées
caret::confusionMatrix(spam.test.cr$spam,as.data.frame(h2o.predict(h2o_arbre,newdata=
h2oTest))$predict)$overall["Accuracy"]

## Accuracy
##    0.901
```

Pour le Naive Bayes,

```
#second modèle - naive bayes
h2o_nb <- h2o.naiveBayes(y="spam",training_frame=h2oTrain,nfolds=5,
keep_cross_validation_predictions = TRUE,seed=1)

#perfs en test
caret::confusionMatrix(spam.test.cr$spam,as.data.frame(h2o.predict(h2o_nb,newdata=h2o
Test))$predict)$overall["Accuracy"]

##  Accuracy
## 0.7433333
```

Enfin, pour le réseau de neurones,

```
#perceptron à 10 neurones dans la couche cachée
h2o_mlp <- h2o.deeplearning(y="spam",training_frame=h2oTrain,hidden=c(10),nfolds=5,
keep_cross_validation_predictions = TRUE,seed=1)


#perfs en test
caret::confusionMatrix(spam.test.cr$spam,as.data.frame(h2o.predict(h2o_mlp,newdata=h2
oTest))$predict)$overall["Accuracy"]

##  Accuracy
## 0.9216667
```

Par rapport aux autres modèles vus précédemment, le naive bayes, avec un taux de reconnaissance de 74.33%, est largement en deçà. A l'inverse, l'arbre (90.1%) et le perceptron multicouche (92.33%) se placent favorablement.

Cette étape de construction des modèles en validation croisée est requise par l'outil qui permettra de les assembler par la suite.

5.3. Construction et évaluation du métamodèle

La fonction **stackedEnsemble()** permet d'assembler les modèles. Nous les énumérons avec l'option base_models. Puis nous indiquons l'algorithme à utiliser pour construire le métaclassifieur. Nous faisons le choix encore une fois de la régression logistique (metalearner_algorithm = "glm"). D'autres algorithmes sont possibles dixit la documentation de H2O (gradient boosting, random forest, réseau de neurones).

J'ai fixé (seed=2019) pour que vous puissiez reproduire l'expérimentation à l'identique. Mais cette option est moins cruciale à ce stade. De même, il n'est pas nécessaire d'effectuer une validation croisée.

```
#modèle empilé
h2o_stack <- h2o.stackedEnsemble(y="spam",training_frame=h2oTrain, base_models =
list(h2o_arbre,h2o_nb,h2o_mlp),metalearner_algorithm = "glm",seed=2019)

print(h2o_stack)
```

```
## Model Details:
## ==============
##
## H2OBinomialModel: stackedensemble
## Model ID:  StackedEnsemble_model_R_1547362471169_221
## Number of Base Models: 3
##
## Base Models (count by algorithm type):
##
## deeplearning          gbm   naivebayes
##            1            1            1
##
## Metalearner:
##
## Metalearner algorithm: glm
##
##
## H2OBinomialMetrics: stackedensemble
## ** Reported on training data. **
##
```

```
## MSE:  0.05017092
## RMSE:  0.2239887
## LogLoss:  0.1843156
## Mean Per-Class Error:  0.06710223
## AUC:  0.976398
## pr_auc:  0.9170852
## Gini:  0.9527959
##
## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
##         no yes    Error       Rate
## no     925  47 0.048354    =47/972
## yes     54 575 0.085851    =54/629
## Totals 979 622 0.063086  =101/1601
##
## Maximum Metrics: Maximum metrics at their respective thresholds
##                         metric threshold    value idx
## 1                       max f1  0.414886 0.919265 179
## 2                       max f2  0.191515 0.928415 236
## 3                 max f0point5  0.684447 0.943726 121
## 4                 max accuracy  0.470718 0.936914 170
## 5                max precision  0.975077 1.000000   0
## 6                   max recall  0.022032 1.000000 389
## 7              max specificity  0.975077 1.000000   0
## 8             max absolute_mcc  0.414886 0.867536 179
## 9   max min_per_class_accuracy  0.262494 0.926868 207
## 10 max mean_per_class_accuracy  0.414886 0.932898 179
##
## Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or
`h2o.gainsLift(<model>, valid=<T/F>, xval=<T/F>)`
```

Les sorties correspondent à celles que produisent habituellement H2O pour les modèles prédictifs. Nous notons que le taux d'erreur en resubstitution est de 93.69%, avec un seul d'affectation ajusté à 0.470718 (une des particularités de H2O, voir section 3.2).

Voyons ce que cela donne sur l'échantillon test.

```
#ses perfromances
caret::confusionMatrix(spam.test.cr$spam,as.data.frame(h2o.predict(h2o_stack,newdata=
h2oTest))$predict)$overall["Accuracy"]

## Accuracy
##    0.927
```

Nous sommes dans les niveaux de performances des autres approches étudiées dans les sections précédentes. Notons néanmoins que la prédiction de H2O est calibrée (seuil = 0.414886) pour optimiser le F1-Score. Pour que les résultats soient réellement comparables avec les autres packages, nous aurions dû utiliser le seuil d'affectation usuel de 0.5 ou, tout du moins, celui que H2O nous a indiqué dans ses sorties ci-dessus (0.470718).

6. Conclusion

Je connaissais le stacking depuis un moment déjà. J'avais planché l'article de Wolpert (1992) en son temps. Je m'étais dit qu'il fallait que j'écrive un tutoriel dessus en le programmant à partir de zéro pour bien détailler les étapes (section 3). Puis la vie est ainsi faite qu'il y a toujours des choses urgentes à faire, j'avais garé l'idée dans un recoin de ma tête. En étudiant H2O récemment, je me suis rendu compte qu'elle proposait des outils intéressants pour l'assemblage des modèles (section 5). Ce qui m'a amené à effectuer quelques recherches sur le net pour découvrir "caretEnsemble" (section 4). Je me suis dit qu'il était temps de rédiger un petit document permettant, d'une part, de décrire sur des exemples simples les mécanismes du stacking, d'autre part, de cerner les possibilités des outils sous R en la matière.

7. Références

Wolpert D., "Stacked Generalization", Neural Networks, 5:241:259, 1992.

Brownlee J., "How to Build an Ensemble of Machine Learning Algorithms in R (ready to use boosting, bagging and stacking)", Février, 2016.

H2O Documentation, "Stacked Ensembles", Version 3.22.1.1.

Mayer Z., "A Brief Introduction to caretEnsemble", Janvier, 2016.