Machine Learning sous R avec le package ''mlr''

Tutoriel Tanagra

1 Introduction

La profusion des packages est à la fois une force et une faiblesse de R. Une force parce que nous disposons d'une richesse telle qu'il est possible de trouver un package qui réponde à nos besoins de traitements, quels qu'ils soient (presque). Une faiblesse parce que, en l'absence d'une coordination forte, ils adoptent souvent des modes opératoires disparates qui déroutent les utilisateurs. C'est en ces termes que j'introduisais le package "caret" qui se propose d'unifier la pratique du machine learning sous R dans un moule unique.

Je pourrais tenir exactement le même discours en ce qui concerne la librairie "mlr" que je présente dans ce tutoriel. Tout comme "caret", "mlr" fait le pont vers un très grand nombre d'autres librairies dédiées au machine learning sous R. Son principal mérite est d'intégrer dans un ensemble cohérent les étapes clés de l'analyse prédictive. Instancier un modèle, en calculer les paramètres, effectuer une prédiction et mesurer les performances, s'effectue avec la même syntaxe quelle que soit la méthode (et le package sous-jacent) appelée. Cette standardisation est un atout déterminant pour les traitements à grande échelle. Mais, a contrario, elle nous impose de nous familiariser avec un mode opératoire spécifique qui, heureusement, sera reproductible *ad vitam* une fois que nous nous serions coulés dans le cadre de "mlr". C'est tout l'intérêt de ce type de méta-package.

Nous traiterons d'un exemple (assez amusant) de "football mining" tiré de l'excellent ouvrage de Zhao et Cen (2014) pour détailler les fonctionnalités de "mlr". Précisons quand-même une information importante avant que vous ne commenciez à exécuter le code source qui accompagne ce document, "mlr" ne charge pas automatiquement les dépendances. Il nous revient de les installer manuellement à chaque fois que le programme plante parce qu'il ne trouve pas les packages sous-jacents. C'est un peu déconcertant (et pénible) au début de voir surgir les messages d'erreur en rouge dans la console, mais bon, on s'y fait au bout d'un moment.

2 Données

Les données sont référencées dans le chapitre "Football Mining in R" de l'ouvrage "Data Mining Applications with R" (2014). Elles décrivent les résultats de matchs de la "Série A" italienne, saison 2010-2011. Elles sont recensées du point de vue de l'équipe hôte (résultat = victoire, défaite [victoire de l'équipe visiteuse] ou nul). Nous disposons de 10 rencontres x 38 journées (380 observations) puisque le championnat est composé de 20 équipes. Les variables explicatives potentielles correspondent aux caractéristiques des matches : nombre d'occasions, nombre de tirs, nombre de passes, etc. J'ai supprimé la variable "météo" qui était qualitative, requérant un prétraitement spécifique qui m'aurait sorti du cadre de ce didacticiel.

Nous importons les données dans un premier temps et nous affichons la liste des variables.

```
#charger les données
setwd("... votre dossier ...")
dtset <- read.table("football_data.txt",sep="\t",dec=".",header=TRUE)

#afficher liste des variables
print(str(dtset))

## 'data.frame':    380 obs. of  481 variables:
##  $ RIS              : Factor w/ 3 levels "1","2","X": 1 3 1 3 1 3 1 3 1 2 ...
##  $ G_MINUTI_C       : int  97 98 95 94 96 94 96 97 94 97 ...
…
##  $ D_PALLE11_C      : int  37 54 27 22 13 26 29 30 38 31 ...
##   [list output truncated]
```

Nous disposons de 481 variables. RIS est la cible, avec 3 modalités :

```
#valeurs de la cible
print(table(dtset$RIS))

##
##   1   2   X
## 180 104  96
```

Le code '1' correspond à une victoire de l'équipe hôte, il y a eu 180 victoires à domicile sur la saison étudiée ; '2' une défaite (victoire de l'équipe visiteuse) (104 matchs), 'X' un match nul (96 matchs).

Nous décidons de coder la variable d'intérêt en victoire / déception (défaite ou nul) dans notre étude.

```
#recodage
resmatch <- factor(ifelse(dtset$RIS=="1","victoire","deception"))
print(table(resmatch))

## resmatch
## deception  victoire
##       200       180

#en pourcentages
print(prop.table(table(resmatch)))

## resmatch
## deception  victoire
## 0.5263158 0.4736842
```

47.36% des rencontres se soldent par une victoire à domicile, ce qui correspond à peu près à ce que nous observons dans le championnat de France par exemple.

Nous définissons un ensemble de données avec cette nouvelle variable cible 'resmatch'.

```
#définir le dataset
df <- cbind(dtset[2:ncol(dtset)],resmatch)
print(dim(df))

## [1] 380 481
```

3 Machine learning avec "mlr"

Pour utiliser le package "mlr", nous devons l'installer puis le charger.

```
#importer la librairie "mlr"
library(mlr)

## Loading required package: ParamHelpers
```

Les librairies fournissant les méthodes sous-jacentes appelées par "mlr" ne sont pas automatiquement installées. La liste est longue (cf. suggests sur la page de présentation). Nous aurons à les installer explicitement au gré des utilisations que nous pourrions en faire.

3.1 Définition d'une tâche

Un des aspects intéressants de "mlr" est que nous spécifions préalablement une fois pour toutes sous forme de tâche (**task**) le problème que nous traitons : nous indiquons le jeu de données à utiliser (df), la variable cible (resmatch) et la modalité cible (victoire). Cela nous dispense d'avoir à le préciser systématiquement lorsque nous procéderons aux différents traitements (filtrage, modélisation, évaluation).

```
#définir une tâche
foot.task <-
mlr::makeClassifTask(data=df,target="resmatch",positive="victoire")
print(foot.task)
```

```
## Supervised task: df
## Type: classif
## Target: resmatch
## Observations: 380
## Features:
##    numerics     factors     ordered functionals
##         480           0           0           0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Has coordinates: FALSE
## Classes: 2
## deception  victoire
##       200       180
## Positive class: victoire
```

"mlr" affiche un bilan. Nous savons notamment que les 480 variables prédictives sont toutes quantitatives, que les fréquences des classes sont (200, 180).

3.2 Filtrage des variables

Dans une première approche, nous cherchons à identifier les variables a priori les plus liées à la cible. Cette procédure peut être un préalable à une procédure de filtrage visant à réduire le nombre de descripteurs à soumettre à la modélisation. Avec les avantages (rapidité) et inconvénients (ne tient pas compte des algorithmes d'apprentissage subséquents) que cela comporte.

Nous utilisons le critère du test non-paramétrique de comparaison de populations de Kruskal-Wallis.

```
#identifier les variables les plus liées avec la cible - vient de FSelector
filter.kruskal = mlr::generateFilterValuesData(task=foot.task,method="kruskal.test")
print(filter.kruskal)
```

```
## FilterValues:
## Task: df
##             name    type kruskal.test
## 1     G_MINUTI_C integer    2.0086095
## 2 G_PALLE_GIOC_C integer    2.7255935
## 3    G_POS_PAL_C integer    5.9052390
## 4    G_SUP_TER_C integer    0.5761523
## 5   G_VEL_GIOC_C numeric    4.2446067
```

```
## 6    G_PAS_RAP_C integer   12.0469661
## ... (#rows: 480, #cols: 3)
```

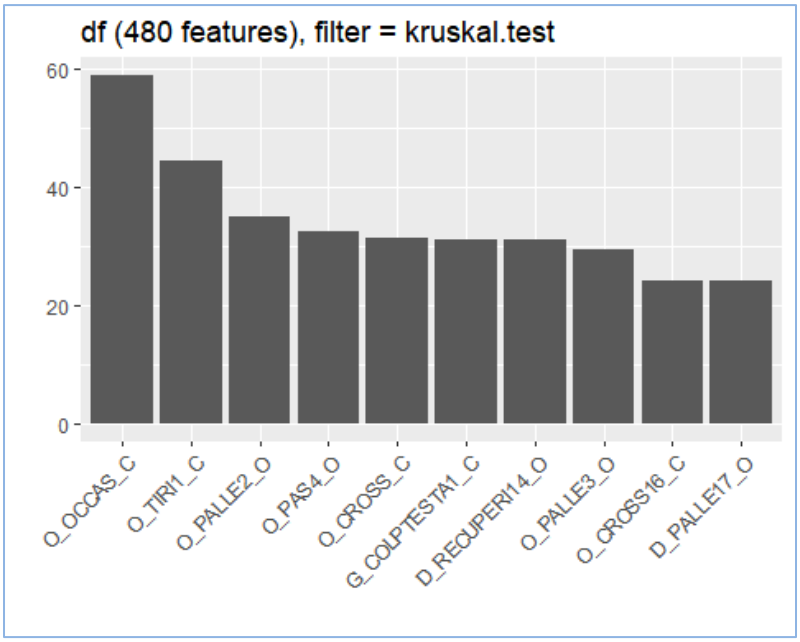
Trions les variables par ordre d'importance pour une meilleure lisibilité. Les 10 les plus pertinentes au sens du critère sont :

```
#tri selon importance décroissante
print(head(filter.kruskal$data[order(filter.kruskal$data['kruskal.test'],decreasing=TRUE),],10))
```

```
##                name    type kruskal.test
## 180       O_OCCAS_C integer     58.98664
## 181       O_TIRI1_C integer     44.36634
## 351      O_PALLE2_O integer     34.91336
## 367        O_PAS4_O integer     32.62282
## 135       O_CROSS_C integer     31.30994
## 27   G_COLPTESTA1_C integer     31.25086
## 332 D_RECUPERI14_O integer     31.16571
## 352      O_PALLE3_O integer     29.40742
## 146     O_CROSS16_C integer     24.30078
## 103     D_PALLE17_O integer     24.19809
```

Un affichage graphique est possible :

```
#affichage graphique
mlr::plotFilterValues(filter.kruskal,n.show=10)
```



Se démarquent :

- O_OCASS_C : nombre d'occasions de but de l'équipe hôte, j'imagine que la relation est positive avec la victoire ;

- O_TIRI1_C : nombre de tirs de l'équipe hôte ;
- O_PALLE2_O : nombre de balles balancées vers l'avant (au-delà de son milieu de terrain) par l'équipe visiteuse ;
- O_PAS4_O : nombre de passes longues des défenseurs vers l'avant des visiteurs ;
- Etc.

Si on connaît un peu le foot, on se rend compte que, même si la prédiction peut être performante par ailleurs, la valeur ajoutée des connaissances produites par le data mining n'est pas toujours au rendez-vous. Je me vois mal dire à Mourinho : "écoute mon gars, la data science a parlé, si tu veux gagner, il faut se procurer des occasions". Je crois qu'il va me rire au nez. En revanche, lui dire : "pour gagner, il faut faire un pressing haut ou encore densifier le milieu de terrain pour obliger les adversaires à balancer le ballon vers l'avant" peut éclairer sa perception des rencontres (je ne suis pas sûr qu'il ait vraiment besoin de moi pour savoir ça, mais ça y est, je suis bon pour faire le consultant TV).

Clairement, il ne s'agit plus seulement de statistique ici. La connaissance du domaine est essentielle pour renforcer la pertinence d'un résultat. On peut débattre longuement là-dessus, ce n'est pas le propos de ce tutoriel qui vise à montrer les fonctionnalités de "mlr". Mais nous devons garder à l'esprit que la question est primordiale (Connaissances des données vs. Connaissances du domaine) dans la pratique de la data science.

D'autres mesures sont disponibles pour filtrer les descripteurs. L'incertitude symétrique fait partie des plus populaires.

```
#tri avec autre critère - vient de FSelectorRcpp, discrétise la variable X au préalable
filter.su <- mlr::generateFilterValuesData(task=foot.task,method="symmetrical.uncertainty")

#tri selon importance décroissante
print(head(filter.su$data[order(filter.su$data['symmetrical.uncertainty'],decreasing=TRUE),],10))
```

```
##                name    type symmetrical.uncertainty
## 180       O_OCCAS_C integer              0.09621463
## 351      O_PALLE2_O integer              0.08346482
## 181       O_TIRI1_C integer              0.06760175
## 135       O_CROSS_C integer              0.06327469
## 332  D_RECUPERI14_O integer              0.06046580
## 27   G_COLPTESTA1_C integer              0.05907517
## 178 O_PERCENT_ATT_C numeric              0.05615347
```

```
## 298       D_.PROT_O numeric              0.05615347
## 146     O_CROSS16_C integer              0.05570177
## 343     D_PALLE16_O integer              0.05157604
```

L'approche provient du package FSelectorRcpp. On se rend compte en lisant la documentation qu'elle nécessite en interne une discrétisation des descripteurs quantitatifs à l'aide de l'algorithme MDLP (Fayyad et Irani, 1993). Nous reproduisons les calculs pour la variable placée en première position O_OCCAS_C.

```
#discretisation
library(discretization)
disc.occase_hote <- discretization::mdlp(cbind(df$O_OCCAS_C,df$resmatch))

#borne(s) de discretisation
print(disc.occase_hote$cutp)

## [[1]]
## [1] 3.5
```

Un découpage en 2 intervalles et effectué avec pour seuil 3.5.

Calculons la distribution des classes dans les intervalles.

```
#croisement
print(prop.table(table(df$resmatch,disc.occase_hote$Disc.data[,1]),margin=2))

##
##                     1         2
##   deception 0.7692308 0.4000000
##   victoire  0.2307692 0.6000000
```

La relation est effectivement positive. La proportion de victoires est de 60% lorsque le nombre d'occasions est supérieur au seuil 3.5, elle est de 23% sinon.

3.3 Schéma "holdout" pour la modélisation prédictive

Plusieurs schémas de rééchantillonnage sont possibles. Dans un premier temps, nous utilisons le schéma apprentissage-test (holdout) pour construire et évaluer un arbre de décision. La procédure **makeResampleInstance**() permet de scinder les observations en 2 parties.

```
#schéma holdout
set.seed(2019)
rs.houldout <- mlr::makeResampleInstance("Holdout",split=0.7,task=foot.task)
print(rs.houldout)
```

```
## Resample instance for 380 cases.
## Resample description: holdout with 0.70 split rate.
## Predict: test
## Stratification: FALSE
```

Nous obtenons les index des 70% réservés à l'apprentissage :

```
#individus en apprentissage
print(rs.houldout$train.inds)

## [[1]]
##   [1] 293 271 115 234  19  17 307   4  39 226 285 244  80  72 255 232  25
##  [18] 366 136 170 147 209 340  62  47 373 210 335 220 160 371 355 282  60
##  [35]  14  30 128 154   5 267 359  88 348 227 272 166 306 310  70  66 101
##  [52]  29  12 114 169 122 323 296  18  16  57  69 377  98 246 333  96  27
##  [69] 263 186 109  42 317  48 184 304  38 157 100 229  13 275 286  75 159
##  [86] 356 156 177 216  94 326 211 233  58 324 318 117 262 106 258 135 222
## [103] 327 365 102 380 357 303  53 346  81 283 287 360 221 363 362 308 208
## [120] 321 309 256 268 376 172 350 369 228  56  50 345  41 291 302  86 297
## [137] 174 107 312  84 178  52 259 162  92 161 214  91 230  54 344   2 295
## [154]  87 276  78 343 191 314 141  32 130 103  95 195 219 217 337 274  89
## [171]  55 378 241 187 218 121  73 270 240 260 150 134  51 108 249  65  74
## [188] 281 225 338  93 353 279 330 201 215 251 113  49  35  36 358 144 189
## [205]  44 148 213 185 379 261 273 223 118 329 183 104 284 367   1  34  24
## [222] 325  71 349 182 165  33 242 264 320  28 245 163 153 336 126   3  61
## [239]  22 328 265 199 110 112 237 288   7 175  79 158 188  11  31 168  46
## [256] 372 167  40  20  43 140 145 105 149 179 347
```

Et ceux dédiés au test.

```
#individus en test
print(rs.houldout$test.inds)

## [[1]]
##   [1]  97 181  64 151 129 290 334 339 132 370 202 331 236 193  63 194   6
##  [18] 146  21 364 180 173  45  99  23 248 294 205 243 289 342 239 119 292
##  [35]  85 203 197   9 280 332 278 155 120 192  77 190 301  59 300  76 152
##  [52] 200 133 354 277 247 361 351  82 250 206 198 171 212 311 235 139  83
##  [69] 138 116  90 257 266 124 207 305   8  68 341 375 254 269 143 125  26
##  [86] 298 142 204 315 374 164 127 253  37 224 111 131 196 319 368  67 322
## [103] 176 352 137 299 238 316 123 252 313 231  15  10
```

Nous construisons un arbre de décision (cl="classif.rpart") en demandant la production des probabilités d'affectation en prédiction (predict.type="prob"). En réalité, l'outil fait appel à l'algorithme rpart() du package éponyme. "mlr" se recharge de réaliser les opérations de préparation idoines avant d'invoquer la procédure sous-jacente. La liste des algorithmes que "mlr" encapsule ainsi est longue (83 méthodes de classement à ce jour, 06 avril 2019).

```
#construire un modèle
arbre.lrn <- mlr::makeLearner(cl="classif.rpart",predict.type="prob")
print(arbre.lrn)
```

```
## Learner classif.rpart from package rpart
## Type: classif
## Name: Decision Tree; Short name: rpart
## Class: classif.rpart
## Properties:
twoclass,multiclass,missings,numerics,factors,ordered,prob,weights,featimp
## Predict-Type: prob
## Hyperparameters: xval=0
```

Nous pouvons maintenant lancer l'apprentissage en lui passant le "learner" (arbre.lrn), la tâche à réaliser (task=foot.task) et les index des données d'apprentissage (subset).

```
#apprentissage et affichage
arbre.modele <-
mlr::train(arbre.lrn,foot.task,subset=rs.houldout$train.inds[[1]])
print(mlr::getLearnerModel(arbre.modele))
```

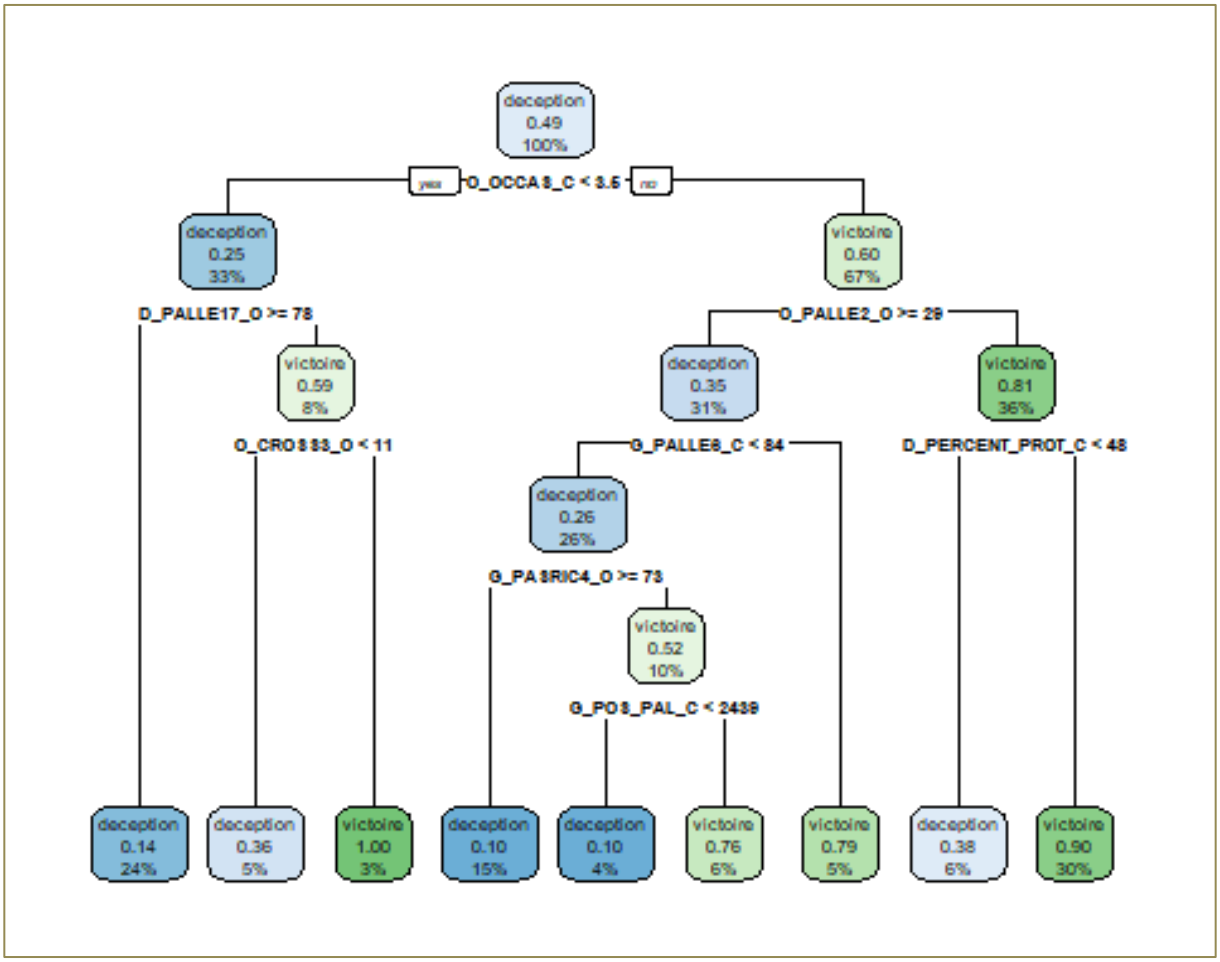
```
## n= 266
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 266 130 deception (0.51127820 0.48872180)
##    2) O_OCCAS_C< 3.5 87  22 deception (0.74712644 0.25287356)
##      4) D_PALLE17_O>=77.5 65   9 deception (0.86153846 0.13846154) *
##      5) D_PALLE17_O< 77.5 22   9 victoire (0.40909091 0.59090909)
##       10) O_CROSS3_O< 10.5 14   5 deception (0.64285714 0.35714286) *
##       11) O_CROSS3_O>=10.5 8   0 victoire (0.00000000 1.00000000) *
##    3) O_OCCAS_C>=3.5 179  71 victoire (0.39664804 0.60335196)
##      6) O_PALLE2_O>=28.5 82  29 deception (0.64634146 0.35365854)
##       12) G_PALLE6_C< 83.5 68  18 deception (0.73529412 0.26470588)
##         24) G_PASRIC4_O>=73 41   4 deception (0.90243902 0.09756098) *
##         25) G_PASRIC4_O< 73 27  13 victoire (0.48148148 0.51851852)
##           50) G_POS_PAL_C< 2438.5 10   1 deception (0.90000000 0.10000000) *
##           51) G_POS_PAL_C>=2438.5 17   4 victoire (0.23529412 0.76470588) *
##       13) G_PALLE6_C>=83.5 14   3 victoire (0.21428571 0.78571429) *
##      7) O_PALLE2_O< 28.5 97  18 victoire (0.18556701 0.81443299)
##       14) D_PERCENT_PROT_C< 47.64 16   6 deception (0.62500000 0.37500000) *
##       15) D_PERCENT_PROT_C>=47.64 81   8 victoire (0.09876543 0.90123457) *
```

On peut faire appel à rpart.plot() pour disposer d'un affichage plus sympathique.

```
#affichage plus sympathique
library(rpart.plot)
```

```
## Loading required package: rpart
```

```
rpart.plot(mlr::getLearnerModel(arbre.modele),roundint=FALSE)
```



On lit par exemple (feuille toute à droite) que l'équipe hôte a 90% de chances d'emporter le match lorsque (O_OCCAS_C >= 3.6) et (O_PALLE2_O < 29) et (D_PERCENT_PROT_C >= 48). Cela est vrai pour 30% des rencontres.

Pour évaluer les performances de l'arbre, nous effectuons une prédiction sur l'échantillon test.

```
#prédiction
arbre.pred <-
predict(arbre.modele,task=foot.task,subset=rs.houldout$test.inds[[1]])
print(arbre.pred)
```

```
## Prediction: 114 observations
## predict.type: prob
## threshold: deception=0.50,victoire=0.50
## time: 0.03
##      id     truth prob.deception prob.victoire  response
## 97   97  victoire     0.64285714    0.35714286 deception
## 181 181 deception     0.64285714    0.35714286 deception
```

```
## 64   64 deception     0.90243902    0.09756098 deception
## 151 151 deception     0.62500000    0.37500000 deception
## 129 129  victoire     0.86153846    0.13846154 deception
## 290 290  victoire     0.09876543    0.90123457  victoire
## ... (#rows: 114, #cols: 5)
```

Comme nous avions demandé les probabilités, la sortie est plus riche que la simple classe prédite. Elle intègre des probabilités d'affectation.

Il nous est dès lors possible de calculer la matrice de confusion (qui comprend les marges).

```
#matrice de confusion
arbre.mc <- mlr::calculateConfusionMatrix(arbre.pred)
print(arbre.mc)
```

```
##            predicted
## true        deception victoire -err.-
##   deception        46       18     18
##   victoire         21       29     21
##   -err.-           21       18     39
```

Et faire calculer les indicateurs de performances dont la liste des possibles est longue pour la tâche qui nous concerne :

```
#mesures de performances possibles
print(mlr::listMeasures(foot.task))
```

```
##  [1] "tnr"              "tpr"              "featperc"
##  [4] "f1"               "mmce"             "mcc"
##  [7] "brier.scaled"     "lsr"              "bac"
## [10] "fn"               "fp"               "fnr"
## [13] "qsr"              "fpr"              "npv"
## [16] "brier"            "auc"              "timeboth"
## [19] "multiclass.aunp"  "timetrain"        "multiclass.aunu"
## [22] "ber"              "timepredict"      "multiclass.brier"
## [25] "ssr"              "ppv"              "acc"
## [28] "logloss"          "wkappa"           "tn"
## [31] "tp"               "multiclass.au1p"  "multiclass.au1u"
## [34] "fdr"              "kappa"            "gpr"
## [37] "gmean"
```

Nous optons pour le logloss et le taux d'erreur (mmce).

```
#mesurer en test
mlr::performance(arbre.pred,measures=list(logloss,mmce))
```

```
##   logloss      mmce
## 1.0024663 0.3421053
```

3.4 Validation croisée

Un schéma "holdout" avec 380 observations n'est pas des plus judicieux. Il vaut mieux passer par un schéma de rééchantillonnage permettant d'utiliser la totalité des données pour l'apprentissage. La validation croisée convient parfaitement dans ce cas. Nous demandons une validation répétée (method = "RepCV"), avec (reps=5) répétitions en (folds=5) blocs.

```
#passer par la validation croisée répétée - définition
cv.desc = mlr::makeResampleDesc(method="RepCV",reps=5,folds=5)
print(cv.desc)

## Resample description: repeated cross-validation with 25 iterations: 5 folds and 5 reps.
## Predict: test
## Stratification: FALSE
```

La répétition permet d'obtenir des résultats plus stables. Nous avons maintenant :

```
#lancer les calculs
cv.rs <-
mlr::resample(learner=arbre.lrn,task=foot.task,resampling=cv.desc,measures=li
st(logloss,mmce))
print(cv.rs)

## Resample Result
## Task: df
## Learner: classif.rpart
## Aggr perf: logloss.test.mean=1.1961762,mmce.test.mean=0.3447368
## Runtime: 2.91936
```

3.5 Comparaison (benchmarking) de méthodes

L'identification de la méthode la plus efficace est une mission récurrente en machine learning. Nous avions choisi initialement un arbre de décision, mais il se peut très bien qu'un autre algorithme soit plus performant. La seule manière de le savoir est de les évaluer sur notre jeu de données. Nous restons sur le schéma de la validation croisée répétée.

Les méthodes à expérimenter sont, en sus de l'arbre : la régression logistique, le gradient boosting, les plus proches voisins, les svm.

Nous les instancions et nous les regroupons dans une liste. Attention, pour tous ces algorithmes, les paramètres par défaut de modélisation sont automatiquement présélectionnés. Si nous souhaitons les connaître, il nous importe de consulter la

documentation. Je me suis rendu compte par exemple que la taille de voisinage est égale à (k=7) pour l'algorithme des plus proches voisins.

```
#régression logistique
lr.lrn <- mlr::makeLearner("classif.LiblineaRL1LogReg",predict.type="prob")

#gradient boosting
gbm.lrn <- mlr::makeLearner("classif.gbm",predict.type="prob")

#plus proches voisins
knn.lrn <- mlr::makeLearner("classif.kknn",predict.type="prob")

#svm
svm.lrn <- mlr::makeLearner("classif.svm",predict.type="prob")

#liste de learners
lrns <- list(arbre.lrn,lr.lrn,gbm.lrn,knn.lrn,svm.lrn)
```

Devant la masse de calculs que cette comparaison va engendrer (5 répétitions de 5-CV pour 5 méthodes), je me propose de passer par la parallélisation des traitements. "mlr" semble savoir le prendre en compte. Mais à sa manière en réalité, nous pouvons demander la possibilité de parallélisation, mais le package seul décide s'il peut réellement la mettre en œuvre. Nous demandons quoiqu'il en soit que 4 cœurs soient utilisés lorsque cela est possible.

```
#paralléliser
library(parallelMap)

#démarrer la parallélisation
parallelStartSocket(cpus=4)

#benchmarking
bmr <- mlr::benchmark(lrns,task=foot.task,resamplings =
cv.desc,measures=list(logloss,mmce))

#stopper la parallélisation
parallelStop()
```

Nous affichons les résultats du benchmarking.

```
#résultats
print(bmr)

##   task.id                learner.id logloss.test.mean mmce.test.mean
## 1      df             classif.rpart         1.3835414      0.3621053
```

```
## 2      df classif.LiblineaRL1LogReg         1.0360783      0.2694737
## 3      df               classif.gbm         0.5294758      0.2673684
## 4      df              classif.kknn         0.9234022      0.3784211
## 5      df               classif.svm         0.5369415      0.2678947
```

2 méthodes se démarquent : le gradient boosting et le SVM (qui s'appuie sur un noyau RBF dixit la doc).

3.6 Optimisation (tuning) d'une méthode

Nous optons pour le gradient boosting (GBM, *gradient boosting machine*) pour la suite. Une autre question clé est la détermination des meilleures valeurs des paramètres (qui pullulent dans les algorithmes de machine learning) pour la modélisation. Pour notre méthode, nous décidons de jouer sur le nombre d'arbres (n.trees) et leur profondeur (interaction.depth). Nous procédons par recherche systématique (recherche en grille, control = makeTuneControlGrid).

De nouveau, nous demandons la parallélisation des calculs, qui sera mieux maîtrisée ici, je l'ai constaté en observant le gestionnaire de tâches de Windows.

```
#optimisation du GBM, avec le paramètre de coût et le noyau
paramset.gbm =
makeParamSet(makeDiscreteParam("n.trees",values=c(10,20,50,100,200)),makeDisc
reteParam("interaction.depth",values=c(1,2,3)))

#controle de l'espace de recherche
ctrl <- mlr::makeTuneControlGrid()

#démarrer la parallélisation
parallelStartSocket(cpus=4)

#processus de recherche
opt.gbm <- mlr::tuneParams(gbm.lrn,task=foot.task,resampling=cv.desc,par.set
= paramset.gbm,measures=list(logloss),control=ctrl)

#stopper la parallélisation
parallelStop()
```

En validation croisée, voici les résultats de l'exploration des paramètres :

```
#affichage des résultats
print(opt.gbm)
```

```
## Tune result:
## Op. pars: n.trees=100; interaction.depth=1
## logloss.test.mean=0.5169509
```

Nous pouvons afficher le détail des résultats.

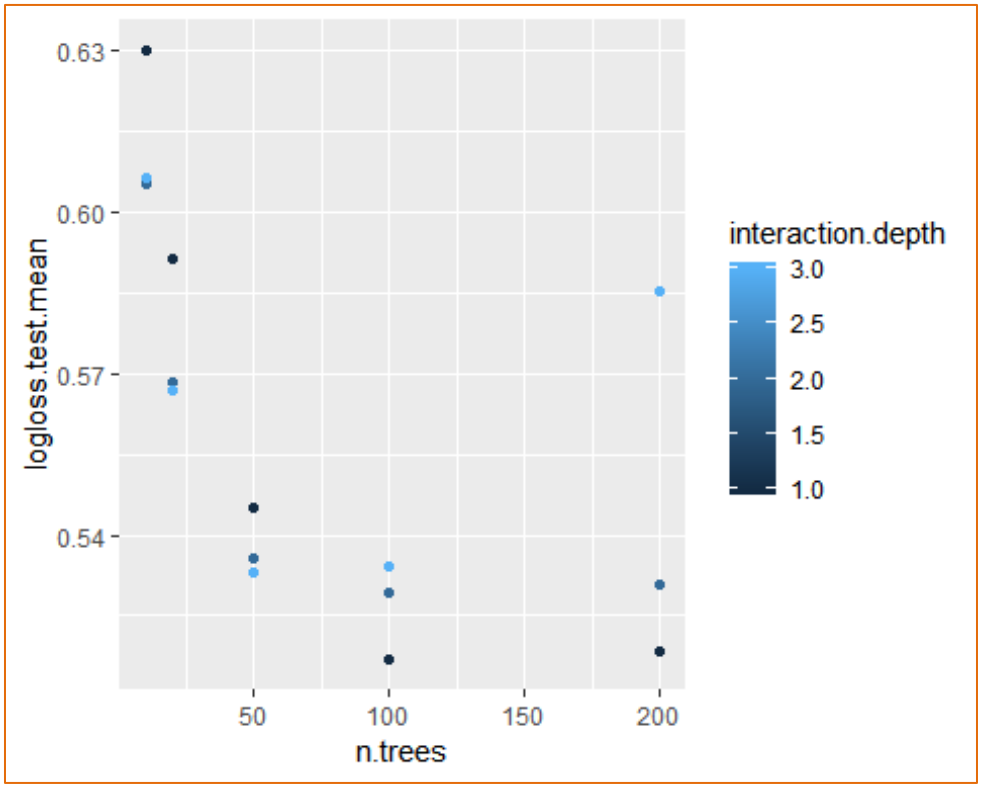
```
#affichage du détail des résultats
res.opt <- mlr::generateHyperParsEffectData(opt.gbm)
print(res.opt$data)

##    n.trees interaction.depth logloss.test.mean iteration exec.time
## 1       10                 1         0.6300944         1      8.07
## 2       20                 1         0.5914529         2      8.51
## 3       50                 1         0.5450080         3      9.73
## 4      100                 1         0.5169509         4     11.46
## 5      200                 1         0.5185479         5     15.62
## 6       10                 2         0.6051030         6      8.28
## 7       20                 2         0.5682736         7      9.38
## 8       50                 2         0.5355718         8     11.42
## 9      100                 2         0.5292088         9     16.06
## 10     200                 2         0.5307514        10     23.48
## 11      10                 3         0.6063893        11      8.60
## 12      20                 3         0.5668082        12      9.86
## 13      50                 3         0.5330886        13     13.27
## 14     100                 3         0.5340189        14     19.56
## 15     200                 3         0.5853201        15     31.07
```

Les paramètres optimaux sont accessibles via un champ de l'objet résultat :

```
#accès aux paramètres optimaux
print(opt.gbm$x)

## $n.trees
## [1] 100
##
## $interaction.depth
## [1] 1
```

Un graphique de synthèse permet de comprendre que le nombre d'arbres est le facteur déterminant dans le processus, et qu'il ne fallait surtout pas instancier des arbres trop profonds.

```
#affichage graphique - selon le noyau
plotHyperParsEffect(res.opt,x='n.trees',y='logloss.test.mean',z='interaction.depth')
```



3.7 Identification du rôle des variables

Le filtrage ci-dessus (section 3.2) a permis de souligner les variables les plus liées avec le résultat des rencontres. Mais c'était en dehors de toute considération de l'algorithme de machine learning. Voyons ce qu'il en est si l'on veut analyser le cas particulier du gradient boosting. Quelles sont les variables déterminantes pour le classement ?

Nous instancions une nouvelle version du gradient boosting avec les paramètres optimaux identifiés précédemment (**par.vals=opt.gbm$x**).

```
#GBM avec ces paramètres optimaux
optgbm.lrn <- mlr::makeLearner("classif.gbm",par.vals=opt.gbm$x,predict.type = "prob")

#apprentissage sur les données
optgbm.modele <- mlr::train(optgbm.lrn,task=foot.task)

print(optgbm.modele)

## Model for learner.id=classif.gbm; learner.class=classif.gbm
## Trained on: task.id = df; obs = 380; features = 480
## Hyperparameters: keep.data=FALSE,n.trees=100,interaction.depth=1
```

GBM (*gradient boosting machine*) sait fournir une importance des variables, liée à leur apparition et la qualité de segmentation induite dans l'ensemble des arbres générés.

```
#importance des variable
varImp.gbm <- mlr::getFeatureImportance(optgbm.modele)

#affichage trié
print(sort(varImp.gbm$res,decreasing=TRUE)[1:15])

##   O_OCCAS_C O_PALLE2_O O_OCCAS_O O_TIRI1_C O_CROSS_C D_PERCENT_PROT_C
## 1  41.61195   24.53586   23.8185  14.57986  13.25935         8.011143
##   G_PASALTI1_O D_RECUPERI14_O G_COLPTESTA1_C O_PAS4_O G_VEL_GIOC_O
## 1     7.730171       6.265094       5.942304  5.52348     5.198719
##   D_RECUPERI14_C D_RECUPERI11_O G_VEL_GIOC_C O_CROSS_O
## 1       5.029072       4.880958      4.03171  4.027991
```

La variable plus importante est O_OCASS_C (occasions des équipes hôtes), suivie de O_OCCAS_O (occasions des visiteurs). Ce n'est pas vraiment une réelle surprise.

Voyons maintenant de quelle manière ces variables influencent la prise de décision. Pour cela, nous utilisons le graphique partial dependance plot. Il montre, pour une série de valeurs de la variable étudiée, les différentes probabilités d'affectation à la modalité cible (probabilité de victoire pour nous) fournie par le modèle. Pour un classifieur linéaire, nous aurions une droite, croissante si la relation est positive, décroissante si elle est négative, horizontale si la variable n'a aucun impact en prédiction (son coefficient est égal à 0). Pour un modèle non-linéaire, nous devrions avoir courbe.

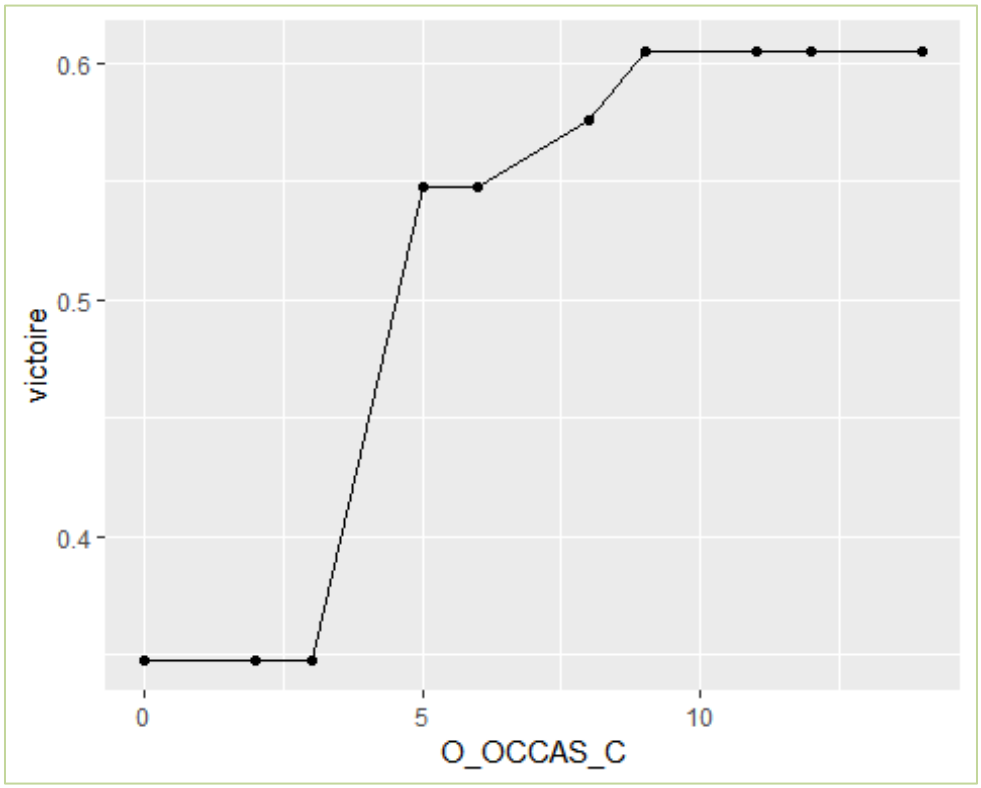
Nous procédons en 2 temps pour O_OCCAS_C. Nous générons les données…

```
#partial dependance plot de la variable la plus impactante : O_OCCAS_C
pd.o_occas_c <-
mlr::generatePartialDependenceData(optgbm.modele,foot.task,"O_OCCAS_C")

## Loading required package: mmpf

print(pd.o_occas_c)

## PartialDependenceData
## Task: df
## Features: O_OCCAS_C
## Target: O_OCCAS_C
## Derivative: FALSE
## Interaction: FALSE
## Individual: FALSE
##     victoire O_OCCAS_C
## 1: 0.3479128         0
```

```
## 2: 0.3479128         2
## 3: 0.3479128         3
## 4: 0.5475925         5
## 5: 0.5475925         6
## 6: 0.5762758         8
## ... (#rows: 10, #cols: 2)
```

… à partir desquelles nous construisons le graphique :

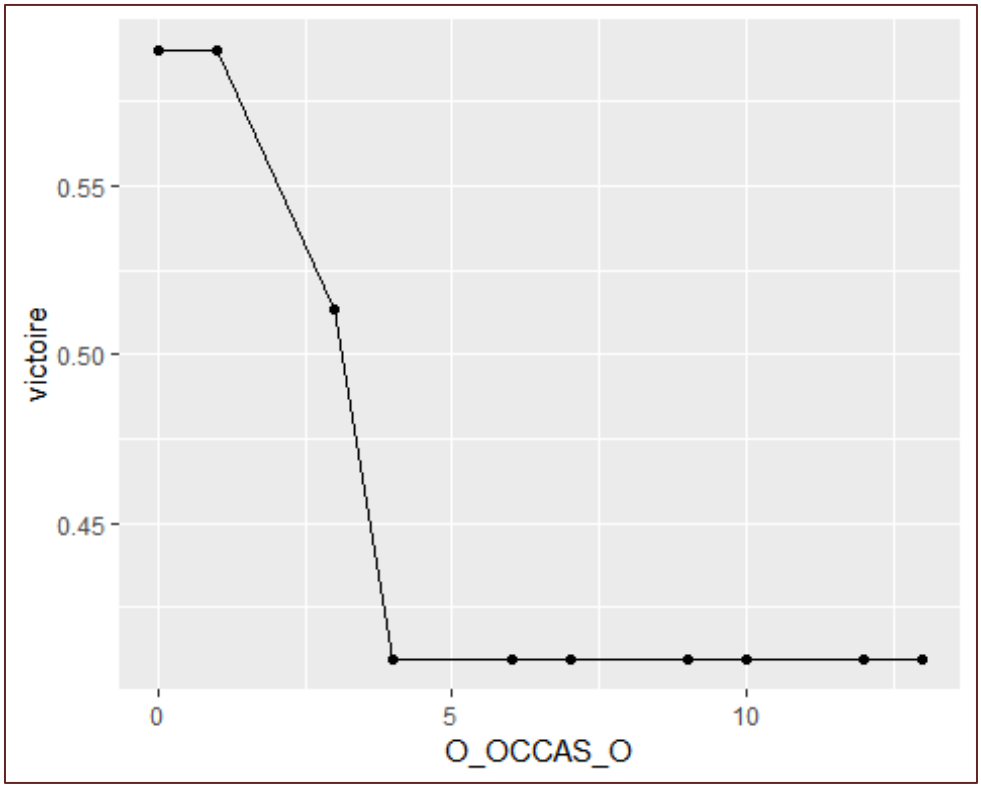
```
#graphique
mlr::plotPartialDependence(pd.o_occas_c)
```



La relation est positive : plus l'équipe hôte a des occasions, plus elle a de chances de gagner. Mais à partir d'un certain stade, la multiplication des occasions n'est pas synonyme de meilleures chances de victoires.

Pour la seconde variable, O_OCCAS_O (occasion des visiteurs)…

```
#partial dependance plot de la variable la plus impactante : O_OCCAS_O
pd.o_occas_o <-
mlr::generatePartialDependenceData(optgbm.modele,foot.task,"O_OCCAS_O")

#graphique
mlr::plotPartialDependence(pd.o_occas_o)
```



… la relation est négative, forcément. Plus les visiteurs ont des occasions de but, moins l'hôte a des chances de gagner, ici également avec un effet seuil.

Bref, pour gagner des matchs à domicile, il faut se procurer des occasions de but et en laisser peu (d'occasions) aux adversaires. Tout ça enfonce un peu des portes ouvertes, on est d'accord. Mais bon, c'est déjà pas mal quand les données confirment les évidences.

3.8 Courbe ROC en validation croisée

Dernier thème de notre tutoriel, nous souhaitons construire la courbe ROC, en validation croisée puisque le schéma holdout n'est pas très raisonnable au regard de la taille de notre base de données. Nous appliquons notre algorithme du gradient boosting sur une validation croisée en (iter = 10) folds.

```
#courbe ROC en validation croisée
#validation croisée
cvdesc.gbm = mlr::makeResampleDesc(method="CV",iter=10)
print(cvdesc.gbm)
```

```
## Resample description: cross-validation with 10 iterations.
## Predict: test
## Stratification: FALSE
```

```
#lancement
cvres.gbm <- mlr::resample(learner=optgbm.lrn,task=foot.task,resampling=cvdesc.gbm,measures=list(logloss,mmce))

print(cvres.gbm)
```

```
## Resample Result
## Task: df
## Learner: classif.gbm
## Aggr perf: logloss.test.mean=0.5134109,mmce.test.mean=0.2394737
## Runtime: 3.96896
```

En accédant au champs **$pred** de l'objet…

```
#résultats détaillés
cvres.gbm$pred
```

```
## Resampled Prediction for:
## Resample description: cross-validation with 10 iterations.
## Predict: test
## Stratification: FALSE
## predict.type: prob
## threshold: deception=0.50,victoire=0.50
## time (mean): 0.03
##   id     truth prob.deception prob.victoire  response iter  set
## 1  4 deception      0.7396876     0.2603124 deception    1 test
## 2 19  victoire      0.7481418     0.2518582 deception    1 test
## 3 32  victoire      0.5807838     0.4192162 deception    1 test
## 4 48 deception      0.5110344     0.4889656 deception    1 test
## 5 60  victoire      0.4801251     0.5198749  victoire    1 test
## 6 80  victoire      0.1521858     0.8478142  victoire    1 test
## ... (#rows: 380, #cols: 7)
```

… nous constatons que l'outil met à notre disposition les prédictions et les probabilités d'affectation en test pour les 10 folds (iter).

Nous exploitons ces informations pour construire les colonnes de FPR (taux de faux positifs, abscisse de la courbe ROC) et de TPR (taux de vrais positifs, ordonnée)….

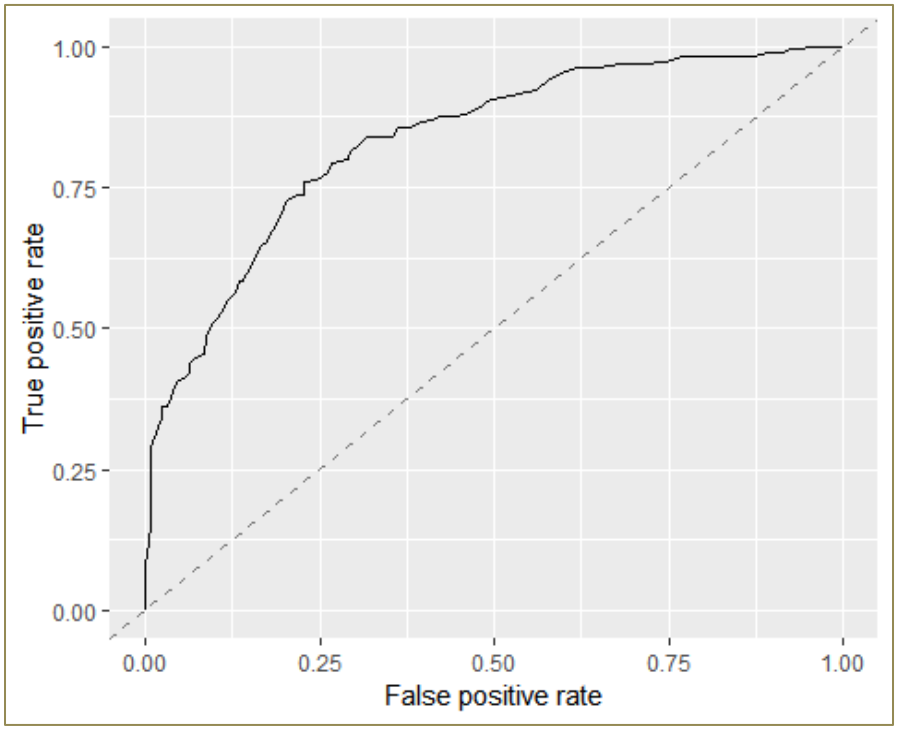
```
#données pour courbe ROC
dataROC <-
mlr::generateThreshVsPerfData(cvres.gbm$pred,measures=list(fpr,tpr))
print(dataROC)
```

```
## $measures
## $measures$fpr
## Name: False positive rate
## Performance measure: fpr
## Properties: classif,req.pred,req.truth
```

```
## Minimize: TRUE
## Best: 0; Worst: 1
## Aggregated by: test.mean
## Arguments:
## Note: Percentage of misclassified observations in the positive class. Also called
false alarm rate or fall-out.
##
## $measures$tpr
## Name: True positive rate
## Performance measure: tpr
## Properties: classif,req.pred,req.truth
## Minimize: FALSE
## Best: 1; Worst: 0
## Aggregated by: test.mean
## Arguments:
## Note: Percentage of correctly classified observations in the positive class. Also
called hit rate or recall or sensitivity.
##
##
## $data
##             fpr        tpr  threshold
## 1   1.000000000 1.00000000 0.00000000
## 2   1.000000000 1.00000000 0.01010101
## 3   1.000000000 1.00000000 0.02020202
## 4   1.000000000 1.00000000 0.03030303
## 5   0.995000000 1.00000000 0.04040404
...

## 92  0.000000000 0.03093137 0.91919192
## 93  0.000000000 0.02504902 0.92929293
## 94  0.000000000 0.01838235 0.93939394
## 95  0.000000000 0.01838235 0.94949495
## 96  0.000000000 0.00625000 0.95959596
## 97  0.000000000 0.00000000 0.96969697
## 98  0.000000000 0.00000000 0.97979798
## 99  0.000000000 0.00000000 0.98989899
## 100 0.000000000 0.00000000 1.00000000
##
## $aggregate
## [1] TRUE
##
## attr(,"class")
## [1] "ThreshVsPerfData"
```

... que nous injectons dans le graphique.

```
#affichage
mlr::plotROCCurves(dataROC)
```



La prédiction (en validation croisée) peut servir également au calcul de l'AUC (aire sous la courbe).

```
#valeur de l'AUC
print(mlr::performance(cvres.gbm$pred,mlr::auc))

##       auc
## 0.8397004
```

4 Conclusion

Tout comme "caret", "mlr" n'invente rien. Il encapsule tout simplement (façon de parler, il y a un énorme travail de programmation et de suivi des versions derrière) dans un ensemble cohérent les principales tâches du data miner susceptibles de faire appel à une multitude de packages. En cela, il nous facilite la vie et canalise notre démarche de traitements.

5 Références

M. Carpita, M. Sandri, A. Simonetto, P. Zuccolotto, "Football Mining with R", in "Data Mining Applications with R", Zhao et Cen editeurs, Chapitre 14, pages 397–433, 2014.

B. Bischl, M. Lang, L. Kotthoff, J. Schiffner, J. Richter, E. Studerus, G. Casalicchio, Z.M. Jones, "mlr: Machine Learning in R", in Journal of Machine Learning Research, 17(170), 1–5, 2016.