Machine Learning avec le package « caret »

Tutoriel Tanagra

5 avril 2018

1 Introduction

La profusion des packages est à la fois une force et une faiblesse de R. Une force parce que cette richesse permet de couvrir une très large fraction de la pratique des statistiques et du machine learning. Aujourd'hui, face à tout type de problème, la première question que l'on se pose est : "est-ce qu'il n'y a pas déjà un package qui permet de le faire simplement ?". Mais c'est aussi une faiblesse parce qu'il y a une très forte hétérogénéité des pratiques et des modes opératoires des packages. Et la documentation n'est pas toujours explicite malheureusement. Il m'arrive d'aller voir dans le code même pour comprendre réellement ce qui est implémenté. L'affaire se corse d'autant plus que nous devons souvent combiner (jongler entre) plusieurs packages pour mettre en place une analyse complète.

Le package "**caret**" (**C**lassification **A**nd **RE**gression **T**raining) est une librairie pour R. Il couvre une large fraction de la pratique de l'analyse prédictive (classement et régression). Un peu à la manière de "scikit-learn" (http://scikit-learn.org/stable/) pour Python, il intègre dans un ensemble cohérent les étapes clés de la modélisation : préparation des données, sélection, apprentissage, évaluation. La standardisation des prototypes des fonctions d'apprentissage et de prédiction notamment permet de simplifier notre code, facilitant les tâches d'optimisation et de comparaison des modèles.

Dans ce tutoriel, à partir d'un exemple d'identification de "spams", nous montrons quelques facettes du package "caret".

2 Données

Nous traitons les données “Spam” (https://archive.ics.uci.edu/ml/datasets/spambase). Il s’agit d’identifier les messages électroniques frauduleux à partir de leurs caractéristiques (occurrence des termes, longueur des documents, proportion des lettres en majuscules, etc.). Nous disposons de **4601 observations** et **55 variables prédictives**, toutes quantitatives. **La variable cible “spam” est binaire {yes, no}**.

Nous chargeons le fichier “spam_caret.txt”. Nous affichons ses dimensions.

```
#chargement des données
spam <- read.table("spam_caret.txt",header=T,sep="\t",dec=".")


#dimensions
print(dim(spam))
```

```
## [1] 4601   56
```

Nous calculons la proportion des classes :

```
#fréquences relatives des classes
print(prop.table(table(spam$spam)))
```

```
##
##        no       yes
## 0.6059552 0.3940448
```

39.4% des messages sont frauduleux (spam = yes), 60.6% ne le sont pas (spam = no).

Dans le pire des cas, si nous classons tous les messages en “spam = no” (null model ou default classifier), notre taux de succès (ou taux de bon classement) sera de 60.6%. On devrait faire mieux avec les différentes méthodes que nous étudierons.

3 Apprentissage et test

3.1 Subdivision apprentissage-test

Même si “caret” propose des techniques de rééchantillonnage pour l’évaluation des modèles, nous allons subdiviser les données en échantillons d’apprentissage (70%) et de test (30%). Nous utilisons la commande createDataPartition() de la librairie “caret” que nous chargeons au préalable (voire installer si le package n’est pas présent sur votre machine). Elle produit un index des individus à inclure dans l’ensemble d’apprentissage (http://topepo.github.io/caret/data-splitting.html).

```
#chargement de la librairie caret
library(caret)

## Loading required package: lattice

## Loading required package: ggplot2

#index des individus en apprentissage
#set.seed pour rendre reproductible les résultats
set.seed(100)
trainIndex <- createDataPartition(spam$spam,p=0.7,list=F)
print(length(trainIndex))

## [1] 3222
```

Nous effectuons un échantillonnage stratifié sur la variable cible (spam$spam). Les proportions des classes devraient être respectées dans les sous-échantillons.

Nous requérons 0.7 (70%) des individus pour l’apprentissage. Le vecteur d’index comprend 0.7 x 4601 = 3222 valeurs.

En affichant les 10 premières valeurs du vecteur trainIndex, nous constatons que l’échantillon d’apprentissage sera constitué des individus n°2, 4, 5, 6, 8, etc. de la base initiale. A contrario, dans l’échantillon test, nous aurons les indices complémentaires, soit les individus n°1, 3, 7, 13, etc.

```
#10 premiers individus de l'échantillon d'apprentissage
print(head(trainIndex,10))
```

```
##       Resample1
##  [1,]         2
##  [2,]         4
##  [3,]         5
##  [4,]         6
##  [5,]         8
##  [6,]         9
##  [7,]        10
##  [8,]        11
##  [9,]        12
## [10,]        14
```

Nous utilisons ce vecteur d'indice pour partitionner le data frame.

Pour l'ensemble d'apprentissage :

```
#data frame pour les individus en apprentissage
spamTrain <- spam[trainIndex,]
print(dim(spamTrain))
```

```
## [1] 3222   56
```

Et pour l'échantillon test, nous passons par l'indiçage négatif qui indique les individus à exclure.

```
#data frame pour les individus en test
spamTest <- spam[-trainIndex,]
print(dim(spamTest))
```

```
## [1] 1379   56
```

Nous disposons de (4601 - 3222) = 1379 observations.

Dans les deux cas, nous vérifions la distribution des classes.

```
#fréquences absolues des classes - éch. d'apprentissage
print(table(spamTrain$spam))
```

```
##
##   no  yes
## 1952 1270
```

```
#fréquences relatives des classes dans l'éch. d'apprentissage
print(prop.table(table(spamTrain$spam)))
```

```
##
##        no       yes
## 0.6058349 0.3941651
```

```
#distribution des classes dans l'éch. test
print(prop.table(table(spamTest$spam)))
```

```
##
##        no       yes
## 0.6062364 0.3937636
```

Les fréquences relatives sont conformes avec la distribution initiale. L'échantillonnage stratifié a répondu aux attentes.

3.2 Modélisation

Nous souhaitons mener une étude à l'aide de la régression logistique.

La commande trainControl() permet de fixer les paramètres du processus d'apprentissage, et non pas de la méthode d'apprentissage. La nuance est d'importance. Dans notre cas, nous demandons un apprentissage simple. Plus bas, nous verrons qu'il est possible de réaliser une modélisation associée à une procédure d'évaluation par technique de rééchantillonnage (validation croisée, bootstrap).

La commande train() encapsule l'appel de la méthode d'apprentissage, spécifiée par l'option "method". La liste des méthodes utilisables est impressionnante, elle intègre des techniques et packages très récents (http://topepo.github.io/caret/train-models-by-tag.html). Dans certains cas, il est nécessaire d'installer au préalable les packages concernés (ex. installer 'adabag' si l'on souhaite utiliser le bagging).

Si la méthode est paramétrée. L'outil propose un jeu de valeurs par défaut. Il est également possible de les spécifier explicitement. Nous verrons cela plus loin.

```
#paramètre du processu d'apprentissage
fitControl <- trainControl(method="none")


#apprentissage - régression logistique
m_lr <- train(spam ~ ., data = spamTrain,method="glm",trControl=fitControl)
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
print(m_lr)
```

```
## Generalized Linear Model
##
## 3222 samples
##   55 predictor
##    2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: None
```

Un message d'avertissement indique que certaines probabilités sont saturées. Ce n'est pas une erreur en soi. Nous pouvons passer outre.

Le modèle sous-jacent est accessible via la propriété $finalModel

```
#modèle sous-jacent issu de train
#coefficients de la régression logistique
print(m_lr$finalModel)
```

```
##
## Call:  NULL
##
## Coefficients:
##                (Intercept)                   wf_make
##                  -2.157e+00                 -3.886e-01
##                 wf_address                    wf_all
##                  -2.029e-01                  1.374e-01
##                       wf_3d                    wf_our
##                   2.171e+00                  6.184e-01
##                     wf_over                  wf_remove
##                   3.563e-01                  2.354e+00
##                 wf_internet                   wf_order
##                   5.055e-01                  9.209e-01
##                     wf_mail                 wf_receive
##                   3.477e-01                  2.281e-01
##                     wf_will                  wf_people
##                  -1.749e-01                  3.059e-03
```

```
##                wf_report              wf_addresses
##                1.127e-01                 6.835e-01
##                  wf_free               wf_business
##                2.988e-01                 1.395e+00
##                 wf_email                    wf_you
##                2.555e-01                 1.040e-01
##                wf_credit                   wf_your
##                7.428e-01                 2.187e-01
##                  wf_font                    wf_000
##                1.664e-01                 2.565e+00
##                 wf_money                     wf_hp
##                6.739e-01                -2.552e+00
##                   wf_hpl                    wf_lab
##               -7.121e-01                -2.433e+00
##                  wf_labs                  wf_telnet
##               -2.017e-01                -2.729e+00
##                   wf_857                   wf_data
##               -1.137e+00                -7.287e-01
##                   wf_415                     wf_85
##               -4.734e-01                -3.184e+00
##            wf_technology                   wf_1999
##                8.472e-01                 4.025e-03
##                 wf_parts                    wf_pm
##               -1.941e-01                -6.515e-01
##                wf_direct                    wf_cs
##               -1.174e+00                -4.142e+01
##               wf_meeting              wf_original
##               -1.996e+00                -7.991e-01
##               wf_project                    wf_re
##               -1.106e+00                -6.914e-01
##                   wf_edu                 wf_table
##               -1.110e+00                -2.157e+00
##            wf_conference              cf_pvirgule
##               -2.942e+00                -9.967e-01
##            cf_parenthese               cf_crochet
##                1.386e-01                -1.623e+00
##                cf_pexlam                 cf_dollar
```

```
##                   7.383e-01                    5.730e+00
##                    cf_diese  capital_run_length_average
##                   1.655e+00                    6.879e-02
## capital_run_length_longest    capital_run_length_total
##                   1.214e-02                    7.842e-04
##
## Degrees of Freedom: 3221 Total (i.e. Null);  3166 Residual
## Null Deviance:       4321
## Residual Deviance: 1411  AIC: 1523
```

L'AIC du modèle est **AIC = 1523**. Nous en reparlerons plus loin lorsqu'il faudra procéder à la sélection de variables.

3.3 Prédiction sur l'échantillon test

Pour évaluer la qualité de notre modèle, nous l'appliquons sur l'échantillon test. La commande predict() encapsule l'appel à la fonction associée à l'objet. Le rôle de "caret" est essentiel ici. Habituellement, nous devions post-traiter la sortie de predict.glm() pour obtenir les classes prédites.

```
#prediction
pred <- predict(m_lr,newdata=spamTest)


#distribution des classes prédites
print(table(pred))
```

```
## pred
##  no yes
## 871 508
```

Il y a 508 prédictions positives c.-à-d. 508 messages de l'échantillon test sont désignées frauduleuses. A tort ou pas, nous le saurons tout de suite.

3.4 Matrice de confusion et indicateurs d'évaluation

La matrice de confusion matérialise la confrontation entre les classes observées et prédites. Des indicateurs (métriques) interprétables en sont déduits (http://topepo.github.io/caret/measuring-performance.html#measures-for-predicted-classes).

Nous utilisons la commande confusionMatrix(). Un troisième paramètre permet de désigner la modalité cible, nécessaire pour le calcul de certains indicateurs. Dans notre cas, nous cherchons avant tout à identifier les messages frauduleux (spam = yes).

```
#matrice de confusion
mat <- confusionMatrix(data=pred,reference=spamTest$spam,positive="yes")
print(mat)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  no yes
##        no  799  72
##        yes  37 471
##
##                Accuracy : 0.921
##                  95% CI : (0.9054, 0.9347)
##     No Information Rate : 0.6062
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8325
##  Mcnemar's Test P-Value : 0.001128
##
##             Sensitivity : 0.8674
##             Specificity : 0.9557
##          Pos Pred Value : 0.9272
##          Neg Pred Value : 0.9173
##              Prevalence : 0.3938
##          Detection Rate : 0.3416
##    Detection Prevalence : 0.3684
##       Balanced Accuracy : 0.9116
##
##        'Positive' Class : yes
##
```

La matrice est transposée par rapport la présentation habituelle. Nous avons les classes prédites en ligne, les observées en colonne. Le **taux de succès** (*accuracy*, à ne pas confondre avec la

précision) est **92.1%**. L'intervalle de confiance à 95% est fournie. C'est assez rare pour être signalé. En effet, l'ensemble de test n'est qu'un échantillon représentatif (au mieux) de la population. Les taux mesurés sont assortis d'une certaine incertitude.

Nous disposons d'autres indicateurs, en particulier la **sensibilité** qui, associée à la classe positive "spam = yes", est égale à 471/(471+72) = **86.74%**

L'objet "matrice de confusion" possède une série de propriétés. Pour accéder aux indicateurs globaux, nous utilisons $overall qui est un vecteur aux valeurs nommées.

```
#accès aux indicateurs globaux
print(mat$overall)
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##   9.209572e-01   8.325489e-01   9.054394e-01   9.346511e-01   6.062364e-01
## AccuracyPValue  McnemarPValue
##  1.111938e-156   1.127515e-03
```

Pour accéder à la cellule "Accuracy", nous ferons :

```
#accuracy
print(mat$overall["Accuracy"])
```

```
##  Accuracy
## 0.9209572
```

L'accès aux indicateurs par classe passe par le champ $byclass

```
#par classe
print(mat$byClass)
```

```
##          Sensitivity          Specificity       Pos Pred Value
##            0.8674033            0.9557416            0.9271654
##       Neg Pred Value            Precision               Recall
##            0.9173364            0.9271654            0.8674033
##                   F1           Prevalence       Detection Rate
##            0.8962892            0.3937636            0.3415518
## Detection Prevalence    Balanced Accuracy
##            0.3683829            0.9115725
```

La **précision** en l'occurrence est égale à 471 / (471 + 37) = **92.71%**

4 Autres approches de l'évaluation des modèles

Dans cette section, nous essayons de dépasser le schéma ultra-classique "apprentissage - test - matrice de confusion" précédemment étudié.

4.1 Courbe LIFT (courbe de gain)

La courbe LIFT ou courbe de gain est utilisée pour mesurer l'efficacité d'un ciblage (scoring) (http://topepo.github.io/caret/measuring-performance.html#lift-curves). Nous travaillons toujours sur l'échantillon test. Pour la construire, en sus des classes observées, nous avons besoin de la probabilité (score) d'être de la classe positive fournie par le modèle [P(spam = yes / description)].

La commande predict() avec un paramétrage différent peut le faire.

```
#score des individus positifs
score <- predict(m_lr,spamTest,type="prob")[,"yes"]
print(quantile(score))
```

```
##           0%          25%          50%          75%         100% 
## 2.220446e-16 6.806093e-03 1.575575e-01 8.935655e-01 1.000000e+00 
```

Avec l'option "type = prob", predict() produit pour chaque individu les probabilités d'appartenance aux classes. Nous avons une matrice pour l'ensemble des individus de l'échantillon test. Ciblant les ''spam = yes'', nous récupérons la colonne "yes" dans un vecteur nommé "score". Nous affichons ses quartiles. Le score est une probabilité dans notre étude, il varie entre 0 et 1.

Nous créons un data frame regroupant les classes observées et les scores.

```
#tableau de données pour le scoring
liftdata <- data.frame(classe=spamTest$spam)
liftdata$score <- score
```

Et nous faisons appel à la fonction lift() de caret en spécifiant la modalité cible pour le scoring, à savoir "class = yes".

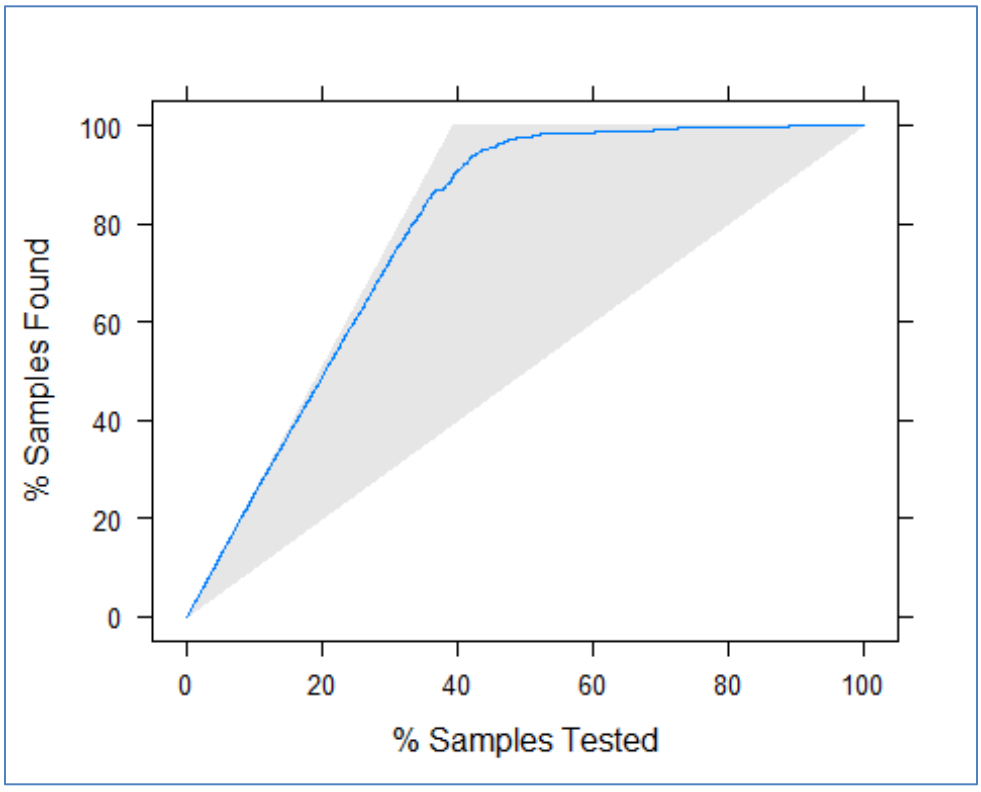
```r
#objet lift
lift_obj <- lift(classe ~ score, data=liftdata, class="yes")
print(lift_obj)
```

```
##
## Call:
## lift.formula(x = classe ~ score, data = liftdata, class = "yes")
##
## Models: score
## Event: yes (39.4%)
```

La fonction print() indique seulement la proportion des observations positives (spam = yes).

Pour obtenir la courbe proprement dite, nous appelons la fonction plot() associée à l'objet.

```r
#affichage de la courbe lift
plot(lift_obj)
```



La courbe est proche de la limite théorique (atteinte lorsque tous les spams = yes se voient attribuer un score plus élevé que les spam = no). Notre ciblage est d'excellente qualité.

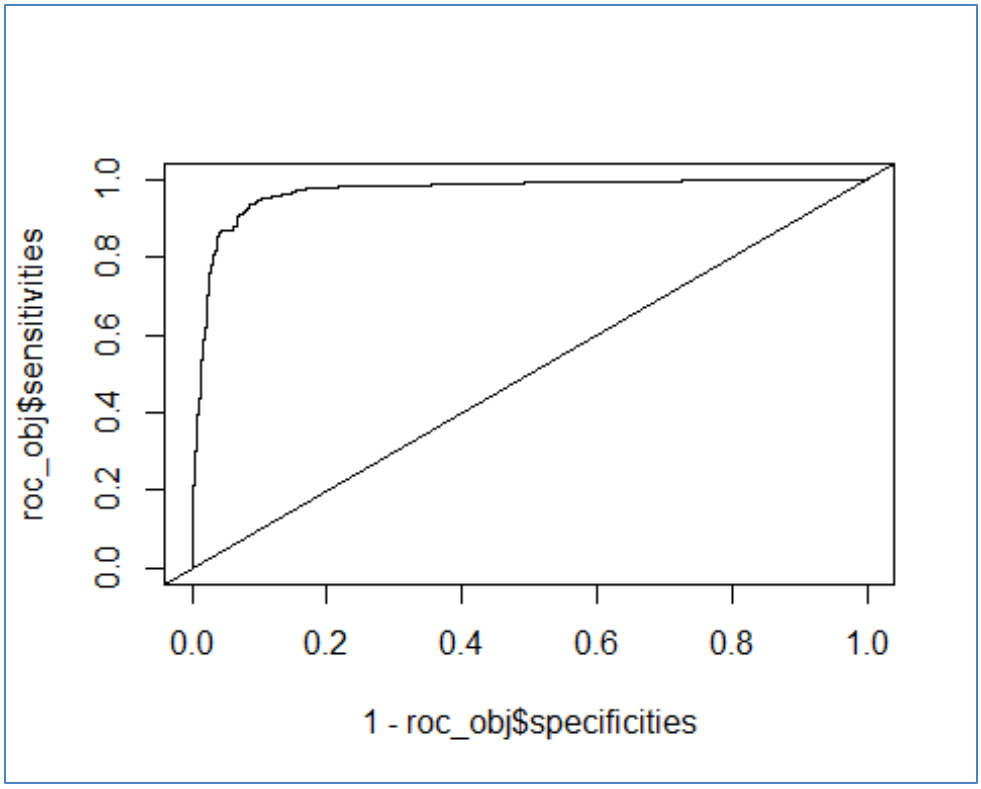
4.2 Courbe ROC

Bien que d'aspect similaire à la courbe LIFT, la courbe ROC répond à une problématique différente. Elle vise à mesurer la qualité d'un modèle en s'affranchissant des coûts de mauvaise affectation et de la représentativité de l'échantillon utilisé (les proportions des classes dans l'échantillon peut être différent de celui de la population).

Curieusement, il n'y a pas d'outil simple pour construire la courbe ROC avec "caret". Nous passons par le package "**pROC**" qu'il faudra au préalable installer. Nous disposons déjà de tous les outils nécessaires avec les classes observées et les scores calculés sur l'échantillon test.

La fonction roc() de "pROC" prend en entrée une indicatrice des classes (1 quand spam = yes, 0 sinon) et le score. Nous affichons la courbe avec plot().

```
#library
library(pROC)

## Type 'citation("pROC")' for a citation.

##
## Attaching package: 'pROC'

## The following objects are masked from 'package:stats':
##
##     cov, smooth, var

#objet roc
roc_obj <- roc(spamTest$spam=="yes",score)


#plot de l'objet roc
plot(1-roc_obj$specificities,roc_obj$sensitivities,type="l")
abline(0,1)
```



Nous pouvons accéder à la valeur de l'AUC (aire sous la courbe).

```
#aire sous la courbe
print(roc_obj$auc)

## Area under the curve: 0.969
```

4.3 Techniques de rééchantillonnage pour l'évaluation

Le schéma apprentissage-test n'est pas adapté lorsque nous traitons des bases de taille réduite. Il est plus judicieux dans ce cas d'utiliser la totalité de la base pour élaborer le modèle prédictif, puis de passer par une technique de rééchantillonnage pour en évaluer les performances, typiquement la validation croisée ou le bootstrap.

Nous allons un peu ruser dans notre tutoriel. Nous conservons notre échantillon d'apprentissage tel quel, et nous procédons par validation croisée pour en estimer les performances. Nous verrons si le taux obtenu est conforme à celui mesuré sur l'échantillon test que nous avons mis à part.

Sous “caret”, il suffit de modifier le trainControl() puis de relancer le processus de modélisation. Nous demandons une validation croisée (method=cv) avec (number=10) blocs (folds).

```
#évaluation par rééchantillonnage
fitControl <- trainControl(method="cv",number=10)
m_lr <- train(spam ~ ., data = spamTrain,method="glm",trControl=fitControl)
print(m_lr)
```

```
## Generalized Linear Model
##
## 3222 samples
##   55 predictor
##    2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 2899, 2899, 2900, 2900, 2900, 2900, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.9174432  0.8259626
```

Le **taux de succès en validation croisée** annoncé est de **91.74%**, celui mesuré sur l’échantillon test était de 92.1%. Tout cela est très cohérent. On aurait pu se passer de la subdivision des données en apprentissage-test dans notre étude.

Nous disposons du détail des résultats, le taux pour chaque fold, avec le champ $resample

```
#taux dans chaque fold
print(m_lr$resample)
```

```
##     Accuracy     Kappa Resample
## 1  0.9102167 0.8094705   Fold01
## 2  0.9226006 0.8375742   Fold02
## 3  0.9037267 0.7976321   Fold03
## 4  0.9472050 0.8893292   Fold04
## 5  0.9285714 0.8494409   Fold05
## 6  0.8944099 0.7746305   Fold06
## 7  0.9254658 0.8431118   Fold07
```

```
## 8  0.9254658 0.8431118   Fold08
## 9  0.8975155 0.7857575   Fold09
## 10 0.9192547 0.8295672   Fold10
```

5 Sélection de variables

Toutes les variables ne sont pas pertinentes pour la prédiction. Opérer une sélection de variables permet de simplifier le modèle et en améliorer la lisibilité, tout en conservant ses qualités prédictives. Parfois, pas toujours, cette réduction de la dimensionalité s'accompagne même d'une amélioration des performances.

5.1 Importance des variables

Quantifier l'impact des variables prédictives dans le modèle est une première approche de la sélection. Certaines méthodes sont capables de la fournir directement. C'est le cas de la régression logistique où l'on s'appuie sur la statistique du test de significativité des coefficients pour ordonnancer les variables.

Voici les 20 variables les plus influentes. La mesure a été normalisée à 100 par rapport à la première variable.

```
#importance des variables - intrinsèque au modèle
print(varImp(m_lr))
```

```
## glm variable importance
##
##   only 20 most important variables shown (out of 55)
##
##                             Overall
## cf_dollar                    100.00
## wf_remove                     98.48
## wf_our                        82.65
## cf_pexlam                     78.70
## wf_hp                         73.82
## wf_business                   71.17
## wf_000                        62.83
## wf_re                         62.14
```

```
## wf_your                        56.62
## wf_edu                         56.12
## capital_run_length_longest     52.19
## wf_85                          50.41
## wf_free                        49.00
## wf_internet                    46.74
## wf_mail                        45.82
## capital_run_length_total       44.38
## wf_meeting                     42.04
## wf_order                       39.52
## wf_you                         38.37
## wf_money                       35.13
```

L'importance des variables est basée sur les propriétés de l'approche glm(). La plus influente est la fréquence des points d'exclamation, puis celle du caractère $, etc.

Remarque : Certaines méthodes (ou tout du moins implémentation de méthodes) ne sont pas capables de fournir ce type d'information (ex. svm). "caret" propose alors une mesure d'influence indépendante des modèles (http://topepo.github.io/caret/variable-importance.html#model-independent-metrics). Elle s'appuie sur l'AUC de la courbe ROC induite par les valeurs des variables c.-à-d. où l'on utiliserait les valeurs de la variable en guise de score.

5.2 Méthode intégrée de sélection

La sélection peut s'appuyer sur les techniques implémentées dans les packages sous-jacents. Pour la régression logistique, il s'agirait de faire appel à la procédure stepAIC() du package **MASS**. Nous faisons appel à la procédure train() de nouveau, mais avec un paramétrage différent.

```
#méthode intégrée de sélection
m_lrs <- train(spam ~ ., data = spamTrain, method="glmStepAIC",
trControl=trainControl("none"))
```

```
## Start:  AIC=1522.74
## .outcome ~ wf_make + wf_address + wf_all + wf_3d + wf_our + wf_over +
##     wf_remove + wf_internet + wf_order + wf_mail + wf_receive +
##     wf_will + wf_people + wf_report + wf_addresses + wf_free +
```

```
##     wf_business + wf_email + wf_you + wf_credit + wf_your + wf_font +
##     wf_000 + wf_money + wf_hp + wf_hpl + wf_lab + wf_labs + wf_telnet +
##     wf_857 + wf_data + wf_415 + wf_85 + wf_technology + wf_1999 +
##     wf_parts + wf_pm + wf_direct + wf_cs + wf_meeting + wf_original +
##     wf_project + wf_re + wf_edu + wf_table + wf_conference +
##     cf_pvirgule + cf_parenthese + cf_crochet + cf_pexlam + cf_dollar +
##     cf_diese + capital_run_length_average + capital_run_length_longest +
##     capital_run_length_total
##
##
##
## Je vous fais grâce du détail des étapes…
##
```

Nous affichons le modèle final obtenu :

```
print(m_lrs$finalModel)
```

```
##
## Call:  NULL
##
## Coefficients:
##                (Intercept)                  wf_address
##                  -2.150062                   -0.197590
##                      wf_3d                      wf_our
##                   2.083277                    0.631954
##                    wf_over                   wf_remove
##                   0.370776                    2.476380
##                wf_internet                    wf_order
##                   0.510667                    0.938999
##                    wf_mail                     wf_will
##                   0.344241                   -0.178474
##                    wf_free                 wf_business
##                   0.297277                    1.374727
##                   wf_email                      wf_you
##                   0.302520                    0.114205
##                  wf_credit                     wf_your
##                   0.856293                    0.215449
##                     wf_000                    wf_money
```

```
##                   2.535339                    0.681695
##                      wf_hp                      wf_hpl
##                  -2.679904                   -0.773425
##                     wf_lab                     wf_data
##                  -2.554590                   -0.751810
##                      wf_85               wf_technology
##                  -3.267218                    0.923913
##                      wf_pm                  wf_direct
##                  -0.751557                   -1.820233
##                      wf_cs                 wf_meeting
##                 -40.461941                   -2.078353
##                 wf_project                       wf_re
##                  -1.143661                   -0.697887
##                     wf_edu                    wf_table
##                  -1.137246                   -2.058078
##              wf_conference                 cf_pvirgule
##                  -2.856604                   -0.650465
##                  cf_pexlam                   cf_dollar
##                   0.755503                    5.955736
##                   cf_diese  capital_run_length_average
##                   2.064740                    0.056847
## capital_run_length_longest    capital_run_length_total
##                   0.012129                    0.000809
##
## Degrees of Freedom: 3221 Total (i.e. Null);  3182 Residual
## Null Deviance:       4321
## Residual Deviance: 1425  AIC: 1505
```

L'AIC est maintenant de 1503. Il était de 1523 pour le modèle incluant l'ensemble des variables (section 3.2). Le nombre de variables est égal au nombre de coefficients - 1.

```
#nombre de variables sélectionnées
print(length(m_lrs$finalModel$coefficients)-1)
```

```
## [1] 39
```

39 variables ont été sélectionnées parmi les 55 candidates.

Voyons ses performances sur l'échantillon test.

```r
#application sur le test set - mesure des performances
print(confusionMatrix(data=predict(m_lrs,newdata =
spamTest),reference=spamTest$spam,positive="yes"))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  no yes
##        no  792  68
##        yes  44 475
##
##                Accuracy : 0.9188
##                  95% CI : (0.9031, 0.9327)
##     No Information Rate : 0.6062
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.8286
##  Mcnemar's Test P-Value : 0.02976
##
##             Sensitivity : 0.8748
##             Specificity : 0.9474
##          Pos Pred Value : 0.9152
##          Neg Pred Value : 0.9209
##              Prevalence : 0.3938
##          Detection Rate : 0.3445
##    Detection Prevalence : 0.3764
##       Balanced Accuracy : 0.9111
##
##        'Positive' Class : yes
##
```

L'*accuracy* est de 91.88%, contre 92.1% avec l'ensemble des variables. Nous avons un modèle plus simple avec des performances (quasiment) identiques. Tout est pour le mieux.

5.3 Recursive feature elimination (RFE)

Le package ‘’caret’’ propose une autre approche pour les méthodes de machine learning qui ne disposent pas intrinsèquement d’un mécanisme de sélection de variables. Elle procède en plusieurs étapes :

- L’importance des variables permet de les ordonner par ordre d’influence décroissante.
- Nous définissons des scénarios de solutions imbriquées où nous fixons le nombre de variables à retenir (ex. les 5 meilleurs variables, puis les 10 meilleurs c.-à-d. les 5 précédentes + les 5 suivantes, etc.).
- Pour chaque scénario, nous mesurons les performances prédictives à l’aide des techniques de rééchantillonnage.

RFE limite l’approche à des familles de techniques de modélisation. La documentation n’est pas très diserte à ce sujet. J’ai cru comprendre que lrFuncs était relative à la régression logistique. Nous paramétrons l’algorithme de recherche comme suit.

```
#méthode recursive feature elimination
selPrm <- rfeControl(functions=lrFuncs,method="cv",number=10)
```

Le taux de succès qui guidera la recherche de la meilleure solution sera mesuré en validation croisée (method = cv) avec 10 folds (number = 10).

Nous lançons la procédure en testant des configurations à (2, 5, 10, 20, 40) variables.

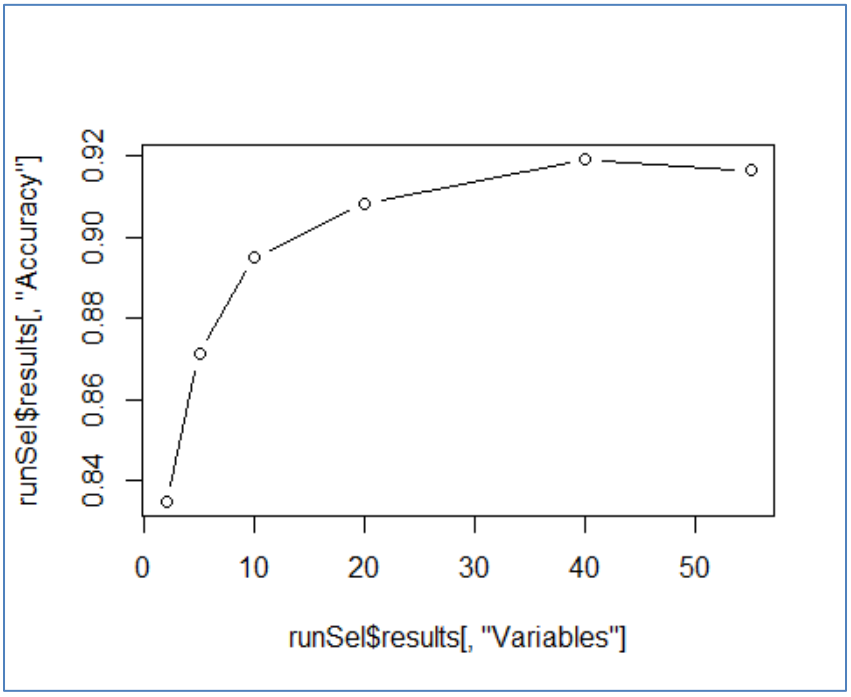
```
#recherche du meilleur subset
runSel <- rfe(spamTrain[-56],spamTrain$spam,sizes=c(2,5,10,20,40),rfeControl=selPrm)
print(runSel)
```

```
##
## Recursive feature selection
##
## Outer resampling method: Cross-Validated (10 fold)
##
## Resampling performance over subset size:
##
##  Variables Accuracy  Kappa AccuracySD KappaSD Selected
##          2   0.8349 0.6350    0.02415 0.05567
```

```
##           5   0.8712 0.7212    0.02231 0.05082
##          10   0.8951 0.7760    0.01616 0.03619
##          20   0.9081 0.8053    0.01346 0.02997
##          40   0.9193 0.8298    0.01312 0.02859        *
##          55   0.9165 0.8239    0.01635 0.03568
##
## The top 5 variables (out of 40):
##    cf_dollar, wf_remove, wf_our, cf_pexlam, wf_hp
```

La solution à 40 variables s'avère être la meilleure. Néanmoins, nous remarquons qu'à partir de 20 variables, les gains consécutifs à l'adjonction de nouvelles variables sont très faibles. Cela apparaît très clairement lorsque nous mettons en relation le nombre de variables et les performances dans un graphique.

```
plot(runSel$results[,"Variables"],runSel$results[,"Accuracy"],type="b")
```



La solution à 20 variables mérite réellement toute notre attention, voici la liste :

```
#liste
print(predictors(runSel)[1:20])

##  [1] "cf_dollar"                 "wf_remove"
##  [3] "wf_our"                    "cf_pexlam"
```

```
##  [5] "wf_hp"                       "wf_business"
##  [7] "wf_000"                      "wf_re"
##  [9] "wf_edu"                      "wf_your"
## [11] "capital_run_length_longest" "wf_85"
## [13] "wf_free"                     "wf_mail"
## [15] "capital_run_length_total"    "wf_internet"
## [17] "wf_meeting"                  "wf_order"
## [19] "wf_you"                      "wf_money"
```

6 Optimisation des paramètres des algorithmes d'apprentissage

Je m'en suis tenu à la régression logistique jusqu'à présent parce qu'elle n'est pas paramétrée. Les manipulations sont simplifiées. Elle est plutôt un cas particulier car, de manière générale, les algorithmes sont assortis de toute une série de paramètres dont on a parfois du mal à cerner la vraie teneur. Dans cette section, nous utiliserons la méthode SVM (support vector machine) avec un noyau linéaire.

6.1 Spécification des paramètres d'apprentissage

Le paramètre de coût (C) est essentiel pour un SVM lineaire. Il détermine l'aptitude du modèle à plus ou moins coller aux donnes d'apprentissage. Nous le fixons à **C = 0.1** dans un premier temps. Nous utilisons une validation croisée en 5 blocs pour estimer les performances.

```
#paramètres d'apprentissage
fitControl <- trainControl(method="cv",number=5)

#modélisation avec paramètre de la technique d'apprentissage
#SVM avec noyau linéaire, C = 0.1
m_svm <- train(spam ~ ., data =
spamTrain,method="svmLinear",trControl=fitControl,tuneGrid=data.frame(C=0.1))
print(m_svm)
```

```
## Support Vector Machines with Linear Kernel
##
## 3222 samples
##   55 predictor
##    2 classes: 'no', 'yes'
```

```
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 2578, 2577, 2578, 2578, 2577
## Resampling results:
##
##   Accuracy   Kappa
##   0.9214786  0.8346672
##
## Tuning parameter 'C' was held constant at a value of 0.1
```

Le **taux de succès en validation croisée** est de **92.14%.**

Si nous appliquons le modèle sur notre échantillon test :

```
#evaluation en test - svm linéaire
print(confusionMatrix(data=predict(m_svm,newdata =
spamTest),reference=spamTest$spam,positive="yes"))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  no yes
##        no  790  59
##        yes  46 484
##
##                Accuracy : 0.9239
##                  95% CI : (0.9086, 0.9373)
##     No Information Rate : 0.6062
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.8398
##  Mcnemar's Test P-Value : 0.2416
##
##             Sensitivity : 0.8913
##             Specificity : 0.9450
##          Pos Pred Value : 0.9132
##          Neg Pred Value : 0.9305
##              Prevalence : 0.3938
```

```
##          Detection Rate : 0.3510
##    Detection Prevalence : 0.3843
##       Balanced Accuracy : 0.9182
##
##        'Positive' Class : yes
##
```

La performance mesurée **en test** est assez proche, avec **92.39%.**

6.2 Grille de recherche des paramètres optimaux

Mais rien ne nous assure que C = 0.1 est la bonne valeur du paramètre de coût. “caret” nous donne la possibilité d’évaluer différents scénarios de coût avec l’option tuneGrid.

```
#svm avec différentes valeurs de C
m_svmg <- train(spam ~ ., data =
spamTrain,method="svmLinear",trControl=fitControl,tuneGrid=data.frame(C=c(0.05,0.1,0.
5,1,10)))
print(m_svmg)
```

```
## Support Vector Machines with Linear Kernel
##
## 3222 samples
##   55 predictor
##    2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 2577, 2578, 2578, 2577, 2578
## Resampling results across tuning parameters:
##
##   C      Accuracy   Kappa
##    0.05  0.9211695  0.8338955
##    0.10  0.9224127  0.8365304
##    0.50  0.9217940  0.8353043
##    1.00  0.9227247  0.8371965
##   10.00  0.9227252  0.8374330
##
```

```
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was C = 10.
```

La procédure nous dit que la meilleure valeur pour C est C = 10, avec un taux de succès en validation croisée de 92.27%.

Remarque : Si on sait un tant soit peu programmer, cette recherche est l'affaire d'une boucle très facile à implémenter. L'énorme avantage de "caret" est qu'il nous facilite grandement la tâche. Une ligne de code suffit à mettre en place l'étude, collecter les résultats, et déterminer la meilleure solution.

Voyons les performances de ce modèle optimisé sur notre échantillon test :

```
#evaluation en test - svm linéaire avec C optimisé
print(confusionMatrix(data=predict(m_svmg,newdata =
spamTest),reference=spamTest$spam,positive="yes"))

## Confusion Matrix and Statistics
##
##           Reference
## Prediction  no yes
##        no  791  60
##        yes  45 483
##
##                Accuracy : 0.9239
##                  95% CI : (0.9086, 0.9373)
##     No Information Rate : 0.6062
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.8397
##  Mcnemar's Test P-Value : 0.1719
##
##             Sensitivity : 0.8895
##             Specificity : 0.9462
##          Pos Pred Value : 0.9148
##          Neg Pred Value : 0.9295
##              Prevalence : 0.3938
##          Detection Rate : 0.3503
```

```
##    Detection Prevalence : 0.3829
##       Balanced Accuracy : 0.9178
##
##        'Positive' Class : yes
##
```

L'accuracy reste de marbre avec **92.39%**.

7 Comparaison de modèles

La comparaison des modèles est l'aspect qui m'a le plus séduit dans "caret". L'harmonisation des prototypes des fonctions d'apprentissage et de test, l'automatisation (même fruste) de la recherche des paramètres optimaux pour les méthodes qui en sont pourvues, constituent autant d'atouts qui nous facilitent la vie.

Mettons que nous souhaitons comparer trois méthodes linéaires suivant notre schéma apprentissage-test : la régression logistique, l'analyse discriminante et le support vector machine. Cette dernière est paramétrée, "caret" va s'appuyer sur la validation croisée intégrée dans la procédure train() pour en déterminer la meilleure configuration. C'est le modèle associé qui sera appliqué sur l'échantillon test.

Nous construisons une fonction pour combiner l'apprentissage et le test. Nous récupérons les taux de succès, à la fois en validation croisée (fournie par train) et en test (prediction et confusionMatrix).

```
#fonction pour train-test
#approche indique la technique de machine learning à évaluer
evaluation <- function(approche){
  modele <- train(spam ~ ., data=spamTrain, trControl=trainControl(method="cv",number=5), method=approche)
  prediction <- predict(modele,newdata=spamTest)
  res <- confusionMatrix(data=prediction,reference=spamTest$spam,positive="yes")
  return(c(max(modele$results[,"Accuracy"]),res$overall["Accuracy"]))
}
```

Il ne nous reste plus qu'à énumérer les méthodes à éprouver, à lancer la fonction evaluation() et à mettre en forme les résultats.

```r
#liste des méthodes à tester
methodes <- c("glm","lda","svmLinear")

#lancement de l'expérimentation
perfs <- sapply(methodes,evaluation)

#mise en forme des résultats
colnames(perfs) <- methodes
rownames(perfs) <- c("CV","Test set")

#affichage
print(perfs)
```

```
##                 glm       lda svmLinear
## CV       0.9189927 0.8839207 0.9230305
## Test set 0.9209572 0.8817984 0.9253082
```

Sur notre jeu de données, le SVM se démarque légèrement de la régression logistique, l'analyse discriminante est à la traine.

Mais la véritable information est la puissance de l'outil qui nous permet de mener des expérimentations à grande échelle relativement simplement.

8 Traitement des classes déséquilibrées

Le traitement des classes déséquilibrées est le dernier aspect de "caret" que je mets en exergue dans ce tutoriel (http://topepo.github.io/caret/subsampling-for-class-imbalances.html).

Nous avons noté que les messages frauduleux étaient plus rares (39.4%) que les messages sains (60.6%). Si nous voulons améliorer la sensibilité du système c.-à-d. la capacité à retrouver les spams, une approche simple consiste à équilibrer les proportions des classes dans notre échantillon d'apprentissage. Cela passe par un sous-échantillonnage chez les individus négatifs (spam = no), ou un sur-échantillonnage chez les positifs (spam = yes).

Rappelons que dans la configuration initiale, utilisation brute de l'échantillon d'apprentissage, la **sensibilité** de la régression logistique était **86.74%** sur l'ensemble de test.

8.1 Sous-échantillonnage des négatifs

Nous passons par une approche explicite dans cette section. Tout d'abord, avec downSample(), nous construisons un data frame pour l'apprentissage où les proportions des classes sont équilibrées en passant par un sous-échantillonnage parmi les négatifs.

```
spamDown <- downSample(x=spamTrain[-56],y=spamTrain$spam,yname="spam")
print(dim(spamDown))
```

```
## [1] 2540   56
```

Sur les 3222 observations initiales de **spamTrain**, 2540 sont retenues. Lorsque nous nous intéressons à la fréquence absolue des classes :

```
print(table(spamDown$spam))
```

```
##
##   no  yes
## 1270 1270
```

Nous notons que vous avons la totalité des positifs (spam yes = 1270) et une fraction des négatifs (1270 sur 1952) (voir section 3.1).

Lançons de nouveau l'apprentissage et le test :

```
#apprentissage
modele <- train(spam ~ ., data=spamDown, trControl=trainControl(method="none"),
method="glm")

#prédiction
prediction <- predict(modele,newdata=spamTest)

#matrice de confusion
res <- confusionMatrix(data=prediction,reference=spamTest$spam,positive="yes")
print(res)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  no yes
##        no  780  43
```

```
##        yes  56 500
##
##               Accuracy : 0.9282
##                 95% CI : (0.9133, 0.9413)
##    No Information Rate : 0.6062
##    P-Value [Acc > NIR] : <2e-16
##
##                  Kappa : 0.8503
##  Mcnemar's Test P-Value : 0.2278
##
##             Sensitivity : 0.9208
##             Specificity : 0.9330
##          Pos Pred Value : 0.8993
##          Neg Pred Value : 0.9478
##              Prevalence : 0.3938
##          Detection Rate : 0.3626
##    Detection Prevalence : 0.4032
##       Balanced Accuracy : 0.9269
##
##        'Positive' Class : yes
##
```

La **sensibilité** est passée à **92.08%** (vs. 86.74%). L'amélioration est notable.

8.2 Sur-échantillonnage des positifs

Nous pourrons utiliser le même cheminement pour le sur-échantillonnage des positifs avec la commande upSample(). Mais je me suis rendu compte à la lecture de la documentation qu'il existe un moyen autrement plus simple de procéder, en modifiant le paramétrage sampling de l'apprentissage.

```
#paramétrage avec sur-échantillonnage
ctrlUp <- trainControl(method="none",sampling="up")
```

Nous relançons l'expérimentation avec ces paramètres.

```
#apprentissage
modele <- train(spam ~ ., data=spamDown, trControl=ctrlUp, method="glm")
```

```r
#prédiction
prediction <- predict(modele,newdata=spamTest)

#matrice de confusion
res <- confusionMatrix(data=prediction,reference=spamTest$spam,positive="yes")
print(res)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  no yes
##        no  780  43
##        yes  56 500
##
##                Accuracy : 0.9282
##                  95% CI : (0.9133, 0.9413)
##     No Information Rate : 0.6062
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.8503
##  Mcnemar's Test P-Value : 0.2278
##
##             Sensitivity : 0.9208
##             Specificity : 0.9330
##          Pos Pred Value : 0.8993
##          Neg Pred Value : 0.9478
##              Prevalence : 0.3938
##          Detection Rate : 0.3626
##    Detection Prevalence : 0.4032
##       Balanced Accuracy : 0.9269
##
##        'Positive' Class : yes
##
```

L'amélioration est du même ordre qu'avec le sous-échantillonnage avec une sensibilité de 92.08%.

Remarque : S'agissant de la régression logistique, j'avais montré dans un de mes tutoriels que l'on pouvait à loisir sur la sensibilité ou la précision en modulant la constante du modèle, et cela sans avoir à passer par un réapprentissage sur des bases artificiellement équilibrées (http://tutoriels-data-mining.blogspot.fr/2010/05/traitement-des-classes-desequilibrees.html). L'avantage des techniques de sous et sur-échantillonnage est qu'elles sont génériques, applicables à toute méthode de machine learning.

9 Conclusion

L'objectif de ce tutoriel était de mettre en avant quelques fonctionnalités intéressantes du package "caret". Entendons-nous bien, un bon programmeur R peut facilement les reproduire. Le principal mérite de ce package est d'encapsuler dans des fonctions et procédures clés en main des processus types de la pratique du data mining. Il nous facilite la vie et ce n'est déjà pas mal…

10 Références

Max Kuhn, "The caret Package", http://topepo.github.io/caret/index.html