Packages R pour le Deep Learning - Perceptrons

Tutoriel Tanagra

16 décembre 2018

1 Introduction

Ce tutoriel fait suite au support de cours consacré aux perceptrons simples et multicouches. L'objectif est d'explorer le mode opératoire et l'efficacité des différents packages qui proposent la méthode. Deux documents accessibles sur le net nous ont inspirés : Deep Learning in R (R Blog, Fév. 2017) et Keras: Deep Learning in R (Datacamp, Juin 2017) [1]. Nous avons essayé de systématiser l'étude en détaillant les étapes clés que constituent la construction du modèle et son évaluation sur un échantillon test. Pour être plus réaliste, plutôt que d'appréhender une base intégrée dans un package quelconque, nous partons d'une base au format CSV (TSV – séparateur tabulation – pour être précis) qu'il faudra charger et prétraiter.

Notre schéma de travail sera donc relativement classique s'agissant d'un contexte d'analyse prédictive. Nous importons une base de données, nous la subdivisons en échantillons d'apprentissage et de test. Nous standardisons les variables avec une subtilité importante que l'on précisera. Puis, pour chaque package, nous implémenterons un perceptron multicouche avec 1 seule couche cachée à 2 neurones (ce n'est pas de l'humour…) dont nous évaluerons les performances prédictives. Certains packages proposent des fonctionnalités additionnelles. Nous essayerons de les cerner.

Pour tous les cas présentés dans ce tutoriel, nous avons voulu utiliser, lorsque cela est possible, une fonction d'activation sigmoïde dans les couches du réseau, excepté la couche d'entrée bien sûr.

Sans tout dévoiler d'entrée, nous pouvons dire que tous les packages ont répondu au cahier des charges, mais avec plus ou moins d'habilité. Voyons cela.

[1] Par rapport à ces documents repères, nous avons constaté que deux packages ne sont plus proposés sur le CRAN à ce jour (16.12.208) : FCNN4R et DARCH. Il est certes possible d'accéder aux versions plus anciennes, mais nous avons préféré les exclure de l'étude.

2 Base de données - Préparation des échantillons

2.1 SPAMBASE dataset

Nous utilisons la base Spambase dataset du dépôt UCI dans ce tutoriel. L'objectif est d'identifier les courriels frauduleux (spam = yes) à partir de leurs caractéristiques (fréquence relative des mots, nombre de caractères en majuscules, etc.).

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

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

#taille de la base
print(dim(spam.all))

## [1] 4601   57
```

Nous disposons de 4601 observations et 57 colonnes.

```
#distribution des classes
print(table(spam.all$spam))

##
##   no  yes
## 2788 1813
```

Avec 1813 courriels frauduleux

2.2 Subdivision en échantillons d'apprentissage et de test

Par rapport à la base initiale, nous avons adjoint aux données une colonne supplémentaire "status" permettant de discerner les individus appartenant aux échantillons d'apprentissage (train) et de test (test).

```
#taille des échantillons d'apprentissage et de test
print(table(spam.all$status))

##
##  test train
##  1000  3601
```

Nous nous en servons pour partitionner la base. Cette colonne "status" est par la suite exclue de l'étude.

```
#apprentissage
spam.train <- spam.all[spam.all$status=="train",-ncol(spam.all)]
print(dim(spam.train))

## [1] 3601   56
```

```r
#test
spam.test <- spam.all[spam.all$status=="test",-ncol(spam.all)]
print(dim(spam.test))
```

```
## [1] 1000   56
```

2.3 Standardisation des données

Les variables étant exprimées dans des unités différentes (fréquences relatives, comptage, longueur de chaînes de caractères…), nous les standardisons. Attention, il faut tenir compte d'une subtilité importante à ce stade. Seuls les paramètres calculés sur l'échantillon d'apprentissage doivent intervenir dans la trasnformation des données. De fait, nous devons utiliser exclusivement les moyennes et écarts-type calculés sur "train" pour normaliser les deux échantillons.

Calculons ces paramètres.

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

```
##                    wf_make                 wf_address
##                 0.10670925                 0.21287698
##                     wf_all                      wf_3d
##                 0.28246043                 0.08264371
##                     wf_our                    wf_over
##                 0.31305748                 0.09693141
## etc…
##                  cf_dollar                    cf_hash
##                 0.07670925                 0.04270647
## capital_run_length_average capital_run_length_longest
##                  5.51936157                54.30658151
##   capital_run_length_total
##                288.61594002
```

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

```
##                    wf_make                 wf_address
##                 0.31189675                 1.28778498
##                     wf_all                      wf_3d
##                 0.50569084                 1.57625573
##                     wf_our                    wf_over
##                 0.68116099                 0.28140461
## etc…
##                   cf_dollar                    cf_hash
##                 0.24686295                 0.41904636
## capital_run_length_average capital_run_length_longest
##                 35.22183527                211.97054927
##   capital_run_length_total
##                584.51434015
```

Transformons l'échantillon d'apprentissage à l'aide de ces paramètres.

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

#vérification
print(colMeans(spam.train.cr))

##                    wf_make                 wf_address
##               -2.072419e-17               -4.564910e-18
##                     wf_all                       wf_3d
##                2.769391e-17               -2.697709e-19
##                     wf_our                    wf_over
##               -1.341581e-17                4.051863e-18
## Etc.
##                   cf_dollar                    cf_hash
##               -1.426125e-17               -1.458931e-17
## capital_run_length_average capital_run_length_longest
##                1.015091e-17                1.463038e-17
##   capital_run_length_total
##                1.778826e-17

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

Il est tout à fait normal que les moyennes des variables soient toutes nulles (aux erreurs de troncature près).

Faisons de même pour l'échantillon 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))

#moyenne pas forcément nulle - c'est tout à fait normal
print(colMeans(spam.test.cr))

##                    wf_make                 wf_address
##               -0.0318029839                0.0004915583
##                     wf_all                       wf_3d
##               -0.0164140361               -0.0502606961
## Etc.
##                   cf_dollar                    cf_hash
##               -0.0167471360                0.0168180188
## capital_run_length_average capital_run_length_longest
##               -0.0428263194               -0.0463157808
##   capital_run_length_total
##               -0.0419287233

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

Et il est tout à fait normal également que les moyennes des variables transformées de l'échantillon test ne soient pas forcément nulles puisque les paramètres de centrage (et de réduction) ont été calculés sur le premier échantillon.

2.4 Fonction d'évaluation des performances

Pour évaluer les performances prédictives des modèles, nous écrivons une petite fonction maison. Elle prend en entrée la cible, la prédiction et une indication sur la modalité-cible ("yes" pas défaut). Elle affiche la matrice de confusion, le taux d'erreur, le rappel, la précision et le F1-Score (F-Mesure).

```
#fonction pour evaluation des modèles
evaluation.prediction <- function(yobs,ypred,posLabel="yes"){
  #matrice de confusion
  mc <- table(yobs,ypred)
  print("Matrice de confusion")
  print(mc)
  #taux d'erreur
  err <- 1-sum(diag(mc))/sum(mc)
  print(paste("Taux d'erreur =", err))
  #rappel
  recall <- mc[posLabel,posLabel]/sum(mc[posLabel,])
  print(paste("Rappel =", round(recall,3)))
  #precision
  precision <- mc[posLabel,posLabel]/sum(mc[,posLabel])
  print(paste("Precision =",round(precision,3)))
  #F1-Measure
  f1 <- 2.0*(precision*recall)/(precision+recall)
  print(paste("F1-Score =",round(f1,3)))
}
```

Ça y est, nous sommes parés pour lancer l'étude des packages.

3 Packages R pour le perceptron

Construire et évaluer le modèle sont les deux tâches essentielles que nous analyserons pour chaque package. Pour certains d'entre eux, nous mettrons en lumière les fonctionnalités additionnelles qui me paraissent intéressantes.

3.1 Le package "nnet"

Le package "nnet" est ancien mais très populaire, notamment parce qu'il est performant, robuste, facile à utiliser. Seule restriction, de taille si je puis dire, son perceptron multicouche n'accepte qu'une seule couche cachée.

3.1.1 Perceptron simple

Dans cette section, nous commençons par un perceptron simple pour circonscrire les performances d'un classifieur linéaire sur nos données. L'intérêt aussi est de pouvoir identifier facilement, grâce à la valeur des coefficients, l'impact des variables dans la prédiction.

Nous chargeons le package et nous lançons la construction du modèle. Les deux paramètres clés sont "skip = TRUE" (pas de couche cachée) et "size = 0" (par conséquent, pas de neurones dans la couche cachée). Pour le reste (formule, data), nous retrouvons les paramètres usuels utilisés dans les fonctions R pour la prédiction.

```
#chargement
library(nnet)

#apprentissage perceptron simple
#set.seed() pour rendre l'expérimentation reproductible pour le lecteur
set.seed(100)
ps.nnet <- nnet(spam ~ ., data = spam.train.cr, skip = TRUE, size = 0)

## # weights:  56
## initial  value 6092.193870
## iter  10 value 1167.543337
## iter  20 value 885.403296
## iter  30 value 821.564406
## iter  40 value 799.519044
## iter  50 value 789.815850
## iter  60 value 786.345302
## iter  70 value 785.407235
## iter  80 value 785.392354
## final  value 785.388675
## converged
```

La convergence et les temps d'exécution sont rapides. Ce sont des caractéristiques fortes de ce package. Il s'appuie sur l'algorithme d'optimisation BFGS, efficace pour les petits ensembles de données. Pour les grandes bases, c'est une autre histoire.

Voyons ce qu'il en est maintenant des performances prédictives : nous effectuons la prédiction sur l'échantillon test, et nous confrontons les valeurs prédites et observées.

```
#prediction
pred.ps.nnet <- predict(ps.nnet,newdata=spam.test.cr,type="class")

#evaluation
evaluation.prediction(spam.test.cr$spam,pred.ps.nnet)

## [1] "Matrice de confusion"
##      ypred
## yobs   no yes
##   no  584  25
```

```
##   yes  72 319
## [1] "Taux d'erreur = 0.097"
## [1] "Rappel = 0.816"
## [1] "Precision = 0.927"
## [1] "F1-Score = 0.868"
```

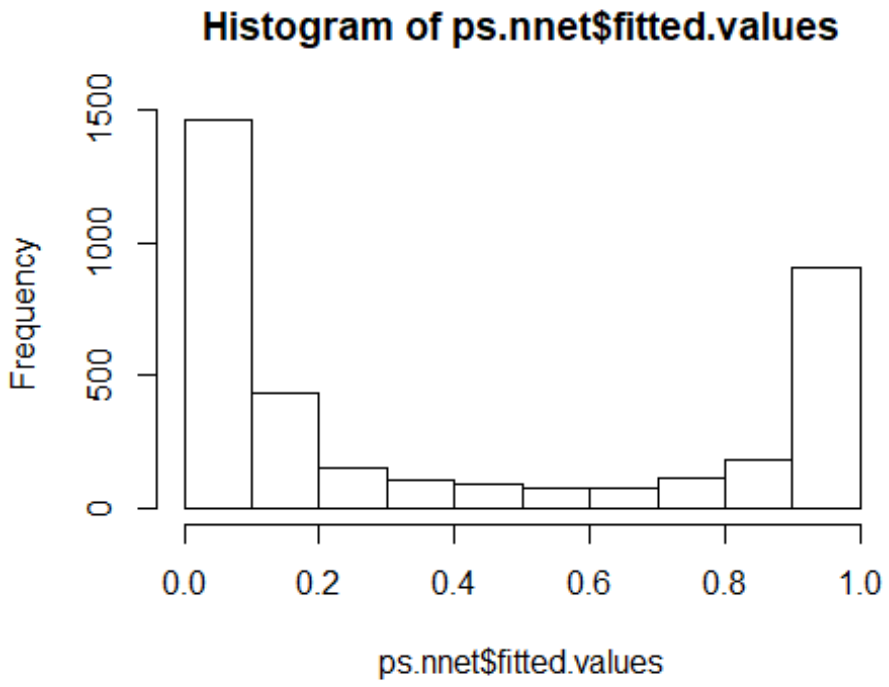
Nous avons (72+25) 97 observations mal classées.

3.1.2 Sorties additionnelles

"nnet" propose des outils supplémentaires pour mieux cerner la qualité du modèle.

Nous disposons par exemple les probabilités conditionnelles calculées durant l'apprentissage. Nous affichons l'histogramme de fréquences.

```
#distribution des probas estimées en apprentissage
hist(ps.nnet$fitted.values)
```



Les décisions sont tranchées sur une très grande majorité des observations, les probabilités sont proches de 0 ou 1, ce qui explique la bonne tenue du modèle en classement.

Nous disposons d'un affichage détaillé du réseau :

```
#les poids
print(summary(ps.nnet))
```

```
## a 55-0-1 network with 56 weights
## options were - skip-layer connections  entropy fitting
##   b->o  i1->o  i2->o  i3->o  i4->o  i5->o  i6->o  i7->o  i8->o  i9->o
##  -4.01  -0.06  -0.15   0.12   3.65   0.37   0.17   1.08   0.30   0.10
## i10->o i11->o i12->o i13->o i14->o i15->o i16->o i17->o i18->o i19->o
##   0.10  -0.01  -0.11  -0.01   0.08   0.13   0.59   0.55   0.11   0.19
## i20->o i21->o i22->o i23->o i24->o i25->o i26->o i27->o i28->o i29->o
##   0.40   0.29   0.30   0.81   0.19  -3.71  -0.59  -1.57  -0.06  -0.93
## i30->o i31->o i32->o i33->o i34->o i35->o i36->o i37->o i38->o i39->o
##  -0.16  -0.53  -0.28  -1.34   0.36  -0.13  -0.04  -0.21  -0.19 -16.46
## i40->o i41->o i42->o i43->o i44->o i45->o i46->o i47->o i48->o i49->o
##  -1.73  -0.50  -0.88  -1.05  -1.22  -0.15  -1.08  -0.37   0.01  -0.12
## i50->o i51->o i52->o i53->o i54->o i55->o
##   0.30   1.62   0.93  -0.50   1.75   0.71
```

L'affichage n'est pas très glamour, mais nous y observons quand-même l'essentiel : "b->o" représente la connexion entre le biais et la sortie, le coefficient associé est -4.01 ; "i1->o", la connexion entre la première variable et la sortie avec -0.06 ; etc.

Nous pouvons récupérer explicitement ces poids synaptiques.

```
#récupération des poids synaptiques
poids <- ps.nnet$wts[2:length(ps.nnet$wts)]
names(poids) <- ps.nnet$coefnames
print(poids)

##                   wf_make                wf_address
##                -0.056684193               -0.151373511
##                      wf_all                      wf_3d
##                 0.124048301                3.652829080
##                      wf_our                    wf_over
##                 0.372617258                0.168939895
##                   wf_remove                wf_internet
##                 1.076394365                0.298568138
##                    wf_order                    wf_mail
##                 0.095705882                0.097601785
##                  wf_receive                    wf_will
##                -0.008604744               -0.107318258
##                   wf_people                   wf_report
##                -0.006194861                0.083509616
##                wf_addresses                    wf_free
##                 0.134693287                0.589113528
##                 wf_business                   wf_email
##                 0.550225373                0.109544653
##                      wf_you                  wf_credit
##                 0.186685410                0.396214353
##                     wf_your                    wf_font
##                 0.293801203                0.300433743
##                      wf_000                   wf_money
##                 0.808928290                0.188198737
##                       wf_hp                    wf_hpl
##                -3.705496004               -0.593039291
##                      wf_lab                    wf_labs
##                -1.568977594               -0.058342886
```

```
##                   wf_telnet                      wf_857
##                -0.934265104                -0.163292520
##                     wf_data                      wf_415
##                -0.534924288                -0.283375143
##                       wf_85               wf_technology
##                -1.337389932                 0.360402421
##                     wf_1999                    wf_parts
##                -0.130959386                -0.039951869
##                       wf_pm                   wf_direct
##                -0.212550147                -0.194756244
##                       wf_cs                   wf_meeting
##               -16.458485015                -1.729773533
##                  wf_original                   wf_project
##                -0.499578886                -0.881455523
##                       wf_re                       wf_edu
##                -1.054289573                -1.224630025
##                    wf_table               wf_conference
##                -0.145702625                -1.080865546
##                    cf_comma                   cf_bracket
##                -0.369737074                 0.005467564
##                cf_sqbracket                    cf_exclam
##                -0.118243045                 0.296864275
##                   cf_dollar                       cf_hash
##                 1.616983738                 0.930475433
## capital_run_length_average capital_run_length_longest
##                -0.497908434                 1.751508874
##   capital_run_length_total
##                 0.710659769
```

Pour identifier les variables importantes, il suffit de les trier selon la valeur absolue décroissante (Remarque : nous pouvons comparer les poids parce que nous avons un classifieur linéaire et que les variables ont été standardisées !).

Les 15 variables qui jouent le plus fortement dans l'identification des "spams" sont :

```r
#les variables avec les poids les plus élevés en valeur absolue
print(sort(abs(poids),decreasing=TRUE)[1:15])
```

```
##                       wf_cs                       wf_hp
##                  16.4584850                   3.7054960
##                       wf_3d capital_run_length_longest
##                   3.6528291                   1.7515089
##                  wf_meeting                   cf_dollar
##                   1.7297735                   1.6169837
##                      wf_lab                       wf_85
##                   1.5689776                   1.3373899
##                      wf_edu               wf_conference
##                   1.2246300                   1.0808655
##                   wf_remove                       wf_re
##                   1.0763944                   1.0542896
##                   wf_telnet                     cf_hash
##                   0.9342651                   0.9304754
##                  wf_project
##                   0.8814555
```

3.1.3 Perceptron multicouche

Nous jouons sur les paramètres “skip” et “size” pour spécifier un perceptron à 2 neurones dans l’unique couche cachée. Nous avons dû augmenter le nombre maximal d’itérations pour aboutir à la convergence.

```
#PMC avec 1 couche cachée à 2 neurones
set.seed(100)
pm.nnet <- nnet(spam ~ ., data = spam.train.cr, skip = FALSE, size = 2, maxit = 1000)

## # weights:  115
## initial  value 2242.727586
## iter  10 value 773.980054
## iter  20 value 706.753408
## iter  30 value 664.859639
## iter  40 value 630.901170
## Etc...
## iter 590 value 467.190834
## iter 600 value 467.176214
## final  value 467.170019
## converged
```

Le modèle est de meilleure tenue avec (49+24) 73 observations mal classées (vs. 97 pour le perceptron simple).

```
#prediction
pred.pm.nnet <- predict(pm.nnet,newdata=spam.test.cr,type="class")

#evaluation
evaluation.prediction(spam.test.cr$spam,pred.pm.nnet)

## [1] "Matrice de confusion"
##      ypred
## yobs   no yes
##   no  585  24
##   yes  49 342
## [1] "Taux d'erreur = 0.073"
## [1] "Rappel = 0.875"
## [1] "Precision = 0.934"
## [1] "F1-Score = 0.904"
```

Voyons l’architecture du réseau :

```
#architecture
print(summary(pm.nnet))

## a 55-2-1 network with 115 weights
## options were - entropy fitting
##   b->h1  i1->h1  i2->h1  i3->h1  i4->h1  i5->h1  i6->h1  i7->h1  i8->h1
##   11.12    0.11   -0.09    0.00   -0.44   -0.09   -0.01   -4.97   -0.68
##  i9->h1 i10->h1 i11->h1 i12->h1 i13->h1 i14->h1 i15->h1 i16->h1 i17->h1
##   -0.06    0.36    0.33    0.05    0.03    0.02   -0.30   -0.06   -1.66
## i18->h1 i19->h1 i20->h1 i21->h1 i22->h1 i23->h1 i24->h1 i25->h1 i26->h1
```

```
##    0.20   -0.42   -0.47   -0.06   -0.08   -1.80   -2.63   16.38    2.86
## i27->h1 i28->h1 i29->h1 i30->h1 i31->h1 i32->h1 i33->h1 i34->h1 i35->h1
##    4.42   -0.43    4.43    1.65   -0.05    0.25    3.61   -0.25   -0.04
## i36->h1 i37->h1 i38->h1 i39->h1 i40->h1 i41->h1 i42->h1 i43->h1 i44->h1
##    0.34    0.10    0.69   16.64    2.25    0.13    1.79    1.95   24.68
## i45->h1 i46->h1 i47->h1 i48->h1 i49->h1 i50->h1 i51->h1 i52->h1 i53->h1
##    0.65    0.66    0.17   -0.17   -0.06   -2.32   -3.14    0.23  -19.83
## i54->h1 i55->h1
##   -0.53    0.01
##   b->h2  i1->h2  i2->h2  i3->h2  i4->h2  i5->h2  i6->h2  i7->h2  i8->h2
##    7.47   26.80  -75.07  -17.20   85.32   98.49   39.82    2.10  -16.22
##  i9->h2 i10->h2 i11->h2 i12->h2 i13->h2 i14->h2 i15->h2 i16->h2 i17->h2
##   -6.75   59.99   91.79  -21.97   -7.60   26.84  -21.25   63.40   -7.41
## i18->h2 i19->h2 i20->h2 i21->h2 i22->h2 i23->h2 i24->h2 i25->h2 i26->h2
##   55.85  -35.16    8.44   48.06   -0.07   21.69  -19.40 -155.61   46.65
## i27->h2 i28->h2 i29->h2 i30->h2 i31->h2 i32->h2 i33->h2 i34->h2 i35->h2
##   15.05  -40.58  -57.73   32.87  -85.64  -81.51  -62.89   54.68  -53.22
## i36->h2 i37->h2 i38->h2 i39->h2 i40->h2 i41->h2 i42->h2 i43->h2 i44->h2
##   10.30   -4.80   20.04  -94.33 -183.63  -18.47  -66.60   21.12   23.36
## i45->h2 i46->h2 i47->h2 i48->h2 i49->h2 i50->h2 i51->h2 i52->h2 i53->h2
##   28.39  -52.81  -10.84  -60.09  -80.56   -6.39   73.97   12.13  -39.87
## i54->h2 i55->h2
##   93.19   66.53
##    b->o   h1->o   h2->o
##    0.30   -7.03    4.04
```

Discerner quelque chose dedans n'est pas très aisé. Quant à la lecture des poids, ...

```
#poids
print(pm.nnet$wts)
```

```
##   [1]  1.111652e+01  1.085586e-01 -8.625471e-02  3.043303e-03 -4.359926e-01
##   [6] -8.812659e-02 -9.620192e-03 -4.967428e+00 -6.832699e-01 -5.906676e-02
##  [11]  3.583233e-01  3.331919e-01  4.778031e-02  3.265505e-02  1.978758e-02
##  [16] -2.989460e-01 -6.035371e-02 -1.659027e+00  2.048520e-01 -4.155431e-01
##  [21] -4.709240e-01 -5.578075e-02 -7.972850e-02 -1.804565e+00 -2.632210e+00
##  [26]  1.637792e+01  2.860613e+00  4.420483e+00 -4.322619e-01  4.431094e+00
##  [31]  1.651461e+00 -5.234741e-02  2.491102e-01  3.605543e+00 -2.530031e-01
##  [36] -4.421720e-02  3.440799e-01  9.708545e-02  6.893440e-01  1.663820e+01
##  [41]  2.247792e+00  1.266412e-01  1.787240e+00  1.951890e+00  2.467519e+01
##  [46]  6.452736e-01  6.595866e-01  1.658608e-01 -1.710733e-01 -5.657152e-02
##  [51] -2.324447e+00 -3.138743e+00  2.276821e-01 -1.982560e+01 -5.331491e-01
##  [56]  1.326875e-02  7.474868e+00  2.679581e+01 -7.507320e+01 -1.720453e+01
##  [61]  8.531726e+01  9.848876e+01  3.981826e+01  2.103782e+00 -1.621735e+01
##  [66] -6.751435e+00  5.998573e+01  9.179174e+01 -2.197498e+01 -7.596535e+00
##  [71]  2.683774e+01 -2.124772e+01  6.340404e+01 -7.408605e+00  5.585275e+01
##  [76] -3.516145e+01  8.440815e+00  4.806254e+01 -6.745573e-02  2.169256e+01
##  [81] -1.940092e+01 -1.556096e+02  4.665017e+01  1.504971e+01 -4.058289e+01
##  [86] -5.773249e+01  3.287364e+01 -8.563585e+01 -8.151474e+01 -6.289351e+01
##  [91]  5.468449e+01 -5.321860e+01  1.029634e+01 -4.799768e+00  2.003631e+01
##  [96] -9.432834e+01 -1.836284e+02 -1.847366e+01 -6.660128e+01  2.111769e+01
## [101]  2.336500e+01  2.839090e+01 -5.280822e+01 -1.083619e+01 -6.009270e+01
## [106] -8.056453e+01 -6.385836e+00  7.396550e+01  1.212751e+01 -3.986648e+01
## [111]  9.318766e+01  6.652723e+01  2.989310e-01 -7.032611e+00  4.038727e+00
```

… ils sont mélangés dans un vecteur unique. Il y a certes bien 115 coefficients puisque nous en avons ((55+1)*2) 112 entre les couches d'entrée et intermédiaire, et (2+1) 3 entre cette dernière et la sortie. Mais pouvoir discerner l'influence des variables dans le modèle n'est pas possible en l'état.

3.2 Package "neuralnet"

Le package "neuralnet" est également populaire, peut-être parce qu'il est relativement souple à utiliser, parce qu'il propose une représentation graphique assez sympathique, parce qu'il est possible d'exploiter les résultats intermédiaires (Exemple : neuralnet: Train and Test Neural Networks using R)

Après avoir installé et chargé le package, nous devons coder la variable cible catégorielle "spam" en une variable numérique. L'utilisation d'une indicatrice 0/1 semble s'imposer naturellement.

```
#chargement - il faut installer au préalable
library(neuralnet)

#codage explicite de la cible - codage 1/0
#y.Train <- ifelse(spam.train$spam=="yes",1.0,0.0)
```

C'était la mauvaise idée. L'algorithme n'a jamais pu converger lors de l'apprentissage du modèle. Et pourtant j'y ai passé du temps, en essayant de jouer sur les différents paramètres.

Le problème vient de l'estimation des probabilités à l'aide de la fonction de transfert sigmoïde. Les valeurs sont très proches de 0 ou 1, à saturation, là où les dérivées (gradients) sont quasi-nulles. De fait, les corrections des coefficients se font mal durant le processus d'apprentissage. Il n'est pas possible de progresser efficacement vers la minimisation de la fonction de perte.

Pour dépasser ce piège, il est conseillé de plutôt coder la cible en (0.8, 0.2) lorsqu'on utilise la fonction de transfert sigmoïde. Nous nous situerons ainsi dans la zone où sa pente reste importante.

Nous adoptons ce codage

```
#codage explicite de la cible - codage 0.8/0.2
y.Train <- ifelse(spam.train$spam=="yes",0.8,0.2)
print(table(y.Train))

## y.Train
##  0.2  0.8
## 2179 1422
```

"neuralnet" ne sait pas gérer les expressions du type "`cible ~ .`" pour les formules. Nous devons spécifier explicitement les variables explicatives. Nous en avons p = 55, écrire la formule au clavier n'est pas très réjouissant. Heureusement, il est possible de la construire relativement facilement à partir de la liste des noms de variables.

D'abord la liste des descripteurs :

```
#formule - liste des explicatives
formule <- paste(colnames(spam.train[-ncol(spam.train)]),collapse="+")
print(formule)

## [1]
"wf_make+wf_address+wf_all+wf_3d+wf_our+wf_over+wf_remove+wf_internet+wf_order+wf_mai
l+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_m
eeting+wf_original+wf_project+wf_re+wf_edu+wf_table+wf_conference+cf_comma+cf_bracket
+cf_sqbracket+cf_exclam+cf_dollar+cf_hash+capital_run_length_average+capital_run_leng
th_longest+capital_run_length_total"
```

Puis associer la variable cible :

```
#suite formule
formule <- paste("y.Train ~ ",formule,sep="")
print(formule)

## [1] "y.Train ~
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+w
f_857+wf_data+wf_415+wf_85+wf_technology+wf_1999+wf_parts+wf_pm+wf_direct+wf_cs+wf_me
eting+wf_original+wf_project+wf_re+wf_edu+wf_table+wf_conference+cf_comma+cf_bracket+
cf_sqbracket+cf_exclam+cf_dollar+cf_hash+capital_run_length_average+capital_run_lengt
h_longest+capital_run_length_total"
```

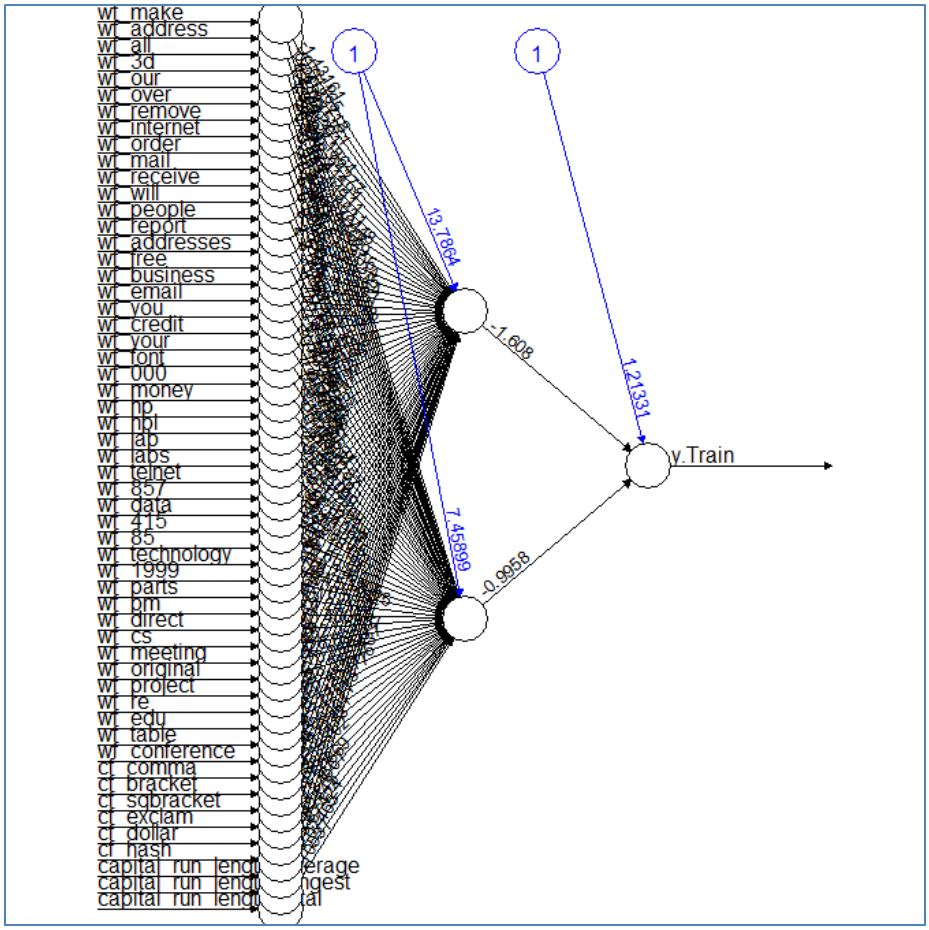
Nous pouvons maintenant lancer l'apprentissage. Grâce au codage judicieux de la variable cible, le processus est très rapide.

```
#apprentissage
set.seed(100)
pm.neural <- neuralnet::neuralnet(as.formula(formule),data=spam.train.cr, hidden =
c(2), linear.output = FALSE)
```

On observe 2 paramètres clés : "hidden" permet de spécifier le nombre de neurones dans la couche cachée, s'il y a plusieurs couches, nous utilisons un vecteur ; "linear.output = FALSE" introduit la fonction d'activation sigmoïde dans le neurone de sortie, elle est par défaut présente dans les couches intermédiaires.

"neuralnet" nous offre la possibilité appréciable d'afficher le réseau avec les poids synaptiques.

```
#affichage
plot(pm.neural)
```



Le biais est bien présent sur chaque couche. Hélas, lire sur ce graphique les poids entre les couches d'entrée et cachée n'est pas trop possible au regard du nombre de variables explicatives.

Nous utilisons la fonction compute() pour obtenir les probabilités d'affectation en prédiction sur l'échantillon test. Voici les 10 premières valeurs.

```
#prédiction - proba d'affectation
proba.pred.pm.neural <- compute(pm.neural,covariate=spam.test.cr[-
ncol(spam.test.cr)])
print(proba.pred.pm.neural$net.result[1:10])

##  [1] 0.7708836649 0.7708836649 0.1993297761 0.4881902065 0.7708836649
##  [6] 0.1993297761 0.7708836649 0.7708831757 0.1993297761 0.1993301936
```

Nous les comparons au seuil **0.5** pour obtenir les classes prédites. Il est dès lors possible de confronter les valeurs observées et prédites de la variable cible.

```
#traduire en "yes" "no" en comparant à 0.5
pred.pm.neural <- ifelse(proba.pred.pm.neural$net.result > 0.5,"yes","no")

#évaluation
evaluation.prediction(spam.test.cr$spam,pred.pm.neural)
```

```
## [1] "Matrice de confusion"
##      ypred
## yobs   no yes
##   no  579  30
##   yes  47 344
## [1] "Taux d'erreur = 0.077"
## [1] "Rappel = 0.88"
## [1] "Precision = 0.92"
## [1] "F1-Score = 0.899"
```

Les performances sont tout à fait correctes par rapport à celles de “nnet” (77 mal classés vs. 73). Les différences ne sont pas significatives.

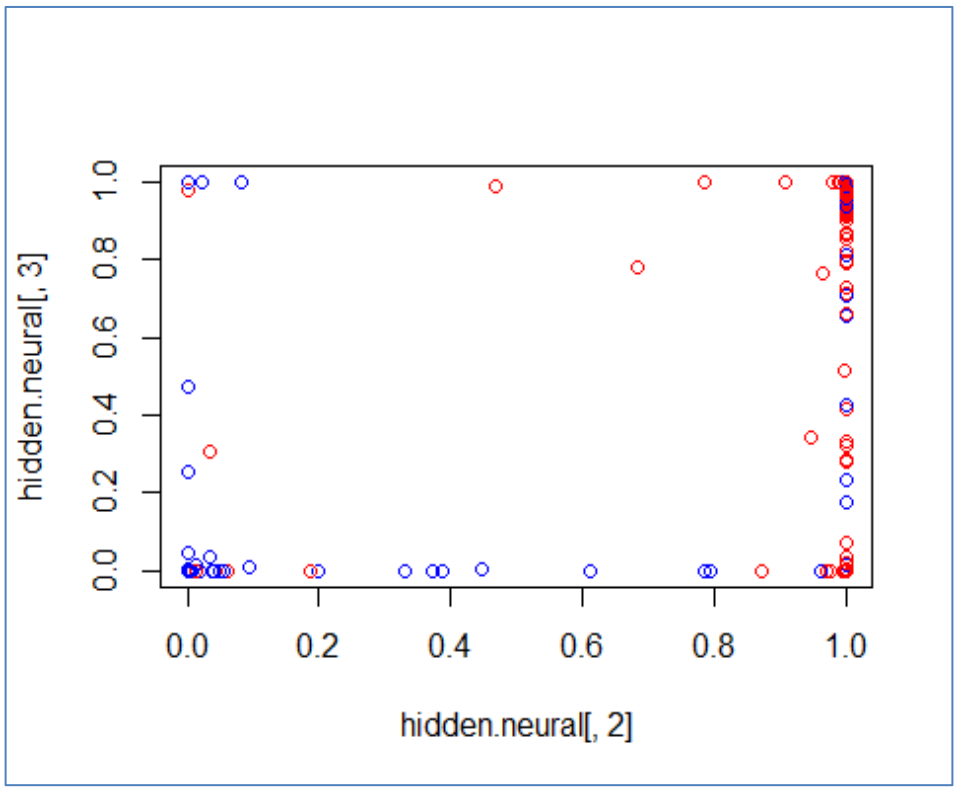
Avec “neuralnet”, il est possible d’accéder aux coordonnées des individus dans les espaces intermédiaires définis par les couches cachées du réseau. Pour les 15 premiers individus de l’échantillon d’apprentissage dans la couche intermédiaire par exemple :

```
#coordonnées intermédiaires
hidden.neural <- proba.pred.pm.neural$neurons[[2]]
print(head(round(hidden.neural,10),15))
```

```
##      [,1]         [,2]         [,3]
## 3602    1 0.0000000000 0.0000000000
## 3603    1 0.0000000000 0.0000000000
## 3604    1 1.0000000000 1.0000000000
## 3605    1 0.7839278922 0.0000000239
## 3606    1 0.0000000000 0.0000000000
## 3607    1 1.0000000000 1.0000000000
## 3608    1 0.0000000000 0.0000000000
## 3609    1 0.0000017225 0.0000000000
## 3610    1 1.0000000000 1.0000000000
## 3611    1 0.9999999984 0.9999973757
## 3612    1 1.0000000000 0.9999999933
## 3613    1 1.0000000000 1.0000000000
## 3614    1 1.0000000000 0.0000000000
## 3615    1 0.0000000000 0.0000000000
## 3616    1 0.0000000000 0.0000000000
```

Ainsi, dans le plan puisque nous avons 2 neurones dans la couche cachée, voici la disposition des individus (la première colonne correspond au biais, toujours égale à 1) :

```
#plot dans l'espace intermédiaire
plot(hidden.neural[,2],hidden.neural[,3],col=c("red","blue")[spam.test$spam])
```



Remarque : Il y a 3601 points dans le graphique, beaucoup d'entre eux ont des coordonnées identiques et sont superposés.

Est-ce que les observations sont linéairement séparables dans cet espace intermédiaire ?

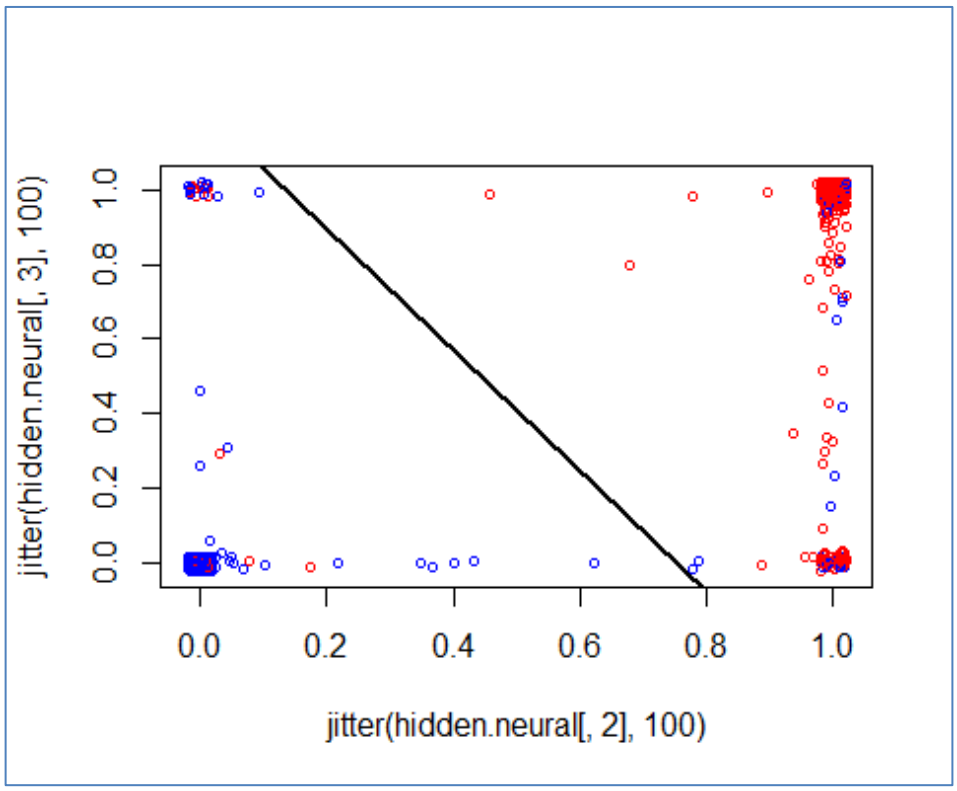
La frontière est exprimée par les poids synaptiques entre la couche cachée et la sortie (visibles dans la représentation graphique du réseau ci-dessus).

```
#poids synaptiques entre couche cachée et sortie
poids.out <- pm.neural$weights[[1]][[2]]
print(poids.out)
```

```
##               [,1]
## [1,]  1.2133075856
## [2,] -1.6079993276
## [3,] -0.9957967956
```

A partir de ces coefficients, traçons la droite de séparation dans l'espace de représentation intermédiaire défini par les neurones de la couche cachée.

```
#doite de séparation dans l'espace intermédiaire
plot(jitter(hidden.neural[,2],100),jitter(hidden.neural[,3],100),col=c("red","blue")[
spam.test$spam],cex=0.75)
abline(a=-poids.out[1,1]/poids.out[3,1],b=-poids.out[2,1]/poids.out[3,1],lwd=2)
```



jitter() rajoute du bruit aux valeurs pour contourner la superposition des points.

Les individus mal classés sont les rouges situés en deçà de la frontière (coin sud-ouest), et les bleus placés au-delà (coin nord-est).

Pédagogiquement, je trouve ce genre de fonctionnalités particulièrement intéressant.

3.3 Package "h2o"

"h2o" est une plate-forme JAVA qui implémente différents algorithmes de machine learning. Elle sait exploiter les capacités de parallélisation des machines (processeurs multicœurs par exemple). Elle propose des interfaces pour différents langages de programmation, dont R justement.

Après avoir installé la librairie, nous la chargeons puis nous démarrons la machine (littéralement) :

```
#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 - nthreads = -1, utiliser tous les cœurs disponibles
h2o.init(nthreads = -1)

##
## H2O is not running yet, starting it now...
##
## Note:  In case of errors look at the following log files:
##     C:\Users\Zatovo\AppData\Local\Temp\RtmpGQKasd/h2o_Zatovo_started_from_r.out
##     C:\Users\Zatovo\AppData\Local\Temp\RtmpGQKasd/h2o_Zatovo_started_from_r.err
##
##
## Starting H2O JVM and connecting:  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         1 seconds 786 milliseconds
##     H2O cluster timezone:       Europe/Paris
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.20.0.8
##     H2O cluster version age:    2 months and 24 days
##     H2O cluster name:           H2O_started_from_R_Zatovo_xgx086
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   1.76 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 3.5.1 (2018-07-02)
```

Nous typons les données d'apprentissage dans un format que "h2o" sait reconnaître.

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

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

print(head(h2oTrain))

##          wf_make    wf_address        wf_all          wf_3d        wf_our
## 1 -0.34213003619  0.2384893637  0.4697327993 -0.05243039477  0.3038085275
## 2 -0.34213003619 -0.1653047529 -0.5585634683 -0.05243039477 -0.4595939737
## 3 -0.34213003619 -0.1653047529  0.7465817944 -0.05243039477 -0.4595939737
## 4 -0.08563490054 -0.1653047529 -0.2421646167 -0.05243039477 -0.4595939737
## 5 -0.34213003619 -0.1653047529 -0.5585634683 -0.05243039477 -0.4595939737
## 6 -0.34213003619  0.1142450201 -0.5585634683 -0.05243039477 -0.4595939737
## Etc.
##   capital_run_length_longest capital_run_length_total spam
## 1              -0.16184598107           -0.26965281977  yes
## 2              -0.23261052856           -0.44928913112   no
## 3              -0.23732816506           -0.39796447074   no
## 4              -0.04862270509            0.64392613513   no
## 5              -0.25148107455           -0.48863803742   no
## 6              -0.08636379708           -0.04211349205  yes
```

Nous pouvons lancer le processus de modélisation.

```
#modélisation
pm.h2o <- h2o.deeplearning(y="spam",training_frame = h2oTrain,standardize =
FALSE,activation="Tanh",hidden=c(2),distribution="bernoulli",seed=100)

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

Outre les variables explicatives et cibles, nous indiquons ici : “standardize = false”, il n’est pas nécessaire de centrer et réduire les données ; “activation = tanh”, en l’absence de la fonction de transfert sigmoïde dans la librairie, nous utilisons la tangente hyperbolique ; “hidden = c(2)”, une seule couche cachée avec 2 neurones ; “distribution = ‘bernoulli’”, la variable cible est binaire, modélisée par la loi de bernoulli ; “seed = 100” pour rendre l’expérimentation reproductible.

Il n’a pas été nécessaire de recoder explicitement la variable cible ici.

Pour mesurer les performances, nous typons également l’ensemble de test et nous faisons appel à la fonction predict().

```
#prédiction
h2oTest <- as.h2o(spam.test.cr)

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

#prédiction
pred.pm.h20 <- h2o.predict(pm.h2o,newdata=h2oTest)

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

print(head(pred.pm.h20))

##   predict            no            yes
## 1     yes 0.01642821240 0.983571787598
## 2     yes 0.01659829930 0.983401700702
## 3      no 0.99820740156 0.001792598444
## 4     yes 0.12793368407 0.872066315934
## 5     yes 0.06472765218 0.935272347816
## 6      no 0.99775404237 0.002245957626
```

Nous disposons de 3 colonnes : les classes prédites et les probabilités d'affectation pour chaque modalité de la cible.

Confrontées aux classes observées, nous avons 77 observations mal classées.

```
#évaluation
evaluation.prediction(spam.test.cr$spam,as.matrix(pred.pm.h20)[,"predict"])

## [1] "Matrice de confusion"
##      ypred
## yobs   no yes
##   no  589  20
##   yes  57 334
## [1] "Taux d'erreur = 0.077"
## [1] "Rappel = 0.854"
## [1] "Precision = 0.944"
## [1] "F1-Score = 0.897"
```

L'affichage fournit toute une panoplie d'informations qui permet d'apprécier la qualité de la modélisation.

```
#affichage
print(pm.h2o)

## Model Details:
## ==============
##
```

```
## H2OBinomialModel: deeplearning
## Model ID:  DeepLearning_model_R_1544970131922_1
## Status of Neuron Layers: predicting spam, 2-class classification, bernoulli
distribution, CrossEntropy loss, 118 weights/biases, 9,3 KB, 36Ã‚Â 010 training
samples, mini-batch size 1
##   layer units    type dropout       l1       l2 mean_rate rate_rms
## 1     1    55   Input  0.00 %       NA       NA        NA       NA
## 2     2     2    Tanh  0.00 % 0.000000 0.000000  0.002397 0.002462
## 3     3     2 Softmax      NA 0.000000 0.000000  0.001694 0.000045
##   momentum mean_weight weight_rms mean_bias bias_rms
## 1       NA          NA         NA        NA       NA
## 2 0.000000   -0.021474   0.233855 -0.055330 0.055317
## 3 0.000000   -0.574447   1.399346 -0.085275 0.547883
##
##
## H2OBinomialMetrics: deeplearning
## ** Reported on training data. **
## ** Metrics reported on full training frame **
##
## MSE:  0.05488421429
## RMSE:  0.234273802
## LogLoss:  0.1987477548
## Mean Per-Class Error:  0.07096604915
## AUC:  0.972369227
## Gini:  0.9447384541
##
## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
##          no  yes    Error       Rate
## no     2072  107 0.049105  =107/2179
## yes     132 1290 0.092827  =132/1422
## Totals 2204 1397 0.066370  =239/3601
##
## Maximum Metrics: Maximum metrics at their respective thresholds
##                         metric threshold    value idx
## 1                       max f1  0.559881 0.915218 176
## 2                       max f2  0.120043 0.931020 298
## 3                 max f0point5  0.635780 0.922853 159
## 4                 max accuracy  0.599718 0.933907 169
## 5                max precision  0.984284 0.977679   1
## 6                   max recall  0.002594 1.000000 397
## 7              max specificity  0.984519 0.997705   0
## 8             max absolute_mcc  0.599718 0.861270 169
## 9   max min_per_class_accuracy  0.364069 0.930243 218
## 10 max mean_per_class_accuracy  0.370089 0.931337 216
##
## Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or
`h2o.gainsLift(<model>, valid=<T/F>, xval=<T/F>)`
```

Dans le dernier panneau (juste ci-dessus), “h2o” affiche les valeurs optimales pour chaque mesure (f1 [F1-Score], accuracy [Taux de reconnaissance], precision [précision], recall [rappel], etc.) lorsque l’on module le seuil d’affectation. Il est normalement de 0.5 dans un problème à 2 classes. Si nous le passons par exemple à 0.463675, nous optimisons le F1-Score à 0.916871 (sur l’échantillon d’apprentissage, cela s’entend).

“h2o” propose aussi une mesure de la contribution des variables dans le modèle (Remarque : le mélange de @ et $ pour accéder aux champs n’est pas très heureux je trouve).

```
#importance des variables - 15 premières variables
print(head(pm.h2o@model$variable_importances,15))
```

```
## Variable Importances:
##                      variable relative_importance scaled_importance
## 1  capital_run_length_average            1.000000          1.000000
## 2                   cf_dollar            0.636199          0.636199
## 3                       wf_hp            0.625378          0.625378
## 4                   cf_exclam            0.622213          0.622213
## 5                   wf_remove            0.617402          0.617402
## 6  capital_run_length_longest            0.517500          0.517500
## 7                      wf_415            0.516979          0.516979
## 8                      wf_edu            0.472986          0.472986
## 9                   wf_telnet            0.465623          0.465623
## 10                 wf_project            0.463678          0.463678
## 11                 wf_meeting            0.462832          0.462832
## 12                     wf_hpl            0.453451          0.453451
## 13                     wf_000            0.434035          0.434035
## 14                      wf_3d            0.373226          0.373226
## 15                    wf_free            0.366931          0.366931
##    percentage
## 1    0.065816
## 2    0.041872
## 3    0.041160
## 4    0.040952
## 5    0.040635
## 6    0.034060
## 7    0.034026
## 8    0.031130
## 9    0.030646
## 10   0.030518
## 11   0.030462
## 12   0.029844
## 13   0.028567
## 14   0.024564
## 15   0.024150
```

Nous avons vu plus haut que cette opération était relativement facile dans un perceptron simple (il suffit de comparer les coefficients en valeur absolue). Elle est en revanche autrement plus complexe dans un perceptron multicouche (parce qu’il y a un enchevêtrement de coefficients).

J’ai un peu cherché, je n’ai pas trouvé d’explications réellement convaincantes sur la manière de procéder du package. Bon, nous sommes plutôt dans une démarche de découverte tous azimuts dans ce tutoriel, je vais laisser l’étude de cette caractéristique à plus tard mais, clairement, il y a des choses à creuser de ce côté-là.

Enfin, une fois l'analyse terminée, il faut arrêter le moteur "h2o".

```
#arrêt
h2o.shutdown()
```

3.4 Package "RSNNS"

"RSNNS" est une surcouche logicielle qui permet d'accéder à une librairie SNSS, assez ancienne, consacrée aux réseaux de neurones.

Après avoir installé et chargé le package, nous devons préparer les données en plaçant dans une structure spécifique les descripteurs.

```
#chargement du package
library(RSNNS)

## Loading required package: Rcpp

#X pour l'apprentissage
X.Train <- as.matrix(spam.train.cr[-ncol(spam.train.cr)])
print(head(X.Train,2))

##         wf_make    wf_address        wf_all          wf_3d        wf_our
## 1 -0.3421300362  0.2384893637  0.4697327993 -0.05243039477  0.3038085275
## 2 -0.3421300362 -0.1653047529 -0.5585634683 -0.05243039477 -0.4595939737
##         wf_over     wf_remove   wf_internet      wf_order       wf_mail
## 1 -0.3444556529 -0.2825090499 -0.2559997164 -0.3287234813 -0.3689557221
## 2 -0.3444556529 -0.2825090499 -0.2559997164 -0.3287234813 -0.3689557221
##      wf_receive       wf_will     wf_people     wf_report  wf_addresses
## 1 -0.3027199547 -0.6352417573 -0.3073447781 -0.1860381771 -0.1937221694
## Etc.
##   capital_run_length_longest capital_run_length_total
## 1              -0.1618459811            -0.2696528198
## 2              -0.2326105286            -0.4492891311
```

Nous lançons la modélisation en utilisant la cible recodée (0.8, 0.2). Remarque : Ici aussi, le codage (1.0, 0) n'a pas fonctionné et m'a causé bien des problèmes. Mais, en cherchant une solution, j'ai découvert une curiosité du package sur laquelle je reviendrai plus bas.

Le seul paramètre que nous avons introduit est le nombre de neurones (size) dans l'unique couche cachée.

```
#apprentissage
set.seed(100)
pm.mlp <- mlp(x=X.Train,y=y.Train,size=c(2))
print(pm.mlp)

## Class: mlp->rsnns
## Number of inputs: 55
## Number of outputs: 1
## Maximal iterations: 100
## Initialization function: Randomize_Weights
```

```
## Initialization function parameters: -0.3 0.3
## Learning function: Std_Backpropagation
## Learning function parameters: 0.2 0
## Update function:Topological_Order
## Update function parameters: 0
## Patterns are shuffled internally: TRUE
## Compute error in every iteration: TRUE
## Architecture Parameters:
## $size
## [1] 2
##
## All members of model:
##  [1] "nInputs"               "maxit"
##  [3] "initFunc"              "initFuncParams"
##  [5] "learnFunc"             "learnFuncParams"
##  [7] "updateFunc"            "updateFuncParams"
##  [9] "shufflePatterns"       "computeIterativeError"
## [11] "snnsObject"            "archParams"
## [13] "IterativeFitError"     "fitted.values"
## [15] "nOutputs"
```

Nous isolons les descripteurs pour prédire sur l'échantillon test, nous faisons appel à predict(), puis nous convertissons les probabilités d'affectations en prédictions.

```
#préparation pour prédiction
X.Test <- as.matrix(spam.test.cr[-ncol(spam.test.cr)])
#print(head(X.Test))

#proba prédiction
proba.pred.mlp <- predict(pm.mlp,newdata = X.Test)
#summary(proba.pred.mlp)

#prédiction
pred.mlp <- ifelse(proba.pred.mlp>0.5,"yes","no")

#évaluation
evaluation.prediction(spam.test.cr$spam,pred.mlp)
```

```
## [1] "Matrice de confusion"
##      ypred
## yobs   no yes
##   no  576  33
##   yes  51 340
## [1] "Taux d'erreur = 0.084"
## [1] "Rappel = 0.87"
## [1] "Precision = 0.912"
## [1] "F1-Score = 0.89"
```

Les résultats tiennent la route par rapport aux autres outils.

Biais ou pas biais. Durant mes recherches, je me suis rendu compte d'une singularité du package. Nous disposons d'informations détaillées en faisant appel à la fonction summary().

```
#affichage détaillé - on n'a pas de biais ?
summary(pm.mlp)
```

```
## SNNS network definition file V1.4-3D
## generated at Sun Dec 16 15:22:21 2018
##
## network name : RSNNS_untitled
## source files :
## no. of units : 58
## no. of connections : 112
## no. of unit types : 0
## no. of site types : 0
##
##
## learning function : Std_Backpropagation
## update function   : Topological_Order
##
##
## unit default section :
##
## act      | bias     | st | subnet | layer | act func     | out func
## ---------|----------|----|--------|-------|--------------|-------------
##  0.00000 |  0.00000 | i  |      0 |     1 | Act_Logistic | Out_Identity
## ---------|----------|----|--------|-------|--------------|-------------
##
##
## unit definition section :
##
## no. | typeName | unitName                          | act      | bias     | st | position | act func     | out func |
sites
## ----|----------|-----------------------------------|----------|----------|----|----------|--------------|----------|----
---
##   1 |          | Input_wf_make                     | -0.34213 | -0.11534 | i  |  1, 0, 0 | Act_Identity |          |
##   2 |          | Input_wf_address                  | -0.16530 | -0.14540 | i  |  2, 0, 0 | Act_Identity |          |
##   3 |          | Input_wf_all                      | -0.55856 |  0.03139 | i  |  3, 0, 0 | Act_Identity |          |
##   4 |          | Input_wf_3d                       | -0.05243 | -0.26617 | i  |  4, 0, 0 | Act_Identity |          |
##   5 |          | Input_wf_our                      |  0.17168 | -0.01887 | i  |  5, 0, 0 | Act_Identity |          |
##   6 |          | Input_wf_over                     |  1.18359 | -0.00974 | i  |  6, 0, 0 | Act_Identity |          |
##   7 |          | Input_wf_remove                   |  0.79111 |  0.18744 | i  |  7, 0, 0 | Act_Identity |          |
##   8 |          | Input_wf_internet                 | -0.25600 | -0.07781 | i  |  8, 0, 0 | Act_Identity |          |
##   9 |          | Input_wf_order                    | -0.32872 |  0.02794 | i  |  9, 0, 0 | Act_Identity |          |
##  10 |          | Input_wf_mail                     |  0.25897 | -0.19784 | i  | 10, 0, 0 | Act_Identity |          |
##  11 |          | Input_wf_receive                  | -0.30272 |  0.07500 | i  | 11, 0, 0 | Act_Identity |          |
##  12 |          | Input_wf_will                     | -0.12367 |  0.22930 | i  | 12, 0, 0 | Act_Identity |          |
##  13 |          | Input_wf_people                   | -0.30734 | -0.13179 | i  | 13, 0, 0 | Act_Identity |          |
##  14 |          | Input_wf_report                   | -0.18604 | -0.06091 | i  | 14, 0, 0 | Act_Identity |          |
##  15 |          | Input_wf_addresses                | -0.19372 |  0.15753 | i  | 15, 0, 0 | Act_Identity |          |
##  16 |          | Input_wf_free                     | -0.31229 |  0.10141 | i  | 16, 0, 0 | Act_Identity |          |
##  17 |          | Input_wf_business                 | -0.32347 | -0.17723 | i  | 17, 0, 0 | Act_Identity |          |
##  18 |          | Input_wf_email                    | -0.33993 | -0.08549 | i  | 18, 0, 0 | Act_Identity |          |
##  19 |          | Input_wf_you                      | -0.44923 | -0.08431 | i  | 19, 0, 0 | Act_Identity |          |
##  20 |          | Input_wf_credit                   | -0.18724 |  0.11417 | i  | 20, 0, 0 | Act_Identity |          |
##  21 |          | Input_wf_your                     | -0.67822 |  0.02149 | i  | 21, 0, 0 | Act_Identity |          |
##  22 |          | Input_wf_font                     |  9.31150 |  0.12648 | i  | 22, 0, 0 | Act_Identity |          |
##  23 |          | Input_wf_000                      | -0.28564 |  0.02301 | i  | 23, 0, 0 | Act_Identity |          |
##  24 |          | Input_wf_money                    | -0.22666 |  0.14938 | i  | 24, 0, 0 | Act_Identity |          |
##  25 |          | Input_wf_hp                       | -0.32677 | -0.04794 | i  | 25, 0, 0 | Act_Identity |          |
##  26 |          | Input_wf_hpl                      | -0.29962 | -0.19715 | i  | 26, 0, 0 | Act_Identity |          |
##  27 |          | Input_wf_lab                      | -0.16573 |  0.16218 | i  | 27, 0, 0 | Act_Identity |          |
##  28 |          | Input_wf_labs                     | -0.22627 |  0.22917 | i  | 28, 0, 0 | Act_Identity |          |
##  29 |          | Input_wf_telnet                   | -0.17420 |  0.02946 | i  | 29, 0, 0 | Act_Identity |          |
##  30 |          | Input_wf_857                      | -0.14789 | -0.13337 | i  | 30, 0, 0 | Act_Identity |          |
##  31 |          | Input_wf_data                     | -0.17580 | -0.00702 | i  | 31, 0, 0 | Act_Identity |          |
##  32 |          | Input_wf_415                      | -0.15043 |  0.25710 | i  | 32, 0, 0 | Act_Identity |          |
##  33 |          | Input_wf_85                       | -0.19695 | -0.09078 | i  | 33, 0, 0 | Act_Identity |          |
##  34 |          | Input_wf_technology               | -0.24288 |  0.27249 | i  | 34, 0, 0 | Act_Identity |          |
##  35 |          | Input_wf_1999                     | -0.33481 |  0.11716 | i  | 35, 0, 0 | Act_Identity |          |
##  36 |          | Input_wf_parts                    | -0.06187 |  0.23367 | i  | 36, 0, 0 | Act_Identity |          |
##  37 |          | Input_wf_pm                       | -0.18753 | -0.19176 | i  | 37, 0, 0 | Act_Identity |          |
##  38 |          | Input_wf_direct                   | -0.18884 |  0.07763 | i  | 38, 0, 0 | Act_Identity |          |
##  39 |          | Input_wf_cs                       | -0.12082 |  0.29374 | i  | 39, 0, 0 | Act_Identity |          |
##  40 |          | Input_wf_meeting                  | -0.17153 | -0.22183 | i  | 40, 0, 0 | Act_Identity |          |
##  41 |          | Input_wf_original                 | -0.20606 | -0.10160 | i  | 41, 0, 0 | Act_Identity |          |
##  42 |          | Input_wf_project                  | -0.12845 |  0.21907 | i  | 42, 0, 0 | Act_Identity |          |
##  43 |          | Input_wf_re                       | -0.28703 |  0.16655 | i  | 43, 0, 0 | Act_Identity |          |
##  44 |          | Input_wf_edu                      | -0.20248 |  0.19638 | i  | 44, 0, 0 | Act_Identity |          |
##  45 |          | Input_wf_table                    | -0.07281 |  0.06199 | i  | 45, 0, 0 | Act_Identity |          |
##  46 |          | Input_wf_conference               | -0.11160 | -0.00526 | i  | 46, 0, 0 | Act_Identity |          |
```

```
##  47 |          | Input_cf_comma                   | -0.15667 |  0.16822 | i  | 47, 0, 0 | Act_Identity |          |
##  48 |          | Input_cf_bracket                 | -0.49533 |  0.23054 | i  | 48, 0, 0 | Act_Identity |          |
##  49 |          | Input_cf_sqbracket               | -0.15764 | -0.17537 | i  | 49, 0, 0 | Act_Identity |          |
##  50 |          | Input_cf_exclam                  | -0.24290 | -0.11575 | i  | 50, 0, 0 | Act_Identity |          |
##  51 |          | Input_cf_dollar                  | -0.31074 | -0.10168 | i  | 51, 0, 0 | Act_Identity |          |
##  52 |          | Input_cf_hash                    |  2.39662 | -0.18079 | i  | 52, 0, 0 | Act_Identity |          |
##  53 |          | Input_capital_run_length_average |  0.10646 | -0.15858 | i  | 53, 0, 0 | Act_Identity |          |
##  54 |          | Input_capital_run_length_longest |  0.16367 | -0.13507 | i  | 54, 0, 0 | Act_Identity |          |
##  55 |          | Input_capital_run_length_total   |  0.33085 |  0.05479 | i  | 55, 0, 0 | Act_Identity |          |
##  56 |          | Hidden_2_1                       |  0.00021 |  2.00741 | h  |  1, 2, 0 |||
##  57 |          | Hidden_2_2                       |  0.97066 | -0.23729 | h  |  2, 2, 0 |||
##  58 |          | Output_1                         |  0.77278 |  0.29809 | o  |  1, 4, 0 |||
## ----|----------|----------------------------------|----------|----------|----|----------|--------------|----------|----
---
##
##
## connection definition section :
##
## target | site | source:weight
## -------|------|-----------------------------------------------------------------------------------------------------------------
##     56 |      | 55:-1.04787, 54:-2.03461, 53:-1.29811, 52:-0.69092, 51:-3.10556, 50:-3.49402, 49: 0.42138, 48: 0.01495, 47: 0.32905,
##                46: 1.33705, 45: 0.00789, 44: 3.85676, 43: 0.75766, 42: 1.77026, 41: 0.52944, 40: 1.74247, 39: 1.18662, 38: 0.27329,
##                37: 0.14506, 36: 0.20232, 35: 1.87895, 34:-0.67197, 33: 1.79405, 32: 0.94707, 31: 0.98858, 30:-0.13934, 29: 1.51133,
##                28:-0.32605, 27: 0.53407, 26: 2.33463, 25: 4.11632, 24:-0.89454, 23:-2.49801, 22:-0.36593, 21:-0.06109, 20:-0.21508,
##                19: 0.25848, 18: 0.13283, 17:-1.20660, 16:-0.40009, 15:-0.19290, 14: 0.27062, 13: 0.05433, 12: 0.25448, 11: 0.29489,
##                10: 0.08150,  9:-0.34878,  8:-0.52367,  7:-2.40879,  6:-0.03050,  5:-1.24574,  4:-0.80143,  3: 0.28671,  2: 0.08419,
##                 1: 0.04292
##     57 |      | 55: 0.06410, 54: 1.76288, 53: 1.19018, 52:-0.02411, 51: 2.15478, 50: 0.00046, 49: 0.02881, 48:-0.33288, 47:-0.26465,
##                46:-0.56647, 45:-0.45726, 44:-1.63781, 43:-1.79442, 42:-1.02514, 41:-0.65121, 40:-1.30619, 39:-0.70654, 38:-0.24103,
##                37:-0.24786, 36:-0.03951, 35: 0.41844, 34: 0.57759, 33:-0.47105, 32:-0.34155, 31:-0.06313, 30:-0.28760, 29:-0.63860,
##                28: 0.05561, 27:-0.59594, 26:-0.92159, 25:-2.52057, 24: 1.18176, 23: 0.49160, 22: 0.24631, 21: 1.57929, 20: 0.55325,
##                19: 0.27216, 18: 0.75897, 17: 0.02216, 16: 2.36577, 15:-0.27745, 14: 1.55941, 13: 0.01035, 12:-0.53629, 11: 0.83207,
##                10: 0.57245,  9:-0.38960,  8: 0.04439,  7: 1.21908,  6: 0.57828,  5:-0.03143,  4: 0.56439,  3: 0.07199,  2:-0.35850,
##                 1: 0.17238
##     58 |      | 57: 0.95434, 56:-1.77850
## -------|------|-----------------------------------------------------------------------------------------------------------------
```

Il n'est fait mention du biais nulle part dans summary(). Il semble ne pas intervenir dans la couche d'entrée, ni dans la couche intermédiaire non plus si l'on se réfère aux informations ci-dessus. Ça paraît étrange.

Dans une régression, si les données sont centrées, la constante est nulle par nature. Dans un problème de classement en revanche, l'absence d'*intercept* impose une contrainte qui n'a pas lieu d'être dans la définition de la frontière de séparation des classes (l'hyperplan séparateur doit nécessairement passer par l'origine s'il n'y a pas d'intercept). La documentation du package n'est pas vraiment disserte à ce sujet. La question reste ouverte.

3.5 Package "deepnet"

"deepnet" est un package assez récent, avec un nombre de fonctions assez réduit (cf. la documentation). La simplicité n'est pas un défaut en soi, l'essentiel est que l'outil réalise efficacement les tâches demandées. Voyons ce qu'il en est.

Après avoir installé et chargé le package, nous pouvons lancer directement la modélisation. Sigmoïde est la fonction activation par défaut, ça nous convient.

```
#chargement
library(deepnet)

#apprentissage
set.seed(100)
pm.dpn <- nn.train(x=X.Train,y=y.Train,hidden=c(2),numepochs=150)
```

Nous pouvons effectuer la prédiction sur l'échantillon test.

```
#proba prediction
proba.pred.dpn <- nn.predict(pm.dpn,X.Test)
summary(proba.pred.dpn)

##        V1
##  Min.   :0.1637793
##  1st Qu.:0.1928587
##  Median :0.2740784
##  Mean   :0.4215172
##  3rd Qu.:0.7506433
##  Max.   :0.7929967

#prédiction
pred.dpn <- ifelse(proba.pred.dpn>0.5,"yes","no")

#evaluation
evaluation.prediction(spam.test.cr$spam,pred.dpn)

## [1] "Matrice de confusion"
##      ypred
## yobs   no yes
##   no  579  30
##   yes  52 339
## [1] "Taux d'erreur = 0.082"
## [1] "Rappel = 0.867"
## [1] "Precision = 0.919"
## [1] "F1-Score = 0.892"
```

Les résultats sont convenables par rapport aux autres packages.

Seul bémol peut-être, la frugalité est la rançon de la simplicité. En guise d'informations complémentaires, nous disposons un peu en vrac des poids synaptiques.

```
#poids synaptiques
print(pm.dpn$W)

## [[1]]
##             wf_make     wf_address        wf_all         wf_3d
## [1,] -0.08632657641  0.03398377898 0.08394511774 -0.3950965411
## [2,] -0.01483136455 -0.05404580733 0.02403849920  0.3176097854
##              wf_our        wf_over    wf_remove    wf_internet
## [1,] -0.75141587405 -0.01629425214 -1.228508366 -0.08409802055
## [2,]  0.03525837612  0.41752348410  1.252575685  0.55725040585
##           wf_order      wf_mail    wf_receive        wf_will
## [1,] -0.3781256948 0.0316463459 -0.7693486456 0.304217259952
## [2,] -0.1389034491 0.3757014968 -0.4237522347 0.009108313688
##           wf_people    wf_report   wf_addresses       wf_free
## [1,] -0.01670871995 0.1708478357 -0.04931010047 -0.9484834516
## [2,] -0.01965254524 0.6528970960  0.08947694764  0.5206835280
##        wf_business       wf_email        wf_you     wf_credit
## [1,] -0.3683680625 -0.19246563801 0.06599135252 -0.1542171819
## [2,]  0.2899911088 -0.02108810001 0.21679755723  0.1407450410
##            wf_your      wf_font        wf_000      wf_money        wf_hp
```

```
## [1,] -0.3033483740 -0.253144751 -1.2281793448 -0.5896270187  1.533234313
## [2,]  0.2040654974  0.235649635  0.6350972664  0.8692662346 -1.120606837
##             wf_hpl        wf_lab       wf_labs     wf_telnet        wf_857
## [1,]  0.5829662337  0.3845414582 0.01581154354  0.1867677602  0.1016556132
## [2,] -0.4266963333 -0.3658567559 0.17046482193 -0.3806999179 -0.1096946394
##            wf_data        wf_415         wf_85 wf_technology      wf_1999
## [1,]  0.6426847779  0.3938537150  0.4251961896 -0.2855776579 1.0508472065
## [2,] -0.1356582887 -0.1578013432 -0.2998180412  0.2772181877 0.1270456361
##           wf_parts          wf_pm     wf_direct         wf_cs
## [1,] 0.13851660205  0.24198396763  0.1971835039  0.4186423113
## [2,] 0.06155617417 -0.05196902408 -0.1285736689 -0.3349956967
##         wf_meeting   wf_original    wf_project         wf_re        wf_edu
## [1,]  0.7784244705  0.2816181515  0.6746735855  0.3915037295  1.3102357463
## [2,] -0.6545826677 -0.2632506871 -0.3476263863 -0.8888775819 -0.6400079537
##            wf_table wf_conference      cf_comma    cf_bracket
## [1,]  0.05973140994  0.4999376555  0.2384626261 0.09657588897
## [2,] -0.11683227571 -0.3314814185 -0.2179898906 0.03804837710
##       cf_sqbracket    cf_exclam     cf_dollar       cf_hash
## [1,]  0.0156379297 -1.833770133 -1.6248391857 -0.2983692598
## [2,] -0.2638840273  1.265495522  0.8879306007  0.2358889083
##      capital_run_length_average capital_run_length_longest
## [1,]              -0.4421727114              -0.8162415543
## [2,]               0.4721954902               0.9672593131
##      capital_run_length_total
## [1,]            -0.5804943087
## [2,]             0.2990519508
##
## [[2]]
##              [,1]        [,2]
## [1,] -1.810925676 1.162531833
```

[[1]] pour les connexions entre les couches d'entrées et cachées ; [[2]] entre l'intermédiaire et la sortie.

Les poids associés aux biais sont également disponibles.

```
#biais
print(pm.dpn$B)

## [[1]]
## [1]  0.22894649643 -0.03202503334
##
## [[2]]
## [1] 0.1805527056
```

3.6 Package "keras"

Le package "keras" est une librairie de deep learning très populaire auprès des data scientists Sondage KDnuggets, Mai 2018. Je l'ai moi-même étudié à plusieurs reprises, sous Python et sous R.

De fait, nous allons à l'essentiel dans cette section.

Après avoir installé et chargé le package, nous instancions le modèle :

```
#chargement
library(keras)

#instanciation du modele
pm.keras <- keras_model_sequential()
```

Puis, nous insérons les différentes couches, de l'entrée vers l'intermédiaire [(55 + 1) x 2 = 112 coefficients], puis de cette dernière vers la sortie [(2 + 1) = 3 coefficients].

```
#architecture
pm.keras %>%
  layer_dense(units=2, input_shape = ncol(X.Train), activation = "sigmoid") %>%
  layer_dense(units=1, activation = "sigmoid")
```

Nous affichons l'architecture du réseau pour vérification.

```
#affichage
print(summary(pm.keras))

## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## dense_1 (Dense)              (None, 2)                 112
## _________________________________________________________________________
## dense_2 (Dense)              (None, 1)                 3
## =========================================================================
## Total params: 115
## Trainable params: 115
## Non-trainable params: 0
## _________________________________________________________________________
## NULL
```

Il y a bien 115 coefficients dont les valeurs sont à estimer à partir des données.

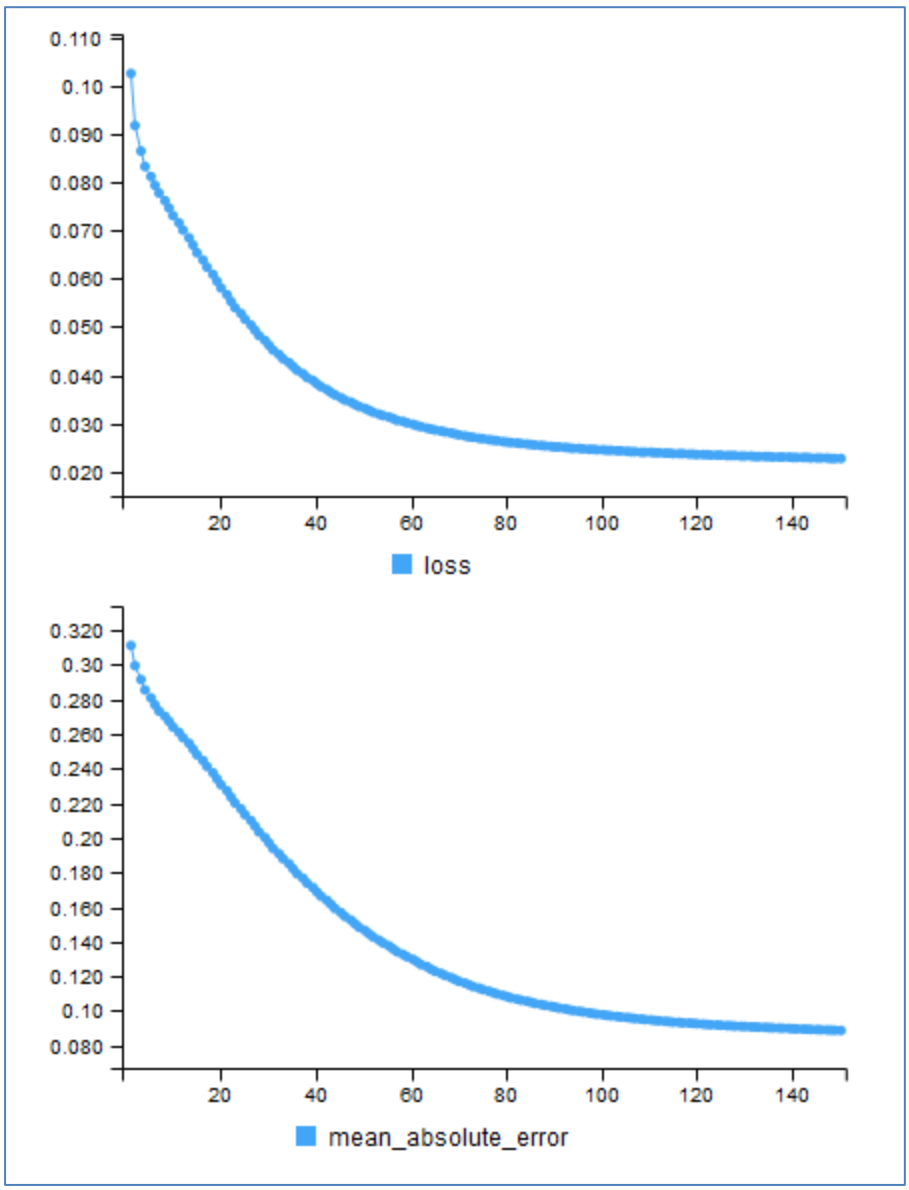
L'étape suivante consiste à paramétrer le processus d'apprentissage en indiquant la fonction de perte « loss » (erreur quadratique moyenne), l'algorithme d'optimisation « optimizer » (gradient stochastique) et la mesure utilisée pour le suivi « metrics » (erreur absolue moyenne).

```
#configuration de l'apprentissage
pm.keras %>% compile(
  loss="mean_squared_error",
  optimizer="sgd",
  metrics="mae"
)
```

Il ne reste plus qu'à lancer la modélisation à partir des données. Nous précisons le nombre d'itérations sur les données (epochs), et le nombre d'observations à traiter entre chaque correction des poids (batch_size) .

```
#lancement de l'apprentissage
pm.keras %>% fit(
  x=X.Train,
  y=y.Train,
  epochs=150,
  batch_size=15
)
```

« keras » produit automatiquement deux graphiques (loss et metrics) qui permettent de suivre l'évolution de l'optimisation sur l'échantillon d'apprentissage.



La fonction predict(), appliquée sur l'échantillon test, renvoie les probabilités d'appartenance aux classes.

```
#proba prediction
proba.pred.keras <- pm.keras %>% predict(
  x = X.Test
)


print(head(proba.pred.keras))

##              [,1]
## [1,] 0.8122746348
## [2,] 0.8181023002
## [3,] 0.1342533678
## [4,] 0.6128672957
## [5,] 0.5149767995
## [6,] 0.2154143304
```

Nous les comparons au seuil **0.5** pour obtenir les classes prédites et effectuer la confrontation avec les classes observées.

```
#prediction
pred.keras <- ifelse(proba.pred.keras[,1] > 0.5, "yes", "no")
print(table(pred.keras))

## pred.keras
##  no yes
## 649 351

#evaluation
evaluation.prediction(spam.test.cr$spam,pred.keras)

## [1] "Matrice de confusion"
##      ypred
## yobs   no yes
##   no  585  24
##   yes  64 327
## [1] "Taux d'erreur = 0.088"
## [1] "Rappel = 0.836"
## [1] "Precision = 0.932"
## [1] "F1-Score = 0.881"
```

“keras” peut également afficher les poids synaptiques.

```
#affichage des poids
get_weights(pm.keras)

## [[1]]
##                   [,1]           [,2]
##  [1,] -0.013538303785 -0.04181677476
##  [2,]  0.025444610044 -0.05606037006
##  [3,]  0.013664296828  0.09095556289
##  [4,] -0.216502502561  0.14132234454
## Etc.
## [53,] -0.050953228027 -0.03551277518
## [54,] -0.082753278315  0.48005545139
## [55,] -0.136340722442  0.27513608336
```

```
##
## [[2]]
## [1]  0.06606011838 -0.08484225720
##
## [[3]]
##              [,1]
## [1,] -1.485105038
## [2,]  1.939517856
##
## [[4]]
## [1] -0.3965438604
```

Bien d'autres choses encore sont possibles. Il faut prendre le temps de parcourir la documentation en ligne qui est plutôt fournie (Cf. Keras Documentation).

3.7 Package "mxnet"

"MXnet" est un package dédié au deep learning (Apache MXNet). Sa particularité est de pouvoir tirer parti : des groupes de machines via le calcul distribué, des machines multiprocesseurs, et même des processeurs graphiques (GPU) des ordinateurs. Il propose des interfaces pour plusieurs langages de programmation, y compris R.

Le package n'est pas disponible sur le dépôt officiel CRAN (Comprehensive Archive Network), il faut le chercher sur une adresse URL spécifique. Nous l'installons (à faire une seule fois).

```
#installation - non disponible sur le CRAN
# cran <- getOption("repos")
# cran["dmlc"] <- "https://apache-mxnet.s3-accelerate.dualstack.amazonaws.com/R/CRAN/"
# options(repos = cran)
# install.packages("mxnet")
```

Nous pouvons par la suite le charger normalement.

```
#chargement
library(mxnet)
```

En suivant un tutoriel où l'on traite la base Sonar. J'ai défini le réseau comme suit et lancé l'apprentissage.

```
#apprentissage -- echec
#mx.set.seed(0)
#pm.mxnet <- mx.mlp(X.Train,spam.train.cr[,"spam"],hidden_node = 2, out_node = 2,
activation = "sigmoid", out_activation = "softmax", num.round=100, array.batch.size =
15, optimizer="sgd", learning.rate = 0.07, array.layout="rowmajor", eval.metric =
mx.metric.accuracy)
```

Nous avons tour à tour : 2 neurones dans la couche cachée (hidden_node), deux neurones dans la couche de sortie (out_node) puisque la cible est à deux modalités, la fonction d'activation sigmoïde pour la couche interne (activation), softmax pour la couche de sortie

(out_activation), le nombre de passage sur la base (num_round), le nombre d'observations à traiter avant chaque mise à jour des poids (array.batch.size), avec un algorithme de gradient stochastique (optimizer), le taux d'apprentissage est de 0.07 (learning.rate). "array.layout" indique à la fonction que les lignes correspondent aux observations, les colonnes aux variables. L'avantage de cette écriture était que nous sommes en phase directe avec la configuration originale des données.

J'ai attendu, j'ai attendu, elle n'est jamais venue (la convergence). Après avoir trituré dans tous les sens les paramètres pour mieux guider l'apprentissage, j'ai de nouveau introduit la variable cible recodée en (0.8, 0.2). Ce qui a impliqué une adaptation des paramètres "out_node" et "out_activation".

```
#apprentissage -- ok
mx.set.seed(0)
pm.mxnet <- mx.mlp(X.Train,y.Train,hidden_node = 2, out_node = 1, activation =
"sigmoid", out_activation = "rmse", num.round=100, array.batch.size = 15,
optimizer="sgd", learning.rate = 0.07, array.layout="rowmajor", eval.metric =
mx.metric.mse)

## Start training with 1 devices

## [1] Train-mse=0.0815066247560177

## [2] Train-mse=0.0508446267098807

## [3] Train-mse=0.0368731230752411

## Etc.

## [98] Train-mse=0.0217710501114409

## [99] Train-mse=0.0217204150469409

## [100] Train-mse=0.0216698587004528
```

La prédiction et l'évaluation viennent facilement par la suite.

```
#proba prédiction
proba.pm.mxnet <- predict(pm.mxnet,X.Test,array.layout="rowmajor")
#print(proba.pm.mxnet[,1:6])

#prediction
pred.mxnet <- ifelse(proba.pm.mxnet[1,]>0.5,"yes","no")
#table(pred.mxnet)

#evaluation
evaluation.prediction(spam.test.cr$spam,pred.mxnet)

## [1] "Matrice de confusion"
##      ypred
## yobs   no yes
##   no  587  22
```

```
##   yes  60 331
## [1] "Taux d'erreur = 0.082"
## [1] "Rappel = 0.847"
## [1] "Precision = 0.938"
## [1] "F1-Score = 0.89"
```

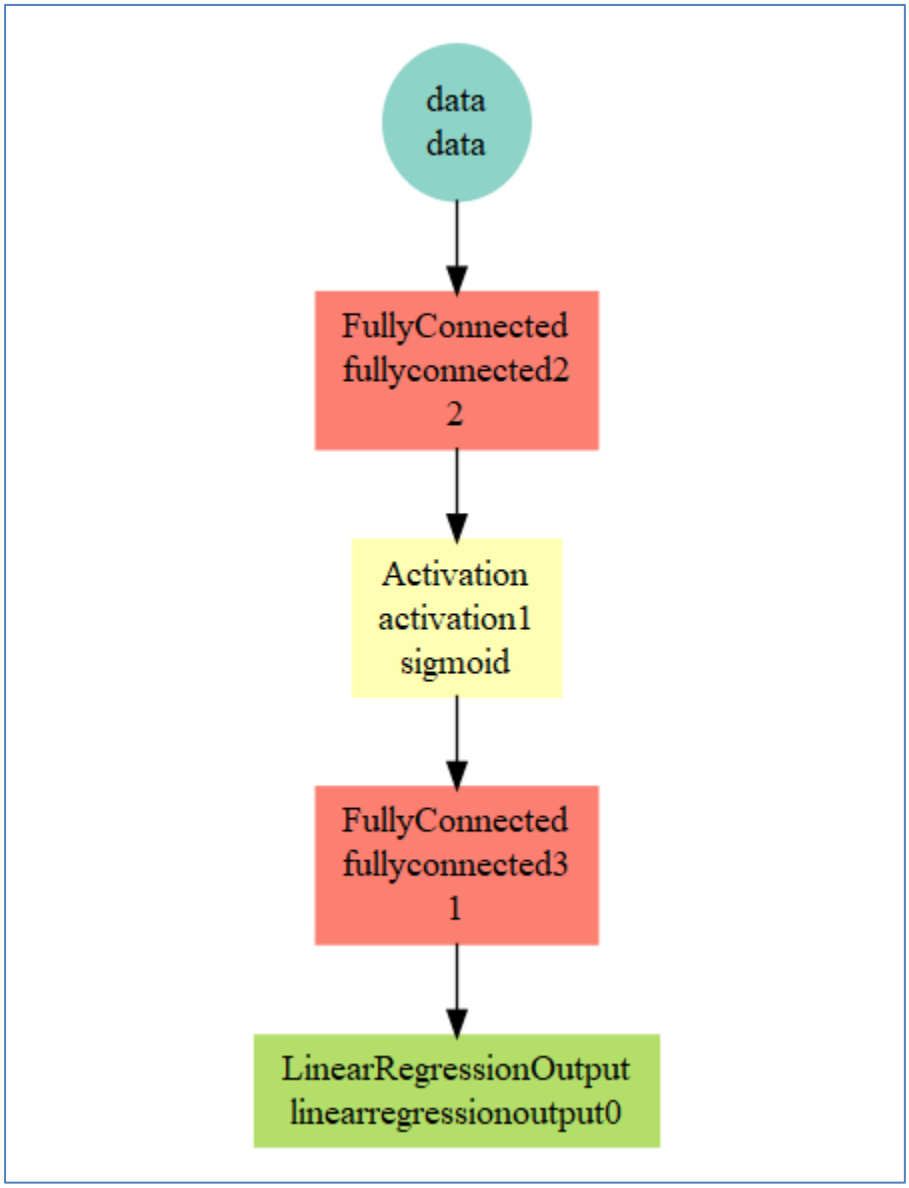
A l'instar des autres librairies, nous disposons des poids.

```
#affichage des poids
print(pm.mxnet)

## $symbol
## C++ object <000000000605f680> of class 'MXSymbol' <0000000015551b80>
##
## $arg.params
## $arg.params$fullyconnected0_weight
##                   [,1]            [,2]
##  [1,] -0.0748307704926 -0.072034664452
##  [2,] -0.0364462062716 -0.044067379087
##  [3,]  0.0354210585356  0.031127706170
##  [4,]  0.1556527912617  0.256239950657
## Etc.
## [52,]  0.1455200314522  0.239175334573
## [53,]  0.0609348081052  0.150194287300
## [54,]  0.2383943796158  0.478795856237
## [55,]  0.2091906964779  0.368462115526
##
## $arg.params$fullyconnected0_bias
## [1]  0.1090205237 -0.1348141283
##
## $arg.params$fullyconnected1_weight
##               [,1]
## [1,] -0.1375128329
## [2,]  0.7460420132
##
## $arg.params$fullyconnected1_bias
## [1] 0.1731384844
##
##
## $aux.params
## list()
##
## attr(,"class")
## [1] "MXFeedForwardModel"
```

Il est possible de représenter graphiquement l'architecture du modèle (de manière très simplifiée quand-même).

```
#affichage du réseau
graph.viz(pm.mxnet$symbol)
```



Bon, ce n'est pas des plus folichons, le graphique reste très basique. Mais l'intérêt est ailleurs pour la librairie « mxnet ». Il faudra tester ses capacités à exploiter au mieux des capacités calculatoires des machines (multiple CPU, GPU).

4 Bilan

Un petit tableau pour récapituler les performances des différents packages. Il n'a pas valeur de preuves bien sûr. Les résultats sont spécifiques à la base utilisée. Ils sont surtout dépendants des valeurs par défaut des paramètres, innombrables, que nous avons laissé inchangés. Je note simplement que pour un modèle relativement simple, un perceptron à 1 couche cachée avec 2 neurones, appliqué à une base qui ne présente pas de difficultés

insurmontables (spam dataset), les packages anciens tels que “nnet” tirent parfaitement leur épingle du jeu.

Package	Taux d'erreur	F1-Score
nnet (P. simple)	0.097	0.868
nnet	0.073	0.904
neuralnet	0.077	0.899
h2o	0.077	0.897
RSNNS	0.084	0.890
deepnet	0.082	0.892
keras	0.088	0.881
mxnet	0.082	0.890

5 Conclusion

Mon idée était de voir un peu le comportement des différents packages R dans l’implémentation d’un perceptron multicouche. Tous ont répondu positivement au cahier des charges, c’est la principale information.

Je retiens également de cette exploration des packages de *deep learning* que l’encodage de la variable cible joue un rôle très important pour la bonne tenue des algorithmes. A plusieurs reprises, l’utilisation des indicatrices 1/0 s’est avérée inefficace, faisant échouer plusieurs packages.

Enfin, on se rend qu’à l’usage, lorsque tout fonctionne tout de suite, les performances sont au rendez-vous pour autant que l’on dimensionne correctement son réseau. En revanche, lorsque nous commençons à rencontrer des problèmes dans le processus d’apprentissage, le nombre pléthoriques de paramètres à manipuler est plus un inconvénient qu’un avantage. Un utilisateur, même averti, ne peut qu’être perplexe face à cette profusion d’outils censés tous peser sur la convergence et sa rapidité.