PMC avec Comparaison

DIARRASSOUBA SAKARIA

18/03/2020

Contents

Introduction

Dans cette analyse, nous utiliserons les données d'une entreprise d'automobile qui vient de faire sortir un nouveau produit: voiture de luxe absolument pas chère. Pour faire le mrketing de cette voiture en masse, elle a mis une publicité de cette voiture sur les réseau sociaux, et les utilisation de ces réseaux sociaux, voyant la voiture, décident d'acheter la voiture "OUI" ou "NON". Cette entreprise contient des données de base comme l'identité du client, l'âge, le sexe, le salaire et la reponse. L'objetif de notre analyse est de comparaison des performances du PMC et d'autres méthodes statistiques

importation des données

```
cust=read.csv("Social_Network_Ads.csv",header = T)
donnes=data.frame(cust[1],cust[2],cust[3],cust[4],cust[5])
names(donnes)=c("ID","Sexe","Age","salaire","acheté")
head(donnes)
```

```
##         ID   Sexe Age salaire acheté
## 1 15624510   Male  19   19000      0
## 2 15810944   Male  35   20000      0
## 3 15668575 Female  26   43000      0
## 4 15603246 Female  27   57000      0
## 5 15804002   Male  19   76000      0
## 6 15728773   Male  27   58000      0
```

Dans la variable **acheté** on a:
** 0 : l'utisateur n'a pas acheté la voiture**
** 1: l'l'utisateur a acheté la voiture**
Chaque ligne représente un client, On n'aura pas besoin de la variable qualitative **"Age"**, car elle n'impacte pas la variable **"acheté"** et de meme la varaiable **"ID"** n'a pas de correlation avec la variable **"acheté"**. Donc suprimons ces variables

```
client=donnes[,c(3:5)]
head(client)
```

```
##   Age salaire acheté
## 1  19   19000      0
## 2  35   20000      0
## 3  26   43000      0
## 4  27   57000      0
## 5  19   76000      0
## 6  27   58000      0
```

On déccoupe le jeu de données client en 2 parties de manière aléatoire, la première partie servira à l'apprentissage du modèle et la seconde pour la validation du modèle:

```
set.seed(1234)
names(client)=c("X1","X2","X3")
apprent_idx <- sample(nrow(client), 7/8 * nrow(client))
client_apprent <- client[apprent_idx, ]
client_test <- client[-apprent_idx, ]
dim(client_apprent)
```

```
## [1] 350   3
```

```
head(client_apprent)
```

```
##     X1     X2 X3
## 46  23  20000  0
## 249 41  52000  0
## 243 50  88000  1
## 248 57 122000  1
## 341 53 104000  1
## 253 48 134000  1
```

Standartdisation des données

On applique la standardisation sur les colonnes **Age** et **salaire**des données d'apprentissages et de test

```
apprent_ech=scale(client_apprent[1:2],center = T,scale = T)
x3=data.frame(client_apprent$X3)
train=cbind(apprent_ech,x3) # on ajoute la colonne ciblé : acheté
```

```
test_ech=scale(client_test[1:2],center = T,scale = T)
x3=data.frame(client_test$X3)
test=cbind(test_ech,x3)
head(test)
```

```
##            X1         X2 client_test.X3
## 8  -0.6217934  2.1971170              1
## 19  0.8486640 -1.0805581              1
## 20  1.0587293 -1.0536920              1
## 22  0.9536966 -0.5163682              1
## 25  0.8486640 -1.2148891              1
## 55 -1.1469568 -0.2745725              0
```

notre variable cible ** acheté** est codée en 0/1. 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.

Selon moi, 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, là où les dérivées (gradients) sont quasi-nulles. Donc 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.

C'est ainsi que j'ai codé la variable cible en **0.8 pour 1** et ** 0.2 pour 0** lorsqu'on utilise la fonction de transfert sigmoïde. On situera ainsi dans la zone où sa pente reste importante.

```
y_train <- ifelse(train$client_apprent.X3=="1",0.8,0.2)
print(table(y_train))
```

```
## y_train
## 0.2 0.8
## 225 125
```

```
y_test=ifelse(test$client_test.X3=="1",0.8,0.2)
print(table(y_test))
```

```
## y_test
## 0.2 0.8
##  32  18
```

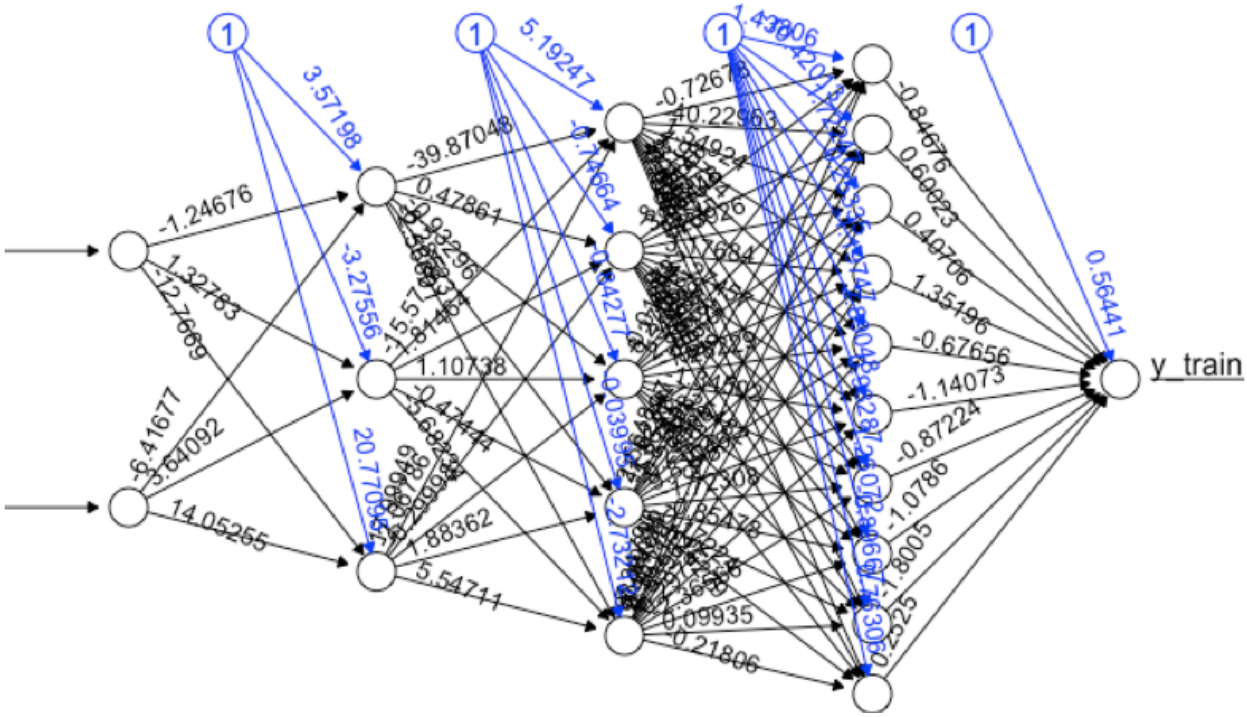
apprentissage du modele

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 = logistic introduit la fonction d'activation logistique dans le neurone de sortie.

```
# le parametre hidden =c(3,5,10) est optimale
modele <- neuralnet(y_train ~ X1 + X2, data= train,hidden = c(3,5,10), act.fct = "logistic")

plot(modele)
```

Le réseau a 3 couches cachés de 3 unités, ensuite 5 unités puis 10 unités et le biais est bien présent sur chaque



couche.

J' utilise 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_modele <- compute(modele,covariate=test[-ncol(test)])
print(proba_pred_modele$net.result[1:10])
```

```
##  [1] 0.7587357 0.6867082 0.7528550 0.6029571 0.7450535 0.2114316 0.2149082
##  [8] 0.2041430 0.2112504 0.2184869
```

Je les compare 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.

Au passage, une fonction pour donner en quelque sorte la matrice de confusion

```
#fonction pour evaluation des modèles
evaluation.prediction <- function(y_obs,y_pred){
 #matrice de confusion
 mc <- table(y_obs,y_pred)
 print("Matrice de confusion")
 print(mc)
 #taux d'erreur
 err <- 1-sum(diag(mc))/sum(mc)
 print(paste("Taux d'erreur =", round(err,3)))
 #precision
 precision <- sum(diag(mc))/sum(mc)
 print(paste("Precision =",round(precision,3)))
}
```

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_modele <- ifelse(proba_pred_modele$net.result > 0.5,"yes","no")

# evaluation
evaluation.prediction(test$client_test.X3,pred_modele)
```

```
## [1] "Matrice de confusion"
##      y_pred
## y_obs no yes
##     0 28   4
##     1  1  17
## [1] "Taux d'erreur = 0.1"
## [1] "Precision = 0.9"
```

Résumé, j ai 3 mal classés, avec un taux d'erreu de **6%**, et une précision de ** 94% **

D'autres classifieurs

méthode SVM

On utilise le package **''e1071"** pour l'implémentation des SVM. Nous demandons à la procédure svm() de construire un classifieur linéaire (kernel = 'linear')

```
#library(e1071)
modele_svm=svm(y_train ~ X1 + X2, data=train,  kernel="linear")
```

```
pred_svm=round(predict(modele_svm,test))
evaluation.prediction(test$client_test.X3,pred_svm)
```

```
## [1] "Matrice de confusion"
##      y_pred
## y_obs  0  1
##     0 30  2
##     1  6 12
## [1] "Taux d'erreur = 0.16"
## [1] "Precision = 0.84"
```

Résumé, j ai 8 mal classés, avec un taux d'erreu de **16%**, et une précision de ** 84% **

```
X_train <- as.matrix(train[-ncol(train)])
X_test <- as.matrix(test[-ncol(test)])
```

méthode deepnet

```
#library(deepnet)
#apprentissage
set.seed(100)

deep_modele <- nn.train(x=X_train,y=y_train,hidden=c(5),numepochs=250)
```

```
#proba prediction
proba.pred.dpn <- nn.predict(deep_modele,X_test)
summary(proba.pred.dpn)
```

```
##        V1
##  Min.   :0.1229
##  1st Qu.:0.1942
##  Median :0.3852
##  Mean   :0.4086
##  3rd Qu.:0.5388
##  Max.   :0.7999
```

```
#prédiction
pred.dpn <- ifelse(proba.pred.dpn>0.5,"yes","no")
#evaluation
evaluation.prediction(test$client_test.X3,pred.dpn)
```

```
## [1] "Matrice de confusion"
##      y_pred
## y_obs no yes
##     0 28   4
##     1  4  14
## [1] "Taux d'erreur = 0.16"
## [1] "Precision = 0.84"
```

Résumé, j ai 8 mal classés, avec un taux d'erreu de **16%**, et une précision de ** 84% **

Package	Taux erreur	prédiction
neuralnet	0.06	0.94
Deepnet	0.16	0.84
SVM	0.16	0.84

l'idée était de voir un peu le comportement des différentes méthodes de R dans l'implémentation d'un perceptron multicouche. Au vu des performences, il ressort que le neuralnet a une très bonne prediction