Implémentation du Naive Bayes sous R

Tutoriel Tanagra

1 Introduction

J'ai fait travailler mes étudiants sur la programmation sous R du classifieur bayésien naïf récemment. Pour l'avoir moi-même implémenté à différentes reprises, y compris pour Tanagra, je pensais avoir circonscrit en amont les difficultés qu'ils allaient rencontrer. C'est très important. Ils doivent produire en temps limité un package respectant les normes usuelles de R, sachant qu'ils ont d'autres évaluations par ailleurs. Il ne s'agit pas de les envoyer au casse-pipe.

Le modèle d'indépendance conditionnelle (naïve bayes) est une technique de classification supervisée. Schématiquement, elle est relativement simple à implémenter, surtout lorsque toutes les variables explicatives sont qualitatives. Il en est de même lorsqu'elles sont toutes quantitatives. Il y a en revanche des choix stratégiques à mettre en place lorsqu'elles sont mixtes. Je pensais que la principale difficulté viendrait de cette partie. Non en réalité. Je me suis rendu compte après coup que le principal écueil était de développer une implémentation performante du déploiement, lorsque nous appliquons le modèle sur des individus supplémentaires. Parce que R possède des caractéristiques qui lui sont propres, les temps d'exécution peuvent varier dans des proportions très importantes selon notre habilité à en tirer profit. Tout se joue sur notre capacité à définir une organisation à la fois des calculs et des structures qui permet à R d'exprimer pleinement son potentiel.

Dans ce tutoriel, nous nous penchons sur la programmation des fonctions **fit()** - construction du modèle sur un jeu de données - et **predict()** - application du modèle en prédiction sur un jeu de données - du Naive Bayes. La variable cible est forcément qualitative. Nous nous en tiendrons au cas des variables explicatives qualitatives. L'objectif n'est pas de proposer une resucée du naive bayes. Plusieurs packages disponibles sur le CRAN s'en chargent très bien. Il s'agit surtout pour nous d'étudier l'impact des choix d'implémentation sur les temps d'exécution sous R.

2 Naive Bayes

2.1 Présentation de la méthode

Nous reprenons dans cette section la présentation d'un précédent tutoriel qui nous servira de référence (REF, "Le classifieur bayésien naïf révisité", mars 2010).

En définitive, la méthode cherche à produire des fonctions de classement, une par modalité (y_k) de la variable cible. Elles s'écrivent (REF, section 2.1) :

$$d(y_k, X) = \ln P(Y = y_k) + \sum_{j=1}^{p} \ln P(X_j / Y = y_k)$$

La distribution a priori des classes est estimée par le rapport :

$$P(Y = y_k) = \frac{n_k + 1}{n + K}$$

Où n_k est le nombre d'observations de la modalité (y_k), n l'effectif total, K le nombre de modalités de la variable à prédire Y.

Les distributions conditionnelles des modalités des variables (X_j) conditionnellement aux classes s'écrivent :

$$P(X_j / Y = y_k) = \frac{n_{kl} + 1}{n_k + L_j}$$

Où L_j est le nombre de modalités de la variable (X_j), n_{kl} le nombre d'observations de la classe (y_k) associée à la modalité (ℓ) de la variable (X_j).

2.2 Un exemple

Prenons un exemple (REF, section 2.2) pour détailler ces calculs. Nous chargeons le jeu de données "**heart_for_naive_bayes.txt**". La variable cible est "disease" (presence ou absence d'une maladie cardiaque), les explicatives sont "exang" (binaire, {yes, no}) et "chest_pain" (4 modalités, {asympt, atyp_angina,nonèanginal,typ_angina}).

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

#chargement des données
heart <- read.table("heart_for_naive_bayes.txt", sep="\t", header = TRUE)
```

```
#affichage des caractéristiques de la base
str(heart)

## 'data.frame':    286 obs. of  3 variables:
##  $ exang     : Factor w/ 2 levels "no","yes": 2 2 2 2 2 2 2 2 2 2 ...
##  $ chest_pain: Factor w/ 4 levels "asympt","atyp_angina",..: 1 1 1 1 1 11 1 1 ...
##  $ disease   : Factor w/ 2 levels "negative","positive": 2 2 2 2 2 2 2 2 2 2 ...
```

Nous disposons de 286 observations.

2.2.1 Distribution a priori des classes

Calculons tout d'abord les distributions absolues des classes.

```
#distributions des classes
print(table(heart$disease))

##
## negative positive
##      182      104
```

Pour les distributions relatives, nous utilisons la fonction prop.table() non sans avoir introduit la constante de lissage (m = 1) (REF, page 2) :

```
#distributions relatives des classes
print(p_k <- prop.table(table(heart$disease)+1))

##
##  negative  positive
## 0.6354167 0.3645833
```

Le passage au logarithme ne pose aucun problème.

```
#passage au logarithme des distributions relatives des classes
print(log(prop.table(table(heart$disease)+1)))

##
##   negative   positive
## -0.4534743 -1.0090001
```

2.2.2 Distributions conditionnelles

Voyons ce qu'il en est des distributions conditionnelles (REF, page 4). Pour "exang" :

```
#distribution de exang, conditionnellement à disease
print(p_exang <- prop.table(table(heart$disease,heart$exang)+1,margin=1))

##
##                    no       yes
##   negative 0.8913043 0.1086957
##   positive 0.3490566 0.6509434
```

Nous introduisons l'option "margin = 1" parce que nous calculons des profils lignes.

Le schéma est le même pour "chest_pain" :

```
#distribution de chest_pain, conditionnellement à disease
print(p_chest <- prop.table(table(heart$disease,heart$chest)+1,margin=1))

##
##                asympt atyp_angina non_anginal typ_angina
##   negative 0.21505376  0.51612903  0.22580645 0.04301075
##   positive 0.75925926  0.08333333  0.11111111 0.04629630
```

Dans les deux cas, le passage au logarithme ne pose aucun problème. Grâce au paramètre de lissage (m = 1), l'écueil d'un logarithme sur une valeur nulle ne peut pas survenir.

2.2.3 Classement d'un individu supplémentaire

Pour le classement, nous identifions les modalités prises par l'individu à classer et nous additionnons les logarithmes des probabilités correspondantes (REF, section 2.2.2).

Par exemple, pour un individu avec les caractéristiques (exang = yes, chest_pain = asympt). Nous calculons le score suivant pour la modalité (disease = positive) :

```
#score d'affectation à disease = positive
s_positive <- log(p_k["positive"]) + log(p_exang["positive","yes"]) +
log(p_chest["positive","asympt"])
print(s_positive)

##  positive
## -1.713745
```

Nous faisons de même pour (disease = absence) :

```
#score d'affectation à disease = negative
s_negative <- log(p_k["negative"]) + log(p_exang["negative","yes"]) +
log(p_chest["negative","asympt"])
print(s_negative)

##  negative
## -4.209545
```

La classe prédite correspond à celle qui maximise le score d'affectation. Soit :

```
#classe d'affectation
prediction <- levels(heart$disease)[which.max(c(s_negative,s_positive))]
print(prediction)

## [1] "positive"
```

Le principe est simple, nous le constatons. Il va falloir mettre en musique tout cela dans des fonctions dédiées pour pouvoir appliquer les phases d'apprentissage (**fit**) et de prédictions (**predict**) sur des jeux de données plus conséquents.

3 Implémentation 1 - Structures de listes

Notre première implémentation s'appuie sur les structures de listes de R. Elles sont souples et permettent de transmettre des informations de nature différente. Elles sont essentielles dans la programmation des objets S3 sous R (voir "Mécanismes des classes sous R", novembre 2017).

3.1 Apprentissage (fit)

Notre fonction *myfit_1(formule, data)* prend en entrée une formule au standard de R (*variable cible ~ liste des variables explicatives*) et un data frame. Elle extrait les données et calcule la collection des tableaux des probabilités. Nous passons directement aux logarithmes pour que la prédiction, par simple addition des valeurs, soit plus rapide par la suite.

Nous collectons également un ensemble d'informations importantes pour la prédiction : le nombre d'explicatives (*$p*), les noms des variables correspondantes (*$var_names*), les modalités de la variable cible (*$modalites*).

```
#pour l'apprentissage
myfit_1 <- function(formule,data){
  #vérification de la formule
  if (!inherits(formule,"formula")) {formule <- as.formula(formule)}
  #data.frame correspondant à la formule
  D <- model.frame(formule,data)
  #y - variable expliquée
  y <- D[,1]
  #X - variables explicatives
  allX <- D[,-1]
  #calcul des tableaux croisés (log des probas conditionnelles)
  tc <- lapply(allX,function(x,y){return(log(prop.table(table(y,x)+1,1)))},y)
  #objet liste pour recenser les informations
  objet <- list()
  #liste des modalités
  objet$modalites <- levels(y)
  #nombre de modalités
  objet$nb_modalites <- nlevels(y)
  #log des probas a priori
  objet$prior <- as.numeric(log(prop.table(table(y)+1)))
  #nombre de variables explicatives
  objet$p <- ncol(allX)
  #noms de ces variables
  objet$var_names <- colnames(allX)
  #tableaux croisés
  objet$tc <- tc
  #return
  return(objet)
}
```

Appliquons-la sur les données "heart" :

```
#apprentissage
modele_heart <- myfit_1(disease ~ ., data = heart)
print(modele_heart)

## $modalites
## [1] "negative" "positive"
##
## $nb_modalites
## [1] 2
##
## $prior
## [1] -0.4534743 -1.0090001
##
## $p
## [1] 2
##
## $var_names
## [1] "exang"      "chest_pain"
##
## $tc
## $tc$exang
##           x
## y                   no        yes
##   negative -0.1150693 -2.2192035
##   positive -1.0525212 -0.4293326
##
## $tc$chest_pain
##           x
## y              asympt atyp_angina non_anginal typ_angina
##   negative -1.5368672  -0.6613985  -1.4880771 -3.1463051
##   positive -0.2754120  -2.4849066  -2.1972246 -3.0726933
```

3.2 Prédiction 1_A - Boucles for

Pour le déploiement sur un data frame composé d'un ensemble d'observations, après les vérifications d'usage (présence effective des variables explicatives), nous appliquons la règle d'affectation détaillée ci-dessus (section 2.2.3) pour classer chaque individu. La fonction renvoie un vecteur de prédictions.

```
#prediction avec utilisation des indices
mypredict_1_A <- function(model,newdata){
  #vérifier les noms de variables
 if (length(intersect(model$var_names,colnames(newdata))) < model$p){stop("souci de variables")}
  #récupérer les variables, et dans le bon ordre
  D <- newdata[model$var_names]
  #transformer les factors en code (indice de modalité)
  D <- data.frame(lapply(D,function(x){return(unclass(x))}))
  #matrice des scores - calcul par addition des log
  scores <- matrix(0,nrow=nrow(newdata),ncol=model$nb_modalites)
  #triple boucles imbriquées (indicées)
  #pour chaque ligne (i) newdata, , (j) les colonnes (variables)
  for (i in 1:nrow(newdata)){
    #pour chaque modalité (k) de la variable cible
```

```
    for (k in 1:model$nb_modalites){
      #pour chaque variable (j) explicative
      for (j in 1:model$p){
        scores[i,k] <- scores[i,k] + model$tc[[j]][k,D[i,j]]
      }
    }
  }
  #il faut rajouter les prior (logarithmes des probas a priori)
  for (k in 1:model$nb_modalites){scores[,k] <- scores[,k]+model$prior[k]}
  #return(scores)
  #prediction
  mPred <- model$modalites[apply(scores,1,which.max)]
  return(mPred)
}
```

Nous observons plusieurs opérations clés dans cette fonction :

- La fonction prend entrée l'objet **model** fourni par *fit()* et un ensemble de données **newdata** à classer.
- La matrice des données **newdata** est transformée en matrice de codes ***D*** pour accélérer la recherche des modalités lors du calcul des scores d'affectation. Manipuler des entiers est toujours plus rapide que d'accéder aux modalités des *factors* de R. Attention cependant, ce raccourci n'est valable que si les listes des modalités par variable sont identiques dans les données passées dans *fit()* et *predict()*.
- La matrice de scores possède **nrow(newdata)** lignes et **model$nb_Modalites** colonnes. Elle contient les scores calculés par addition des logarithmes des probabilités conditionnelles pour chaque observation.
- Il ne faut pas oublier d'ajouter aux scores ainsi calculés les logarithmes de la distribution a priori des classes.
- La prédiction est attribuée par identification des scores maximum pour chaque individu.

Appliquons cette fonction sur les mêmes données "heart".

```
#prédiction en resubstitution
print(system.time(pred_1A <- mypredict_1_A(modele_heart,newdata=heart)))
```

```
##    user  system elapsed
##    0.04    0.00    0.05
```

La fonction semble rapide. Ne crions pas trop victoire quand-même, la base ne comporte que 286 observations et 2 variables explicatives.

Nous calculons la matrice de confusion…

```
#matrice de confusion
mc <- table(heart$disease,pred_1A)
print(mc)

##           pred_1A
##            negative positive
##   negative      167       15
##   positive       41       63
```

… et le taux de reconnaissance en resubstitution.

```
#taux de succès
success <- sum(diag(mc))/sum(mc)
print(success)

## [1] 0.8041958
```

3.3 Prédiction 1_B - Boucles sapply

Depuis le temps que j'enseigne la programmation R, j'ai toujours dit à mes étudiants de proscrire les boucles **for** dont les performances sont désastreuses (Remarque : c'est un peu moins vrai pour les versions récentes de R – depuis la 3.4 pour être précis. Voir "Performances des boucles sous R, octobre 2019). Nous en avons 3 imbriquées dans la fonction *mypredict_1_A()*. On peut imaginer d'emblée que les temps d'exécution seront désastreux sur les grandes bases.

Une astuce simple pour gagner en rapidité consiste (consistait, avec R 3.4 donc) à les émuler avec les fonctions **sapply** ou **lapply**. Réécrivons la fonction *predict()* à la lumière de cette perspective.

```
#prediction avec utilisation des sapply
mypredict_1_B <- function(model,newdata){
  #vérifier les noms de variables
  if (length(intersect(model$var_names,colnames(newdata))) < model$p){stop("souci de
variables")}
  #récupérer les variables, et dans le bon ordre
  D <- newdata[model$var_names]
  #transformer les factors en code (indice de modalité)
  D <- data.frame(lapply(D,function(x){return(unclass(x))}))


  #matrice des scores - calcul par addition des log
  #triple boucles imbriquées (indicées) : (i) les lignes de newdata,
  #(k) les modalités de la classe, (j) les colonnes (variables)
  #scores possède NB_Modalites lignes et NB_Examples(newdata) colonnes
  scores <-
sapply(1:nrow(newdata),function(i,model,D){colSums(sapply(1:model$nb_modalites,functi
on(k){sapply(1:model$p,function(j){model$tc[[j]][k,D[i,j]]})}))},model,D)


  #il faut rajouter les prior
  for (k in 1:model$nb_modalites){scores[k,] <- scores[k,]+model$prior[k]}
  #return(scores)
```

```
  #prediction
  mPred <- model$modalites[apply(scores,2,which.max)]
  return(mPred)
}
```

La ligne de code associée au calcul des scores semble démoniaque à souhait. Mais il ne s'agit que d'une réexpression sous forme de **sapply()** de la triple boucle sur les individus (i), les modalités de la variable cible (k) et les variables explicatives (j).

Voyons son comportement sur notre jeu de données.

```
#prédiction
print(system.time(pred_1B <- mypredict_1_B(modele_heart,newdata=heart)))

##    user  system elapsed
##    0.07    0.00    0.06
```

Nous verrons plus tard ce qu'il en sera des performances. Assurons-nous déjà que le vecteur de prédictions ainsi calculé est cohérent avec le précédent.

```
#vérifications - croisement des prédictions
print(table(pred_1A,pred_1B))

##           pred_1B
## pred_1A    negative positive
##   negative      208        0
##   positive        0       78
```

Oui ! C'est toujours rassurant.

3.4 Prédiction 1_C - Utilisation de mapply

Nous le verrons dans la section suivante, ces deux solutions assez intuitives au demeurant s'avèreront désastreuses en termes de rapidité d'exécution. Notamment parce que nous multiplions les accès indicés sur les structures, et R n'aime pas ça.

Pour dépasser cet inconvénient, j'ai écrit une autre version de *predict()* en essayant de minimiser cet aspect du programme en passant par une combinaison de **apply** et **mapply**. Structurellement, la fonction *mypredict_1_c()* est identique, seule la ligne de code dévolue au calcul des scores est modifiée. Nous ne manipulons plus d'indices.

```
#prediction - sans utiliser des indices
mypredict_1_C <- function(model,newdata){
  #vérifier les noms de variables
  if (length(intersect(model$var_names,colnames(newdata))) < model$p){stop("souci de variables")}
  #récupérer les variables, et dans le bon ordre
  D <- newdata[model$var_names]
```

```
  #transformer les factors en code (indice de modalité)
  D <- as.matrix(data.frame(lapply(D,function(x){return(unclass(x))})))


  #matrice des scores (addition en log)
  #//!\\ utilisation de apply et mapply à la place des boucles
  scores <-
apply(D,1,function(ligne,tc){rowSums(mapply(function(value,tableau){return(tableau[,v
alue])},ligne,tc))},model$tc)


  #rajouter les prior
 scores <- apply(scores,2,function(ligne,prior){return(mapply(sum,ligne,prior))},prior=model$prior)
  #return(scores)
  #prediction
  mPred <- model$modalites[apply(scores,2,which.max)]
  return(mPred)
}
```

Voyons son comportement sur notre base "heart".

```
#prédiction
print(system.time(pred_1C <- mypredict_1_C(modele_heart,newdata=heart)))

##    user  system elapsed
##    0.03    0.00    0.03
```

La prédiction est cohérente avec les deux autres.

```
#vérifications - croisement des prédictions
print(table(pred_1A,pred_1C))

##           pred_1C
## pred_1A    negative positive
##   negative      208        0
##   positive        0       78
```

3.5 Etude des performances

Sur une petite base, les différentes implémentations se valent. Pour mieux cristalliser les écarts, étudions leurs comportements sur des bases plus conséquentes, larges ("**large.txt**", 202 observations et 7143 variables explicatives, toutes binaires) et longues ("**long.txt**", 10000 observations et 19 variables, certaines à plus de 2 modalités).

3.5.1 Base "large"

Nous chargeons le jeu de données et nous en affichons les caractéristiques.

```
#chargement
large <- read.table("large.txt",sep="\t",header=TRUE)
print(dim(large))
```

```
## [1]  202 7144
```

Nous construisons le modèle prédictif.

```
#apprentissage
print(system.time(m_large <- myfit_1(CLASSE ~ ., data = large)))

##    user  system elapsed
##    3.68    0.23    3.92
```

Nous le constatons, le temps de calcul n'est pas vraiment un enjeu ici. La construction des tables de probabilités avec la fonction **table()** de R est particulièrement rapide.

Voyons ce qu'il en est de la première version de la prédiction, celle s'appuyant sur des boucles indicées.

```
#prediction - première version (boucles for)
print(system.time(mypredict_1_A(m_large,newdata = large)))

##    user   system   elapsed
##    207.8   42.43   250.55
```

Argh ! Plus de 4 minutes ! Même si la base est quelque peu conséquente (202 observations seulement, mais 7143 variables à scanner pour chaque classement), elle n'est pas exceptionnelle (on rencontre souvent ce type de configuration en text mining par exemple) : le temps de calcul est pathétique. C'est le moins qu'on puisse dire. La répétition (nombre d'individus x nombre de modalités de la cible) de la boucle **for** sur les variables s'avère tragique.

Passons à la solution **sapply** pour voir ce qu'il en est.

```
#prediction - seconde version (boucles sapply)
print(system.time(mypredict_1_B(m_large,newdata = large)))

##    user   system   elapsed
##    215.87  5.89   222.30
```

Censé être plus rapide, le passage aux sapply pour émuler les boucles n'est plus aussi déterminant pour les versions post 3.4 de R. Nous en avons la preuve encore une fois (voir "Performances des boucles sous R, octobre 2019).

Les accès indicés constituent le véritable goulot d'étranglement de la fonction en réalité. L'objectif de la troisième version est justement de les réduire. Voyons si vraiment nous gagnons en rapidité de calcul.

```
#prediction - troisième version (utilisation de apply + mapply)
print(system.time(mypredict_1_C(m_large,newdata = large)))
```

```
##    user   system   elapsed
##    15.22  0.00     15.26
```

Plus concluant que cela, c'est difficile ! Avec exactement les mêmes structures de données, la procédure est 14 fois plus rapide pour le même résultat en prédiction. Clairement, la bonne connaissance des fonctionnalités de R (il fallait connaître et savoir utiliser à bon escient les fonctions **apply** et **mapply**) a été déterminante pour gagner en performances.

3.5.2 Base "longue"

Inversons les caractéristiques de nos données. La base "**long.txt**" est plus longue (10000 observations) que large (19 explicatives). Nous la chargeons :

```
#chargement
long <- read.table("long.txt",sep="\t",header=TRUE)
print(dim(long))

## [1] 10000    20
```

De nouveau, l'apprentissage, ultra-rapide, n'est pas un enjeu.

```
#apprentissage
print(system.time(m_long <- myfit_1(ONDE ~ ., data = long)))

##    user  system elapsed
##    0.02    0.00    0.02
```

Voyons les différentes déclinaisons de la prédiction. Celle basée sur les boucles imbriquées…

```
#prediction - première version (boucles for)
print(system.time(mypredict_1_A(m_long,newdata = long)))

##    user   system   elapsed
##    10.91  0.00     10.91
```

… est moins pénalisée par la "longueur" de la base.

Pour les boucles émulées avec sapply()…

```
#prediction - seconde version (boucles sapply)
print(system.time(mypredict_1_B(m_long,newdata = long)))

##    user   system   elapsed
##    12.0    0.00     12.03
```

… nous avons le même type de comportement.

Enfin, en utilisant la combinaison de **apply** et **mapply**…

```
#prediction - troisième version (utilisation de apply + mapply)
print(system.time(mypredict_1_C(m_long,newdata = long)))
```

```
##    user   system   elapsed
##    1.81    0.00      1.81
```

… il y a toujours un gap, mais il est moins important (6 fois plus rapide à peu près).

Manifestement, et c'est la principale conclusion de cette première partie, éviter de passer par les accès indicés est bénéfique pour les performances sous R.

4 Implémentation 2 - Structures matricielles

Malgré tout il me restait une certaine insatisfaction à ce stade. En effet, il est de notoriété publique que R excelle dans le calcul matriciel. Je n'exploite pas du tout cette caractéristique dans mes implémentations ci-dessus. Pouvoir exprimer la prédiction à l'aide d'un calcul matriciel implique un choix de structures internes différent et par conséquent une modification de la fonction **fit()**.

4.1 Reprogrammation de fit()

Plutôt qu'une liste de tableaux, nous collectons les logarithmes des probabilités conditionnelles dans une matrice (**$MLog**). Le stockage de la liste des tableaux croisés n'est plus nécessaire.

```
#pour l'apprentissage
myfit_2 <- function(formule,data){
  #vérification
  if (!inherits(formule,"formula")) {formule <- as.formula(formule)}
  #data.frame
  D <- model.frame(formule,data)
  #y vecteur de la variable cible
  y <- D[,1]
  #X matrice des explicatives
  allX <- D[,-1]
  #calcul des tableaux croisés
  tc <- lapply(allX,function(x,y){return(log(prop.table(table(y,x)+1,1)))},y)
  #nombre de modalités des X
  nbModaX <- sapply(allX,function(x){return(nlevels(x))})
  #//!\\conversion en matrice des log de proba
  K <- nlevels(y)
  MLog <- matrix(0,nrow=sum(nbModaX),ncol=K)
  numBloc <- 0
  #pour chaque tableau croisé
  for (i in 1:length(tc)){
    temp <- tc[[i]]
    #pour chaque modalité de Y
    for (j in 1:K){
      MLog[(numBloc+1):(numBloc+nbModaX[i]),j] <- temp[j,]
    }
    numBloc <- numBloc + nbModaX[i]
  }
  #objet
  objet <- list()
```

```
  #liste des modalités
  objet$modalites <- levels(y)
  #nombre de modalités
  objet$nb_modalites <- K
  #log des probas a priori
  objet$prior <- as.numeric(log(prop.table(table(y)+1)))
  #nombre de variables explicatives
  objet$p <- ncol(allX)
  #noms de ces variables
  objet$var_names <- colnames(allX)
  #//!\\ tableaux croisés - pas nécessaire
  #objet$tc <- tc
  #nombre total de modalités des X
  objet$NbAllValues <- sum(nbModaX)
  #tableau de décalage des modalités
  objet$DecValues <- cumsum(c(0,nbModaX[-length(nbModaX)]))
  #matrice pour la projection
  objet$MLog <- MLog
  #return
  return(objet)
}
```

Comme nous pouvons le constater, il n'y a aucun recalcul. Il faut veiller à la cohérence des positions pour pouvoir appliquer correctement les coefficients en prédiction. Le vecteur **$DecValues** sert à cela.

Appliquons la fonction sur notre base jouet "heart".

```
#apprentissage heart - 2è version
modele_heart2 <- myfit_2(disease ~ ., data = heart)
print(modele_heart2)
```

```
## $modalites
## [1] "negative" "positive"
##
## $nb_modalites
## [1] 2
##
## $prior
## [1] -0.4534743 -1.0090001
##
## $p
## [1] 2
##
## $var_names
## [1] "exang"      "chest_pain"
##
## $NbAllValues
## [1] 6
##
## $DecValues
##       exang
##     0     2
```

```
##
## $MLog
##             [,1]       [,2]
## [1,] -0.1150693 -1.0525212
## [2,] -2.2192035 -0.4293326
## [3,] -1.5368672 -0.2754120
## [4,] -0.6613985 -2.4849066
## [5,] -1.4880771 -2.1972246
## [6,] -3.1463051 -3.0726933
```

Il n'y aucune information nouvelle par rapport à la version précédente. Seule l'organisation des logarithmes des probabilités conditionnelles est différente (matrice **$MLog** plutôt que liste **$tc**).

4.2 Programmation de predict()

En prédiction, la matrice des codes des explicatives **D** est transformée en matrice d'indicatrices **M** via un codage disjonctif complet. L'augmentation de l'occupation mémoire qui en découle est la principale faiblesse de cette approche. Il faut en être conscient. Sur une grande base avec des variables qualitatives comportant un grand nombre de modalités, elle peut être conséquente. Il n'y a pas de solution parfaite. En revanche, je le dis toujours à mes étudiants, nous devons absolument mesurer (soupeser) les avantages et inconvénients des approches que nous mettons en place.

Cette transformation permet de réduire le calcul des **scores** à un produit matriciel, domaine où R est irrésistible. Nous devrions y gagner très largement en temps de calcul.

Reprenons notre exemple de l'individu (exang = yes, chest_pain = asympt) pour détailler les calculs. Après binarisation, il est décrit par le vecteur (la matrice à une ligne puisque nous avons un seul individu à classer) suivant : **(0, 1, 1, 0, 0, 0, 0)**. Le produit matriciel avec la propriété **$MLog** de l'objet issu de l'apprentissage fournit : (-2.219 + (-1.537), -0.429 + (-0.257)) = (-3.756, -0.704). Auquel nous ajoutons les logarithmes des probabilités a priori **$prior** (-0.4354, -1.009). Nous obtenons finalement : (-3.756 + (-0.4354), -0.704 + (-1.009)) = **(-4.209, -1.713)**. Les scores d'affectation sont bien les mêmes (section 2.2.3).

Voici la fonction R correspondante.

```
#prediction via le calcul matriciel
mypredict_2 <- function(model,newdata){
  #vérifier les noms de variables
 if (length(intersect(model$var_names,colnames(newdata))) < model$p){stop("souci de variables")}
  #récupérer les variables, et dans le bon ordre
  D <- newdata[model$var_names]
  #transformer les factors en code (indice de modalité)
  D <- as.matrix(data.frame(lapply(D,function(x){return(unclass(x))})))
```

```
  #//!\\ transformer D en matrice de codes binaires -- attention inflation de
colonnes si nombreuses modalités
  M <- t(apply(D,1,function(x){v <- rep(0,model$NbAllValues);v[model$DecValues+x] <- 1; return(v)}))
  #//!\\ simple produit matriciel pour la prédiction
  scores <- M %*% model$MLog
  #il faut rajouter les prior
  for (k in 1:model$nb_modalites){scores[,k] <- scores[,k]+model$prior[k]}
  #return(scores)
  #prediction
  mPred <- model$modalites[apply(scores,1,which.max)]
  return(mPred)
}
```

Appliquons la prédiction sur notre base jouet.

```
#prédiction en resubstitution
print(system.time(pred_2 <- mypredict_2(modele_heart2,newdata=heart)))

##    user  system elapsed
##       0       0       0
```

Elle est cohérente avec les versions précédentes.

```
#vérification des prédictions
print(table(pred_1A,pred_2))

##           pred_2
## pred_1A    negative positive
##   negative      208        0
##   positive        0       78
```

4.3 Comparaison des performances

Est-ce que ces modifications sont bénéfiques en termes de temps de calcul ? Etudions la question sur nos deux bases benchmark.

4.3.1 Sur la base “large”

Sur “large”, la durée de l’apprentissage n’est pas impactée par la recopie des coefficients (on peut les voir comme cela maintenant) dans une structure différente.

```
#apprentissage - structure matricielle - base "large"
print(system.time(m_large2 <- myfit_2(CLASSE ~ ., data = large)))

##    user  system elapsed
##    3.96    0.11    4.08
```

Le bénéfice en prédiction est incommensurable.

```
#prédiction - structure matricielle
print(system.time(mypredict_2(m_large2,newdata = large)))
```

```
##    user  system elapsed
##    0.50    0.03    0.53
```

Le calcul est quasi-instantané. Nous rappelons que nous étions partis d'une durée de l'ordre de 200 secondes dans la version basée sur les listes. Et que notre meilleure implémentation plafonnait à 12 secondes à peu près.

4.3.2 Sur la base "long"

Sur notre seconde base, la phase d'apprentissage *fit()* est tout aussi rapide que pour la version précédente :

```
#apprentissage - structure matricielle - base "long"
print(system.time(m_long2 <- myfit_2(ONDE ~ ., data = long)))

##    user  system elapsed
##    0.02    0.00    0.01
```

Et encore une fois, la phase de prédiction est quasi-instantanée…

```
#prédiction - structure matricielle
print(system.time(mypredict_2(m_long2,newdata = long)))

##    user  system elapsed
##    0.06    0.00    0.06
```

… alors que nous étions à 1.81 secondes dans la meilleure version précédemment.

Il n'y a pas de doute. Le placer en position de réaliser des calculs matriciels permet de réduire considérablement les durées d'exécutions sous R.

5 Bilan

Voici un tableau récapitulatif des durées d'exécution (''*elaspsed*'' en secondes) des fonctions predict() sur nos deux bases ''benchmark''.

Stratégie	**Base ''large'' (202 obs., 7143 descripteurs)**	**Base ''longue'' (10000 obs., 19 descripteurs)**
Liste, accès indicé, boucles **for**	250.5	10.91
Liste, Accès indicé, boucles **sapply**	222.30	12.03
Liste, Accès non indicé	15.26	1.81
Calcul matriciel	0.53	0.06

6 Conclusion

Il n'y a pas de mauvais langages, il n'y a que des mauvais programmeurs… des programmeurs qui ne s'adaptent pas aux spécificités des outils qu'ils utilisent en tous les cas (restons raisonnables dans nos affirmations). La citation est cruelle mais elle illustre bien notre propos. J'entends à plusieurs reprises des personnes, très sensées par ailleurs, m'affirmer haut et fort que R est lent, que leur programme est plombé par le déficit de performances de l'interpréteur, et qu'ils ont dû changer de langage (suivez mon regard vers Python). Je leur dis, d'accord, montrez-moi votre code et on va faire une petite analyse approfondie de ce que vous faites pour voir ce qui pourrait coincer. Etrangement, je sens une certaine réticence.

Dans l'exemple de la programmation du naive bayes que nous avons abordé dans ce tutoriel, l'enjeu était de mettre R dans la configuration où il s'exprime le mieux c.-à-d. le calcul matriciel. Au prix d'une petite gymnastique de préparation des données, et une occupation mémoire un peu plus conséquente il faut le reconnaître, on n'a rien sans rien, les gains en rapidité des implémentations sont faramineux.

7 Références

(REF) Tutoriel Tanagra, "Le classifieur Bayésien Naïf revisité", mars 2010.

Tutoriel Tanagra, "Bayésien naïf pour prédicteurs continus", octobre 2010.

Tutoriel Tanagra, "Programmer efficacement sous R", février 2019.