Discrétisation supervisée rapide

Tutoriel Tanagra

1 Introduction

La discrétisation est un processus de transformation qui permet de recoder une variable quantitative en qualitative ordinale par découpage en intervalles ("La discrétisation des variables quantitatives", octobre 2014). Deux questions clés se posent : comment définir le nombre d'intervalles, comment en choisir les seuils de découpage. Dans le cadre de la classification supervisée, la situation est à la fois plus complexe et plus simple. Plus complexe parce que le procédé doit, en plus, tenir compte des informations portées par les classes d'appartenance. Plus simple parce que l'objectif est clair, il s'agit de produire un ensemble d'intervalles où, pour chacun d'entre eux, une des classes (une des modalités de la variable cible) est largement surreprésentée par rapport aux autres.

Sous R, "discretization" (Kim, 2012) est le package le plus populaire en matière de discrétisation supervisée (qui arrive en premier dans les requêtes Google tout du moins). Il propose les deux familles d'approches les plus souvent citées dans ce cadre.

Les techniques ascendantes (bottom-up) consistent à, en partant d'une partition initiale (soit triviale, un intervalle par individu ; soit un premier étage par découpage à fréquences égales assez fin), effectuer une sorte de CAH (classification ascendante hiérarchique) où l'on fusionne en priorité les groupes adjacents où les distributions des classes sont similaires. La méthode ChiMerge (Kerber, 1992) en est certainement un des représentants les plus emblématiques. Mais, pour avoir moi-même beaucoup œuvré dans le domaine, je peux dire que le paramétrage des approches ascendantes est difficile. Elles aboutissent souvent à un partitionnement trop fin, les temps d'exécution étant de surcroît dissuasifs sur les grandes bases de données.

La démarche est inverse pour les techniques descendantes (top down). Il s'agit de découper de manière récursive l'espace des valeurs de la variable à discrétiser avec pour objectif de différencier au maximum les distributions des classes. La méthode MDLP (Fayyad et Irani, 1993) est la plus citée. Elle présente l'avantage et l'inconvénient de ne pas présenter de paramètres à régler en fonction des bases à traiter. Avantage parce qu'elle nous dispense ainsi d'un épineux problème à résoudre. Inconvénient parce que proposer une méthode qui serait adaptée pour tout type de bases est illusoire et, dans les faits, on se rend compte qu'elle exagère le nombre d'intervalles lorsque nous traitons les grandes volumétries. En termes de temps d'exécution, MDLP est censée être rapide, plus rapide que les approches ascendantes en tous les cas. Mais tout dépend de la qualité de son implémentation en réalité. En inspectant de près le code du package "discretization", j'ai constaté qu'il était pour le moins perfectible. Je me suis dit qu'on pouvait faire mieux et, pourquoi pas, proposer une technique apparentée mais plus efficace.

En effet, à bien y regarder, MDLP n'est plus ni moins qu'un algorithme d'induction d'arbres de décision avec une seule variable prédictive quantitative. Par segmentations binaires successives, nous aboutissons à un partitionnement en intervalles. MDLP se démarque par l'adoption d'une règle d'arrêt basé sur la théorie de la description minimale des messages. La cohérence théorique y est, assurément. Mais l'efficacité pratique est une autre histoire. Dans cette même veine, pourquoi ne pas adopter tout simplement une implémentation existante de la construction d'arbres de décision et de l'adapter pour qu'il réponde à un objectif de discrétisation de variables. Dans ce tutoriel, je creuse cette idée en m'appuyant sur la fonction `rpart()` du package éponyme. Je montre qu'il est très facile de la tourner vers la résolution d'un problème de discrétisation. Son paramétrage à cet effet ne présente pas de grandes difficultés. Il nous ouvre même des perspectives d'analyses plus riches par rapport au MDLP original. Enfin, et ce n'est pas moindre de ses mérites, la nouvelle fonction de discrétisation supervisée ainsi définie sous R s'avère autrement plus rapide que l'implémentation de MDLP proposée par le package "discretization" (et "RWeka" que j'ai également identifié par la suite) sur les grandes bases de données.

2 Principe de la discrétisation supervisée

Nous prenons pour exemple la fameuse base des Iris de Fisher pour illustrer le fonctionnement de la discrétisation supervisée. Elle est intégrée dans le package "datasets" de R. Nous la chargeons en mémoire avec l'instruction `data()`.

```
#données iris
data(iris)
print(str(iris))

## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1
...
## NULL
```

La base est constituée de 4 descripteurs et d'une variable cible catégorielle, "Species", à 3 modalités {"setosa", "versicolor", "virginica"}.

2.1 Distributions conditionnelles

Nous essayons d'isoler les classes dans des intervalles définies sur la variable "Petal.Width" dans un premier temps.

Nous l'intégrons avec ''Species'' dans un data frame spécifique.

```
#sous data.frame avec les 2 dernières variables
subIris <- iris[,c("Petal.Width","Species")]
```

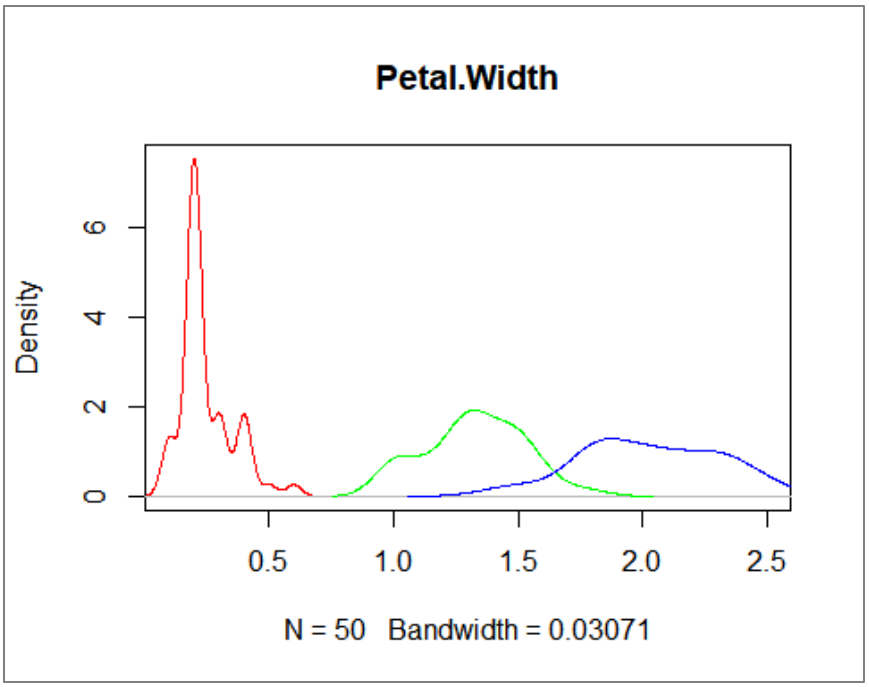
Avant de commencer les calculs, nous illustrons graphiquement les courbes de densité estimées de "Petal.Width" conditionnellement aux valeurs de "Species". Ce type de graphique est présent dans plusieurs packages. J'ai préféré écrire ma propre fonction pour pouvoir appréhender efficacement les très grandes bases par la suite.

```
#fonction pour courbe de densité conditionnelle
mydensityplot <- function(x,y,var_name=""){
  #étendue des valeurs
  etendue <- range(x)
  #test si y est factor, y remédier sinon
  if (!is.factor(y)){
    y <- factor(y)
  }
  #calcul des courbes de densité conditionnellement à y
```

```
  d <- lapply(levels(y),function(k){density(x[y==k])})
  #ordonnée max.
  max_height <- sapply(d, function(v){return(max(v$y))})
  #palette de couleurs
  couleurs <- rainbow(nlevels(y))
  #graphique de la première courbe
  plot(d[[1]],col=couleurs[1],ylim=c(0,max(max_height)),xlim=etendue,main=var_name)
  #les autres courbes dans le même graphique
  for (k in 2:length(d)){
    lines(d[[k]],col=couleurs[k])
  }
}
```

Voici le graphique obtenu lorsque la fonction est appliquée sur “Petal.Width” vs. “Species”.

```
#pour la variable à traiter
mydensityplot(subIris$Petal.Width,subIris$Species,"Petal.Width")
```



Pour les 3 modalités de “Species” (Setosa, Versicolor, Virginica), nous distinguons nettement les décalages entre les distributions de “Petal.width”. Pour obtenir des groupes “purs” au sens de “Species”, nous imaginons très bien un découpage en 3 intervalles avec pour seuils, approximativement, 0.8 et 1.75. Dans le premier intervalle]-∞,0.8], la modalité “Setosa” serait sur représentée ; dans le second]0.8, 1.75], ce serait “Versicolor” ; dans le dernier]1.75, +∞[, nous aurons “Virginica”.

Comment obtenir ces résultats par le calcul ?

2.2 Discrétisation MDL du package "discretization"

La fonction MDLP est implémentée dans le package "discretization" que nous chargeons. Nous l'appliquons à notre sous-ensemble de données. Elle fait l'hypothèse que les variables à discrétiser sont dans les premières colonnes, la variable cible étant en dernière position.

```
#chargement du package discretization
library(discretization)

#discrétisation supervisée de la variable Petal.Width
discPW <- mdlp(subIris)
```

Nous accédons aux bornes de découpages comme suit :

```
#bornes de discrétisation
print(discPW$cutp)

## [[1]]
## [1] 0.80 1.75
```

Le data frame avec la variable recodée est également accessible :

```
#données recodées - 6 premières observations
print(head(discPW$Disc.data))

##   Petal.Width Species
## 1           1  setosa
## 2           1  setosa
## 3           1  setosa
## 4           1  setosa
## 5           1  setosa
## 6           1  setosa
```

Pour apprécier la qualité du découpage, nous croisons la variable recodée avec les classes définies par la variable cible.

```
#qualité du découpage
print(table(subIris$Species,discPW$Disc.data$Petal.Width))

##
##                1  2  3
##   setosa      50  0  0
##   versicolor   0 49  1
##   virginica    0  5 45
```

Dans le premier intervalle, la classe "setosa" est la seule représentée ; dans le second, nous observons très majoritairement "versicolor" ; dans le troisième, "virginica". L'idéal aurait

été d'obtenir des intervalles purs, non-bruités. Mais ce type de configuration arrive rarement en pratique. Notre discrétisation est déjà d'excellente facture.

2.3 Arbre de décision avec une seule variable prédictive

Quel parallèle pourrions-nous faire entre cet algorithme et les arbres de décision ? Nous appliquons `rpart()` sur ces mêmes données où la seule "Petal.Width" serait l'explicative candidate. L'arbre est binaire par nature, mais rien ne l'empêche d'utiliser la même variable de partitionnement sur les différents nœuds, avec de seuils de segmentation différents bien entendu.

Nous chargeons la librairie et nous spécifions les paramètres d'analyse.

```
#charger la librairie rpart
library(rpart)

#paramètres d'analyse
paramArbre <-
rpart.control(minsplit=20,minbucket=5,maxdepth=3,cp=0.15,xval=0,maxcompete=0,maxsurro
gate=0,usesurrogate=0)
```

Arrêtons-nous un instant sur ces paramètres :

- "maxcompete = 0", "maxsurrogate = 0", "usesurrogate = 0" visent avant tout à réduire au minimum les calculs et les informations additionnelles stockées lors de la mise en œuvre de l'algorithme, inutiles dans le contexte de la discrétisation.
- "xval = 0" désactive le processus de validation croisée sur lequel s'appuie `rpart()` pour définir la taille adéquate de l'arbre en post-élagage. Nous ne l'exploitons pas dans notre contexte.
- "minsplit = 20" indique qu'il faut au moins 20 observations sur un sommet pour tenter une segmentation.
- "minbucket = 5" indique qu'il faut au moins 5 observations sur chaque feuille pour accepter le fruit d'une segmentation. "minbucket" et "minsplit" permettent de peser sur le support des intervalles, à régler en fonction de la taille de la base.
- "maxdepth = 3" correspond à la profondeur maximale de l'arbre. Celle-ci étant binaire, nous avons une discrétisation en 8 intervalles au maximum avec ce paramétrage (la racine correspond à depth = 0).

- "cp = 0.15" est la réduction relative minimale du critère de qualité de l'arbre pour qu'une segmentation soit acceptée. Ce paramètre est moins intuitif. Définir la valeur adaptée n'est pas aisée. Il faut retenir l'idée que le nombre d'intervalles sera moindre si nous augmentons sa valeur, l'inverse sinon.

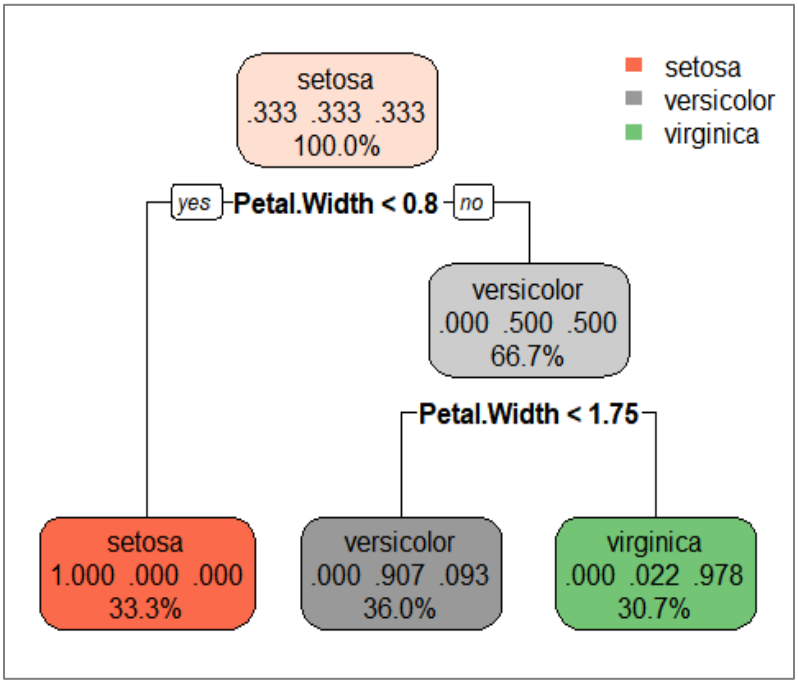
Nous lançons la construction de l'arbre :

```
#arbre
arbre <- rpart(Species ~ Petal.Width, data = subIris, control = paramArbre)
print(arbre)

## n= 150
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 150 100 setosa (0.33333333 0.33333333 0.33333333)
##   2) Petal.Width< 0.8 50   0 setosa (1.00000000 0.00000000 0.00000000) *
##   3) Petal.Width>=0.8 100  50 versicolor (0.00000000 0.50000000 0.50000000)
##     6) Petal.Width< 1.75 54   5 versicolor (0.00000000 0.90740741 0.09259259) *
##     7) Petal.Width>=1.75 46   1 virginica (0.00000000 0.02173913 0.97826087) *
```

Avec un affichage graphique, c'est mieux…

```
#affichage sympathique
library(rpart.plot)
rpart.plot(arbre,digits=3)
```



L'unique variable prédictive est utilisée à deux reprises dans l'arbre, avec des bornes de découpages différents… correspondants aux seuils de discrétisation mises en évidence par MDLP. De fait, utiliser un arbre de décision pour la discrétisation supervisée est tout à fait licite au regard de cet exemple.

La propriété "`$splits`" nous donne des informations sur les segmentations successives initiées dans l'arbre.

```
#description des découpages
print(arbre$splits)

##             count ncat improve index adj
## Petal.Width   150   -1 50.0000  0.80   0
## Petal.Width   100   -1 38.9694  1.75   0
```

Elle correspond à une matrice où la colonne 'index' décrit les seuils de découpage. Nous pouvons dès lors en déduire les limites des intervalles définies par la discrétisation.

```
#délimitation des intervalles de discrétisation
bornes <- arbre$splits[,'index']
bornes <- c(-Inf,sort(bornes),+Inf)
print(bornes)

##             Petal.Width Petal.Width
##        -Inf        0.80        1.75         Inf
```

Appliquées aux données, nous obtenons la variable recodée.

```
#appliquer aux données
discPWArbre <- cut(subIris$Petal.Width,bornes,labels=FALSE)
print(discPWArbre)

##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [71] 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3
## [106] 3 2 3 3 3 3 3 3 3 3 3 3 3 2 3 3 3 3 3 3 3 3 2 3 3 3 2 2 3 3 3 3 3
## [141] 3 3 3 3 3 3 3 3 3 3
```

Pour cet exemple, nous constatons qu'elle concorde parfaitement avec celle issue de MDLP.

```
#comparaison discrétisation arbre avec celui de MDLP (package discretization)
print(table(discPWArbre,discPW$Disc.data$Petal.Width))

##
## discPWArbre  1  2  3
##           1 50  0  0
##           2  0 54  0
##           3  0  0 46
```

2.4 Implémentation d'un algorithme de discrétisation basé sur rpart()

Forts de ces éléments, nous implémentons une fonction de discrétisation basée sur l'algorithme d'induction d'arbres `rpart()` :

```
#fonction réalisant la discrétisation
mydiscVar <- function(x,y,minsplit=20,minbucket=5,maxdepth=3,cp=0.15){
  #paramètres de l'arbre
  paramTree <-
rpart.control(minsplit=minsplit,minbucket=minbucket,maxdepth=maxdepth,cp=cp,xval=0,ma
xcompete=0,maxsurrogate=0,usesurrogate=0)
  #trier pour plus de rapidité
  index <- order(x)
  yTemp <- y[index]
  xTemp <- x[index]
  #induction de l'abre
  tree <- rpart(yTemp ~ xTemp, control=paramTree)
  #récupération des bornes
  #si l'arbre s'est arrêté à la racine
  if (is.null(tree$splits)){
    seuils <- c(-Inf,+Inf)
  } else
  {
    seuils <- c(-Inf,sort(tree$splits[,'index']),+Inf)
  }
  #appliquer aux données
  z <- cut(x,seuils,labels=FALSE)
  #return
  return(list(cuts=seuils,data=z))
}

#tester la fonction
discPWMine <- mydiscVar(subIris$Petal.Width,subIris$Species)

#bornes
print(discPWMine$cuts)

##       xTemp xTemp
##  -Inf  0.80  1.75   Inf
```

Deux remarques :

- Il faudrait aller dans le cœur de `rpart()` pour savoir si le tri préalable des données, avant la construction de l'arbre, est réellement bénéfique ou non.
- Si l'arbre s'arrête à la racine, aucun découpage n'a été effectué, la propriété "`$splits`" est égale à NULL. D'où le test avant la récupération des seuils.

Testons la fonction sur notre jeu de données exemple. Nous affichons les bornes inférées…

```
#tester la fonction
discPWMine <- mydiscVar(subIris$Petal.Width,subIris$Species)
```

```
#bornes
print(discPWMine$cuts)

##       xTemp xTemp
##  -Inf  0.80  1.75   Inf
```

… et la variable recodée.

```
#variable recodée
print(discPWMine$data)

##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [71] 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3
## [106] 3 2 3 3 3 3 3 3 3 3 3 3 3 2 3 3 3 3 3 3 3 3 3 2 3 3 3 2 2 3 3 3 3 3
## [141] 3 3 3 3 3 3 3 3 3 3
```

2.5 Généralisation au traitement d'un ensemble de variables

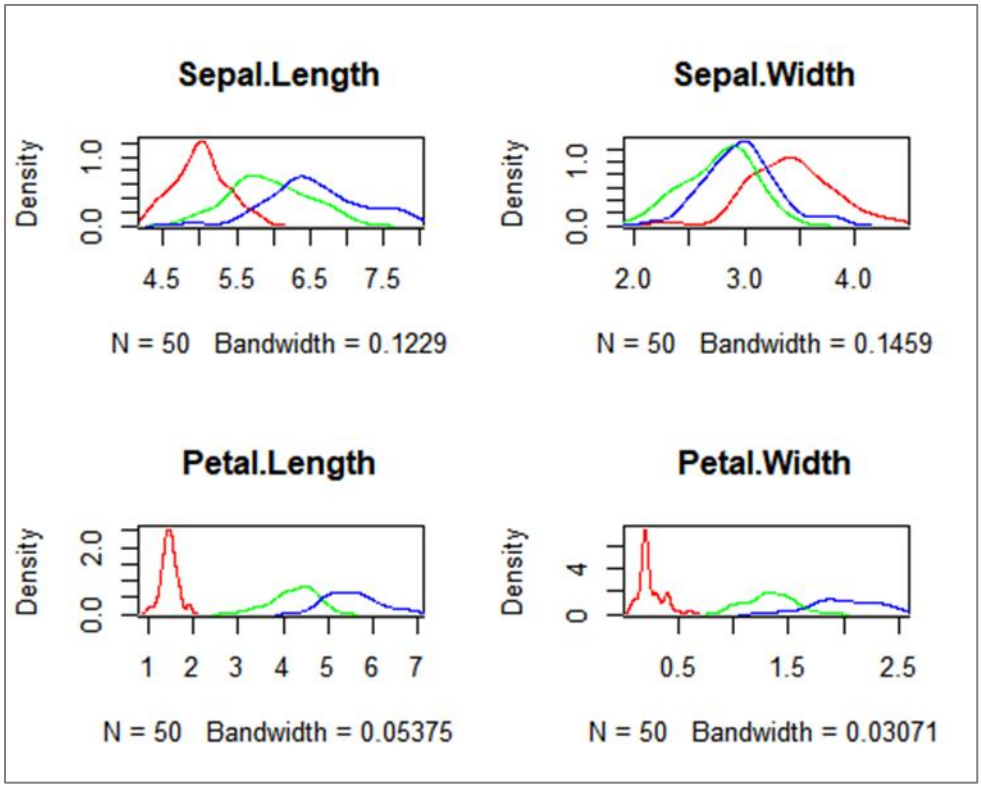
2.5.1 Les autres variables de la base IRIS

La base IRIS est constituée de 4 descripteurs continus que nous allons tour à tour discrétiser. Nous affichons leurs distributions conditionnelles pour se faire une idée des solutions que nous sommes susceptibles d'obtenir.

```
#traitement de l'ensemble des variables
#densités conditionnelles
par(mfrow=c(2,2))
for (j in 1:4){
  mydensityplot(iris[,j],iris$Species,colnames(iris)[j])
}

par(mfrow=c(1,1))
```

Le découpage en 3 intervalles semble s'imposer pour 3 des variables (Sepal.Length, Petal.Length, Petal.Width). On devine à peu près les valeurs des seuils de discrétisation. La situation est plus ambigüe pour Sepal.Width. Un découpage en 2 intervalles (bornes autour de 3.3) seulement semble être la bonne solution. Dans le premier intervalle seront mélangées les observations "versicolor" et "virginica", dans le second sont isolés principalement les ''setosa''. Voyons si nos algorithmes sont capables d'identifier ces résultats.



2.5.2 Fonctionnement de MDLP

MDLP sait traiter directement les data frame comportant les variables à discrétiser avec, en dernière position, la variable cible de référence. La conformation de la base IRIS est totalement accord avec cela. Il ne nous reste qu'à appeler la fonction. Nous affichons les bornes de discrétisation :

```
#discrétisation avec mdlp
#par hypothèse, var. classe en dernière position
discIrisMdlp <- mdlp(iris)

#bornes
print(discIrisMdlp$cutp)

## [[1]]
## [1] 5.55 6.15
##
## [[2]]
## [1] 2.95 3.35
##
## [[3]]
## [1] 2.45 4.75
##
## [[4]]
## [1] 0.80 1.75
```

Pour (Sepal.Length [[1]], Petal.Length [[3]], Petal.Width [[4]]), les résultats sont en accord avec les décalages observés dans les courbes de densité conditionnelles.

Pour Sepal.Width, nous constatons que l'algorithme propose un découpage en 3 intervalles quand-même. Voyons ce qu'il en est de la distribution des classes dans ces intervalles :

```
#croisement de Sepal.Wdth recodée avec Species -- MDLPC
print(table(iris$Species,discIrisMdlp$Disc.data$Sepal.Width))
```

```
##
##                1  2  3
##   setosa       2 17 31
##   versicolor 34 15  1
##   virginica  21 24  5
```

La solution n'est pas très satisfaisante. Dans le troisième intervalle, "setosa" est certes surreprésentée. Dans les deux autres en revanche, aucune des classes n'émerge et les distributions ne se démarquent pas fortement. Le partitionnement paraît superflu.

2.5.3 Ecriture d'une fonction de discrétisation basée sur rpart()

Nous écrivons une fonction qui englobe les appels à la fonction de discrétisation par arbre ci-dessus, de manière à répondre aux mêmes spécifications que MDLP. Elle renvoie également les bornes de découpage et les variables recodées.

```
#fonction pour discrétisation data.frame
mydiscDataFrame.fit <- function(df,minsplit=20,minbucket=5,maxdepth=3,cp=0.15){
  #nombre de variables à discrétiser
  p <- ncol(df) - 1
  #discrétiser chaque variable
  cutp <- list()
  disc.data <- list()
  for (j in 1:p){
    #traiter une variable
    res <- mydiscVar(df[,j],df[,p+1],minsplit,minbucket,maxdepth,cp)
    #ajouter les résultats
    cutp[[j]] <- res$cuts
    disc.data[[j]] <- res$data
  }
  #ajouter la dernière colonne
  disc.data[[j+1]] <- df[,p+1]
  #transformer en data.frame
  disc.data <- data.frame(disc.data)
  #renommer
  colnames(disc.data) <- colnames(df)
  #return
```

```
  return(list(cutp = cutp, disc.data = disc.data))
}
```

Appliquées sur les données IRIS.

```
#traiter iris
discIrisArbre <- mydiscDataFrame.fit(iris)

#bornes
print(discIrisArbre$cutp)

## [[1]]
##       xTemp xTemp
##  -Inf  5.45  6.15   Inf
##
## [[2]]
## [1] -Inf 3.35  Inf
##
## [[3]]
##       xTemp xTemp
##  -Inf  2.45  4.75   Inf
##
## [[4]]
##       xTemp xTemp
##  -Inf  0.80  1.75   Inf
```

Elle produit les mêmes résultats que MDLP sur les variables (Sepal.Length [[1]], Petal.Length [[3]], Petal.Width [[4]]). Elle propose un découpage en deux intervalles seulement en revanche pour Sepal.Width. L'approche rejoint en cela notre première intuition suite à l'inspection du graphique des densités conditionnelles ci-dessus.

En croisant Sepal.Width recodée avec Species, nous avons :

```
#croisement de Sepal.Wdth recodée avec Species -- disc. par arbre
print(table(iris$Species,discIrisArbre$disc.data$Sepal.Width))

##
##                1  2
##   setosa      19 31
##   versicolor 49  1
##   virginica   45  5
```

3 Discrétisation sur les grandes bases de données

3.1 Wave dataset

Maintenant que nous avons assez bien décrypté le fonctionnement des méthodes de discrétisation et de leur implémentation. Nous nous intéressons dans cette section à leur

application sur des grandes bases de données. Nous souhaitons procéder au découpage des 21 variables qui composent la base des Ondes de Breiman. Avec le générateur, nous avons créé un dataset de 100.000 observations. La variable cible "onde" est à 3 modalités {A, B, C}.

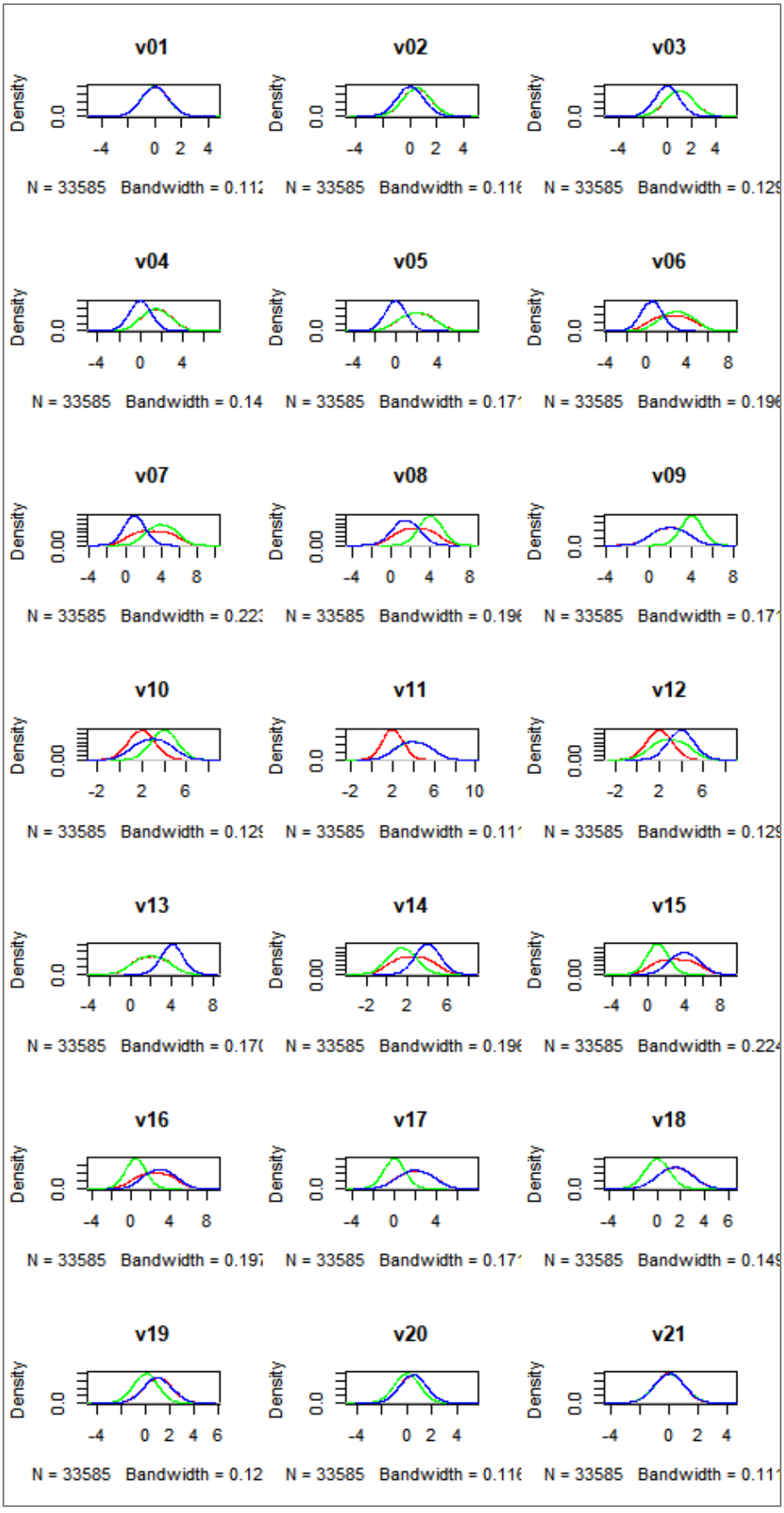
Nous chargeons et inspectons les données.

```
#traiter le fichier wave
setwd("... votre dossier de travail ...")
wave <- read.csv("wave100k.txt",header=T,sep="\t")
print(str(wave))
```

```
## 'data.frame':    100000 obs. of  22 variables:
##  $ v01 : num  1.09 -0.44 0.74 2.29 0.19 -1.5 0.24 -0.34 0.71 -1.3 ...
##  $ v02 : num  -0.8 0.08 0.55 1.46 0.17 -1.46 1.72 -0.2 -0.06 0.67 ...
##  $ v03 : num  -0.54 -0.36 0.14 2.08 -0.21 -0.68 1.52 4.12 3.13 -1.04 ...
##  $ v04 : num  0.54 1.43 -0.82 1.27 -0.48 0.73 0.41 2.33 2.54 -0.73 ...
##  $ v05 : num  -0.35 0.07 0.47 1.76 0.99 -0.37 2.1 2.18 2.27 1.18 ...
##  $ v06 : num  -0.83 0.77 0.58 4.89 1.53 0.64 2.92 3.46 5.49 1.02 ...
##  $ v07 : num  -0.14 -0.06 2.4 5.02 1.73 -0.1 2.97 6.79 4.86 0.76 ...
##  $ v08 : num  -0.03 1.96 1.42 3.75 -0.52 0.13 2.48 4.87 3.44 3.44 ...
##  $ v09 : num  -1.27 0.09 4.34 3.92 -0.72 1.36 4.12 4.3 4.27 0.82 ...
##  $ v10 : num  1 2.07 1.7 2.9 -0.2 0.7 4.36 1.62 2.92 1.08 ...
##  $ v11 : num  0.53 3.99 3.85 1.44 0.75 1.52 2.43 2.44 0.77 2.94 ...
##  $ v12 : num  3.94 2.66 3.16 0.27 3.47 3.22 4.25 -1.02 1.61 1.58 ...
##  $ v13 : num  2.32 4.57 2.63 1.51 4.15 4.13 2.82 -0.28 2.06 3.69 ...
##  $ v14 : num  4.3 5.13 5.05 0.89 3.81 5.83 1.85 1.57 0.85 3.82 ...
##  $ v15 : num  7.2 2.98 4.59 2.93 5.22 4.97 2.37 0.41 0.44 3.82 ...
##  $ v16 : num  4.51 3.11 4.79 0.48 3.95 4.48 -0.13 -0.18 0.62 1.77 ...
##  $ v17 : num  4.63 2.28 2.63 0.58 3.94 3.09 0.53 0.07 -0.15 2.05 ...
##  $ v18 : num  2.46 2.51 2.14 0.99 2.45 3.45 0.63 1.45 -0.45 1.63 ...
##  $ v19 : num  2.45 -0.35 2.98 2.35 1.24 2.46 0 -0.29 0.8 0.98 ...
##  $ v20 : num  1.98 1.07 1.13 -0.64 1.73 1.04 0.42 -0.32 2.81 1 ...
##  $ v21 : num  0.58 -2.34 -0.42 -1.01 0.9 0.59 -1 -0.03 0.33 0.91 ...
##  $ onde: Factor w/ 3 levels "A","B","C": 1 1 3 1 1 1 2 1 1 3 ...
```

Avec les distributions conditionnelles, nous pouvons nous faire une première idée de l'intérêt de la discrétisation pour les 21 variables de la base.

```
#distributions conditionnelles
par(mfrow=c(7,3))
for (j in 1:21){
  mydensityplot(wave[,j],wave$onde,colnames(wave)[j])
}
par(mfrow=c(1,1))
```


v01
v02
v03
v04
v05
v06
v07
v08
v09
v10
v11
v12
v13
v14
v15
v16
v17
v18
v19
v20
v21
Density
N = 33585 Bandwidth = 0.11
N = 33585 Bandwidth = 0.11
N = 33585 Bandwidth = 0.12
N = 33585 Bandwidth = 0.14
N = 33585 Bandwidth = 0.17
N = 33585 Bandwidth = 0.19
N = 33585 Bandwidth = 0.22
N = 33585 Bandwidth = 0.19
N = 33585 Bandwidth = 0.17
N = 33585 Bandwidth = 0.12
N = 33585 Bandwidth = 0.11
N = 33585 Bandwidth = 0.12
N = 33585 Bandwidth = 0.17
N = 33585 Bandwidth = 0.19
N = 33585 Bandwidth = 0.22
N = 33585 Bandwidth = 0.19
N = 33585 Bandwidth = 0.17
N = 33585 Bandwidth = 0.14
N = 33585 Bandwidth = 0.12
N = 33585 Bandwidth = 0.11
N = 33585 Bandwidth = 0.11

Un coup d'œil rapide laisse à penser qu'une discrétisation en 2 intervalles convient dans la plupart des cas. Et trouver un découpage pertinent pour les variables V01 et V21 (un peu moins évident pour V02 et V20) est illusoire.

Voyons comment se comportent nos fonctions.

3.2 Discrétisation top-down avec "discretization"

Nous lançons la fonction MDLP du package "discretization" sur les données WAVE.

```
#mdlp
print(system.time(waveMdlp <- mdlp(wave)))

##    user  system elapsed
##  127.58    1.16  129.50
```

Le temps de traitement est de 127.58 secondes. Ne disposant pas d'autres références, nous ne pouvons rien dire pour l'instant

Voyons les solutions de découpage pour chaque variable :

```
#bornes pour WAVE -- fonction MDLP
print(waveMdlp$cutp)

## [[1]]
## [1] "All"
##
## [[2]]
## [1] -2.465 -0.975 -0.395  0.535  0.185  0.985  1.515  2.165
##
## [[3]]
##  [1] -1.535 -0.995 -0.195  0.815 -0.485  0.125  0.435  0.955  1.285  1.865
## [11]  1.595  2.165  2.605  3.405
##
## [[4]]
##  [1] -1.175 -0.765  0.095  1.135 -0.265  0.415  0.685  0.875  1.415  1.685
## [11]  2.115  1.835  2.595  3.215
##
## [[5]]
##  [1] -1.085 -0.545  0.345  1.375  0.045  0.615  0.935  1.105  1.495  1.885
## [11]  2.285  2.065  2.595  3.175
##
## [[6]]
##  [1] -0.825  0.045  0.915  2.005  0.625  1.165  1.405  1.695  2.265  2.445
## [11]  3.015  2.705  3.295  3.685  4.415
##
## [[7]]
##  [1] -0.185  0.645  1.295  2.535  1.065  1.525  1.895  2.185  2.805  2.985
## [11]  3.695  3.385  3.995  4.495  5.395
##
```

```
## [[8]]
##  [1] 0.525 1.035 1.575 2.715 1.335 1.895 2.265 2.485 3.015 3.255 3.965
## [12] 3.575 4.095 4.485 4.885 5.615
##
## [[9]]
##  [1] 0.175 1.185 1.725 2.615 2.045 2.265 2.855 3.065 3.545 3.405 3.865
## [12] 4.525 5.145
##
## [[10]]
##  [1] 0.425 1.125 1.685 2.955 2.085 2.375 2.755 3.315 3.925 3.645 4.345
## [12] 4.685 5.265
##
## [[11]]
##  [1] 0.855 1.645 2.265 3.375 1.875 2.645 2.785 3.015 3.565 3.765 4.405
## [12] 4.015 4.825 5.245
##
## [[12]]
##  [1] 0.335 1.235 1.905 3.125 1.565 2.195 2.605 2.855 3.495 4.035 3.845
## [12] 4.415 4.805 5.355
##
## [[13]]
##  [1] 0.635 0.915 1.685 2.695 1.325 1.955 2.275 2.475 2.935 3.175 3.745
## [12] 3.395 4.055 4.645 5.295
##
## [[14]]
##  [1] 0.145 0.775 1.575 2.745 1.225 1.945 2.215 2.485 2.975 3.325 3.825
## [12] 3.655 4.155 4.505 5.045
##
## [[15]]
##  [1] -0.595  0.125  0.725  1.335  2.625  1.695  2.015  2.335  2.905  3.215
## [11]  3.675  3.945  4.325  4.835
##
## [[16]]
##  [1] -0.785 -0.085  0.765  1.985  0.365  0.615  1.135  1.435  1.625  2.195
## [11]  2.415  3.035  2.695  3.665  4.515
##
## [[17]]
##  [1] -1.295 -0.515  0.345  1.415 -0.115  0.695  0.945  1.165  1.795  2.325
## [11]  2.055  2.705  2.965  3.435
##
## [[18]]
##  [1] -1.575 -0.855  0.145  1.235 -0.175  0.375  0.795  1.035  1.475  1.705
## [11]  2.125  1.845  2.545  2.865  3.545
##
## [[19]]
##  [1] -1.825 -0.835 -0.375  0.635 -0.125  0.255  0.875  1.165  1.625  2.125
## [11]  2.965  3.645
##
## [[20]]
## [1] -1.585 -0.765  0.285 -0.305  0.795  1.185  1.675  2.065  2.645
##
## [[21]]
## [1] "All"
```

Mise à part les soutions qui semblent correctes pour V01 et V21, le moins qu'on puisse dire est que l'approche a été très optimiste pour les autres variables. Manifestement, alors qu'elle semblait partitionner fort à propos pour la base IRIS, MDLP paraît inadaptée pour notre version de la base WAVE avec 100.000 observations. Ce résultat est connu. MDLP a tendance à produire trop d'intervalles sur les grandes bases de données.

3.3 Discrétisation top-down avec "RWeka"

Assez déçu par ces résultats (qui ont motivé la rédaction de ce tutoriel d'ailleurs), à la fois en termes de temps de calcul et de qualité de partitionnement, j'ai regardé s'il n'y avait pas d'autres packages qui proposeraient la méthode de Fayyad et Irani (1993). Bien m'en a pris, j'ai découvert qu'elle est également présente dans le package RWeka.

L'appel de la fonction est un peu différent, mais l'algorithme de discrétisation est bien appliqué à l'ensemble des variables quantitatives qui composent la base WAVE.

```
#MDL de RWeka
library(RWeka)
print(system.time(waveMdlpWeka <- Discretize(onde ~ ., data = wave)))

##    user  system elapsed
##    5.78    0.17    5.31
```

Ah ? L'implémentation est largement plus rapide avec 5.78 secondes. Je m'en doutais un peu j'avoue après avoir inspecté le code de "discretization". Il y avait matière à faire mieux. Après, est-ce que nous avons exactement les mêmes résultats ? Croisons-les pour le vérifier.

Pour ce faire, nous construisons un tableau de contingence entre chaque paire de variables recodées par les deux fonctions. Nous calculons le ratio entre la somme des éléments diagonaux et la somme totale. S'il est égal à 1, cela veut que toutes les valeurs sont bien regroupées sur la diagonale, les découpages sont cohérents….

```
#croisement des variables discrétisées
for (j in 1:21){
  m <- table(waveMdlp$Disc.data[,j],waveMdlpWeka[,j])
  ratio <- sum(diag(m))/sum(m)
  print(ratio)
}

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

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

… et c'est le cas. Heureusement, ça ferait désordre si deux packages censés programmer le même algorithme produisaient des résultats dissemblables.

Les mêmes intervalles donc, mais avec une efficacité bien différente. Le gap entre les temps d'exécution laisse pantois, **RWeka est 22 fois plus rapide** (127.58 / 5.78).

3.4 Discrétisation top-down basé sur un arbre rpart()

Voyons maintenant ce que donne l'approche basée sur `rpart()` pour le traitement de la base WAVE.

```
#notre algo
print(system.time(waveMine <- mydiscDataFrame.fit(wave)))

##    user  system elapsed
##    1.17    0.19    1.36
```

Le temps de traitement est tout autre (**1.17** secondes, presque 5 fois plus rapide que RWeka). Mais bon, il serait injuste de l'opposer directement aux implémentations de la méthode MDLP, les solutions obtenues étant différentes. Nous disposons d'un découpage plus sobre, avec moins d'intervalles, pour la plupart des variables :

```
#bornes
print(waveMine$cutp)

## [[1]]
## [1] -Inf  Inf
##
## [[2]]
```

```
## [1] -Inf  Inf
##
## [[3]]
## [1]  -Inf 0.725   Inf
##
## [[4]]
## [1]  -Inf 0.875   Inf
##
## [[5]]
## [1]  -Inf 1.085   Inf
##
## [[6]]
## [1]  -Inf 1.695   Inf
##
## [[7]]
## [1]  -Inf 2.415   Inf
##
## [[8]]
## [1]  -Inf 2.725   Inf
##
## [[9]]
## [1]  -Inf 2.875   Inf
##
## [[10]]
## [1]  -Inf 2.955   Inf
##
## [[11]]
## [1]  -Inf 3.015   Inf
##
## [[12]]
## [1]  -Inf 3.125   Inf
##
## [[13]]
## [1]  -Inf 2.935   Inf
##
## [[14]]
## [1]  -Inf 2.785   Inf
##
## [[15]]
## [1]  -Inf 2.415   Inf
##
## [[16]]
## [1]  -Inf 1.565   Inf
##
## [[17]]
## [1]  -Inf 1.035   Inf
##
## [[18]]
## [1]  -Inf 0.835   Inf
##
## [[19]]
## [1]  -Inf 0.635   Inf
##
## [[20]]
## [1] -Inf  Inf
```

```
##
## [[21]]
## [1] -Inf  Inf
```

Ces résultats semblent plus en accord avec la représentation visuelle des distributions conditionnelles ci-dessus. Mais je me garderais quand-même de tirer des conclusions définitives à partir d'une seule expérimentation. Je dirais surtout qu'il est tout à fait possible d'utiliser une fonction implémentant un algorithme d'arbre de décision – `rpart()` en l'occurrence – pour instancier une discrétisation supervisée. Et, outre l'avantage en rapidité d'exécution, les solutions (nombre d'intervalles, identification des bornes de découpage) ne sont pas illégitimes.

Enfin, contrairement à MDLP, l'avantage ici est qu'il est possible d'affiner les résultats en modifiant les paramètres d'analyse (minsplit, minbucket, maxdepth, cp) si d'aventure ils ne sont pas conformes à ce que nous espérions.

4 Déploiement sur un nouveau dataset

Reste un dernier problème. Comment appliquer le recodage sur une seconde base composée des mêmes variables ? Cela peut être utile par exemple si l'on calcule les seuils de discrétisation sur une base d'apprentissage et qu'on souhaite les appliquer sur une base test dans un schéma d'apprentissage-test en analyse prédictive.

Nous chargeons une seconde version de la base WAVE avec les 21 descripteurs et 50.000 observations.

```
#fichier de déploiement
waveBis <- read.csv("wave50k_deploiement.txt",header=T,sep="\t")
print(str(waveBis))

## 'data.frame':    50000 obs. of  21 variables:
##  $ v01: num  1.39 -1.72 -0.22 1.87 0.27 2.23 0.98 -2.56 -0.07 -0.3 ...
##  $ v02: num  -0.81 0.89 -0.65 1.11 0.99 -0.81 0.07 0.69 0.64 0.16 ...
##  $ v03: num  2.96 1.34 1.95 1.74 3.13 -0.53 0.88 -0.99 -0.1 1.62 ...
##  $ v04: num  4 1.18 3.95 1.09 3.59 2.59 0.71 2.65 -0.12 1.72 ...
##  $ v05: num  3.39 1.1 1.99 -0.75 2.87 2.41 0.45 2.4 -0.06 -0.07 ...
##  $ v06: num  4.77 3.02 4.33 3.65 6.44 3.89 2.03 3.39 -0.08 0.38 ...
##  $ v07: num  4.82 3.44 3.98 2.7 7.57 0.99 3.62 5.7 -0.1 -1.97 ...
##  $ v08: num  4.08 2.58 4.01 2.08 4.93 3.88 4.26 4.3 0.76 -0.32 ...
##  $ v09: num  4.08 5.06 4.89 1.64 3.39 1.74 2.95 3.36 -0.21 -1.35 ...
##  $ v10: num  3.22 4.3 2.21 1.98 3.56 1.91 5.08 1.75 0.15 2.15 ...
##  $ v11: num  1.59 5.17 0.18 2.94 2.86 2.64 4.48 1.21 2.53 1.43 ...
```

```
##  $ v12: num  1.62 1.21 2.25 0.85 1.81 1.77 5.01 0.81 3.57 4.02 ...
##  $ v13: num  0.6 2.61 -0.01 0.64 0.07 1.03 4.64 0.52 3.52 3.36 ...
##  $ v14: num  2.59 2.07 1.77 3.18 -0.69 1.68 3.37 0.33 5.81 5.37 ...
##  $ v15: num  -0.41 2.11 2.12 2.48 0.94 3.32 2.85 0.77 4.24 5.83 ...
##  $ v16: num  3.06 0.19 2.79 0.47 -1.1 4.15 0.11 0.12 4.21 3.71 ...
##  $ v17: num  -1.4 0.95 1.5 2.8 1.84 3.28 -0.48 1.29 4.36 0.41 ...
##  $ v18: num  0.73 0.86 1.66 2.91 -0.51 2.6 1.46 0.14 3.15 4.14 ...
##  $ v19: num  -0.08 1.4 0.14 2.03 0.13 0.67 0.98 1.25 0.34 1.65 ...
##  $ v20: num  0.17 -1.41 0.62 -1.06 0.77 0.51 -0.67 -0.2 3.05 -0.82 ...
##  $ v21: num  0.37 -0.63 1.13 -1.08 2.18 0.43 -0.82 1.12 1.09 -0.27 ...
```

Nous écrivons une fonction qui prend en entrée la base à traiter et les résultats issus du processus de discrétisation de la section précédente.

```
#fonction pour le déploiement de la discrétisation
mydiscDataFrame.transform <- function(objetDisc,newdata){
  #nombre de variables à traiter
  p <- length(objetDisc$cutp)
  #vérifier les noms des variables
  ok <- (sum(colnames(objetDisc$disc.data)[1:p] == colnames(newdata)[1:p]) == p)
  #si pas ok on arrête
  if (!ok){
    stop("Variables non cohérentes")
  }
  #on continue sinon
  resdata <- lapply(1:p,function(j){return(cut(newdata[,j],objetDisc$cutp[[j]],labels=FALSE))})
  #transformer en data.frame
  resdata <- data.frame(resdata)
  #renommer les variables
  colnames(resdata) <- colnames(newdata)[1:p]
  #et return
  return(list(codec=objetDisc$cutp,disc.data=resdata))
}
```

Après quelques vérifications, la cohérence des noms de variables notamment, nous appliquons la fonction `cut()` en lui passant les bornes de découpage. Nous renvoyons les seuils utilisés et les variables nouvellement recodées.

Appliquons la fonction sur WAVEBIS.

```
#encoder waveBis à partir des bornes calculés sur wave
print(system.time(waveBisDisc <-
mydiscDataFrame.transform(objetDisc=waveMine,newdata=waveBis)))
```

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

L'opération est quasi-instantanée. On le comprend aisément puisqu'il n'y a plus de paramètres à calculer à partir des données.

Nous avons un aperçu des variables recodées de la nouvelle base.

```
#variables recodées en déploiement
print(str(waveBisDisc$disc.data))

## 'data.frame':    50000 obs. of  21 variables:
##  $ v01: int  1 1 1 1 1 1 1 1 1 1 ...
##  $ v02: int  1 1 1 1 1 1 1 1 1 1 ...
##  $ v03: int  2 2 2 2 2 1 2 1 1 2 ...
##  $ v04: int  2 2 2 2 2 2 1 2 1 2 ...
##  $ v05: int  2 2 2 1 2 2 1 2 1 1 ...
##  $ v06: int  2 2 2 2 2 2 2 2 1 1 ...
##  $ v07: int  2 2 2 2 2 1 2 2 1 1 ...
##  $ v08: int  2 1 2 1 2 2 2 2 1 1 ...
##  $ v09: int  2 2 2 1 2 1 2 2 1 1 ...
##  $ v10: int  2 2 1 1 2 1 2 1 1 1 ...
##  $ v11: int  1 2 1 1 1 1 2 1 1 1 ...
##  $ v12: int  1 1 1 1 1 1 2 1 2 2 ...
##  $ v13: int  1 1 1 1 1 1 2 1 2 2 ...
##  $ v14: int  1 1 1 2 1 1 2 1 2 2 ...
##  $ v15: int  1 1 1 2 1 2 2 1 2 2 ...
##  $ v16: int  2 1 2 1 1 2 1 1 2 2 ...
##  $ v17: int  1 1 2 2 2 2 1 2 2 1 ...
##  $ v18: int  1 2 2 2 1 2 2 1 2 2 ...
##  $ v19: int  1 2 1 2 1 2 2 2 1 2 ...
##  $ v20: int  1 1 1 1 1 1 1 1 1 1 ...
##  $ v21: int  1 1 1 1 1 1 1 1 1 1 ...
```

5 Conclusion

L'idée de ce tutoriel me vient d'une discussion que j'ai eu avec mes étudiants il y a quelques années de cela. Je disais en substance que les méthodes top-down étaient les plus efficaces pour discrétiser sur des gros volumes. Une réponse tranchée fusait alors du fond de la salle "oui, mais ça a planté sous R". Difficile de discuter après ce constat implacable.

Et pourtant, si, justement, il faut en parler. On confond souvent les algorithmes et leurs implémentations. Ils sont liés bien sûr. Mais il se peut qu'un algorithme soit mal programmé, laissant à penser - à tort - que la méthode est inefficace. De même, il ne faut pas confondre R et les librairies qui l'accompagnent. Tous les packages ne sont pas de la même qualité. Il nous appartient de les vérifier.

Dans ce tutoriel, nous étudions le comportement des méthodes top-down en discrétisation supervisée. D'une part, nous avons constaté que la méthode Fayyad et Iran (1993) peut être implémentée avec plus ou moins d'efficacité en comparant les durées d'exécution des

fonctions de “discretization” et “RWeka”. D’autre part, en y regardant de plus près, on se rend compte que l’approche s’apparente à un algorithme d’induction d’arbre de décision appliqué à une seule et unique variable à discrétiser. S’inspirant de cette idée, nous avons essayé d’instancier une discrétisation top-down s’appuyant sur la fonction `rpart()` du package du même nom, bénéficiant ainsi du travail d’optimisation réalisé par leurs auteurs. Les résultats, tant en termes de pertinence du partitionnement qu’en rapidité des calculs, montrent que la piste est tout à fait viable et constitue une alternative tout à fait crédible face aux packages spécialisés sus-cités.

6 Références

Fayyad, U. M., Irani, K. B., ‘’Multi-interval discretization of Continuous-valued Attributes for Classification Learning’’, Artificial Intelligence, 1022-1027, 1993.

Hornik, K., Butcha, C., Hothorn, T., Karatzoglou, A., Meyer, D., Zeilis, A., ‘’RWeka: R/Weka Interface’’, 2019, https://cran.r-project.org/package=RWeka

HyunJi, K., ‘’discretization: Data preprocessing, discretization for classification’’, 2012, https://cran.r-project.org/package=discretization

Kerber, R., ‘’ChiMerge: Discretization of numeric attributes’’, in Proceedings of the Tenth National Conference on Artificial Intelligence, 123-128, 1992.

Tutoriel Tanagra, ‘’La discrétisation des variables quantitatives’’, octobre 2014, http://tutoriels-data-mining.blogspot.com/2014/10/la-discretisation-des-variables.html

Tutoriel Tanagra, ‘’Discrétisation – Comparaison de logiciels’’, février 2010, http://tutoriels-data-mining.blogspot.com/2010/02/discretisation-comparaison-de-logiciels.html