Introduction à R

Manipulations de données

Pascal Bessonneau

06/2015

A ne pas faire (ou avec prudence)

R permet de concaténer des lignes (*rbind*), des colonnes (*cbind*), des *data.frames* (mêmes fonctions) ensemble. Toutefois il convient d'utiliser avec sagesse cette fonctionnalité.
Si en sciences expérimentales faire une fusion de table avec une simple concaténation est très pratique, cette opération n'est pas raisonnable sur des tables plus complexes et surtout sur des tables contenant des identifiants qui permettent de réaliser une fusion plutôt qu'une concaténation.

A ne pas faire (ou avec prudence)

En tout cas dès que *cbind* est utilisé il faut vérifier :

- que les deux tableaux ont la même taille
- chaque ligne identifie une observation
- que les observations sont strictement dans le même ordre dans les deux tableaux

A ne pas faire (ou avec prudence)

En tout cas dès que *rbind* est utilisé il faut vérifier :

- que le nombre de colonnes sont identiques
- que le type de chaque colonnes sont identiques

rbind est un peu plus sûr car R généralement refuse d'opérer en cas de différence de noms et/ou de types de variables dans les deux tableaux de données.

A ne pas faire (ou avec prudence)

rbind s'avère quand même pratique si on souhaite travailler par exemple sur une base public et une privé... et réassembler le tout à la fin du traitement.
C'est typiquement le cas par exemple quand on utilise *split*.

Fusion avec une seule variable

Ce cas est en fait beaucoup plus fréquent qu'il n'y parait. On veux par exemple ajouter une variable avec une couleur pour les graphiques, le nombre d'élèves dans l'établissement, ...
Et ce type de fusion se fait avec un vecteur.

```
> couleurs <- c( "red", "green", "blue" )
> names(couleurs) <- levels(iris$Species)
> iris$couleur <- couleurs[as.character(iris$Species)]
> with( iris, table(couleur,Species) )

##         Species
## couleur setosa versicolor virginica
##   blue       0          0        50
##   green      0         50         0
##   red       50          0         0
```

Fusion avec une seule variable

Un exemple numérique, si on veut ajouter la longueur moyenne par espèce pour les orchidées :

```
> longueur_par_spe <- tapply( iris$Sepal.Length, iris$Species, mean )
> iris$Sepal.Length.Moy <- longueur_par_spe[as.character(iris$Species)]
> with( iris, table(Sepal.Length.Moy,Species) )

##                  Species
## Sepal.Length.Moy setosa versicolor virginica
##            5.006     50          0         0
##            5.936      0         50         0
##            6.588      0          0        50
```

Fusions avec merge

La fonction *merge* dans R permet de fusionner des tables avec un identifiant (clef) commun entre les tables.
La fusion peut être réalisée en utilisant des variables *factor* mais il est préférable de les transformer variable *character* avant la fusion.
Les fusions possibles sont des fusions de 1 à 1 ou de 1 à n.

Fusions avec merge

x, y	les 2 *data.frames* que l'on veut fusionner
by	si la variable porte le même nom dans les deux *data.frame*, il suffit de préciser le nom de la variable précédé de *by*
by.x, by.y	dans ce cas on spécifie le nom de la colonne pour *x* (la première *data.frame* et pour *y* (la deuxième).

TABLE – Les principaux arguments de *merge*

Voilà l'essentiel de la fonction.

Fusions avec merge

Il faut noter qu'on a la possibilité de fusionner les tables non pas en utilisant le nom d'une variable de la *data.frame* mais les *row.names*. Dans ce cas, l'argument que l'on passe à *by* est *'row.names'*.

```
> res <- merge( eleves, scores, by="id" )
> dim(res)

## [1] 5000    7
```

Fusions avec merge

Dans le cas de l'utilisation des rownames :

```
> rownames(eleves) <- eleves$id
> rownames(scores) <- scores$id
>
> res <- merge( eleves, scores, by="row.names" )
> dim(res)

## [1] 5000    9
```

Fusions avec merge

Après la fusion, la fonction utile est *dim* qui donne le nombre de lignes et de colonnes :

```
> dim(eleves);dim(scores);dim(res)

## [1] 5000    6
## [1] 5000    2
## [1] 5000    9

> colnames(res)

## [1] "Row.names" "id.x"      "sexe"
## [4] "age3e"     "retard"    "secteur"
## [7] "acad"      "id.y"      "score"
```

Fusions avec merge

La fonction *merge* effectue une jointure naturelle. C'est-à-dire que seules les lignes présentes dans *x* et dans *y* seront présentes dans la *data.frame* finale.
Pour changer ce comportement, il existe trois arguments *all*

all	Si vrai alors toutes les lignes des deux *data.frame* seront conservées dans la *data.frame* finale.
all.x	Si *TRUE* alors toutes les lignes de la *data.frame x* seront conservées dans la *data.frame* finale. Les lignes de *y* ne trouvant pas de correspondance seront éliminées.
all.y	Si *TRUE* alors toutes les lignes de la *data.frame x* seront conservées dans la *data.frame* finale. Les lignes de *y* ne trouvant pas de correspondance seront éliminées.

TABLE – Le type de jointure

Fusions avec merge

Jointure naturelle

```
> conatif <- read.csv2( "data/evaluation-conatif.csv" )
> conatif$id <- as.character( conatif$id )
> dim(conatif)

## [1] 4987    8

> ec <- merge( eleves, conatif, by="id" )
> dim(ec)

## [1] 4987   13
```

Fusions avec merge

Fusion à gauche

```
> ec <- merge( eleves, conatif, by="id", all.x = T )
> dim(ec)

## [1] 5000   13
```

Fusions avec merge

Si des colonnes de x et de y portent le même nom, les colonnes provenant de x seront suffixés avec x. Et pour y, la colonne sera suffixés par y.
Il est possible de spécifier des suffixes personnalisés plutôt que ces suffixes par défaut avec l'argument *suffixes*.

Fusions avec merge

Il attends un vecteur *character* de longueur 2 comme par exemple...

```
> res <- merge( eleves, scores, by="row.names",
+               suffixes=c(".eleves",".scores" )
+               )
> dim(res)

## [1] 5000    9

> colnames(res)

## [1] "Row.names" "id.eleves" "sexe"
## [4] "age3e"     "retard"    "secteur"
## [7] "acad"      "id.scores" "score"
```

Fusions avec merge

Pour trouver les lignes qui n'ont pas été importées...
La syntaxe et très simple et fait appel à l'opérateur *%in%*.
Ici on cherche les lignes, de *eleves* pour lesquelles il n'y a pas de données pour la partie conative.

```
> res <- merge( eleves, conatif, by="id", all.x=T )
> (perdus <- res$id[ !(res$id %in% conatif$id) ])

##  [1] "e014161" "e03592"  "e041612" "e044139"
##  [5] "e1123"   "e121165" "e151894" "e162289"
##  [9] "e184897" "e213974" "e242770" "e243719"
## [13] "e251862"
```

Fusions avec merge

Il est possible de spécifier un vecteur de noms de variables pour l'argument *by*.
Mais les identifiants composite ne sont pas conseillés (dans l'absolu).

R langage fonctionnel

R est un langage fonctionnel. Si cela signifie que "tout est fonction dans R", cela signifie également qu'il faut privilégier le traitement des vecteurs au détriment des boucles.
Au début cela peut paraître contre-intuitif mais cela permet souvent de gagner en vitesse d'exécution, en possibilité de rendre le calcul parallèle et en lisibilité (si si...).

R langage fonctionnel

Par exemple, sur un vecteur, il doit vous paraître évident que :

```
> x <- 1:4
> x*4

## [1]  4  8 12 16

> # n'est autre que l'équivalent implicite de
> vreponse <- c()
> for ( ii in 1:length(x) ) vreponse <- c( vreponse, x[ii]*4 )
> vreponse

## [1]  4  8 12 16
```

R langage fonctionnel

L'utilisation et la production de statistiques va en grande partie utilisé ce principe illustré ici par un vecteur mais qui est utilisé dans les fonctions de type apply sur des objets plus complexes.
Ici on utilise l'opérateur de multiplication qui est une fonction parmi d'autres.

```
> "*"(3,4)

## [1] 12
```

R langage fonctionnel

Cela oblige à savoir utiliser les fonctions sous R. La définition se fait avec la syntaxe suivante :

```
> mafonction <- function ( arg1, arg2, arg3=F, ... ) {
+   # code
+ }
```

R langage fonctionnel

Mais souvent, dans les opérations de manipulations de données, des fonctions *anonymes* seront utilisées.
C'est-à-dire directement des fonctions : sans nom, jetables.

R langage fonctionnel

Cela ressemble à ça par exemple :

```
> apply(iris[1:4],2,function(x){
+   c(mean(x),sd(x))
+ })

##      Sepal.Length Sepal.Width Petal.Length
## [1,]    5.8433333   3.0573333     3.758000
## [2,]    0.8280661   0.4358663     1.765298
##      Petal.Width
## [1,]   1.1993333
## [2,]   0.7622377
```

Considérations sur les aggrégations

Contrairement à d'autres logiciels, R peut paraître strict voire pénible lors des aggrégations. En fait, la pratique de R permet de réaliser que R impose cette syntaxe notamment pour éviter de réaliser des regroupements n'ayant pas de sens.
Une exemple simple, cette requête SQL peut tout à fait renvoyer un résultat valide :

```
SELECT *, uai FROM base_eleves GROUP BY uai ;
```

Hors le sexe de l'élève, présent dans la ligne élève, va devenir une variable vide de sens. En effet elle a un sens au niveau individuel mais pas au niveau d'un établissement.

Considérations sur les aggrégations

R va rendre difficile ce type d'aggrégation.
L'aggrégation ne sera possible que si on obtient un vecteur cohérent avant aggrégation.

Pourquoi aggréger ?

Il existe de nombreuses façons d'aggréger des données sous R.
L'utilisation de chacune dépend des goûts de chacun et surtout de la finalité de l'aggrégation.
Par exemple, l'aggrégation peut servir à...

- créer un enregistrement pour constituer un unité plus grande que celle d'origine (ex : passer élève à établissement)
- créer des statistiques pour des unités plus importantes (ex : établissement, pays, ...)
- ...

Aggrégations statistiques

Beaucoup de statistiques peuvent réalisées avec certaines fonctions de R qui appartiennet à la famille *apply*.
Par exemple, *tapply* permet de réaliser des regroupements en fonction d'une ou plusieurs variables en calculant des statistiques sur une variable.

```
> res <- with( xtfme, tapply( vali_f, num_etab, mean, na.rm=T ) )
> res[1:5]

##  0010529V  0010560D  0011110B  0011238R  0011289W
## 0.9500000 0.9444444 0.6521739 0.8709677 0.8750000
```

Aggrégations statistiques

Dans le cas précédent, on demande la moyenne (vecteur de longueur 1) et un variable de regroupement. Mais *tapply* permet de faire des choses plus complexes. Dans ce cas, il y a un résultat par croisement de modalité. Ce qui donne un tableau.

```
> with( xtfme, tapply(
+   vali_f,
+   list( strate=strate, sexe=sexe ),
+   mean,
+   na.rm=T
+   )
+ )

##       sexe
## strate         1         2
##      1 0.8695652 0.9317269
##      2 0.7598647 0.8289183
##      3 0.6610360 0.7807487
##      4 0.8760246 0.9536935
```

Aggrégations statistiques

C'est la limite (ou la puissance) de tapply. On peut ainsi s'amuser à obtenir des tableaux à k dimensions pour k variables de regroupement.
Inversement on peut être limité par le nombre de valeurs renvoyées par la fonction de calcul.

```
> with( xtfme, tapply(
+   vali_f,
+   list(strate=strate),
+   function(x,na.rm=T) {
+     c( mean(x,na.rm=na.rm), sd(x,na.rm=na.rm) )
+     }
+   )
+ )
```

Aggrégations statistiques

```
## $`1`
## [1] 0.9003984 0.2995427
##
## $`2`
## [1] 0.7947574 0.4039915
##
## $`3`
## [1] 0.7224355 0.4479202
##
## $`4`
## [1] 0.9134360 0.2812698
```

Aggrégations statistiques

Deux illustrations pour calculer le nombre d'élèves dans chaque strate :

```
> with(xtfme, tapply( rep(1,length(strate)), strate, sum) )

##    1    2    3    4
## 2008 1793 1823 1883

> with(xtfme, tapply( num_etab, strate, length) )

##    1    2    3    4
## 2008 1793 1823 1883
```

et pour les poids. . .

```
> with(xtfme, tapply( poids, strate, sum) )

##      1      2      3      4
## 5220.8  537.9  182.3  941.5
```

Aggrégations statistiques

La fonction *aggregate* permet des choses similaires ou un peu plus complexes.

```
> with( xtfme, aggregate(
+   cbind(vali_f, vali_m),
+   list(strate=strate),
+   mean
+   )
+ )

##   strate    vali_f    vali_m
## 1      1 0.9003984 0.9213147
## 2      2 0.7947574 0.8315672
## 3      3 0.7224355 0.7761931
## 4      4 0.9134360 0.9373340
```

Aggrégations statistiques

```
> with( xtfme, aggregate(
+   cbind(vali_f, vali_m),
+   list(strate=strate,sexe=sexe),
+   mean
+   )
+ )

##   strate sexe    vali_f    vali_m
## 1      1    1 0.8695652 0.9268775
## 2      2    1 0.7598647 0.8410372
## 3      3    1 0.6610360 0.7815315
## 4      4    1 0.8760246 0.9344262
## 5      1    2 0.9317269 0.9156627
## 6      2    2 0.8289183 0.8222958
## 7      3    2 0.7807487 0.7711230
## 8      4    2 0.9536935 0.9404631
```

Aggrégations statistiques

Les indications de variables peuvent se faire en formule.

```
> aggregate(
+   vali_m ~ strate + sexe,
+   data=xtfme, mean
+   )

##   strate sexe    vali_m
## 1      1    1 0.9268775
## 2      2    1 0.8410372
## 3      3    1 0.7815315
## 4      4    1 0.9344262
## 5      1    2 0.9156627
## 6      2    2 0.8222958
## 7      3    2 0.7711230
## 8      4    2 0.9404631
```

Aggrégations statistiques

Ou comme dans l'aide...

```
> data(iris)
> aggregate( . ~ Species, data=iris, mean, na.rm=T  )

##      Species Sepal.Length Sepal.Width
## 1     setosa        5.006       3.428
## 2 versicolor        5.936       2.770
## 3  virginica        6.588       2.974
##   Petal.Length Petal.Width
## 1        1.462       0.246
## 2        4.260       1.326
## 3        5.552       2.026
```

plyr

Un paquet de Hadley Wickham, *plyr*, permet de réaliser ce type d'opérations assez facilement.
Le paquet *plyr* permet de traiter des *array*, des vecteurs, des *data.frames*.
Il offre des fonctions génériques permettant de créer, transformer ou faire des calculs sur des *data.frames*.
Outre ce côté générique, il offre quelques avantages sur les fonctions de base.

plyr

Par exemple les statistiques sur une variable deviennent :

```
> ddply( xtfme, .(strate), summarize,
+         moy_f=mean(vali_f), sd_f=sd(vali_f),
+         moy_m=mean(vali_m), sd_m=sd(vali_m)
+         )

##   strate     moy_f       sd_f     moy_m       sd_m
## 1      1 0.9003984 0.2995427 0.9213147 0.2693140
## 2      2 0.7947574 0.4039915 0.8315672 0.3743546
## 3      3 0.7224355 0.4479202 0.7761931 0.4169085
## 4      4 0.9134360 0.2812698 0.9373340 0.2424255
```

On récupère une *data.frame* ...

plyr

Idem. . .

```
> head(
+   ddply( xtfme, .(strate,sexe), summarize,
+         moy_f=mean(vali_f), sd_f=sd(vali_f),
+         moy_m=mean(vali_m), sd_m=sd(vali_m)
+         ), 4
+ )

##   strate sexe     moy_f       sd_f     moy_m
## 1      1    1 0.8695652 0.3369477 0.9268775
## 2      1    2 0.9317269 0.2523407 0.9156627
## 3      2    1 0.7598647 0.4274065 0.8410372
## 4      2    2 0.8289183 0.3767883 0.8222958
##         sd_m
## 1 0.2604662
## 2 0.2780327
## 3 0.3658477
## 4 0.3824747
```

plyr

etc. . .

aggrégation personnalisée

Une fonction d'aggrégation complètement personnalisée par exemple... avec les fonctions classiques

```
> agg <- lapply( split(xtfme,xtfme$num_etab), function(x) {
+       data.frame(
+         uai=unique(x$num_etab), vali_f=mean(x$vali_f),
+         vali_m=mean(x$vali_m), poids=sum(x$poids),
+         prop_garcons=mean(ifelse(x$sexe==1,0,1))
+       )
+     }
+   )
> head( do.call( rbind, agg), 3 )
```

aggrégation personnalisée

```
##                uai    vali_f    vali_m poids
## 0010529V 0010529V 0.9500000 0.9500000  52.0
## 0010560D 0010560D 0.9444444 0.9444444  46.8
## 0011110B 0011110B 0.6521739 0.8260870   2.3
##          prop_garcons
## 0010529V    0.4500000
## 0010560D    0.5000000
## 0011110B    0.5217391
```

aggrégation personnalisée

```
##                uai    vali_f    vali_m poids
## 0010529V 0010529V 0.9500000 0.9500000  52.0
## 0010560D 0010560D 0.9444444 0.9444444  46.8
## 0011110B 0011110B 0.6521739 0.8260870   2.3
##          prop_garcons
## 0010529V    0.4500000
## 0010560D    0.5000000
## 0011110B    0.5217391
```

aggrégation personnalisée

Quelque chose de discutable d'un point de vue méthodologique mais possible...

```
> elev <- merge(scores,eleves,by="id")
> elev$sexe <- as.character(elev$sexe)
> elev <- elev[elev$sexe!="M",]
>
> coef <- function(score,age3e,n) { coef(lm( score ~ age3e ))[n] }
>
> res <- ddply(
+   elev, .( , secteur ), summarize,
+   coef1_age3e = coef1(score,age3e,1),
+   coef2_age3e = coef2(score,age3e,2)
+ )
```

Transposition de matrices

La transposition simple d'une matrice ou d'une data.frame se fait avec la fonction *t* :

```
> (a = matrix( 1:16, nrow=4, ncol=4 ))

##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16

> t(a)

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
```

reshape2

Encore un paquet d'Hadley Wickham...
Il permet de faire un peu se qu'on veut au niveau des transpositions.
Les deux fonctions centrales sont *cast* et *melt*.

melt

Cette fonction permet de passer d'un tableau large à un tableau long...

```
> names(airquality) <- tolower(names(airquality))
> head(airquality)

##   ozone solar.r wind temp month day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

melt

Cette fonction permet de passer d'un tableau large à un tableau long...

```
> head(
+   melt( airquality,
+         id=c("month", "day"),
+         measure.vars=c("ozone"),
+         na.rm=TRUE
+         )
+   )

##   month day variable value
## 1     5   1    ozone    41
## 2     5   2    ozone    36
## 3     5   3    ozone    12
## 4     5   4    ozone    18
## 6     5   6    ozone    28
## 7     5   7    ozone    23
```

melt

On peux utiliser plusieurs variables comme variables de mesure.

```
> head(
+   z <- melt( airquality,
+         id=c("month", "day"),
+         measure.vars=c("wind","ozone"),
+         na.rm=TRUE
+         ), 3
+   )

##   month day variable value
## 1     5   1     wind   7.4
## 2     5   2     wind   8.0
## 3     5   3     wind  12.6

> table(z$variable)

##
##  wind ozone
##   153   116
```

cast

A l'inverse pour passer d'un tableau long à un tableau large...

```
> head(
+   dcast( z, month + day ~ variable )
+   )

##   month day wind ozone
## 1     5   1  7.4    41
## 2     5   2  8.0    36
## 3     5   3 12.6    12
## 4     5   4 11.5    18
## 5     5   5 14.3    NA
## 6     5   6 14.9    28
```

cast

La fonction *(a/d)cast* peut également être utilisé pour réaliser des statistiques...

```
> head(
+   dcast( z, month ~ variable, mean, na.rm=T )
+   )

##   month      wind    ozone
## 1     5 11.622581 23.61538
## 2     6 10.266667 29.44444
## 3     7  8.941935 59.11538
## 4     8  8.793548 59.96154
## 5     9 10.180000 31.44828
```