# analyse-R

# Introduction à l'analyse d'enquêtes avec R et RStudio

Dernière mise à jour : 17 mai 2015

Ce projet de site web est basé sur le support de cours Introduction à l'analyse d'enquêtes avec R de Joseph Larmarange, qui est lui-même une adaptation de l'Introduction à R de Julien Barnier.

# Contributeurs

**Par ordre alphabétique :**

Julien Barnier, Julien Biaudet, Milan Bouchet-Valat, Frédérique Giraud, Joël Gombin, Mayeul Kauffmann, Joseph Larmarange, Nicolas Robette.

**Maintenance :**

Joseph Larmarange — http://joseph.larmarange.net

# Table des matières

**Prise en main**

Présentation et Installation

Premier contact

Premier travail avec des données

Statistique univariée

Import / Export de données

Manipulation de données

### Analyses

- Statistique bivariée
- Régression logistique
- Données pondérées
- Analyse des correspondances multiples (ACM)
- Classification ascendante hiérarchique (CAH)
- Analyse de séquences

### Avancé

- Astuces
- Calculer un âge

# Licence

Le contenu de ce site est diffusé sous licence *Creative Commons Attribution - Pas d'utilisation commerciale - Partage dans les mêmes conditions* (https://creativecommons.org/licenses/by-nc-sa/3.0/fr/).

# Présentation et Installation

## Présentation de R

**R** est un langage orienté vers le traitement de données et l'analyse statistique dérivé du langage **S**. Il est développé depuis une vingtaine d'années par un groupe de volontaires de différents pays. C'est un logiciel libre[1], publié sous licence GNU GPL.

L'utilisation de **R** présente plusieurs avantages :

- c'est un logiciel multiplateforme, qui fonctionne aussi bien sur des sytèmes **Linux**, **Mac OS X** ou **Windows** ;
- c'est un logiciel libre, développé par ses utilisateurs et modifiable par tout un chacun ;
- c'est un logiciel gratuit ;
- c'est un logiciel très puissant, dont les fonctionnalités de base peuvent être étendues à l'aide de plusieurs milliers d'extensions[2] ;
- c'est un logiciel dont le développement est très actif et dont la communauté d'utilisateurs ne cesse de s'élargir ;
- les possibilités de manipulation de données sous **R** sont en général largement supérieures à celles des autres logiciels usuels d'analyse statistique ;
- c'est un logiciel avec d'excellentes capacités graphiques et de nombreuses possibilités d'export ;
- avec *Rmarkdown*[3], il est devenu très aisé de produire des rapports automatisés dans divers format (**Word**, **PDF**, **HTML**, …) ;
- **R** est de plus utilisé dans tous les secteurs scientifiques, y compris dans le domaine des analyses d'enquêtes et, plus généralement, des sciences sociales.

Comme rien n'est parfait, on peut également trouver quelques inconvénients :

- le logiciel, la documentation de référence et les principales ressources sont en anglais. Il est toutefois parfaitement possible d'utiliser **R** sans spécialement maîtriser cette langue ;
- il n'existe pas encore d'interface graphique pour **R** équivalente à celle d'autres logiciels comme **SPSS** ou **Modalisa**. **R** fonctionne à l'aide de *scripts* (des petits programmes) édités et exécutés au fur et à mesure de l'analyse et se rapprocherait davantage de **SAS** dans son utilisation (mais avec une syntaxe et une philosophie très différentes). Ce point, qui peut apparaître comme un gros handicap, s'avère après un temps d'apprentissage être un mode d'utilisation d'une grande souplesse ;
- comme **R** s'apparente davantage à un langage de programmation qu'à un logiciel proprement dit, la courbe d'apprentissage peut être un peu « raide », notamment pour ceux n'ayant jamais programmé auparavant.

Il est à noter que le développement autour de **R** a été particulièrement actif ces dernières années. On trouvera dès lors aujourd'hui de nombreuses extensions permettant de se « faciliter la vie » au quotidien, ce qui n'était pas vraiment encore le cas il y a 5 ans.

# Philosophie de R

Quelques points particuliers dans le fonctionnement de **R** peuvent parfois dérouter les utilisateurs habitués à d'autres logiciels :

- Sous **R**, en général, on ne voit pas directement les données sur lesquelles on travaille ; on ne dispose pas en permanence d'une vue des données sous forme de tableau[4], comme sous **Modalisa ou SPSS**. Ceci peut être déroutant au début, mais on se rend vite compte qu'on n'a pas besoin de voir en permanence les données pour les analyser.
- Alors qu'avec la plupart des logiciels on réfléchira avec un fichier de données ouvert à la fois, chaque fichier de données correspondra à un objet différent, permettant de manipuler très facilement plusieurs objets à la fois (par exemple dans le cadre de fusion de tables^[Voir par exemple la section dédiée à ce sujet dans le chapitre sur la manipulation de données.).
- Avec les autres logiciels, en général la production d'une analyse génère un grand nombre de résultats de toutes sortes dans lesquels l'utilisateur

est censé retrouver et isoler ceux qui l'intéressent. Avec **R**, c'est l'inverse : par défaut l'affichage est réduit au minimum, et c'est l'utilisateur qui demande à voir des résultats supplémentaires ou plus détaillés.
- Sous **R**, les résultats des analyses sont eux aussi stockés dans des objets et sont dès lors manipulables

Inhabituel au début, ce fonctionnement permet en fait assez rapidement de gagner du temps dans la conduite des analyses.

# Présentation de RStudio

L'interface de base de **R** est assez rudimentaire (voir figure ci-après).

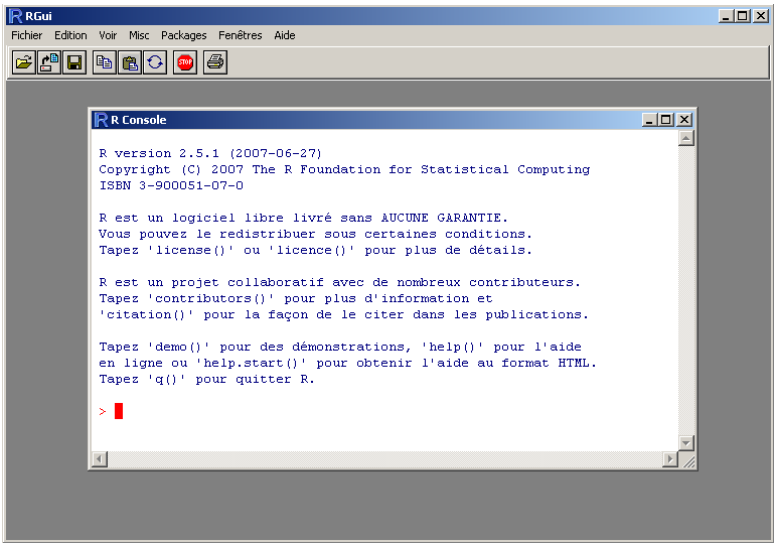



*Figure 1. Interface de R sous Windows*

**RStudio** est un environnement de développement intégré libre, gratuit, et qui fonctionne sous **Windows**, **Mac OS X** et **Linux**. Il complète **R** et fournit un éditeur de script avec coloration syntaxique, des fonctionnalités pratiques d'édition et d'exécution du code (comme l'autocomplétion), un affichage simultané du code, de la console R, des fichiers, graphiques et pages d'aide, une gestion des extensions, une intégration avec des systèmes de contrôle de versions comme **git**, etc. Il intègre de base divers outils comme par exemple la production de rapports au format **Rmarkdown**. Il est en développement actif et

de nouvelles fonctionnalités sont ajoutées régulièrement. Son principal défaut est d'avoir une interface uniquement anglophone.

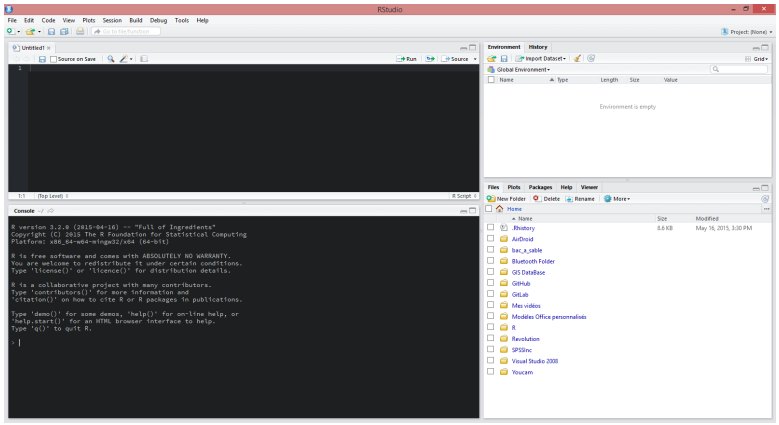



*Figure 2. Interface de RStudio sous Windows*

Pour une présentation plus générale de **RStudio** on pourra se référer au site du projet : http://www.rstudio.com/.

**RStudio** peut tout à fait être utilisé pour découvrir et démarrer avec **R**. Les différents chapitres d'**analyse-R** partent du principe que vous utilisez **R** avec **RStudio**. Cependant, à part les éléments portant sur l'interface de **RStudio**, l'ensemble du code et des fonctions **R** peuvent être utilisés directement dans **R**, même en l'absence de **RStudio**.

La documentation de **RStudio** (en anglais) est disponible en ligne à https://support.rstudio.com. Pour être tenu informé des dernières évolutions de **RStudio**, mais également de plusieurs extensions développées dans le cadre de ce projet, vous pouvez suivre le blog dédié http://blog.rstudio.org/.

# Installation

Il est préférable de commencer par installer **R** avant d'installer **RStudio**.

## Installation de R

Pour une installation sous **Windows**, on se rendra sur cette page : http://cran.r-project.org/bin/windows/base/ et l'on suivra le premier lien pour télécharger le programme d'installation. Une fois le programme d'installation lancé, il suffira d'installer **R** avec les options par défaut[5].

Pour **Mac OS X**, les fichiers d'installation sont disponibles à http://cran.r-project.org/bin/macosx/.

Si vous travaillez sous **Linux**, vous devriez pouvoir trouver **R** via votre gestionnaire de paquets, cela pouvant dépendre d'une distribution de **Linux** à une autre.

## Installation de RStudio

Une fois **R** correctement installé, rendez-vous sur http://www.rstudio.com/products/rstudio/download/ pour télécharger la dernière version stable de **RStudio**. Plus précisément, il s'agit de l'édition *Open Source* de **RStudio Desktop** (en effet, il existe aussi une version serveur).

Choisissez l'installateur correspondant à votre système d'exploitation et suivez les instructions du programme d'installation.

Si vous voulez tester les dernières fonctionnalités de **RStudio**, vous pouvez télécharger la version de développement (plus riche en fonctionnalité que la version stable, mais pouvant contenir des bugs) sur http://www.rstudio.com/products/rstudio/download/preview/.

# Installer des extensions

## Présentation

L'installation par défaut du logiciel **R** contient le cœur du programme ainsi qu'un ensemble de fonctions de base fournissant un grand nombre d'outils de traitement de données et d'analyse statistiques.

**R** étant un logiciel libre, il bénéficie d'une forte communauté d'utilisateurs qui peuvent librement contribuer au développement du logiciel en lui ajoutant des fonctionnalités supplémentaires. Ces contributions prennent la forme d'extensions (*packages*) pouvant être installées par l'utilisateur et fournissant alors diverses fonctions supplémentaires.

Il existe un très grand nombre d'extensions (plus de 6500 à ce jour), qui sont diffusées par un réseau baptisé **CRAN** (*Comprehensive R Archive Network*).

La liste de toutes les extensions disponibles sur **CRAN** est disponible ici : http://cran.r-project.org/web/packages/.

Pour faciliter un peu le repérage des extensions, il existe un ensemble de regroupements thématiques (économétrie, finance, génétique, données spatiales…) baptisés *Task views* : http://cran.r-project.org/web/views/.

On y trouve notamment une *Task view* dédiée aux sciences sociales, listant de nombreuses extensions potentiellement utiles pour les analyses statistiques dans ce champ disciplinaire : http://cran.r-project.org/web/views/SocialSciences.html.

## Installation depuis CRAN

L'installation d'une extension se fait par la fonction `install.packages` {utils}, à qui on fournit le nom de l'extension. Par exemple, si on souhaite installer l'extension **ade4** :

install.packages("ade4", dep = TRUE)

L'option `dep=TRUE` indique à R de télécharger et d'installer également toutes les extensions dont l'extension choisie dépend pour son fonctionnement.

Sous **RStudio**, on pourra également cliquer sur *Install* dans l'onglet *Packages* du quadrant inférieur droit.

Une fois l'extension installée, elle peut être appelée depuis la console ou un fichier script avec la fonction `library` ou la fonction `require` :

library(ade4)

À partir de là, on peut utiliser les fonctions de l'extension, consulter leur page d'aide en ligne, accéder aux jeux de données qu'elle contient, etc.

Pour mettre à jour l'ensemble des extensions installées, la fonction `update.packages` suffit :

update.packages()

Sous **RStudio**, on pourra alternativement cliquer sur *Update* dans l'onglet *Packages* du quadrant inférieur droit.

Si on souhaite désinstaller une extension précédemment installée, on peut utiliser la fonction `remove.packages` :

remove.packages("ade4")

> **IMPORTANT**
>
> Il est important de bien comprendre la différence entre `install.packages` et `library`. La première va chercher les extensions sur internet et les installe en local sur le disque dur de l'ordinateur. On n'a besoin d'effectuer cette opération qu'une seule fois. La seconde lit les informations de l'extension sur le disque dur et les met à disposition de **R**. On a besoin de l'exécuter à chaque début de session ou de script.

## Installation depuis GitHub

Certains packages sont développés sur GitHub. Dès lors, la version de développement sur **GitHub** peut contenir des fonctions qui ne sont pas encore disponibles dans la version stable disponible sur **CRAN**. Ils arrivent aussi parfois que certains packages ne soient disponibles que sur **GitHub**.

L'installation d'un package depuis **GitHub** est très facile grâce à la fonction `install_github` de l'extension **devtools** (que l'on aura préalablement installée depuis **CRAN** ;-) ).

# Mise à jour de R sous Windows

Pour mettre à jour **R** sous **Windows**, il suffit de télécharger et d'installer la dernière version du programme d'installation.

Petite particularité, la nouvelle version sera installée à côté de l'ancienne version. Si vous souhaitez faire de la place sur votre disque dur, vous pouvez désinstaller l'ancienne version en utilisant l'utilitaire *Désinstaller un programme* de **Windows**.

Lorsque plusieurs versions de **R** sont disponibles, **RStudio** choisit par défaut la plus récente. Il est vous est possible de spécifier à **RStudio** quelle version de **R** utiliser via le menu *Tools* > *Global Options* > *General*.

Petit défaut, les extensions (*packages*) sont installées par défaut sous **Windows** dans le répertoire `Documents de l'utilisateur > R > win-library > x.y` avec `x.y` correspondant au numéro de la version de **R**. Ainsi, si l'on travaillait avec la version 3.0 et que l'on passe à la version 3.2, les extensions que l'on avait sous l'ancienne version ne sont plus disponibles pour la nouvelle version. Une astuce consiste à recopier le contenu du répertoire `3.0` dans le répertoire `3.2`. Puis, on lancera **RStudio** (s'il était déjà ouvert, on le fermera puis relancera) et on mettra à jour l'ensemble des packages, soit avec la fonction, `update.packages` soit en cliquant sur *Update* dans l'onglet *Packages* du quadrant inférieur droit.

---

1. Pour plus d'informations sur ce qu'est un logiciel libre, voir : http://www.gnu.org/philosophy/free-sw.fr.html↩
2. Voir MAJ_LIEN.↩
3. Abordé dans un chapitre dédié, MAJ_LIEN↩
4. On verra qu'il est possible avec **RStudio** de disposer d'une telle vue↩
5. Dans le cas particulier où votre ordinateur est situé derrière un *proxy*, il est préférable de choisir *Options de démarrage personnalisées* lorsque cela vous sera demandé par le programme d'installation, puis *Internet2* lorsqu'on vous demandera le mode de connexion à Internet. Ainsi, **R** utilisera par défaut la configuration internet du navigateur **Internet Explorer** et prendra ainsi en compte les paramètres du *proxy*.↩

# Premier contact

> **NOTE**
>
> Ce chapitre est inspiré de la section *Prise en main* du support de cours Introduction à R réalisé par Julien Barnier.

Une fois **RStudio** lancé, vous devriez obtenir une fenêtre similaire à la figure ci-après.

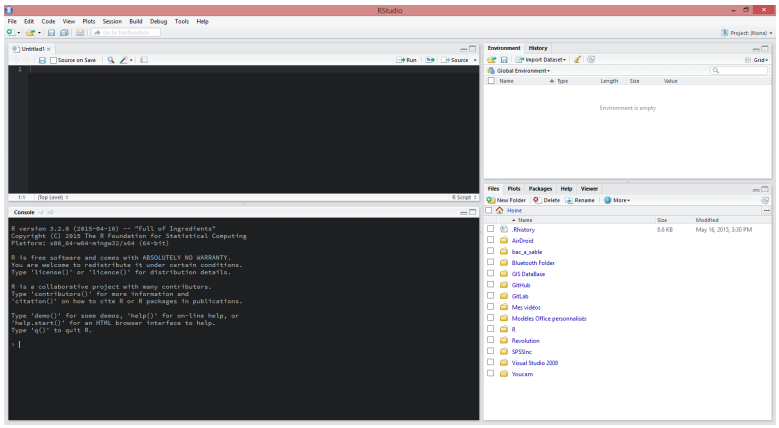



*Figure 1. Interface de RStudio au démarrage*

L'interface de **RStudio** est divisée en quatre quadrants :

- le quadrant supérieur gauche est dédié aux différents fichiers de travail (nous y reviendrons dans le chapitre Premier travail avec les données) ;
- le quadrant inférieur gauche correspond à ce que l'on appelle la *console*, c'est-à-dire à **R** proprement dit ;
- le quadrant supérieur droit permet de connaître
    - la liste des objets en mémoire (onglet *Environment*)
    - ainsi que l'historique des commandes saisies dans la console (onglet *History*) ;
- le quadrant inférieur droit affiche

- la liste des fichiers du répertoire de travail (onglet *Files*),
- les graphiques réalisés (onglet *Plots*),
- la liste des extensions disponibles (onglet *Packages*),
- l'aide en ligne (onglet *Help*)
- et un *Viewer* utilisé pour visualiser certains types de graphiques au format web.

Inutile de tout retenir pour le moment. Nous aborderons chaque outil en temps utile. Pour l'heure, concentrons-nous sur la *console*, c'est-à-dire le quadrant inférieur gauche.

# L'invite de commandes

Au démarrage, la console contient un petit texte de bienvenue ressemblant à peu près à ce qui suit :

```
R version 3.2.0 (2015-04-16) -- "Full of Ingredients"
Copyright (C) 2015 The R Foundation for Statistical
Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain
conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in
publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
```

suivi d'une ligne commençant par le caractère `>` et sur laquelle devrait se trouver votre curseur. Cette ligne est appelée l'*invite de commande* (ou *prompt* en anglais). Elle signifie que **R** est disponible et en attente de votre prochaine commande.

Nous allons tout de suite lui fournir une première commande. Tapez `2 + 3` dans la console et validez avec la touche `Entrée`.

```
R> 2 + 3
[1] 5
```

En premier lieu, vous pouvez noter la convention typographique utilisée dans ce documents. Les commandes saisies dans la console sont indiquées sur un fond gris et précédé de `R>`. Le résultat renvoyé par **R** est quant à lui affiché juste en-dessous sur fond blanc.

Bien, nous savons désormais que **R** sait faire les additions à un chiffre[1]. Nous pouvons désormais continuer avec d'autres opérations arithmétiques de base :

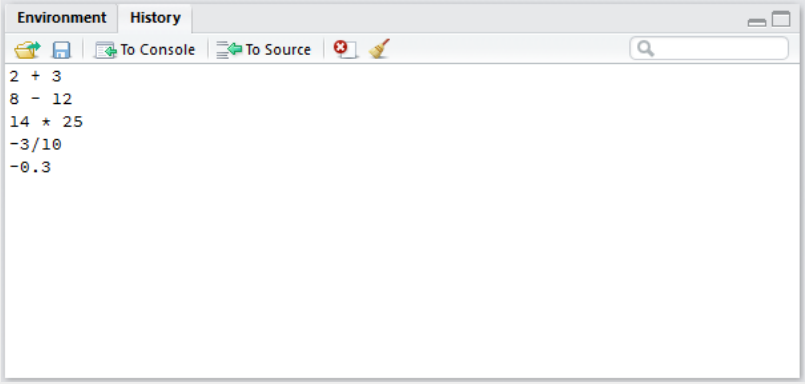
```
R> 8 - 12
[1] -4
R> 14 * 25
[1] 350
R> -3/10
[1] -0.3
R> -0.3
[1] -0.3
```

On remarquera que **R** est anglo-saxon. Les nombres sont donc saisies « à l'anglaise », c'est-à-dire en utilisant le point ( `.` ) comme séparateur pour les décimales.

**ASTUCE**

Une petite astuce très utile lorsque vous tapez des commandes directement dans la console : en utilisant les flèches Haut et Bas du clavier, vous pouvez naviguer dans l'historique des commandes tapées précédemment, que vous pouvez alors facilement réexécuter ou modifier une commande particulière.

Sous **RStudio**, l'onglet History du quadrant haut-droite vous permet de consulter l'historique des commandes que vous avez transmises à **R**.




Un double-clic sur une commande la recopiera automatiquement dans la console. Vous pouvez également sélectionner une ou plusieurs commandes puis cliquer sur *To Console*.

Voir également (en anglais) : http://www.rstudio.com/ide/docs/using/history

Lorsqu'on fournit à **R** une commande incomplète, celui-ci nous propose de la compléter en nous présentant une invite de commande spéciale utilisant les signe `+`. Imaginons par exemple que nous avons malencontreusement tapé sur `Entrée` alors que nous souhaitions calculer `4 * 3` :

```
R> 4 *
```

On peut alors compléter la commande en saisissant simplement `3` :

```
R> 4 *
   + 3
```

```
[1] 12
```

**INFO**

Pour des commandes plus complexes, il arrive parfois qu'on se retrouve coincé avec une invite `+` sans plus savoir comment compléter la saisie correctement. On peut alors annuler la commande en utilisant la touche `Echap` ou `Esc` sous **Windows**.

Sous **Linux** on utilise le traditionnel `Control + C`.

À noter que les espaces autour des opérateurs n'ont pas d'importance lorsque l'on saisit les commandes dans **R**. Les trois commandes suivantes sont donc équivalentes, mais on privilégie en général la deuxième pour des raisons de lisibilité du code.

```
R> 10+2
   10 + 2
   10    +    2
```

# Des objets

## Objets simples

Faire des opérations arithmétiques, c'est bien, mais sans doute pas totalement suffisant. Notamment, on aimerait pouvoir réutiliser le résultat d'une opération sans avoir à le resaisir ou à le copier/coller.

Comme tout langage de programmation, **R** permet de faire cela en utilisant des objets. Prenons tout de suite un exemple :

```
R> x <- 2
```

Que signifie cette commande ? L'opérateur `<-` est appelé *opérateur d'assignation*. Il prend une valeur quelconque à droite et la place dans l'objet indiqué à gauche. La commande pourrait donc se lire *mettre la valeur 2 dans l'objet nommé* `x`.

On va ensuite pouvoir réutiliser cet objet dans d'autres calculs ou simplement afficher son contenu :

```
R> x + 3
```

```
[1] 5
```

```
R> x
```

```
[1] 2
```

**INFO**

Par défaut, si on donne à **R** seulement le nom d'un objet, il va se débrouiller pour nous présenter son contenu d'une manière plus ou moins lisible.

On peut utiliser autant d'objets qu'on veut. Ceux-ci peuvent contenir des nombres, des chaînes de caractères (indiquées par des guillemets droits doubles `"` ou simples `'`) et bien d'autres choses encore :

```
R> x <- 27
   y <- 10
   foo <- x + y
   foo
```

```
[1] 37
```

```
R> x <- "Hello"
   foo <- x
   foo
```

```
[1] "Hello"
```

**IMPORTANT**

Les noms d'objets peuvent contenir des lettres, des chiffres (mais ils ne peuvent pas commencer par un chiffre), les symboles `.` et `_`. Ils doivent impérativement commencer par une lettre. **R** fait la différence entre les majuscules et les minuscules, ce qui signifie que `x` et `X` sont deux objets différents. On évitera également d'utiliser des caractères accentués dans les noms d'objets. Comme les espaces ne sont pas autorisés on pourra les remplacer par un point ou un tiret bas.

Enfin, signalons que certains noms courts sont réservés par **R** pour son usage interne et doivent être évités. On citera notamment `c`, `q`, `t`, `C`, `D`, `F`, `I`, `T`, `max`, `min`...

Dans **RStudio**, l'onglet *Environment* dans le quadrant supérieur droit indique la liste des objets que vous avez précédemment créés, leur type et la taille qu'ils occupent en mémoire.

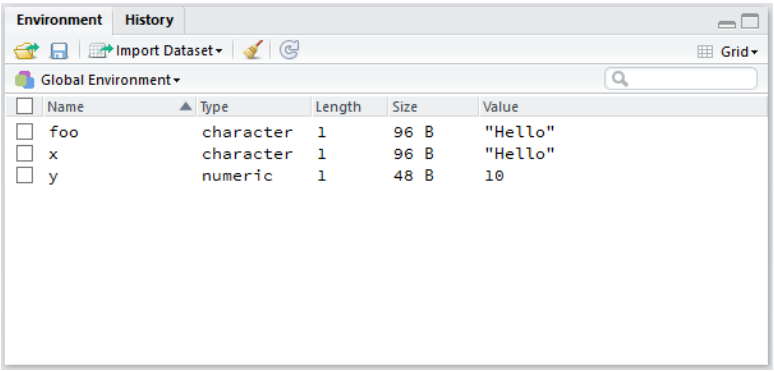



*Figure 2. Onglet Environment de RStudio*

# Vecteurs

Imaginons maintenant que nous avons interrogé dix personnes au hasard dans la rue et que nous avons relevé pour chacune d'elle sa taille en centimètres. Nous avons donc une série de dix nombres que nous souhaiterions pouvoir réunir de manière à pouvoir travailler sur l'ensemble de nos mesures.

Un ensemble de données de même nature constituent pour **R** un vecteur (en anglais *vector*) et se construit à l'aide d'une fonction nommée `c`[2]. On l'utilise en lui donnant la liste de nos données, entre parenthèses, séparées par des virgules :

```
R> tailles <- c(167, 192, 173, 174, 172, 167, 171, 185,
    163, 170)
```

Ce faisant, nous avons créé un objet nommé `tailles` et comprenant l'ensemble de nos données, que nous pouvons afficher :

```
R> tailles
```

```
 [1] 167 192 173 174 172 167 171 185 163 170
```

```
R> c(144, 168, 179, 175, 182, 188, 167, 152, 163, 145,
    176, 155, 156, 164, 167, 155, 157, 185, 155, 169,
    124, 178, 182, 195, 151, 185, 159, 156, 184, 172)
```

```
 [1] 144 168 179 175 182 188 167 152 163 145 176
[12] 155 156 164 167 155 157 185 155 169 124 178
[23] 182 195 151 185 159 156 184 172
```

On a bien notre suite de trente tailles, mais on peut remarquer la présence de nombres entre crochets au début de chaque ligne ( `[1]` , `[12]` et `[23]` ). En fait ces nombres entre crochets indiquent la position du premier élément de la ligne dans notre vecteur. Ainsi, le 155 en début de deuxième ligne est le 12$^{e}$ élément du vecteur, tandis que le 182 de la troisième ligne est à la 23$^{e}$ position.

On en déduira d'ailleurs que lorsque l'on fait :

```
R> 2
```

```
[1] 2
```

**R** considère en fait le nombre 2 comme un vecteur à un seul élément.

On peut appliquer des opérations arithmétiques simples directement sur des vecteurs :

```
R> tailles <- c(167, 192, 173, 174, 172, 167, 171, 185,
     163, 170)
   tailles + 20
```

```
 [1] 187 212 193 194 192 187 191 205 183 190
```

```
R> tailles/100
```

```
 [1] 1.67 1.92 1.73 1.74 1.72 1.67 1.71 1.85 1.63
[10] 1.70
```

```
R> tailles^2
```

```
 [1] 27889 36864 29929 30276 29584 27889 29241
 [8] 34225 26569 28900
```

On peut aussi combiner des vecteurs entre eux. L'exemple suivant calcule l'indice de masse corporelle à partir de la taille et du poids :

```
R> tailles <- c(167, 192, 173, 174, 172, 167, 171, 185,
     163, 170)
   poids <- c(86, 74, 83, 50, 78, 66, 66, 51, 50, 55)
   tailles.m <- tailles/100
   imc <- poids/(tailles.m^2)
   imc
```

```
 [1] 30.83653 20.07378 27.73230 16.51473 26.36560
 [6] 23.66524 22.57105 14.90139 18.81892 19.03114
```

**IMPORTANT**

Quand on fait des opérations sur les vecteurs, il faut veiller à soit utiliser un vecteur et un chiffre (dans des opérations du type `v * 2` ou `v + 10`), soit à utiliser des vecteurs de même longueur (dans des opérations du type `u + v`).

Si on utilise des vecteurs de longueur différentes, on peut avoir quelques surprises. Quand **R** effectue une opération avec deux vecteurs de longueurs différentes, il recopie le vecteur le plus court de manière à lui donner la même taille que le plus long, ce qui s'appelle la règle de recyclage (*recycling rule*). Ainsi, `c(1,2) + c(4,5,6,7,8)` vaudra l'équivalent de `c(1,2,1,2,1) + c(4,5,6,7,8)`.

On a vu jusque-là des vecteurs composés de nombres, mais on peut tout à fait créer des vecteurs composés de chaînes de caractères, représentant par exemple les réponses à une question ouverte ou fermée :

reponse <- c("Bac+2", "Bac", "CAP", "Bac", "Bac", "CAP", "BEP")

Enfin, notons que l'on peut accéder à un élément particulier du vecteur en faisant suivre le nom du vecteur de crochets contenant le numéro de l'élément désiré. Par exemple :

```
R> reponse <- c("Bac+2", "Bac", "CAP", "Bac", "Bac", "CAP",
     "BEP")
   reponse[2]
```

```
[1] "Bac"
```

Cette opération s'appelle l'*indexation* d'un vecteur. Il s'agit ici de sa forme la plus simple, mais il en existe d'autres beaucoup plus complexes. L'indexation des vecteurs et des tableaux dans **R** est l'un des éléments particulièrement souples et puissants du langage (mais aussi l'un des plus délicats à comprendre et à maîtriser). Nous en reparlerons dans le chapitre sur la manipulation de données.

Sous **RStudio**, vous avez du remarquer que ce dernier effectue une coloration syntaxique. Lorsque vous tapez une commande, les valeurs numériques sont affichées dans une certaine couleur, les valeurs textuelles dans une autre et les

noms des fonctions dans une troisième. De plus, si vous tapez une parenthèse ouvrante, **RStudio** va créer automatiquement après le curseur la parenthèse fermante correspondante (de même avec les guillements ou les crochets). De plus, si vous placez le curseur juste après une parenthèse fermante, la parenthèse ouvrante correspondante sera surlignée, ce qui sera bien pratique lors de la rédaction de commandes complexes.

# Des fonctions

Nous savons désormais faire des opérations simples sur des nombres et des vecteurs, stocker ces données et résultats dans des objets pour les réutiliser par la suite.

Pour aller un peu plus loin nous allons aborder, après les objets, l'autre concept de base de **R**, à savoir les *fonctions*. Une fonction se caractérise de la manière suivante :

- elle a un nom ;
- elle accepte des arguments (qui peuvent avoir un nom ou pas) ;
- elle retourne un résultat et peut effectuer une action comme dessiner un graphique ou lire un fichier.

En fait rien de bien nouveau puisque nous avons déjà utilisé plusieurs fonctions jusqu'ici, dont la plus visible est la fonction `c`. Dans la ligne suivante :

reponse <- c("Bac+2", "Bac", "CAP", "Bac", "Bac", "CAP", "BEP")

on fait appel à la fonction nommée `c`, on lui passe en arguments (entre parenthèses et séparées par des virgules) une série de chaînes de caractères et elle retourne comme résultat un vecteur de chaînes de caractères, que nous stockons dans l'objet `reponse`.

Prenons tout de suite d'autres exemples de fonctions courantes :

tailles <- c(167, 192, 173, 174, 172, 167, 171, 185, 163, 170) length(tailles) mean(tailles) var(tailles)

Ici, la fonction `length` nous renvoie le nombre d'éléments du vecteur, la fonction `mean` nous donne la moyenne des éléments du vecteur et fonction `var` sa variance.

## Arguments

Les arguments de la fonction lui sont indiqués entre parenthèses, juste après son nom. En général les premiers arguments passés à la fonction sont des données servant au calcul et les suivants des paramètres influant sur ce calcul. Ceux-ci sont en général transmis sous la forme d'argument nommés.

Reprenons l'exemple des tailles précédent :

tailles <- c(167, 192, 173, 174, 172, 167, 171, 185, 163, 170)

Imaginons que le deuxième enquêté n'ait pas voulu nous répondre. Nous avons alors dans notre vecteur une valeur manquante. Celle-ci est symbolisée dans **R** par le code `NA` :

tailles <- c(167, NA, 173, 174, 172, 167, 171, 185, 163, 170)

Recalculons notre taille moyenne :

mean(tailles)

Et oui, par défaut, **R** renvoie `NA` pour un grand nombre de calculs (dont la moyenne) lorsque les données comportent une valeur manquante. On peut cependant modifier ce comportement en fournissant un paramètre supplémentaire à la fonction `mean`, nommé `na.rm` :

mean(tailles, na.rm = TRUE)

Positionner le paramètre `na.rm` à `TRUE` (vrai) indique à la fonction `mean` de ne pas tenir compte des valeurs manquantes dans le calcul.

Lorsqu'on passe un argument à une fonction de cette manière, c'est-à-dire sous la forme `nom=valeur`, on parle d'*argument nommé*.

> **IMPORTANT**
>
> `NA` signifie *not available*. Cette valeur particulière peut être utilisée pour indiquer une valeur manquante pour tout type de liste (nombres, textes, valeurs logique, etc.).

## Quelques fonctions utiles

Récapitulons la liste des fonctions que nous avons déjà rencontrées :

| Fonction | Description |
|---|---|
| `c` | construit un vecteur à partir d'une série de valeurs |
| `length` | nombre d'éléments d'un vecteur |
| `mean` | moyenne d'un vecteur de type numérique |
| `var` | variance d'un vecteur de type numérique |
| `+`, `-`, `*`, `/` | opérateurs mathématiques de base |
| `^` | passage à la puissance |

On peut rajouter les fonctions de base suivantes :

| Fonction | Description |
|---|---|
| `min` | valeur minimale d'un vecteur numérique |
| `max` | valeur maximale d'un vecteur numérique |
| `sd` | écart-type d'un vecteur numérique |
| `:` | génère une séquence de nombres. `1:4` équivaut à `c(1,2,3,4)` |

## Aide sur une fonction

Il est très fréquent de ne plus se rappeler quels sont les paramètres d'une fonction ou le type de résultat qu'elle retourne. Dans ce cas on peut très facilement accéder à l'aide décrivant une fonction particulière avec `?` ou `help`. Ainsi, pour obtenir de l'aide sur la fonction `mean`, on saisira l'une des deux entrées équivalentes suivantes :

```
R> `?`(mean)
   help("mean")
```

> **NOTE**
>
> L'utilisation du raccourci `?` ne fonctionne pas pour certains opérateurs comme `*`. Dans ce cas on pourra utiliser `?'*'` ou bien simplement `help("*")`.

Sous **RStudio**, le fichier d'aide associé apparaitra dans le quadrant inférieur droit sous l'onglet *Help*.

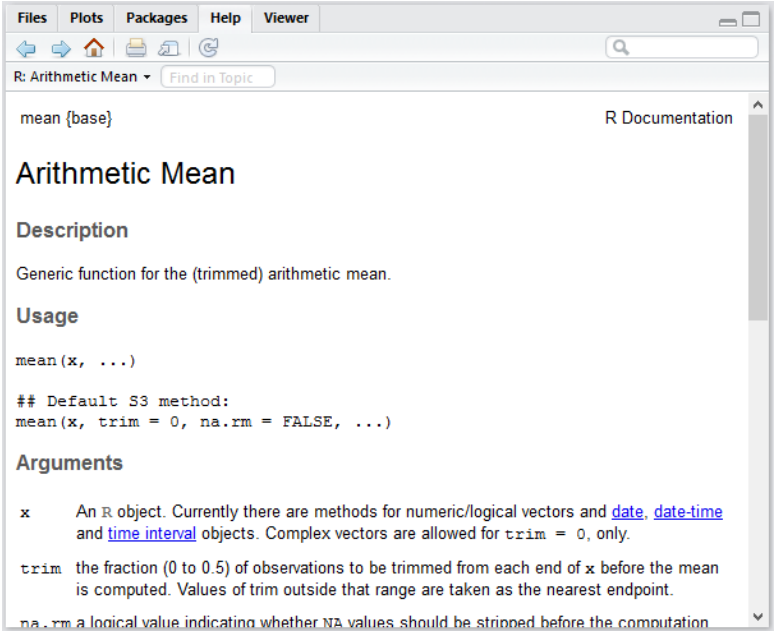



*Figure 3. Onglet Help de RStudio*

Cette page décrit (en anglais) la fonction, ses arguments, son résultat, le tout accompagné de diverses notes, références et exemples. Ces pages d'aide contiennent à peu près tout ce que vous pourrez chercher à savoir, mais elles ne sont pas toujours d'une lecture aisée.

Un autre cas très courant dans **R** est de ne pas se souvenir ou de ne pas connaître le nom de la fonction effectuant une tâche donnée. Dans ce cas on se reportera aux différentes manières de trouver de l'aide décrites dans le chapitre [Où trouver de l'aide ?] MAJ_LIEN.

## Interprétation des arguments

Prenons l'exemple de la fonction `format` dont la version de base permet de mettre en forme un nombre. Affichons le fichier d'aide associé.

```
R> `?`(format)
```

La section *Usage* présente les arguments de cette fonction et leur valeur par défaut :

```
format(x, trim = FALSE, digits = NULL, nsmall = 0L,
       justify = c("left", "right", "centre", "none"),
       width = NULL, na.encode = TRUE, scientific = NA,
       big.mark = "",   big.interval = 3L,
       small.mark = "", small.interval = 5L,
       decimal.mark = ".", zero.print = NULL,
       drop0trailing = FALSE, ...)
```

Regardons ce que cette fonction peut faire. Passons-lui un vecteur avec deux nombres :

```
R> format(c(12.3, 5678))
```

```
[1] "  12.3" "5678.0"
```

Elle renvoie un vecteur de chaînes de caractères. Le nombre de décimales a été harmonisé et des espaces ont été ajoutés au début du premier nombre afin que l'ensemble des valeurs soient alignées vers la droite.

L'argument `trim` permet de supprimer les espaces ajoutés en début de chaîne.

```
R> format(c(12.3, 5678), TRUE)
```

```
[1] "12.3"   "5678.0"
```

Dans le cas présent, nous avons saisi les arguments de la fonction sans les nommer. Dès lors, **R** considère l'ordre dans lesquels nous avons saisi les arguments, ordre qui correspond à celui du fichier d'aide. Il a dès lors considéré que `c(12.3, 5678)` correspond à la valeur attribuée à `x` et que `TRUE` est la valeur attribuée à `trim`.

L'argument `nsmall` permet d'indiquer le nombre minimum de décimales que l'on souhaite afficher. Il est en quatrième position. Dès lors, pour pouvoir le renseigner avec des arguments non nommés, il faut fournir également une valeur pour le troisième argument `digits`.

```
R> format(c(12.3, 5678), TRUE, NULL, 2)
```

```
[1] "12.30"   "5678.00"
```

Ce n'est pas forcément ce qu'il y a de plus pratique. D'où l'intérêt des *arguments nommés*. En précisant `nsmall =` dans l'appel de la fonction, on pourra indiquer que l'on souhaite modifier spécifiquement cet argument. Lorsque l'on utilise des arguments non nommés, l'ordre n'importe plus puisque **R** sera en capacité de reconnaître ses petits.

```
R> format(nsmall = 2, x = c(12.3, 5678))
```

```
[1] "  12.30" "5678.00"
```

À l'usage, on aura le plus souvent recours à une combinaison d'arguments non nommés et d'arguments nommés. On indiquera les premiers arguments (qui correspondent en général aux données de départ) sans les nommer et on précisera les options souhaitées avec des arguments nommés. Par exemple, pour un affichage à la française :

```
R> format(c(12.3, 5678), decimal.mark = ",", big.mark = " ")
```

```
[1] "   12,3" "5 678,0"
```

Lorsque l'on regarde la section *Usage* du fichier d'aide, il apparait que certains arguments, suivi par le symbole `=`, ont une valeur par défaut. Il n'est donc pas nécessaire de les inclure dans l'appel de la fonction, auquel cas la valeur pas défaut sera prise en compte. Par contre, d'autres arguments, ici `x`, n'ont pas de valeur par défaut et il est donc nécessaire de fournir systématique une valeur.

```
R> format(decimal.mark = ",")
```

```
Error in format.default(decimal.mark = ","): l'argument
"x" est manquant, avec aucune valeur par défaut
```

Enfin, pour certaines fonctions, on verra parfois apparaître le symbole `...` Ce dernier correspond à un nombre indéterminé d'arguments. Il peut s'agir, comme dans le cas de `format` d'arguments additionnels qui seront utilisés dans certains cas de figure, ou bien d'arguments qui seront transmis à une fonction

secondaire appelée par la fonction principale, ou encore, comme pour le cas de la fonction `c`, de la possibilité de saisir un nombre indéfini de données sources.

## Auto-complétion

**RStudio** fournit un outil bien pratique appelé *auto-complétion*. Saisissez les premières lettres d'une fonction, par exemple `me` puis appuyez sur la touche `Tabulation`. **RStudio** affichera la liste des fonctions dont le nom commence par `me` ainsi qu'un court descriptif de chacune. Un appui sur la touche `Entrée` provoquera la saisie du nom complet de la fonction choisie.

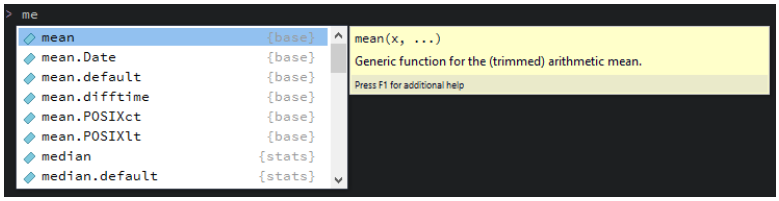



*Figure 4. Auto-complétion sous RStudio*

À l'intérieur des parenthèses d'une fonctionn, vous pouvez utiliser l'auto-complétion pour retrouver un argument de cette fonction.

Vous pouvez également utiliser l'auto-complétion pour retrouver le nom d'un objet que vous avez précédemment créé.

Pour plus de détails, voir la documentation officielle de **RStudio** (MAJ_LIEN).

---

1. La présence du `[1]` en début de ligne sera expliquée par la suite dans la section sur les vecteurs.↩
2. `c` est l'abbréviation de *combine*. Le nom de cette fonction est très court car on l'utilise très souvent.↩

# Premier travail avec des données

> **NOTE**
>
> Ce chapitre est inspiré de la section *Premier travail avec les données* du support de cours Introduction à R réalisé par Julien Barnier.

## Regrouper les commandes dans des scripts

Jusqu'à maintenant nous avons utilisé uniquement la console pour communiquer avec **R** via l'invite de commandes. Le principal problème de ce mode d'interaction est qu'une fois qu'une commande est tapée, elle est pour ainsi dire « perdue », c'est-à-dire qu'on doit la saisir à nouveau si on veut l'exécuter une seconde fois. L'utilisation de la console est donc restreinte aux petites commandes « jetables », le plus souvent utilisées comme test.

La plupart du temps, les commandes seront stockées dans un fichier à part, que l'on pourra facilement ouvrir, éditer et exécuter en tout ou partie si besoin. On appelle en général ce type de fichier un *script*.

Pour comprendre comment cela fonctionne, dans **RStudio** cliquez sur l'icône en haut à gauche représentant un fichier avec un signe + vert, puis choisissez *R script*.

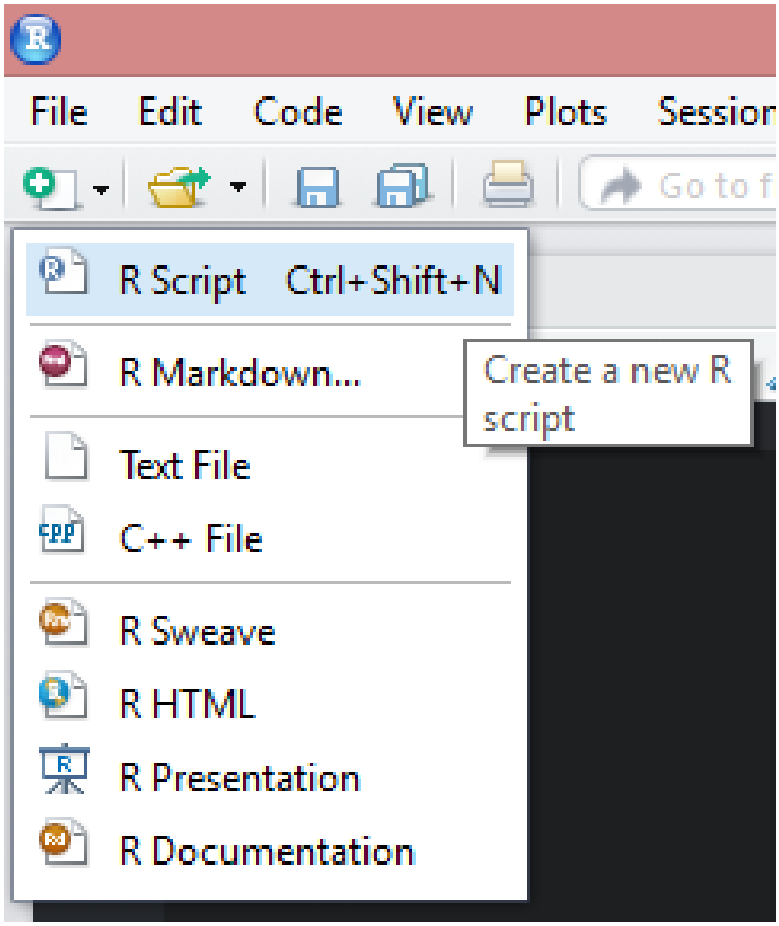



*Figure 1. Créer un nouveau script R dans RStudio*

Un nouvel onglet apparaît dans le quadrant supérieur gauche.

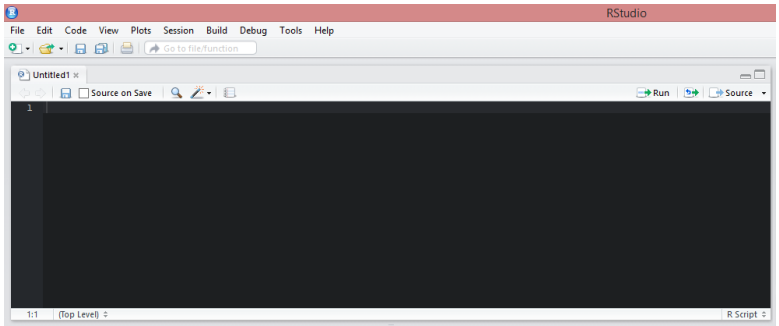



*Figure 2. Onglet d'un script R dans RStudio*

Nous pouvons désormais y saisir des commandes. Par exemple, tapez sur la première ligne la commande suivante : `2 + 2`. Ensuite, cliquez sur l'icône *Run* (en haut à droite de l'onglet du script) ou bien pressez simulatément les touches `CTRL` et `Entrée`[1].

Les lignes suivantes ont dû faire leur apparition dans la console :

```
R> 2 + 2
```

```
[1] 4
```

Voici donc comment soumettre rapidement à **R** les commandes saisies dans votre fichier. Vous pouvez désormais l'enregistrer, l'ouvrir plus tard, et en exécuter tout ou partie. À noter que vous avez plusieurs possibilités pour soumettre des commandes à **R** :

- vous pouvez exécuter la ligne sur laquelle se trouve votre curseur en cliquant sur *Run* ou en pressant simulatément les touches `CTRL` et `Entrée` ;
- vous pouvez sélectionner plusieurs lignes contenant des commandes et les exécuter toutes en une seule fois exactement de la même manière ;
- vous pouvez exécuter d'un coup l'intégralité de votre fichier en cliquant sur l'icône *Source*.

La plupart du travail sous **R** consistera donc à éditer un ou plusieurs fichiers de commandes et à envoyer régulièrement les commandes saisies à **R** en utilisant les raccourcis clavier *ad hoc*.

Pour plus d'information sur l'utilisation des scripts **R** dans **RStudio**, voir (en anglais) : http://www.rstudio.com/ide/docs/using/source.

> **NOTE**
>
> Quand vous enregistrez un script sous **RStudio**, il est possible qu'il vous demande de choisir un type d'encodage des caractères (*Choose Encoding*). Si tel est le cas, utilisez de préférence **UTF-8**.

# Ajouter des commentaires

Un commentaire est une ligne ou une portion de ligne qui sera ignorée par **R**. Ceci signifie qu'on peut y écrire ce qu'on veut et qu'on va les utiliser pour ajouter tout un tas de commentaires à notre code permettant de décrire les différentes étapes du travail, les choses à se rappeler, les questions en suspens, etc.

Un commentaire sous **R** commence par un ou plusieurs symboles `#` (qui s'obtient avec les touches `Alt Gr` et `3` sur les claviers de type PC). Tout ce qui suit ce symbole jusqu'à la fin de la ligne est considéré comme un commentaire. On peut créer une ligne entière de commentaire en la faisant débuter par `##`. Par exemple :

```
R> ## Tableau croisé de la CSP par le nombre de livres
   ## lus Attention au nombre de non réponses !
```

On peut aussi créer des commentaires pour une ligne en cours :

```
R> x <- 2  # On met 2 dans x, parce qu'il le vaut bien
```

> **IMPORTANT**
>
> Dans tous les cas, il est très important de documenter ses fichiers **R** au fur et à mesure, faute de quoi on risque de ne plus y comprendre grand chose si on les reprend ne serait-ce que quelques semaines plus tard.

Avec **RStudio**, vous pouvez également utiliser les commentaires pour créer des sections au sein de votre script et naviguer plus rapidement. Il suffit de faire suivre une ligne de commentaires d'au moins 4 signes moins ( `----` ). Par exemple, si vous saisissez ceci dans votre script :

```
R> ## Créer les objets ----

   x <- 2
   y <- 5

   ## Calculs ----

   x + y
```

Vous verrez apparaître en bas à gauche de la fenêtre du script un symbole dièse orange. Si vous cliquez dessus, un menu de navigation s'affichera vous permettant de vous déplacez rapidement au sein de votre script. Pour plus d'information, voir la documentation de **RStudio** (en anglais) : http://www.rstudio.com/ide/docs/using/code_folding.

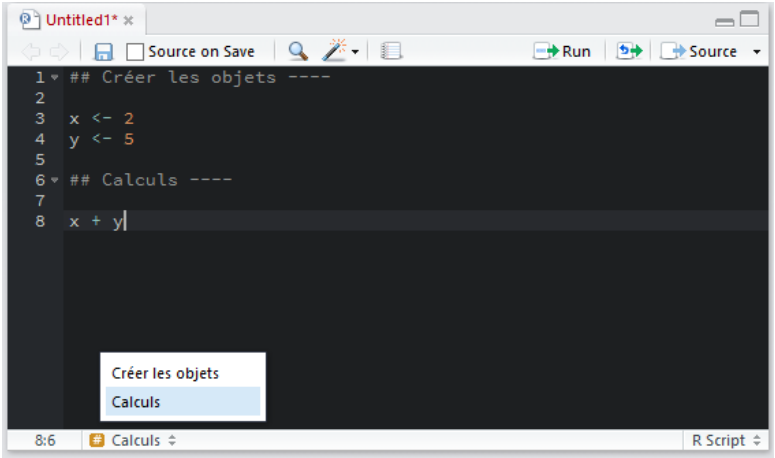



*Figure 3. Navigation rapide dans les scripts sous RStudio*

Note : on remarquera au passage que le titre de l'onglet est affiché en rouge et suivi d'une astérisque ( `*` ), nous indiquant ainsi qu'il y a des modifications non enregistrées dans notre fichier.

# Tableaux de données

Dans cette partie nous allons utiliser un jeu de données inclus dans l'extension **questionr**. L'installation d'extension est décrite dans le chapitre Présentation et Installation.

Le jeu de données en question est un extrait de l'enquête *Histoire de vie* réalisée par l'INSEE en 2003. Il contient 2000 individus et 20 variables. Pour pouvoir utiliser ces données, il faut d'abord charger l'extension **questionr** (après l'avoir installée, bien entendu). Le chargement d'une extension en mémoire se fait à l'aide de la fonction `library`. Sous **RStudio**, vous pouvez également charger une extension en allant dans l'onglet *Packages* du quadrant inférieur droit qui liste l'ensemble des packages disponibles et en cliquant la case à cocher située à gauche du nom du package désiré.

```
R> library(questionr)
```

Puis nous allons indiquer à **R** que nous souhaitons accéder au jeu de données `hdv2003` à l'aide de la fonction `data` :

```
R> data(hdv2003)
```

Bien. Et maintenant, elles sont où mes données ? Et bien elles se trouvent dans un objet nommé `hdv2003` désormais chargé en mémoire et accessible directement. D'ailleurs, cet objet est maintenant visible dans l'onglet *Environment* du quadrant supérieur droit.

Essayons de taper son nom à l'invite de commande :

```
R> hdv2003
```

Le résultat (non reproduit ici) ne ressemble pas forcément à grand-chose… Il faut se rappeler que par défaut, lorsqu'on lui fournit seulement un nom d'objet, **R** essaye de l'afficher de la manière la meilleure (ou la moins pire) possible. La réponse à la commande `hdv2003` n'est donc rien moins que l'affichage des données brutes contenues dans cet objet.

Ce qui signifie donc que l'intégralité de notre jeu de données est inclus dans l'objet nommé `hdv2003` ! En effet, dans **R**, un objet peut très bien contenir un simple nombre, un vecteur ou bien le résultat d'une enquête tout entier. Dans ce cas, les objets sont appelés des *data frames*, ou tableaux de données. Ils peuvent être manipulés comme tout autre objet. Par exemple :

```
R> d <- hdv2003
```

va entraîner la copie de l'ensemble de nos données dans un nouvel objet nommé `d`, ce qui peut paraître parfaitement inutile mais a en fait l'avantage de fournir un objet avec un nom beaucoup plus court, ce qui diminuera la quantité de texte à saisir par la suite.

**Résumons**

Comme nous avons désormais décidé de saisir nos commandes dans un script et non plus directement dans la console, les premières lignes de notre fichier de travail sur les données de l'enquête *Histoire de vie* pourraient donc ressembler à ceci :

```
R> ## Chargement des extensions nécessaires ----
   library(questionr)
   ## Jeu de données hdv2003 ----
   data(hdv2003)
   d <- hdv2003
```

# Inspecter les données

## Structure du tableau

Avant de travailler sur les données, nous allons essayer de voir à quoi elles ressemblent. Dans notre cas il s'agit de se familiariser avec la stucture du fichier. Lors de l'import de données depuis un autre logiciel, il s'agira souvent de vérifier que l'importation s'est bien déroulée.

Un tableau de données est organisé en lignes et en colonnes, les lignes correspondant aux observations et les colonnes aux variables. Les fonctions

`nrow`, `ncol` et `dim` donnent respectivement le nombre de lignes, le nombre de colonnes et les dimensions de notre tableau. Nous pouvons donc d'ores et déjà vérifier que nous avons bien 2000 lignes et 20 colonnes :

```
R> nrow(d)
```

```
[1] 2000
```

```
R> ncol(d)
```

```
[1] 20
```

```
R> dim(d)
```

```
[1] 2000   20
```

La fonction `names` donne les noms des colonnes de notre tableau, c'est-à-dire les noms des variables :

```
R> names(d)
```

```
 [1] "id"            "age"
 [3] "sexe"          "nivetud"
 [5] "poids"         "occup"
 [7] "qualif"        "freres.soeurs"
 [9] "clso"          "relig"
[11] "trav.imp"      "trav.satisf"
[13] "hard.rock"     "lecture.bd"
[15] "peche.chasse"  "cuisine"
[17] "bricol"        "cinema"
[19] "sport"         "heures.tv"
```

La fonction `str` est plus complète. Elle liste les différentes variables, indique leur type et donne le cas échéant des informations supplémentaires ainsi qu'un échantillon des premières valeurs prises par cette variable :

```
R> str(d)
```

```
'data.frame':   2000 obs. of  20 variables:
 $ id           : int  1 2 3 4 5 6 7 8 9 10 ...
 $ age          : int  28 23 59 34 71 35 60 47 20 28 ...
 $ sexe         : Factor w/ 2 levels "Homme","Femme": 2 2
1 1 2 2 2 1 2 1 ...
 $ nivetud      : Factor w/ 8 levels "N'a jamais fait
d'etudes",..: 8 NA 3 8 3 6 3 6 NA 7 ...
 $ poids        : num  2634 9738 3994 5732 4329 ...
 $ occup        : Factor w/ 7 levels "Exerce une
profession",..: 1 3 1 1 4 1 6 1 3 1 ...
 $ qualif       : Factor w/ 7 levels "Ouvrier
specialise",..: 6 NA 3 3 6 6 2 2 NA 7 ...
 $ freres.soeurs: int  8 2 2 1 0 5 1 5 4 2 ...
 $ clso         : Factor w/ 3 levels "Oui","Non","Ne sait
pas": 1 1 2 2 1 2 1 2 1 2 ...
 $ relig        : Factor w/ 6 levels "Pratiquant
regulier",..: 4 4 4 3 1 4 3 4 3 2 ...
 $ trav.imp     : Factor w/ 4 levels "Le plus
important",..: 4 NA 2 3 NA 1 NA 4 NA 3 ...
 $ trav.satisf  : Factor w/ 3 levels "Satisfaction",..: 2
NA 3 1 NA 3 NA 2 NA 1 ...
 $ hard.rock    : Factor w/ 2 levels "Non","Oui": 1 1 1 1
1 1 1 1 1 1 ...
 $ lecture.bd   : Factor w/ 2 levels "Non","Oui": 1 1 1 1
1 1 1 1 1 1 ...
 $ peche.chasse : Factor w/ 2 levels "Non","Oui": 1 1 1 1
1 1 2 2 1 1 ...
 $ cuisine      : Factor w/ 2 levels "Non","Oui": 2 1 1 2
1 1 2 2 1 1 ...
 $ bricol       : Factor w/ 2 levels "Non","Oui": 1 1 1 2
1 1 1 2 1 1 ...
 $ cinema       : Factor w/ 2 levels "Non","Oui": 1 2 1 2
1 2 1 1 2 2 ...
 $ sport        : Factor w/ 2 levels "Non","Oui": 1 2 2 2
1 2 1 1 1 2 ...
 $ heures.tv    : num  0 1 0 2 3 2 2.9 1 2 2 ...
```

La première ligne nous informe qu'il s'agit bien d'un tableau de données avec 2000 observations et 20 variables. Vient ensuite la liste des variables. La première se nomme *id* et est de type nombre entier (*int*). La seconde se nomme

*age* et est de type numérique. La troisième se nomme *sexe*, il s'agit d'un facteur (*factor*).

Un facteur et une variable pouvant prendre un nombre limité de modalités (*levels*). Ici notre variable a deux modalités possibles : « Homme » et « Femme ». Ce type de variable est décrit plus en détail dans le chapitre sur la manipulation de données.

> **IMPORTANT**
>
> La fonction `str` est essentielle à connaître et peut s'appliquer à n'importe quel type d'objet. C'est un excellent moyen de connaître la structure d'un objet.

## Inspection visuelle

La particularité de **R** par rapport à d'autres logiciels comme **Modalisa** ou **SPSS** est de ne pas proposer, par défaut, de vue des données sous forme de tableau. Ceci peut parfois être un peu déstabilisant dans les premiers temps d'utilisation, même si on perd vite l'habitude et qu'on finit par se rendre compte que « voir » les données n'est pas forcément un gage de productivité ou de rigueur dans le traitement.

Néanmoins, **R** propose une interface permettant de visualiser le contenu d'un tableau de données à l'aide de la fonction `View` :

```
R> View(d)
```

Sous **RStudio**, on peut aussi afficher la visionneusee (*viewer*) en cliquant sur la petite icône en forme de tableau située à droite de la ligne d'un tableau de données dans l'onglet *Environment* du quadrant supérieur droit (cf. figure ci-après).

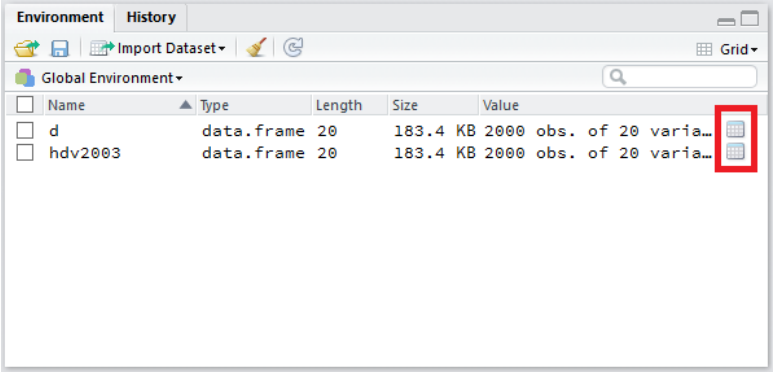



*Figure 4. Icône pour afficher une vue du contenu d'un tableau*

Dans tous les cas, **RStudio** lancera le *viewer* dans un onglet dédié dans le quadrant supérieur gauche. Le visualiseur de **RStudio** est plus avancé que celui-de base fournit par **R**. Il est possible de trier les données selon une variable en cliquant sur le nom de cette dernière. Il y a également un champs de recherche et un bouton *Filter* donnant accès à des options de filtrage avancées.

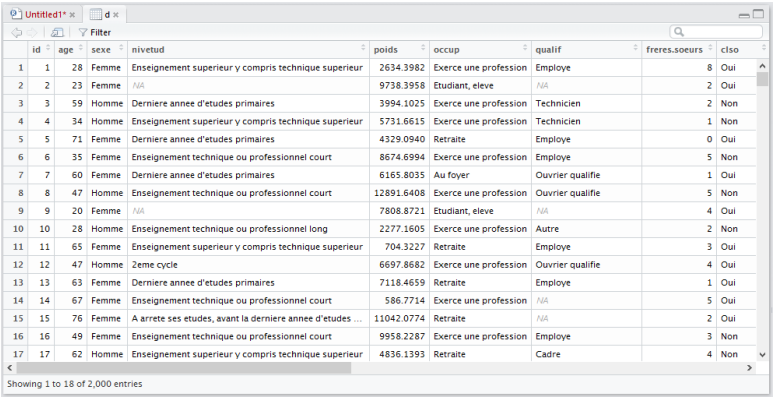



*Figure 5. La visionneuse de données de RStudio*

# Accéder aux variables

`d` représente donc l'ensemble de notre tableau de données. Nous avons vu que si l'on saisit simplement `d` à l'invite de commandes, on obtient un affichage du tableau en question. Mais comment accéder aux variables, c'est à dire aux colonnes de notre tableau ?

La réponse est simple : on utilise le nom de l'objet, suivi de l'opérateur `$`, suivi du nom de la variable, comme ceci :

```
R> d$sexe
```

Au regard du résultat (non reproduit ici), on constate alors que **R** a bien accédé au contenu de notre variable *sexe* du tableau `d` et a affiché son contenu, c'est-à-dire l'ensemble des valeurs prises par la variable.

Les fonctions `head` et `tail` permettent d'afficher seulement les premières (respectivement les dernières) valeurs prises par la variable. On peut leur passer en argument le nombre d'éléments à afficher :

```
R> head(d$sport)
```

```
[1] Non Oui Oui Oui Non Oui
Levels: Non Oui
```

```
R> tail(d$age, 10)
```

```
 [1] 52 42 50 41 46 45 46 24 24 66
```

À noter que ces fonctions marchent aussi pour afficher les lignes du tableau `d` :

```
R> head(d, 2)
```

```
  id age  sexe
1  1  28 Femme
2  2  23 Femme
                                                  nivetud
1 Enseignement superieur y compris technique superieur
2                                                    <NA>
     poids                occup  qualif
1 2634.398 Exerce une profession Employe
2 9738.396       Etudiant, eleve    <NA>
  freres.soeurs clso                      relig
1             8  Oui Ni croyance ni appartenance
2             2  Oui Ni croyance ni appartenance
       trav.imp    trav.satisf hard.rock
1 Peu important Insatisfaction       Non
2          <NA>           <NA>       Non
  lecture.bd peche.chasse cuisine bricol cinema
1        Non          Non     Oui    Non    Non
2        Non          Non     Non    Non    Oui
  sport heures.tv
1   Non         0
2   Oui         1
```

1. Sous **Mac OS X**, on utilise les touches `Pomme` et `Entrée`.↩

# Statistique univariée

**NOTE**

Ce chapitre est inspiré de la section *Premier travail avec les données* du support de cours Introduction à R réalisé par Julien Barnier.

Faisant suite au chapitre Premier travail avec les données, nous reprendrons dans ce chapitre les données de l'enquête *Histoire de vie 2003* fournies avec l'extension **questionr**.

```
R> library(questionr)
   data("hdv2003")
   d <- hdv2003
   str(d)
```

```
'data.frame':   2000 obs. of  20 variables:
 $ id           : int  1 2 3 4 5 6 7 8 9 10 ...
 $ age          : int  28 23 59 34 71 35 60 47 20 28 ...
 $ sexe         : Factor w/ 2 levels "Homme","Femme": 2 2
1 1 2 2 2 1 2 1 ...
 $ nivetud      : Factor w/ 8 levels "N'a jamais fait
d'etudes",..: 8 NA 3 8 3 6 3 6 NA 7 ...
 $ poids        : num  2634 9738 3994 5732 4329 ...
 $ occup        : Factor w/ 7 levels "Exerce une
profession",..: 1 3 1 1 4 1 6 1 3 1 ...
 $ qualif       : Factor w/ 7 levels "Ouvrier
specialise",..: 6 NA 3 3 6 6 2 2 NA 7 ...
 $ freres.soeurs: int  8 2 2 1 0 5 1 5 4 2 ...
 $ clso         : Factor w/ 3 levels "Oui","Non","Ne sait
pas": 1 1 2 2 1 2 1 2 1 2 ...
 $ relig        : Factor w/ 6 levels "Pratiquant
regulier",..: 4 4 4 3 1 4 3 4 3 2 ...
 $ trav.imp     : Factor w/ 4 levels "Le plus
important",..: 4 NA 2 3 NA 1 NA 4 NA 3 ...
 $ trav.satisf  : Factor w/ 3 levels "Satisfaction",..: 2
NA 3 1 NA 3 NA 2 NA 1 ...
 $ hard.rock    : Factor w/ 2 levels "Non","Oui": 1 1 1 1
1 1 1 1 1 1 ...
 $ lecture.bd   : Factor w/ 2 levels "Non","Oui": 1 1 1 1
1 1 1 1 1 1 ...
 $ peche.chasse : Factor w/ 2 levels "Non","Oui": 1 1 1 1
1 1 2 2 1 1 ...
 $ cuisine      : Factor w/ 2 levels "Non","Oui": 2 1 1 2
1 1 2 2 1 1 ...
 $ bricol       : Factor w/ 2 levels "Non","Oui": 1 1 1 2
1 1 1 2 1 1 ...
 $ cinema       : Factor w/ 2 levels "Non","Oui": 1 2 1 2
1 2 1 1 2 2 ...
 $ sport        : Factor w/ 2 levels "Non","Oui": 1 2 2 2
1 2 1 1 1 2 ...
 $ heures.tv    : num  0 1 0 2 3 2 2.9 1 2 2 ...
```

# Variable quantitative

## Principaux indicateurs

Comme la fonction `str` nous l'a indiqué, notre tableau `d` contient plusieurs valeurs numériques, dont la variable *heures.tv* qui représente le nombre moyen passé par les enquêtés à regarder la télévision quotidiennement. On peut essayer de déterminer quelques caractéristiques de cette variable, en utilisant les fonctions `mean`, `sd`, `min`, `max` et `range` :

```
R> mean(d$heures.tv)
[1] NA
R> mean(d$heures.tv, na.rm = TRUE)
[1] 2.246566
R> sd(d$heures.tv, na.rm = TRUE)
[1] 1.775853
R> min(d$heures.tv, na.rm = TRUE)
[1] 0
R> max(d$heures.tv, na.rm = TRUE)
[1] 12
R> range(d$heures.tv, na.rm = TRUE)
[1]  0 12
```

On peut lui ajouter la fonction `median` qui donne la valeur médiane, `quantile` qui calcule plus généralement tout type de quantiles, et le très utile `summary`

qui donne toutes ces informations ou presque en une seule fois, avec en prime le nombre de valeurs manquantes ( `NA` ) :

```
R> median(d$heures.tv, na.rm = TRUE)
```

```
[1] 2
```

```
R> quantile(d$heures.tv, na.rm = TRUE)
```

```
  0%  25%  50%  75% 100%
   0    1    2    3   12
```

```
R> summary(d$heures.tv)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.000   1.000   2.000   2.247   3.000  12.000
   NA's
      5
```

La fonction `summary` est une fonction générique qui peut être utilisée sur tout type d'objet, y compris un tableau de données. Essayez donc `summary(d)` .

## Histogramme

Tout cela est bien pratique, mais pour pouvoir observer la distribution des valeurs d'une variable quantitative, il n'y a quand même rien de mieux qu'un bon graphique.

On peut commencer par un histogramme de la répartition des valeurs. Celui-ci peut être généré très facilement avec la fonction `hist` :

```
R> hist(d$heures.tv, main = "Nombre d'heures passées
   devant la télé par jour",
     xlab = "Heures", ylab = "Effectif")
```

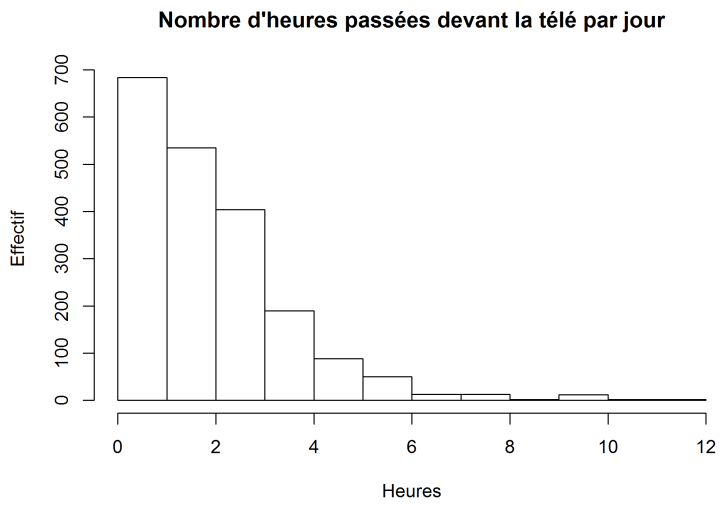



*Figure 1. Exemple d'histogramme*

Sous **RStudio**, les graphiques s'affichent dans l'onglet *Plots* du quadrant inférieur droit. Il est possible d'afficher une version plus grande de votre graphique en cliquant sur *Zoom*.

Ici, les options `main`, `xlab` et `ylab` permettent de personnaliser le titre du graphique, ainsi que les étiquettes des axes. De nombreuses autres options existent pour personnaliser l'histogramme, parmi celles-ci on notera :

- `probability` si elle vaut `TRUE`, l'histogramme indique la proportion des classes de valeurs au lieu des effectifs.
- `breaks` permet de contrôler les classes de valeurs. On peut lui passer un chiffre, qui indiquera alors le nombre de classes, un vecteur, qui indique alors les limites des différentes classes, ou encore une chaîne de caractère ou une fonction indiquant comment les classes doivent être calculées.
- `col` la couleur de l'histogramme[1].

Voir la page d'aide de la fonction `hist` pour plus de détails sur les différentes options. Les deux figures ci-après sont deux autres exemples d'histogramme.

```
R> hist(d$heures.tv, main = "Heures de télé en 7
   classes",
     breaks = 7, xlab = "Heures", ylab = "Proportion",
     probability = TRUE, col = "orange")
```

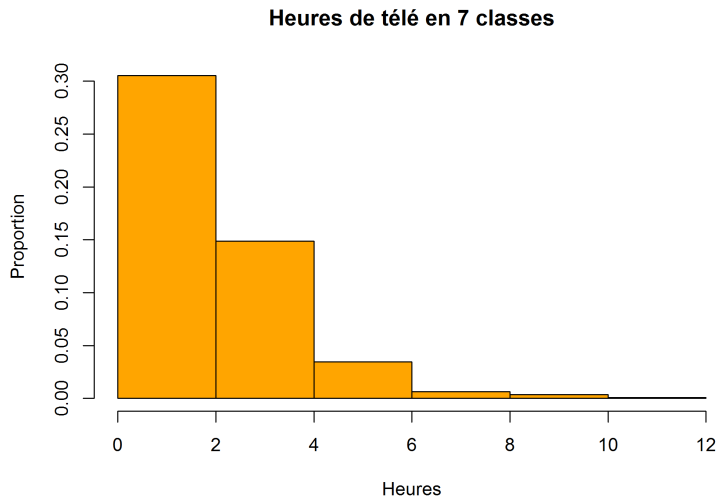



*Figure 2. Un autre exemple d'histogramme*

```
R> hist(d$heures.tv, main = "Heures de télé avec
   classes spécifiées",
     breaks = c(0, 1, 4, 9, 12), xlab = "Heures", ylab
   = "Proportion",
     col = "red")
```

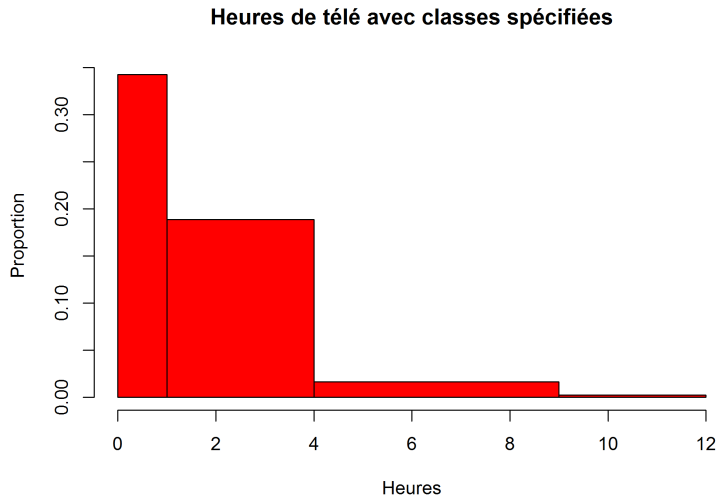



*Figure 3. Encore un autre exemple d'histogramme*

## Boîtes à moustaches

Les boîtes à moustaches, ou *boxplot* en anglais, sont une autre représentation graphique de la répartition des valeurs d'une variable quantitative. Elles sont particulièrement utiles pour comparer les distributions de plusieurs variables ou d'une même variable entre différents groupes, mais peuvent aussi être utilisées pour représenter la dispersion d'une unique variable. La fonction qui produit ces graphiques est la fonction `boxplot`.

```
R> boxplot(d$heures.tv, main = "Nombre d'heures
   passées devant la télé par jour",
     ylab = "Heures")
```

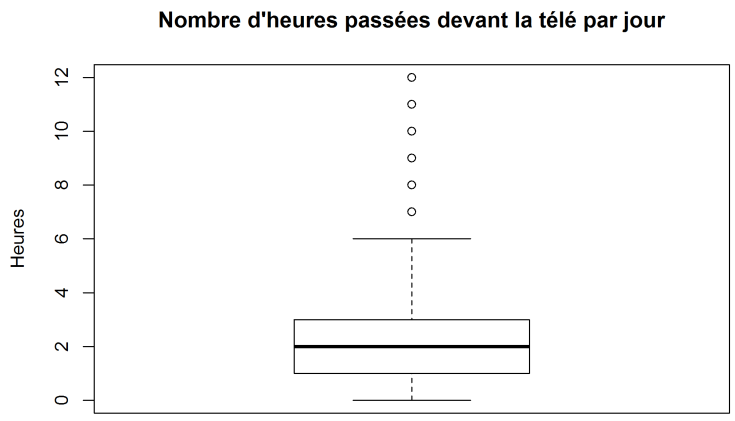



*Figure 4. Exemple de boîte à moustaches*

Comment interpréter ce graphique ? On le comprendra mieux à partir de la figure ci-après[2].

```
R> boxplot(d$heures.tv, col = grey(0.8), main =
   "Nombre d'heures passées devant la télé par jour",
     ylab = "Heures")
   abline(h = median(d$heures.tv, na.rm = TRUE), col =
   "navy",
     lty = 2)
   text(1.35, median(d$heures.tv, na.rm = TRUE) +
   0.15,
     "Médiane", col = "navy")
   Q1 <- quantile(d$heures.tv, probs = 0.25, na.rm =
   TRUE)
   abline(h = Q1, col = "darkred")
   text(1.35, Q1 + 0.15, "Q1 : premier quartile", col
   = "darkred",
     lty = 2)
   Q3 <- quantile(d$heures.tv, probs = 0.75, na.rm =
   TRUE)
   abline(h = Q3, col = "darkred")
   text(1.35, Q3 + 0.15, "Q3 : troisième quartile",
   col = "darkred",
     lty = 2)
   arrows(x0 = 0.7, y0 = quantile(d$heures.tv, probs =
   0.75,
     na.rm = TRUE), x1 = 0.7, y1 =
   quantile(d$heures.tv,
     probs = 0.25, na.rm = TRUE), length = 0.1, code =
   3)
   text(0.7, Q1 + (Q3 - Q1)/2 + 0.15, "h", pos = 2)
   mtext("L'écart inter-quartile h contient 50 % des
   individus",
     side = 1)
   abline(h = Q1 - 1.5 * (Q3 - Q1), col = "darkgreen")
   text(1.35, Q1 - 1.5 * (Q3 - Q1) + 0.15, "Q1 -1.5
   h",
     col = "darkgreen", lty = 2)
   abline(h = Q3 + 1.5 * (Q3 - Q1), col = "darkgreen")
   text(1.35, Q3 + 1.5 * (Q3 - Q1) + 0.15, "Q3 +1.5
```

```
h",
  col = "darkgreen", lty = 2)
```

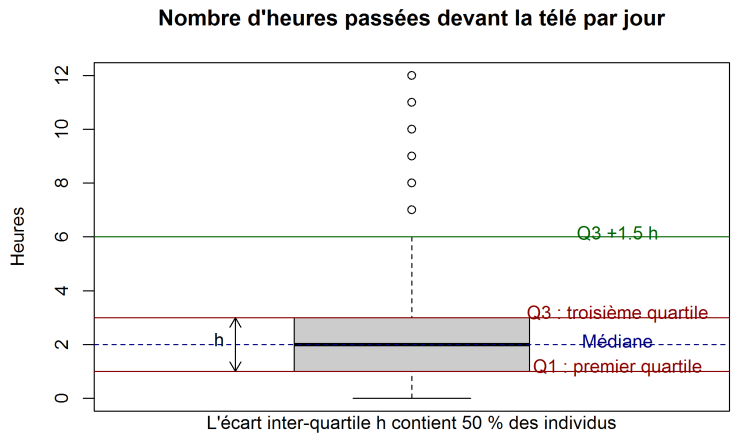



*Figure 5. Interprétation d'une boîte à moustaches*

Le carré au centre du graphique est délimité par les premiers et troisième quartiles, avec la médiane représentée par une ligne plus sombre au milieu. Les « fourchettes » s'étendant de part et d'autres vont soit jusqu'à la valeur minimale ou maximale, soit jusqu'à une valeur approximativement égale au quartile le plus proche plus 1,5 fois l'écart inter-quartile. Les points se situant en-dehors de cette fourchette sont représentés par des petits ronds et sont généralement considérés comme des valeurs extrêmes, potentiellement aberrantes.

On peut ajouter la représentation des valeurs sur le graphique pour en faciliter la lecture avec des petits traits dessinés sur l'axe vertical (fonction `rug`) :

```
R> boxplot(d$heures.tv, main = "Nombre d'heures
   passées devant la télé par\njour",
     ylab = "Heures")
   rug(d$heures.tv, side = 2)
```

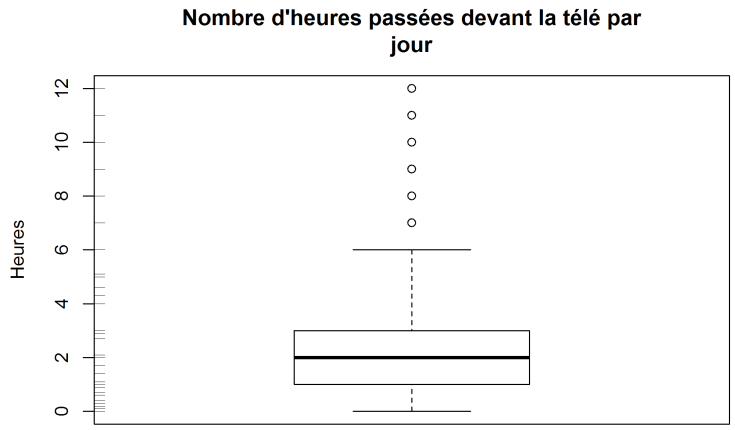



*Figure 6. Boîte à moustaches avec représentation des valeurs*

# Intervalle de confiance

L'intervalle de confiance d'une moyenne peut être calculé avec la fonction `t.test` (fonction qui permet également de réaliser un test *t* de Student comme nous le verrons dans le chapitre dédié à la statistique bivariée) :

```
R> t.test(d$heures.tv)
```

```
    One Sample t-test

data:  d$heures.tv
t = 56.505, df = 1994, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 2.168593 2.324540
sample estimates:
mean of x
 2.246566
```

Le niveau de confiance peut être précisé via l'argument `conf.level` :

```
R> t.test(d$heures.tv, conf.level = 0.9)
```

```
    One Sample t-test

data:  d$heures.tv
t = 56.505, df = 1994, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
90 percent confidence interval:
 2.181138 2.311995
sample estimates:
mean of x
 2.246566
```

Le nombre d'heures moyennes à regarder la télévision parmi les enquêtés s'avère être de 2,2 heures, avec un intervalle de confiance à 95 % de [2,17 - 2,33] et un intervalle de confiance à 90 % de [2,18 - 2,31].

# Variable qualitative

## Tris à plat

La fonction la plus utilisée pour le traitement et l'analyse des variables qualitatives (variable prenant ses valeurs dans un ensemble de modalités) est

sans aucun doute la fonction `table`, qui donne les effectifs de chaque modalité de la variable.

```
R> table(d$sexe)
```

```
Homme Femme
  899  1101
```

La tableau précédent nous indique que parmi nos enquêtés on trouve 899 hommes et 1101 femmes.

Quand le nombre de modalités est élevé, on peut ordonner le tri à plat selon les effectifs à l'aide de la fonction `sort`.

```
R> table(d$occup)
```

```
Exerce une profession               Chomeur
                 1049                   134
      Etudiant, eleve              Retraite
                   94                   392
  Retire des affaires              Au foyer
                   77                   171
        Autre inactif
                   83
```

```
R> sort(table(d$occup))
```

```
  Retire des affaires         Autre inactif
                   77                    83
      Etudiant, eleve               Chomeur
                   94                   134
             Au foyer              Retraite
                  171                   392
Exerce une profession
                 1049
```

```
R> sort(table(d$occup), decreasing = TRUE)
```

```
Exerce une profession              Retraite
                 1049                   392
             Au foyer               Chomeur
                  171                   134
      Etudiant, eleve         Autre inactif
                   94                    83
  Retire des affaires
                   77
```

À noter que la fonction `table` exclut par défaut les non-réponses du tableau résultat. L'argument `useNA` de cette fonction permet de modifier ce comportement :

- avec `useNA="no"` (valeur par défaut), les valeurs manquantes ne sont jamais incluses dans le tri à plat ;
- avec `useNA="ifany"`, une colonne `NA` est ajoutée si des valeurs manquantes sont présentes dans les données ;
- avec `useNA="always"`, une colonne `NA` est toujours ajoutée, même s'il n'y a pas de valeurs manquantes dans les données.

On peut donc utiliser :

```
R> table(d$trav.satisf, useNA = "ifany")
```

```
  Satisfaction Insatisfaction      Equilibre
           480            117            451
          <NA>
           952
```

L'utilisation de `summary` permet également l'affichage du tri à plat et du nombre de non-réponses :

```
R> summary(d$trav.satisf)
```

```
  Satisfaction Insatisfaction      Equilibre
           480            117            451
          NA's
           952
```

Pour obtenir un tableau avec la répartition en pourcentages, on peut utiliser la fonction `freq` de l'extension **questionr**[3].

```
R> freq(d$qualif)
```

```
                          n    % val%
Ouvrier specialise      203 10.2 12.3
Ouvrier qualifie        292 14.6 17.7
Technicien               86  4.3  5.2
Profession intermediaire 160  8.0  9.7
Cadre                   260 13.0 15.7
Employe                 594 29.7 35.9
Autre                    58  2.9  3.5
NA                      347 17.3   NA
```

La colonne `n` donne les effectifs bruts, la colonne `%` la répartition en pourcentages et `val%` la répartition en pourcentages, données manquantes exclues. La fonction accepte plusieurs paramètres permettant d'afficher les totaux, les pourcentages cumulés, de trier selon les effectifs ou de contrôler l'affichage. Par exemple :

```
R> freq(d$qualif, cum = TRUE, total = TRUE, sort = "inc",
    digits = 2, exclude = NA)
```

```
                           n      %   %cum
Autre                     58   3.51   3.51
Technicien                86   5.20   8.71
Profession intermediaire 160   9.68  18.39
Ouvrier specialise       203  12.28  30.67
Cadre                    260  15.73  46.40
Ouvrier qualifie         292  17.66  64.07
Employe                  594  35.93 100.00
Total                   1653 100.00 100.00
```

La colonne `%cum` indique ici le pourcentage cumulé, ce qui est ici une très mauvaise idée puisque pour ce type de variable cela n'a aucun sens. Les lignes du tableau résultat ont été triés par effectifs croissants, les totaux ont été ajoutés, les non-réponses exclues et les pourcentages arrondis à deux décimales.

Pour plus d'informations sur la fonction `freq`, consultez sa page d'aide en ligne avec `?freq` ou `help("freq")`.

## Représentation graphique

Pour représenter la répartition des effectifs parmi les modalités d'une variable qualitative, on a souvent tendance à utiliser des diagrammes en secteurs (camemberts). Ceci est possible sous **R** avec la fonction `pie`, mais la page d'aide de la dite fonction nous le déconseille assez vivement : les diagrammes en secteur sont en effet une mauvaise manière de présenter ce type d'information, car l'oeil humain préfère comparer des longueurs plutôt que des surfaces[4].

On privilégiera donc d'autres formes de représentations, à savoir les diagrammes en bâtons et les *diagrammes de Cleveland*.

Les diagrammes en bâtons sont utilisés automatiquement par **R** lorsqu'on applique la fonction générique `plot` à un tri à plat obtenu avec `table`. On privilégiera cependant ce type de représentations pour les variables de type numérique comportant un nombre fini de valeurs. Le nombre de frères, soeurs, demi-frères et demi-soeurs est un bon exemple :

```
R> plot(table(d$freres.soeurs), main = "Nombre de
   frères, soeurs, demi-frères et demi-soeurs",
     ylab = "Effectif")
```

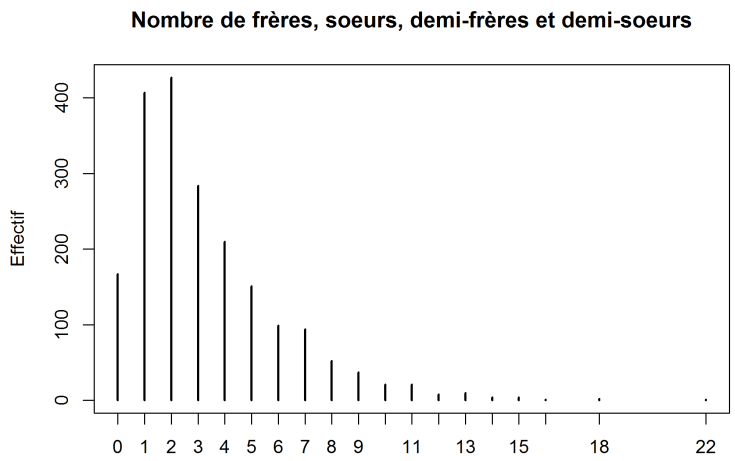



*Figure 7. Exemple de diagramme en bâtons*

Pour les autres types de variables qualitatives, on privilégiera les *diagrammes de Cleveland*, obtenus avec la fonction `dotchart`. On doit appliquer cette fonction au tri à plat de la variable, obtenu avec `table`[5] :

```
R> dotchart(as.matrix(table(d$clso)), main =
   "Sentiment d'appartenance à une classe sociale",
     pch = 19)
```

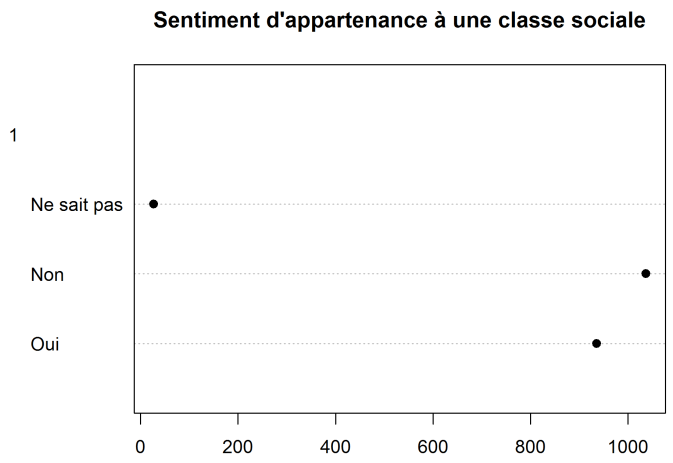



*Figure 8. Exemple de diagramme de Cleveland*

Quand la variable comprend un grand nombre de modalités, il est préférable d'ordonner le tri à plat obtenu à l'aide de la fonction `sort` :

```
R> dotchart(as.matrix(sort(table(d$qualif))), main =
   "Niveau de qualification")
```

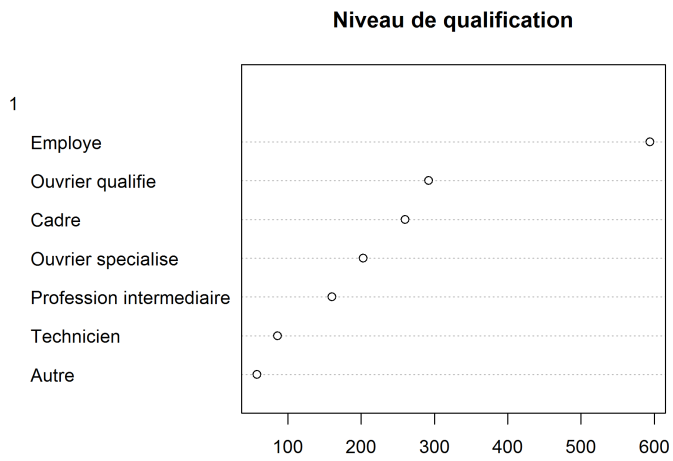
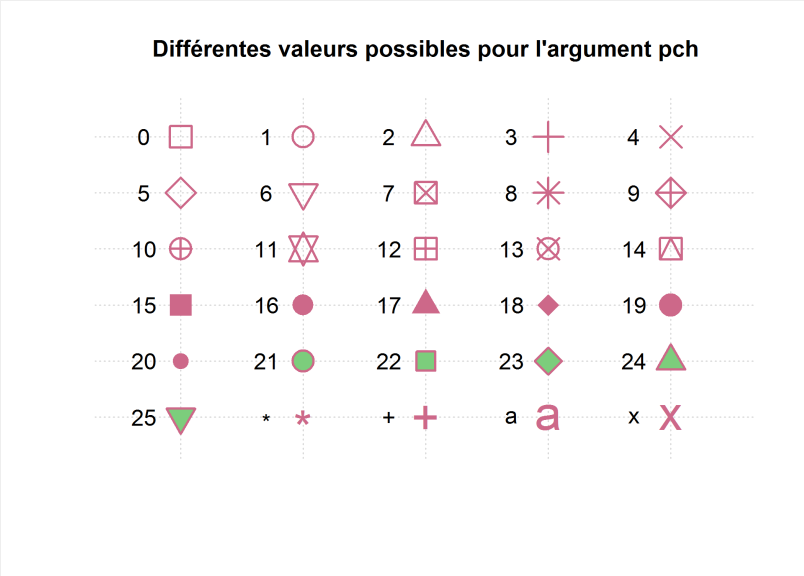



*Figure 9. Exemple de diagramme de Cleveland ordonné*

**ASTUCE**

L'agument `pch`, qui est utilisé par la plupart des graphiques de type points, permet de spécifier le symbole à utiliser. Il peut prendre soit un nombre entier compris entre 0 et 25, soit un charactère textuel (voir ci-dessous).




## Intervalle de confiance

La fonction `prop.test` permet de calculer l'intervalle de confiance d'une proportion. Une première possibilité consiste à lui transmettre une table à une dimension et deux entrées. Par exemple, si l'on s'intéresse à la proportion de personnes ayant pratiqué une activité physique au cours des douze derniers mois :

```
R> freq(d$sport)
```

```
        n    % val%
Non 1277 63.8 63.8
Oui  723 36.1 36.1
NA     0  0.0   NA
```

```
R> prop.test(table(d$sport))
```

```
    1-sample proportions test with continuity
    correction

data:  table(d$sport), null probability 0.5
X-squared = 152.9, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
 0.6169447 0.6595179
sample estimates:
     p
0.6385
```

On remarquera que la fonction a calculé l'intervalle de confiance correspondant à la première entrée du tableau, autrement dit celui de la proportion d'enquêtés n'ayant pas pratiqué une activité sportive. Or, nous sommes intéressé par la proportion complémentaire, à savoir celle d'enquêtés ayant pratiqué une activité sportive. On peut dès lors modifier l'ordre de la table en indiquant notre modalité d'intérêt avec la fonction `relevel` ou bien indiquer à `prop.test` d'abord le nombre de succès puis l'effectif total :

```
R> prop.test(table(relevel(d$sport, "Oui")))
```

```
    1-sample proportions test with continuity
    correction

data:  table(relevel(d$sport, "Oui")), null probability 0.5
X-squared = 152.9, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
 0.3404821 0.3830553
sample estimates:
     p
0.3615
```

```
R> prop.test(sum(d$sport == "Oui"), length(d$sport))
```

```
    1-sample proportions test with continuity
    correction

data:  sum(d$sport == "Oui") out of length(d$sport), null
probability 0.5
X-squared = 152.9, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
 0.3404821 0.3830553
sample estimates:
     p
0.3615
```

Enfin, le niveau de confiance peut être modifié via l'argument `conf.level` :

```
R> prop.test(table(relevel(d$sport, "Oui")), conf.level =
   0.9)
```

```
    1-sample proportions test with continuity
    correction

data:  table(relevel(d$sport, "Oui")), null probability 0.5
X-squared = 152.9, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is not equal to 0.5
90 percent confidence interval:
 0.3437806 0.3795989
sample estimates:
     p
0.3615
```

**INFO**

Il existe de nombreuses manières de calculer un intervalle de confiance pour une proportion. En l'occurence, l'intervalle calculé par `prop.test` correspond dans le cas présent à un intervalle bilatéral selon la méthode des scores de Wilson avec correction de continuité. Pour plus d'information, on pourra lire http://joseph.larmarange.net/?Intervalle-de-confiance-bilateral.

1. Il existe un grand nombre de couleurs prédéfinies dans **R**. On peut récupérer leur liste en utilisant la fonction `colors` en tapant simplement `colors()` dans la console, ou en consultant le document suivant : http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf.↩
2. Le code ayant servi à générer cette figure est une copie quasi conforme de celui présenté dans l'excellent document de Jean Lobry sur les graphiques de base avec **R**, téléchargeable sur le site du Pôle bioinformatique lyonnais : http://pbil.univ-lyon1.fr/R/pdf/lang04.pdf.↩
3. En l'absence de l'extension **questionr**, on pourra se rabattre sur la fonction `prop.table` avec la commande suivante : `prop.table(table(d$qualif))`.↩
4. On trouvera des exemples illustrant cette idée dans le document de Jean Lobry cité précédemment.↩
5. Pour des raisons liées au fonctionnement interne de la fonction `dotchart`, on doit transformer le tri à plat en matrice, d'où l'appel à la fonction `as.matrix`.↩

# Import / Export de données

à venir

# Manipulation de données

**NOTE**

La version originale de ce chapitre a été écrite par Julien Barnier dans le cadre du support de cours Introduction à R.

**INFO**

Cette partie est un peu aride et pas forcément très intuitive. Elle aborde cependant la base de tous les traitements et manipulation de données sous **R**, et mérite donc qu'on s'y arrête un moment, ou qu'on y revienne un peu plus tard en cas de saturation…

## Variables

Le type d'objet utilisé par **R** pour stocker des tableaux de données s'appelle un *data frame*. Celui-ci comporte des observations en ligne et des variables en colonnes. On accède aux variables d'un *data frame* avec l'opérateur `$`.

Dans ce qui suit on travaillera sur le jeu de données tiré de l'enquête *Histoire de vie*, fourni avec l'extension **questionr**, mais aussi sur le jeu de données tiré du recensement 1999 et disponible dans la même extension.

```
R> library(questionr)
   data(hdv2003)
   d <- hdv2003
   data(rp99)
```

## Types de variables

On peut considérer qu'il existe quatre type de variables dans **R** :

- les variables *numériques*, ou quantitatives ;
- les *facteurs*, qui prennent leurs valeurs dans un ensemble défini de modalités. Elles correspondent en général aux questions fermées d'un questionnaire ;
- les variables *caractères* ou *texte*, qui contiennent des chaînes de caractères plus ou moins longues. On les utilise pour les questions ouvertes ou les champs libres ;
- les variables *booléennes*, qui ne peuvent prendre que la valeur vrai ( `TRUE` ) ou faux ( `FALSE` ). On les utilise dans **R** pour les calculs et les recodages.

Pour connaître le type d'une variable donnée, on peut utiliser la fonction `class`.

| Classe R | Type de variable |
|---|---|
| factor | facteur |
| integer | numérique |
| double | numérique |
| numeric | numérique |
| character | caractère/texte |
| logical | booléenne |

```
R> class(d$age)
```

```
[1] "integer"
```

```
R> class(d$sexe)
```

```
[1] "factor"
```

```
R> class(c(TRUE, TRUE, FALSE))
```

```
[1] "logical"
```

La fonction `str` permet également d'avoir un listing de toutes les variables d'un tableau de données et indique le type de chacune d'elle.

```
R> str(d)
```

```
'data.frame':   2000 obs. of  20 variables:
 $ id           : int  1 2 3 4 5 6 7 8 9 10 ...
 $ age          : int  28 23 59 34 71 35 60 47 20 28 ...
 $ sexe         : Factor w/ 2 levels "Homme","Femme": 2 2
1 1 2 2 2 1 2 1 ...
 $ nivetud      : Factor w/ 8 levels "N'a jamais fait
d'etudes",..: 8 NA 3 8 3 6 3 6 NA 7 ...
 $ poids        : num  2634 9738 3994 5732 4329 ...
 $ occup        : Factor w/ 7 levels "Exerce une
profession",..: 1 3 1 1 4 1 6 1 3 1 ...
 $ qualif       : Factor w/ 7 levels "Ouvrier
specialise",..: 6 NA 3 3 6 6 2 2 NA 7 ...
 $ freres.soeurs: int  8 2 2 1 0 5 1 5 4 2 ...
 $ clso         : Factor w/ 3 levels "Oui","Non","Ne sait
pas": 1 1 2 2 1 2 1 2 1 2 ...
 $ relig        : Factor w/ 6 levels "Pratiquant
regulier",..: 4 4 4 3 1 4 3 4 3 2 ...
 $ trav.imp     : Factor w/ 4 levels "Le plus
important",..: 4 NA 2 3 NA 1 NA 4 NA 3 ...
 $ trav.satisf  : Factor w/ 3 levels "Satisfaction",..: 2
NA 3 1 NA 3 NA 2 NA 1 ...
 $ hard.rock    : Factor w/ 2 levels "Non","Oui": 1 1 1 1
1 1 1 1 1 1 ...
 $ lecture.bd   : Factor w/ 2 levels "Non","Oui": 1 1 1 1
1 1 1 1 1 1 ...
 $ peche.chasse : Factor w/ 2 levels "Non","Oui": 1 1 1 1
1 1 2 2 1 1 ...
 $ cuisine      : Factor w/ 2 levels "Non","Oui": 2 1 1 2
1 1 2 2 1 1 ...
 $ bricol       : Factor w/ 2 levels "Non","Oui": 1 1 1 2
1 1 1 2 1 1 ...
 $ cinema       : Factor w/ 2 levels "Non","Oui": 1 2 1 2
1 2 1 1 2 2 ...
 $ sport        : Factor w/ 2 levels "Non","Oui": 1 2 2 2
1 2 1 1 1 2 ...
 $ heures.tv    : num  0 1 0 2 3 2 2.9 1 2 2 ...
```

## Renommer des variables

Une opération courante lorsqu'on a importé des variables depuis une source de données externe consiste à renommer les variables importées. Sous **R** les noms de variables doivent être à la fois courts et explicites.

> **IMPORTANT**
>
> Les noms de variables peuvent contenir des lettres, des chiffres (mais ils ne peuvent pas commencer par un chiffre), les symboles `.` et `_` et doivent commencer par une lettre. **R** fait la différence entre les majuscules et les minuscules, ce qui signifie que *x* et **X** sont deux noms de variable différents. On évitera également d'utiliser des caractères accentués dans les noms de variable. Comme les espaces ne sont pas autorisés, on pourra les remplacer par un point ou un tiret bas.

On peut lister les noms des variables d'un tableau de données (*data.frame*) à l'aide de la fonction `names` :

```
R> names(d)
```

```
 [1] "id"            "age"
 [3] "sexe"          "nivetud"
 [5] "poids"         "occup"
 [7] "qualif"        "freres.soeurs"
 [9] "clso"          "relig"
[11] "trav.imp"      "trav.satisf"
[13] "hard.rock"     "lecture.bd"
[15] "peche.chasse"  "cuisine"
[17] "bricol"        "cinema"
[19] "sport"         "heures.tv"
```

Cette fonction peut également être utilisée pour renommer l'ensemble des variables. Si par exemple on souhaitait passer les noms de toutes les variables en majuscules, on pourrait faire :

```
R> d.maj <- d
   names(d.maj) <- c("ID", "AGE", "SEXE", "NIVETUD",
   "POIDS",
     "OCCUP", "QUALIF", "FRERES.SOEURS", "CLSO", "RELIG",
     "TRAV.IMP", "TRAV.SATISF", "HARD.ROCK", "LECTURE.BD",
     "PECHE.CHASSE", "CUISINE", "BRICOL", "CINEMA",
     "SPORT", "HEURES.TV")
   summary(d.maj$SEXE)
```

```
Homme Femme
  899  1101
```

Ce type de renommage peut être utile lorsqu'on souhaite passer en revue tous les noms de variables d'un fichier importé pour les corriger le cas échéant. Pour faciliter un peu ce travail pas forcément passionant, on peut utiliser la fonction `dput` :

```
R> dput(names(d))
```

```
c("id", "age", "sexe", "nivetud", "poids", "occup",
"qualif",
"freres.soeurs", "clso", "relig", "trav.imp",
"trav.satisf",
"hard.rock", "lecture.bd", "peche.chasse", "cuisine",
"bricol",
"cinema", "sport", "heures.tv")
```

On obtient en résultat la liste des variables sous forme de vecteur déclaré. On n'a plus alors qu'à copier/coller cette chaîne, rajouter `names(d) <-` devant et modifier un à un les noms des variables.

Si on souhaite seulement modifier le nom d'une variable, on peut utiliser la fonction `rename.variable` de l'extension **questionr**. Celle-ci prend en argument le tableau de données, le nom actuel de la variable et le nouveau nom. Par exemple, si on veut renommer la variable *bricol* du tableau de données `d` en *bricolage* :

```
R> d <- rename.variable(d, "bricol", "bricolage")
   table(d$bricolage)
```

```
 Non  Oui
1147  853
```

## Facteurs

Parmi les différents types de variables, les facteurs (*factor*) sont à la fois à part et très utilisés, car ils vont correspondre à la plupart des variables issues d'une question fermée dans un questionnaire.

Les facteurs prennent leurs valeurs dans un ensemble de modalités prédéfinies et ne peuvent en prendre d'autres. La liste des valeurs possibles est donnée par la fonction `levels` :

```
R> levels(d$sexe)
```

```
[1] "Homme" "Femme"
```

Si on veut modifier la valeur du sexe du premier individu de notre tableau de données avec une valeur non autorisée, on obient un message d'erreur et une valeur manquante est utilisée à la place :

```
R> d$sexe[1] <- "Chihuahua"
```

```
Warning in `[<-.factor`(`*tmp*`, 1, value =
structure(c(NA, 2L, 1L, 1L, : invalid factor
level, NA generated
```

```
R> d$sexe[1]
```

```
[1] <NA>
Levels: Homme Femme
```

```
R> d$sexe[1] <- "Homme"
   d$sexe[1]
```

```
[1] Homme
Levels: Homme Femme
```

On peut très facilement créer un facteur à partir d'une variable de type caractères avec la fonction `factor` :

```
R> v <- factor(c("H", "H", "F", "H"))
   v
```

```
[1] H H F H
Levels: F H
```

Par défaut, les niveaux d'un facteur nouvellement créés sont l'ensemble des valeurs de la variable caractères, ordonnées par ordre alphabétique. Cette ordre des niveaux est utilisé à chaque fois qu'on utilise des fonctions comme `table`, par exemple :

```
R> table(v)
```

```
v
F H
1 3
```

On peut modifier cet ordre au moment de la création du facteur en utilisant l'option `levels` :

```
R> v <- factor(c("H", "H", "F", "H"), levels = c("H",
     "F"))
   table(v)
```

```
v
H F
3 1
```

On peut aussi modifier l'ordre des niveaux d'une variable déjà existante :

```
R> d$qualif <- factor(d$qualif, levels = c("Ouvrier
   specialise",
     "Ouvrier qualifie", "Employe", "Technicien",
   "Profession intermediaire",
     "Cadre", "Autre"))
   table(d$qualif)
```

```
      Ouvrier specialise         Ouvrier qualifie
                     203                      292
                 Employe               Technicien
                     594                       86
Profession intermediaire                    Cadre
                     160                      260
                   Autre
                      58
```

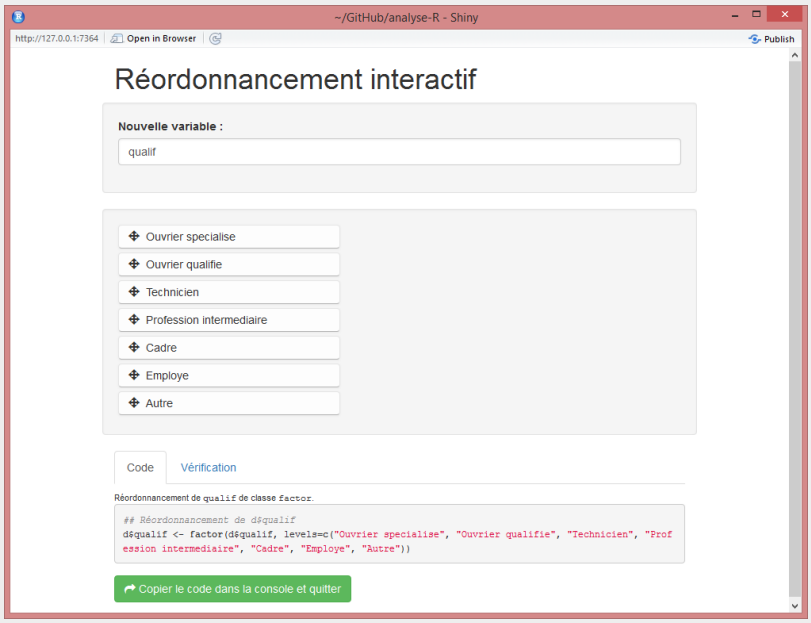
**ASTUCE**

L'extension **questionr** propose une *interface interactive* pour le réordonnancement des niveaux d'un facteur. Cette fonction, nommée `iorder`, vous permet de réordonner les modalités de manière graphique et de générer le code **R** correspondant.

Dans l'exemple précédant, si vous exécutez :

```
R> iorder(d, "qualif")
```

**RStudio** devrait ouvrir une fenêtre semblable à celle de la figure ci-dessous.




Vous pouvez alors déplacer les modalités par glisser-déposer, vérifier le résultat dans l'onglet *Vérification* et, une fois le résultat satisfaisant, récupérer le code généré pour l'inclure dans votre script.

On peut également modifier les niveaux eux-mêmes. Imaginons que l'on souhaite créer une nouvelle variable *qualif.abr* contenant les noms abrégés des catégories socioprofessionnelles de *qualif*. On peut alors procéder comme suit :

```
R> d$qualif.abr <- factor(d$qualif, levels = c("Ouvrier
   specialise",
     "Ouvrier qualifie", "Employe", "Technicien",
   "Profession intermediaire",
     "Cadre", "Autre"), labels = c("OS", "OQ", "Empl",
     "Tech", "Interm", "Cadre", "Autre"))
   table(d$qualif.abr)
```

```
    OS     OQ   Empl   Tech Interm  Cadre  Autre
   203    292    594     86    160    260     58
```

Dans ce qui précède, le paramètre `levels` de `factor` permet de spécifier quels sont les niveaux retenus dans le facteur résultat, ainsi que leur ordre. Le paramètre `labels`, lui, permet de modifier les noms de ces niveaux dans le facteur résultat. Il est donc capital d'indiquer les noms de `labels` exactement dans le même ordre que les niveaux de `levels`. Pour s'assurer de ne pas avoir commis d'erreur, il est recommandé d'effectuer un tableau croisé entre l'ancien et le nouveau facteur :

```
R> table(d$qualif, d$qualif.abr)
```

```
                          OS  OQ Empl Tech
  Ouvrier specialise     203   0    0    0
  Ouvrier qualifie         0 292    0    0
  Employe                  0   0  594    0
  Technicien               0   0    0   86
  Profession intermediaire 0   0    0    0
  Cadre                    0   0    0    0
  Autre                    0   0    0    0

                         Interm Cadre Autre
  Ouvrier specialise          0     0     0
  Ouvrier qualifie            0     0     0
  Employe                     0     0     0
  Technicien                  0     0     0
  Profession intermediaire  160     0     0
  Cadre                       0   260     0
  Autre                       0     0    58
```

On a donc ici un premier moyen d'effectuer un recodage des modalités d'une variable de type facteur. D'autres méthodes existent, voir chapitre (MAJ_LIEN).

À noter que par défaut, les valeurs manquantes ne sont pas considérées comme un niveau de facteur. On peut cependant les transformer en niveau en utilisant la fonction `addNA`. Ceci signifie cependant qu'elle ne seront plus considérées comme manquantes par **R** mais comme une modalité à part entière :

```
R> summary(d$trav.satisf)
```

```
  Satisfaction Insatisfaction      Equilibre
           480            117            451
          NA's
           952
```

```
R> summary(addNA(d$trav.satisf))
```

```
  Satisfaction Insatisfaction      Equilibre
           480            117            451
          <NA>
           952
```

# Indexation

L'indexation est l'une des fonctionnalités les plus puissantes mais aussi les plus difficiles à maîtriser de **R**. Il s'agit d'opérations permettant de sélectionner des sous-ensembles d'observations et/ou de variables en fonction de différents critères. L'indexation peut porter sur des vecteurs, des matrices ou des tableaux de données.

Le principe est toujours le même : on indique, entre crochets et à la suite du nom de l'objet à indexer, une série de conditions indiquant ce que l'on garde ou non. Ces conditions peuvent être de différents types.

## Indexation directe

Le mode le plus simple d'indexation consiste à indiquer la position des éléments à conserver. Dans le cas d'un vecteur cela permet de sélectionner un ou plusieurs éléments de ce vecteur.

Soit le vecteur suivant :

```
R> v <- c("a", "b", "c", "d", "e", "f", "g")
```

Si on souhaite le premier élément du vecteur, on peut faire :

```
R> v[1]
[1] "a"
```

Si on souhaite les trois premiers éléments ou les éléments 2, 6 et 7 :

```
R> v[1:3]
[1] "a" "b" "c"
R> v[c(2, 6, 7)]
[1] "b" "f" "g"
```

Si on veut le dernier élément :

```
R> v[length(v)]
[1] "g"
```

Dans le cas de matrices ou de tableaux de données, l'indexation prend deux arguments séparés par une virgule : le premier concerne les lignes et le second les colonnes. Ainsi, si on veut l'élément correspondant à la troisième ligne et à la cinquième colonne du tableau de données `d` :

```
R> d[3, 5]
```

```
[1] 3994.102
```

On peut également indiquer des vecteurs :

```
R> d[1:3, 1:2]
```

```
  id age
1  1  28
2  2  23
3  3  59
```

Si on laisse l'un des deux critères vides, on sélectionne l'intégralité des lignes ou des colonnes. Ainsi si l'on veut seulement la cinquième colonne ou les deux premières lignes :

```
R> str(d[, 5])
```

```
 num [1:2000] 2634 9738 3994 5732 4329 ...
```

```
R> str(d[1:2, ])
```

```
'data.frame':   2 obs. of  21 variables:
 $ id           : int  1 2
 $ age          : int  28 23
 $ sexe         : Factor w/ 2 levels "Homme","Femme": 1 2
 $ nivetud      : Factor w/ 8 levels "N'a jamais fait
d'etudes",..: 8 NA
 $ poids        : num  2634 9738
 $ occup        : Factor w/ 7 levels "Exerce une
profession",..: 1 3
 $ qualif       : Factor w/ 7 levels "Ouvrier
specialise",..: 3 NA
 $ freres.soeurs: int  8 2
 $ clso         : Factor w/ 3 levels "Oui","Non","Ne sait
pas": 1 1
 $ relig        : Factor w/ 6 levels "Pratiquant
regulier",..: 4 4
 $ trav.imp     : Factor w/ 4 levels "Le plus
important",..: 4 NA
 $ trav.satisf  : Factor w/ 3 levels "Satisfaction",..: 2
NA
 $ hard.rock    : Factor w/ 2 levels "Non","Oui": 1 1
 $ lecture.bd   : Factor w/ 2 levels "Non","Oui": 1 1
 $ peche.chasse : Factor w/ 2 levels "Non","Oui": 1 1
 $ cuisine      : Factor w/ 2 levels "Non","Oui": 2 1
 $ bricolage    : Factor w/ 2 levels "Non","Oui": 1 1
 $ cinema       : Factor w/ 2 levels "Non","Oui": 1 2
 $ sport        : Factor w/ 2 levels "Non","Oui": 1 2
 $ heures.tv    : num  0 1
 $ qualif.abr   : Factor w/ 7 levels "OS","OQ","Empl",..:
3 NA
```

Enfin, si on préfixe les arguments avec le signe `-`, ceci signifie « tous les éléments sauf ceux indiqués ». Si par exemple on veut tous les éléments de `v` sauf le premier :

```
R> v[-1]
```

```
[1] "b" "c" "d" "e" "f" "g"
```

Bien sûr, tous ces critères se combinent et on peut stocker le résultat dans un nouvel objet. Dans cet exemple `d2` contiendra les trois premières lignes de `d` ainsi que la 50e ligne mais sans les colonnes 2 et 5 à 14.

```
R> d2 <- d[c(1:3, 50), -c(2, 5:13)]
   str(d2)
```

```
'data.frame':   4 obs. of  11 variables:
 $ id          : int  1 2 3 50
 $ sexe        : Factor w/ 2 levels "Homme","Femme": 1 2 1
2
 $ nivetud     : Factor w/ 8 levels "N'a jamais fait
d'etudes",..: 8 NA 3 8
 $ lecture.bd  : Factor w/ 2 levels "Non","Oui": 1 1 1 1
 $ peche.chasse: Factor w/ 2 levels "Non","Oui": 1 1 1 1
 $ cuisine     : Factor w/ 2 levels "Non","Oui": 2 1 1 1
 $ bricolage   : Factor w/ 2 levels "Non","Oui": 1 1 1 1
 $ cinema      : Factor w/ 2 levels "Non","Oui": 1 2 1 1
 $ sport       : Factor w/ 2 levels "Non","Oui": 1 2 2 1
 $ heures.tv   : num  0 1 0 0
 $ qualif.abr  : Factor w/ 7 levels "OS","OQ","Empl",..: 3
NA 4 6
```

## Indexation par nom

Un autre mode d'indexation consiste à fournir non pas un numéro mais un nom sous forme de chaîne de caractères. On l'utilise couramment pour sélectionner les variables d'un tableau de données. Ainsi, les deux écritures suivantes sont équivalentes[1] :

```
R> d$clso
   d[, "clso"]
```

Là aussi on peut utiliser un vecteur pour sélectionner plusieurs noms et récupérer un « sous-tableau » de données :

```
R> d2 <- d[, c("id", "sexe", "age")]
```

Les noms peuvent également être utilisés pour les observations (lignes) d'un tableau de données si celles-ci ont été munies d'un nom avec la fonction `row.names`. Par défaut les noms de ligne sont leur numéro d'ordre, mais on peut leur assigner comme nom la valeur d'une variable d'identifiant. Ainsi, on peut assigner aux lignes du jeu de données `rp99` le nom des communes correspondantes :

```
R> data(rp99)
   row.names(rp99) <- rp99$nom
```

On peut alors accéder directement aux communes en donnant leur nom :

```
R> rp99[c("VILLEURBANNE", "OULLINS"), ]
```

Par contre il n'est pas possible d'utiliser directement l'opérateur `-` comme pour l'indexation directe. Pour exclure une colonne en fonction de son nom, on doit uiliser une autre forme d'indexation, l'*indexation par condition*, expliquée dans la section suivante. On peut ainsi faire…

```
R> d[, names(d) != "qualif"]
```

… pour sélectionner toutes les colonnes sauf celle qui s'appelle *qualif*.

# Indexation par condition

## Tests et conditions

Une condition est une expression logique dont le résultat est soit `TRUE` (vrai) soit `FALSE` (faux).

Une condition comprend la plupart du temps un opérateur de comparaison. Les plus courants sont les suivants :

| Opérateur | Signification |
|---|---|
| `==` | égal à |
| `!=` | différent de |
| `>` | strictement supérieur à |
| `<` | strictement inférieur à |
| `>=` | supérieur ou égal à |
| `<=` | inférieur ou égal à |

Voyons tout de suite un exemple :

```
R> str(d$sexe == "Homme")
```

```
 logi [1:2000] TRUE FALSE TRUE TRUE FALSE FALSE ...
```

Que s'est-il passé ? Nous avons fourni à **R** une condition qui signifie « la valeur de la variable *sexe* vaut "Homme" ». Et il nous a renvoyé un vecteur avec autant d'éléments qu'il y'a d'observations dans `d`, et dont la valeur est `TRUE` si l'observation correspond à un homme et `FALSE` dans les autres cas.

Prenons un autre exemple. On n'affichera cette fois que les premiers éléments de notre variable d'intérêt à l'aide de la fonction `head` :

```
R> head(d$age)
```

```
[1] 28 23 59 34 71 35
```

```
R> head(d$age > 40)
```

```
[1] FALSE FALSE  TRUE FALSE  TRUE FALSE
```

On voit bien ici qu'à chaque élément du vecteur `d$age` dont la valeur est supérieure à 40 correspond un élément `TRUE` dans le résultat de la condition.

On peut combiner ou modifier des conditions à l'aide des opérateurs logiques habituels :

| Opérateur | Signification |
|---|---|
| `&` | et logique |
| `\|` | ou logique |
| `!` | négation logique |

Comment les utilise-t-on ? Voyons tout de suite des exemples. Supposons que je veuille déterminer quels sont dans mon échantillon les hommes ouvriers spécialisés :

```
R> d$sexe == "Homme" & d$qualif == "Ouvrier specialise"
```

Si je souhaite identifier les personnes qui bricolent ou qui font la cuisine :

```
R> d$bricol == "Oui" | d$cuisine == "Oui"
```

Si je souhaite isoler les femmes qui ont entre 20 et 34 ans :

```
R> d$sexe == "Femme" & d$age >= 20 & d$age <= 34
```

Si je souhaite récupérer les enquêtés qui ne sont pas cadres, on peut utiliser l'une des deux formes suivantes :

```
R> d$qualif != "Cadre"
   !(d$qualif == "Cadre")
```

Lorsqu'on mélange « et » et « ou » il est nécessaire d'utiliser des parenthèses pour différencier les blocs. La condition suivante identifie les femmes qui sont soit cadre, soit employée :

```
R> d$sexe == "Femme" & (d$qualif == "Employe" | d$qualif ==
     "Cadre")
```

L'opérateur `%in%` peut être très utile : il teste si une valeur fait partie des éléments d'un vecteur. Ainsi on pourrait remplacer la condition précédente par :

```
R> d$sexe == "Femme" & d$qualif %in% c("Employe", "Cadre")
```

Enfin, signalons qu'on peut utiliser les fonctions `table` ou `summary` pour avoir une idée du résultat de notre condition :

```
R> table(d$sexe)
```

```
Homme  Femme
  900   1100
```

```
R> table(d$sexe == "Homme")
```

```
FALSE  TRUE
 1100   900
```

```
R> summary(d$sexe == "Homme")
```

```
   Mode   FALSE    TRUE    NA's
logical    1100     900       0
```

## Utilisation pour l'indexation

L'utilisation des conditions pour l'indexation est assez simple : si on indexe un vecteur avec un vecteur booléen, seuls les éléments correspondant à TRUE seront conservés.

Ainsi, si on fait :

```
R> dh <- d[d$sexe == "Homme", ]
```

On obtiendra un nouveau tableau de données comportant l'ensemble des variables de `d`, mais seulement les observations pour lesquelles la variable *sexe* vaut « Homme ».

La plupart du temps ce type d'indexation s'applique aux lignes, mais on peut aussi l'utiliser sur les colonnes d'un tableau de données. L'exemple suivant, un peu compliqué, sélectionne uniquement les variables dont le nom commence par *a* ou *s* :

```
R> d2 <- d[, substr(names(d), 0, 1) %in% c("a", "s")]
```

On peut évidemment combiner les différents type d'indexation. L'exemple suivant sélectionne les femmes de plus de 40 ans et ne conserve que les variables *qualif* et *relig*.

```
R> d2 <- d[d$sexe == "Femme" & d$age > 40, c("qualif",
     "relig")]
```

## Valeurs manquantes dans les conditions

Une remarque importante : quand l'un des termes d'une condition comporte une valeur manquante ( `NA` ), le résultat de cette condition n'est pas toujours `TRUE` ou `FALSE` , il peut aussi être à son tour une valeur manquante.

```
R> v <- c(1:5, NA)
   v
```

```
[1]  1  2  3  4  5 NA
```

```
R> v > 3
```

```
[1] FALSE FALSE FALSE  TRUE  TRUE    NA
```

On voit que le test `NA > 3` ne renvoie ni vrai ni faux, mais `NA` .

Le résultat d'une condition peut donc comporter un grand nombre de valeurs manquantes :

```
R> summary(d$trav.satisf == "Satisfaction")
```

```
   Mode   FALSE    TRUE    NA's
logical     568     480     952
```

Une autre conséquence importante de ce comportement est qu'on ne peut pas utiliser l'opérateur l'expression `== NA` pour tester la présence de valeurs manquantes. On utilisera à la place la fonction *ad hoc* `is.na` .

On comprendra mieux le problème avec l'exemple suivant :

```
R> v <- c(1, NA)
   v
```

```
[1]  1 NA
```

```
R> v == NA
```

```
[1] NA NA
```

```
R> is.na(v)
```

```
[1] FALSE  TRUE
```

Pour compliquer encore un peu le tout, lorsqu'on utilise une condition pour l'indexation, si la condition renvoie `NA`, **R** ne sélectionne pas l'élément mais retourne quand même la valeur `NA`. Ceci aura donc des conséquences pour l'extraction de sous-populations (voir section ci-après MAJ LIEN)[2].

## Indexation et assignation

Dans tous les exemples précédents, on a utilisé l'indexation pour extraire une partie d'un vecteur ou d'un tableau de données, en plaçant l'opération d'indexation à droite de l'opérateur `<-`.

Mais l'indexation peut également être placée à gauche de cet opérateur. Dans ce cas, les éléments sélectionnés par l'indexation sont alors remplacés par les valeurs indiquées à droite de l'opérateur `<-`.

Prenons donc un exemple simple :

```
R> v <- 1:5
   v
```

```
[1] 1 2 3 4 5
```

```
R> v[1] <- 3
   v
```

```
[1] 3 2 3 4 5
```

Cette fois, au lieu d'utiliser quelque chose comme `x <- v[1]`, qui aurait placé la valeur du premier élément de `v` dans `x`, on a utilisé `v[1] <- 3`, ce qui a mis à jour le premier élément de `v` avec la valeur 3. Ceci fonctionne également pour les tableaux de données et pour les différents types d'indexation évoqués précédemment :

```
R> d[257, "sexe"] <- "Homme"
```

Enfin on peut modifier plusieurs éléments d'un seul coup soit en fournissant un vecteur, soit en profitant du mécanisme de recyclage. Les deux commandes suivantes sont ainsi rigoureusement équivalentes :

```
R> d[c(257, 438, 889), "sexe"] <- c("Homme", "Homme",
     "Homme")
   d[c(257, 438, 889), "sexe"] <- "Homme"
```

On commence à voir comment l'utilisation de l'indexation par conditions et de l'assignation va nous permettre de faire des recodages.

```
R> d$gr.age[d$age >= 20 & d$age <= 30] <- "20-30 ans"
   d$gr.age[is.na(d$age)] <- "Inconnu"
```

# Sous-populations

## Par indexation

La première manière de construire des sous-populations est d'utiliser l'indexation par conditions. On peut ainsi facilement sélectionner une partie des observations suivant un ou plusieurs critères et placer le résultat dans un nouveau tableau de données.

Par exemple si on souhaite isoler les hommes et les femmes :

```
R> dh <- d[d$sexe == "Homme", ]
   df <- d[d$sexe == "Femme", ]
   table(d$sexe)
```

```
Homme Femme
  901  1099
```

```
R> dim(dh)
```

```
[1] 901  22
```

```
R> dim(df)
```

```
[1] 1099   22
```

On a à partir de là trois tableaux de données, `d` comportant la population totale, `dh` seulement les hommes et `df` seulement les femmes.

On peut évidemment combiner plusieurs critères :

```
R> dh.25 <- d[d$sexe == "Homme" & d$age <= 25, ]
   dim(dh.25)
```

```
[1] 87 22
```

Si on utilise directement l'indexation, il convient cependant d'être extrêmement prudent avec les valeurs manquantes. Comme indiqué précédemment, la présence d'une valeur manquante dans une condition fait que celle-ci est évaluée en `NA` et qu'au final la ligne correspondante est conservée par l'indexation :

```
R> summary(d$trav.satisf)
```

```
  Satisfaction Insatisfaction      Equilibre
           480            117            451
          NA's
           952
```

```
R> d.satisf <- d[d$trav.satisf == "Satisfaction", ]
   dim(d.satisf)
```

```
[1] 1432   22
```

Comme on le voit, ici `d.satisf` contient les individus ayant la modalité *Satisfaction* mais aussi ceux ayant une valeur manquante `NA`. C'est pourquoi il faut toujours soit vérifier au préalable qu'on n'a pas de valeurs manquantes dans les variables de la condition, soit exclure explicitement les `NA` de la manière suivante :

```
R> d.satisf <- d[d$trav.satisf == "Satisfaction" &
   !is.na(d$trav.satisf),
     ]
   dim(d.satisf)
```

```
[1] 480  22
```

C'est notamment pour cette raison qu'on préfèrera le plus souvent utiliser la fonction `subset`.

## Fonction subset

La fonction `subset` permet d'extraire des sous-populations de manière plus simple et un peu plus intuitive que l'indexation directe.

Celle-ci prend trois arguments principaux : * le nom de l'objet de départ ; * une condition sur les observations ( `subset` ) ; * éventuellement une condition sur les colonnes ( `select` ).

Reprenons tout de suite un exemple déjà vu :

```
R> dh <- subset(d, sexe == "Homme")
   df <- subset(d, sexe == "Femme")
```

L'utilisation de `subset` présente plusieurs avantages. Le premier est d'économiser quelques touches. On n'est en effet pas obligé de saisir le nom du tableau de données dans la condition sur les lignes. Ainsi les deux commandes suivantes sont équivalentes :

```
R> dh <- subset(d, d$sexe == "Homme")
   dh <- subset(d, sexe == "Homme")
```

Le second avantage est que `subset` s'occupe du problème des valeurs manquantes évoquées précédemment et les exclut de lui-même, contrairement au comportement par défaut :

```
R> summary(d$trav.satisf)
```

```
  Satisfaction Insatisfaction      Equilibre
           480            117            451
          NA's
           952
```

```
R> d.satisf <- d[d$trav.satisf == "Satisfaction", ]
   dim(d.satisf)
```

```
[1] 1432   22
```

```
R> d.satisf <- subset(d, trav.satisf == "Satisfaction")
   dim(d.satisf)
```

```
[1] 480  22
```

Enfin, l'utilisation de l'argument `select` est simplifié pour l'expression de condition sur les colonnes. On peut ainsi spécifier les noms de variable sans guillemets et leur appliquer directement l'opérateur d'exclusion `-` :

```
R> d2 <- subset(d, select = c(sexe, sport))
   d2 <- subset(d, age > 25, select = -c(id, age, cinema))
```

## Fonction tapply

> **NOTE**
>
> Cette section documente une fonction qui peut être très utile, mais pas forcément indispensable au départ.

La fonction `tapply` n'est qu'indirectement liée à la notion de sous-population, mais peut permettre d'éviter d'avoir à créer ces sous-populations dans certains cas.

Son fonctionnement est assez simple, mais pas forcément intuitif. La fonction prend trois arguments : un vecteur, un facteur et une fonction. Elle applique ensuite la fonction aux éléments du vecteur correspondant à un même niveau du facteur. Vite, un exemple !

```
R> tapply(d$age, d$sexe, mean)
```

```
   Homme    Femme
48.10655 48.19836
```

Qu'est-ce que ça signifie ? Ici `tapply` a sélectionné toutes les observations correspondant à « Homme », puis appliqué la fonction `mean` aux valeurs de *age* correspondantes. Puis elle a fait de même pour les observations correspondant à « Femme ». On a donc ici la moyenne d'âge chez les hommes et chez les femmes.

On peut fournir à peu près n'importe quelle fonction à `tapply` :

```
R> tapply(d$bricol, d$sexe, freq)
```

```
$Homme
      n    % val%
Non 386 42.8 42.8
Oui 515 57.2 57.2
NA    0  0.0   NA

$Femme
      n    % val%
Non 761 69.2 69.2
Oui 338 30.8 30.8
NA    0  0.0   NA
```

Les arguments supplémentaires fournis à `tapply` sont en fait fournis directement à la fonction appelée.

```
R> tapply(d$bricol, d$sexe, freq, total = TRUE)
```

```
$Homme
        n     %  val%
Non   386  42.8  42.8
Oui   515  57.2  57.2
NA      0   0.0    NA
Total 901 100.0 100.0

$Femme
         n     %  val%
Non    761  69.2  69.2
Oui    338  30.8  30.8
NA       0   0.0    NA
Total 1099 100.0 100.0
```

**NOTE**

La fonction `by` est un équivalent (pour les tableaux de données) de `tapply`. La présentation des résultats diffère légèrement.

```
R> tapply(d$age, d$sexe, mean)
```

```
   Homme    Femme
48.10655 48.19836
```

```
R> by(d$age, d$sexe, mean)
```

```
d$sexe: Homme
[1] 48.10655
------------------------------------
d$sexe: Femme
[1] 48.19836
```

# Recodages

Le recodage de variables est une opération extrêmement fréquente lors du traitement d'enquête. Celuici utilise soit l'une des formes d'indexation décrites précédemment, soit des fonctions *ad hoc* de **R**.

On passe ici en revue différents types de recodage parmi les plus courants. Les exemples s'appuient, comme précédemment, sur l'extrait de l'enquête *Histoire de vie* :

```
R> data(hdv2003)
   d <- hdv2003
```

## Convertir une variable

Il peut arriver qu'on veuille transformer une variable d'un type dans un autre.

Par exemple, on peut considérer que la variable numérique *freres.soeurs* est une « fausse » variable numérique et qu'une représentation sous forme de facteur serait plus adéquate. Dans ce cas il suffit de faire appel à la fonction `factor` :

```
R> d$fs.fac <- factor(d$freres.soeurs)
   levels(d$fs.fac)
```

```
 [1] "0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"
[10] "9"  "10" "11" "12" "13" "14" "15" "16" "18"
[19] "22"
```

La conversion d'une variable caractères en facteur se fait de la même manière.

La conversion d'un facteur ou d'une variable numérique en variable caractères peut se faire à l'aide de la fonction `as.character` :

```
R> d$fs.char <- as.character(d$freres.soeurs)
   d$qualif.char <- as.character(d$qualif)
```

La conversion d'un facteur en caractères est fréquemment utilisé lors des recodages du fait qu'il est impossible d'ajouter de nouvelles modalités à un

facteur de cette manière. Par exemple, la première des commandes suivantes génère un message d'avertissement, tandis que les deux autres fonctionnent :

```
R> d.temp <- d
   d.temp$qualif[d.temp$qualif == "Ouvrier specialise"] <-
   "Ouvrier"
```

```
Warning in `[<-.factor`(`*tmp*`, d.temp$qualif ==
"Ouvrier specialise", : invalid factor level, NA
generated
```

```
R> d$qualif.char <- as.character(d$qualif)
   d$qualif.char[d$qualif.char == "Ouvrier specialise"] <-
   "Ouvrier"
```

Dans le premier cas, le message d'avertissement indique que toutes les modalités « Ouvrier specialise » de notre variable *qualif* ont été remplacées par des valeurs manquantes `NA`.

Enfin, une variable de type caractères dont les valeurs seraient des nombres peut être convertie en variable numérique avec la fonction `as.numeric`.

```
R> v <- c("1", "3.1415", "4", "5.6", "1", "4")
   v
```

```
[1] "1"      "3.1415" "4"      "5.6"    "1"
[6] "4"
```

```
R> as.numeric(v)
```

```
[1] 1.0000 3.1415 4.0000 5.6000 1.0000 4.0000
```

**ATTENTION**

Lorsque l'on convertit un facteur avec `as.numeric`, on obtient le numéro de chaque facteur (première modalité, seconde modalité, etc.). Si la valeur numérique qui nous intéresse est en fait contenu dans le nom des modalités, il faut connvertir au préalable notre facteur en variable textuelle.

```
R> vf <- factor(v)
   vf
```

```
[1] 1      3.1415 4      5.6    1      4
Levels: 1 3.1415 4 5.6
```

```
R> as.numeric(vf)
```

```
[1] 1 2 3 4 1 3
```

```
R> as.numeric(as.character(vf))
```

```
[1] 1.0000 3.1415 4.0000 5.6000 1.0000 4.0000
```

## Découper une variable numérique en classes

Le premier type de recodage consiste à découper une variable de type numérique en un certain nombre de classes. On utilise pour cela la fonction `cut`.

Celle-ci prend, outre la variable à découper, un certain nombre d'arguments :

- `breaks` indique soit le nombre de classes souhaité, soit, si on lui fournit un vecteur, les limites des classes ;
- `labels` permet de modifier les noms de modalités attribués aux classes ;
- `include.lowest` et `right` influent sur la manière dont les valeurs situées à la frontière des classes seront inclues ou exclues ;

- `dig.lab` indique le nombre de chiffres après la virgule à conserver dans les noms de modalités.

Prenons tout de suite un exemple et tentons de découper notre variable *age* en cinq classes et de placer le résultat dans une nouvelle variable nommée *age5cl* :

```
R> d$age5cl <- cut(d$age, 5)
   table(d$age5cl)
```

```
(17.9,33.8] (33.8,49.6] (49.6,65.4] (65.4,81.2]
        454         628         556         319
(81.2,97.1]
         43
```

Par défaut **R** nous a bien créé cinq classes d'amplitudes égales. La première classe va de 16,9 à 32,2 ans (en fait de 17 à 32), etc.

Les frontières de classe seraient plus présentables si elles utilisaient des nombres ronds. On va donc spécifier manuellement le découpage souhaité, par tranches de 20 ans :

```
R> d$age20 <- cut(d$age, c(0, 20, 40, 60, 80, 100))
   table(d$age20)
```

```
  (0,20]  (20,40]  (40,60]  (60,80] (80,100]
      72      660      780      436       52
```

On aurait pu tenir compte des âges extrêmes pour la première et la dernière valeur :

```
R> range(d$age)
```

```
[1] 18 97
```

```
R> d$age20 <- cut(d$age, c(17, 20, 40, 60, 80, 93))
   table(d$age20)
```

```
(17,20] (20,40] (40,60] (60,80] (80,93]
     72     660     780     436      50
```

Les symboles dans les noms attribués aux classes ont leur importance : `(` signifie que la frontière de la classe est exclue, tandis que `[` signifie qu'elle est incluse. Ainsi, `(20,40]` signifie « strictement supérieur à 20 et inférieur ou égal à 40 ».

On remarque que du coup, dans notre exemple précédent, la valeur minimale, 17, est exclue de notre première classe, et qu'une observation est donc absente de ce découpage. Pour résoudre ce problème on peut soit faire commencer la première classe à 16, soit utiliser l'option `include.lowest=TRUE` :

```
R> d$age20 <- cut(d$age, c(16, 20, 40, 60, 80, 93))
   table(d$age20)
```

```
 (16,20] (20,40] (40,60] (60,80] (80,93]
      72     660     780     436      50
```

```
R> d$age20 <- cut(d$age, c(17, 20, 40, 60, 80, 93),
   include.lowest = TRUE)
   table(d$age20)
```

```
 [17,20] (20,40] (40,60] (60,80] (80,93]
      72     660     780     436      50
```

On peut également modifier le sens des intervalles avec l'option `right=FALSE`, et indiquer manuellement les noms des modalités avec `labels` :

```
R> d$age20 <- cut(d$age, c(16, 20, 40, 60, 80, 93), right =
   FALSE,
     include.lowest = TRUE)
   table(d$age20)
```

```
 [16,20) [20,40) [40,60) [60,80) [80,93]
      48     643     793     454      60
```

```
R> d$age20 <- cut(d$age, c(17, 20, 40, 60, 80, 93),
   include.lowest = TRUE,
     labels = c("<20ans", "21-40 ans", "41-60ans",
   "61-80ans",
       ">80ans"))
   table(d$age20)
```

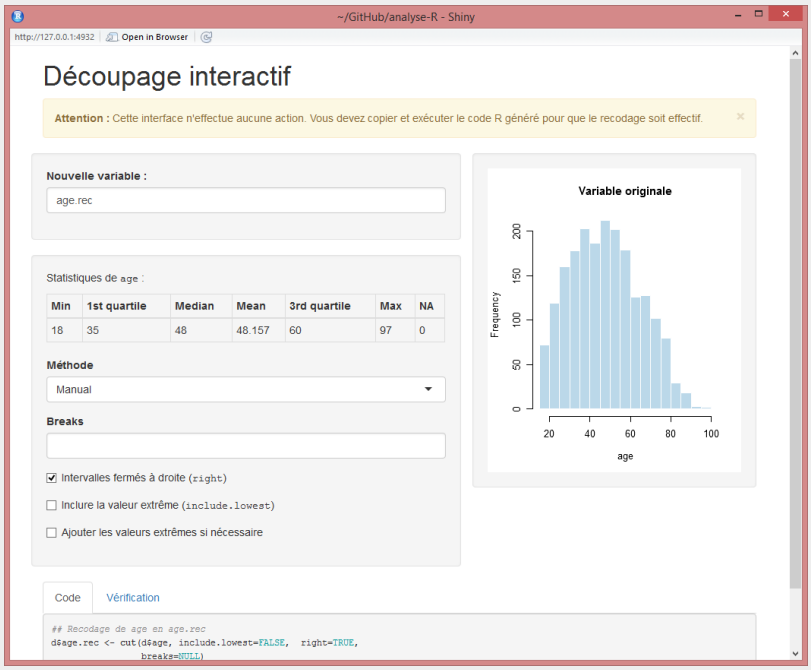
```
<20ans 21-40 ans  41-60ans  61-80ans    >80ans
    72       660       780       436        50
```

**ASTUCE**

L'extension **questionr** propose une interface interactive à la fonction `cut`, nommée `icut`. Elle s'utilise de la manière suivante :

```
R> icut(d, age)
```

**RStudio** devrait ouvrir une fenêtre semblable à l'image ci-dessous.




Vous pouvez alors indiquer les limites de vos classes ainsi que quelques options complémentaires. Ces limites sont représentées graphiquement sur l'histogramme de la variable d'origine.

L'onglet *Vérification* affiche un tri à plat et un graphique en barres de la nouvelle variable. Une fois le résultat satisfaisant, vous pouvez récupérer le code généré pour l'inclure dans votre script.

L'extension **questionr** propose aussi une fonction `quant.cut` permettant de découper une variable numérique en un nombre de classes donné ayant des efffectifs semblables. Il suffit de lui passer le nombre de classes en argument :

```
R> d$age6cl <- quant.cut(d$age, 6)
   table(d$age6cl)
```

```
     [18,30)      [30,39)      [39,48) [48,55.667)
         302          337          350         344
 [55.667,66)      [66,97]
         305          362
```

`quant.cut` admet les mêmes autres options que `cut` (`include.lowest`, `right`, `labels`...).

## Regrouper les modalités d'une variable

Pour regrouper les modalités d'une variable qualitative (d'un facteur le plus souvent), on peut utiliser directement l'indexation.

Ainsi, si on veut recoder la variable qualif dans une variable qualif.reg plus « compacte », on peut utiliser :

```
R> table(d$qualif)
```

```
      Ouvrier specialise         Ouvrier qualifie
                     203                      292
              Technicien Profession intermediaire
                      86                      160
                   Cadre                   Employe
                     260                      594
                   Autre
                      58
```

```
R> d$qualif.reg[d$qualif == "Ouvrier specialise"] <-
   "Ouvrier"
   d$qualif.reg[d$qualif == "Ouvrier qualifie"] <- "Ouvrier"
   d$qualif.reg[d$qualif == "Employe"] <- "Employe"
   d$qualif.reg[d$qualif == "Profession intermediaire"] <-
   "Intermediaire"
   d$qualif.reg[d$qualif == "Technicien"] <- "Intermediaire"
   d$qualif.reg[d$qualif == "Cadre"] <- "Cadre"
   d$qualif.reg[d$qualif == "Autre"] <- "Autre"
   table(d$qualif.reg)
```

```
        Autre         Cadre       Employe
           58           260           594
Intermediaire       Ouvrier
          246           495
```

On aurait pu représenter ce recodage de manière plus compacte, notamment en commençant par copier le contenu de *qualif* dans *qualif.reg*, ce qui permet de ne pas s'occuper de ce qui ne change pas.

Il est cependant nécessaire de ne pas copier *qualif* sous forme de facteur, sinon on ne pourrait ajouter de nouvelles modalités. On copie donc la version caractères de qualif grâce à la fonction `as.character` :

```
R> d$qualif.reg <- as.character(d$qualif)
   d$qualif.reg[d$qualif == "Ouvrier specialise"] <-
   "Ouvrier"
   d$qualif.reg[d$qualif == "Ouvrier qualifie"] <- "Ouvrier"
   d$qualif.reg[d$qualif == "Profession intermediaire"] <-
   "Intermediaire"
   d$qualif.reg[d$qualif == "Technicien"] <- "Intermediaire"
   table(d$qualif.reg)
```

```
        Autre         Cadre       Employe
           58           260           594
Intermediaire       Ouvrier
          246           495
```

On peut faire une version encore plus compacte en utilisant l'opérateur logique *ou* ( `|` ) :

```
R> d$qualif.reg <- as.character(d$qualif)
   d$qualif.reg[d$qualif == "Ouvrier specialise" | d$qualif
   ==
     "Ouvrier qualifie"] <- "Ouvrier"
   d$qualif.reg[d$qualif == "Profession intermediaire" |
     d$qualif == "Technicien"] <- "Intermediaire"
   table(d$qualif.reg)
```

```
        Autre         Cadre       Employe
           58           260           594
Intermediaire       Ouvrier
          246           495
```

Enfin, pour terminer ce petit tour d'horizon, on peut également remplacer l'opérateur `|` par `%in%`, qui peut parfois être plus lisible :

```
R> d$qualif.reg <- as.character(d$qualif)
   d$qualif.reg[d$qualif %in% c("Ouvrier specialise",
     "Ouvrier qualifie")] <- "Ouvrier"
   d$qualif.reg[d$qualif %in% c("Profession intermediaire",
     "Technicien")] <- "Intermediaire"
   table(d$qualif.reg)
```

```
        Autre         Cadre       Employe
           58           260           594
Intermediaire       Ouvrier
          246           495
```

Dans tous les cas le résultat obtenu est une variable de type caractère. On pourra la convertir en facteur par un simple :

```
R> d$qualif.reg <- factor(d$qualif.reg)
```

Si on souhaite recoder les valeurs manquantes, il suffit de faire appel à la fonction `is.na` :

```
R> table(d$trav.satisf)
```

```
  Satisfaction Insatisfaction      Equilibre
           480            117            451
```

```
R> d$trav.satisf.reg <- as.character(d$trav.satisf)
   d$trav.satisf.reg[is.na(d$trav.satisf)] <- "Manquant"
   table(d$trav.satisf.reg)
```

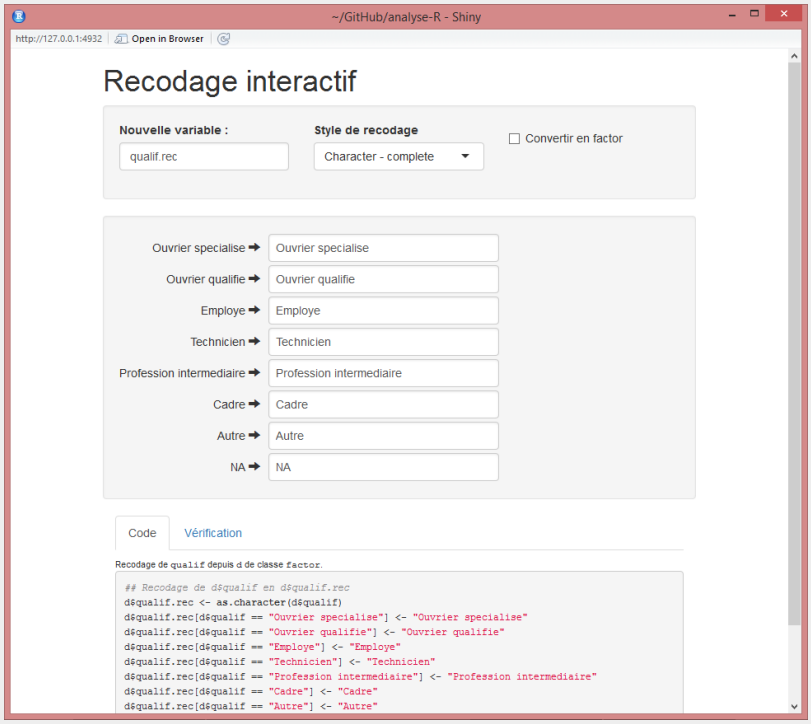
```
     Equilibre Insatisfaction       Manquant
           451            117            952
  Satisfaction
           480
```

**ASTUCE**

**questionr** propose une interface interactive pour le recodage d'une variable qualitative (renommage et regroupement de modalités). Cette fonction, nommée `irec`, s'utilise de la manière suivante :

```
R> irec(d, qualif)
```

**RStudio** va alors ouvrir une fenêtre semblable à l'image ci-dessous :




Vous pouvez alors sélectionner différentes options, et pour chaque ancienne modalité, indiquer la nouvelle valeur correspondante. Pour regrouper des modalités, il suffit de leur assigner des nouvelles valeurs identiques. Dans tous les cas n'hésitez pas à expérimenter, l'interface se contente de générer du code R à copier/coller dans votre script mais ne l'exécute pas, et ne modifie donc jamais vos données !

L'onglet *Vérification* affiche un tri croisé de l'ancienne et de la nouvelle variable pour vérifier que le recodage est correct. Une fois le résultat

satisfaisant, vous pouvez récupérer le code généré dans l'onglet *Code* pour l'inclure dans votre script.

## Variables calculées

La création d'une variable numérique à partir de calculs sur une ou plusieurs autres variables numériques se fait très simplement.

Supposons que l'on souhaite calculer une variable indiquant l'écart entre le nombre d'heures passées à regarder la télévision et la moyenne globale de cette variable. On pourrait alors faire :

```
R> range(d$heures.tv, na.rm = TRUE)
```

```
[1]  0 12
```

```
R> mean(d$heures.tv, na.rm = TRUE)
```

```
[1] 2.246566
```

```
R> d$ecart.heures.tv <- d$heures.tv - mean(d$heures.tv,
     na.rm = TRUE)
   range(d$ecart.heures.tv, na.rm = TRUE)
```

```
[1] -2.246566  9.753434
```

```
R> mean(d$ecart.heures.tv, na.rm = TRUE)
```

```
[1] 4.714578e-17
```

Autre exemple tiré du jeu de données `rp99` : si on souhaite calculer le pourcentage d'actifs dans chaque commune, on peut diviser la population active *pop.act* par la population totale *pop.tot*.

```
R> rp99$part.actifs <- rp99$pop.act/rp99$pop.tot * 100
```

## Combiner plusieurs variables

La combinaison de plusieurs variables se fait à l'aide des techniques d'indexation déjà décrites précédemment. Le plus compliqué est d'arriver à formuler des conditions parfois complexes de manière rigoureuse.

On peut ainsi vouloir combiner plusieurs variables qualitatives en une seule :

```
R> d$act.manuelles <- NA
   d$act.manuelles[d$cuisine == "Oui" & d$bricol == "Oui"]
   <- "Cuisine et Bricolage"
   d$act.manuelles[d$cuisine == "Oui" & d$bricol == "Non"]
   <- "Cuisine seulement"
   d$act.manuelles[d$cuisine == "Non" & d$bricol == "Oui"]
   <- "Bricolage seulement"
   d$act.manuelles[d$cuisine == "Non" & d$bricol == "Non"]
   <- "Ni cuisine ni bricolage"
   table(d$act.manuelles)
```

```
     Bricolage seulement    Cuisine et Bricolage
                     437                     416
       Cuisine seulement Ni cuisine ni bricolage
                     465                     682
```

On peut également combiner variables qualitatives et variables quantitatives :

```
R> d$age.sexe <- NA
   d$age.sexe[d$sexe == "Homme" & d$age < 40] <- "Homme
   moins de 40 ans"
   d$age.sexe[d$sexe == "Homme" & d$age >= 40] <- "Homme
   plus de 40 ans"
   d$age.sexe[d$sexe == "Femme" & d$age < 40] <- "Femme
   moins de 40 ans"
   d$age.sexe[d$sexe == "Femme" & d$age >= 40] <- "Femme
   plus de 40 ans"
   table(d$age.sexe)
```

```
Femme moins de 40 ans  Femme plus de 40 ans
                  376                   725
Homme moins de 40 ans  Homme plus de 40 ans
                  315                   584
```

Les combinaisons de variables un peu complexes nécessitent parfois un petit travail de réflexion. En particulier, l'ordre des commandes de recodage a parfois une influence dans le résultat final.

## Variables scores

Une variable score est une variable calculée en additionnant des poids accordés aux modalités d'une série de variables qualitatives.

Pour prendre un exemple tout à fait arbitraire, imaginons que nous souhaitons calculer un score d'activités extérieures. Dans ce score on considère que le fait d'aller au cinéma « pèse » 10, celui de pêcher ou chasser vaut 30 et celui de faire du sport vaut 20. On pourrait alors calculer notre score de la manière suivante :

```
R> d$score.ext <- 0
   d$score.ext[d$cinema == "Oui"] <- d$score.ext[d$cinema
   ==
     "Oui"] + 10
   d$score.ext[d$peche.chasse == "Oui"] <-
   d$score.ext[d$peche.chasse ==
     "Oui"] + 30
   d$score.ext[d$sport == "Oui"] <- d$score.ext[d$sport ==
     "Oui"] + 20
   table(d$score.ext)
```

```
  0  10  20  30  40  50  60
800 342 229 509  31  41  48
```

Cette notation étant un peu lourde, on peut l'alléger un peu en utilisant la fonction `ifelse`. Celle-ci prend en argument une condition et deux valeurs. Si la condition est vraie elle retourne la première valeur, sinon elle retourne la seconde.

```
R> d$score.ext <- 0
   d$score.ext <- ifelse(d$cinema == "Oui", 10, 0) +
   ifelse(d$peche.chasse ==
     "Oui", 30, 0) + ifelse(d$sport == "Oui", 20, 0)
   table(d$score.ext)
```

```
   0  10  20  30  40  50  60
 800 342 229 509  31  41  48
```

## Vérification des recodages

Il est très important de vérifier, notamment après les recodages les plus complexes, qu'on a bien obtenu le résultat escompté. Les deux points les plus sensibles étant les valeurs manquantes et les erreurs dans les conditions.

Pour vérifier tout cela le plus simple est sans doute de faire des tableaux croisés entre la variable recodée et celles ayant servi au recodage, à l'aide de la fonction `table`, et de vérifier le nombre de valeurs manquantes dans la variable recodée avec `summary`, `freq` ou `table`.

Par exemple :

```
R> d$act.manuelles <- NA
   d$act.manuelles[d$cuisine == "Oui" & d$bricol == "Oui"]
   <- "Cuisine et Bricolage"
   d$act.manuelles[d$cuisine == "Oui" & d$bricol == "Non"]
   <- "Cuisine seulement"
   d$act.manuelles[d$cuisine == "Non" & d$bricol == "Oui"]
   <- "Bricolage seulement"
   d$act.manuelles[d$cuisine == "Non" & d$bricol == "Non"]
   <- "Ni cuisine ni bricolage"
   table(d$act.manuelles, d$cuisine)
```

```
                         Non Oui
  Bricolage seulement     437   0
  Cuisine et Bricolage      0 416
  Cuisine seulement         0 465
  Ni cuisine ni bricolage 682   0
```

```
R> table(d$act.manuelles, d$bricol)
```

```
                          Non Oui
  Bricolage seulement       0 437
  Cuisine et Bricolage      0 416
  Cuisine seulement       465   0
  Ni cuisine ni bricolage 682   0
```

# Tri de tables

La fonction `sort` permet de trier les éléments d'un vecteur.

```
R> sort(c(2, 5, 6, 1, 8))
```

```
[1] 1 2 5 6 8
```

On peut appliquer cette fonction à une variable, mais celle-ci ne permet que d'ordonner les valeurs de cette variable, et pas l'ensemble du tableau de données dont elle fait partie. Pour cela nous avons besoin d'une autre fonction, nommée `order`. Celle-ci ne renvoie pas les valeurs du vecteur triées, mais les emplacements de ces valeurs.

Un exemple pour comprendre :

```
R> order(c(15, 20, 10))
```

```
[1] 3 1 2
```

Le résultat renvoyé signifie que la plus petite valeur est la valeur située en 3$^{e}$ position, suivie de celle en 1$^{ère}$ position et de celle en 2$^{e}$ position. Tout cela ne paraît pas passionnant à première vue, mais si on mélange ce résultat avec un peu d'indexation directe, ça devient intéressant. . .

```
R> head(order(d$age))
```

```
[1] 162 215 346 377 511 646
```

Ce que cette fonction renvoie, c'est l'ordre dans lequel on doit placer les éléments de age, et donc par extension les lignes de `d`, pour que la variable soit triée par ordre croissant. Par conséquent, si on fait :

```
R> d.tri <- d[order(d$age), ]
```

Alors on a trié les lignes de d par ordre d'âge croissant ! Et si on fait un petit :

```
R> head(d.tri, 3)
```

```
     id age  sexe nivetud    poids
162 162  18 Homme    <NA> 4982.964
215 215  18 Homme    <NA> 4631.188
346 346  18 Femme    <NA> 1725.410
              occup qualif freres.soeurs clso
162 Etudiant, eleve   <NA>             2  Non
215 Etudiant, eleve   <NA>             2  Oui
346 Etudiant, eleve   <NA>             9  Non
                          relig trav.imp
162  Appartenance sans pratique     <NA>
215 Ni croyance ni appartenance     <NA>
346         Pratiquant regulier     <NA>
    trav.satisf hard.rock lecture.bd peche.chasse
162        <NA>       Non        Non          Non
215        <NA>       Non        Non          Non
346        <NA>       Non        Non          Non
    cuisine bricol cinema sport heures.tv fs.fac
162     Non    Non    Non   Oui         3      2
215     Oui    Non    Oui   Oui         2      2
346     Non    Non    Oui   Non         2      9
    fs.char qualif.char      age5cl  age20
162       2        <NA> (17.9,33.8] <20ans
215       2        <NA> (17.9,33.8] <20ans
346       9        <NA> (17.9,33.8] <20ans
     age6cl qualif.reg trav.satisf.reg
162 [18,30)       <NA>        Manquant
215 [18,30)       <NA>        Manquant
346 [18,30)       <NA>        Manquant
    ecart.heures.tv           act.manuelles
162       0.7534336 Ni cuisine ni bricolage
215      -0.2465664       Cuisine seulement
346      -0.2465664 Ni cuisine ni bricolage
                 age.sexe score.ext
162 Homme moins de 40 ans        20
215 Homme moins de 40 ans        30
346 Femme moins de 40 ans        10
```

On a les caractéristiques des trois enquêtés les plus jeunes.

On peut évidemment trier par ordre décroissant en utilisant l'option `decreasing=TRUE`. On peut donc afficher les caractéristiques des trois individus les plus âgés avec :

```
R> head(d[order(d$age, decreasing = TRUE), ], 3)
```

```
       id age  sexe
1916 1916  97 Femme
270   270  96 Femme
1542 1542  93 Femme
                              nivetud    poids
1916 Derniere annee d'etudes primaires 2162.835
270  Derniere annee d'etudes primaires 9993.020
1542 Derniere annee d'etudes primaires 7107.841
                  occup qualif freres.soeurs
1916       Autre inactif  Autre             5
270             Retraite   <NA>             1
1542 Retire des affaires   <NA>             7
     clso                       relig trav.imp
1916  Non      Pratiquant occasionnel     <NA>
270   Oui Ni croyance ni appartenance     <NA>
1542  Non      Pratiquant occasionnel     <NA>
     trav.satisf hard.rock lecture.bd
1916        <NA>       Non        Non
270         <NA>       Non        Non
1542        <NA>       Non        Non
     peche.chasse cuisine bricol cinema sport
1916          Non     Non    Non    Non   Non
270           Non     Non    Non    Non   Non
1542          Non     Non    Non    Oui   Non
     heures.tv fs.fac fs.char qualif.char
1916         3      5       5       Autre
270          6      1       1        <NA>
1542         3      7       7        <NA>
          age5cl  age20  age6cl qualif.reg
1916 (81.2,97.1]   <NA> [66,97]      Autre
270  (81.2,97.1]   <NA> [66,97]       <NA>
1542 (81.2,97.1] >80ans [66,97]       <NA>
     trav.satisf.reg ecart.heures.tv
1916        Manquant       0.7534336
270         Manquant       3.7534336
1542        Manquant       0.7534336
               act.manuelles             age.sexe
1916 Ni cuisine ni bricolage Femme plus de 40 ans
270  Ni cuisine ni bricolage Femme plus de 40 ans
1542 Ni cuisine ni bricolage Femme plus de 40 ans
```

```
     score.ext
1916         0
270          0
1542        10
```

# Fusion de tables

Lorsqu'on traite de grosses enquêtes, notamment les enquêtes de l'INSEE, on a souvent à gérer des données réparties dans plusieurs tables, soit du fait de la construction du questionnaire, soit du fait de contraintes techniques (fichiers **dbf** ou **Excel** limités à 256 colonnes, par exemple).

Une opération relativement courante consiste à fusionner plusieurs tables pour regrouper tout ou partie des données dans un unique tableau.

Nous allons simuler artificiellement une telle situation en créant deux tables à partir de l'extrait de l'enquête *Histoire de vie* :

```
R> data(hdv2003)
   d <- hdv2003
   dim(d)
```

```
[1] 2000   20
```

```
R> d1 <- subset(d, select = c("id", "age", "sexe"))
   dim(d1)
```

```
[1] 2000    3
```

```
R> d2 <- subset(d, select = c("id", "clso"))
   dim(d2)
```

```
[1] 2000    2
```

On a donc deux tableaux de données, `d1` et `d2`, comportant chacun 2000 lignes et respectivement 3 et 2 colonnes. Comment les rassembler pour n'en former qu'un ?

Intuitivement, cela paraît simple. Il suffit de « coller » `d2` à la droite de `d1`, comme dans l'exemple suivant.

| id | v1 | v2 | id | v3 | | id | v1 | v2 | v3 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | H | 12 | 1 | rouge | | 1 | H | 12 | rouge |
| 2 | H | 17 | 2 | bleu | | 2 | H | 17 | bleu |
| 3 | F | 41 | 3 | bleu | = | 3 | F | 41 | bleu |
| 4 | F | 9 | 4 | rouge | | 4 | F | 9 | rouge |
| ..... | | | ..... | | ... | ..... | | | |

Cela semble fonctionner. La fonction qui permet d'effectuer cette opération sous **R** s'appelle `cbind`, elle « colle » des tableaux côte à côte en regroupant leurs colonnes[3].

```
R> head(cbind(d1, d2))
```

```
  id age  sexe id clso
1  1  28 Femme  1  Oui
2  2  23 Femme  2  Oui
3  3  59 Homme  3  Non
4  4  34 Homme  4  Non
5  5  71 Femme  5  Oui
6  6  35 Femme  6  Non
```

À part le fait qu'on a une colonne *id* en double, le résultat semble satisfaisant. À première vue seulement. Imaginons maintenant que nous avons travaillé sur `d1` et `d2`, et que nous avons ordonné les lignes de d1 selon l'âge des enquêtés :

```
R> d1 <- d1[order(d1$age), ]
```

Répétons l'opération de collage :

```
R> head(cbind(d1, d2))
```

```
     id age  sexe id clso
162 162  18 Homme  1  Oui
215 215  18 Homme  2  Oui
346 346  18 Femme  3  Non
377 377  18 Homme  4  Non
511 511  18 Homme  5  Oui
646 646  18 Homme  6  Non
```

Que constate-t-on ? La présence de la variable *id* en double nous permet de voir que les identifiants ne coïncident plus ! En regroupant nos colonnes nous avons donc attribué à des individus les réponses d'autres individus.

La commande `cbind` ne peut en effet fonctionner que si les deux tableaux ont exactement le même nombre de lignes, et dans le même ordre, ce qui n'est pas le cas ici.

On va donc être obligé de pocéder à une *fusion* des deux tableaux, qui va permettre de rendre à chaque ligne ce qui lui appartient. Pour cela nous avons besoin d'un identifiant qui permet d'identifier chaque ligne de manière unique et qui doit être présent dans tous les tableaux. Dans notre cas, c'est plutôt rapide, il s'agit de la variable *id*.

Une fois l'identifiant identifié[4], on peut utiliser la commande `merge`. Celle-ci va fusionner les deux tableaux en supprimant les colonnes en double et en regroupant les lignes selon leurs identifiants :

```
R> d.complet <- merge(d1, d2, by = "id")
   head(d.complet)
```

```
  id age  sexe clso
1  1  28 Femme  Oui
2  2  23 Femme  Oui
3  3  59 Homme  Non
4  4  34 Homme  Non
5  5  71 Femme  Oui
6  6  35 Femme  Non
```

Ici l'utilisation de la fonction est plutôt simple car nous sommes dans le cas de figure idéal : les lignes correspondent parfaitement et l'identifiant est clairement identifié. Parfois les choses peuvent être un peu plus compliquées :

- parfois les identifiants n'ont pas le même nom dans les deux tableaux. On peut alors les spécifier par les options `by.x` et `by.y` ;
- parfois les deux tableaux comportent des colonnes (hors identifiants) ayant le même nom. `merge` conserve dans ce cas ces deux colonnes mais les renomme en les suffixant par *.x* pour celles provenant du premier tableau et *.y* pour celles du second ;
- parfois on n'a pas d'identifiant unique préétabli, mais on en construit un à partir de plusieurs variables. On peut alors donner un vecteur en paramètres de l'option `by`, par exemple `by=c("nom","prenom","date.naissance")`.

Une subtilité supplémentaire intervient lorsque les deux tableaux fusionnés n'ont pas exactement les mêmes lignes. Par défaut, `merge` ne conserve que les lignes présentes dans les deux tableaux :

| **idv1** | | | **idv2** | | | | **idv1** | **v2** |
|---|---|---|---|---|---|---|---|---|
| 1 | H | | 1 | 10 | | | 1 | H | 10 |
| 2 | H | + | 2 | 15 | = | | 2 | H | 15 |
| 3 | F | | 5 | 31 | | | | | |

On peut cependant modifier ce comportement avec les options `all.x` et `all.y`.

Ainsi, `all.x=TRUE` indique de conserver toutes les lignes du premier tableau. Dans ce cas `merge` donne une valeur `NA` pour ces lignes aux colonnes provenant du second tableau. Ce qui donnerait :

| **idv1** | | | **idv2** | | | | **idv1** | **v2** |
|---|---|---|---|---|---|---|---|---|
| 1 | H | | 1 | 10 | | | 1 | H | 10 |
| 2 | H | + | 2 | 15 | = | | 2 | H | 15 |
| 3 | F | | 5 | 31 | | | 3 | F | `NA` |

L'option `all.y=TRUE` fait la même chose en conservant toutes les lignes du second tableau.

| **idv1** | | | **idv2** | | | | **id** | **v1** | **v2** |
|---|---|---|---|---|---|---|---|---|---|
| 1 | H | | 1 | 10 | | | 1 | H | 10 |
| 2 | H | + | 2 | 15 | = | | 2 | H | 15 |
| 3 | F | | 5 | 31 | | | 5 | `NA` | 31 |

Enfin, on peut décider de conserver toutes les lignes des deux tableaux en utilisant à la fois `all.x=TRUE` et `all.y=TRUE`, ce qui donne :

| idv1 | | idv2 | | id | v1 | v2 |
|---|---|---|---|---|---|---|
| 1 | H | 1 | 10 | 1 | H | 10 |
| 2 | H | 2 | 15 | 2 | H | 15 |
| 3 | F | 5 | 31 | 3 | F | `NA` |
| | | | | 5 | `NA` | 31 |

Parfois, l'un des identifiants est présent à plusieurs reprises dans l'un des tableaux (par exemple lorsque l'une des tables est un ensemble de ménages et que l'autre décrit l'ensemble des individus de ces ménages). Dans ce cas les lignes de l'autre table sont dupliquées autant de fois que nécessaires :

| idv1 | | idv2 | | idv1 | v2 |
|---|---|---|---|---|---|
| 1 | H | 1 | 10 | 1 | H 10 |
| 2 | H | 1 | 18 | 1 | H 18 |
| 3 | F | 1 | 21 | 1 | H 21 |
| | | 2 | 15 | 2 | H 15 |
| | | 3 | 42 | 3 | F 42 |

---

1. Une différence entre les deux est que `$` admet une correspondance partielle du nom de variable, si celle-ci est unique. Ainsi, `d$cls` renverra bien la variable *clso*, tandis que `d$c` renverra `NULL`, du fait que plusieurs variables de *d* commencent par la lettre *c*.↩
2. Si vous utilisez l'extenstion **data.table**, la gestion des valeurs manquantes est quelque peu différente. Voir chapitre MAJ_LIEN.↩
3. L'équivalent de `cbind` pour les lignes s'appelle `rbind`.↩
4. Si vous me passez l'expression. . .↩

# Statistique bivariée

> **NOTE**
>
> La version originale de ce chapitre a été écrite par Julien Barnier dans le cadre du support de cours Introduction à R.

On entend par *statistique bivariée* l'étude des relations entre deux variables, celles-ci pouvant être quantitatives ou qualitatives.

La *statistique univariée* a quant à elle déjà été abordée dans un chapitre dédié.

Comme dans la partie précédente, on travaillera sur les jeux de données fournis avec l'extension **questionr** et tiré de l'enquête *Histoire de vie* et du recensement 1999 :

```
R> library(questionr)
   data(hdv2003)
   d <- hdv2003
   data(rp99)
```

## Deux variables quantitatives

La comparaison de deux variables quantitatives se fait en premier lieu graphiquement, en représentant l'ensemble des couples de valeurs. On peut ainsi représenter les valeurs du nombre d'heures passées devant la télévision selon l'âge.

```
R> plot(d$age, d$heures.tv)
```

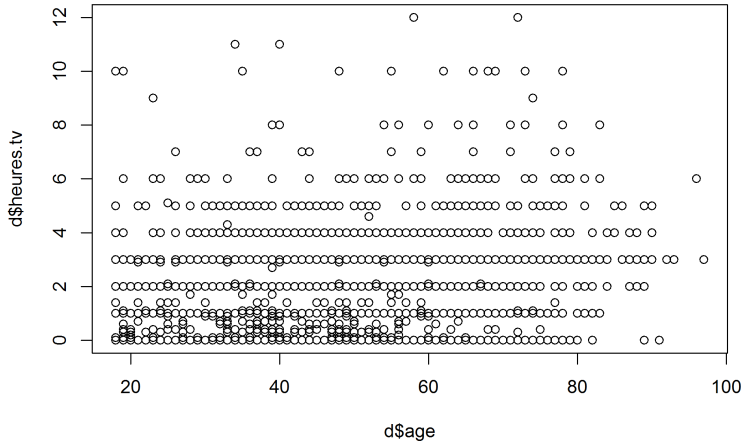



*Figure 1. Nombre d'heures de télévision selon l'âge*

Le fait que des points sont superposés ne facilite pas la lecture du graphique. On peut utiliser une représentation avec des points semi-transparents.

```
R> plot(d$age, d$heures.tv, pch = 19, col = rgb(1, 0,
     0, 0.1))
```

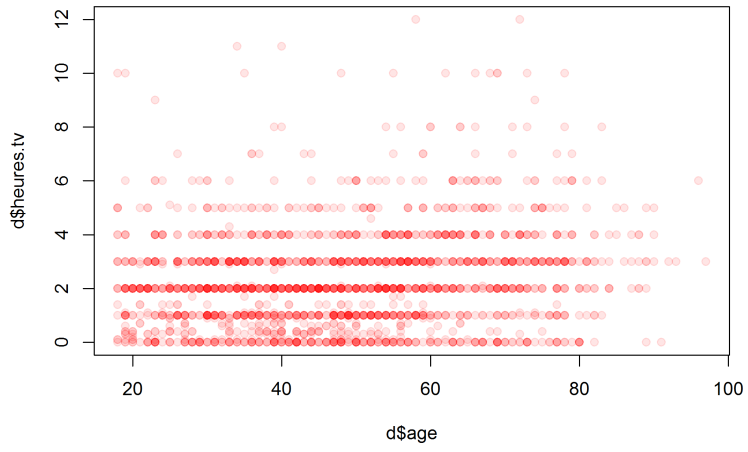



*Figure 2. Nombre d'heures de télévision selon l'âge avec semi-transparence*

Plus sophistiqué, on peut faire une estimation locale de densité et représenter le résultat sous forme de « carte ». Pour cela on commence par isoler les deux variables, supprimer les observations ayant au moins une valeur manquante à l'aide de la fonction `complete.cases`, estimer la densité locale à l'aide de la fonction `kde2d` de l'extension **MASS**[1] et représenter le tout à l'aide d'une des fonctions `image`, `contour` ou `filled.contour`...

```
R> library(MASS)
   tmp <- d[, c("age", "heures.tv")]
   tmp <- tmp[complete.cases(tmp), ]
   filled.contour(kde2d(tmp$age, tmp$heures.tv), color
   = terrain.colors)
```

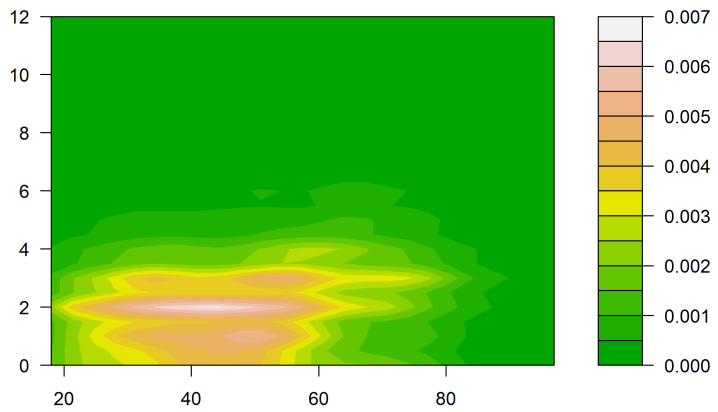



*Figure 3. Représentation de l'estimation de densité locale*

Dans tous les cas, il n'y a pas de structure très nette qui semble se dégager. On peut tester ceci mathématiquement en calculant le coefficient de corrélation entre les deux variables à l'aide de la fonction `cor` :

```
R> cor(d$age, d$heures.tv, use = "complete.obs")
```

```
[1] 0.1776249
```

L'option `use` permet d'éliminer les observations pour lesquelles l'une des deux valeurs est manquante. Le coefficient de corrélation est très faible.

On va donc s'intéresser plutôt à deux variables présentes dans le jeu de données `rp99`, la part de diplômés du supérieur et la proportion de cadres dans les communes du Rhône en 1999.

À nouveau, commençons par représenter les deux variables.

```
R> plot(rp99$dipl.sup, rp99$cadres, ylab = "Part des
   cadres",
     xlab = "Part des diplomês du supérieur")
```

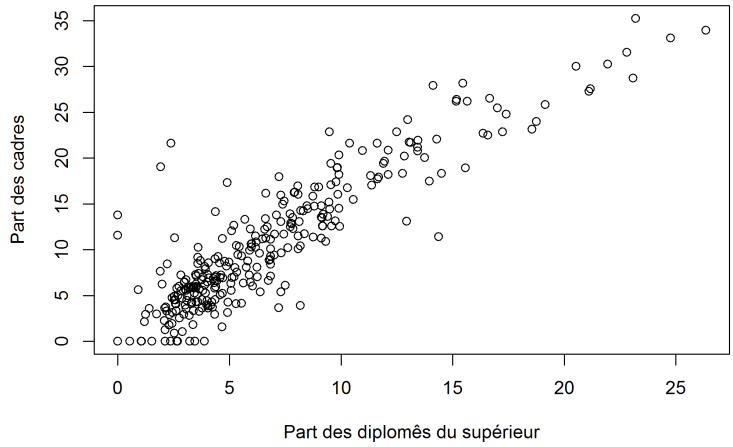



*Figure 4. Proportion de cadres et proportion de diplômés du supérieur*

Ça ressemble déjà beaucoup plus à une relation de type linéaire.

Calculons le coefficient de corrélation :

```
R> cor(rp99$dipl.sup, rp99$cadres)
```

```
[1] 0.8975282
```

C'est beaucoup plus proche de 1. On peut alors effectuer une régression linéaire complète en utilisant la fonction `lm` :

```
R> reg <- lm(cadres ~ dipl.sup, data = rp99)
   summary(reg)
```

```
Call:
lm(formula = cadres ~ dipl.sup, data = rp99)

Residuals:
    Min      1Q  Median      3Q     Max 
-9.6905 -1.9010 -0.1823  1.4913 17.0866 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  1.24088    0.32988   3.762 0.000203 ***
dipl.sup     1.38352    0.03931  35.196  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.281 on 299 degrees of freedom
Multiple R-squared:  0.8056,	Adjusted R-squared:  
0.8049 
F-statistic:  1239 on 1 and 299 DF,  p-value: < 2.2e-16
```

Le résultat montre que les coefficients sont significativement différents de 0. La part de cadres augmente donc avec celle de diplômés du supérieur (ô surprise). On peut très facilement représenter la droite de régression à l'aide de la fonction `abline`.

```
R> plot(rp99$dipl.sup, rp99$cadres, ylab = "Part des
   cadres",
     xlab = "Part des diplômés du supérieur")
   abline(reg, col = "red")
```

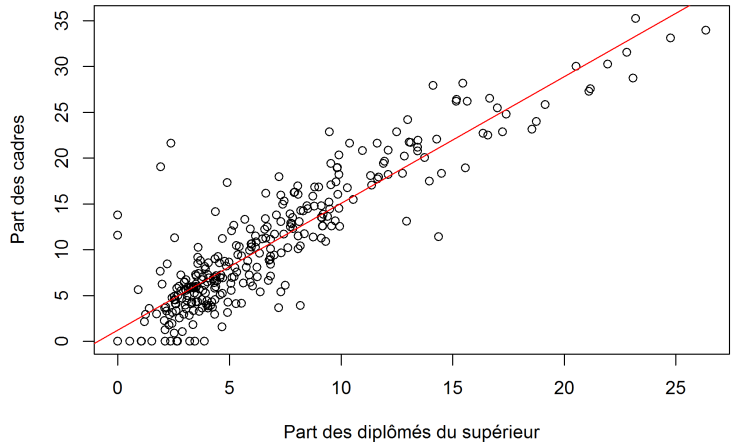



*Figure 5. Régression de la proportion de cadres par celle de diplômés du supérieur*

**INFO**

On remarquera que le premier argument passé à la fonction `lm` a une syntaxe un peu particulière. Il s'agit d'une *formule*, utilisée de manière générale dans les modèles statistiques. On indique la variable d'intérêt à gauche et la variable explicative à droite, les deux étant séparées par un tilde `~` (obtenu sous **Windows** en appuyant simultanément sur les touches `Alt Gr` et `2`). On remarquera que les noms des colonnes de notre tableau de données ont été écrites sans guillemets.

Dans le cas présent, nous avons calculé une régression linéaire simple entre deux variables, d'où l'écriture `cadres ~ dipl.sup`. Si nous avions voulu expliquer une variable *z* par deux variables *x* et *y*, nous aurions écrit `z ~ x + y`. Il est possible de spécifier des modèles encore plus complexes.

Pour un aperçu de la syntaxe des formules sous **R**, voir http://ww2.coastal.edu/kingw/statistics/R-tutorials/formulae.html.

# Une variable quantitative et une variable qualitative

## Représentations graphiques

Quand on parle de comparaison entre une variable quantitative et une variable qualitative, on veut en général savoir si la distribution des valeurs de la variable quantitative est la même selon les modalités de la variable qualitative. En clair : est ce que l'âge de ceux qui écoutent du hard rock est différent de l'âge de ceux qui n'en écoutent pas ?

Là encore, l'idéal est de commencer par une représentation graphique. Les boîtes à moustaches sont parfaitement adaptées pour cela.

Si on a construit des sous-populations d'individus écoutant ou non du hard rock, on peut utiliser la fonction `boxplot`.

```
R> d.hard <- subset(d, hard.rock == "Oui")
   d.non.hard <- subset(d, hard.rock == "Non")
   boxplot(d.hard$age, d.non.hard$age)
```

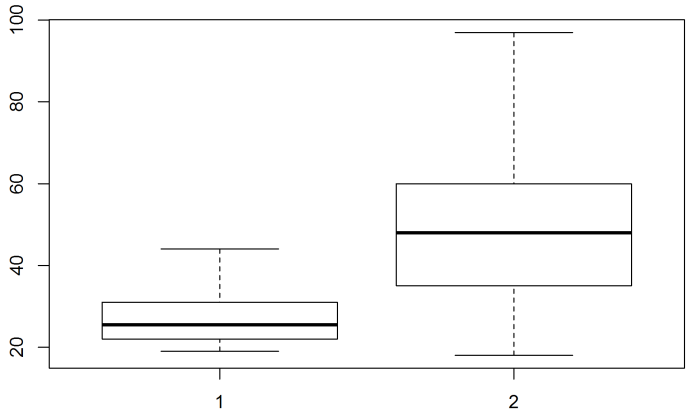



*Figure 6. Boxplot de la répartition des âges (sous-populations)*

Mais construire les sous-populations n'est pas nécessaire. On peut utiliser directement la version de `boxplot` prenant une formule en argument.

```
R> boxplot(age ~ hard.rock, data = d)
```

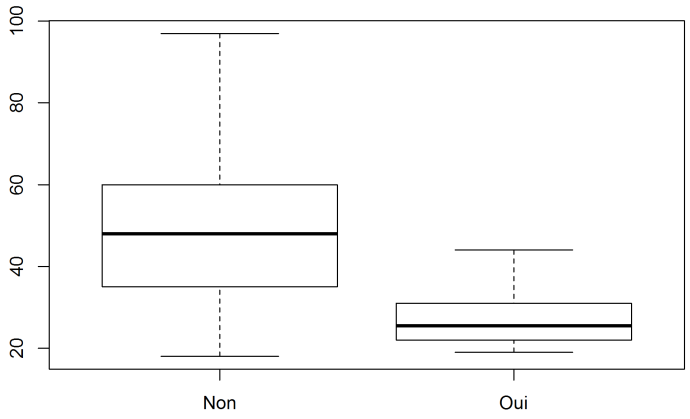



*Figure 7. Boxplot de la répartition des âges (formule)*

À première vue, ô surprise, la population écoutant du hard rock a l'air sensiblement plus jeune. Peut-on le tester mathématiquement ?

## Comparaison de moyennes

On peut calculer la moyenne d'âge des deux groupes en utilisant la fonction `tapply`[2] :

```
R> tapply(d$age, d$hard.rock, mean)
```

```
     Non      Oui
48.30211 27.57143
```

L'écart est important. Est-il statistiquement significatif ? Pour cela on peut faire un test *t* de comparaison de moyennes à l'aide de la fonction `t.test` :

```
R> t.test(d$age ~ d$hard.rock)
```

```
    Welch Two Sample t-test

data:  d$age by d$hard.rock
t = 9.6404, df = 13.848, p-value = 1.611e-07
alternative hypothesis: true difference in means is not
equal to 0
95 percent confidence interval:
 16.11379 25.34758
sample estimates:
mean in group Non mean in group Oui
         48.30211          27.57143
```

Le test est extrêmement significatif. L'intervalle de confiance à 95 % de la différence entre les deux moyennes va de 14,5 ans à 21,8 ans.

**INFO**

La valeur affichée pour *p* est de `1.611e-07`. Cette valeur peut paraître étrange pour les non avertis. Cela signifie tout simplement 1,611 multiplié par 10 à la puissance -7, autrement dit 0,0000001611. Cette manière de représenter un nombre est couramment appelée *notation scientifique*.

Pour plus de détails, voir http://fr.wikipedia.org/wiki/Notation_scientifique.

Nous sommes cependant allés un peu vite en besogne, car nous avons négligé une hypothèse fondamentale du test *t* : les ensembles de valeur comparés doivent suivre approximativement une loi normale et être de même variance[3]. Comment le vérifier ?

D'abord avec un petit graphique :

```
R> par(mfrow = c(1, 2))
   hist(d$age[d$hard.rock == "Oui"], main = "Hard
   rock",
     col = "red")
   hist(d$age[d$hard.rock == "Non"], main = "Sans hard
   rock",
     col = "red")
```

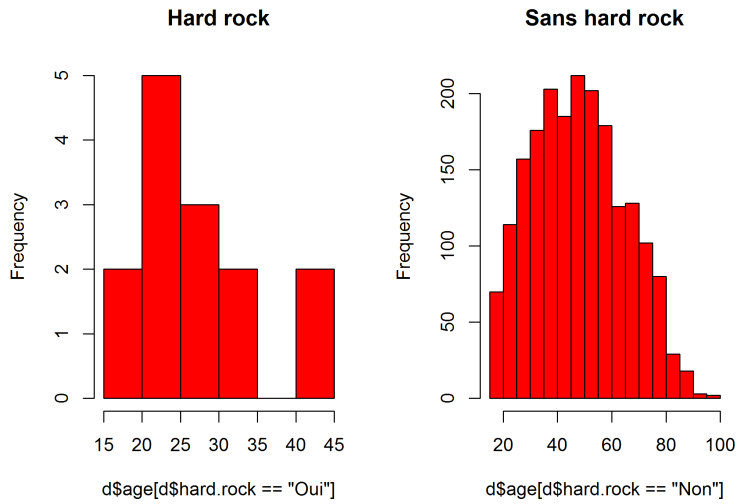



*Figure 8. Distribution des âges pour appréciation de la normalité*

**NOTE**

La fonction `par` permet de modifier de nombreux paramètres graphiques. `par(mfrow = c(1, 2))` sert à indiquer que l'on souhaite afficher deux graphiques sur une même fenêtre, plus précisément que la fenêtre doit comporter une ligne et deux colonnes.

Ça a l'air à peu près bon pour les « Sans hard rock », mais un peu plus limite pour les fans de *Metallica*, dont les effectifs sont d'ailleurs assez faibles. Si on veut en avoir le coeur net on peut utiliser le test de normalité de *Shapiro-Wilk* avec la fonction `shapiro.test` :

```
R> shapiro.test(d$age[d$hard.rock == "Oui"])
```

```
    Shapiro-Wilk normality test

data:  d$age[d$hard.rock == "Oui"]
W = 0.86931, p-value = 0.04104
```

```
R> shapiro.test(d$age[d$hard.rock == "Non"])
```

```
    Shapiro-Wilk normality test

data:  d$age[d$hard.rock == "Non"]
W = 0.98141, p-value = 2.079e-15
```

Visiblement, le test estime que les distributions ne sont pas suffisamment proches de la normalité dans les deux cas.

Et concernant l'égalité des variances ?

```
R> tapply(d$age, d$hard.rock, var)
```

```
       Non       Oui
285.62858  62.72527
```

L'écart n'a pas l'air négligeable. On peut le vérifier avec le test fourni par la fonction `var.test` :

```
R> var.test(d$age ~ d$hard.rock)
```

```
    F test to compare two variances

data:  d$age by d$hard.rock
F = 4.5536, num df = 1985, denom df = 13,
p-value = 0.003217
alternative hypothesis: true ratio of variances is not
equal to 1
95 percent confidence interval:
 1.751826 8.694405
sample estimates:
ratio of variances
          4.553644
```

La différence est très significative. En toute rigueur le test *t* n'aurait donc pas pu être utilisé.

*Damned* ! Ces maudits tests statistiques vont-ils nous empêcher de faire connaître au monde entier notre fabuleuse découverte sur l'âge des fans de *Sepultura* ? Non ! Car voici qu'approche à l'horizon un nouveau test, connu sous le nom de *Wilcoxon/Mann-Whitney*. Celui-ci a l'avantage d'être non-paramétrique, c'est à dire de ne faire aucune hypothèse sur la distribution des échantillons comparés. Par contre il ne compare pas des différences de moyennes mais des différences de médianes :

```
R> wilcox.test(d$age ~ d$hard.rock)
```

```
    Wilcoxon rank sum test with continuity
    correction

data:  d$age by d$hard.rock
W = 23980, p-value = 2.856e-06
alternative hypothesis: true location shift is not equal
to 0
```

Ouf ! La différence est hautement significative[4]. Nous allons donc pouvoir entamer la rédaction de notre article pour la *Revue française de sociologie*.

# Deux variables qualitatives

La comparaison de deux variables qualitatives s'appelle en général un *tableau croisé*. C'est sans doute l'une des analyses les plus fréquentes lors du traitement d'enquêtes en sciences sociales.

## Tableau croisé

La manière la plus simple d'obtenir un tableau croisé est d'utiliser la fonction `table` en lui donnant en paramètres les deux variables à croiser. En l'occurrence nous allons croiser un recodage du niveau de qualification regroupé avec le fait de pratiquer un sport.

On commence par calculer la variable recodée et par afficher le tri à plat des deux variables :

```
R> d$qualreg <- as.character(d$qualif)
   d$qualreg[d$qualif %in% c("Ouvrier specialise", "Ouvrier
   qualifie")] <- "Ouvrier"
   d$qualreg[d$qualif %in% c("Profession intermediaire",
     "Technicien")] <- "Intermediaire"
   table(d$qualreg)
```

```
        Autre         Cadre       Employe
           58           260           594
Intermediaire       Ouvrier
          246           495
```

Le tableau croisé des deux variables s'obtient de la manière suivante :

```
R> table(d$sport, d$qualreg)
```

```
      Autre Cadre Employe Intermediaire Ouvrier
  Non    38   117     401           127     381
  Oui    20   143     193           119     114
```

**INFO**

Il est tout à fait possible de croiser trois variables ou plus. Par exemple :

```
R> table(d$sport, d$cuisine, d$sexe)
```

```
, ,  = Homme


      Non Oui
  Non 401 129
  Oui 228 141

, ,  = Femme


      Non Oui
  Non 358 389
  Oui 132 222
```

On n'a cependant que les effectifs, ce qui rend difficile les comparaisons. L'extension **questionr** fournit des fonctions permettant de calculer facilement les pourcentages lignes, colonnes et totaux d'un tableau croisé.

Les pourcentages lignes s'obtiennent avec la fonction `lprop`[5]. Celle-ci s'applique au tableau croisé généré par `table` :

```
R> tab <- table(d$sport, d$qualreg)
   lprop(tab)
```

```
          Autre Cadre Employe Intermediaire
Non         3.6  11.0  37.7    11.9
Oui         3.4  24.3  32.8    20.2
Ensemble    3.5  15.7  35.9    14.9

          Ouvrier Total
Non        35.8   100.0
Oui        19.4   100.0
Ensemble   29.9   100.0
```

Les pourcentages ligne ne nous intéressent guère ici. On ne cherche pas à voir quelle est la proportion de cadres parmi ceux qui pratiquent un sport, mais plutôt quelle est la proportion de sportifs chez les cadres. Il nous faut donc des pourcentages colonnes, que l'on obtient avec la fonction `cprop` :

```
R> cprop(tab)
```

```
       Autre Cadre Employe Intermediaire Ouvrier
Non     65.5  45.0  67.5    51.6          77.0
Oui     34.5  55.0  32.5    48.4          23.0
Total  100.0 100.0 100.0   100.0         100.0

       Ensemble
Non     64.4
Oui     35.6
Total  100.0
```

Dans l'ensemble, le pourcentage de personnes ayant pratiqué un sport est de 35,6 %. Mais cette proportion varie fortement d'une catégorie professionnelle à l'autre : 55,0 % chez les cadres contre 23,0 % chez les ouvriers.

Enfin, les pourcentage totaux s'obtiennent avec la fonction `prop` :

```
R> prop(tab)
```

```
       Autre Cadre Employe Intermediaire Ouvrier
Non      2.3   7.1  24.3     7.7          23.0
Oui      1.2   8.7  11.7     7.2           6.9
Total    3.5  15.7  35.9    14.9          29.9

       Total
Non     64.4
Oui     35.6
Total  100.0
```

À noter qu'on peut personnaliser l'affichage de ces tableaux de pourcentages à l'aide de différentes options, dont `digits` qui règle le nombre de décimales à afficher et `percent` qui indique si on souhaite ou non rajouter un symbole `%` dans chaque case du tableau. Cette personnalisation peut se faire directement au moment de la génération du tableau et dans ce cas elle sera utilisée par défaut :

```
R> ctab <- cprop(tab, digits = 2, percent = TRUE)
   ctab
```

```
       Autre   Cadre   Employe Intermediaire
Non     65.52%  45.00%  67.51%  51.63%
Oui     34.48%  55.00%  32.49%  48.37%
Total 100.00% 100.00% 100.00% 100.00%

       Ouvrier Ensemble
Non     76.97%  64.37%
Oui     23.03%  35.63%
Total 100.00% 100.00%
```

Ou bien ponctuellement en passant les mêmes arguments à la fonction `print` :

```
R> ctab <- cprop(tab)
   print(ctab, percent = TRUE)
```

```
       Autre  Cadre  Employe Intermediaire
Non     65.5%  45.0%  67.5%   51.6%
Oui     34.5%  55.0%  32.5%   48.4%
Total 100.0% 100.0% 100.0%  100.0%

       Ouvrier Ensemble
Non     77.0%   64.4%
Oui     23.0%   35.6%
Total 100.0%  100.0%
```

## $\chi^2$ et dérivés

Pour tester l'existence d'un lien entre les modalités des deux variables, on va utiliser le très classique test du $\chi^2$[6]. Celui-ci s'obtient grâce à la fonction `chisq.test`, appliquée au tableau croisé obtenu avec `table`[7] :

```
R> chisq.test(tab)
```

```
    Pearson's Chi-squared test

data:  tab
X-squared = 96.798, df = 4, p-value <
2.2e-16
```

Le test est hautement significatif, on ne peut pas considérer qu'il y a indépendance entre les lignes et les colonnes du tableau.

On peut affiner l'interprétation du test en déterminant dans quelle case l'écart à l'indépendance est le plus significatif en utilisant les *résidus* du test. Ceux-ci sont notamment affichables avec la fonction `chisq.residuals` de **questionr** :

```
R> chisq.residuals(tab)
```

```
      Autre Cadre Employe Intermediaire Ouvrier
Non    0.11 -3.89    0.95         -2.49    3.49
Oui   -0.15  5.23   -1.28          3.35   -4.70
```

Les cases pour lesquelles l'écart à l'indépendance est significatif ont un résidu dont la valeur est supérieure à 2 ou inférieure à -2. Ici on constate que la pratique d'un sport est sur-représentée parmi les cadres et, à un niveau un peu moindre, parmi les professions intermédiaires, tandis qu'elle est sousreprésentée chez les ouvriers.

Enfin, on peut calculer le coefficient de contingence de Cramer du tableau, qui peut nous permettre de le comparer par la suite à d'autres tableaux croisés. On peut pour cela utiliser la fonction `cramer.v` de **questionr** :

```
R> cramer.v(tab)
```

```
[1] 0.24199
```

**INFO**

Pour un tableau à 2×2 entrées, il est possible de calculer le test exact de Fisher avec la fonction `fisher.test`. On peut soit lui passer le résultat de `table`, soit directement les deux variables à croiser.

```
R> lprop(table(d$sexe, d$cuisine))
```

```
           Non   Oui   Total
  Homme     70.0  30.0 100.0
  Femme     44.5  55.5 100.0
  Ensemble  56.0  44.0 100.0
```

```
R> fisher.test(table(d$sexe, d$cuisine))
```

```
    Fisher's Exact Test for Count Data

data:  table(d$sexe, d$cuisine)
p-value < 2.2e-16
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 2.402598 3.513723
sample estimates:
odds ratio
  2.903253
```

## Représentation graphique

Enfin, on peut obtenir une représentation graphique synthétisant l'ensemble des résultats obtenus sous la forme d'un graphique en mosaïque, grâce à la fonction `mosaicplot`.

```
R> mosaicplot(qualreg ~ sport, data = d, shade = TRUE,
    main = "Graphe en mosaïque")
```

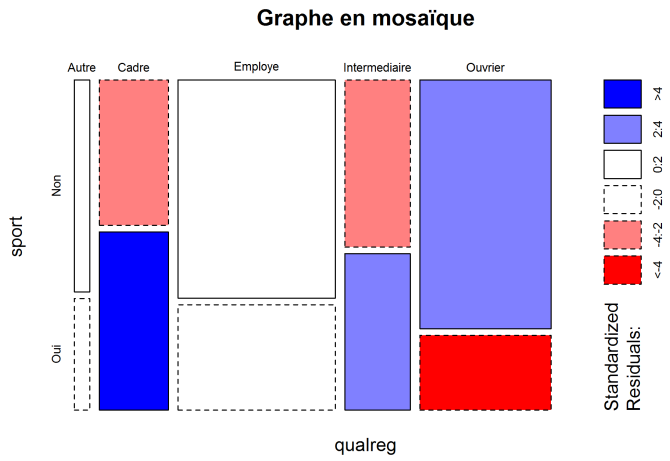



*Figure 9. Exemple de graphe en mosaïque*

Comment interpréter ce graphique haut en couleurs[8] ? Chaque rectangle représente une case de tableau. Sa largeur correspond aux pourcentages en colonnes (il y'a beaucoup d'employés et d'ouvriers et très peu d'« Autre »). Sa hauteur correspond aux pourcentages en lignes : la proportion de sportifs chez les cadres est plus élevée que chez les employés. Enfin, la couleur de la case correspond au résidu du test du $\chi^2$ correspondant : les cases en rouge sont sous-représentées, les cases en bleu sur-représentées, et les cases blanches sont statistiquement proches de l'hypothèse d'indépendance.

Lorsque l'on s'intéresse principalement aux variations d'une variable selon une autre, par exemple ici à la pratique du sport selon le niveau de qualification, il peut être intéressant de présenter les pourcentages en colonne sous la forme de barres cumulées.

```
R> barplot(cprop(tab, total = FALSE), main = "Pratique
   du sport selon le niveau de qualification")
```

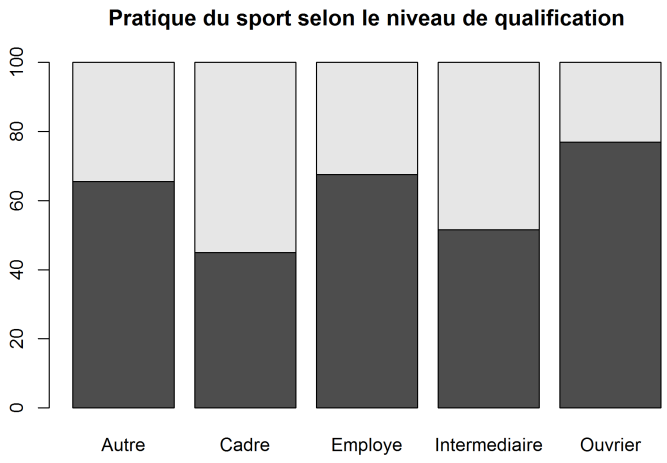



*Figure 10. Exemple de barres cumulées*

## Comparaison de proportions

La fonction `prop.test`, que nous avons déjà rencontrer pour calculer l'interface de confiance d'une proportion (MAJ_LIEN) permets également de comparer deux proportions.

Supposons que l'on souhaite comparer la proportion de personnes faisant du sport entre ceux qui lisent des bandes dessinées et les autres :

```
R> lprop(table(d$lecture.bd, d$sport))
```

```
          Non   Oui   Total
  Non      64.2  35.8 100.0
  Oui      48.9  51.1 100.0
  Ensemble 63.8  36.1 100.0
```

Il suffit de transmettre notre tableau croisé (à 2×2 dimensions) à `prop.test` :

```
R> prop.test(table(d$lecture.bd, d$sport))
```

```
    2-sample test for equality of proportions
    with continuity correction

data:  table(d$lecture.bd, d$sport)
X-squared = 4, df = 1, p-value = 0.0455
alternative hypothesis: two.sided
95 percent confidence interval:
 -0.002652453  0.308107236
sample estimates:
   prop 1    prop 2
0.6420891 0.4893617
```

1. MASS est installée par défaut avec la version de base de R.↩
2. La fonction `tapply` est présentée plus en détails dans le chapitre Manipulation de données.↩
3. Concernant cette seconde condition, `t.test` propose une option nommée `var.equal` qui permet d'utiliser une approximation dans le cas où les variances ne sont pas égales↩
4. Ce test peut également fournir un intervalle de confiance avec l'option `conf.int=TRUE`.↩
5. Il s'agit en fait d'un alias pour les francophones de la fonction `rprop`↩
6. On ne donnera pas plus d'indications sur le test du $\chi^2$ ici. Les personnes désirant une présentation plus détaillée pourront se reporter (attention, séance d'autopromotion !) à la page suivante : http://alea.fr.eu.org/pages/khi2.↩
7. On peut aussi appliquer directement le test en spécifiant les deux variables à croiser via `chisq.test(d$qualreg, d$sport)`.↩
8. Sauf s'il est imprimé en noir et blanc. . .↩

# Régression logistique

**NOTE**

La version originale de ce chapitre a été écrite par Joseph Larmarange dans le cadre du support de cours Introduction à l'analyse d'enquêtes avec R.

La régression logistique est fréquemment utilisée en sciences sociales car elle permet d'effectuer un raisonnement dit *toutes choses étant égales par ailleurs*. Plus précisément, la régression logistique a pour but d'isoler les effets de chaque variable, c'est-à-dire d'identifier les effets résiduels d'une *variable explicative* sur une *variable d'intérêt*, une fois pris en compte les autres variables explicatives introduites dans le modèle. La régression logistique est ainsi prisée en épidémiologie pour identifier les facteurs associés à telle ou telle pathologie.

La régression logistique ordinaire ou régression logistique binaire vise à expliquer une variable d'intérêt binaire (c'est-à-dire de type « Oui/Non »). Les variables explicatives qui seront introduites dans le modèle peuvent être quantitatives ou qualitatives.

## Préparation des données

Dans ce chapite, nous allons encore une fois utiliser les données de l'enquête *Histoire de vie*, fournies avec l'extension **questionr**.

```
R> library(questionr)
   data(hdv2003)
   d <- hdv2003
```

À titre d'exemple, nous allons étudier l'effet de l'âge, du sexe, du niveau d'étude, de la pratique religieuse et du nombre moyen d'heures passées à regarder la télévision par jour.

En premier lieu, il importe de vérifier que notre variable d'intérêt (ici *sport*) est correctement codée. Une possibilité consiste à créer une variable booléenne (vrai / faux) selon que l'individu a pratiqué du sport ou non :

```
R> d$sport2 <- FALSE
   d$sport2[d$sport == "Oui"] <- TRUE
```

Dans le cas présent, cette variable n'a pas de valeur manquante. Mais le cas échéant il faut bien renseigner `NA` pour les valeurs manquantes, les individus en question étant alors exclu de l'analyse.

Il n'est pas forcément nécessaire de transformer notre variable d'intérêt en variable booléenne. En effet, **R** accepte sans problème une variable de type facteur. Cependant, l'ordre des valeurs d'un facteur a de l'importance. En effet, **R** considère toujours la première modalité comme étant la modalité de référence. Dans le cas de la variable d'intérêt, la modalité de référence correspond au fait de ne pas remplir le critère étudié, dans notre exemple au fait de ne pas avoir eu d'activité sportive au cours des douze derniers mois.

Pour connaître l'ordre des modalités d'une variable de type facteur, on peut utiliser la fonction `levels` ou bien encore tout simplement la fonction `freq` :

```
R> levels(d$sport)
```

```
[1] "Non" "Oui"
```

```
R> freq(d$sport)
```

```
       n    % val%
Non 1277 63.8 63.8
Oui  723 36.1 36.1
NA     0  0.0   NA
```

Dans notre exemple, la modalité « Non » est déjà la première modalité. Il n'y a donc pas besoin de modifier notre variable. Si ce n'est pas le cas, il faudra

modifier la modalité de référence avec la fonction `relevel` comme nous allons le voir un peu plus loin.

> **IMPORTANT**
>
> Il est possible d'indiquer un facteur à plus de deux modalités. Dans une telle situation, **R** considérera que tous les modalités, sauf la modalité de référence, est une réalisation de la variable d'intérêt. Cela serait correct, par exemple, si notre variable *sport* était codée ainsi : « Non », « Oui, toutes les semaines », « Oui, au moins une fois par mois », « Oui, moins d'une fois par mois ». Cependant, afin d'éviter tout risque d'erreur ou de mauvaise interprétation, il est vivement conseillé de recoder au préalable sa variable d'intérêt en un facteur à deux modalités.

La notion de modalité de référence s'applique également aux variables explicatives qualitatives. En effet, dans un modèle, tous les coefficients sont calculés par rapport à la modalité de référence. Il importe de choisir une modalité de référence qui fasse sens afin de faciliter l'interprétation. Par ailleurs, ce choix peut également dépendre de la manière dont on souhaite présenter les résultats. De manière générale on évitera de choisir comme référence une modalité peu représentée dans l'échantillon ou bien une modalité correspondant à une situation atypique.

Prenons l'exemple de la variable *sexe*. Souhaite-t-on connaitre l'effet d'être une femme par rapport au fait d'être un homme ou bien l'effet d'être un homme par rapport au fait d'être une femme ? Si l'on opte pour le second, alors notre modalité de référence sera le sexe féminin. Comme est codée cette variable ?

```
R> freq(d$sexe)
```

```
         n   % val%
Homme  899 45   45
Femme 1101 55   55
NA       0  0   NA
```

La modalité « Femme » s'avère ne pas être la première modalité. Nous devons appliquer la fonction `relevel` :

```
R> d$sexe <- relevel(d$sexe, "Femme")
   freq(d$sexe)
```

```
         n   % val%
Femme 1101 55   55
Homme  899 45   45
NA       0  0   NA
```

Les variables *age* et *heures.tv* sont des variables quantitatives. Il importe de vérifier qu'elles sont bien enregistrées en tant que variables numériques. En effet, il arrive parfois que dans le fichier source les variables quantitatives soient renseignées sous forme de valeur textuelle et non sous forme numérique.

```
R> str(d$age)
```

```
 int [1:2000] 28 23 59 34 71 35 60 47 20 28 ...
```

```
R> str(d$heures.tv)
```

```
 num [1:2000] 0 1 0 2 3 2 2.9 1 2 2 ...
```

Nos deux variables sont bien renseignées sous forme numérique.

Cependant, l'effet de l'âge est rarement linéaire. Un exemple trivial est par exemple le fait d'occuper un emploi qui sera moins fréquent aux jeunes âges et aux âges élevés. Dès lors, on pourra transformer la variable *age* en groupe d'âges (voir MAJ_LIEN) :

```
R> d$grpage <- cut(d$age, c(16, 25, 45, 65, 93), right =
   FALSE,
     include.lowest = TRUE)
   freq(d$grpage)
```

```
          n    % val%
[16,25) 169  8.5  8.5
[25,45) 706 35.3 35.3
[45,65) 745 37.2 37.3
[65,93] 378 18.9 18.9
NA        2  0.1   NA
```

Jetons maintenant un oeil à la variable *nivetud* :

```
R> freq(d$nivetud)
```

```
n
N'a jamais fait
d'etudes                                          39
A arrete ses etudes, avant la derniere annee d'etudes
primaires  86
Derniere annee d'etudes
primaires                                 341
1er
cycle
204
2eme
cycle
183
Enseignement technique ou professionnel
court                    463
Enseignement technique ou professionnel
long                     131
Enseignement superieur y compris technique
superieur            441
NA
112

%
N'a jamais fait
d'etudes                                          2.0
A arrete ses etudes, avant la derniere annee d'etudes
primaires  4.3
Derniere annee d'etudes
primaires                                 17.1
1er
cycle
10.2
2eme
cycle
9.2
Enseignement technique ou professionnel
court                    23.2
Enseignement technique ou professionnel
long                      6.6
```

```
Enseignement superieur y compris technique
superieur            22.1
NA
5.6

val%
N'a jamais fait
d'etudes                                          2.1
A arrete ses etudes, avant la derniere annee d'etudes
primaires  4.6
Derniere annee d'etudes
primaires                                18.1
1er
cycle
10.8
2eme
cycle
9.7
Enseignement technique ou professionnel
court                   24.5
Enseignement technique ou professionnel
long                     6.9
Enseignement superieur y compris technique
superieur            23.4
NA
NA
```

En premier lieu, cette variable est détaillée en pas moins de huit modalités dont certaines sont peu représentées (seulement 39 individus soit 2 % n'ont jamais fait d'études par exemple). Afin d'améliorer notre modèle logistique, il peut être pertinent de regrouper certaines modalités (voir MAJ_LIEN) :

```
R> d$etud <- d$nivetud
   levels(d$etud) <- c("Primaire", "Primaire", "Primaire",
     "Secondaire", "Secondaire", "Technique/Professionnel",
     "Technique/Professionnel", "Supérieur")
   freq(d$etud)
```

```
                          n    % val%
Primaire                466 23.3 24.7
Secondaire              387 19.4 20.5
Technique/Professionnel 594 29.7 31.5
Supérieur               441 22.1 23.4
NA                      112  5.6   NA
```

Notre variable comporte également 112 individus avec une valeur manquante. Si nous conservons cette valeur manquante, ces 112 individus seront, par défaut, exclus de l'analyse. Ces valeurs manquantes n'étant pas négligeable (5,6 %), nous pouvons également faire le choix de considérer ces valeurs manquantes comme une modalité supplémentaire. Auquel cas, nous utiliserons la fonction `add.NA` :

```
R> levels(d$etud)
```

```
[1] "Primaire"
[2] "Secondaire"
[3] "Technique/Professionnel"
[4] "Supérieur"
```

```
R> d$etud <- addNA(d$etud)
   levels(d$etud)
```

```
[1] "Primaire"
[2] "Secondaire"
[3] "Technique/Professionnel"
[4] "Supérieur"
[5] NA
```

# Régression logistique binaire

La fonction `glm` (pour *generalized linear models*) permet de calculer une grande variété de modèles statistiques. La régression logistique ordinaire correspond au modèle *logit* de la famille des modèles binomiaux, ce que l'on indique à `glm` avec l'argument `family=binomial(logit)`.

Le modèle proprement dit sera renseigné sous la forme d'une formule (MAJ_LIEN). On indiquera d'abord la variable d'intérêt, suivie du signe `~` puis de

la liste des variables explicatives séparées par un signe `+`. Enfin, l'argument `data` permettra d'indiquer notre tableau de données.

```
R> reg <- glm(sport ~ sexe + grpage + etud + relig +
   heures.tv,
     data = d, family = binomial(logit))
   reg
```

```
Call:  glm(formula = sport ~ sexe + grpage + etud + relig
+ heures.tv,
    family = binomial(logit), data = d)

Coefficients:
                     (Intercept)
                       -0.797364
                       sexeHomme
                        0.439002
                  grpage[25,45)
                       -0.420306
                  grpage[45,65)
                       -1.085463
                  grpage[65,93]
                       -1.379274
                 etudSecondaire
                        0.948312
     etudTechnique/Professionnel
                        1.047156
                  etudSupérieur
                        1.889621
                          etudNA
                        2.148545
     religPratiquant occasionnel
                       -0.020597
 religAppartenance sans pratique
                       -0.006176
religNi croyance ni appartenance
                       -0.214067
                      religRejet
                       -0.382744
                religNSP ou NVPR
                       -0.083361
                       heures.tv
                       -0.120724

Degrees of Freedom: 1992 Total (i.e. Null);  1978 Residual
  (7 observations deleted due to missingness)
Null Deviance:      2607
Residual Deviance: 2206     AIC: 2236
```

**INFO**

Il est possible de spécifier des modèles plus complexes. Par exemple, `x:y` permet d'indiquer l'interaction entre les variables *x* et *y*. `x * y` sera équivalent à `x + y + x:y`. Pour aller plus loin, voir http://ww2.coastal.edu/kingw/statistics/R-tutorials/formulae.html.

Une présentation plus complète des résultats est obtenue avec la méthode `summary` :

```
R> summary(reg)
```

```
Call:
glm(formula = sport ~ sexe + grpage + etud + relig +
heures.tv,
    family = binomial(logit), data = d)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.8783  -0.8864  -0.4814   1.0033   2.4202

Coefficients:
                                  Estimate
(Intercept)                      -0.797364
sexeHomme                         0.439002
grpage[25,45)                    -0.420306
grpage[45,65)                    -1.085463
grpage[65,93]                    -1.379274
etudSecondaire                    0.948312
etudTechnique/Professionnel       1.047156
etudSupérieur                     1.889621
etudNA                            2.148545
religPratiquant occasionnel      -0.020597
religAppartenance sans pratique  -0.006176
religNi croyance ni appartenance -0.214067
religRejet                       -0.382744
religNSP ou NVPR                 -0.083361
heures.tv                        -0.120724
                                 Std. Error
(Intercept)                        0.323833
sexeHomme                          0.106063
grpage[25,45)                      0.228042
grpage[45,65)                      0.237704
grpage[65,93]                      0.273794
etudSecondaire                     0.197427
etudTechnique/Professionnel        0.189777
etudSupérieur                      0.195190
etudNA                             0.330198
religPratiquant occasionnel        0.189203
religAppartenance sans pratique    0.174709
religNi croyance ni appartenance   0.193096
religRejet                         0.285878
```

```
religNSP ou NVPR                    0.410968
heures.tv                           0.033589
                                z value Pr(>|z|)
(Intercept)                      -2.462 0.013806
sexeHomme                         4.139 3.49e-05
grpage[25,45)                    -1.843 0.065313
grpage[45,65)                    -4.566 4.96e-06
grpage[65,93]                    -5.038 4.71e-07
etudSecondaire                    4.803 1.56e-06
etudTechnique/Professionnel       5.518 3.43e-08
etudSupérieur                     9.681  < 2e-16
etudNA                            6.507 7.67e-11
religPratiquant occasionnel      -0.109 0.913311
religAppartenance sans pratique  -0.035 0.971801
religNi croyance ni appartenance -1.109 0.267600
religRejet                       -1.339 0.180624
religNSP ou NVPR                 -0.203 0.839260
heures.tv                        -3.594 0.000325

(Intercept)                     *
sexeHomme                       ***
grpage[25,45)                   .
grpage[45,65)                   ***
grpage[65,93]                   ***
etudSecondaire                  ***
etudTechnique/Professionnel     ***
etudSupérieur                   ***
etudNA                          ***
religPratiquant occasionnel
religAppartenance sans pratique
religNi croyance ni appartenance
religRejet
religNSP ou NVPR
heures.tv                       ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)
```

```
    Null deviance: 2607.4  on 1992  degrees of freedom
Residual deviance: 2205.9  on 1978  degrees of freedom
  (7 observations deleted due to missingness)
AIC: 2235.9

Number of Fisher Scoring iterations: 4
```

Dans le cadre d'un modèle logistique, généralement on ne présente pas les coefficients du modèle mais leur valeur exponentielle, cette dernière correspondant en effet à des *odds ratio*, également appelés *rapports des cotes*. L'odds ratio diffère du *risque relatif*. Cependent son interprétation est similaire. Un odds ratio de 1 signifie l'absence d'effet. Un odds ratio largement supérieur à 1 correspond à une augmentation du phénomène étudié et un odds ratio largement inféieur à 1 correspond à une diminution du phénomène étudié[1].

La fonction `coef` permet d'obtenir les coefficients d'un modèle, `confint` leurs intervalles de confiance et `exp` de calculer l'exponentiel. Les odds ratio et leurs intervalles de confiance s'obtiennent ainsi :

```
R> exp(coef(reg))
```

```
                      (Intercept)
                        0.4505151
                        sexeHomme
                        1.5511577
                   grpage[25,45)
                        0.6568456
                   grpage[45,65)
                        0.3377455
                   grpage[65,93]
                        0.2517614
                  etudSecondaire
                        2.5813476
     etudTechnique/Professionnel
                        2.8495358
                   etudSupérieur
                        6.6168632
                           etudNA
                        8.5723745
     religPratiquant occasionnel
                        0.9796133
 religAppartenance sans pratique
                        0.9938431
religNi croyance ni appartenance
                        0.8072940
                       religRejet
                        0.6819874
                 religNSP ou NVPR
                        0.9200190
                        heures.tv
                        0.8862785
```

```
R> exp(confint(reg))
```

```
Waiting for profiling to be done...
```

```
                                    2.5 %
(Intercept)                      0.2379725
sexeHomme                        1.2605519
grpage[25,45)                    0.4195082
grpage[45,65)                    0.2115297
grpage[65,93]                    0.1466915
etudSecondaire                   1.7613423
etudTechnique/Professionnel      1.9764492
etudSupérieur                    4.5427464
etudNA                           4.5265421
religPratiquant occasionnel      0.6767575
religAppartenance sans pratique  0.7067055
religNi croyance ni appartenance 0.5531366
religRejet                       0.3872078
religNSP ou NVPR                 0.3998878
heures.tv                        0.8291800
                                    97.5 %
(Intercept)                       0.8480538
sexeHomme                         1.9106855
grpage[25,45)                     1.0275786
grpage[45,65)                     0.5380619
grpage[65,93]                     0.4296919
etudSecondaire                    3.8240276
etudTechnique/Professionnel       4.1639745
etudSupérieur                     9.7745023
etudNA                           16.5563346
religPratiquant occasionnel       1.4217179
religAppartenance sans pratique   1.4026763
religNi croyance ni appartenance  1.1798422
religRejet                        1.1898605
religNSP ou NVPR                  2.0221544
heures.tv                         0.9459484
```

On pourra faciliter la lecture en combinant les deux :

```
R> exp(cbind(coef(reg), confint(reg)))
```

```
Waiting for profiling to be done...
```

```
(Intercept)                      0.4505151
sexeHomme                        1.5511577
grpage[25,45)                    0.6568456
grpage[45,65)                    0.3377455
grpage[65,93]                    0.2517614
etudSecondaire                   2.5813476
etudTechnique/Professionnel      2.8495358
etudSupérieur                    6.6168632
etudNA                           8.5723745
religPratiquant occasionnel      0.9796133
religAppartenance sans pratique  0.9938431
religNi croyance ni appartenance 0.8072940
religRejet                       0.6819874
religNSP ou NVPR                 0.9200190
heures.tv                        0.8862785
                                     2.5 %
(Intercept)                      0.2379725
sexeHomme                        1.2605519
grpage[25,45)                    0.4195082
grpage[45,65)                    0.2115297
grpage[65,93]                    0.1466915
etudSecondaire                   1.7613423
etudTechnique/Professionnel      1.9764492
etudSupérieur                    4.5427464
etudNA                           4.5265421
religPratiquant occasionnel      0.6767575
religAppartenance sans pratique  0.7067055
religNi croyance ni appartenance 0.5531366
religRejet                       0.3872078
religNSP ou NVPR                 0.3998878
heures.tv                        0.8291800
                                    97.5 %
(Intercept)                       0.8480538
sexeHomme                         1.9106855
grpage[25,45)                     1.0275786
grpage[45,65)                     0.5380619
grpage[65,93]                     0.4296919
etudSecondaire                    3.8240276
etudTechnique/Professionnel       4.1639745
```

```
etudSupérieur                      9.7745023
etudNA                            16.5563346
religPratiquant occasionnel        1.4217179
religAppartenance sans pratique    1.4026763
religNi croyance ni appartenance   1.1798422
religRejet                         1.1898605
religNSP ou NVPR                   2.0221544
heures.tv                          0.9459484
```

Pour savoir si un odds ratio diffère significativement de 1 (ce qui est identique au fait que le coefficient soit différent de 0), on pourra se référer à la colonne *Pr(>|z|)* obtenue avec `summary`.

Si vous disposez de l'extension **questionr**, la fonction `odds.ratio` permet de calculer directement les odds ratio, leur intervalles de confiance et les p-value :

```
R> library(questionr)
   odds.ratio(reg)
```

```
Waiting for profiling to be done...
```

```
                                   OR 2.5 %
(Intercept)                     0.451 0.238
sexeHomme                       1.551 1.261
grpage[25,45)                   0.657 0.420
grpage[45,65)                   0.338 0.212
grpage[65,93]                   0.252 0.147
etudSecondaire                  2.581 1.761
etudTechnique/Professionnel     2.850 1.976
etudSupérieur                   6.617 4.543
etudNA                          8.572 4.527
religPratiquant occasionnel     0.980 0.677
religAppartenance sans pratique 0.994 0.707
religNi croyance ni appartenance 0.807 0.553
religRejet                      0.682 0.387
religNSP ou NVPR                0.920 0.400
heures.tv                       0.886 0.829
                                97.5 %         p
(Intercept)                      0.848 0.0138062
sexeHomme                        1.911 3.488e-05
grpage[25,45)                    1.028 0.0653132
grpage[45,65)                    0.538 4.961e-06
grpage[65,93]                    0.430 4.713e-07
etudSecondaire                   3.824 1.560e-06
etudTechnique/Professionnel      4.164 3.432e-08
etudSupérieur                    9.775 < 2.2e-16
etudNA                          16.556 7.674e-11
religPratiquant occasionnel      1.422 0.9133105
religAppartenance sans pratique  1.403 0.9718010
religNi croyance ni appartenance 1.180 0.2675998
religRejet                       1.190 0.1806239
religNSP ou NVPR                 2.022 0.8392596
heures.tv                        0.946 0.0003255

(Intercept)                     *
sexeHomme                       ***
grpage[25,45)                   .
grpage[45,65)                   ***
grpage[65,93]                   ***
etudSecondaire                  ***
etudTechnique/Professionnel     ***
```

```
etudSupérieur                          ***
etudNA                                 ***
religPratiquant occasionnel
religAppartenance sans pratique
religNi croyance ni appartenance
religRejet
religNSP ou NVPR
heures.tv                              ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

L'extension **effects** propose une représentation graphique résumant les effets de chaque variable du modèle. Pour cela, il suffit d'appliquer la méthode `plot` au résultat de la fonction `allEffects`. Nous obtenons alors la figure ci-dessous.

```
R> library(effects)
   plot(allEffects(reg))
```

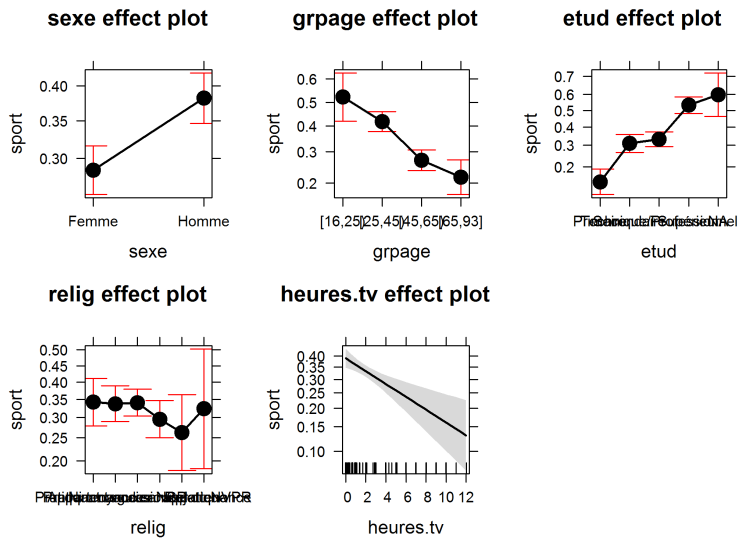



*Figure 1. Représentation graphique de l'effet de chaque variable du modèle logistique*

Une manière de tester la qualité d'un modèle est le calcul d'une *matrice de confusion*, c'est-à-dire le tableau croisé des valeurs observées et celles des valeurs prédites en appliquant le modèle aux données d'origine.

La méthode `predict` avec l'argument `type="response"` permet d'appliquer notre modèle logistique à un tableau de données et renvoie pour chaque individu la probabilité qu'il ait vécu le phénomène étudié.

```
R> sport.pred <- predict(reg, type = "response", newdata =
   d)
   head(sport.pred)
```

```
         1          2          3          4
0.61251214 0.73426878 0.16004519 0.70335574
         5          6
0.07318189 0.34841147
```

Or notre variable étudiée est de type binaire. Nous devons donc transformer nos probabilités prédites en une variable du type « oui / non ». Usuellement, les probabilités prédites seront réunies en deux groupes selon qu'elles soient supérieures ou inférieures à la moitié. La matrice de confusion est alors égale à :

```
R> table(sport.pred > 0.5, d$sport)
```

```
         Non  Oui
  FALSE 1074  384
  TRUE   199  336
```

Nous avons donc 583 (384+199) prédictions incorrectes sur un total de 1993, soit un taux de mauvais classement de 29,3 %.

# Sélection de modèles

Il est toujours tentant lorsque l'on recherche les facteurs associés à un phénomène d'inclure un nombre important de variables explicatives potentielles dans un mmodèle logistique. Cependant, un tel modèle n'est pas forcément le

plus efficace et certaines variables n'auront probablement pas d'effet significatif sur la variable d'intérêt.

La technique de *sélection descendante pas à pas* est une approche visant à améliorer son modèle explicatif[2]. On réalise un premier modèle avec toutes les variables spécifiées, puis on regarde s'il est possible d'améliorer le modèle en supprimant une des variables du modèle. Si plusieurs variables permettent d'améliorer le modèle, on supprimera la variable dont la suppression améliorera le plus le modèle. Puis on recommence le même procédé pour voir si la suppression d'une seconde variable peut encore améliorer le modèle et ainsi de suite. Lorsque le modèle ne peut plus être améliorer par la suppresion d'une variable, on s'arrête.

Il faut également définir un critère pour déterminer la qualité d'un modèle. L'un des plus utilisés est le *Akaike Information Criterion* ou AIC. Plus l'AIC sera faible, meilleure sera le modèle.

La fonction `step` permet justement de sélectionner le meilleur modèle par une procédure pas à pas descendante basée sur la minimisation de l'AIC. La fonction affiche à l'écran les différentes étapes de la sélection et renvoie le modèle final.

```
R> reg2 <- step(reg)
```

```
Start:  AIC=2235.94
sport ~ sexe + grpage + etud + relig + heures.tv

            Df Deviance    AIC
- relig      5   2210.2 2230.2
<none>           2205.9 2235.9
- heures.tv  1   2219.3 2247.3
- sexe       1   2223.2 2251.2
- grpage     3   2258.7 2282.7
- etud       4   2329.6 2351.6

Step:  AIC=2230.15
sport ~ sexe + grpage + etud + heures.tv

            Df Deviance    AIC
<none>           2210.2 2230.2
- heures.tv  1   2223.7 2241.7
- sexe       1   2226.1 2244.1
- grpage     3   2260.3 2274.3
- etud       4   2333.8 2345.8
```

Le modèle initial a un AIC de 2114,8. À la première étape, il apparait que la suppression de la variable religion permet diminuer l'AIC à 2109,1. Lors de la seconde étape, toute suppression d'une autre variable ferait augmenter l'AIC. La procédure s'arrête donc.

# Régression logistique multinomiale

La régression logistique multinomiale est une extension de la régression logistique aux variables qualitatives à trois modalités ou plus. Dans ce cas de figure, chaque modalité de la variable d'intérêt sera comparée à la modalité de réference. Les odds ratio seront donc exprimés par rapport à cette dernière.

Nous allons prendre pour exemple la variable *trav.satisf*, à savoir la satisfaction ou l'insatisfaction au travail.

```
R> freq(d$trav.satisf)
```

```
                  n    % val%
Satisfaction    480 24.0 45.8
Insatisfaction  117  5.9 11.2
Equilibre       451 22.6 43.0
NA              952 47.6   NA
```

Nous allons choisir comme modalité de référence la position intermédiaire, à savoir l'« équilibre ».

```
R> d$trav.satisf <- relevel(d$trav.satisf, "Equilibre")
```

Enfin, nous allons aussi en profiter pour raccourcir les étiquettes de la variable *trav.imp* :

```
R> levels(d$trav.imp) <- c("Le plus", "Aussi", "Moins",
     "Peu")
```

Pour calculer un modèle logistique multinomial, nous allons utiliser la fonction `multinom` de l'extension **nnet**[3]. La syntaxe de `multinom` est similaire à celle de `glm`, le paramètre `family` en moins.

```
R> library(nnet)
   regm <- multinom(trav.satisf ~ sexe + etud + grpage +
     trav.imp, data = d)
```

```
# weights:  39 (24 variable)
initial  value 1151.345679
iter  10 value 977.348901
iter  20 value 969.849189
iter  30 value 969.522965
final  value 969.521855
converged
```

Comme pour la régression logistique, il est possible de réaliser une sélection pas à pas descendante :

```
R> regm2 <- step(regm)
```

```
Start:  AIC=1987.04
trav.satisf ~ sexe + etud + grpage + trav.imp

trying - sexe
# weights:  36 (22 variable)
initial  value 1151.345679
iter  10 value 978.538886
iter  20 value 970.453555
iter  30 value 970.294459
final  value 970.293988
converged
trying - etud
# weights:  27 (16 variable)
initial  value 1151.345679
iter  10 value 987.907714
iter  20 value 981.785467
iter  30 value 981.762800
final  value 981.762781
converged
trying - grpage
# weights:  30 (18 variable)
initial  value 1151.345679
iter  10 value 979.485430
iter  20 value 973.175923
final  value 973.172389
converged
trying - trav.imp
# weights:  30 (18 variable)
initial  value 1151.345679
iter  10 value 998.803976
iter  20 value 994.417973
iter  30 value 994.378914
final  value 994.378869
converged
           Df      AIC
- grpage   18 1982.345
- sexe     22 1984.588
<none>     24 1987.044
- etud     16 1995.526
- trav.imp 18 2024.758
```

```
# weights:  30 (18 variable)
initial  value 1151.345679
iter  10 value 979.485430
iter  20 value 973.175923
final  value 973.172389
converged

Step:  AIC=1982.34
trav.satisf ~ sexe + etud + trav.imp

trying - sexe
# weights:  27 (16 variable)
initial  value 1151.345679
iter  10 value 976.669670
iter  20 value 973.928385
iter  20 value 973.928377
iter  20 value 973.928377
final  value 973.928377
converged
trying - etud
# weights:  18 (10 variable)
initial  value 1151.345679
iter  10 value 988.413720
final  value 985.085797
converged
trying - trav.imp
# weights:  21 (12 variable)
initial  value 1151.345679
iter  10 value 1001.517287
final  value 998.204280
converged
           Df      AIC
- sexe     16 1979.857
<none>     18 1982.345
- etud     10 1990.172
- trav.imp 12 2020.409
# weights:  27 (16 variable)
initial  value 1151.345679
iter  10 value 976.669670
iter  20 value 973.928385
```

```
iter  20 value 973.928377
iter  20 value 973.928377
final  value 973.928377
converged

Step:  AIC=1979.86
trav.satisf ~ etud + trav.imp

trying - etud
# weights:  15 (8 variable)
initial  value 1151.345679
iter  10 value 986.124104
final  value 986.034023
converged
trying - trav.imp
# weights:  18 (10 variable)
initial  value 1151.345679
iter  10 value 1000.225356
final  value 998.395273
converged
           Df      AIC
<none>     16 1979.857
- etud      8 1988.068
- trav.imp 10 2016.791
```

La plupart des fonctions vues précédemment fonctionnent :

```
R> summary(regm2)
```

```
Call:
multinom(formula = trav.satisf ~ etud + trav.imp, data = d)

Coefficients:
               (Intercept) etudSecondaire
Satisfaction    -0.1110996     0.04916210
Insatisfaction  -1.1213760    -0.09737523
               etudTechnique/Professionnel
Satisfaction                    0.07793241
Insatisfaction                  0.08392603
               etudSupérieur      etudNA
Satisfaction      0.69950061 -0.53841577
Insatisfaction    0.07755307 -0.04364055
               trav.impAussi trav.impMoins
Satisfaction       0.2578973    -0.1756206
Insatisfaction    -0.2279774    -0.5330349
               trav.impPeu
Satisfaction    -0.5995051
Insatisfaction   1.3401509

Std. Errors:
               (Intercept) etudSecondaire
Satisfaction     0.4520902      0.2635573
Insatisfaction   0.6516992      0.3999875
               etudTechnique/Professionnel
Satisfaction                     0.2408483
Insatisfaction                   0.3579684
               etudSupérieur    etudNA
Satisfaction       0.2472571 0.5910993
Insatisfaction     0.3831110 0.8407592
               trav.impAussi trav.impMoins
Satisfaction       0.4260623     0.4115818
Insatisfaction     0.6213781     0.5941721
               trav.impPeu
Satisfaction     0.5580115
Insatisfaction   0.6587383

Residual Deviance: 1947.857
AIC: 1979.857
```

```
R> odds.ratio(regm2)
```

```
                                              OR
Satisfaction/(Intercept)                   0.895
Satisfaction/etudSecondaire                1.050
Satisfaction/etudTechnique/Professionnel   1.081
Satisfaction/etudSupérieur                 2.013
Satisfaction/etudNA                        0.584
Satisfaction/trav.impAussi                 1.294
Satisfaction/trav.impMoins                 0.839
Satisfaction/trav.impPeu                   0.549
Insatisfaction/(Intercept)                 0.326
Insatisfaction/etudSecondaire              0.907
Insatisfaction/etudTechnique/Professionnel 1.088
Insatisfaction/etudSupérieur               1.081
Insatisfaction/etudNA                      0.957
Insatisfaction/trav.impAussi               0.796
Insatisfaction/trav.impMoins               0.587
Insatisfaction/trav.impPeu                 3.820
                                           2.5 %
Satisfaction/(Intercept)                   0.369
Satisfaction/etudSecondaire                0.627
Satisfaction/etudTechnique/Professionnel   0.674
Satisfaction/etudSupérieur                 1.240
Satisfaction/etudNA                        0.183
Satisfaction/trav.impAussi                 0.561
Satisfaction/trav.impMoins                 0.374
Satisfaction/trav.impPeu                   0.184
Insatisfaction/(Intercept)                 0.091
Insatisfaction/etudSecondaire              0.414
Insatisfaction/etudTechnique/Professionnel 0.539
Insatisfaction/etudSupérieur               0.510
Insatisfaction/etudNA                      0.184
Insatisfaction/trav.impAussi               0.236
Insatisfaction/trav.impMoins               0.183
Insatisfaction/trav.impPeu                 1.050
                                           97.5 %
Satisfaction/(Intercept)                    2.171
Satisfaction/etudSecondaire                 1.761
Satisfaction/etudTechnique/Professionnel    1.733
Satisfaction/etudSupérieur                  3.268
Satisfaction/etudNA                         1.859
```

```
Satisfaction/trav.impAussi                   2.983
Satisfaction/trav.impMoins                   1.880
Satisfaction/trav.impPeu                     1.639
Insatisfaction/(Intercept)                   1.169
Insatisfaction/etudSecondaire                1.987
Insatisfaction/etudTechnique/Professionnel  2.194
Insatisfaction/etudSupérieur                 2.290
Insatisfaction/etudNA                        4.974
Insatisfaction/trav.impAussi                 2.691
Insatisfaction/trav.impMoins                 1.880
Insatisfaction/trav.impPeu                  13.891
                                                p
Satisfaction/(Intercept)                    0.805878
Satisfaction/etudSecondaire                 0.852027
Satisfaction/etudTechnique/Professionnel   0.746260
Satisfaction/etudSupérieur                  0.004669
Satisfaction/etudNA                         0.362363
Satisfaction/trav.impAussi                  0.544977
Satisfaction/trav.impMoins                  0.669600
Satisfaction/trav.impPeu                    0.282661
Insatisfaction/(Intercept)                  0.085306
Insatisfaction/etudSecondaire               0.807660
Insatisfaction/etudTechnique/Professionnel 0.814635
Insatisfaction/etudSupérieur                0.839581
Insatisfaction/etudNA                       0.958603
Insatisfaction/trav.impAussi                0.713701
Insatisfaction/trav.impMoins                0.369663
Insatisfaction/trav.impPeu                  0.041909

Satisfaction/(Intercept)
Satisfaction/etudSecondaire
Satisfaction/etudTechnique/Professionnel
Satisfaction/etudSupérieur                  **
Satisfaction/etudNA
Satisfaction/trav.impAussi
Satisfaction/trav.impMoins
Satisfaction/trav.impPeu
Insatisfaction/(Intercept)                  .
Insatisfaction/etudSecondaire
Insatisfaction/etudTechnique/Professionnel
```

```
Insatisfaction/etudSupérieur
Insatisfaction/etudNA
Insatisfaction/trav.impAussi
Insatisfaction/trav.impMoins
Insatisfaction/trav.impPeu                    *
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

De même, il est possible de calculer la matrice de confusion :

```
R> table(predict(regm2, newdata = d), d$trav.satisf)
```

```
                Equilibre Satisfaction
  Equilibre           262          211
  Satisfaction        171          258
  Insatisfaction       18           11

                Insatisfaction
  Equilibre                49
  Satisfaction             45
  Insatisfaction           23
```

1. Pour plus de détails, voir http://www.spc.univ-lyon1.fr/polycop/odds%20ratio.htm.↩
2. Il existe également des méthodes de *sélection ascendante pas à pas*, mais nous les aborderons pas ici.↩
3. Une alternative est d'avoir recours à l'extension **mlogit** que nous n'aborderons pas ici. Voir http://www.ats.ucla.edu/stat/r/dae/mlogit.htm (en anglais) pour plus de détails.↩

# Données pondérées

**NOTE**

La version originale de ce chapitre a été écrite par Julien Barnier dans le cadre du support de cours Introduction à R, complétée par Joseph Larmarange dans Introduction à l'analyse d'enquêtes avec R.

S'il est tout à fait possible de travailler avec des données pondérées sous R, cette fonctionnalité n'est pas aussi bien intégrée que dans la plupart des autres logiciels de traitement statistique. En particulier, il y a plusieurs manières possibles de gérer la pondération.

Dans ce qui suit, on utilisera le jeu de données tiré de l'enquête *Histoire de vie* et notamment sa variable de pondération *poids*[1].

```
R> library(questionr)
   data(hdv2003)
   d <- hdv2003
   range(d$poids)
```

```
[1]    78.07834 31092.14132
```

## Options de certaines fonctions

Tout d'abord, certaines fonctions de **R** acceptent en argument un vecteur permettant de pondérer les observations (l'option est en général nommée `weights` ou `row.w`). C'est le cas par exemple des méthodes d'estimation de

modèles linéaires ( `lm` ) ou de modèles linéaires généralisés ( `glm` ), ou dans les analyses de correspondances[2] des extensions **ade4** ou **FactoMineR**.

Par contre cette option n'est pas présente dans les fonctions de base comme `mean` , `var` , `table` ou `chisq.test` .

# Fonctions de l'extension questionr

L'extension **questionr** propose quelques fonctions permettant de calculer des statistiques simples pondérées[3] :

- `wtd.mean` : moyenne pondérée
- `wtd.var` : variance pondérée
- `wtd.table` : tris à plat et tris croisés pondérés

On les utilise de la manière suivante :

library(questionr) mean(d*age*)*wtd.mean*(*d*age, weights = d*poids*)*wtd.var*(*d*age, weights = d$poids)

Pour les tris à plat, on utilise la fonction wtd.table à laquelle on passe la variable en paramètre :

wtd.table(d*sexe*, *weights* = *d*poids)

Pour un tri croisé, il suffit de passer deux variables en paramètres :

wtd.table(d*sexe*, *d*hard.rock, weights = d$poids)

Ces fonctions admettent notamment les deux options suivantes :

- `na.rm` : si `TRUE` , on ne conserve que les observations sans valeur manquante.
- `normwt` : si `TRUE` , on normalise les poids pour que les effectifs totaux pondérés soient les mêmes que les effectifs initiaux. Il faut utiliser cette option, notamment si on souhaite appliquer un test sensible aux effectifs comme le $\chi^2$.

Ces fonctions rendent possibles l'utilisation des statistiques descriptives les plus simples et le traitement des tableaux croisés (les fonctions `lprop` , `cprop` ou `chisq.test` peuvent être appliquées au résultat d'un `wtd.table` ) mais restent limitées en termes de tests statistiques ou de graphiques. . .

# Données pondérées avec l'extension survey

L'extension **survey** est spécialement dédiée au traitement d'enquêtes ayant des techniques d'échantillonnage et de pondération potentiellement très complexes.

L'extension s'installe comme la plupart des autres :

```
R> install.packages("survey")
```

Le site officiel (en anglais) comporte beaucoup d'informations, mais pas forcément très accessibles : http://faculty.washington.edu/tlumley/survey/.

Pour utiliser les fonctionnalités de l'extension, on doit d'abord définir le *plan d'échantillonnage* ou *design* de notre enquête. C'est-à-dire indiquer quel type de pondération nous souhaitons lui appliquer.

Dans notre cas nous utilisons plan d'échantillonnage le plus simple, avec une variable de pondération déjà calculée. Ceci se fait à l'aide de la fonction `svydesign` :

```
R> library(survey)
   dw <- svydesign(ids = ~1, data = d, weights = ~d$poids)
```

Cette fonction crée un nouvel objet, que nous avons nommé `dw`. Cet objet n'est pas à proprement parler un tableau de données, mais plutôt un tableau de données plus une méthode de pondération. `dw` et `d` sont des objets distincts, les opérations effectuées sur l'un n'ont pas d'influence sur l'autre. On peut cependant retrouver le contenu de `d` depuis `dw` en utilisant `dw$variables` :

```
R> mean(d$age)
[1] 48.157
R> mean(dw$variables$age)
[1] 48.157
```

Lorsque notre plan d'échantillonnage est déclaré, on peut lui appliquer une série de fonctions permettant d'effectuer diverses opérations statistiques en tenant compte de la pondération. On citera notamment :

- `svymean`, `svyvar`, `svytotal`, `svyquantile` : statistiques univariées
- `svytable` : tableaux croisés
- `svychisq` : test du $\chi^2$
- `svyby` : statistiques selon un facteur
- `svyttest` : test *t* de comparaison de moyennes
- `svyglm` : modèles linéaires généralisés (dont régression logistique)
- `svyplot`, `svyhist`, `svyboxplot` : fonctions graphiques

D'autres fonctions sont disponibles, comme `svyratio`, mais elles ne seront pas abordées ici.

Pour ne rien arranger, ces fonctions prennent leurs arguments sous forme de formules, c'est-à-dire pas de la manière habituelle. En général l'appel de fonction se fait en spécifiant d'abord les variables d'intérêt sous forme de formule, puis l'objet *survey.design*.

L'intervalle de confiance d'une moyenne s'obtient avec `confint` et celui d'une proportion avec `svyciprop`.

Voyons tout de suite quelques exemples[4] :

```
R> svymean(~age, dw)
```

```
      mean     SE
age 46.347 0.5284
```

```
R> confint(svymean(~age, dw))  # Intervalle de confiance
```

```
      2.5 %   97.5 %
age 45.3117 47.38282
```

```
R> svyquantile(~age, dw, quantile = c(0.25, 0.5, 0.75),
    ci = TRUE)
```

```
$quantiles
    0.25 0.5 0.75
age   31  45   60

$CIs
, , age

       0.25 0.5 0.75
(lower   30  43   58
upper)   32  47   62
```

```
R> svyvar(~heures.tv, dw, na.rm = TRUE)
```

```
          variance     SE
heures.tv   2.9886 0.1836
```

Pour comparer deux moyennes à l'aide d'un test *t* on aura recours à `svyttest` :

```
R> svyttest(age ~ sexe, dw)
```

```
    Design-based t-test

data:  age ~ sexe
t = 2.0404, df = 1998, p-value = 0.04144
alternative hypothesis: true difference in mean is not
equal to 0
sample estimates:
difference in mean
          2.141129
```

Les tris à plat se déclarent en passant comme argument le nom de la variable précédé d'un tilde ( `~` ), tandis que les tableaux croisés utilisent les noms des deux variables séparés par un signe plus ( `+` ) et précédés par un tilde ( `~` ).

```
R> svytable(~sexe, dw)
```

```
sexe
  Homme   Femme
5149382 5921844
```

```
R> svyciprop(~sexe, dw)  # Intervalle de confiance
```

```
            2.5% 97.5%
sexe 0.535 0.507  0.56
```

```
R> svytable(~sexe + clso, dw)
```

```
       clso
sexe           Oui        Non Ne sait pas
  Homme 2658744.04 2418187.64    72450.75
  Femme 2602031.76 3242389.36    77422.79
```

On peut récupérer le tableau issu de `svytable` dans un objet et le réutiliser ensuite comme n'importe quel tableau croisé :

```
R> tab <- svytable(~sexe + clso, dw)
   tab
```

```
       clso
sexe           Oui        Non Ne sait pas
  Homme 2658744.04 2418187.64    72450.75
  Femme 2602031.76 3242389.36    77422.79
```

Les fonctions `lprop` et `cprop` de **questionr** sont donc tout à fait compatibles avec l'utilisation de **survey**.

```
R> lprop(tab)
```

```
          clso
sexe       Oui   Non   Ne sait pas Total
  Homme     51.6  47.0   1.4       100.0
  Femme     43.9  54.8   1.3       100.0
  Ensemble  47.5  51.1   1.4       100.0
```

La fonction `freq` peut également être utilisée si on lui passe en argument non pas la variable elle-même, mais son tri à plat obtenu avec `svytable` svychisq(~sexe + clso, dw):

```
R> tab <- svytable(~peche.chasse, dw)
   freq(tab, total = TRUE)
```

```
              n     %  val%
Non     9716683  87.8  87.8
Oui     1354544  12.2  12.2
Total 11071226 100.0 100.0
```

Par contre, il ne faut pas utiliser `chisq.test` sur un tableau généré par `svytable`{data-package="survey">. Les effectifs étant extrapolés à partir de la pondération, les résultats du test seraient complètement faussés. Si on veut faire un test du $\chi^2$ sur un tableau croisé pondéré, il faut utiliser `svychisq` :

```
R> svychisq(~sexe + clso, dw)
```

```
    Pearson's X^2: Rao & Scott adjustment

data:  svychisq(~sexe + clso, dw)
F = 3.3331, ndf = 1.9734, ddf = 3944.9000,
p-value = 0.03641
```

Le principe de la fonction `svyby` est similaire à celui de `tapply`[5]. Elle permet de calculer des statistiques selon plusieurs sous-groupes définis par un facteur. Par exemple :

```
R> svyby(~age, ~sexe, dw, svymean)
```

```
       sexe      age        se
Homme Homme 45.20200 0.7419450
Femme Femme 47.34313 0.7420836
```

```
R> confint(svyby(~age, ~sexe, dw, svymean))  # Intervalles
   de confiance
```

```
         2.5 %   97.5 %
Homme 43.74781 46.65618
Femme 45.88867 48.79758
```

survey est également capable de produire des graphiques à partir des données pondérées. Quelques exemples :

```
R> par(mfrow = c(2, 2))
   svyplot(~age + heures.tv, dw, col = "red", main =
   "Bubble plot")
   svyhist(~heures.tv, dw, col = "peachpuff", main =
   "Histogramme")
   svyboxplot(age ~ 1, dw, main = "Boxplot simple",
   ylab = "Âge")
   svyboxplot(age ~ sexe, dw, main = "Boxplot double",
     ylab = "Âge", xlab = "Sexe")
```

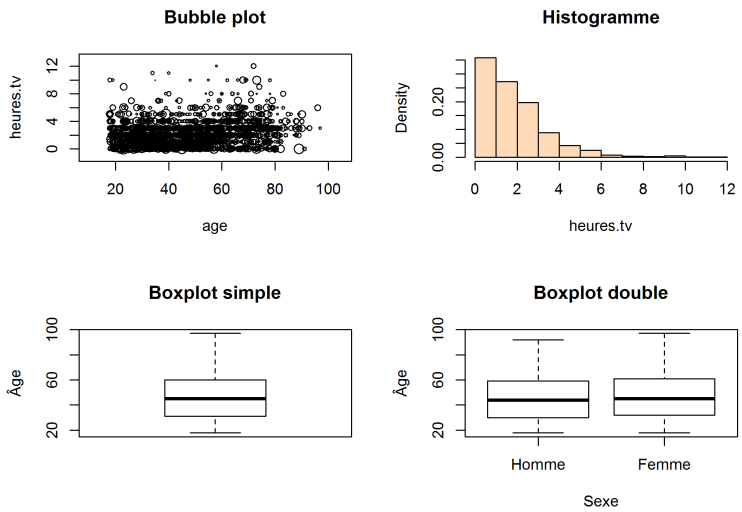



*Figure 1. Fonctions graphiques de l'extension survey*

Enfin, survey fournit une fonction `svyglm` permettant de calculer un modèle statistique tout en prenant en compte le plan d'échantillonnage spécifié. La syntaxe de `svyglm` est proche de celle de `glm`. Cependant, le cadre d'une régression logistique, il est nécessaire d'utiliser `family = quasibinomial()` afin d'éviter un message d'erreur indiquant un nombre non entier de succès :

```
R> reg <- svyglm(sport ~ sexe + age + relig + heures.tv,
     dw, family = binomial())
```

```
Warning: non-integer #successes in a binomial glm!
```

```
R> reg <- svyglm(sport ~ sexe + age + relig + heures.tv,
     dw, family = quasibinomial())
   reg
```

```
Independent Sampling design (with replacement)
svydesign(ids = ~1, data = d, weights = ~d$poids)

Call:  svyglm(formula = sport ~ sexe + age + relig +
heures.tv, dw,
    family = quasibinomial())

Coefficients:
                      (Intercept)
                          1.90117
                        sexeFemme
                         -0.36526
                              age
                         -0.04127
     religPratiquant occasionnel
                          0.05577
 religAppartenance sans pratique
                          0.16367
religNi croyance ni appartenance
                          0.03988
                       religRejet
                         -0.14862
                 religNSP ou NVPR
                         -0.22682
                        heures.tv
                         -0.18204

Degrees of Freedom: 1994 Total (i.e. Null);  1986 Residual
  (5 observations deleted due to missingness)
Null Deviance:      2672
Residual Deviance: 2378     AIC: NA
```

Le résultat obtenu est similaire à celui de `glm` et l'on peut utiliser sans problème les fonctions `coef`, `confint`, `odds.ratio` ou `predict` abordées dans le chapitre sur la régression logistique.

```
R> odds.ratio(reg)
```

```
                                   OR 2.5 %
(Intercept)                     6.694 3.670
sexeFemme                       0.694 0.540
age                             0.960 0.952
religPratiquant occasionnel     1.057 0.659
religAppartenance sans pratique 1.178 0.759
religNi croyance ni appartenance 1.041 0.646
religRejet                      0.862 0.428
religNSP ou NVPR                0.797 0.325
heures.tv                       0.834 0.770
                                97.5 %         p
(Intercept)                     12.209 6.848e-10
sexeFemme                        0.892  0.004325
age                              0.967 < 2.2e-16
religPratiquant occasionnel      1.696  0.817180
religAppartenance sans pratique  1.827  0.465321
religNi croyance ni appartenance 1.676  0.869658
religRejet                       1.738  0.677858
religNSP ou NVPR                 1.956  0.620504
heures.tv                        0.902 6.786e-06

(Intercept)                     ***
sexeFemme                       **
age                             ***
religPratiquant occasionnel
religAppartenance sans pratique
religNi croyance ni appartenance
religRejet
religNSP ou NVPR
heures.tv                       ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Dans ses dernières versions, **survey** fournit une méthode `AIC.svyglm` permettant d'estimer un AIC sur un modèle calculé avec `svyglm`. Il est dès lors possible d'utiliser la fonction `step` pour réaliser une *sélection descendante pas à pas*.

L'extension **effects** n'est quant à elle pas compatible avec `svyglm`[6].

# Définir un plan d'échantillonage complexe avec survey

L'extension **survey** ne permet pas seulement d'indiquer une variable de pondération mais également de prendre les spécificités du plan d'échantillonnage (strates, grappes, …). Le plan d'échantillonnage ne joue pas seulement sur la pondération des données, mais influence le calcul des variances et par ricochet tous les tests statistiques. Deux échantillons identiques avec la même variable de pondération mais des designs différents produiront les mêmes moyennes et proportions mais des intervalles de confiance différents.

## Différents types d'échantillonnage

L'*échantillonnage aléatoire simple* ou *échantillonnage équiprobable* est une méthode pour laquelle tous les échantillons possibles (de même taille) ont la même probabilité d'être choisis et tous les éléments de la population ont une chance égale de faire partie de l'échantillon. C'est l'échantillonnage le plus simple : chaque individu à la même probabilité d'être sélectionné.

L'*échantillonnage stratifié* est une méthode qui consiste d'abord à subdiviser la population en groupes homogènes (strates) pour ensuite extraire un échantillon aléatoire de chaque strate. Cette méthode suppose la connaissance de la structure de la population. Pour estimer les paramètres, les résultats doivent être pondérés par l'importance relative de chaque strate dans la population.

L'*échantillonnage par grappes* est une méthode qui consiste à choisir un échantillon aléatoire d'unités qui sont elles-mêmes des sous-ensembles de la population (« grappes »). Cette méthode suppose que les unités de chaque grappe sont représentatives. Elle possède l'avantage d'être souvent plus économique.

Il est possible de combiner plusieurs de ces approches. Par exemple, les *Enquêtes Démographiques et de Santé[7] (EDS) sont des enquêtes stratifiées en grappes à deux degrés. Dans un premier temps, la population est divisée en strates par région et milieu de résidence. Dans chaque strate, des zones d'enquêtes, correspondant à des unités de recensement, sont tirées au sort avec une probabilité proportionnelle au nombre de ménages de chaque zone au dernier recensement de population. Enfin, au sein de chaque zone d'enquête sélectionnée, un recensement de l'ensemble des ménages est effectué puis un nombre identique de ménages par zone d'enquête est tiré au sort de manière alétoire simple.

## Les options de svydesign

La fonction `svydesign` accepte plusieus arguments décrits sur sa page d'aide (obtenue avec la commande `?svydesign`).

L'agument `data` permet de spécifier le tableau de données contenant les observations.

L'argument `ids` est obligatoire et spécifie sous la forme d'une formule les identifiants des différents niveaux d'un tirage en grappe. S'il s'agit d'un échantillon aléatoire simple, on entrera `ids=~1`. Autre situation : supposons une étude portant sur la population française. Dans un premier temps, on a tiré au sort un certain nombre de départements français. Dans un second temps, on tire au sort dans chaque département des communes. Dans chaque commune sélectionnée, on tire au sort des quartiers. Enfin, on interroge de manière exhaustive toutes les personnes habitant les quartiers enquêtés. Notre fichier de données devra donc comporter pour chaque observation les variables *id_departement*, *id_commune* et *id_quartier*. On écrira alors pour l'argument `ids` la valeur suivante : `ids=~id_departement+id_commune+id_quartier`.

Si l'échantillon est stratifié, on spécifiera les strates à l'aide de l'argument `strata` en spécifiant la variable contenant l'identifiant des strates. Par exemple : `strata=~id_strate`.

Il faut encore spécifier les probabilités de tirage de chaque cluster ou bien la pondération des individus. Si l'on dispose de la probabilité de chaque observation d'être sélectionnée, on utilisera l'argument `probs`. Si, par contre,

on connaît la pondération de chaque observation (qui doit être proportionnelle à l'inverse de cette probabilité), on utilisera l'argument `weights`.

Si l'échantillon est stratifié, qu'au sein de chaque strate les individus ont été tirés au sort de manière aléatoire et que l'on connaît la taille de chaque strate, il est possible de ne pas avoir à spécifier la probabilité de tirage ou la pondération de chaque observation. Il est préférable de fournir une variable contenant la taille de chaque strate à l'argument `fpc`. De plus, dans ce cas-là, une petite correction sera appliquée au modèle pour prendre en compte la taille finie de chaque strate.

Quelques exemples :

```
R> # Échantillonnage aléatoire simple
   plan <- svydesign(ids = ~1, data = donnees)

   # Échantillonnage stratifié à un seul niveau (la
   # taille de chaque strate est connue)
   plan <- svydesign(ids = ~1, data = donnees, fpc =
   ~taille)

   # Échantillonnage en grappes avec tirages à quatre
   # degrés (departement, commune, quartier,
   # individus). La probabilité de tirage de chaque
   # niveau de cluster est connue.
   plan <- svydesign(ids = ~id_departement + id_commune +
     id_quartier, data = donnees, Probs =
   ~proba_departement +
     proba_commune + proba_quartier)

   # Échantillonnage stratifié avec tirage à deux
   # degrés (clusters et individus). Le poids
   # statistiques de chaque observation est connu.
   plan <- svydesign(ids = ~id_cluster, data = donnees,
     strata = ~id_strate, weights = ~poids)
```

Prenons l'exemple d'une enquête démographique et de santé. Le nom des différentes variables est standardisé et commun quelle que soit l'enquête. Nous supposerons que vous avez importé le fichier *individus* dans un tableau de données nommés `eds`. Le poids statistique de chaque individu est fourni par la variable *V005* qui doit au préalable être divisée par un million. Les grappes d'échantillonnage au premier degré sont fournies par la variable *V021 (primary*

*sample unit)*. Si elle n'est pas renseignée, on pourra utilisier le numéro de grappe *V001*. Enfin, le milieu de résidence (urbain / rural) est fourni par *V025* et la région par *V024*. Pour rappel, l'échantillon a été stratifié à la fois par région et par mileu de résidence. Certaines enquêtes fournissent directement un numéro de strate via *V022*. Si tel est le cas, on pourra préciser le plan d'échantillonnage ainsi :

```
R> eds$poids <- eds$V005/1e+06
   design.eds <- svydesign(ids = ~V021, data = eds, strata
   = ~V022,
     weights = ~poids)
```

Si V022 n'est pas fourni mais que l'enquête a bien été stratifié par région et milieu de résidence (vérifiez toujours le premier chapitre du rapport d'enquête), on pourra créer une variable strate ainsi [8] :

```
R> eds$strate <- as.factor(as.integer(eds$V024) * 10 +
     as.integer(eds$V025))
   levels(eds$strate) <- c(paste(levels(eds$V024),
   "Urbain"),
     paste(levels(eds$V024), "Rural"))
   design.eds <- svydesign(ids = ~V021, data = eds, strata
   = ~strate,
     weights = ~poids)
```

## Extraire un sous-échantillon

Si l'on souhaite travailler sur un sous-échantillon tout en gardant les informations d'échantillonnage, on utilisera la fonction `subset` présentée en détail dans le chapitre Manipulation de données.

```
R> sous <- subset(dw, sexe == "Femme" & age >= 40)
```

# Conclusion

En attendant mieux, la gestion de la pondération sous **R** n'est sans doute pas ce qui se fait de plus pratique et de plus simple. On pourra quand même donner les conseils suivants :

- utiliser les options de pondération des fonctions usuelles ou les fonctions d'extensions comme **questionr** pour les cas les plus simples ;
- si on utilise **survey**, effectuer autant que possible tous les recodages et manipulations sur les données non pondérées ;
- une fois les recodages effectués, on déclare le design et on fait les analyses en tenant compte de la pondération ;
- surtout ne jamais modifier les variables du design. Toujours effectuer recodages et manipulations sur les données non pondérées, puis redéclarer le design pour que les mises à jour effectuées soient disponibles pour l'analyse.

---

1. On notera que cette variable est utilisée à titre purement illustratif. Le jeu de données étant un extrait d'enquête et la variable de pondération n'ayant pas été recalculée, elle n'a ici à proprement parler aucun sens.↩
2. Voir le chapitre dédié à l'analyse des correspondances, MAJ LIEN.↩
3. Les fonctions `wtd.mean` et `wtd.var` sont des copies conformes des fonctions du même nom de l'extension **Hmisc** de Frank Harrel. **Hmisc** étant une extension « de taille », on a préféré recopié les fonctions pour limiter le poids des dépendances.↩
4. Pour d'autres exemples, voir http://www.ats.ucla.edu/stat/r/faq/svy_r_oscluster.htm (en anglais).↩
5. La fonction `tapply` est présentée plus en détails dans le chapitre Manipulation de données.↩
6. Compatibilité qui pourra éventuellement être introduite dans une future version de l'extension.↩
7. Vaste programme d'enquêtes réalisées à intervalles réguliers dans les pays en dévelop'ent, disponibles sur http://www.dhsprogram.com/.↩

8. L'astuce consiste à utiliser as.integer pour obetnir le code des facteurs et non leur valeur textuelle. L'addition des deux valeurs après multiplication du code de la région par 10 permet d'obtenir une valeur unique pour chaque combinaison des deux variables. On retransforme le résultat en facteurs puis on modifie les étiquettes des modalités.↩

# Analyse des correspondances multiples (ACM)

**NOTE**

La version originale de ce chapitre a été écrite par Joseph Larmarange dans le cadre du support de cours Introduction à l'analyse d'enquêtes avec R.

Il existe plusieurs techniques d'analyse factorielle dont les plus courantes sont l'*analyse en composante principale* (ACP) portant sur des variables quantitatives, l'*analyse factorielle des correspondances* (AFC) portant sur deux variables qualitatives et l'*analyse des correspondances multiples* (ACM) portant sur plusieurs variables qualitatives (il s'agit d'une extension de l'AFC). Pour combiner des variables à la fois quantitatives et qualitatives, on pourra avoir recours à l'*analyse mixte de Hill et Smith*.

Bien que ces techniques soient disponibles dans les extensions standards de **R**, il est souvent préférable d'avoir recours à deux autres extensions plus complètes, **ade4** et **FactoMineR**, chacune ayant ses avantages et des possibilités différentes. Voici les fonctions les plus fréquentes :

| Analyse | Variables | Fonction standard | Fonction ade4 | Fonctions FactoMineR |
|---|---|---|---|---|
| ACP | plusieurs variables quantitatives | `princomp` | `dudi.pca` | `PCA` |
| AFC | deux variables qualitatives | `corresp` | `dudi.coa` | `CA` |
| ACM | plusieurs variables qualitatives | `mca` | `dudi.acm` | `MCA` |
| Analyse mixte de Hill et Smith | plusieurs variables quantitatives et/ou qualitatives | — | `dudi.mix` | — |

Dans la suite de ce chapitre, nous n'arboderons que l'analyse des correspondances multiples (ACM).

> **INFO**
>
> On trouvera également de nombreux supports de cours en français sur l'analyse factorielle sur le site de François Gilles Carpentier : http://geai.univ-brest.fr/~carpenti/.

# Principe général

L'analyse des correspondances multiples est une technique descriptive visant à résumer l'information contenu dans un grand nombre de variables afin de faciliter l'interprétention des corrélations existantes entre ces différentes variables. On cherche à savoir quelles sont les modalités corrélées entre elles.

L'idée générale est la suivante[1]. L'ensemble des individus peut être représenté dans un espace à plusieurs dimensions où chaque axe représente les différentes variables utilisées pour décrire chaque individu. Plus précisément, pour chaque variable qualitative, il y a autant d'axes que de modalités moins un. Ainsi il faut trois axes pour décrire une variable à quatre modalités. Un tel nuage de points est aussi difficile à interpréter que de lire directement le fichier de données. On ne voit pas es corrélations qu'il peut y avoir entre modalités, par exemple qu'aller au cinéma est plus fréquent chez les personnes habitant en milieu urbain. Afin de mieux représenter ce nuage de points, on va procéder à un changement de systèmes de coordonnées. Les individus seront dès lors projetés et représentés sur un nouveau système d'axe. Ce nouveau système d'axes est choisis de telle

manière que la majorité des variations soit concentrées sur les premiers axes. Les deux-trois premiers axes permettront d'expliquer la majorité des différences observées dans l'échantillon, les autres axes n'apportant qu'une faible part additionnelle d'information. Dès lors, l'analyse pourra se concentrer sur ses premiers axes qui constitueront un bon résumé des variations observables dans l'échantillon.

Avant toute ACM, il est indispensable de réaliser une analyse préliminaire de chaque variable, afin de voir si toutes les classes sont aussi bien représentées ou s'il existe un déséquilibre. L'ACM est sensible aux effectifs faibles, aussi regrouper les classes quand cela est nécessaire.

# ACM avec ade4

Si l'extension **ade4** n'est pas présente sur votre PC, il vous faut l'installer :

```
R> install.packages("ade4", dep = TRUE)
```

Dans tous les cas, il faut penser à la charger en mémoire :

```
R> library(ade4)
```

Comme précédemment, nous utiliserons le fichier de données `hdv2003` fourni avec l'extension **questionr**.

```
R> library(questionr)
   data(hdv2003)
   d <- hdv2003
```

En premier lieu, comme dans le chapitre sur la régression logistique, nous allons créer une variable groupe d'âges et regrouper les modalités de la variable « niveau d'étude ».

```
R> d$grpage <- cut(d$age, c(16, 25, 45, 65, 93), right =
   FALSE,
     include.lowest = TRUE)
   d$etud <- d$nivetud
   levels(d$etud) <- c("Primaire", "Primaire", "Primaire",
     "Secondaire", "Secondaire", "Technique/Professionnel",
     "Technique/Professionnel", "Supérieur")
```

Ensuite, nous allons créer un tableau de données ne contenant que les variables que nous souhaitons prendre en compte pour notre analyse factorielle.

```
R> d2 <- d[, c("grpage", "sexe", "etud", "peche.chasse",
     "cinema", "cuisine", "bricol", "sport", "lecture.bd")]
```

Le calcul de l'ACM se fait tout simplement avec la fonction `dudi.acm`{data-package="ade4">.

```
R> acm <- dudi.acm(d2)
```

Par défaut, la fonction affichera le graphique des valeurs propres de chaque axe (nous y reviendrons) et vous demandera le nombre d'axes que vous souhaitez conserver dans les résultats. Le plus souvent, cinq axes seront largement plus que suffisants. Vous pouvez également éviter cette étape en indiquant directement à dudi.acm de vous renvoyer les cinq premiers axes ainsi :

```
R> acm <- dudi.acm(d2, scannf = FALSE, nf = 5)
```

Le graphique des valeurs propres peut être reproduit avec `screeplot` :

```
R> screeplot(acm)
```

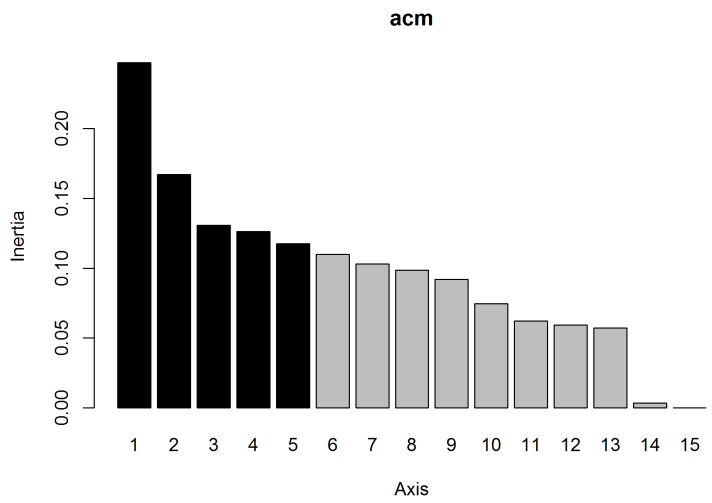



*Figure 1. Valeurs propres ou inerties de chaque axe*

Les mêmes valeurs pour les premiers axes s'obtiennent également avec `summary`[2] :

```
R> summary(acm)
```

```
Class: acm dudi
Call: dudi.acm(df = d2, scannf = FALSE, nf = 5)

Total inertia: 1.451

Eigenvalues:
    Ax1     Ax2     Ax3     Ax4     Ax5
 0.2474  0.1672  0.1309  0.1263  0.1176

Projected inertia (%):
    Ax1     Ax2     Ax3     Ax4     Ax5
 17.055  11.525   9.022   8.705   8.109

Cumulative projected inertia (%):
    Ax1   Ax1:2   Ax1:3   Ax1:4   Ax1:5
  17.06   28.58   37.60   46.31   54.42

(Only 5 dimensions (out of 15) are shown)
```

L'inertie totale est de 1,451 et l'axe 1 en explique 0,1474 soit 17 %. L'inertie projetée cumulée nous indique que les deux premiers axes expliquent à eux seuls 29 % des variations observées dans notre échantillon.

Pour comprendre la signification des différents axes, il importe d'identifier quelles sont les variables/ modalités qui contribuent le plus à chaque axe. Une première représentation graphique est le cercle de corrélation des modalités. Pour cela, on aura recours à `s.corcicle`. On indiquera d'abord `acm$co` si l'on souhaite représenter les modalités ou `acm$li` si l'on souhaite représenter les individus. Les deux chiffres suivant indiquent les deux axes que l'on souhaite afficher (dans le cas présent les deux premiers axes). Enfin, le paramètre `clabel` permet de modifier la taille des étiquettes.

```
R> s.corcircle(acm$co, 1, 2, clabel = 0.7)
```

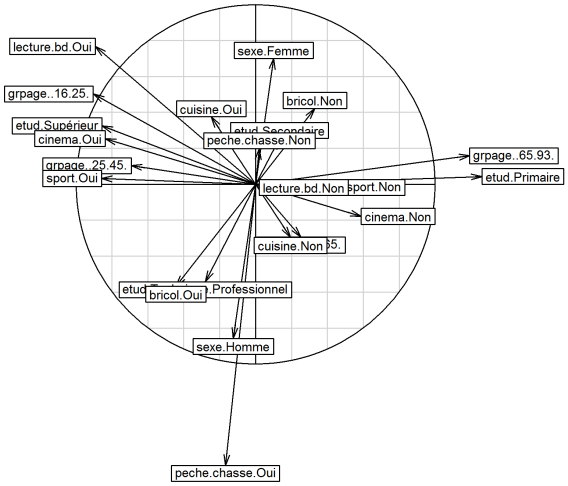



*Figure 2. Cercle de corrélations des modalités sur les deux premiers axes*

On pourra avoir également recours à `boxplot` pour visualiser comment se répartissent les modalités de chaque variable sur un axe donné[3].

```
R> boxplot(acm)
```

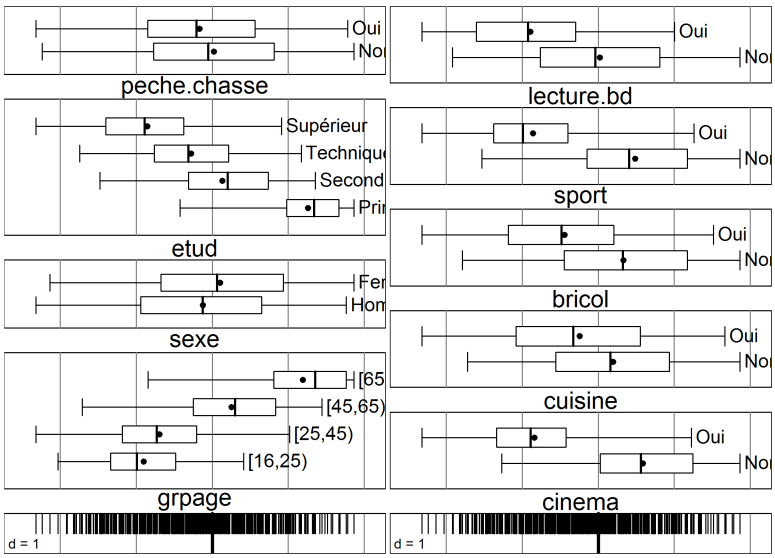



*Figure 3. Répartition des modalités selon le premier axe*

```
R> boxplot(acm, 2)
```

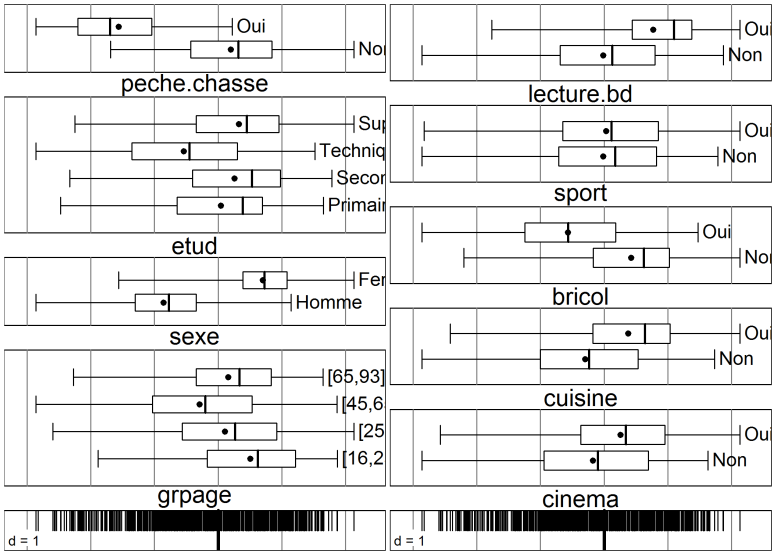
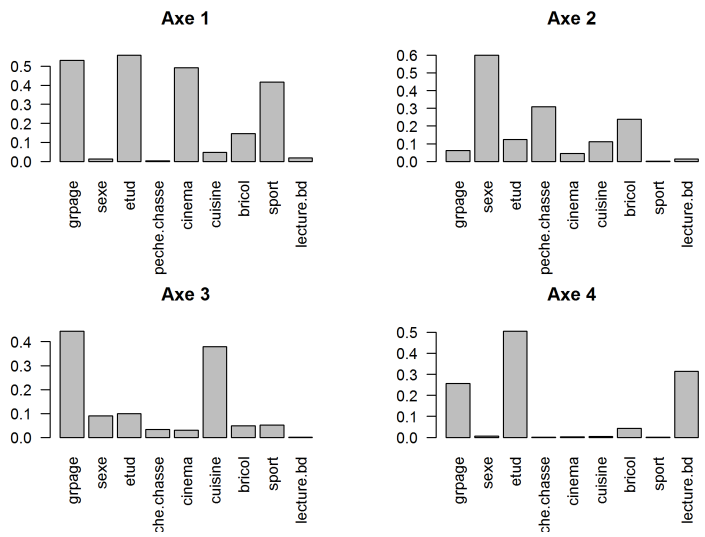



*Figure 4. Répartition des modalités selon le second axe*

Le tableau `acm$cr` contient les rapports de corrélation (variant de 0 à 1) entre les variables et les axes choisis au départ de l'ACM. Pour représenter graphiquement ces rapports, utiliser la fonction `barplot` ainsi : `barplot(acm$cr[,num],names.arg=row.names( acm$cr),las=2)` où `num` est le numéro de l'axe à représenter. Pour l'interprétation des axes, se concentrer sur les variables les plus structurantes, c'est-à-dire dont le rapport de corrélation est le plus proche de 1.

```
R> par(mfrow = c(2, 2))
   for (i in 1:4) barplot(acm$cr[, i], names.arg =
   row.names(acm$cr),
     las = 2, main = paste("Axe", i))
```




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

*Figure 5. Rapports de corrélation des variables sur les 4 premiers axes*

**ASTUCE**

Le paramètre `mfrow` de la fonction `par` permet d'indiquer à **R** que l'on souhaite afficher plusieurs graphiques sur une seule et même fenêtre, plus précisément que l'on souhaite diviser la fenêtre en deux lignes et deux colonnes.

Dans l'exemple précédent, après avoir produit notre graphique, nous avons réinitilisé cette valeur à `c(1, 1)` (un seul graphique par fenêtre) pour ne pas affecter les prochains graphiques que nous allons produire.

Pour représenter, les modalités dans le plan factoriel, on utilisera la fonction `s.label`. Par défaut, les deux premiers axes sont représentés.

```
R> s.label(acm$co, clabel = 0.7)
```

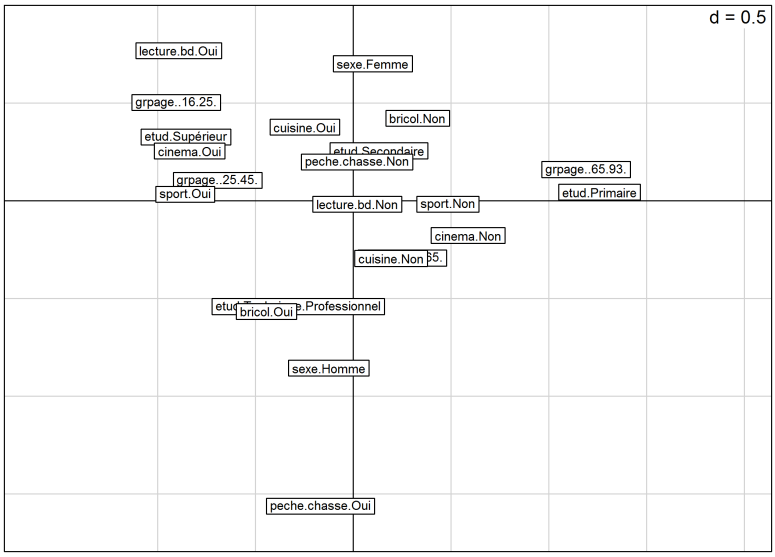



*Figure 6. Répartition des modalités selon les deux premiers axes*

Il est bien sur possible de préciser les axes à représenter. L'argument `boxes` permet quant à lui d'indiquer si l'on souhaite tracer une boîte pour chaque modalité.

```
R> s.label(acm$co, 3, 4, clabel = 0.7, boxes = FALSE)
```

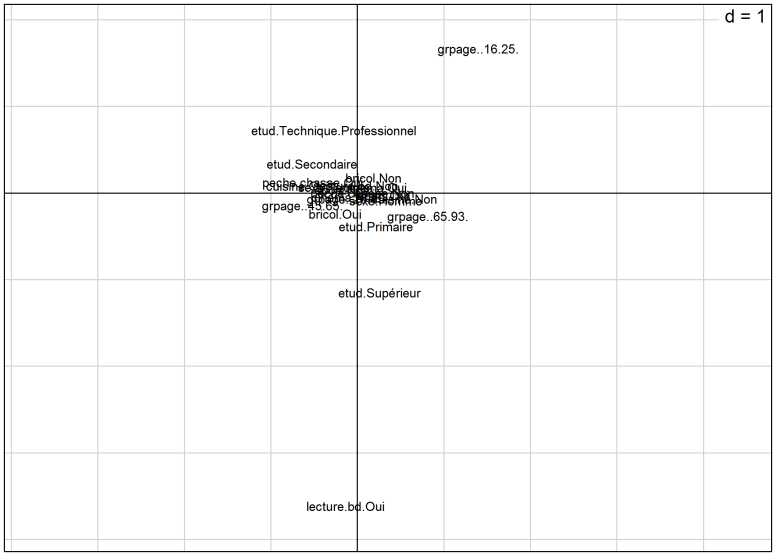



*Figure 7. Répartition des modalités selon les axes 3 et 4*

Bien entendu, on peut également représenter les individus. En indiquant `clabel=0` (une taille nulle pour les étiquettes), `s.label` remplace chaque observation par un symbole qui peut être spécifié avec `pch`[4].

```
R> s.label(acm$li, clabel = 0, pch = 17)
```

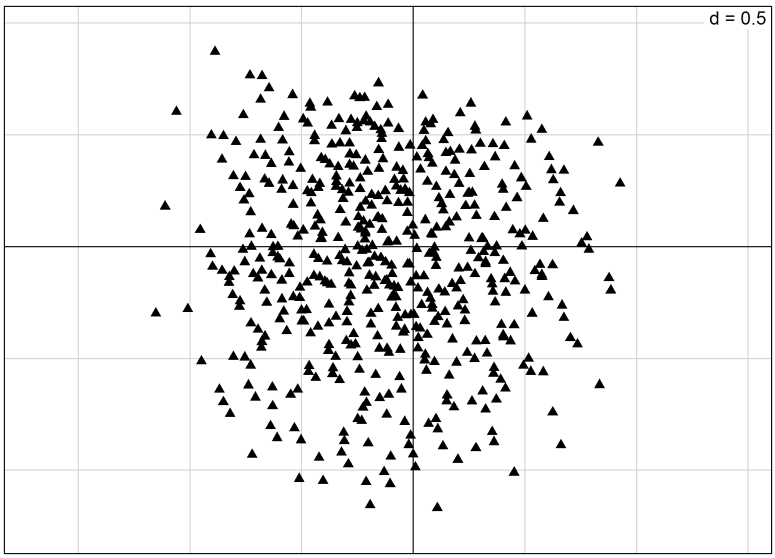



*Figure 8. Répartition des individus selon les deux premiers axes*

**ASTUCE**

Lorsque l'on réalise une ACM, il n'est pas rare que plusieurs observations soient identiques, c'est-à-dire correspondent à la même combinaison de modalités. Dès lors, ces observations seront projetées sur le même point dans le plan factoriel. Une représentation classique des observations avec `s.label` ne permettra pas de rendre compte les effectifs de chaque point.

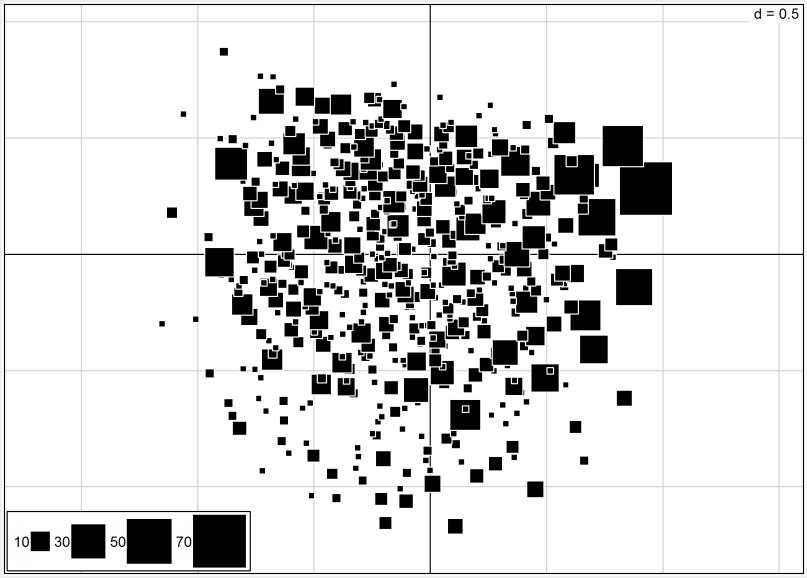
Le package **JLutils**, disponible seulement sur GitHub, propose une petite fonction `s.freq` représentant chaque point par un carré proportionnel au nombre d'individus.

Pour installer **JLutils**, on aura recours au package **devtools** et à sa fonction `install_github` :

```
R> library(devtools)
   install_github("larmarange/JLutils")
```

La fonction `s.freq` s'emploie de manière similaire aux autres fonctions graphiques de **ade4**. Le paramètre `csize` permet d'ajuster la taille des carrés.

```
R> library(JLutils)
   s.freq(acm$li)
```




L'interprétation est tout autre, non ?

**ASTUCE**

Gaston Sanchez propose un graphique amélioré des modalités dans le plan factoriel à cette adresse : http://rpubs.com/gaston/MCA.

La fonction `s.value` permet notamment de représenter un troisième axe factoriel. Dans l'exemple ci-après, nous projettons les individus selon les deux premiers axes factoriels. La taille et la couleur des carrés dépendent pour leur part de la coordonnée des individus sur le troisième axe factoriel. Le paramètre `csi` permet d'ajuster la taille des carrés.

```
R> s.value(acm$li, acm$li[, 3], 1, 2, csi = 0.5)
```

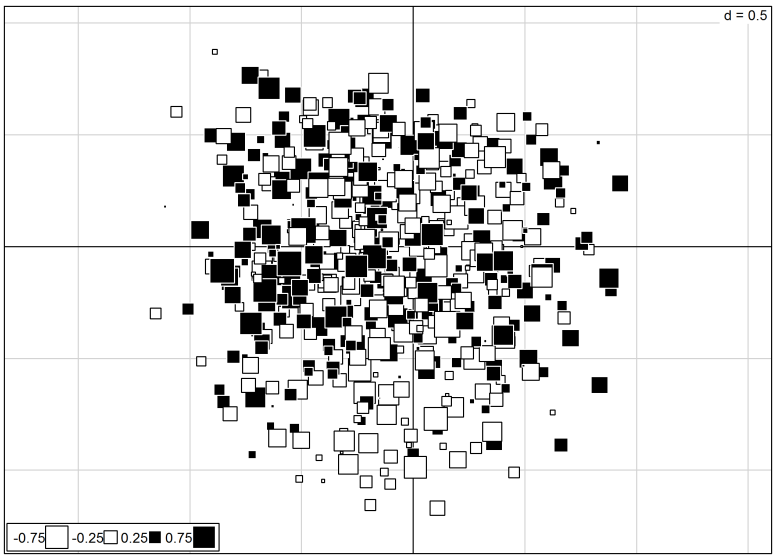



*Figure 9. Répartition des individus selon les trois premiers axes*

`s.arrow` permet de représenter les vecteurs variables ou les vecteurs individus sous la forme d'une flèche allant de l'origine du plan factoriel aux coordonnées des variables/individus :

```
R> s.arrow(acm$co, clabel = 0.7)
```

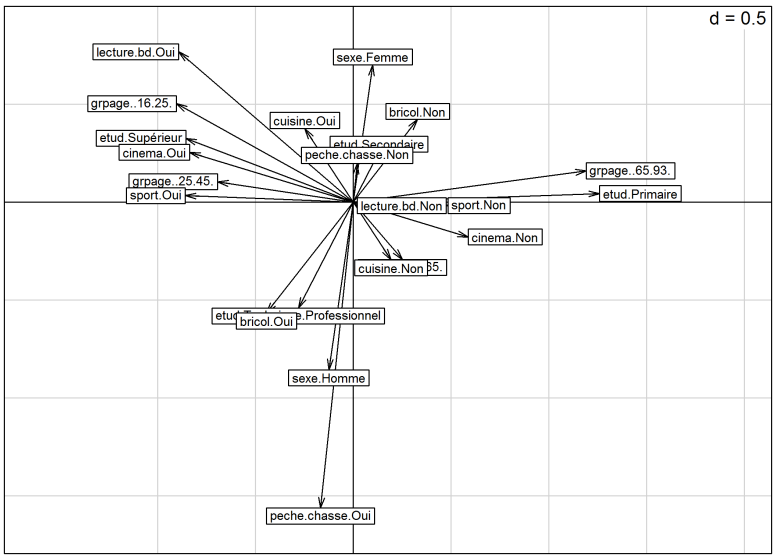



*Figure 10. Vecteurs des modalités selon les deux premiers axes*

`s.hist` permet de représenter des individus (ou des modalités) sur le plan factoriel et d'afficher leur distribution sur chaque axe :

```
R> s.hist(acm$li, clabel = 0, pch = 15)
```

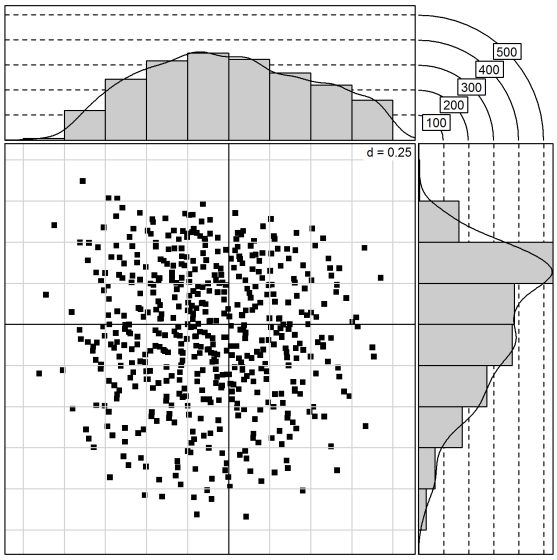



*Figure 11. Distribution des individus dans le plan factoriel*

`s.class` et `s.chull` permettent de représenter les différentes observations classées en plusieurs catégories. Cela permet notamment de projeter certaines variables.

`s.class` représente les observations par des points, lie chaque observation au barycentre de la modalité à laquelle elle appartient et dessine une ellipse représentant la forme générale du nuage de points :

```
R> library(RColorBrewer)
   s.class(acm$li, d2$sexe, col = brewer.pal(4,
   "Set1"))
```

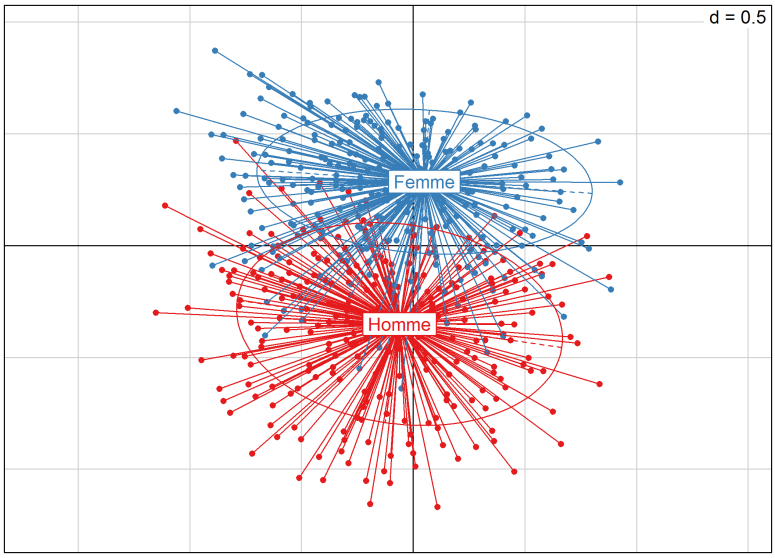



*Figure 12. Individus dans le plan factoriel selon le sexe (s.class)*

`s.chull` représente les barycentres de chaque catégorie et dessine des lignes de niveaux représentant la distribution des individus de cette catégorie. Les individus ne sont pas directement représentés :

```
R> s.chull(acm$li, d2$sexe, col = brewer.pal(4,
   "Set1"))
```

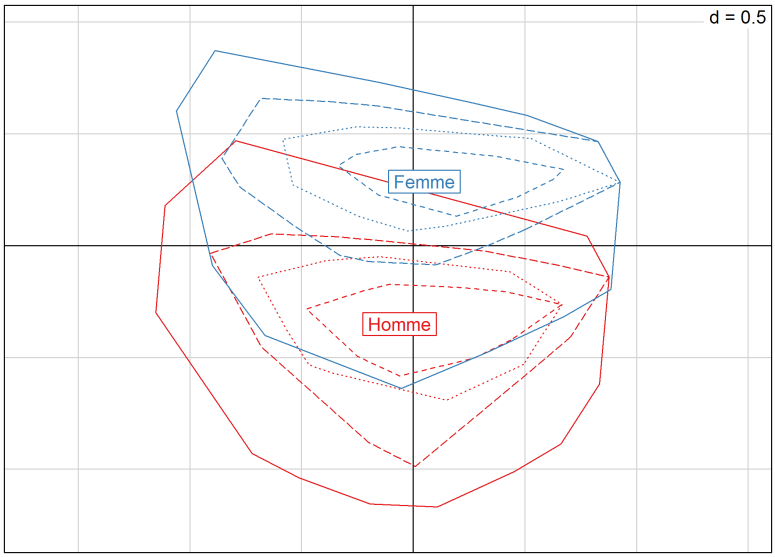
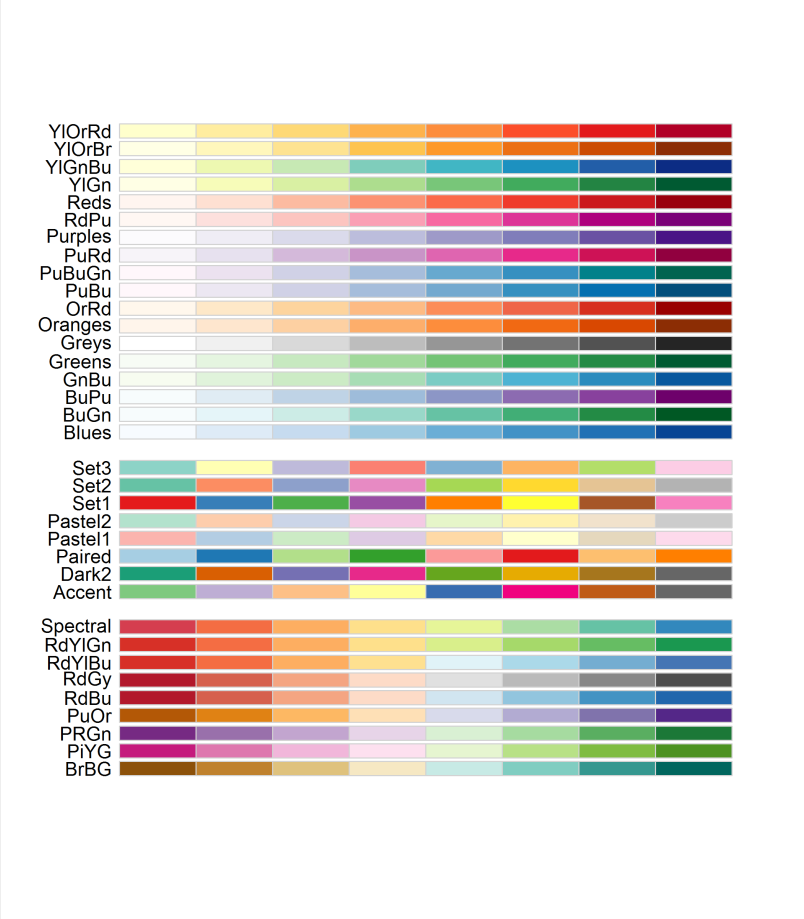



*Figure 13. Individus dans le plan factoriel selon le sexe (s.chull)*

**ASTUCE**

Il est préférable de fournir une liste de couleurs (via le paramètre `col`) pour rendre le graphique plus lisible. Si vous avez installé l'extension **RColorBrewer**, vous pouvez utiliser les différentes palettes de couleurs proposées. Pour afficher les palettes disponibles, utilisez `display.brewer.all`.

```
R> library(RColorBrewer)
   display.brewer.all(8)
```




Pour obtenir une palette de couleurs, utilisez la fonction `brewer.pal` avec les arguments `n` (nombre de couleurs demandées) et `pal` (nom de la palette de couleurs désirée).

Pour plus d'informations sur les palettes *Color Brewer*, voir http://colorbrewer2.org/.

La variable catégorielle transmise à `s.class` ou `s.chull` n'est pas obligatoirement une des variables retenues pour l'ACM. Il est tout à fait possible d'utiliser une autre variable. Par exemple :

```
R> s.class(acm$li, d$trav.imp, col = brewer.pal(4,
   "Set1"))
```

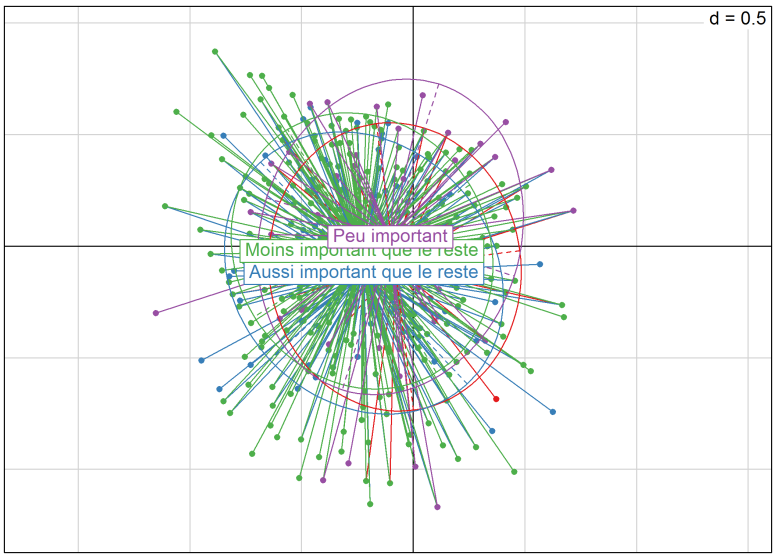



*Figure 14. Individus dans le plan factoriel selon l'importance donnée au travail*

Les fonctions `scatter` et `biplot` sont équivalentes : elles appliquent `s.class` à chaque variable utilisée pour l'ACM.

```
R> scatter(acm, col = brewer.pal(4, "Set1"))
```

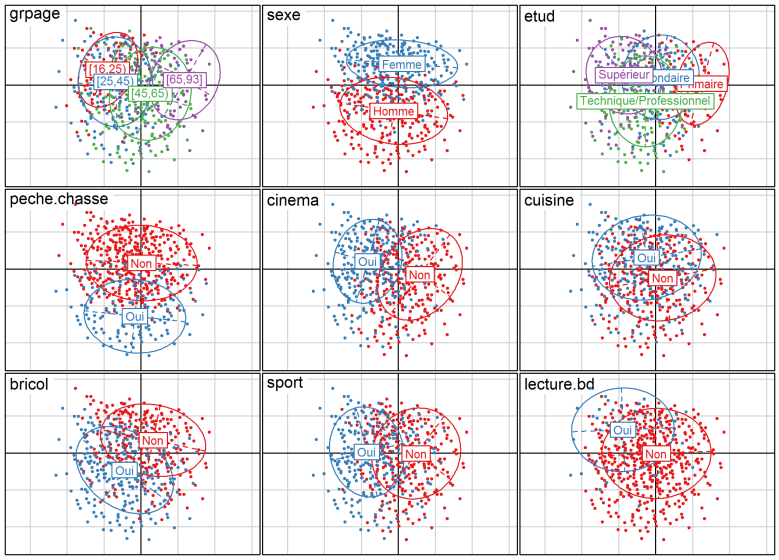



*Figure 15. La fonction scatter appliquée au résultat d'une ACM*

# ACM avec FactoMineR

Comme avec **ade4**, il est nécessaire de préparer les données au préalable (voir section précédente).

L'ACM se calcule avec la fonction `MCA`, l'argument `ncp` permettant de choisir le nombre d'axes à retenir :

```
R> library(FactoMineR)
```

```
R> acm2 <- MCA(d2, ncp = 5, graph = FALSE)
   acm2
```

```
**Results of the Multiple Correspondence Analysis (MCA)**
The analysis was performed on 2000 individuals, described
by 9 variables
*The results are available in the following objects:

   name
1  "$eig"
2  "$var"
3  "$var$coord"
4  "$var$cos2"
5  "$var$contrib"
6  "$var$v.test"
7  "$ind"
8  "$ind$coord"
9  "$ind$cos2"
10 "$ind$contrib"
11 "$call"
12 "$call$marge.col"
13 "$call$marge.li"
   description
1  "eigenvalues"
2  "results for the variables"
3  "coord. of the categories"
4  "cos2 for the categories"
5  "contributions of the categories"
6  "v-test for the categories"
7  "results for the individuals"
8  "coord. for the individuals"
9  "cos2 for the individuals"
10 "contributions of the individuals"
11 "intermediate results"
12 "weights of columns"
13 "weights of rows"
```

```
R> acm2$eig
```

```
       eigenvalue percentage of variance
dim 1  0.25757489              15.454493
dim 2  0.18363502              11.018101
dim 3  0.16164626               9.698776
dim 4  0.12871623               7.722974
dim 5  0.12135737               7.281442
dim 6  0.11213331               6.727999
dim 7  0.10959377               6.575626
dim 8  0.10340564               6.204338
dim 9  0.09867478               5.920487
dim 10 0.09192693               5.515616
dim 11 0.07501208               4.500725
dim 12 0.06679676               4.007805
dim 13 0.06002063               3.601238
dim 14 0.05832024               3.499215
dim 15 0.03785276               2.271166
       cumulative percentage of variance
dim 1                           15.45449
dim 2                           26.47259
dim 3                           36.17137
dim 4                           43.89434
dim 5                           51.17579
dim 6                           57.90378
dim 7                           64.47941
dim 8                           70.68375
dim 9                           76.60424
dim 10                          82.11985
dim 11                          86.62058
dim 12                          90.62838
dim 13                          94.22962
dim 14                          97.72883
dim 15                         100.00000
```

```
R> sum(acm2$eig$eigenvalue)
```

```
[1] 1.666667
```

En premier lieu, il apparait que l'inertie totale obtenue avec `MCA` est différente de celle observée avec `dudi.acm`{data-package="ade4">. Cela est dû à un traitement différents des valeurs manquantes. Alors que `dudi.acm` exclu les

valeurs manquantes, `MCA` les considèrent, par défaut, comme une modalité additionnelle. Pour calculer l'ACM uniquement sur les individus n'ayant pas de valeur manquante, on aura recours à `complete.cases` :

```
R> acm2 <- MCA(d2[complete.cases(d2), ], ncp = 5, graph =
   FALSE)
   acm2$eig
```

```
       eigenvalue percentage of variance
dim 1  0.24790700              17.162792
dim 2  0.16758465              11.602014
dim 3  0.13042357               9.029324
dim 4  0.12595105               8.719688
dim 5  0.11338629               7.849820
dim 6  0.10976674               7.599236
dim 7  0.10060204               6.964757
dim 8  0.09802387               6.786268
dim 9  0.09283131               6.426783
dim 10 0.07673502               5.312425
dim 11 0.06609694               4.575942
dim 12 0.05950655               4.119684
dim 13 0.05562942               3.851267
       cumulative percentage of variance
dim 1                           17.16279
dim 2                           28.76481
dim 3                           37.79413
dim 4                           46.51382
dim 5                           54.36364
dim 6                           61.96287
dim 7                           68.92763
dim 8                           75.71390
dim 9                           82.14068
dim 10                          87.45311
dim 11                          92.02905
dim 12                          96.14873
dim 13                         100.00000
```

```
R> sum(acm2$eig$eigenvalue)
```

```
[1] 1.444444
```

Les possibilités graphiques de **FactoMineR** sont différentes de celles de **ade4**. Un recours à la fonction `plot` affichera par défaut les individus, les modalités et les variables. La commande `?plot.MCA` permet d'accéder au fichier d'aide de cette fonction (i.e. de la méthode générique `plot` appliquée aux objets de type `MCA`) et de voir toutes les options graphiques. L'argument `choix` permet de spécifier ce que l'on souhaite afficher (« ind » pour les individus et les catégories, « var » pour les variables). L'argument `invisible` quant à lui permet de spécifier ce que l'on souhaite masquer. Les axes à afficher se précisent avec `axes`. Voir les exemples ci-dessous.

```
R> plot(acm2)
```

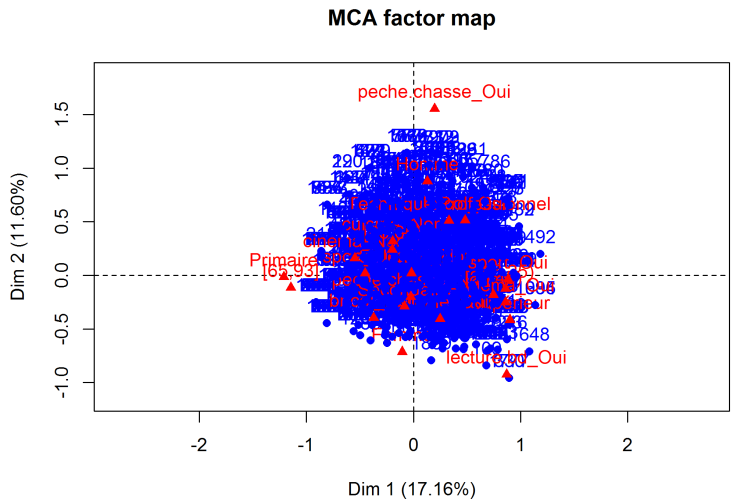



*Figure 16. Plan factoriel (deux premiers axes)*

```
R> plot(acm2, axes = c(3, 4))
```

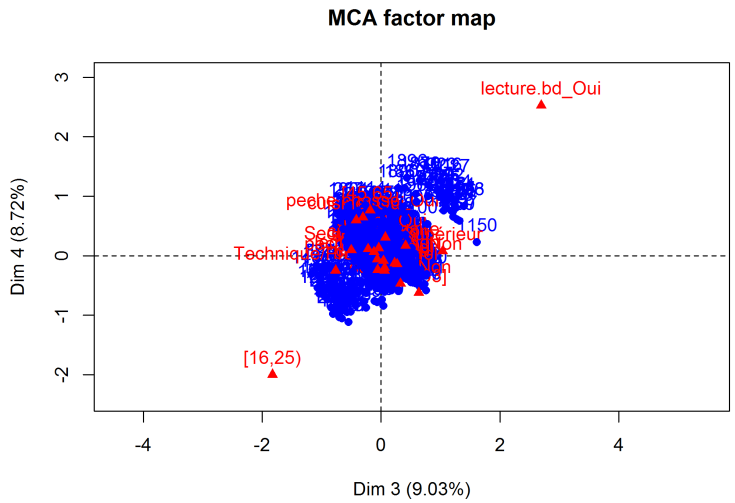



*Figure 17. Plan factoriel (axes 3 et 4)*

```
R> plot(acm2, choix = "ind")
```

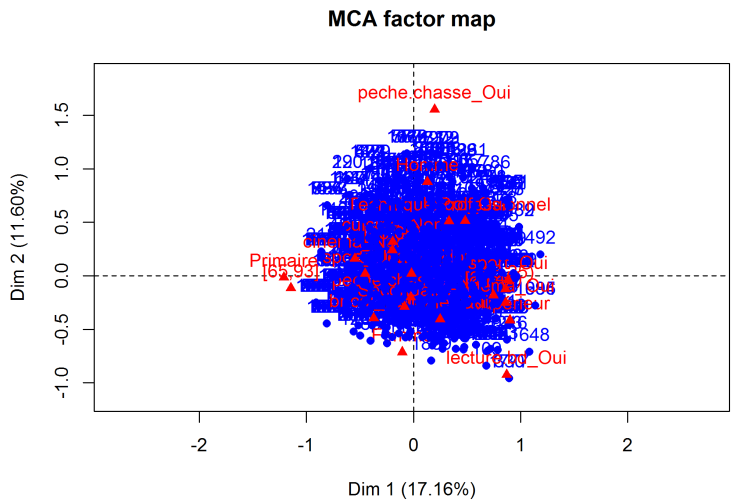



*Figure 18. Plan factoriel (seulement les individus et les catégories)*

```
R> plot(acm2, choix = "ind", invisible = "ind")
```

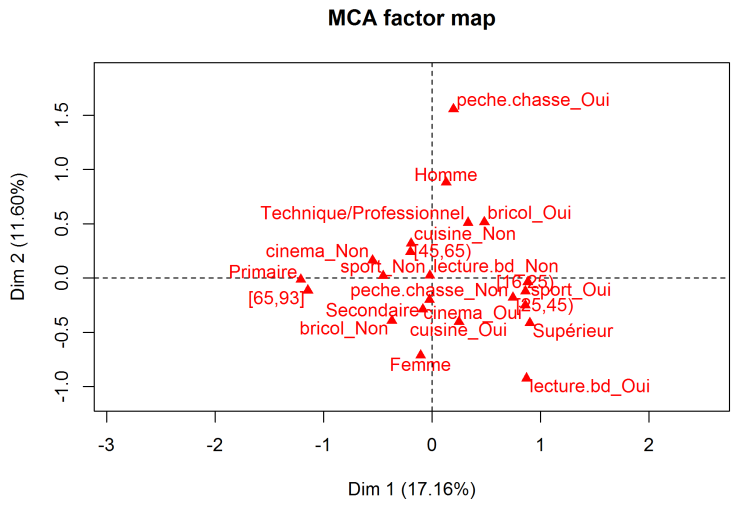



*Figure 19. Plan factoriel (seulement les catégories)*

```
R> plot(acm2, choix = "var")
```

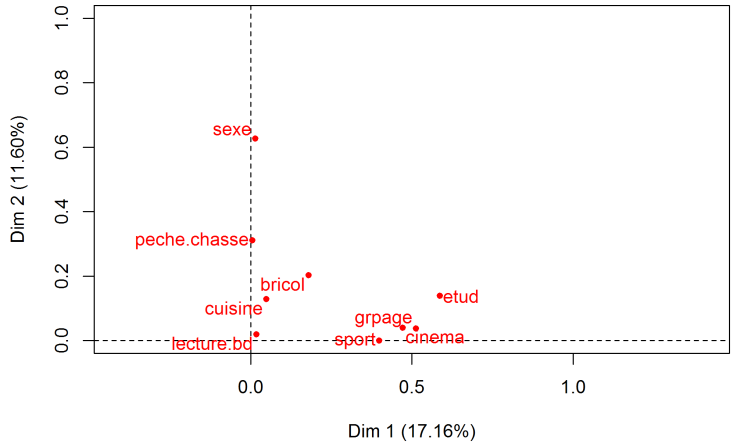



*Figure 20. Plan factoriel (seulement les variables)*

La fonction `plotellipses` trace des ellipses de confiance atour des modalités de variables qualitatives. L'objectif est de voir si les modalités d'une variable qualitative sont significativement différentes les unes des autres.

Par défaut ( `means=TRUE` ), les ellipses de confiance sont calculées pour les coordonnées moyennes de chaque catégorie.

```
R> plotellipses(acm2)
```

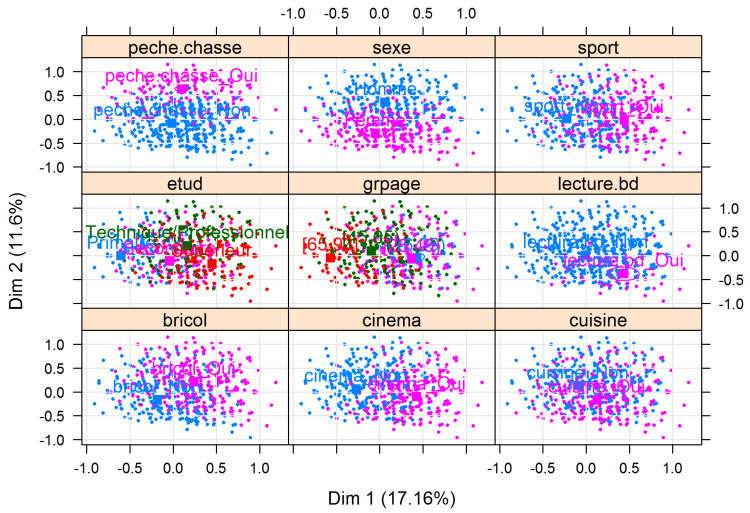



*Figure 21. Ellipses de confiance (means=TRUE) dans le plan factoriel*

L'option `means=FALSE` calculera les ellipses de confiance pour l'ensemble des coordonnées des observations relevant de chaque catégorie.

```
R> plotellipses(acm2, means = FALSE)
```

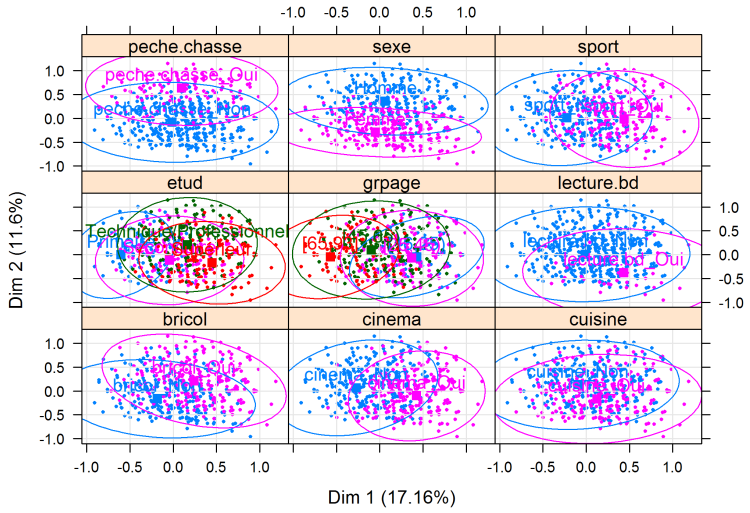



*Figure 22. Ellipses de confiance (means=FALSE) dans le plan factoriel*

La fonction `dimdesc` aide à décrire et interpréter les dimensions de l'ACM. Cette fonction est très utile quand le nombre de variables est élevé. Elle permet de voir à quelles variables les axes sont le plus liés : quelles variables et quelles modalités décrivent le mieux chaque axe ?

> *Pour les variables qualitatives, un modèle d'analyse de variance à un facteur est réalisé pour chaque dimension ; les variables à expliquer sont les coordonnées des individus et la variable explicative est une des variables qualitatives. Un test F permet de voir si la variable a un effet significatif sur la dimension et des tests T sont réalisés modalité par modalité (avec le contraste somme des alpha_i=0). Cela montre si les coordonnées des individus de la sous-population définie par une modalité sont significativement différentes de celles de l'ensemble de la population (i.e. différentes de 0). Les variables et modalités sont triées par probabilité critique et seules celles qui sont significatives sont gardées dans le résultat.*

> — Source : http://factominer.free.fr/factosbest/description-des-dimensions.html

```
R> dimdesc(acm2, axes = 1:2)
```

```
$`Dim 1`
$`Dim 1`$quali
                      R2       p.value
etud         0.586058164  0.000000e+00
grpage       0.512231318 1.008479e-292
cinema       0.471002163 8.339548e-263
sport        0.398103140 5.940105e-210
bricol       0.179188677  7.142716e-83
cuisine      0.048233515  4.941749e-22
lecture.bd   0.017667936  6.856650e-09
sexe         0.013670717  3.546801e-07
peche.chasse 0.005007337  2.105728e-03

$`Dim 1`$category
                          Estimate       p.value
cinema_Oui               0.35033716 8.339548e-263
sport_Oui                0.33199860 5.940105e-210
[25,45)                  0.33895697 1.959716e-159
Supérieur                0.45630756 1.376719e-118
bricol_Oui               0.21255190  7.142716e-83
Technique/Professionnel  0.17291856  4.703868e-23
cuisine_Oui              0.11015793  4.941749e-22
[16,25)                  0.39553122  3.635181e-15
lecture.bd_Oui           0.22169306  6.856650e-09
Homme                    0.05853263  3.546801e-07
peche.chasse_Oui         0.05543091  2.105728e-03
peche.chasse_Non        -0.05543091  2.105728e-03
Femme                   -0.05853263  3.546801e-07
lecture.bd_Non          -0.22169306  6.856650e-09
[45,65)                 -0.13173232  2.477610e-12
cuisine_Non             -0.11015793  4.941749e-22
bricol_Non              -0.21255190  7.142716e-83
[65,93]                 -0.60275587 2.906563e-165
sport_Non               -0.33199860 5.940105e-210
cinema_Non              -0.35033716 8.339548e-263
Primaire                -0.59465967 5.269497e-268


$`Dim 2`
$`Dim 2`$quali
```

```
                     R2       p.value
sexe         0.62723828  0.000000e+00
peche.chasse 0.31109226 1.161746e-154
bricol       0.20276579  8.014713e-95
etud         0.13925513  6.754592e-61
cuisine      0.12908453  1.461380e-58
cinema       0.04039994  1.215838e-18
grpage       0.03776900  1.257795e-15
lecture.bd   0.01995653  7.190474e-10

$`Dim 2`$category
                           Estimate       p.value
Homme                    0.32598031  0.000000e+00
peche.chasse_Oui         0.35922450 1.161746e-154
bricol_Oui               0.18590016  8.014713e-95
cuisine_Non              0.14816688  1.461380e-58
Technique/Professionnel  0.23013181  5.919911e-54
cinema_Non               0.08436024  1.215838e-18
[45,65)                  0.11638978  3.028836e-17
lecture.bd_Non           0.19371997  7.190474e-10
[65,93]                 -0.02872480  1.229757e-02
[25,45)                 -0.05570792  2.294799e-09
lecture.bd_Oui          -0.19371997  7.190474e-10
Secondaire              -0.09715846  1.240732e-10
cinema_Oui              -0.08436024  1.215838e-18
Supérieur               -0.14813997  6.942611e-24
cuisine_Oui             -0.14816688  1.461380e-58
bricol_Non              -0.18590016  8.014713e-95
peche.chasse_Non        -0.35922450 1.161746e-154
Femme                   -0.32598031  0.000000e+00
```

1. Pour une présentation plus détaillée, voir http://www.math.univ-toulouse.fr/~baccini/zpedago/asdm.pdf.↩
2. On pourra également avoir recours à la fonction `inertia.dudi` pour l'ensemble des axes.↩
3. La fonction `score` constituera également une aide à l'interprétation des axes.↩

4. Voir le chapitre sur les graphiques pour la liste des différents symboles utilisables. (MAJ LIEN)↩

# Classification ascendante hiérarchique (CAH)

**NOTE**

La version originale de ce chapitre a été écrite par Joseph Larmarange dans le cadre du support de cours Introduction à l'analyse d'enquêtes avec R.

Il existe de nombreuses techniques statistiques visant à partinionner une population en différentes classes ou sous-groupes. La *classification ascendante hiérarchique* (CAH) est l'une d'entre elles. On cherche à ce que les individus regroupés au sein d'une même classe (homogénéité intra-classe) soient le plus semblables possibles tandis que les classes soient le plus dissemblables (hétérogénéité inter-classe).

Le principe de la CAH est de rassembler des individus selon un critère de ressemblance défini au préalable qui s'exprimera sous la forme d'une *matrice de distances*, exprimant la distance existant entre chaque individu pris deux à deux. Deux observations identiques auront une distance nulle. Plus les deux observations seront dissemblables, plus la distance sera importante. La CAH va ensuite rassembler les individus de manière itérative afin de produire un *dendrogramme* ou *arbre de classification*. La classification est *ascendante* car elle part des observations individuelles ; elle est *hiérarchique* car elle produit des classes ou groupes de plus en plus vastes, incluant des sous-groupes en leur sein.

En découpant cet arbre à une certaine hauteur choisie, on produira la partition désirée.

> **INFO**
>
> On trouvera également de nombreux supports de cours en français sur la CAH sur le site de François Gilles Carpentier : http://geai.univ-brest.fr/~carpenti/.

# Calculer une matrice des distances

La notion de *ressemblance* entre observations est évaluée par une distance entre individus. Plusieurs type de ditances existent selon les données utilisées.

Il existe de nombreuses distances mathématiques pour les variables quantitatives (euclidiennes, Manhattan…) que nous n'aborderons pas ici[1]. La plupart peuvent être calculées avec la fonction `dist`.

Usuellement, pour un ensemble de variables qualitatives, on aura recours à la distance du $\Phi^2$ qui est celle utilisée pour l'analyse des correspondances multiples (voir le chapitre dédié). Avec l'extension **ade4**, la distance du $\Phi^2$ s'obtient avec la fonction `dist.dudi`[2]. Le cas particulier de la CAH avec l'extension **FactoMineR** sera abordée dans une section spécifique (MAJ LIEN). Nous évoquerons également la *distance de Gower* qui peut s'appliquer à un ensemble de variables à la fois qualitatives et quantitatives et qui se calcule avec la fonction `daisy` de l'extension **cluster**. Enfin, dans le chapitre sur l'analyse de séquences (MAJ LIEN), nous verrons également la fonction `seqdist` (extension **TraMineR**) permettant de calculer une distance entre séquences.

## Distance de Gower

En 1971, Gower a proposé un indice de similarité qui porte son nom. L'objectif de cet indice consiste à mesurer dans quelle mesure deux individus sont semblables. L'indice de Gower varie entre 0 et 1. Si l'indice vaut 1, les deux individus sont identiques. À l'opposé, s'il vaut 0, les deux individus considérés n'ont pas de point commun. Si l'on note $S_g$ l'indice de similarité de Gower, la

distance de Gower $D_g$ s'obtient simplement de la manière suivante : $D_g = 1\text{-}S_g$. Ainsi, la distance sera nulle entre deux individus identiques et elle sera égale à 1 entre deux individus totalement différents. Cette distance s'obtient sous **R** avec la fonction `daisy` du package **cluster**.

L'indice de similarité de Gower entre deux individus $x_1$ et $x_2$ se calcule de la manière suivante :

$$S_g(x_1, x_2) = \frac{1}{p}\sum_{j=1}^{p} s_{12j}$$

$p$ représente le nombre total de caractères (ou de variables) descriptifs utilisés pour comparer les deux individus. $s_{12j}$ représente la similarité partielle entre les individus 1 et 2 concernant le descripteur $j$. Cette similarité partielle se calcule différemment s'il s'agit d'une variable qualitative ou quantitative :

- **variable qualitative :** $s_{12j}$ vaut 1 si la variable $j$ prend la même valeur pour les individus 1 et 2, et vaut 0 sinon. Par exemple, si 1 et 2 sont tous les deux « grand », alors $s_{12j}$ vaudra 1. Si 1 est « grand » et 2 « petit », $s_{12j}$ vaudra 0.
- **variable quantitative :** la différence absolue entre les valeurs des deux variables est tout d'abord calculée, soit $|y_{1j} - y_{2j}|$. Puis l'écart maximum observé sur l'ensemble du fichier est déterminé et noté $R_j$. Dès lors, la similarité partielle vaut $S_{12j} = |y_{1j} - y_{2j}| / R_j$.

Dans le cas où l'on n'a que des variables qualitatives, la valeur de l'indice de Gower correspond à la proportion de caractères en commun. Supposons des individus 1 et 2 décris ainsi :

1. homme / grand / blond / étudiant / urbain
2. femme / grande / brune / étudiante / rurale

Sur les 5 variables utilisées pour les décrire, 1 et 2 ont deux caractéristiques communes : ils sont grand(e)s et étudiant(e)s. Dès lors, l'indice de similarité de Gower entre 1 et 2 vaut 2/5 = 0,4 (soit une distance de 1 - 0,4 = 0,6).

Plusieurs approches peuvent être retenues pour traiter les valeurs manquantes :

- supprimer tout individu n'étant pas renseigné pour toutes les variables de l'analyse ;
- considérer les valeurs manquantes comme une modalité en tant que telle ;

- garder les valeurs manquantes en tant que valeurs manquantes.

Le choix retenu modifiera les distances de Gower calculées. Supposons que l'on ait :

1. homme / grand / blond / étudiant / urbain
2. femme / grande / brune / étudiante / manquant

Si l'on supprime individus ayant des valeurs manquantes, 2 est retirée du fichier d'observations et aucune distance n'est calculée.

Si l'on traite les valeurs manquantes comme une modalité particulière, 1 et 2 partagent alors 2 caractères sur les 5 analysés, la distance de Gower entre eux est alors de 1 - 2/5 =1 - 0,4 = 0,6.

Si on garde les valeurs manquantes, l'indice de Gower est dès lors calculé sur les seuls descripteurs renseignés à la fois pour 1 et 2. La distance de Gower sera calculée dans le cas présent uniquement sur les 4 caractères renseignés et vaudra 1 - 2/4 = 0,5.

## Distance du $\Phi^2$

Il s'agit de la distance utilisée dans les analyses de correspondance multiples (ACM). C'est une variante de la distance du $\chi^2$. Nous considérons ici que nous avons $Q$ questions (soit $Q$ variables initiales de type facteur). À chaque individu est associé un *patron* c'est-à-dire une certaine combinaison de réponses aux $Q$ questions. La distance entre deux individus correspond à la distance entre leurs deux patrons. Si les deux individus présentent le même patron, leur distance sera nulle. La distance du $\Phi^2$ peut s'exprimer ainsi :

$$d^2_{\Phi^2}(L_i, L_j) = \frac{1}{Q}\sum_k \frac{(\delta_{ik} - \delta_{jk})^2}{f_k}$$

où $L_i$ et $L_j$ sont deux patrons, $Q$ le nombre total de questions. $\delta_{ik}$ vaut 1 si la modalité $k$ est présente dans le patron $L_i$, 0 sinon. $f_k$ est la fréquence de la modalité $k$ dans l'ensemble de la population.

Exprimé plus simplement, on fait la somme de l'inverse des modalités non communes aux deux patrons, puis on divise par le nombre total de question. Si nous reprenons notre exemple précédent :

1. homme / grand / blond / étudiant / urbain
2. femme / grande / brune / étudiante / rurale

Pour calculer la distance entre 1 et 2, il nous faut connaître la proportion des différentes modalités dans l'ensemble de la population étudiée. En l'occurrence :

- hommes : 52 % / femmes : 48 %
- grand : 30 % / moyen : 45 % / petit : 25 %
- blond : 15 % / châtain : 45 % / brun : 30 % / blanc : 10 %
- étudiant : 20 % / salariés : 65 % / retraités : 15 %
- urbain : 80 % / rural : 20 %

Les modalités non communes entre les profils de 1 et 2 sont : homme, femme, blond, brun, urbain et rural. La distance du $\Phi^2$ entre 1 et 2 est donc la suivante :

$$d^2_{\Phi^2}(L_1, L_2) = \frac{1}{5}\left(\frac{1}{0,52} + \frac{1}{0,48} + \frac{1}{0,15} + \frac{1}{0,30} + \frac{1}{0,80} + \frac{1}{0,20}\right) = 4,05$$

Cette distance, bien que moins intuitive que la distance de Gower évoquée précédemment, est la plus employée pour l'analyse d'enquêtes en sciences sociales. Il faut retenir que la distance entre deux profils est dépendante de la distribution globale de chaque modalité dans la population étudiée. Ainsi, si l'on recalcule les distances entre individus à partir d'un sous-échantillon, le résultat obtenu sera différent. De manière générale, les individus présentant des caractéristiques rares dans la population vont se retrouver éloignés des individus présentant des caractéristiques fortement représentées.

## Exemple

Nous allons reprendre l'ACM calculée avec `dudi.acm` (**ade4**) dans le chapitre consacré à l'ACM :

```
R> library(questionr)
   data(hdv2003)
   d <- hdv2003
   d$grpage <- cut(d$age, c(16, 25, 45, 65, 93), right =
   FALSE,
     include.lowest = TRUE)
   d$etud <- d$nivetud
   levels(d$etud) <- c("Primaire", "Primaire", "Primaire",
     "Secondaire", "Secondaire", "Technique/Professionnel",
     "Technique/Professionnel", "Supérieur")
   d2 <- d[, c("grpage", "sexe", "etud", "peche.chasse",
     "cinema", "cuisine", "bricol", "sport", "lecture.bd")]
   library(ade4)
   acm <- dudi.acm(d2, scannf = FALSE, nf = 5)
```

La matrice des distances s'obtient dès lors avec la fonction `dist.dudi` :

```
R> md <- dist.dudi(acm)
```

# Calcul du dendrogramme

Il faut ensuite choisir une méthode d'agrégation pour construire le dendrogramme. De nombreuses solutions existent (saut minimum, distance maximum, moyenne, Ward…). Chacune d'elle produira un dendrogramme différent. Nous ne détaillerons pas ici ces différentes techniques[3]. Cependant, à l'usage, on privilégiera le plus souvent la *méthode de Ward*. Cette méthode se distingue de toutes les autres en ce sens qu'elle utilise une analyse de la variance approchée afin d'évaluer les distances entre groupes. La méthode de Ward se justifie bien lorsque lorsque l'on utilise le carré de la distance. Choisir de regrouper les deux individus les plus proches revient alors à choisir la paire de points dont l'agrégation entraîne la diminution minimale de l'inertie du nuage. En résumé, cette méthode cherche à minimiser l'inertie intra-classe et à maximiser l'inertie inter-classe afin d'obtenir des classes les plus homogènes possibles.

En raison de la variété des distances possibles et de la variété des techniques d'agrégation, on pourra être amené à réaliser plusieurs dendrogrammes différents sur un même jeu de données jusqu'à obtenir une classification qui fait « sens ».

La fonction de base pour le calcul d'un dendrogramme est `hclust` en précisant le critère d'aggrégation avec `method`. Dans notre cas, nous allons opter pour la méthode de Ward appliquée au carré des distances (ce qu'on indique avec `md^2`) :

```
R> arbre <- hclust(md^2, method = "ward")
```

```
The "ward" method has been renamed to "ward.D"; note new
"ward.D2"
```

**ASTUCE**

Le temps de calcul d'un dendrogramme peut être particulièrement important sur un gros fichier de données. L'extension **flashClust** permet de réduire significativement le temps de calcul. Il suffit d'installer puis d'appeler cette extension. La fonction `hclust` sera automatiquement remplacée par cette version optimisée :

```
R> library(flashClust)
   arbre <- hclust(md^2, method = "ward")
```

Le dendrogramme obtenu peut être affiché simplement avec `plot`. Lorsque le nombre d'individus est important, il peut être utile de ne pas afficher les étiquettes des individus avec `labels=FALSE`.

```
R> plot(arbre, labels = FALSE, main = "Dendrogramme")
```

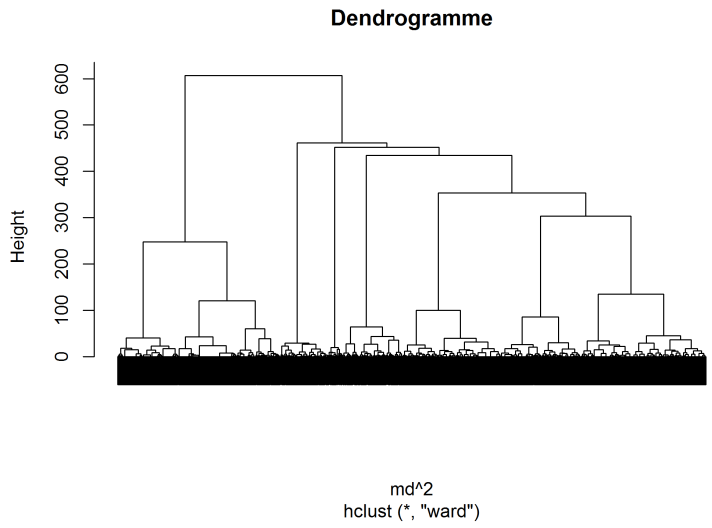



*Figure 1. Dendrogramme obtenu avec hclust*

La fonction `agnes` de l'extension **cluster** peut également être utilisée pour calculer le dendrogramme. Cependant, à l'usage, elle semble être un peu plus lente que `hclust`.

```
R> library(cluster)
   arbre2 <- agnes(md^2, method = "ward")
```

Le résultat obtenu n'est pas au même format que celui de `hclust`. Il est possible de transformer un objet `agnes` au format `hclust` avec `as.hclust`.

```
R> as.hclust(arbre2)
```

**INFO**

De nombreuses possibilités graphiques sont possibles avec les dendrogrammes. Des exemples documentés sont disponibles à cette adresse : http://rpubs.com/gaston/dendrograms.

# Découper le dendrogramme

Pour obtenir une partition de la population, il suffit de découper le dendrogramme obtenu à une certaine hauteur. En premier lieu, une analyse de la forme du dendrogramme pourra nous donner une indication sur le nombre de classes à retenir. Dans notre exemple, deux branches bien distinctes apparaissent sur l'arbre.

Pour nous aider, nous pouvons représenter les sauts d'inertie du dendrogramme selon le nombre de classes retenues.

```
R> inertie <- sort(arbre$height, decreasing = TRUE)
   plot(inertie[1:20], type = "s", xlab = "Nombre de
   classes",
     ylab = "Inertie")
```

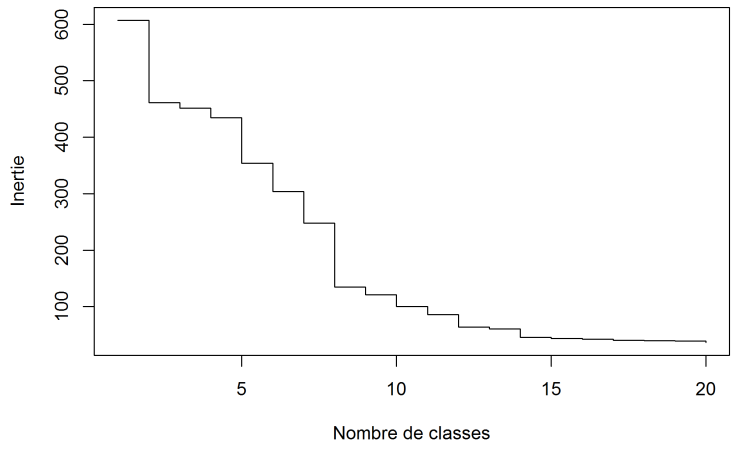



*Figure 2. Inertie du dendrogramme*

On voit trois sauts assez nets, à 2, 5 et 8 classes, que nous avons représentés ci-dessous respectivement en vert, en rouge et en bleu.

```
R> plot(inertie[1:20], type = "s", xlab = "Nombre de
   classes",
     ylab = "Inertie")
   points(c(2, 5, 8), inertie[c(2, 5, 8)], col =
   c("green3",
     "red3", "blue3"), cex = 2, lwd = 3)
```

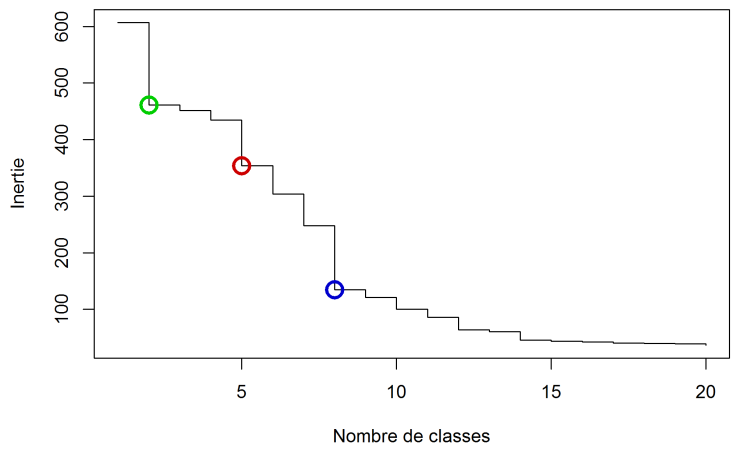



*Figure 3. Sauts d'inertie du dendrogramme*

La fonction `rect.hclust` permet de visualiser les différentes partitions directement sur le dendrogramme.

```
R> plot(arbre, labels = FALSE, main = "Partition en 2,
   5 ou 8 classes",
     xlab = "", ylab = "", sub = "", axes = FALSE,
   hang = -1)
   rect.hclust(arbre, 2, border = "green3")
   rect.hclust(arbre, 5, border = "red3")
   rect.hclust(arbre, 8, border = "blue3")
```

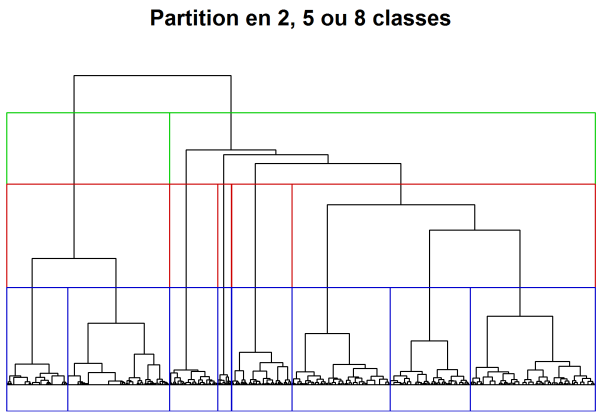



*Figure 4. Différentes partitions du dendrogramme*

L'extension **FactoMineR** (que nous aborderons dans une section dédiée ci-après) suggère d'utiliser la partition ayant la plus grande perte relative d'inertie.

Dans l'extension **JLutils** (disponible sur GitHub), nous avons développer une fonction `best.cutree` qui permet de calculer cette indicateur à partir de n'importe quel dendrogramme calculé avec `hclust` ou `agnes`.

Pour installer **JLutils**, on aura recours au package **devtools** et à sa fonction `install_github` :

```
R> library(devtools)
   install_github("larmarange/JLutils")
```

Par défaut, `best.cutree` regarde quelle serait la meilleure partition entre 3 et 20 classes.

```
R> library(JLutils)
   best.cutree(arbre)
```

```
[1] 5
```

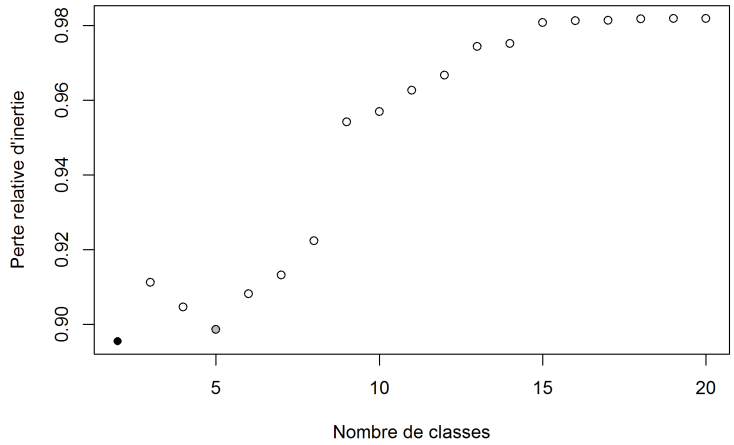
En l'occurence il s'agirait d'une partition en 5 classes. Il est possible de modifier le minimum et le maximum des partitions recherchées avec `min` et `max`.

```
R> best.cutree(arbre, min = 2)
```

```
[1] 2
```

On peut également représenter le graphique des pertes relatives d'inertie avec `graph=TRUE`. La meilleure partition selon ce critère est représentée par un point noir et la seconde par un point gris.

```
R> best.cutree(arbre, min = 2, graph = TRUE, xlab =
   "Nombre de classes",
     ylab = "Perte relative d'inertie")
```




```
[1] 2
```

*Figure 5. Perte relative d'inertie selon le nombre de classes*

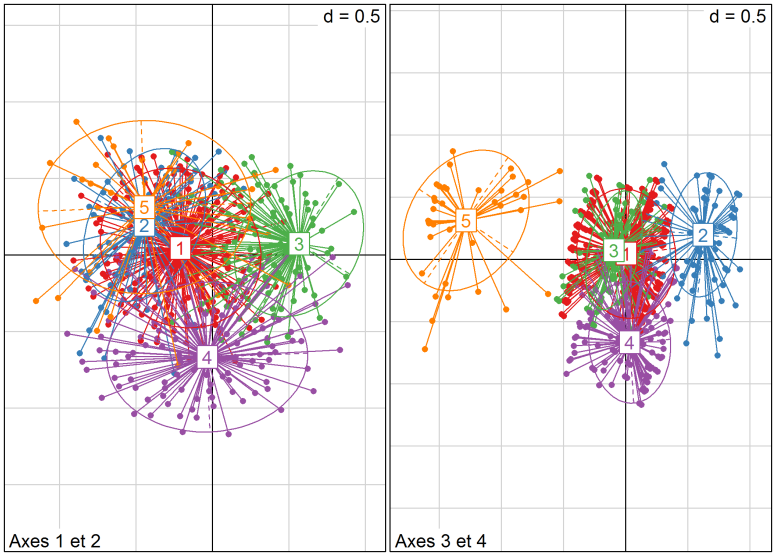
Un découpage en deux classes minimise ce critère. Cependant, si l'on souhaite réaliser une analyse un peu plus fine, un nombre de classes plus élevé serait pertinent. Nous allons donc retenir un découpage en cinq classes. Le découpage s'effectue avec la fonction `cutree` {stats}.

```
R> typo <- cutree(arbre, 5)
   freq(typo)
```

```
      n    % val%
1  1031 51.5 51.5
2   164  8.2  8.2
3   553 27.7 27.7
4   205 10.2 10.2
5    47  2.4  2.4
NA    0  0.0   NA
```

La typologie obtenue peut être représentée dans le plan factoriel avec `s.class`.

```
R> par(mfrow = c(1, 2))
   library(RColorBrewer)
   s.class(acm$li, as.factor(typo), col =
   brewer.pal(5,
     "Set1"), sub = "Axes 1 et 2")
   s.class(acm$li, as.factor(typo), 3, 4, col =
   brewer.pal(5,
     "Set1"), sub = "Axes 3 et 4")
```




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

*Figure 6. Projection de la typologie obtenue par CAH selon les 4 premiers axes*

Romain François a developpé une fonction `A2Rplot` permettant de réaliser facilement un dendrogramme avec les branches colorées[4]. Par commodité, cette fonction est disponible directement au sein de l'extension **JLutils**.

Pour réaliser le graphique, on indiquera le nombre de classes et les couleurs à utiliser pour chaque branche de l'arbre :

```
R> A2Rplot(arbre, k = 5, boxes = FALSE, col.up =
   "gray50",
     col.down = brewer.pal(5, "Dark2"), show.labels =
   FALSE)
```

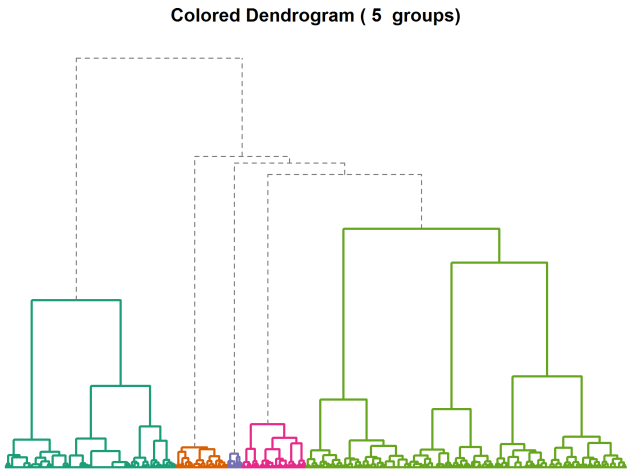



*Figure 7. Un dendrogramme coloré*

# CAH avec l'extension FactoMineR

L'extension **FactoMineR** fournit une fonction `HCPC` permettant de réaliser une classification hiérarchique à partir du résultats d'une analyse factorielle réalisée avec la même extension (voir la section dédiée du chapitre sur l'ACM).

`HCPC` réalise à la fois le calcul de la matrice des distances, du dendrogramme et le partitionnement de la population en classes. Par défaut, `HCPC` calcule le dendrogramme à partir du carré des distances du $\Phi^2$ et avec la méthode de Ward.

Par défaut, l'arbre est affiché à l'écran et l'arbre sera coupé selon la partition ayant la plus grande perte relative d'inertie (comme avec `best.cutree`). Utilisez `graph=FALSE` pour ne pas afficher le graphique et l'argument `nb.clust` pour indiquer le nombre de classes désirées.

```
R> library(FactoMineR)
   acm2 <- MCA(d2[complete.cases(d2), ], ncp = 5,
   graph = FALSE)
   cah <- HCPC(acm2)
```

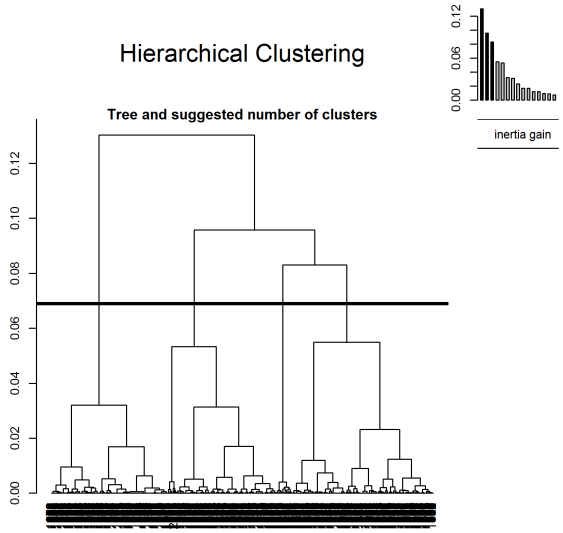



*Figure 8. Graphique produit par HCPC*

On pourra représenter le dendrogramme avec `plot` et l'argument `choice="tree"`.

```
R> plot(cah, choice = "tree")
```

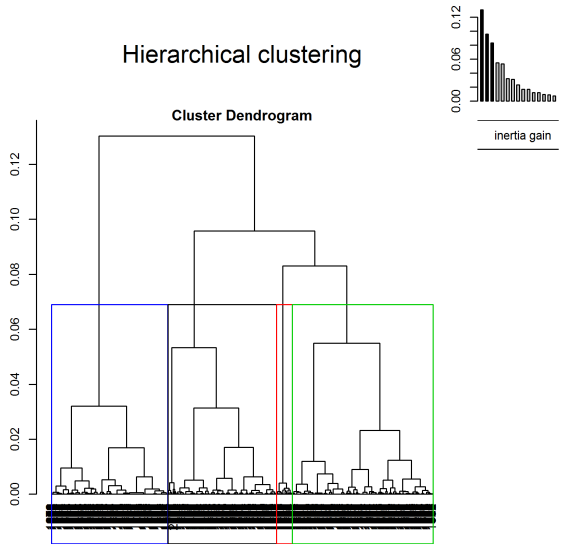



*Figure 9. Dendrogramme obtenu avec HCPC (5 axes)*

Il apparait que le dendrogramme obtenu avec `HCPC` diffère de celui que nous avons calculé précédemment en utilisant la matrice des distances fournies par `dist.dudi`. Cela est dû au fait que `HCPC` procède différement pour calculer la matrice des distances en ne prenant en compte que les axes retenus dans le cadre de l'ACM. Pour rappel, nous avions retenu que 5 axes dans le cadre de notre ACM :

```
R> acm2 <- MCA(d2[complete.cases(d2), ], ncp = 5, graph =
   FALSE)
```

`HCPC` n'a donc pris en compte que ces 5 premiers axes pour calculer les distances entre les individus, considérant que les autres axes n'apportent que du « bruit » rendant la classification instable. Cependant, comme le montre `summary(acm2)`, nos cinq premiers axes n'expliquent que 54 % de la variance. Il usuellement préférable de garder un plus grande nombre d'axes afin de couvrir au moins 80 à 90 % de la variance[5]. De son côté, `dist.dudi` prends en compte l'ensemble des axes pour calculer la matrice des distances. On peut reproduire cela avec **FactoMineR** en indiquant `ncp=Inf` lors du calcul de l'ACM.

```
R> acm2 <- MCA(d2[complete.cases(d2), ], ncp = Inf, graph =
   FALSE)
   cah <- HCPC(acm2, nb.clust = -1, graph = FALSE)
```

On obtient bien cette fois-ci le même résultat.

```
R> plot(cah, choice = "tree")
```

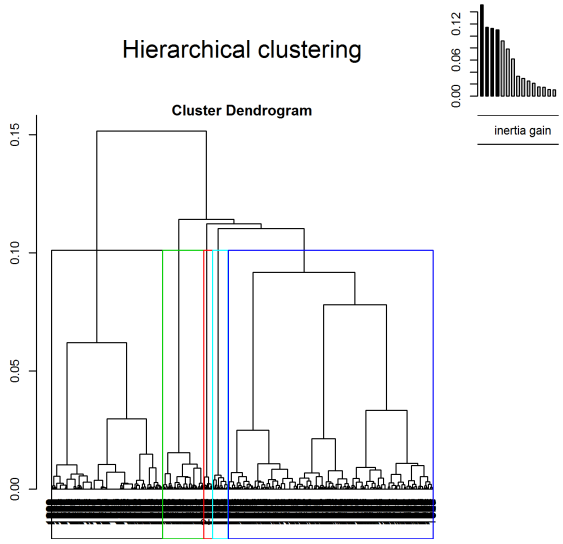



*Figure 10. Dendrogramme obtenu avec HCPC (tous les axes)*

D'autres graphiques sont disponibles, en faisant varier la valeur de l'argument `choice` :

```
R> plot(cah, choice = "3D.map")
```

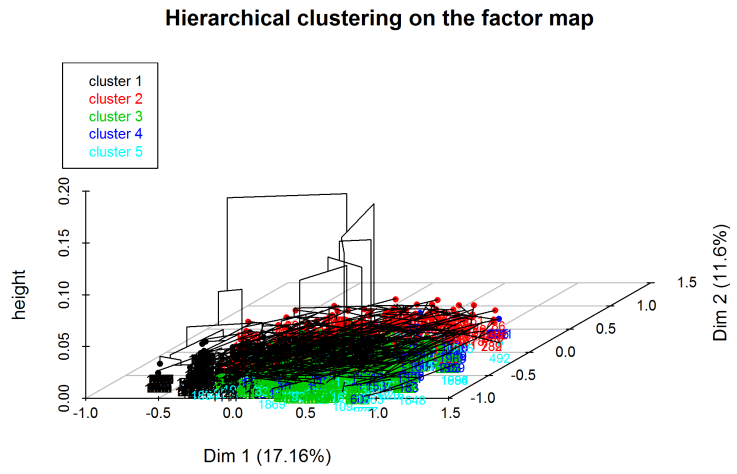



*Figure 11. Représentation en 3 dimensions du dendrogramme*

```
R> plot(cah, choice = "bar")
```

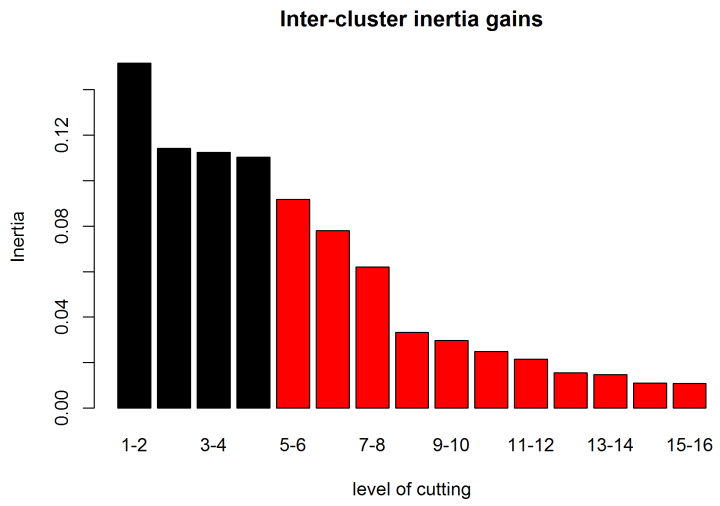



*Figure 12. Gains d'inertie*

```
R> plot(cah, choice = "map")
```

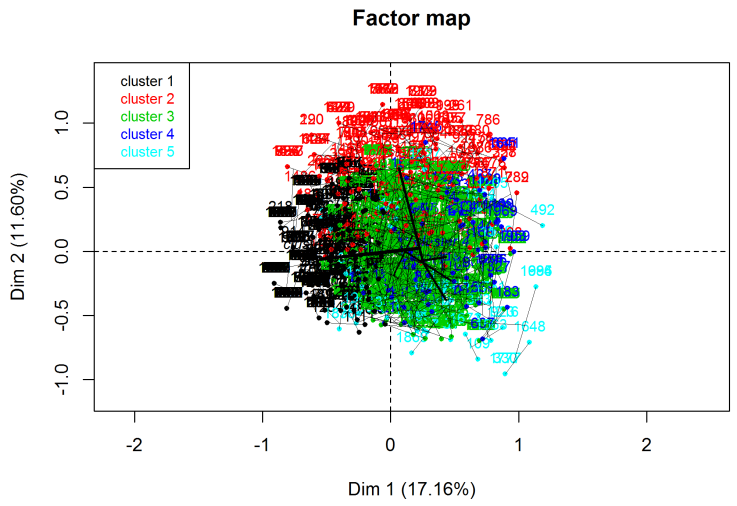



*Figure 13. Projection des catégories sur le plan factoriel*

L'objet renvoyé par `HCPC` contient de nombreuses informations. La partition peut notamment être récupérée avec `cah$data.clust$clust`. Il y a également diverses statistiques pour décrire les catégories.

```
R> cah
```

```
**Results for the Hierarchical Clustering on Principal
Components**
   name
1  "$data.clust"
2  "$desc.var"
3  "$desc.var$test.chi2"
4  "$desc.axes$category"
5  "$desc.axes"
6  "$desc.axes$quanti.var"
7  "$desc.axes$quanti"
8  "$desc.ind"
9  "$desc.ind$para"
10 "$desc.ind$dist"
11 "$call"
12 "$call$t"

description
1  "dataset with the cluster of the
individuals"
2  "description of the clusters by the
variables"
3  "description of the cluster var. by the categorical
var."
4  "description of the clusters by the
categories."
5  "description of the clusters by the
dimensions"
6  "description of the cluster var. by the
axes"
7  "description of the clusters by the
axes"
8  "description of the clusters by the
individuals"
9  "parangons of each
clusters"
10 "specific
individuals"
11 "summary
statistics"
```

```
12 "description of the
tree"
```

```
R> freq(cah$data.clust$clust)
```

```
      n    % val%
1   513 27.2 27.2
2   201 10.7 10.7
3  1051 55.7 55.7
4    78  4.1  4.1
5    43  2.3  2.3
NA    0  0.0   NA
```

1. Pour une présentation de ces différentes distances, on pourra se référer à http://old.biodiversite.wallonie.be/outils/methodo/similarite_distance.htm ou encore à ce support de cours par D. Chessel, J. Thioulouse et A.B. Dufour disponible à http://pbil.univ-lyon1.fr/R/pdf/stage7.pdf.↩
2. Cette même fonction peut aussi être utilisée pour calculer une distance après une analyse en composantes principales ou une analyse mixte de Hill et Smith.↩
3. On pourra consulter le cours de FG Carpentier déjà cité ou bien des ouvrages d'analyse statistique.↩
4. Voir http://addicted2or.free.fr/packages/A2R/lastVersion/R/code.R.↩
5. Voir http://factominer.free.fr/classical-methods/classification-hierarchique-sur-composantes-principales.html↩

# Analyse de séquences

**NOTE**

La version originale de ce chapitre est une reprise, avec l'aimable autorisation de son auteur, d'un article de Nicolas Robette intitulé *L'analyse de séquences : une introduction avec le logiciel R et le package TraMineR* et publié sur le blog Quanti (http://quanti.hypotheses.org/686/).

Depuis les années 1980, l'étude quantitative des trajectoires biographiques (*life course analysis*) a pris une ampleur considérable dans le champ des sciences sociales. Les collectes de données micro-individuelles longitudinales se sont développées, principalement sous la forme de panels ou d'enquêtes rétrospectives. Parallèlement à cette multiplication des données disponibles, la méthodologie statistique a connu de profondes évolutions. L'analyse des biographies (*event history analysis*) — qui ajoute une dimension diachronique aux modèles économétriques mainstream — s'est rapidement imposée comme l'approche dominante : il s'agit de modéliser la durée des situations ou le risque d'occurrence des événements.

## L'analyse de séquences

Cependant, ces dernières années ont vu la diffusion d'un large corpus de méthodes descriptives d'analyse de séquences, au sein desquelles l'appariement optimal (*optimal matching*) occupe une place centrale[1]. L'objectif principal de ces méthodes est d'identifier — dans la diversité d'un corpus de séquences constituées de séries d'états successifs — les régularités, les ressemblances, puis le plus souvent de construire des typologies de « séquences-types ». L'analyse de

séquences constitue donc un moyen de décrire mais aussi de mieux comprendre le déroulement de divers processus.

La majeure partie des applications de l'analyse de séquences traite de trajectoires biographiques ou de carrières professionnelles. Dans ces cas, chaque trajectoire ou chaque carrière est décrite par une séquence, autrement dit par une suite chronologiquement ordonnée de « moments » élémentaires, chaque moment correspondant à un « état » déterminé de la trajectoire (par exemple, pour les carrières professionnelles : être en emploi, au chômage ou en inactivité). Mais on peut bien sûr imaginer des types de séquences plus originaux : Andrew Abbott[2], le sociologue américain qui a introduit l'*optimal matching* dans les sciences scientifiques ou des séquences de pas de danses traditionnelles.

En France, les premiers travaux utilisant l'appariement optimal sont ceux de Claire Lemercier[3] sur les carrières des membres des institutions consulaires parisiennes au XIXe siècle (Lemercier, 2005), et de Laurent Lesnard[4] sur les emplois du temps (Lesnard, 2008). Mais dès les années 1980, les chercheurs du Céreq construisaient des typologies de trajectoires d'insertion à l'aide des méthodes d'analyse des données « à la française » (analyse des correspondances, etc.)[5]. Au final, on dénombre maintenant plus d'une centaine d'articles de sciences sociales contenant ou discutant des techniques empruntées à l'analyse de séquences.

Pour une présentation des différentes méthodes d'analyse de séquences disponibles et de leur mise en oeuvre pratique, il existe un petit manuel en français, publié en 2011 dernière aux éditions du Ceped (collection « Les clefs pour »[6]) et disponible en pdf[7] (Robette, 2011). De plus, un article récemment publié dans le *Bulletin de Méthodologie Sociologique* compare de manière systématique les résultats obtenus par les principales méthodes d'analyse de séquences (Robette & Bry, 2012). La conclusion en est qu'avec des données empiriques aussi structurées que celles que l'on utilise en sciences sociales, l'approche est robuste, c'est-à-dire qu'un changement de méthode aura peu d'influence sur les principaux résultats. Cependant, l'article tente aussi de décrire les spécificités de chaque méthode et les différences marginales qu'elles font apparaître, afin de permettre aux chercheurs de mieux adapter leurs choix méthodologiques à leur question de recherche.

Afin d'illustrer la démarche de l'analyse de séquences, nous allons procéder ici à la description « pas à pas » d'un corpus de carrières professionnelles, issues de

l'enquête *Biographies et entourage* (Ined, 2000)[8]. Et pour ce faire, on va utiliser le logiciel **R**, qui propose la solution actuellement la plus complète et la plus puissante en matière d'analyse de séquences. Les méthodes d'analyse de séquences par analyses factorielles ou de correspondances ne nécessitent pas de logiciel spécifique : tous les logiciels de statistiques généralistes peuvent être utilisés (**SAS**, **SPSS**, **Stata**, R, etc.). En revanche, il n'existe pas de fonctions pour l'appariement optimal dans **SAS** ou **SPSS**. Certains logiciels gratuits implémentent l'appariement optimal (comme **Chesa**[9] ou **TDA**[10]) mais il faut alors recourir à d'autres programmes pour dérouler l'ensemble de l'analyse (classification, représentation graphique). **Stata** propose le module **sq**[11], qui dispose d'un éventail de fonctions intéressantes. Mais c'est **R** et le package **TraMineR**[12], développé par des collègues de l'Université de Genève (Gabadinho et al, 2011), qui fournit la solution la plus complète et la plus puissante à ce jour : on y trouve l'appariement optimal mais aussi d'autres algorithmes alternatifs, ainsi que de nombreuses fonctions de description des séquences et de représentation graphique.

# Installer TraMineR et récupérer les données

Tout d'abord, à quoi ressemblent nos données ? On a reconstruit à partir de l'enquête les carrières de 1000 hommes. Pour chacune, on connaît la position professionnelle chaque année, de l'âge de 14 ans jusqu'à 50 ans. Cette position est codée de la manière suivante : les codes 1 à 6 correspondent aux groupes socioprofessionnels de la nomenclature des PCS de l'INSEE 13 (agriculteurs exploitants ; artisans, commerçants et chefs d'entreprise ; cadres et professions intellectuelles supérieures ; professions intermédiaires ; employés ; ouvriers) ; on y a ajouté « études » (code 7), « inactivité » (code 8) et « service militaire » (code 9). Le fichier de données comporte une ligne par individu et une colonne par année : la variable *csp1* correspond à la position à 14 ans, la variable *csp2* à la position à 15 ans, etc. Par ailleurs, les enquêtés étant tous nés entre 1930 et 1950, on ajoute à notre base une variable « génération » à trois modalités, prenant les valeurs suivantes : 1="1930-1938" ; 2="1939-1945" ; 3="1946-1950". Au final, la base est constituée de 500 lignes et de 37 + 1 = 38 colonnes et se présente sous la forme d'un fichier texte au format **csv** (téléchargeable à http://larmarange.github.io/analyse-R/data/trajpro.csv).

Une fois **R** ouvert, on commence par installer les extensions nécessaires à ce programme (opération à ne réaliser que lors de leur première utilisation) et par les charger en mémoire. L'extension **TraMineR** propose de nombreuses fonctions pour l'analyse de séquences. L'extension **cluster** comprend un certain nombre de méthodes de classification automatique[13].

```
R> install.packages(c("TraMineR"))
```

```
R> library(TraMineR)
```

```
TraMineR stable version 1.8-9 (Built: 2015-05-07)
Website: http://mephisto.unige.ch/traminer
Please type 'citation("TraMineR")' for citation
information.
```

```
R> library(cluster)
```

On importe ensuite les données, on recode la variable « génération » pour lui donner des étiquettes plus explicites. On jette également un coup d'oeil à la structure du tableau de données :

```
R> donnees <- read.csv("http://larmarange.github.io/
   analyse-R/data/trajpro.csv",
     header = T)
```

```
R> donnees$generation <- factor(donnees$generation, labels
   = c("1930-38",
     "1939-45", "1946-50"))
   str(donnees)
```

```
'data.frame':   1000 obs. of  38 variables:
 $ csp1      : int  1 7 6 7 7 6 7 7 7 6 ...
 $ csp2      : int  1 7 6 7 7 6 7 7 6 6 ...
 $ csp3      : int  1 7 6 6 7 6 7 7 6 6 ...
 $ csp4      : int  1 7 6 6 7 6 7 7 6 6 ...
 $ csp5      : int  1 7 6 6 7 6 7 7 6 6 ...
 $ csp6      : int  1 7 6 6 7 6 9 7 6 6 ...
 $ csp7      : int  6 9 6 6 7 6 9 7 9 6 ...
 $ csp8      : int  6 9 9 6 7 6 9 7 4 6 ...
 $ csp9      : int  6 6 9 6 7 6 9 3 4 9 ...
 $ csp10     : int  6 6 9 6 7 6 4 3 4 9 ...
 $ csp11     : int  6 6 6 6 3 6 4 3 4 6 ...
 $ csp12     : int  6 6 6 6 3 6 4 3 4 6 ...
 $ csp13     : int  6 6 6 6 3 6 4 3 4 6 ...
 $ csp14     : int  6 4 6 6 3 6 4 3 4 6 ...
 $ csp15     : int  6 4 6 6 3 6 4 3 4 6 ...
 $ csp16     : int  6 4 6 6 3 6 6 3 4 6 ...
 $ csp17     : int  6 4 6 6 3 6 6 3 4 6 ...
 $ csp18     : int  6 4 6 6 3 6 6 3 4 6 ...
 $ csp19     : int  6 4 6 6 3 6 6 3 4 6 ...
 $ csp20     : int  6 4 6 6 3 6 6 3 4 6 ...
 $ csp21     : int  6 4 6 6 6 6 6 3 4 6 ...
 $ csp22     : int  6 4 6 6 6 6 6 3 4 4 ...
 $ csp23     : int  6 4 6 6 6 6 6 3 4 4 ...
 $ csp24     : int  6 6 6 6 5 6 6 3 4 4 ...
 $ csp25     : int  6 6 6 6 5 6 6 3 4 4 ...
 $ csp26     : int  6 6 6 6 5 6 6 3 4 4 ...
 $ csp27     : int  6 6 6 6 5 6 6 3 4 4 ...
 $ csp28     : int  6 6 6 6 5 6 6 3 4 4 ...
 $ csp29     : int  6 6 6 6 5 6 6 3 4 4 ...
 $ csp30     : int  4 6 6 6 5 6 6 3 4 4 ...
 $ csp31     : int  4 6 6 6 5 6 6 3 4 4 ...
 $ csp32     : int  4 6 6 6 5 6 6 3 4 4 ...
 $ csp33     : int  4 6 6 6 5 6 6 3 4 4 ...
 $ csp34     : int  4 6 6 6 5 6 6 3 4 4 ...
 $ csp35     : int  4 6 6 6 5 6 6 3 4 4 ...
 $ csp36     : int  4 6 6 6 5 6 6 3 4 4 ...
 $ csp37     : int  4 6 6 6 5 6 6 3 4 4 ...
 $ generation: Factor w/ 3 levels "1930-38","1939-45",..:
2 1 1 3 2 3 1 1 2 1 ...
```

On a bien 1000 observations et 38 variables. On définit maintenant des *labels* pour les différents états qui composent les séquences et on crée un objet « séquence » avec `seqdef` :

```
R> labels <- c("agric", "acce", "cadr", "pint", "empl",
     "ouvr", "etud", "inact", "smil")
   seq <- seqdef(donnees[, 1:37], states = labels)
```

```
 [>] state coding:
       [alphabet]  [label]  [long label]
     1  1           agric    agric
     2  2           acce     acce
     3  3           cadr     cadr
     4  4           pint     pint
     5  5           empl     empl
     6  6           ouvr     ouvr
     7  7           etud     etud
     8  8           inact    inact
     9  9           smil     smil
 [>] 1000 sequences in the data set
 [>] min/max sequence length: 37/37
```

# Appariement optimal et classification

Ces étapes préalables achevées, on peut comparer les séquences en calculant les dissimilarités entre paires de séquences. On va ici utiliser la méthode la plus répandue, l'appariement optimal (*optimal matching*). Cette méthode consiste, pour chaque paire de séquences, à compter le nombre minimal de modifications (substitutions, suppressions, insertions) qu'il faut faire subir à l'une des séquences pour obtenir l'autre. On peut considérer que chaque modification est équivalente, mais il est aussi possible de prendre en compte le fait que les « distances » entre les différents états n'ont pas toutes la même « valeur » (par exemple, la distance sociale entre emploi à temps plein et chômage est plus grande qu'entre emploi à temps plein et emploi à temps partiel), en assignant aux différentes modifications des « coûts » distincts. Dans notre exemple, on va créer

avec `seqsubm` une « matrice des coûts de substitution » dans laquelle tous les coûts sont constants et égaux à 2[14] :

```
R> couts <- seqsubm(seq, method = "CONSTANT", cval = 2)
```

```
 [>] creating 9x9 substitution-cost matrix using 2 as
constant value
```

Ensuite, on calcule la matrice de distance entre les séquences (i.e contenant les « dissimilarités » entre les séquences) avec `seqdist`, avec un coût d'insertion/suppression (*indel*) que l'on fixe ici à 1,1 :

```
R> seq.om <- seqdist(seq, method = "OM", indel = 1.1,
     sm = couts)
```

```
 [>] 1000 sequences with 9 distinct events/states
 [>] 818 distinct sequences
 [>] min/max sequence length: 37/37
 [>] computing distances using OM metric
 [>] total time: 1.98 secs
```

Cette matrice des distances ou des dissimilarités entre séquences peut ensuite être utilisée pour une classification ascendante hiérarchique (CAH), qui permet de regrouper les séquences en un certain nombre de « classes » en fonction de leur proximité :

```
R> seq.agnes <- agnes(as.dist(seq.om), method = "ward",
     keep.diss = FALSE)
```

Avec la fonction `plot`, il est possible de tracer l'arbre de la classification (dendrogramme).

```
R> plot(as.dendrogram(seq.agnes), leaflab = "none")
```

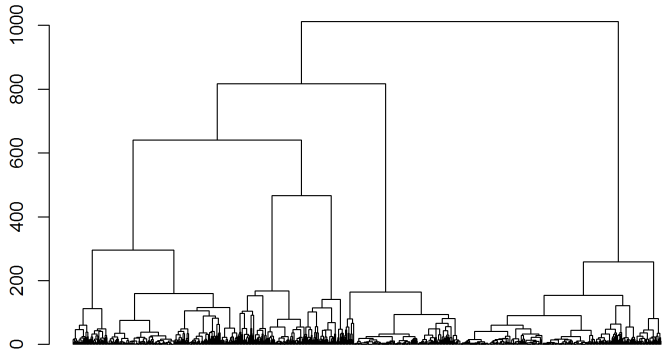



*Figure 1. Dendrogramme de la classification des séquences*

De même, on peut représenter les sauts d'inertie.

```
R> plot(sort(seq.agnes$height, decreasing =
   TRUE)[1:20],
     type = "s", xlab = "nb de classes", ylab =
   "inertie")
```

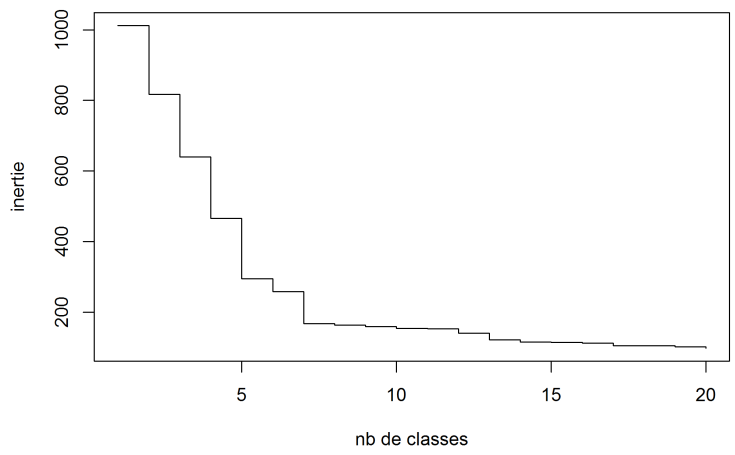



*Figure 2. Sauts d'inertie de la classification des séquences*

L'observation, sur ce dendogramme ou sur la courbe des sauts d'inertie, des sauts d'inertie des dernières étapes de la classification peut servir de guide pour déterminer le nombre de classes que l'on va retenir pour la suite des analyses. Une première inflexion dans la courbe des sauts d'inertie apparaît au niveau d'une partition en 5 classes. On voit aussi une seconde inflexion assez nette à 7 classes. Mais il faut garder en tête le fait que ces outils ne sont que des guides, le choix devant avant tout se faire après différents essais, en fonction de l'intérêt des résultats par rapport à la question de recherche et en arbitrant entre exhaustivité et parcimonie.

On fait ici le choix d'une partition en 5 classes :

```
R> nbcl <- 5
   seq.part <- cutree(seq.agnes, nbcl)
   seq.part <- factor(seq.part, labels = paste("classe",
     1:nbcl, sep = "."))
```

# Représentations graphiques

Pour se faire une première idée de la nature des classes de la typologie, il existe un certain nombre de représentations graphiques. Les chronogrammes (*state distribution plots*) présentent une série de coupes transversales : pour chaque âge, on a les proportions d'individus de la classe dans les différentes situations (agriculteur, étudiant, etc.). Ce graphique s'obtient avec `seqdplot` :

```
R> seqdplot(seq, group = seq.part, xtlab = 14:50,
   border = NA,
     withlegend = T)
```

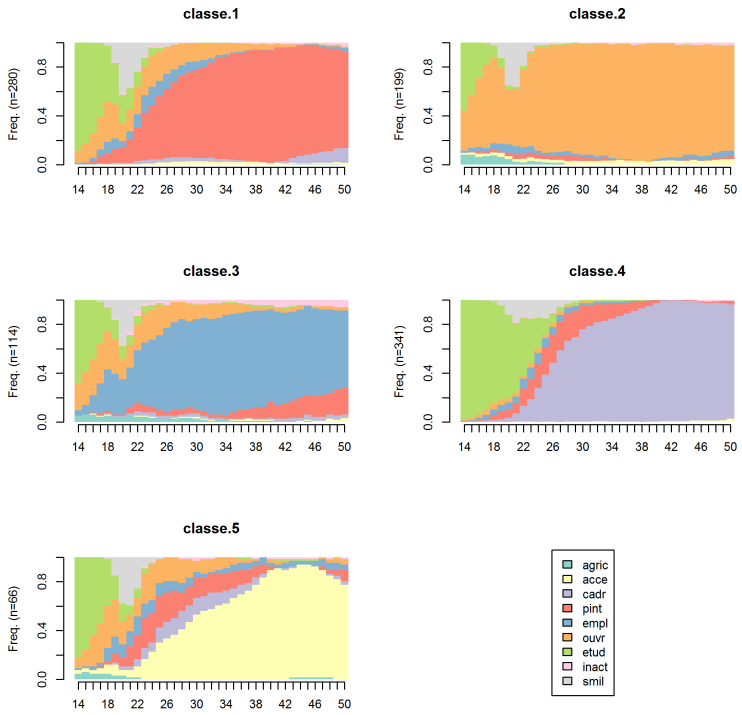



*Figure 3. Chronogrammes*

Chacune des classes semble caractérisée par un groupe professionnel principal : profession intermédiaire pour la classe 1, ouvrier pour la 2, employé pour la 3, cadre pour la 4 et indépendant pour la 5. Cependant, on aperçoit aussi des

« couches » d'autres couleurs, indiquant que l'ensemble des carrières ne sont probablement pas stables.

Les « tapis » (*index plots*), obtenus avec `seqiplot`, permettent de mieux visualiser la dimension individuelle des séquences. Chaque segment horizontal représente une séquence, découpée en sous-segments correspondant aux aux différents états successifs qui composent la séquence.

```
R> seqiplot(seq, group = seq.part, xtlab = 14:50, tlim
   = 0,
     space = 0, border = NA, withlegend = T, yaxis =
   FALSE)
```

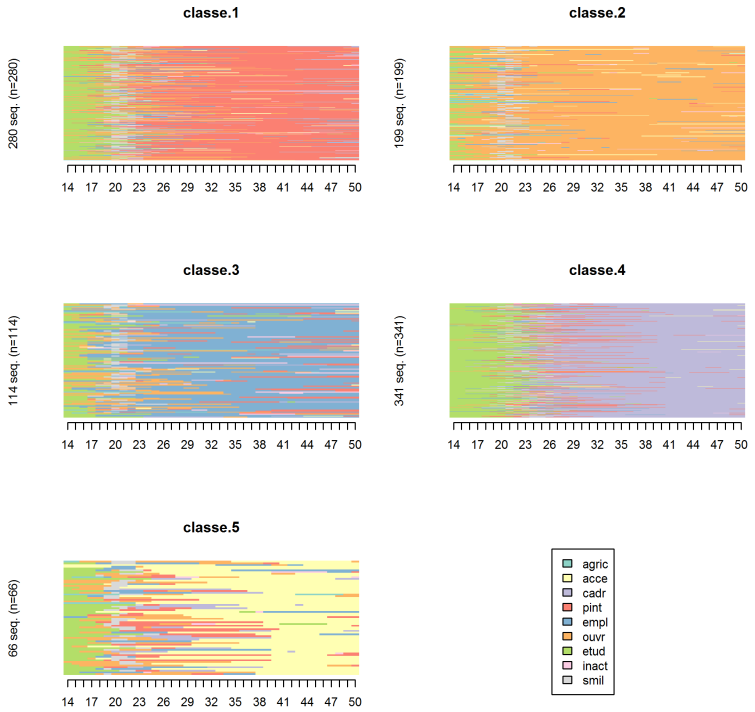



*Figure 4. Tapis des séquences triés*

Il est possible de trier les séquences pour rendre les tapis plus lisibles (on trie ici par *multidimensional scaling* à l'aide de la fonction `cmdscale`).

```
R> ordre <- cmdscale(as.dist(seq.om), k = 1)
   seqiplot(seq, group = seq.part, sortv = ordre,
   xtlab = 14:50,
     tlim = 0, space = 0, border = NA, withlegend = T,
     yaxis = FALSE)
```

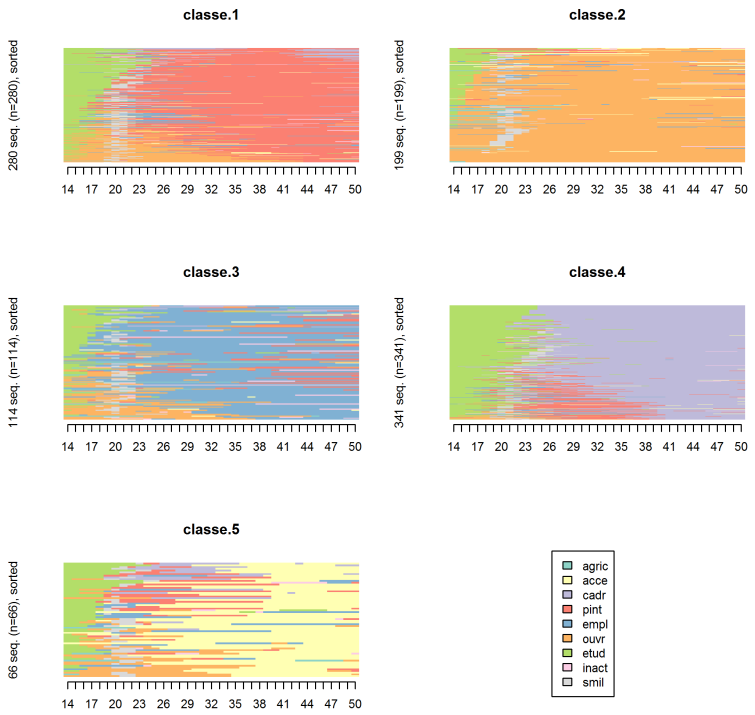



*Figure 5. Tapis des séquences triés par multidimensional scaling*

On voit mieux apparaître ainsi l'hétérogénéité de certaines classes. Les classes 1, 3 et 4, par exemple, semblent regrouper des carrières relativement stables (respectivement de professions intermédiaires, d'employés et de cadres) et des carrières plus « mobiles » commencées comme ouvrier (classes 1 et 3, en orange) ou comme profession intermédiaire (classe 4, en rouge). De même, la majorité des membres de la dernière classe commencent leur carrière dans un groupe professionnel distinct de celui qu'ils occuperont par la suite (indépendants). Ces distinctions apparaissent d'ailleurs si on relance le programme avec un nombre plus élevé de classes (en remplaçant le 5 de la ligne `nbcl <- 5` par 7, seconde inflexion de la courbe des sauts d'inertie, et en exécutant de nouveau le

programme à partir de cette ligne) : les stables et les mobiles se trouvent alors dans des classes distinctes.

Le package **JLutils**, disponible seulement sur GitHub, propose une fonction `seq_heatmap` permettant de représenter le tapis de l'ensemble des séquences selon l'ordre du dendrogramme.

Pour installer **JLutils**, on aura recours au package **devtools** et à sa fonction `install_github` :

```
R> library(devtools)
   install_github("larmarange/JLutils")
```

On peut ensuite générer le graphique ainsi :

```
R> library(JLutils)
   seq_heatmap(seq, seq.agnes, labCol = 14:50)
```

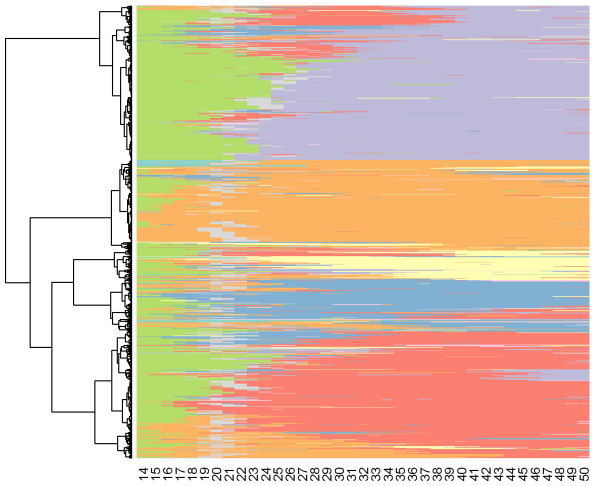



*Figure 6. Tapis des séquences trié selon le dendrogramme*

La distance moyenne des séquences d'une classe au centre de cette classe, obtenue avec `disscenter`, permet de mesurer plus précisément l'homogénéité des classes :

```
R> round(aggregate(disscenter(as.dist(seq.om), group =
   seq.part),
     list(seq.part), mean)[, -1], 1)
```

```
[1] 13.1  8.9 15.6  9.7 16.5
```

Cela nous confirme que les classes 1, 3 et 5 sont nettement plus hétérogènes que les autres, alors que la classe 2 est la plus homogène.

D'autres représentations graphiques existent pour poursuivre l'examen de la typologie. On peut visualiser les 10 séquences les plus fréquentes de chaque classe avec `seqfplot`.

```
R> seqfplot(seq, group = seq.part, withlegend = T)
```

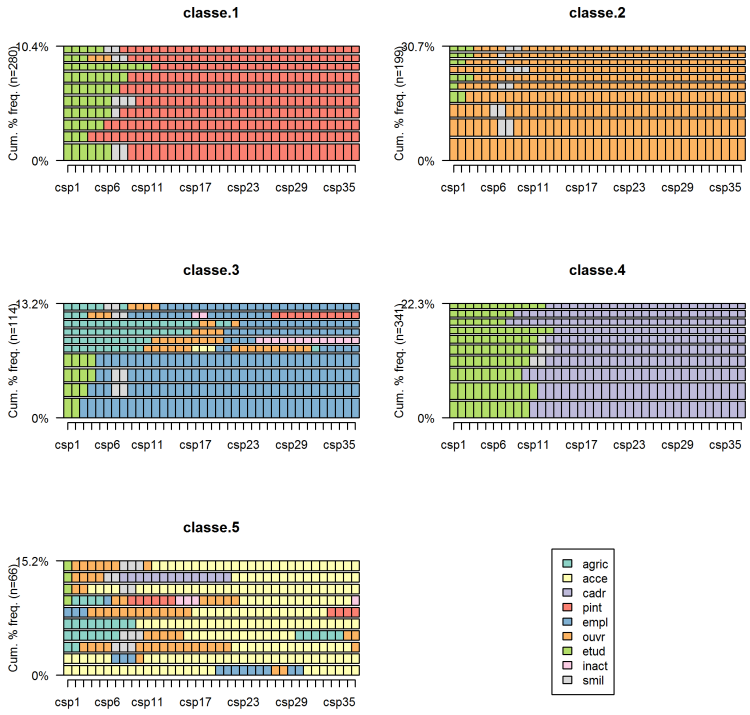



*Figure 7. Séquences les plus fréquentes de chaque classe*

On peut aussi visualiser avec `seqmsplot` l'état modal (celui qui correspond au plus grand nombre de séquences de la classe) à chaque âge.

```
R> seqmsplot(seq, group = seq.part, xtlab = 14:50,
   withlegend = T,
     title = "classe")
```

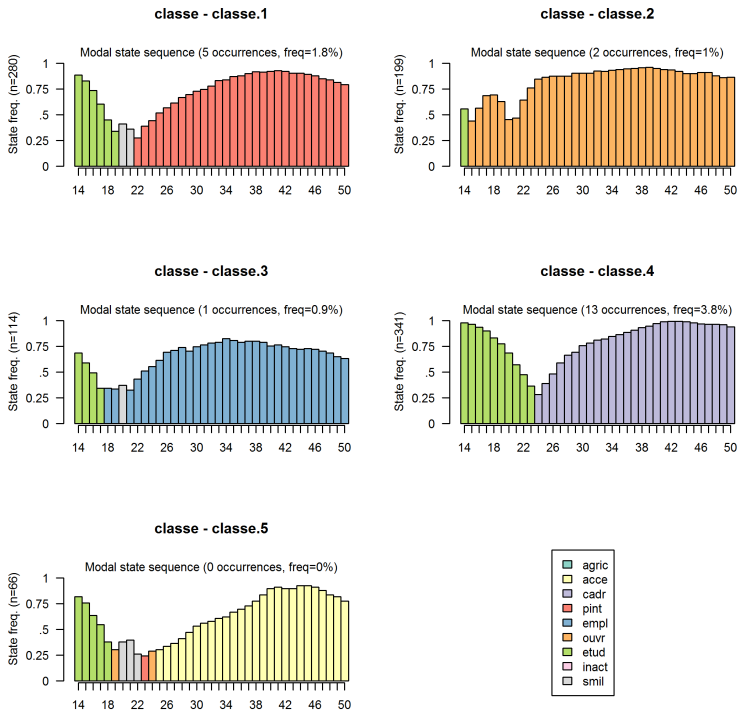



*Figure 8. Statut modal à chaque âge*

On peut également représenter avec `seqmtplot` les durées moyennes passées dans les différents états.

```
R> seqmtplot(seq, group = seq.part, withlegend = T)
```

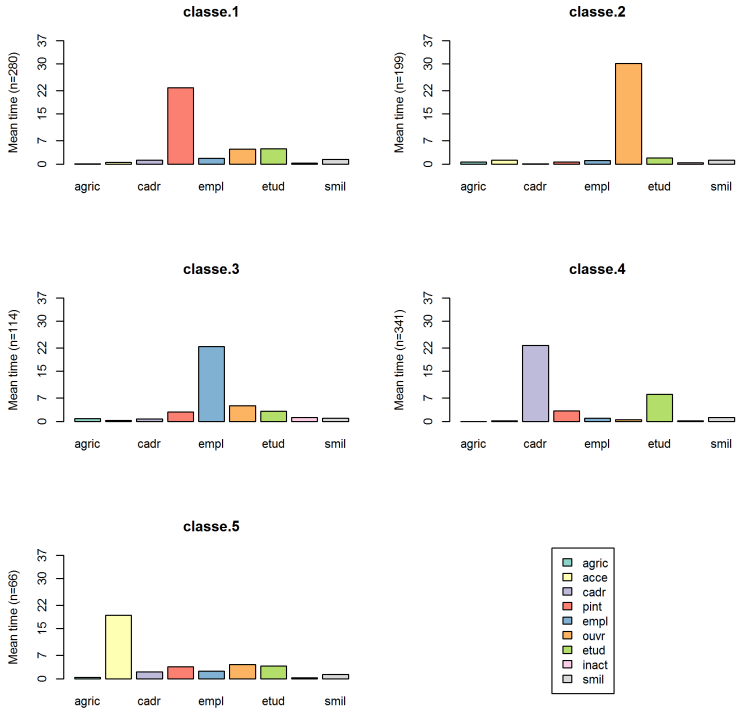



*Figure 9. Durée moyenne dans chaque statut*

Enfin, l'entropie transversale décrit l'évolution de l'homogénéité de la classe. Pour un âge donné, une entropie proche de 0 signifie que tous les individus de la classe (ou presque) sont dans la même situation. À l'inverse, l'entropie est de 1 si les individus sont dispersés dans toutes les situations. Ce type de graphique produit par `seqHtplot` peut être pratique pour localiser les moments de transition, l'insertion professionnelle ou une mobilité sociale ascendante.

```
R> seqHtplot(seq, group = seq.part, xtlab = 14:50,
   withlegend = T)
```

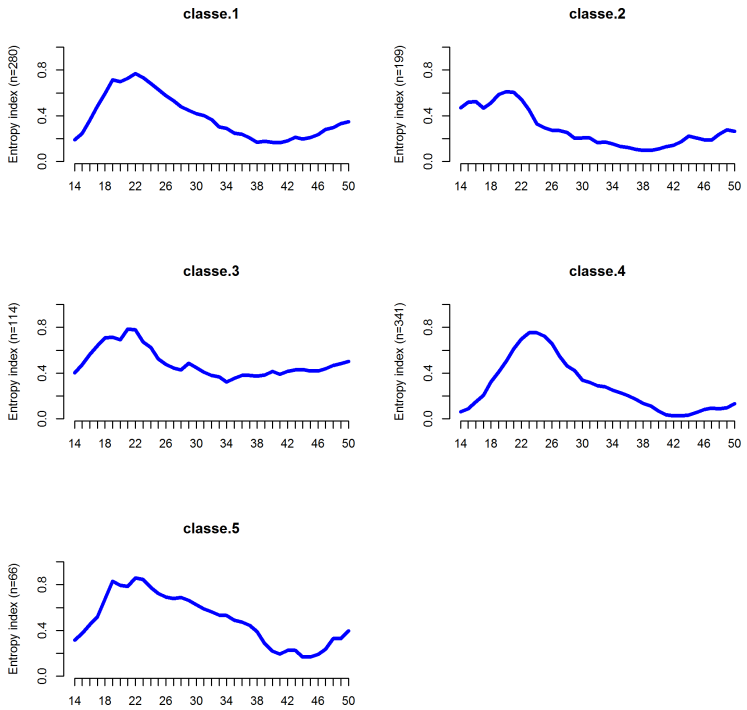



*Figure 10. Entropie transversale*

On souhaite maintenant connaître la distribution de la typologie (en effectifs et en pourcentages) :

```
R> library(questionr)
   freq(seq.part)
```

```
           n    % val%
classe.1 280 28.0 28.0
classe.2 199 19.9 19.9
classe.3 114 11.4 11.4
classe.4 341 34.1 34.1
classe.5  66  6.6  6.6
NA         0  0.0   NA
```

On poursuit ensuite la description des classes en croisant la typologie avec la variable *generation* :

```
R> cprop(table(seq.part, donnees$generation))
```

```
seq.part   1930-38 1939-45 1946-50 Ensemble
  classe.1  25.6    27.1    31.0    28.0
  classe.2  20.9    20.0    18.9    19.9
  classe.3   7.9    13.6    12.9    11.4
  classe.4  38.2    31.9    32.1    34.1
  classe.5   7.4     7.5     5.2     6.6
  Total    100.0   100.0   100.0   100.0
```

```
R> chisq.test(table(seq.part, donnees$generation))
```

```
    Pearson's Chi-squared test

data:  table(seq.part, donnees$generation)
X-squared = 12.032, df = 8, p-value = 0.1498
```

Le lien entre le fait d'avoir un certain type de carrières et la cohorte de naissance est significatif à un seuil de 15 %. On constate par exemple l'augmentation continue de la proportion de carrières de type « professions intermédiaires » (classe 1) et, entre les deux cohortes les plus anciennes, l'augmentation de la part des carrières de type « employés » (classe 3) et la baisse de la part des carrières de type « cadres » (classe 4).

Bien d'autres analyses sont envisageables : croiser la typologie avec d'autres variables (origine sociale, etc.), construire l'espace des carrières possibles, étudier les interactions entre trajectoires familiales et professionnelles, analyser la variance des dissimilarités entre séquences en fonction de plusieurs variables « explicatives[15] »…

Mais l'exemple proposé est sans doute bien suffisant pour une première introduction !

# Bibliographie


- Abbott A., 2001, *Time matters. On theory and method*, The University of Chicago Press.
- Abbott A., Hrycak A., 1990, « Measuring ressemblance in sequence data: an optimal matching analysis of musicians' careers», *American journal of sociology*, (96), p.144-185. http://www.jstor.org/stable/10.2307/2780695
- Abbott A., Tsay A., 2000, « Sequence analysis and optimal matching methods in sociology: Review and prospect », *Sociological methods & research*, 29(1), p.3-33. http://smr.sagepub.com/content/29/1/3.short
- Gabadinho, A., Ritschard, G., Müller, N.S. & Studer, M., 2011, « Analyzing and visualizing state sequences in R with TraMineR », *Journal of Statistical Software*, 40(4), p.1-37. http://archive-ouverte.unige.ch/downloader/vital/pdf/tmp/4hff8pe6uhukqiavvgaluqmjq2/out.pdf
- Grelet Y., 2002, « Des typologies de parcours. Méthodes et usages », *Document Génération* 92, (20), 47 p. http://www.cmh.greco.ens.fr/programs/Grelet_typolparc.pdf
- Lelièvre É., Vivier G., 2001, « Évaluation d'une collecte à la croisée du quantitatif et du qualitatif : l'enquête Biographies et entourage », *Population*, (6), p.1043-1073. http://www.persee.fr/web/revues/home/prescript/article/pop_0032-4663_2001_num_56_6_7217
- Lemercier C., 2005, « Les carrières des membres des institutions consulaires parisiennes au XIX^e^ siècle », *Histoire et mesure*, XX (1-2), p.59-95. http://histoiremesure.revues.org/786
- Lesnard L., 2008, « Off-Scheduling within Dual-Earner Couples: An Unequal and Negative Externality for Family Time », *American Journal of Sociology*, 114(2), p.447-490. http://laurent.lesnard.free.fr/IMG/pdf/lesnard_2008_off-scheduling_within_dual-earner_couples-2.pdf
- Lesnard L., Saint Pol T. (de), 2006, « Introduction aux Méthodes d'Appariement Optimal (Optimal Matching Analysis) », *Bulletin de Méthodologie Sociologique*, 90, p.5-25. http://bms.revues.org/index638.html
- Robette N., 2011, *Explorer et décrire les parcours de vie : les typologies de trajectoires*, Ceped (Les Clefs pour), 86 p. http://nicolas.robette.free.fr/Docs/Robette2011_Manuel_TypoTraj.pdf

- Robette N., 2012, « Du prosélytisme à la sécularisation. Le processus de diffusion de l'Optimal Matching Analysis », *document de travail*. http://nicolas.robette.free.fr/Docs/Proselytisme_secularisation_NRobette.pdf
- Robette N., Bry X., 2012, « Harpoon or bait? A comparison of various metrics to fish for life course patterns », *Bulletin de Méthodologie Sociologique*, 116, p.5-24. http://nicolas.robette.free.fr/Docs/Harpoon_maggot_RobetteBry.pdf
- Robette N., Thibault N., 2008, « L'analyse exploratoire de trajectoires professionnelles : analyse harmonique qualitative ou appariement optimal ? », *Population*, 64(3), p.621-646. http://www.cairn.info/revue-population-2008-4-p-621.htm
- Savage M., 2009, « Contemporary Sociology and the Challenge of Descriptive Assemblage », *European Journal of Social Theory*, 12(1), p.155-174. http://est.sagepub.com/content/12/1/155.short


---

1. Pour une analyse des conditions sociales de la diffusion de l'analyse de séquences dans le champ des sciences sociales, voir Robette, 2012.↩
2. http://home.uchicago.edu/~aabbott/↩
3. http://lemercier.ouvaton.org/document.php?id=62↩
4. http://laurent.lesnard.free.fr/article.php3?id_article=22↩
5. Voir par exemple l'article d'Yvette Grelet (2002).↩
6. http://www.ceped.org/?rubrique57↩
7. http://nicolas.robette.free.fr/Docs/Robette2011_Manuel_TypoTraj.pdf↩
8. Pour une analyse plus poussée de ces données, avec deux méthodes différentes, voir Robette & Thibault, 2008. Pour une présentation de l'enquête, voir Lelièvre & Vivier, 2001.↩
9. http://home.fsw.vu.nl/ch.elzinga/↩
10. http://steinhaus.stat.ruhr-uni-bochum.de/tda.html↩
11. http://www.stata-journal.com/article.html?article=st0111↩
12. http://mephisto.unige.ch/traminer/↩

13. Pour une présentation plus détaillée, voir le chapitre sur la classification ascendante hiérarchique (CAH).↩

14. Le fonctionnement de l'algorithme d'appariement optimal — et notamment le choix des coûts — est décrit dans le chapitre 9 du manuel de **TraMineR** (http://mephisto.unige.ch/pub/TraMineR/doc/TraMineR-Users-Guide.pdf).↩

15. L'articulation entre méthodes « descriptives » et méthodes « explicatives » est un prolongement possible de l'analyse de séquences. Cependant, l'analyse de séquences était envisagée par Abbott comme une alternative à la sociologie quantitative *mainstream*, i.e le « paradigme des variables » et ses hypothèses implicites souvent difficilement tenables (Abbott, 2001). Une bonne description solidement fondée théoriquement vaut bien des « modèles explicatifs » (Savage, 2009).↩

# Calculer un âge

**NOTE**

La version originale de cette astuce a été publiée par Joseph Larmarange sur http://joseph.larmarange.net/?Calculer-proprement-un-age-sous-R.

Le calcul d'un âge sous **R** n'est pas forcément aussi trivial qu'il n'y parait.

## Rappel sur les âges

Il convient en premier lieu de rappeler les principaux âges utilisés les démographes :

> *L'âge – on précise parfois âge chronologique – est une des caractéristiques fondamentales de la structure des populations. On l'exprime généralement en années, ou en années et mois, voire en mois et jours, pour les enfants en bas âge ; parfois en années et fractions décimales d'année. Les démographes arrondissent d'ordinaire l'âge à l'unité inférieure, l'exprimant ainsi en années révolues, ou années accomplies, le cas échéant en mois révolus, ou mois accomplis. Cet âge est aussi l'âge au dernier anniversaire. On trouve aussi, dans les statistiques, l'âge atteint dans l'année, qui est égal à la différence de millésimes entre l'année considérée et l'année de naissance. […] On est parfois conduit à préciser que l'on considère un âge exact, pour éviter toute confusion avec un âge en années révolues, qui représente en fait une classe d'âges exacts.*
>
> — Source : *Demopædia* (322)

# Le package lubridate

Le package **lubridate** est spécialement développé pour faciliter la manipulation et le calcul autour des dates. La version de développement intègre depuis peu une fonction `time_length` adaptée au calcul des âges exacts.

**INFO**

La fonction `time_length` étant récente, elle n'est pas encore disponible dans la version stable du package. Pour installer la version de développement de **lubridate**, on aura recours au package **devtools** :

```
R> library(devtools)
   install_github("hadley/lubridate")
```

Nous noterons `naiss` la date de naissance et `evt` la date à laquelle nous calculerons l'âge.

# Calcul d'un âge exact

Une approche simple consiste à calculer une différence en jours puis à diviser par 365. Or, le souci c'est que toutes les années n'ont pas le même nombre de jours. Regardons par exemple ce qui se passe si l'on calcule l'âge au 31 décembre 1999 d'une personne née le 1er janvier 1900.

```
R> library(lubridate)
   naiss <- ymd("1900-01-01")
   evt <- ymd("1999-12-31")
   time_length(interval(naiss, evt), "days")
```

```
[1] 36523
```

```
R> time_length(interval(naiss, evt), "days")/365
```

```
[1] 100.063
```

Or, au 31 décembre 1999, cette personne n'a pas encore fêté son centième anniversaire. Le calcul précédent ne prend pas en compte les années bissextiles. Une approche plus correcte serait de considérer que les années durent en moyenne 365,25 jours.

```
R> time_length(interval(naiss, evt), "days")/365.25
```

```
[1] 99.99452
```

Si cette approche semble fonctionner avec cet exemple, ce n'est plus le cas dans d'autres situations.

```
R> evt <- ymd("1903-01-01")
   time_length(interval(naiss, evt), "days")/365.25
```

```
[1] 2.997947
```

Or, à la date du premier janvier 1903, cette personne a bien fêté son troisième anniversaire.

Pour calculer proprement un âge en années (ou en mois), il est dès lors nécessaire de prendre en compte la date anniversaire et le fait que la durée de chaque année (ou mois) est variable. C'est justement ce que fait la fonction `time_length` appliquée à un objet de type `Interval`. On détermine le dernier et le prochain anniversaire et l'on rajoute, à l'âge atteint au dernier anniversaire, le ratio entre le nombre de jours entre l'événement et le dernier anniversaire par le nombre de jours entre le prochain et le dernier anniversaire.

```
R> naiss <- ymd("1900-01-01")
   evt <- ymd("1999-12-31")
   time_length(interval(naiss, evt), "years")
```

```
[1] 99.99726
```

```
R> evt <- ymd("1903-01-01")
   time_length(interval(naiss, evt), "years")
```

```
[1] 3
```

```
R> evt <- ymd("1918-11-11")
   time_length(interval(naiss, evt), "years")
```

```
[1] 18.86027
```

Attention, cela n'est valable que si l'on présente à la fonction `time_length` un objet de type `Interval` (pour lequel on connait dès lors la date de début et la date de fin). Si l'on passe une durée (objet de type `Duration`) à la fonction `time_length`, le calcul s'effectuera alors en prenant en compte la durée moyenne d'une année (plus précisément 365 jours).

```
R> naiss <- ymd("1900-01-01")
   evt <- ymd("1999-12-31")
   time_length(interval(naiss, evt), "years")
```

```
[1] 99.99726
```

```
R> time_length(evt - naiss, "years")
```

```
[1] 100.063
```

```
R> time_length(as.duration(interval(naiss, evt)), "years")
```

```
[1] 100.063
```

## Cas particulier des personnes nées un 29 février

Pour les personnes nées un 29 février, il existe un certain flou concernant leur date d'anniversaire pour les années non bissextiles. Doit-on considérer qu'il s'agit du 28 février ou du 1er mars ?

Au sens strict, on peut considérer que leur anniversaire a lieu entre le 28 février soir à minuit et le 1er mars à 0 heure du matin, autrement dit que le 28 février ils n'ont pas encore fêté leur anniversaire. C'est la position adoptée par la fonction `time_length`.

```
R> naiss <- ymd("1992-02-29")
   evt <- ymd("2014-02-28")
   time_length(interval(naiss, evt), "years")
```

```
[1] 21.99726
```

```
R> evt <- ymd("2014-03-01")
   time_length(interval(naiss, evt), "years")
```

```
[1] 22
```

Cette approche permets également d'être cohérent avec la manière dont les dates sont prises en compte informatiquement. On considère en effet que lorsque seule la date est précisée (sans mention de l'heure), l'heure correspondante est `0:00`. Autrement dit, `"2014-03-01"` est équivalent à `"2014-03-01 00:00:00"`. L'approche adoptée permet donc d'être cohérent lorsque l'anniversaire est calculé en tenant compte des heures.

```
R> naiss <- ymd("1992-02-29")
   evt <- ymd_hms("2014-02-28 23:00:00")
   time_length(interval(naiss, evt), "years")
```

```
[1] 21.99989
```

```
R> evt <- ymd_hms("2014-03-01 00:00:00")
   time_length(interval(naiss, evt), "years")
```

```
[1] 22
```

```
R> evt <- ymd_hms("2014-03-01 01:00:00")
   time_length(interval(naiss, evt), "years")
```

```
[1] 22.00011
```

```
R> naiss <- ymd_hms("1992-02-29 12:00:00")
   evt <- ymd_hms("2014-03-01 01:00:00")
   time_length(interval(naiss, evt), "years")
```

```
[1] 22.00011
```

# Âge révolu ou âge au dernier anniversaire

Une fois que l'on sait calculer un âge exact, le calcul d'un âge révolu est assez simple. Il suffit de ne garder que la partie entière de l'âge exact (approche conseillée).

```
R> naiss <- ymd("1980-01-09")
   evt <- ymd("2015-01-01")
   time_length(interval(naiss, evt), "years")
```

```
[1] 34.97808
```

```
R> trunc(time_length(interval(naiss, evt), "years"))
```

```
[1] 34
```

Une autre approche consiste à convertir l'intervalle en objet de type `Period` et à ne prendre en compte que les années.

```
R> as.period(interval(naiss, evt))
```

```
[1] "34y 11m 23d 0H 0M 0S"
```

```
R> as.period(interval(naiss, evt))@year
```

```
[1] 34
```

# Âge par différence de millésimes

L'âge par différence de millésimes, encore appelé âge atteint dans l'année, s'obtient tout simplement en soustrayant l'année de naissance à l'année de l'événement.

```
R> naiss <- ymd("1980-01-09")
   evt <- ymd("2015-01-01")
   year(evt) - year(naiss)
```

```
[1] 35
```

**NOTE**

L'ensemble des fonctions présentées peuvent être appliquées à des vecteurs et, par conséquent, aux colonnes d'un tableau de données (*data.frame*).

Le package **eeptools** fournit de son côté une fonction `age_calc`[1] qui permet le calcul des âges exacts et révolus.

```
R> library(eeptools)
```

```
Loading required package: ggplot2
Loading required package: MASS
```

```
R> naiss <- as.Date("1980-01-09")
   evt <- as.Date("2015-01-01")
   age_calc(naiss, evt, units = "years", precise = TRUE)
```

```
[1] 34.97814
```

```
R> time_length(interval(naiss, evt), "years")
```

```
[1] 34.97808
```

La méthode utilisée par `age_calc` donne des résultats légèrement différent de ceux de `time_length`. Il est donc conseillé d'utiliser de préférence le package **lubridate**.

En l'absence du package lubridate, il reste facile de calculer une durée en jours avec les fonctions de base de **R**.

```
R> naiss <- as.Date("1900-01-01")
   evt <- as.Date("1999-12-31")
   evt - naiss
```

```
Time difference of 36523 days
```

```
R> as.integer(evt - naiss)
```

```
[1] 36523
```

1. https://github.com/jknowles/eeptools/blob/master/R/age_calc.R↩